Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Multiscale Finite Element method application to Canopies in Earth System

Dissertation
with the aim of achieving a doctoral degree at the
Faculty of Mathematics, Informatics and Natural Sciences
Department of Earth Sciences at Universität Hamburg

Heena Kiritbhai Patel
from Surat, Gujarat, India

Faculty:	University of Hamburg, Department of Mathematics / Center for Earth System Research and Sustainability (CEN)
Supervisor:	Prof. Jörn Behrens, Dr. Konrad Simon
Date of oral defense:	16 July 2024

Abstract

Urban canopies consist of buildings and trees that are aligned along a street in the horizontal direction. These canopies in cities and forests modulate the local climate in an intricate and complex way. Canopies constitute very fine subgrid features that actually have a significant impact on other components of earth system models. However, their feedback on larger scales is represented in rather heuristic ways. The problem with simulating their impact is twofold: first, their local modeling is delicate and second, the numerical modeling of the scale interaction between fine and large scales is complicated. We will mostly focus on the second aspect.

Multiscale finite element methods (MsFEM) in their classical form have been applied to various porous media problems, but the situation in climate and hence flow-dominated regimes is different from porous media applications. In order to study the effect of various parameters like the concentration of pollutants or the dynamics of the background velocity and of the temperature in the atmospheric boundary layer, a semi-Lagrangian multiscale reconstruction method (SLMsR) based multiscale finite element framework developed in [38, 39, 40] for passive tracer transport modeled by an advection-diffusion equation with high-contrast oscillatory diffusion is applied.

These methods are composed of two parts: a local-in-time semi-Lagrangian offline phase that precomputes basis functions and an online phase that uses these basis functions to compute the solution on a coarse Eulerian simulation mesh. The overhead of pre-computing the basis functions in each coarse block can be further reduced by parallelization. The online phase is approximately as fast as a low-resolution standard finite element method. Using the modified basis that carries subgrid information, however, still reveals fine scale features and is therefore accurate. This approach is studied in order to reveal the feedback of processes in the canopy layer on different scales present in climate simulation models. In particular, it is studied in the atmospheric boundary layer.

We will show the results of massively parallel simulations of passive tracer transport in an urban region using the novel multiscale approach and compare them with classical approaches. Also, data from the wind tunnel experiment were used to test the high resolution finite element method. Further the lidar data for Hamburg mesh is taken and a single building of 30 m is upscaled in a 2 km grid cell as shown in Figure (1).

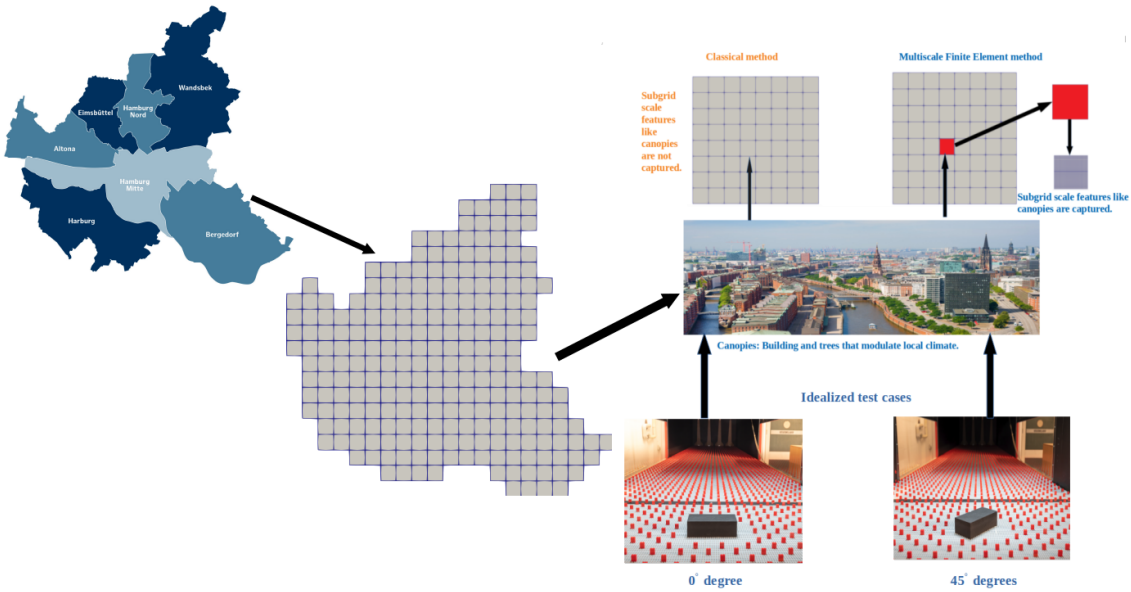


Figure 1: Graphical overview of our work, from the wind tunnel tests at the University of Hamburg's Environmental Wind Tunnel Laboratory to models that incorporate the multiscale finite element method (MsFEM). Figure courtesy of Heena Patel and the Environmental Wind Tunnel Laboratory at the University of Hamburg.

Kurzfassung

Städtische grenzschicht (canopy layer) bestehen aus Gebäuden und Bäumen, die entlang einer Straße in horizontaler Richtung angeordnet sind. Diese grenzschicht in Städten und Wäldern modulieren das lokale Klima auf komplizierte und komplexe Weise. Baumkronen stellen sehr feine Teilgitter dar, die tatsächlich einen erheblichen Einfluss auf andere Komponenten von Erdsystemmodellen haben. Ihre Rückkopplung auf größeren Skalen wird heute jedoch eher heuristisch dargestellt. Das Problem bei der Simulation ihrer Auswirkungen ist ein zweifaches: Erstens ist ihre lokale Modellierung sensibel und zweitens ist die numerische Modellierung der Wechselwirkung zwischen feinen und großen Skalen kompliziert. Wir werden uns hauptsächlich auf den zweiten Aspekt konzentrieren.

Multiskalen-Finite-Elemente-Methoden (MsFEM) wurden in ihrer klassischen Form auf verschiedene Probleme in porösen Medien angewandt, aber die Situation im Klima und damit in strömungsdominierten Regimen unterscheidet sich von den Anwendungen in porösen Medien. Um die Auswirkungen verschiedener Parameter wie der Schadstoff-konzentration oder der Dynamik der Hintergrundgeschwindigkeit und der Temperatur in der atmosphärischen Grenzschicht zu untersuchen, wird ein auf semi-Lagrangescher Rekonstruktionsmethode (SLMsR) basierendes Multiskalen-Finite-Elemente-Framework angewandt, das in [38, 39, 40] für den passiven Tracer-Transport entwickelt wurde, der durch eine Advektions-Diffusionsgleichung mit kontrastreicher oszillierender Diffusion modelliert wird.

Diese Methoden bestehen aus zwei Teilen: einer zeitlich begrenzten semi-Lagrangeschen Offline-Phase, in der Basisfunktionen vorberechnet werden, und einer Online-Phase, in der diese Basisfunktionen zur Berechnung der Lösung auf einem groben Eulerschen Simulationsnetz verwendet werden. Der Aufwand für die Vorberechnung der Basisfunktionen in jedem groben Block kann durch Parallelisierung weiter reduziert werden. Die Online-Phase ist ungefähr so schnell wie eine Standard Finite Elemente Methode mit niedriger Auflösung. Die Verwendung der modifizierten Basis, die unterhalb der gitterauflösung informationen enthält, zeigt jedoch immer noch Merkmale auf feiner Skala und ist daher genau. Dieser Ansatz wird untersucht, um die Rückkopplung von Prozessen in der Baumkronenschicht auf verschiedenen Skalen in Klimasimulationsmodellen aufzuzeigen. Insbesondere wird er in der atmosphärischen Grenzschicht untersucht.

Wir werden die Ergebnisse massiv paralleler Simulationen des passiven Tracer-Transports in einer städtischen Region unter Verwendung des neuartigen Multiskalen-Ansatzes

zeigen und sie mit klassischen Ansätzen vergleichen. Außerdem wurden Daten aus dem Windkanal-Experiment verwendet, um die hochauflösende Finite-Elemente-Methode zu testen. Weiterhin werden die Lidar-Daten für Hamburg herangezogen und ein einzelnes Gebäude von 30 m in eine 2 km-Gitterzelle hochskaliert wie in der Abbildung (1) gezeigt.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to supervisor Dr. Konrad Simon for introducing me to the world of multiscale finite element method, C++, and deal.II as when I started my PhD, I was a blank piece of paper to put my baby steps in this research field. I learned a lot from you that I could not give back. The knowledge and discussion you have was the baseline for my research, as it was the only your two papers on Google I have on to start with my topic. It was a big favor you did for us when you presented at the CLICCS retreat in 2020. I would like to give a big thank you to my supervisor, Big Boss, of my project CLICCS A3 and Faculty advisor of SIAM Student chapter Hamburg Prof. Jörn Behrens as you are the person you have always given me the space to explore my ideas and openness to talk to you about science and other aspects of life. We have shared many professional positions in different duties. It has always been a pleasure to discuss funding, presentation, and my professional future plans with you. You have so much patience to hear all my supernatural ideas and have generously given me the opportunity to attend many scientific conferences. I am super grateful for the big spaghetti ice cream you gave us for your 10th work anniversary. It has always acted as motivation during tough times.

I am blessed to be part of and funded by Cluster of Excellence "Climate, Climatic Change, and Society" (CLICCS) subproject A3: Canopies in the Earth System. This project is my adopted professional child. CLICCS is the most international and gender equal project in the world. I have met many wonderful colleagues worldwide from various disciplines. It enhances my scientific, leadership, and project management skills. Since I was the first person to be hired for the project, it has always been my responsibility to get everyone together. I have had the best moment of my life leading the whole presentation in a Theme A project meeting on 13 Feburary 2020. This was in my first presentation in Hamburg, thanks to the chairs of A3: Prof. Jörn Behrens, Prof. Bernd Leitl and Prof. Felix Ament. I would like to express my gratitude to Dr. Sylvio Freitas, who conducted a wind tunnel experiment in a pandemic when 1.5 distance rules exist. It was always a pleasure to discuss the wind tunnel setup with you. I would also like to thank Frank Harms, Bernd Leitl, and the entire Environmental Wind Tunnel Laboratory (EWTL) group for clearing all the basic doubts we had about the experiment. I would like to thank PhD student and co-worker Ge Cheng, Dr. David Grawe, and Prof. Dr. Heinke Schlünzen Mesoscale from Microscale Modelling (MeMi) for discussing the meteorological aspects of the project. It was always a pleasure when we all from the Numerical Methods in Geosciences group, ETWL group, and MeMi group colleagues

had project meetings as the beauty of seeing the research and discussion we had was always a wonderful experience. The CLICCS retreat was always fun to attend to discuss science and also enjoy dinner and dancing with all colleagues. Thanks to Dr. Martina Bachmann, Anke Allner, Rita Moller, and Charlotta Mirbach for organizing more than 200 people together. Prof. Felix Ament is also my panel chair. I am thankful for the meetings and discussions.

It is a pleasure to be a part of the School of Integrated Climate System Sciences (SICSS) at the University of Hamburg. It is a structured doctoral program where I have learned many soft skills, met many international colleagues. It always feels like you are part of a large community. Thanks to Dr. Berit Hachfeld, Dr. Ingo Hams, Dr. Sebastian Zubrzycki, and Dr. Alexandra Franzke for all the Christmas parties, retreats, summer schools, and workshops you have organized for us. It was a thrilling experience to be part of Student Chapter Hamburg. Thanks to Hannes von Allwörden for enrolling me in the chapter and inspiring me to serve as Secretary and as the first woman of color president of Student Chapter Hamburg. It was always a pleasure to host events such as Life after PhD for alumni students in the mathematics department. In addition, Stages of Academia where faculty members expressed their journey. Again, a beautiful experience to take my South Asian origin, representing Student Chapter Hamburg Germany Europe as chapter president at SIAM Annual meeting 2022 in USA and historic discussion to hire GAMM Faculty Advisor for Student Chapter Hamburg, toughest decision as PhD student but strongest decision as the President of Student Chapter Hamburg.

It was nice to be part of the Numerical Methods in Geosciences group and my colleagues Yumeng Chen, Anusha Sunkisala, Michel Bänsch, Ezra Rozier, Mouhanned Gabsi and Maša Avakumović in office 410, Grindelberg 5, 20144 Hamburg. Teffy Sam for help to save files from server to laptop and Dr. Claudine von Hallern for getting to know each other. Yumeng, you said something exquisite when I was going through tough times “Many people develop numerical models, but people like you give life to them by implementing applications". It was always a heart-touching experience with colleagues in the Grindelberg 5-7 corridor with the Atmospheric Dynamics and Predictability group, formerly the Theoretical Meteorology working group. On paper, I have no terms with you, but you all have always reserved a place for me at every birthday and Christmas party. You also had a postdoc position for me. I am always grateful to have lunches with you, even when the 1.5 distance rule exists. I might have gone into depression if we did not have these lunches. Thanks to Denny Gohlke for being a German friend who cooks Indian for me, PD Dr Richard Blender for Swiss chocolates, Iana Strigunova for hugs, Dr Sergiy Vasylkevych for being my bestie, Qiung Ma for the compliment you are as strong as man, Yuan-Bing Zhao for taking my photo with giant panda furries, Gözde Özden

for my autograph in your copy of Hamburg Climate future outlook. Katharina Holube and Chen Wang for the band we play together, my favourite Sándor Mahó for being shy always reminds me of Finland, Valentino Neduhal for bearing my words and desire to work for me in the future, Dr. Frank Lunkeit for the morning talks, Frank Seilmann and Alexia Krawat for helping with German bureaucracy and someone who somehow inspires Prof. Nedjeljka Žagar: Thank you for allowing me to break many protocols for cakes and spending time with your group.

I am very thankful to my Mom Gitaben Kiritbhai Patel for giving me permission to fly to Europe. It was not easy to change apartments 7 times in Europe single-handedly. But I have grown stronger and braver through these years. Thank you for understanding that it was more than three years not going to India that was difficult for you without me. You have always trusted me that I will survive the toughest situations. I had given my professional life more priority than my personal life, and it would not have been possible without my sister Khyati Kiritbhai Patel to be with Mom. Thanks for taking care of Mom and Dad and all the sacrifices you have made. To my late father Kiritbhai Kantibhai Patel for the strong personality I inherited from you. You would be proud that my male colleagues say they cannot move like I have moved in Europe. Thank you to all my friends and colleagues worldwide for their wise words. Thanks to Emmi Koskimies, Iiro-Juhani Pylkkänen and their wonderful kids Viljo, Väino, Vilma and Veikko from my Finnish host family, Väino I missed you a lot my boy Minä rakastan sinua, it took almost 4 years and 10 months for me to fly to Finland. And Antje Mizera, Mike Mizera, Andrea Mizera, Rino Mizera and meine Oma Else Priebe ich liebe dich from my German host family for my European family experiences. It is hard to maintain my soul in India, my heart in Finland, and my brain in Germany. But I am born to break normal. To my Hamburg city, I love you for all reasons for strength, for new friendships, for Hamburg DOM, for Cruise days and Hafengeburtstag. Thanks to all readers of my thesis.

List of Contents

List of Figures

List of Tables

List of Algorithms

Listings

Notations

Ω	open set in R^n.
Γ	$\partial\Omega$.
Γ_D	part of the boundary on which Dirichlet conditions are prescribed.
Γ_N	part of the boundary on which Neumann conditions are prescribed.
Δ	Laplace operator.
$L^2(\Omega)$	space of square-integrable functions over Ω.
$H^m(\Omega)$	Sobolev space of L^2 functions with square-integrable derivatives up to order m.
$H_0^m(\Omega)$	subspace of $H^m(\Omega)$ of functions with generalized zero bounary conditions.
$C^k(\Omega)$	set of functions with continuous derivatives up to order k.
$C_0^k(\Omega)$	subspace of $C^k(\Omega)$ of functions with compact support.
$\|\|.\|\|_m$	Sobolev norm of order m.
$\|.\|$	Sobolev semi-norm of order m.
$\|\|.\|\|_\infty$	supremum norm.
H^1	dual space of H.
n	exterior normal.
∂, ∂_n	derivative in the direction of the exterior normal.
∇f	$(\partial f/\partial x_1, \partial f/\partial x_2, ..., \partial f/\partial x_n)$.
$divf$	$\sum_{i=1}^n (\partial f/\partial x_i)$.
V_h	finite element space.
φ_h	basis function in V_h.
$\mathcal{T}_h$	partition of Ω.
K	(triangular or quadrilateral) element in $\mathcal{T}_h$.
$\hat{K}$	reference element.
$\sum$	set of linear functionals in the definition of affine families.
a_ε	diffusion tensor.
c_δ	velocity.

1 Introduction

"Give me a place to stand, and I will move the earth."

- Archimedes

1.1 Background and Motivation

According to UN estimates, more than 6.6 billion people (68%) will live in cities by 2050. The skylines of many large cities are already dominated by tall (> 50 m high) and super-high (> 300 m high) buildings. As urban populations grow, tall buildings will become increasingly common outside of city centers, especially if urban sprawl is geographically limited [5]. Industrialization resulted in the emission of various gases into the atmosphere that harmed the environment. Consequently, the Earth's temperature increased abnormally and weather patterns fluctuated. With the Paris climate agreement from December 2015, climate policy and climate research have been given a powerful impetus to curb climate change. In order to address the resulting new challenges, the **Cluster of Excellence for Climate, Climatic Change, and Society (CLICCS)** developed a long-term program that covers a broad range of topics ranging from climate dynamics to climate-related social dynamics to transdisciplinary explorations of human–environment interactions [4]. The human interaction with the Earth creates a question about future climate possibilities. The **A3: Canopies in the Earth System** sub-project examines local scale climate futures that are possible and plausible from a global perspective. It explains the variability of climate in canopies, a crucial part of the Earth's System. According to [3], the purpose of this study is to answer the fundamental question in climate modeling as to **what extent and how much canopies modulate climate in the lowest 100 meters of the atmosphere**.

"Urban" refers to a complex buildings including houses, commercial buildings, roads, industrial facilities, and city parks built in densely populated areas. Urban areas include towns, cities, and suburbs. "Rural" refers to areas surrounding urban environments. Rural areas may consist of natural areas, where human intervention is minimal or not evident, or anthropogenically modified areas, such as agricultural and forestry areas. Water bodies can also be found in rural areas. **Urban heat islands (UHI)** in the lower part of the urban canopy layer (UCL), occur when cities experience much warmer temperatures than nearby rural areas. There is a difference in temperature between urban and less-developed rural areas due to how well the surfaces absorb and hold heat in each environment. Humans directly experience the **canopy layer UHI (CL-UHI)** in the lower part of the

atmosphere's canopy layer. CL-UHI refers to the microscale to mesoscale warming effect of cities on the atmosphere. CL-UHI is generally calculated from synchronous differences in near-surface air temperatures between urban and non-urban areas. For both areas, the typical measurement height is approximately 1.5 m above ground level (AGL). Those who live in cities may be at risk from the CL-UHI effect, as exposure to high temperatures, especially in hot weather, can increase morbidity and mortality. As nighttime temperatures rise, CL-UHI directly affects human health [22].

In order to simulate CL-UHI intensities, three mathematical models are available. A statistical model, an obstacle-resolving model, and a numerical weather prediction model (NWP). With a spatial resolution of 1 m, the obstacle-resolving model (ORM) attempts to resolve processes in the urban canopy layer (UCL). Model domains typically range from 1 to 100 km^2. Due to Courant-Friedrichs-Lewy's (CFL) criterion for ensuring numerical method stability, the ORM's time step is small due to its high resolution. The amount of grid points also determines the computing power needed for integration. Due to this, ORM runs are typically limited to a few hours or days. Most models cover only one or two neighborhoods (1–2 km) and not the entire region of interest with urban and rural components. If large cities that cover several grid cells are properly resolved, they can have significant effects on the modeled atmosphere. A 1-km grid does not represent urban effects at the neighborhood level since the model captures atmospheric processes between 5 and 8 grid lengths. Mesoscale variations in CL-UHI intensity may be simulated by models of regional atmospheric circulation and numerical weather prediction (NWP) that include urban energy and momentum fluxes. With a grid resolution of 1 km, they cover domains 10^2–10^3 km and more horizontally. These models can resolve mesoscale atmospheric phenomena at these spatial scales. To avoid confusion, regional models (RMs) are called regional scales (Table 1.1), which are used to classify urban forms [22].

Scale	Urban form	Horizontal length (HL)	Vertical extent	Related parameters	Atmospheric phenomena HL scale
Micro	Facet (roof, wall, road)	1-10 m	UCL	Materials	Microscale γ
	Building	10+ m	UCL	H	Microscale γ
	Street canyon	130-200 m	UCL	H, W	Microscale β
Local	Block (bounded by canyons, interior courtyards)	300-500 m	RSL	λ_p, H_{max}, σ_H	Microscale α
	Neighbourhood	1-2 km	RSL, ISL	λ_p, H_{max}, σ_H	Microscale α
Meso	Urban area (city centre to low-density residential areas that are contiguous)	10-100 km	UBL	λ_p, H_{max}, σ_H	Mesoscale γ, Mesoscale β
Regional	Region (urban and non-urban surroundings)	> 100 km	PBL	λ_p, H_{max}, σ_H	Mesoscale β, Mesoscale α

Note: The vertical extent depends on the urban form, where H: building/urban canyon height; W: street/urban canyon width; λ_p: plan area fraction of buildings; H_{max}: maximum H; σ_H : standard deviation of H; UCL: urban canopy layer, RSL: roughness sublayer, ISL: inertial sublayer, UBL: urban boundary layer, PBL: planetary boundary layer.

Source: Modified from Cleugh and Grimmond (2012) and Oke et al. (2017).

Atmospheric phenomena characteristic scales based on Orlanski (1975).

Table 1.1: The characteristics of urban form are scaled based on the characteristics of the surface [22].

If a model covers the entire globe, it is referred to as a global climate model (GCM). Regional climate model (RCM) simulates the local climate using GCM outputs as inputs to high-resolution climate models.

There are no operational weather forecasting models that consider urban characteristics performed by the National Meteorological and Hydrological Services (NMHSs); however, such models are already used by research institutions in 13 countries across

Europe, Brazil, Canada, and northern China, with typical resolutions of 1–4 km. For some models, they also calculate time-dependent 3D fields of temperature, airflow, and humidity, as well as clouds, rain, and air pollutant concentrations. Additionally, ORMs' computational cost is influenced by their spatial extent, resolution, and time step (CFL). All models simulate heat, moisture, and momentum exchanges using land surface schemes. In order for CL-UHI to be effective, both urban and rural settings must follow a consistent approach. Urban canopy parameterization (UCP) is based on a coarser resolution than ORMs, so individual buildings and other urban structures (such as trees) are not resolved. The North American Mesoscale Forecast System (NAM), which has a horizontal grid spacing of 12 km, and the High Resolution Rapid Refresh (HRRR) model have too coarse resolutions, which prevent direct CL-UHI calculations [22].

The urban extent of most GCMs is not sufficient to cover enough grid cells to be meaningful, since the grid resolution is generally coarse ($\sim$ 50 km to $\sim$ 200 km). At the global level, many models do not provide adequate urban parameters or only provide a simple representation of urban areas. To improve the model's ability to determine CL-UHI, it is necessary to refine the model and input data. Model runs with high resolution (10 km) are now available, which will help to overcome some of the current limitations for megacities. Grid resolution of RCMs ($\sim$ 1 km to $\sim$ 50 km) is generally better than that of GCMs. CL-UHI can be predicted using km-scale RCMs in larger cities [22].

High-resolution climate projections are computationally expensive. Methods for downscaling the CL-UHI include using statistical models along with GCM or RCM projections [43]. For estimating the CL-UHI, GCM and RCM projections can also be used. As ensemble techniques require less computing power, they can be used. Bias correction must be applied to GCM/RCM input data in order to prevent statistical models from being influenced by predictor bias. In contrast, statistical models are more reliable when used with training data because they don't account for changes in city characteristics. Given that climatic timescales are likely to influence urban characteristics, both must be considered. Due to the uncertainty associated with urban development projections, scenarios are used instead. In modeling CL-UHI, the main limitations are the mesoscale (O(1km)) and microscale ($\sim$O(1m)) model resolutions. Despite advances in expressing complexity within the urban canopy layer (UCL), many studies aim to improve the flux from the UCL to the upper atmosphere in order to assess the NWP and the climate of the city. In addition to correctly accounting for horizontal heterogeneity in urban areas, the effects of individual tall buildings or small groups of tall buildings are not yet well parameterized. There is still a challenge in simulating the interaction of urban vegetation with other surfaces [22].

1.2 Numerical Modelling of subgrid scales

Physical and dynamical processes occur in the atmosphere, which contribute to atmospheric circulation, as described by numerical models. Our atmosphere and oceans have a wide range of spatial and temporal scales. As a result, climate models must consider the interactions among many scales, since each affects the others. Climate sciences simulations typically use coarse grids because of computational constraints. However, unresolved subscale information affects prognostic variables significantly and cannot be ignored for reliable long-term predictions. Even if microscales are highly heterogeneous, algorithms for modeling complex physical processes at macroscales attempt to elicit their effective behavior at large scales. It would be safer to resolve microscopic processes, but due to their computational cost, such a strategy is usually prohibitive. Microscopic processes, however, have a significant influence on macroscopic behavior and should not be overlooked [38]. Microscale effects are challenging to incorporate into macro simulations in a mathematically consistent manner. This demands mathematically rigorously justified multiscale methods [39].

The challenge lies in coming up with a solution that is computationally inexpensive and capable of representing the impact of small-scale features, such as canopy, on a large scale.

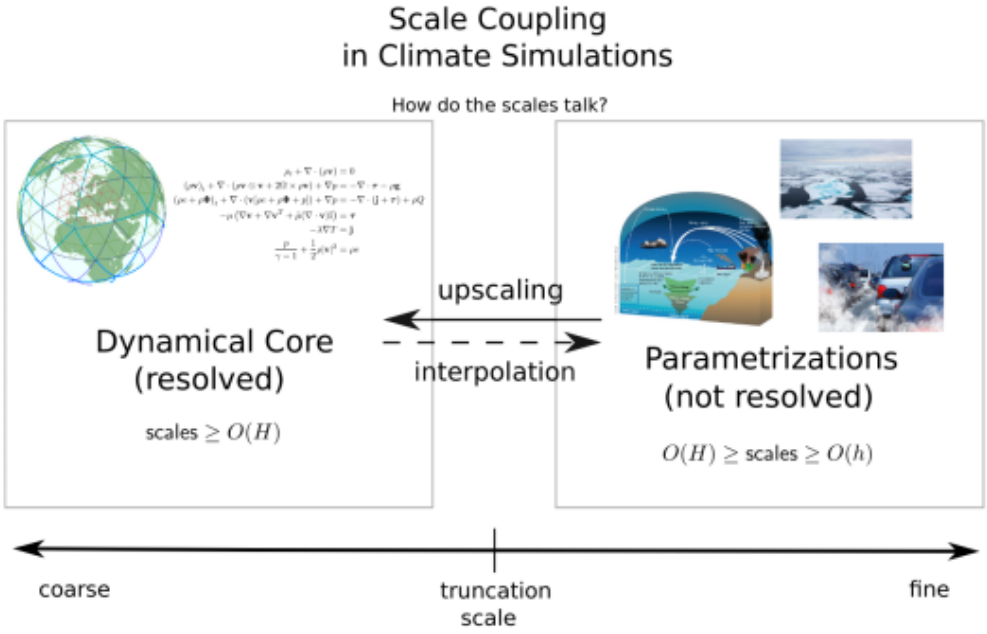


Figure 1.1: Scale Interaction between resolved and unresolved regions of climate models [36].

Figure (1.1) presents two regions of climate models: the dynamical core, which deals with grid-resolved problems, and parameterizations, which represent sub-grid processes that cannot be resolved by grids. If you want to project a large-scale effect onto a small scale, it is normally done by interpolation. If you want to know whether a fine -scale feature is relevant to a coarse-scale feature, refer to this as an upscale problem.

1.3 Multiscale Modelling

Complex systems in nature operate on a variety of length scales. In order to accurately describe the true behavior of a system, multiple length scales are required as scientific models become more complex. Multiscale modeling involves multiple scales. By using multiscale models, global degrees of freedom can be reduced while flow and transport processes are preserved as shown in Figure (1.2).

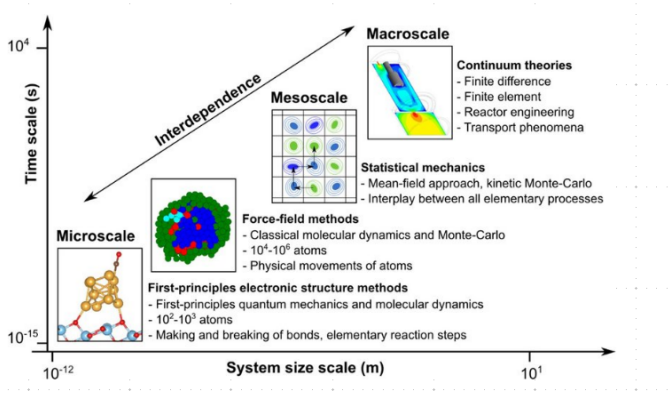


Figure 1.2: Multiscale Modelling at various scales [26].

In order to transfer information from the subgrid scale to the coarse grid, a mathematically consistent method must be developed. Multiscale numerical modeling at multiple scales offers a promising mathematical framework for achieving this goal. Although such methods are well established in other communities, such as porous media, they are rarely used in climate simulations. As basis functions resolve fine scale behavior, multiscale finite element methods have been extensively investigated in the porous media community [44] and [17]. Multiscale methods can be found in [39]. A variational multiscale method divides the solution space into coarse and fine scale parts, and tests coarse scale functions in one part and fine scale functions in another part of the variational form. Due to the use of non-polynomial basis functions, the multiscale finite element method (MsFEM) and variational multiscale method are closely related. The method is highly scalable since it allows for massive parallelization. For a basis function to be constructed, it must satisfy an equation leading order.

Multiscale finite element method is composed of two components: multiscale basis

functions and global numerical formulations that combine multiscale-based functions. A multiscale basis function captures the fine scale features of a solution. It is important that these localized basis functions include information about scales that are smaller than the local numerical scale determined by these localized basis functions. In particular, we need to incorporate the features of the solution that can be localized. We need to use functions to capture information about features to be separately included in the coarse space. An accurate approximation of the solution [15] can be obtained by coupling these basis functions. A key objective of this research is to examine and apply MsFEM to urban climate simulations and take into account the advection-diffusion equation to observe various flow parameters - such as velocity and pollutants - extending the work in [38, 39, 40]. Here we use deal.II (Differential Equations Analysis Library) [7] a C++ program library that targets the computational solution of partial differential equations.

Massively parallel high-performance computing features are available for various applications and have several advantages, including user control of mathematical implementation, availability of different preconditioner solvers, and detailed documentation. Using deal.II, we implemented a parallelization algorithm for semi-Lagrangian meshes on different processors to support the application of MsFEM to transport-dominated problems.

To implement the canopy effect in numerical weather prediction (NWP) or climate models, parameterization schemes are developed for various models. In [10], the numerical aspects of physical parameterization are discussed mainly within the context of the ECMWF Integrated Forecasting System. Clouds and land cover were incorporated into the equation by adding source terms. Different processes have different numerical problems. Numerical design requires a thorough understanding of physics and the interactions between different processes. It was found that a more accurate canopy representation comes with a higher computational cost when different urban canopy models are coupled with weather forecast models in [20]. Hence, it is necessary to understand the effect of the canopy at various scales. The size, shape, urban infrastructure, and climatic conditions of cities all play a significant role in the formation of the Urban Heat Island (UHI). UHI is studied using various modelling techniques, but none of them accurately represent the physical phenomena and complex urban infrastructure. Different canopy models were discussed and their limitations were outlined in [35]. It is necessary to investigate mathematical methods that can integrate mesoscale and microscale models so that the Urban Heat Island (UHI) effect can be captured. A velocity and diffusion coefficient effect below and above canopy height was included in the advection-diffusion equation [28, 23].

1.4 Knowledge Gap

In this study, we use a novel approach to show how buildings are represented as obstructions with differences in diffusivity and the effect of tracer transport near the building. The entire setup mimics the real situation of a single building under wind and concentration conditions. The idea here is to implement a simple stationary obstacle in the advection-diffusion equation, solved with high resolution finite element method, low resolution finite element method and low resolution multiscale finite element method. The result of the high resolution finite element method is then used to verify the experimental results obtained in a wind tunnel experiment conducted in the Environmental Wind Tunnel Laboratory (EWTL) at the University of Hamburg. A simple step toward multiscale finite element method application can be observed here, from the microscale scale to larger scales to overcome the limitations of modelling Urban Heat Island (UHI) as described in [35]. The code development for fully parallelized code in C++ is developed in the deal.II library. The code was developed for 2D and 3D cases.

Research Questions

These prompt questions will be addressed in the current study.

- Does MsFEM effectively account for the non-trivial subgrid scale structure in canopies? i.e., when representation of canopies in the global km scale normally the whole city is considered in a few grid cells and average effect is considered, which does not take into account the local climate effects at the m scale.

- How much computational cost does MsFEM require for effective representation of tracer transport?

- What is the high- resolution FEM model's performance compared with real-world wind tunnel data?

Figure (1.3) shows the significant phenomenon around canopy at 100 metres atmospheric height.

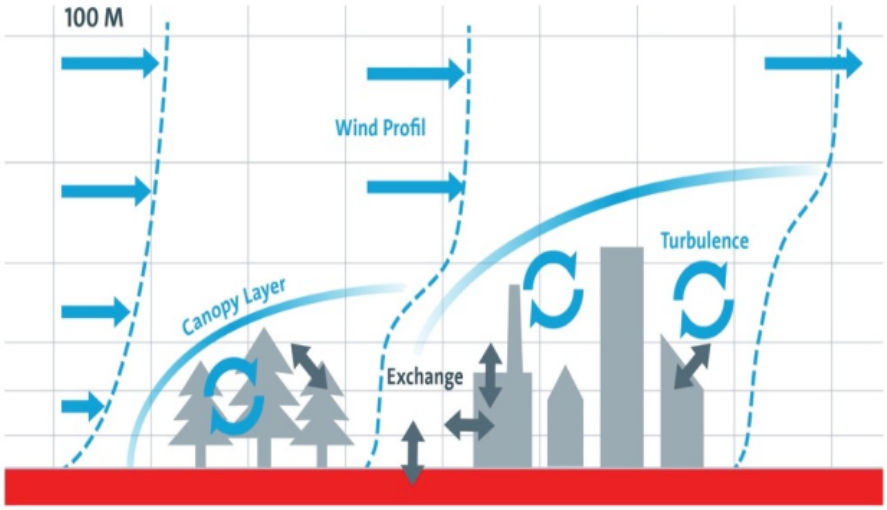


Figure 1.3: Canopy research in CLICCS A3 proposal.

1.5 Aim of the thesis

This study investigates and implements multiscale FEM methods in urban climate simulations. We have used deal.II in the present study [7], a C++ program library for solving partial differential equations. Various applications can be implemented using this advanced mathematics library with high-performance computing capabilities. There are a number of preconditioner solvers available, as well as detailed documentation, which makes it a desirable mathematical implementation tool. The parallelization algorithm is employed for semi-Lagrangian meshes on different processors. The implementation is fully parallelized and is tested for millions of degrees of freedom but could also be run across billions of processors. The new canopy parameterization considers the subgrid scale feature using the diffusion coefficient as the building geometric. In the next step, it is upscaled to a larger scale using a multiscale basis function.

As the diffusion coefficient, the canopies are implemented, extending the work done by [38, 39, 40]. Diffusion coefficients have been previously described as the components of the flow that advects. This study, shows how buildings are represented as obstructions to show the difference in diffusivity and the effect of tracer transport on a building. The entire setup mimics a real situation with a single building under wind and concentration conditions. Our goal is to incorporate a simple stationary obstacle into the advection-diffusion equation and verify the experimental results obtained in the University of Hamburg's Environmental Wind Tunnel Laboratory (EWTL). A simple step toward multiscale finite element method application can be seen here, from microscale to macroscale.

1.6 Talks at conferences

Based on this work, the following publications were derived: 6 a set of talks given in international conferences.

- Patel, H., Simon, K., and Behrens, J.: Massively Parallel Multiscale Simulations of the Feedback of Urban Canopies, EGU General Assembly 2021, online, 19–30 Apr 2021, EGU21-2507, https://doi.org/10.5194/egusphere-egu21-2507, 2021.

- Heena Patel Konrad Simon and Jörn Behrens : Multiscale Method for Application to Urban Canopies,PDEs on the Sphere 2021, online 17-21 May 2021, https://www.dwd.de/EN/specialusers/research_education/seminar/2021/pdes_on_the_sphere/pdes_2020_en_node.html.

- Heena Patel Konrad Simon and Jörn Behrens : Multiscale Finite Element Method for Advection Dominated Geoscientific Applications,SIAM Conference on Mathematical and Computational Issues in the Geosciences (GS21), online, 21-24 June 2021, https://meetings.siam.org/sess/dsp_talk.cfm?p=111778.

- Patel, H., Simon, K., and Behrens, J.: Towards Canopy parameterization for Multiscale Finite Element Method, EGU General Assembly 2022, Vienna, Austria, 23–27 May 2022, EGU22-3807, https://doi.org/10.5194/egusphere-egu22-3807,2022.

- Heena Patel: MultiScale Finite Element for Transport- Dominant Equations Applied to Canopies, Hybrid:2022 SIAM Annual Meeting (AN22), Philadelphia, Pennsylvania, USA. 11–15 July 2022, https://www.siam.org/Portals/0/Conferences/AN/AN22/AN22_ABSTRACTS_V2.pdf.

- Heena Patel: Multiscale Finite Element Method for Urban Canopies in Climate Models, Offline :2023 SIAM Conference on Mathematical & Computational Issues in the Geosciences (GS23), 19-23 June 2023, https://meetings.siam.org/sess/dsp_talk.cfm?p=129644.

1.7 Publication

- Reviewer of CLICCS Hamburg Future Outlook Report 2021 https://www.cliccs.uni-hamburg.de/results/hamburg-climate-futures-outlook/documents/cliccs-hamburg-climate-futures-outlook-2021.pdf
- Blog Research Article published in SIAM news 9 September 2022 for talk in SIAM Annual Meeting USA
  https://sinews.siam.org/Details-Page/multiscale-finite-element-method-for-urban-canopies-in-climate-models
- Reviewer of CLICCS Hamburg Future Outlook Report 2023 https://www.cliccs.uni-hamburg.de/results/hamburg-climate-futures-outlook/download.html

2 Finite Element Method

"Just because we can't find a solution it doesn't mean that there isn't one."

- Andrew Wiles

2.1 Poisson's Equation

Let us consider Poisson's equation (D). Readers unfamiliar with some basic mathematical theories should refer to Appendix Section [I, II and III] .

$$(D) \qquad \begin{cases} -\nabla(a\nabla u) = f & \text{in } \Omega \subset \mathbb{R}^d \\ u = u_0 & \text{on } \Gamma_D \subset \partial\Omega \\ \nabla_n u = g & \text{on } \Gamma_N \subset \partial\Omega \end{cases} \tag{2.1}$$

The solution u to the boundary value differential equation (D) also is the solution to the minimization problem (M) and the variational problem (V). In order to formulate problems (M) and (V) we introduce the notation

$$(v,w) = \int_{\Omega} v(x)w(x)dx,$$

for real-valued piecewise continuous bounded functions. We also introduce the linear space

$$V = \{v : v \quad \text{is a continuous function on } \Omega, \nabla v \quad \text{are piecewise continuous on } \Omega, \\ \text{and} \quad v(x) = 0 \quad on \quad \Gamma_D\}$$

and the linear functional $F : V \to R$ given by

$$F(v) = \frac{1}{2}(\nabla v, \nabla v) - (f,v).$$

The problems (M) and (V) are the following:

(M) Find $u \in V$ such that $F(u) \leq F(v) \qquad \forall v \in V$,

(V) Find $u \in V$ such that $(\nabla u, \nabla v) = (f,v) \qquad \forall v \in V$.

If u is the solution to (D), then u is the solution to the equivalent problems (M) and (V)

which we write symbolically as

$$(D) \Rightarrow (V) \Leftrightarrow (M).$$

For proof of above expression refer [21]. When u is the solution of (V) and in addition satisfies a regularity assumption (u'' is continuous), then u is the solution of (D). By now, we can show that if u is the solution to (V) , then u satisfies the desired regularity assumption and thus we have $(V) \Rightarrow (D)$, demonstrating that the three problems $(D), (V)$ and (M) are equivalent. Hence we will now construct the (V) form of (D).

We also introduce the linear space

Definition 1. *Lebesgue space*
The Lebesgue space $L^2(\Omega)$ is defined by

$$L^2(\Omega) = \left\{ u : \Omega \to \mathbb{R} | \quad \int_\Omega |u(x)|^2 \, dx < \infty \right\}$$

It is a Hilbert space with scalar product

$$< u, v >_{L^2} = \int_\Omega uv \, dx$$

Definition 2. *Sobolev space*
The Sobolev space $H^1(\Omega)$ is defined as

$$H^1(\Omega) = \left\{ u \in L^2(\Omega) | u \text{ has a weak gradient with } \quad \int_\Omega |\nabla u(x)|^2 \, dx < \infty \right\}$$

It is also a Hilbert space with the scalar product

$$< u, v >_{H^1} = \int_\Omega uv \, dx + \int_\Omega \nabla u \cdot \nabla v \, dx$$

Definition 3. *The Sobolev space with zero boundary conditions $H^1_\Gamma(\Omega)$, where $\Gamma \subset \partial\Omega$, is defined as*

$$H^1_\Gamma(\Omega) = \left\{ u \in H^1(\Omega) | u|_\Gamma = 0 \right\}$$

2.2 Steps for Solution of Poisson's Equation using the finite element method

Let's consider a 2-dimensional unit square $[0,1] \times [0,1]$, the Neumann boundary condition (Γ_N) is at right and top edge while the Dirichlet boundary condition (Γ_D) is at left and bottom edge. We will use the deal.II [7] library with a template (template is a feature provided by the C++ language in which the code basic structure is written once and can be used to extend different dimensions of the problem with the change in template parameter.) code that can be extended to 3 dimensions in order to solve the problem with the finite element method as shown in Figure (2.1).

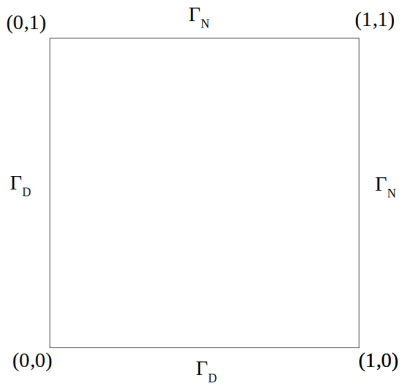


Figure 2.1: Unit square domain.

As shown in Figure (2.2), this is the flow chart to be followed in present Chapter 2 for mathematical theory and later in Chapter 5 for implementation of code.

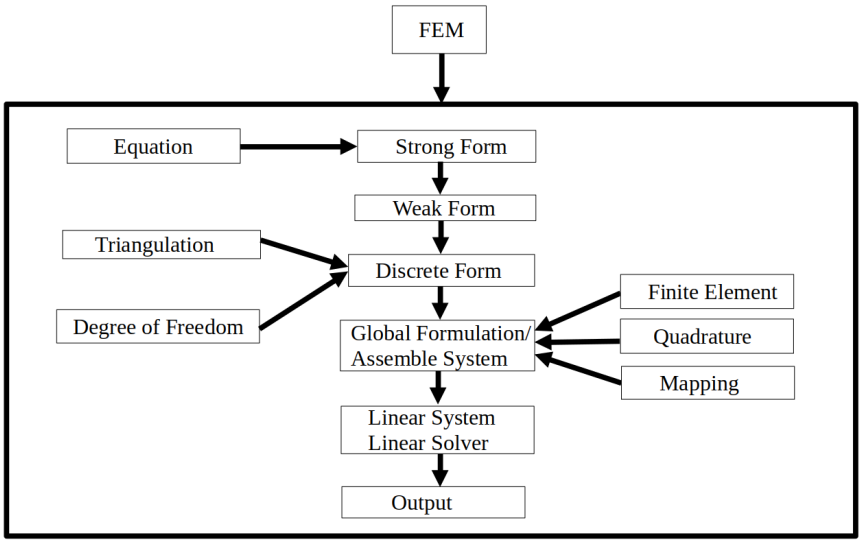


Figure 2.2: Steps of Finite Element Method.

The following section will walk you through all steps of the finite element method solution for stationary diffusion equation.

2.2.1 Step 1: Strong formulation of Poisson's equation

A partial differential equation with elliptic form exists in the domain. This is second order stationary diffusion equation (2.2). The boundary conditions or boundary values are imposed on the solution at the boundary.

$$\begin{cases} -\nabla(a\nabla u) = f & \text{in } \Omega = [0,1]^2 \\ u = u_0 & \text{on } \Gamma_D \subset \partial\Omega \\ \nabla_n u = g & \text{on } \Gamma_N \subset \partial\Omega \end{cases} \tag{2.2}$$

where

- $u = u(x)$ is unknown defined on domain Ω.
- $a : \Omega \to R^{(d\times d)}$ is some positive definite and symmetric coefficient matrix. a is the average diffusion coefficient. Such a matrix can for example describe the diffusivity of a certain medium (i.e., the domain) $\Omega \subset \mathbb{R}^d$ for a tracer gas.
- The source function $f = f(x)$ is given. In other words, it is the concentration of pollutants emitted from a building or chimney.
- $\partial\Omega$ of Ω is formed by the union of two subboundaries, $\partial\Omega = \Gamma_D \cup \Gamma_N$, $\Gamma_D \cap \Gamma_N = \emptyset$, where Γ_D is the Dirichlet boundary and Γ_N is the Neumann boundary.
- Dirichlet boundary conditions specify a prescribed value for u on Γ_D for the unknown u. Neumann's boundary condition $\nabla_n u = g$ specifies a value for the normal derivative of u on Γ_N.
- ∇_n stands for the exterior normal derivative, i.e., $\nabla_n u = \nabla u \cdot n$, where n stands for the unit normal vector pointing always outward, and ∇u stands for the gradient of u.

2.2.2 Step 2: Weak formulation of Poisson's equation

Using a weak or variational formulation, we can replace the classical representation equation (2.2).

Given the strong form

$$-\nabla \cdot (a\nabla u) = f \tag{2.3}$$

We test the equation with any **smooth test function** (infinitely continuously differentiable) that does not change the value of the solution on Γ_D, i.e., the test function must be zero on the boundary Γ_D. The unknown function u is called the solution whereas v is the so-called **test function**.
In order to obtain the weak form of the equation, we multiply the strong form by a test function v, and then integrate the result.

$$-\int_\Omega (\nabla \cdot (a\nabla u)) \cdot v\, dx = \int_\Omega f \cdot v\, dx \quad \forall v \in V \tag{2.4}$$

Now by integration by parts and with Green theorem we get,

$$\int_\Omega a\nabla u \cdot \nabla v\, dx - \int_{\partial\Omega} a\nabla_n u \cdot v\, dS = \int_\Omega f \cdot v\, dx \qquad \forall v \in V \tag{2.5}$$

The set of test functions vanishes on Γ_D.

$$\int_\Omega a\nabla u \cdot \nabla v\, dx - \int_{\Gamma_N} a\underbrace{\nabla_n u}_{=g} \cdot v\, dS = \int_\Omega f \cdot v\, dx \quad \forall v|_{\Gamma_D} = 0 \tag{2.6}$$

The test space $H^1_{0,\Gamma_D}(\Omega)$ is a Sobolev space with zero boundary condition that consists of a set of test functions vanishing on Γ_D to test with functions that do not alter the Dirichlet boundary condition:

$$H^1_{0,\Gamma_D}(\Omega) = \{v \in H^1(\Omega) : v = 0 \quad on \quad \Gamma_D\}$$

and the trial space $H^1_{u_0,\Gamma_D}$, containing the unknown function u, is defined similar to $H^1_{0,\Gamma_D}(\Omega)$ but with a shifted Dirichlet condition:

$$H^1_{u_0,\Gamma_D}(\Omega) = \{v \in H^1(\Omega) : v = u_0 \quad on \quad \Gamma_D\}$$

The final weak form or variational form of equation (2.2) is then to find u in an appropriate space such that

$$\int_\Omega a\nabla u \cdot \nabla v\, dx = \int_{\Gamma_N} a\, g \cdot v\, dS + \int_\Omega f \cdot v\, dx \quad \forall v \in H^1_{0,\Gamma_D}(\Omega) \tag{2.7}$$

The equation (2.7) only requires first order weak derivative of the solution u, rather than

the equation (2.2) requiring second order derivatives. One can easily see that any classical solution is a solution to the weak form. The reverse statement is not necessarily true.

Without specifying the spaces where u and v are, the weak formulation can be written as follows:

$$(V) \qquad \begin{cases} find \quad u \in H^1(\Omega), \quad such \quad that \\ u = u_0 \quad \text{on } \Gamma_D \subset \partial\Omega \\ \int_\Omega a\nabla u \cdot \nabla v \, dx = \int_{\Gamma_N} a\, g \cdot v \, dS + \int_\Omega f \cdot v \, dx \quad \forall v \in H^1_{0,\Gamma_D}(\Omega) \end{cases} \tag{2.8}$$

In this formulation, the two boundary conditions appear at very different places: The boundary condition $\nabla_n u = g$ on Γ_N is called **natural** since it is part of the variational form and is enforced weakly while the condition $u = g$ on Γ_D is called **essential**.

Theorem 1. *For each $f \in L^2(\Omega)$ the weak form of the Possion equation (2.7) has a unique weak solution $u \in h + H^1_{\Gamma_D}(\Omega)$ (i.e., u has boundary values h on Γ_D and g on Γ_N) that continuously depends on f, g, h [37].*

2.2.3 Step 3: Finite Element Approximation

We now know about existence and uniqueness of (weak) solutions to the Poisson problem, and we can introduce a very simple approximation. The solution space and the test function space are simply replaced with finite-dimensional subspaces $V_h^{\Gamma_D} \subset H^1(\Omega)$. The finite element problem is to find $u_h \in V_h$ such that

$$\int_\Omega a\nabla u_h \cdot \nabla v_h \, dx = \int_{\Gamma_N} a\, g \cdot v_h \, dS + \int_\Omega f \cdot v_h dx \quad \forall v_h \in V_h^{\Gamma_D} \tag{2.9}$$

The parameter $h > 0$ usually indicates the quality of the approximation.

$$(V_h) \qquad \begin{cases} find \quad u_h \in V_h, \quad such \quad that \\ u = u_0 \quad \text{on } \Gamma_D \subset \partial\Omega \\ \int_\Omega a\nabla u_h \cdot \nabla v_h \, dx = \int_{\Gamma_N} a\, g \cdot v_h \, dS + \int_\Omega f \cdot v_h \, dx \quad \forall v \in V_h^{\Gamma_D} \end{cases} \tag{2.10}$$

This idea is called the **Galerkin projection** and transforms the infinite-dimensional problem (V) into a finite-dimensional problem (V_h). For minimization problems (M), this process of converting into (M_h) is called **Ritz's method**.

2.2.4 Step 4: Derivation of a Linear System of Equations

Since V_h is finite-dimensional, say the dimension is denoted by $\dim V_h = n$, for the solution of V_h we can find a basis $(v_i)_{i=1}^n$ of (V_h) and replace in equation (2.9) we get

$$\int_\Omega a\nabla u_h \cdot \nabla v_i \, dx = \int_{\Gamma_N} a\, g \cdot v_i \, dS + \int_\Omega f \cdot v_i \, dx \quad \forall i = 1, ..., n \tag{2.11}$$

The solution $u_h \in V_h$ can also be expanded in terms of the basis

$$u_h = \sum_{j=1}^{n} u_j \, v_j(x) \tag{2.12}$$

which we plug into equation (2.11) and get

$$\sum_{i=1}^{n} u_j \int_\Omega a\nabla v_j \cdot \nabla v_i \, dx = \int_{\Gamma_N} a\, g \cdot v_i \, dS + \int_\Omega f \cdot v_i \, dx \quad \forall i = 1, ..., n \tag{2.13}$$

which is similar to liner system of equation $Ax = b$ is called Galerkin system.

$$Au = b \tag{2.14}$$

where $u = (u_i)_{i=1}^n$ and the system matrix and the right hand-side are given by

$$A_{ij} = \int_\Omega a\nabla v_j \cdot \nabla v_i dx \quad b_i = \int_{\Gamma_N} a\, g \cdot v_i \, dS + \int_\Omega f \cdot v_i \, dx \tag{2.15}$$

The system matrix A_{ij} is called stiffness matrix and b_i on the right-hand side is the load vector. Stiffness matrix is symmetric and semidefinite.

2.3 Compute the discrete solution with building blocks of Finite Element Method

Let us discuss some of the essential features of a computer program implementing a finite element method. We will follow the Flow chart (2.2).

- Triangulations: Create mesh on the domain.
- Linear Finite elements: Define on each mesh element $K_i := [x_i, x_{i+1}], i = 0, ..., n$ polynomials for trial and test functions.
- Use the variational (or weak) form of the given problem and derive a discrete version.

- Evaluate the arising integrals.
- Collect all contributions on all K_i leading to a linear equation system $AU = B$.
- Solve this linear equation system; the solution vector $U = (u_0, ..., u_n)^T$ contains the discrete solution at the nodal points $x_0, ..., x_n$.
- Compute error of solution.

We will now see the finite element for 1 dimension and then 2 dimensions. Here 1 dimension is introduced in order to make things simple when we introduce 2 dimensions.

2.3.1 Finite elements in 1D

Triangulations in 1D

Consider the partition of 1D domain $\Omega = [0,1]$ into many small interval $K_i = [x_i, x_{i+1}], i = 0, ..., n$ where $0 = x_0 < x_1 < ... < x_{n+1} = 1$. The points x_i are called nodes, The interval K_i are called elements and their union $\mathcal{T}^h = \bigcup_{k=1}^{n} K_i$ is called mesh or triangulation (it is not always triangles). The mesh parameter h is simply $h := max_{j=0,.....,n+1} h_j$ where $h_j = |x_{j+1} - x_j|$.

Linear finite elements in 1D

Definition 4. *Let P_k the space that contains all polynomials up to order k with coefficients in $\mathbb{R}$:*

$$P_k := \left\{ \sum_{i=0}^{k} a_i x^i | a_i \in \mathbb{R} \right\}$$

For 1D case : The space of linear polynomials is

$$P_1 := \{a_0 + a_1 x | a_0, a_1 \in \mathbb{R}\}. \tag{2.16}$$

A finite element is now a function localized to an element $K_i \in \mathcal{T}_h$ and is uniquely defined by the values in the nodal points x_i, x_{i+1} called **local degrees of freedom**.

We define the space $V_h \subset V$ as the space of all functions v such that

1. v is continuous on $[0,1]$,
2. $v|_{K_i}$ is a linear polynomial for each $i = 0, ..., n+1$,
3. $u = 0$ on Γ_D.

The space $V_h^{(1)}$ is then written as:

$$V_h^{(1)} = V_h := \{v \in C[0,1] | u_{h|K} \in P_1, \forall K_i = [x_i, x_{i+1}], u_h = 0 \quad on \quad \Gamma_D\}.$$

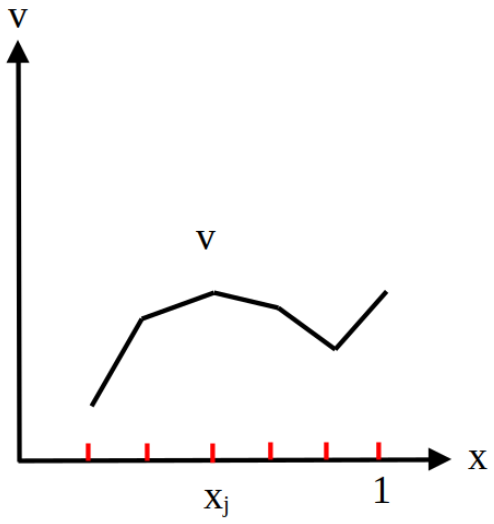


Figure 2.3: A function $v \in V_h$.

All function in V_h are known as **shape functions** and can be represented by **hat functions**. Hat functions are linear functions on each element K_i. Connecting them gives a hat geometrically. It is easy to show that the set of n functions in V_h as shown in Figure (2.3) defined through

$$\varphi(x_j) = \delta_{ij}, \quad i, j = 1, ..., n, \tag{2.17}$$

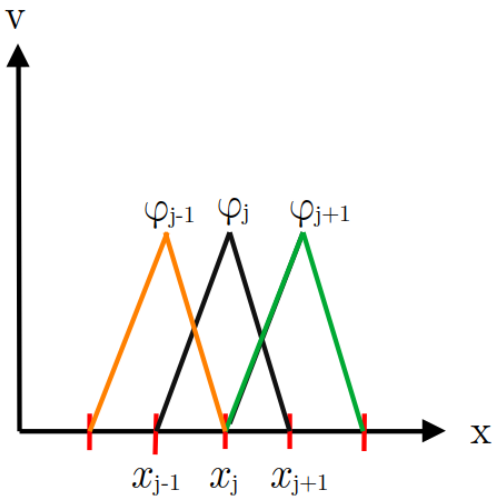


Figure 2.4: Basis in 1 dimension.

where δ_{ij} is the Kronecker delta constitutes a basis in V_h as shown in Figure (2.4). hence

$$\varphi_j(x_i) = \delta_{ij} = \begin{cases} 1 & j = i \\ 0 & j \neq i \quad \text{i, j = 1,2,....,n} \end{cases} \tag{2.18}$$

The aspect of one of these functions is shown in Figure (2.4) and (2.9).

In the piecewise linear case, we can show that

$$\varphi_i(x) = \begin{cases} 0 & x < x_{i-1}, \\ (x - x_{i-1})/(x_i - x_{i-1}) & x_{i-1} < x < x_i, \\ 1 - (x - x_i)/(x_{i+1} - x_i) & x_i < x < x_{i+1}, \\ 0 & x \geq x_{i+1} \end{cases}$$

Lemma 1. *[42] The space V_h is a subspace of $V := H^1(\Omega)$ and has dimension n (because we deal with n basis functions). Thus the such constructed finite element method is a conforming method. Furthermore, for each function $v_h \in V_h$ we have a unique representation:*

$$v_h(x) = \sum_{j=1}^{n} v_{h,j} \varphi_j(x) \quad \forall x \in [0,1], \quad v_{h,j} \in \mathbb{R}.$$

The process to construct the shape functions

We define properties of a finite element as follows:

- Interval $[x_i, x_{i+1}]$,
- A linear polynomial $\varphi(x) = a_0 + a_1 x$,
- Nodal values at x_i and x_{i+1} (the so-called degrees of freedom).

Let's find the unknown coefficients a_0 and a_1 of the shape function. The key property (2.18) using it we get

$$\begin{aligned} \varphi_j(x_j) &= a_0 + a_1 x_j = 1, \\ \varphi_j(x_i) &= a_0 + a_1 x_i = 0, \end{aligned} \tag{2.19}$$

We solve a small linear equation system to find a_0 and a_1 as follows:

$$\begin{pmatrix} 1 & x_j \\ 1 & x_i \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

We get

$$a_1 = -\frac{1}{x_i - x_j}$$

and

$$a_0 = \frac{x_i}{x_i - x_j}$$

Then

$$\varphi_j(x) = a_0 + a_1 x = \frac{x_i - x}{x_i - x_j} \tag{2.20}$$

2.3.2 Finite elements in 2D

Now, we divide the 2D domain $\Omega = [0,1]^2$ into triangles. The method of partitioning domain Ω into triangles is called triangulation. The partitions can also be quadrilaterals. The triangle must contain all sides but not more than that, and it must meet the following:

Definition 5. *A partition $\mathcal{T}_h = K_1, K_2, ..., K_n$ of Ω into triangular or quadrilateral elements is called admissible provided the following properties hold*

1. $\overline{\Omega} = \bigcup_{i=1}^{n} K_i$.
2. *If $K_i \cap K_j$ consists of exactly one point, then it is a common vertex of K_i and K_j.*
3. *If for $i \neq j, K_i \cap K_j$ consists of more than one point, then $K_i \cap K_j$ is a common edge of K_i and K_j .*

Conformality: Conformal meshes are characterized by a perfect match of edges and faces between neighbouring elements. Non-conforming meshes exhibit edges and faces that do not match perfectly between neighbouring elements, giving rise to so-called hanging nodes or overlapped zones. Figure (2.5). a) conforming mesh, b) non-conforming mesh.

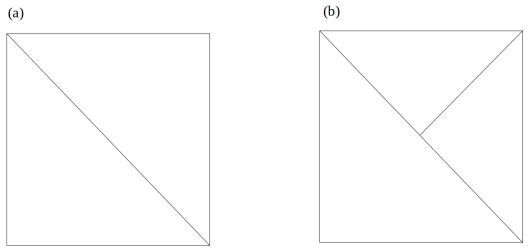


Figure 2.5: (a) Conformal mesh and (b) Non-Conformal mesh.

Let us construct the triangulation of Ω by subdividing Ω into a set $\mathcal{T}_h = K_1, ..., K_n$ of non-overlapping triangles $K_i, i = 1, ..., n$, as shown in Figure (2.6) and (2.7). The points $x_i, i = 1, ..., n$ are called nodes.

$$\Omega = \bigcup_{K \in \mathcal{T}_h} K = K_1 \cup K_2 ..... \cup K_n,$$

such that no vertex of one triangle lies on the edge of another triangle.

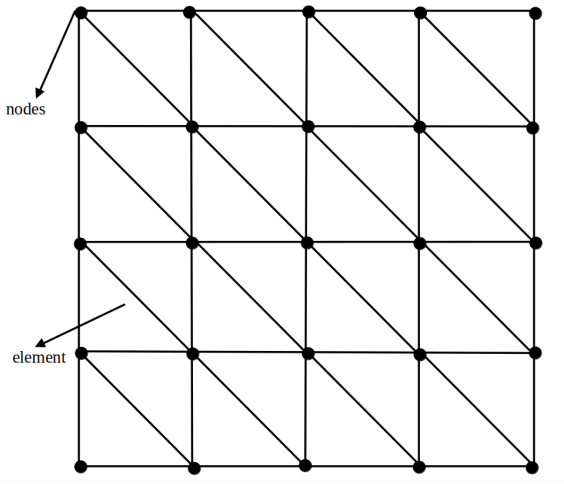


Figure 2.6: 2D triangular mesh.

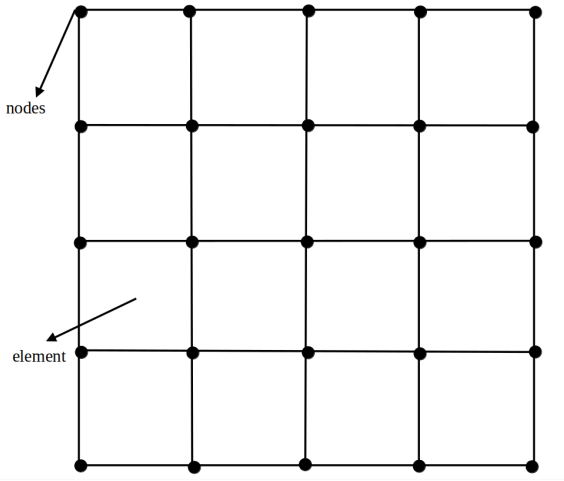


Figure 2.7: 2D square mesh.

The mesh parameter is defined as

$$h = \max_{K \in \mathcal{T}_h} \text{diam(K)}, \text{diam(K)= diameter of K} \simeq \text{longest side of K.}$$

$$V_h = \{v \in C[0,1]^2 | v_{h|K} \in P_1, \forall K \in \mathcal{T}_h, v_h = 0 \quad on \quad \Gamma_D\}.$$

If we fix values on the vertices of the triangulation $\mathcal{T}_h$, then there will exist a unique $v \in V_h$ containing those values on the vertices. As a result, an element of V_h can be uniquely

determined by its values at the triangulation's vertex points. **Global degrees of freedom** is the coefficients at vertices define in the function u_h. We will refer to the nodes as points where values are taken at the vertices in this context.

Let's now look at how triangulation nodes (or vertices) are numbered. The nodes of the triangulation of our model domain are numbered in Figure (2.8).

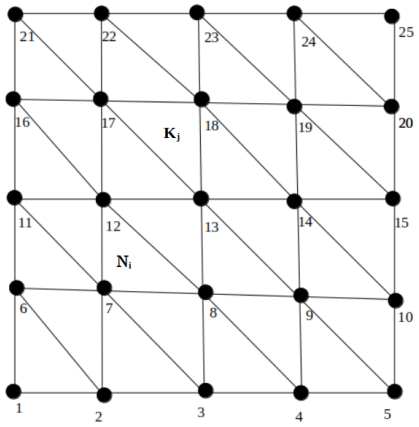


Figure 2.8: Global numbering of nodes.

The space V_h consists of all piecewise linear functions in Ω and vanishes on Γ. As parameters to describe a function $u_h \in V_h$ we choose the values $u_h(N_i)$ of u_h at the nodes $N_i, i = l, ..., n$, of $\mathcal{T}_h$ (see Fig (2.8)) but exclude the nodes on the boundary because $u_h = 0$ on Γ. The corresponding $\varphi_j \in V_h, j = 1, ..., n$, then leads to property as follows as shown in Figure (2.9):

$$\varphi_j(x_i) = \delta_{ij} = \begin{cases} 1 & j = i \\ 0 & j \neq i \end{cases} \quad \text{i, j = 1,2,....,n}$$

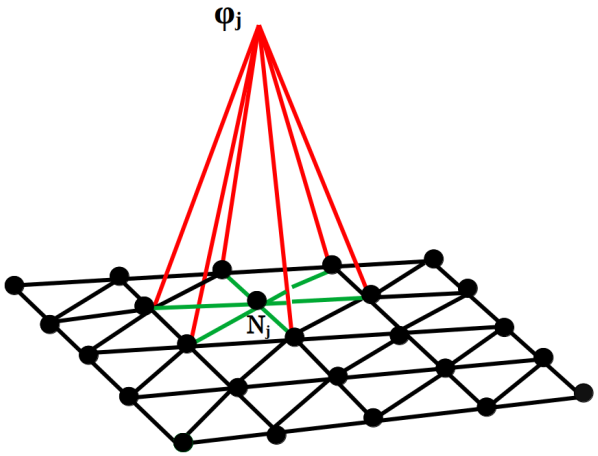


Figure 2.9: Basis in two dimension.

It can seen from Figure (2.10) that the support of φ_j (the set of points x for which $\varphi_j(x) \neq$

0) consists of the triangles with the common node N_j (the green shaded area in Figure (2.10)). A function $v \in V_h$ can be represented as

$$v(x) = \sum_{j=1}^{n} v(N_j)\varphi_j(x). \quad for \quad x \in \Omega \cup \Gamma$$

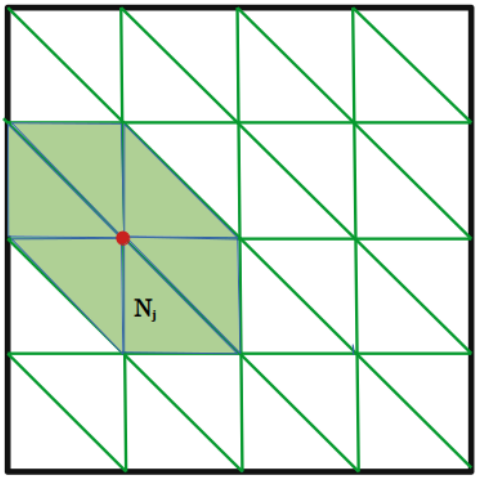


Figure 2.10: Supports of nodal basis functions.

Linear finite elements in 2D

Let K be an element of $\mathcal{T}_h$. A $P_1(K)$ function is defined as

$$v(x_1, x_2) = a_{00} + a_{10}x_1 + a_{01}x_2$$

and coefficients $a_{ij} \in \mathbb{R}$

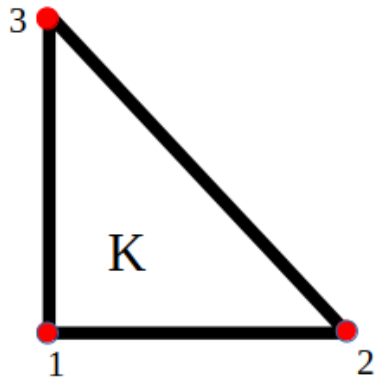


Figure 2.11: A triangle and its three vertices.

Definition 6. *(Basis of P_1) A basis of the polynomial space P_1 is denoted by*

$$P_1 = \{\varphi_1, \varphi_2, \varphi_3\}$$

with the basis functions

$$\varphi_1 = 1 - x_1 - x_2, \varphi_2 = x_1, \varphi_3 = x_2.$$

for unit triangle with (0,0),(0,1) and (1,0) coordinate points. The dimension is $dim(P_1) = 3$. As in Figure (2.11), marking its three vertices, we mean a function as follows

$$p \in P_1 = \{a_{00}\varphi_1 + a_{10}\varphi_2 + a_{01}\varphi_3 | a_{ij} \in \mathbb{R}\}$$

is uniquely determined at values of these points.

Definition 7. *(Basis of P_n) A basis of the polynomial space P_n is denoted by*

$$P_n = \left\{ v : v(x_1, x_2) = \sum_{0 \le i+j \le n} a_{ij} x_1^i x_2^j \right\}$$

The dimension is $dim(P_n) = \frac{(n+1)(n+2)}{2}$.

In deal.II we will use square mesh as shown in Figure (2.7) with Q_1 finite element method define as follows.

$$p \in Q_1 = \{a_{00}\varphi_1 + a_{10}\varphi_2 + a_{01}\varphi_3 + a_{11}\varphi_4 | a_{ij} \in \mathbb{R}\}$$

Definition 8. *(Local degrees of freedom) Let K be a triangle with vertices $a^i, i = 1,2,3$. The values on vertices a^i is called* ***local degrees of freedom****.*

Theorem 2. *[21] Let $K \in \mathcal{T}_h$ be a triangle with vertices $a^i = (a_1^i, a_2^i), i = 1,2,3$. A function $v \in P_1(K)$ is uniquely determined by the degrees of freedom definition (8), i.e, given the values $\alpha_i, i = 1,2,3$, there is a uniquely determined function $v \in P_1(K)$ such that*

$$v(a^i) = \alpha_i$$

Algorithm (Construction of P_1 shape function). For (nodal) basis function of $P_1(K)$ we need to solve

$$v(a^i) = a_{00} + a_{10}a_1^i + a_{01}a_2^i = \alpha_i, \quad i = 1,2,3$$

for the unknown coefficients a_{ij}. This requires three basis functions

$$\psi_i(a^j) = \delta_{ij}$$

One need to solve the following system of equation

$$\psi_i(a^j) = a_{00} + a_{10}a_1^j + a_{01}a_2^j = \alpha_j, \quad i,j = 1,2,3$$

Then we construct local basis function with

$$\psi_i(x) = a_{00}^i + a_{10}^i x_1 + a_{01}^i x_2 = \alpha_i, \quad i = 1,2,3$$

The nodes a^i is sufficient to obtain a globally continuous function, which is in particular also continuous at the edges between two elements. Extending to all edges of $\mathcal{T}_h$ be conclude that the function v is a globally continuous function and that indeed $v \in C[0,1]^2$.

Definition 9. *(C^0 elements). The class of finite elements that fulfills the continuity condition at the edges are called C^0 elements.*

Definition 10. *(Basis of Q_1) A basis of the polynomial space Q_1 is denoted by*

$$Q_1 = \{\varphi_1, \varphi_2, \varphi_3, \varphi_4\}$$

with the basis functions

$$\varphi_1 = 1, \varphi_2 = x_1, \varphi_3 = x_2 . \varphi_4 = x_1 x_2$$

The dimension is $dim(Q_1) = 4$.

$$p \in P_1 = \{a_{00}\varphi_1 + a_{10}\varphi_2 + a_{01}\varphi_3 + a_{11}\varphi_4 | a_{ij} \in \mathbb{R}\}$$

Definition 11. *(Finite Element.) A finite element introduced by Ciarlet is a triple $(K, P_k, \sum)$ where*

- *$K \subset \mathbb{R}^d$ open, bounded connected and with piece-wise smooth boundary is a cell (e.g., triangle, quadrilateral, tetrahedron etc).*
- *P_k is a finite-dimensional space of simple functions (mostly polynomials) on $\mathbb{R}^d$, often called shape functions.*
- *$\sum = \sigma_{i=1}^{dimP_K}$ is a basis of space P_k , the so-called degrees of freedom (dof).*

2.3.3 Assemble System

Assembly involves looping over all cells since all cells must be assembled together. The elements $A_{ij}^K(\varphi_i, \varphi_j)$ in the stiffness matrix A are computed by summing the contributions from the different triangles:

$$A_{ij}^K(\varphi_i, \varphi_j) = \sum_{K \in \mathcal{T}^h} A_K(\varphi_i, \varphi_j) = \sum_{K \in \mathcal{T}^h} \int_K a \nabla \varphi_i \nabla \varphi_j \, dx \tag{2.21}$$

and

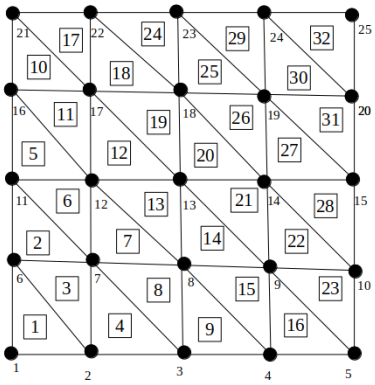
$$b_i = \int_{\Gamma_N} a \, g \cdot \varphi_i \, dS + \int_\Omega \varphi_i f \, dx = \sum_{K \in \mathcal{T}^h} \int_{K \cap \Gamma} a \, g \cdot \varphi_i \, dS + \sum_{K \in \mathcal{T}^h} \int_K \varphi_i f \, dx \tag{2.22}$$

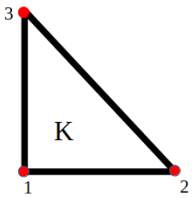
Let N_i, N_j and N_k be the vertices of the triangle K. We define element (local) stiffness matrix for K as follows:

$$\begin{bmatrix} A_K(\varphi_i, \varphi_i) & A_K(\varphi_i, \varphi_j) & A_K(\varphi_i, \varphi_k) \\ A_K(\varphi_i, \varphi_j) & A_K(\varphi_i j, \varphi_j) & A_K(\varphi_j, \varphi_k) \\ A_K(\varphi_i, \varphi_k) & A_K(\varphi_j, \varphi_k) & A_K(\varphi_k, \varphi_k) \end{bmatrix} \tag{2.23}$$

The global stiffness matrix A is computed by first computing the element stiffiness matrices for each $K \in \mathcal{T}_h$ and then sum the contribution from each triangle according to equation (2.21). Similarly the right-hand side b is computed. The process of computing A and b by summation is called assembly of A and b.

Let $N_i, i = 1, ..., M$ and $K_n, n = 1, ..., N$ be enumerations of the nodes and triangles of $\mathcal{T}_h$, respectively. Then $\mathcal{T}_h$ may be defined with two arrays $Y(2,M)$ and $Z(3,N)$, where $Y(j,i), j = 1,2$, are the coordinates of nodes N_i and $Z(j,n), j = 1,2,3$, are the number of the vertices of triangle K_n. Consider the following triangulation where the numbers of the triangles are denoted by a square see the Figure (2.12).





local		global
1	↔	5
2	↔	10
3	↔	9

Figure 2.12: The 23rd triangle and their vertex numberings.

As shown in Figure (2.12), the assembly process requires a given number of triangles. The numbering order is only used for computations and does not affect the final results.

$$Z = \begin{bmatrix} \begin{bmatrix} 1 & 6 & 2 & 2 & 11 & 7 & 7 & 3 & 3 & 16 & 12 & 12 & 8 & 8 & 4 & 4 \\ 2 & 7 & 7 & 3 & 12 & 12 & 8 & 8 & 4 & 17 & 17 & 13 & 13 & 9 & 9 & 5 \\ 6 & 11 & 6 & 7 & 16 & 11 & 12 & 7 & 8 & 21 & 16 & 17 & 12 & 13 & 8 & 9 \end{bmatrix} \\ \begin{bmatrix} 17 & 17 & 13 & 13 & 9 & 9 & 5 & 18 & 18 & 14 & 14 & 10 & 19 & 19 & 15 & 20 \\ 22 & 18 & 18 & 14 & 14 & 10 & 10 & 23 & 19 & 19 & 15 & 15 & 24 & 20 & 20 & 25 \\ 21 & 22 & 17 & 18 & 13 & 14 & 9 & 22 & 23 & 18 & 19 & 14 & 23 & 24 & 19 & 24 \end{bmatrix} \end{bmatrix}$$

Computation of the element stiffness matrices

Now to compute the element stiffness matrices with elements A_{ij}^K given by equation (2.21). The matrix $A_{ij}^K \neq 0$ if both N_i and N_j are nodes of K. Let $K_n \in \mathcal{T}_h$. Then $Z(\alpha, n), \alpha = 1,2,3$ are the numbers of the vertices of K_n, and the x_i-coordinates, $i = 1,2$, for these vertices are given by $Y(i, Z(\alpha, n)), \alpha = 1,2,3$. Since the vertices of K_n are known one can compute the element stiffiness matrix $A^n = (A_{\alpha\beta}^n), \alpha, \beta = 1,2,3$, for element K_n

$$A_{\alpha\beta}^n = \int_{K_n} \nabla\psi_\alpha \cdot \nabla\psi_\beta dx,$$

where ψ_α is the linear function on K_n that takes the following values:

$$\psi_\alpha(N_{Z(\beta,n)}) = \begin{cases} 1 & \alpha = \beta \\ 0 & \alpha \neq \beta \end{cases} \quad \alpha,\beta = 1,2,3.$$

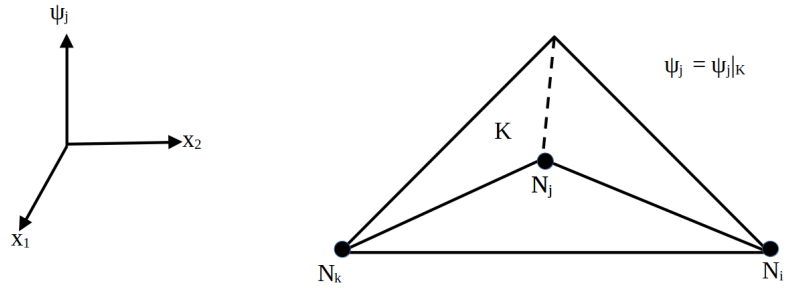


Figure 2.13: The basis function ψ_j associated with K.

The stiffness matrix (2.23) is computed using the basis functions ψ_i, ψ_j and ψ_k which are restricted to the triangle K. By denoting these by ψ_i, ψ_j and ψ_k, we have that each ψ is a linear function on K that takes one value at one vertex and vanishes at the other two. The basis functions on the triangle K are ψ_i, ψ_j and ψ_k as shown in Figure (2.13). When w is a linear function on K, then w is written as

$$w(x) = w(N_i)\psi_i(x) + w(N_j)\psi_j(x) + w(N_k)\psi_k(x), \quad x \in K \tag{2.24}$$

Similarly b is computed as

$$b^n_\alpha = \int_{K_n} f\psi_\alpha \, dx + \int_{K\cap\Gamma} a\, g \cdot \psi_\alpha \, dS, \quad \alpha = 1,2,3.$$

In terms of code. we need subrotine that computes the element stiffiness matix $A^n = (A^n_{\alpha\beta})$ and the right hand side $b^n = (b^n_\alpha)$ for a given triangle K_n. This process is done by looping over all elements of K_n and store local matrix.

Assembly of global stiffness matrix

For global stiffness matrix $A = (A_{ij})$ we loop over all elements K_n and add their contributions from different K_n as follows for $A(M,M)$ and $b(M)$ arrays for matrix A and right hand side b :

Set $A(i,j) = 0, \quad b(i) = 0, i, j = 1, ..M.$
For $n = 1, ..., N$ fetch $A^n = (A^n_{\alpha\beta})$ and $b^n = (b^n_\alpha)$ from the local matrix and set
$A(Z(\alpha,n), Z(\beta,n)) = A(Z(\alpha,n), Z(\beta,n)) + A^n_{\alpha\beta},$
$b(Z(\alpha,n)) = b(Z(\alpha,n)) + b^n_\alpha \quad \alpha,\beta = 1,2,3.$

Algorithm 1 Algorithm for Assembly of global stiffness matrix.

Let $K_n, \quad n = 0, ...N$ be an element and let i and j be the indices of the degrees of freedom (namely the basis functions).
The basic algorithm to compute all entries of the system matrix and right hand side vector is:
for all elements K_n with $n = 0, ..., N$
for all DoFs i with $i = 0, ..., n+1$
for all DoFs j with $j = 0, ..., n+1$
$A_{ij} += \int_{K_n} \nabla\varphi_i(x)\nabla\varphi_j(x)dx$
where + = means that entries with the same indices i, j are summed.
For the right hand side, we have
for all elements K_n with $n = 0, ..., N$
for all DoFs i with $i = 0, ..., n+1$
$b_i += \int\limits_{K_n \cap \Gamma} a\, g \cdot \varphi_i \, dS + \int_{K_n} \varphi_i(x)dx.$
Again + = means that only the bi with the same i are summed.

In order to optimize the evaluation of the nodal basis functions and their gradients (as well as to increase performance and accuracy), all calculations are performed on a reference cell.

The reference element for P_1 finite element method

Let $(\hat{K}, P_K, \hat{\Sigma})$ be a finite element where $\hat{K}$ is the reference triangle in the $(\hat{x}_1, \hat{x}_2)$- plane with vertices at $\hat{a}^1 = (0,0), \hat{a}^2 = (1,0)$ and $\hat{a}^3 = (0,1)$, as shown in Figure (2.14).

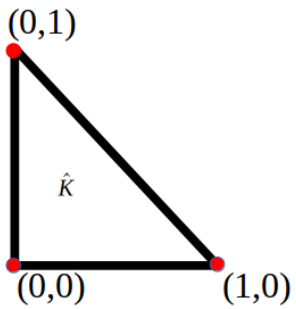


Figure 2.14: The reference element.

2.3.4 2-D elements: coordinate transformation

Let $\hat{\Sigma}$ as the Langrage type degree of freedom, i.e. $\hat{\Sigma}$ is a set of function values at certain points $\hat{a}^i \in \hat{K}, i = 1, ..., n$. Let F be a one to one mapping of $\hat{K}$ onto the traingle K in the (x_1, x_2) plane as shown in Figure (2.15) with inverse F^{-1}

Let us now define transformation F by

$$F(\hat{x}) = \sum_{j=1}^{3} a^j \varphi_j(\hat{x}), \quad \hat{x} \in \hat{K},$$

and let us write

$$K = F(\hat{K}) = x \in \mathbb{R}^2 : x = F(\hat{x}) = a^1 + (a^2 - a^1)\hat{x}_1 + (a^3 - a^1)\hat{x}_2, \quad \hat{x} \in \hat{K}$$

Then $F : \hat{K} \to K$ and $F(\hat{a}^j) = a^j, \quad j = 1,2,3$ the points $\hat{a}^j$ in the $\hat{x}$-plane are mapped onto the points $\hat{a}^j$ in the x-plane as shown in Figure (2.15).

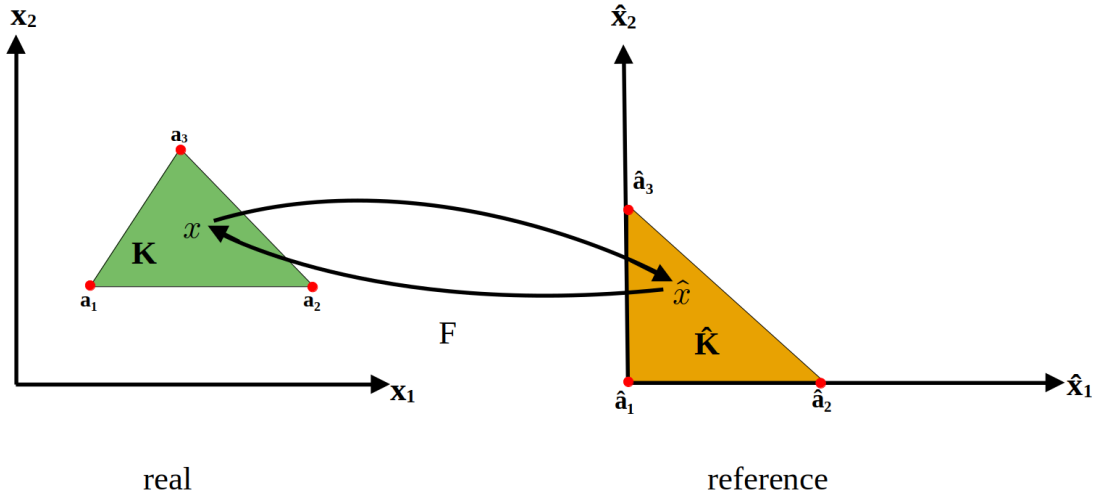


Figure 2.15: The real element in left and reference element right.

The local basis functions on K are given by

$$\varphi_j = \hat{\varphi}_j(F^{-1}(x)), j = 1, ..., 6$$

where $\hat{\varphi}_j, j = 1,2,3$ is basis for $P_1(\hat{K})$.
To compute the integrals

$$a_{ij}^K = \int_K \nabla\varphi_i \nabla\varphi_j \, dx, \quad i, j = 1,2,3.$$

By chain rule

$$\frac{\partial \varphi_i}{\partial x_j} = \frac{(\hat{\varphi}_i(F^{-1}(x)))}{\partial x_j} = \frac{\partial \hat{\varphi}_i}{\partial \hat{x}_1}\frac{\partial \hat{x}_1}{\partial x_j}\frac{\partial \hat{x}_2}{\partial \hat{x}_j}$$

so that

$$\nabla \varphi_i = J^{-T} \nabla \hat{\varphi}_i,$$

where J^{-T} is the transposed Jacobian of the mapping F^{-1},

$$J^{-T} = \begin{bmatrix} \frac{\partial \hat{x}_1}{\partial x_1} & \frac{\partial \hat{x}_2}{\partial x_1} \\ \frac{\partial \hat{x}_1}{\partial x_2} & \frac{\partial \hat{x}_2}{\partial x_2} \end{bmatrix}$$

To transform the integral in equation 2.3.4 to an integral $\hat{K}$ using the mapping $F : \hat{K} \to K$, we get

$$J^{-T} = (J^{-1})^T = \frac{1}{detJ} J_0,$$

where

$$J_0 = \begin{bmatrix} \frac{\partial F_2}{\partial \hat{x}_2} & -\frac{\partial F_2}{\partial \hat{x}_1} \\ \frac{\partial F_1}{\partial \hat{x}_2} & -\frac{\partial F_1}{\partial \hat{x}_1} \end{bmatrix}$$

so finally

$$a_{ij}^K = \int_{\hat{K}} (J_0 \nabla \hat{\varphi}_i)(J_0 \nabla \hat{\varphi}_j) \frac{d\hat{x}}{|detJ|} \tag{2.25}$$

Thus the matrix element a_{ij}^K can be computed by evaluating an integral over the reference element $\hat{K}$.

The reference space for Q_1 finite element method

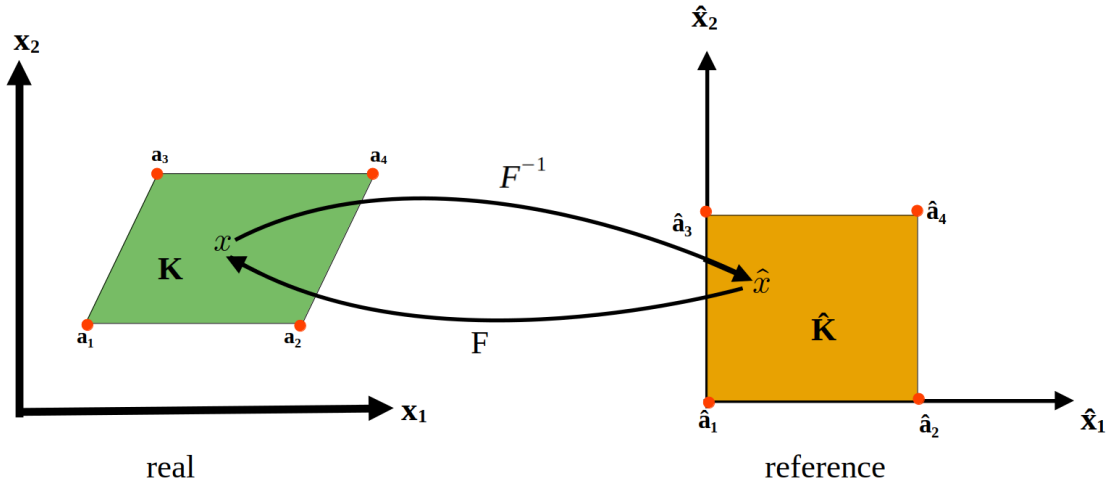


Figure 2.16: Q_1 Basis Function.

2.3.5 Computing with quadrature rules

The stiffness matrix (2.25) is computed using suitable quadrature formula. In practice, such integrals would be evaluated using a numerical quadrature formula of the form:

$$\int_K f(x)dx \sim \sum_{j=1}^{q} f(y^j)w_j \tag{2.26}$$

where the w_j, $j = 1,...q$, are certain weights and the y^j are points in the element K. Based on the complexity of the problem (we are dealing with a very simple model problem), all computations can be performed on the reference element or directly on the triangle K. Here are two quadrature rules for triangles: the three-point rule with vertices

$$\int_K \varphi \approx \frac{areaK}{3}\left(\varphi(p_1^K)+\varphi(p_2^K)+\varphi(p_3^K)\right)$$

and the midpoints approximation

$$\int_K \varphi \approx \frac{areaK}{3}\left(\varphi(m_1^K)+\varphi(m_2^K)+\varphi(m_3^K)\right)$$

where m_α^K are the midpoints of the edges of K. If φ is a polynomial of degree one, the first formula gives the exact value. The second formula is even better : if φ is a polynomial of degree two, the midpoint formula is exact. The local matrix contribution a_{ij}^K can be

approximated using quadrature as follows

$$a_{ij}^K = \sum_q \nabla\hat{\varphi}_i(x_q) \cdot \nabla\hat{\varphi}_j(x_q) w_q, \quad 0 \leq i, j < q, \tag{2.27}$$

where x_q are a set of quadrature points and w_q are the corresponding weights.

In deal.II using Q_1 finite element with square mesh 2D basis function as shown in Figure (2.17) and (2.18).

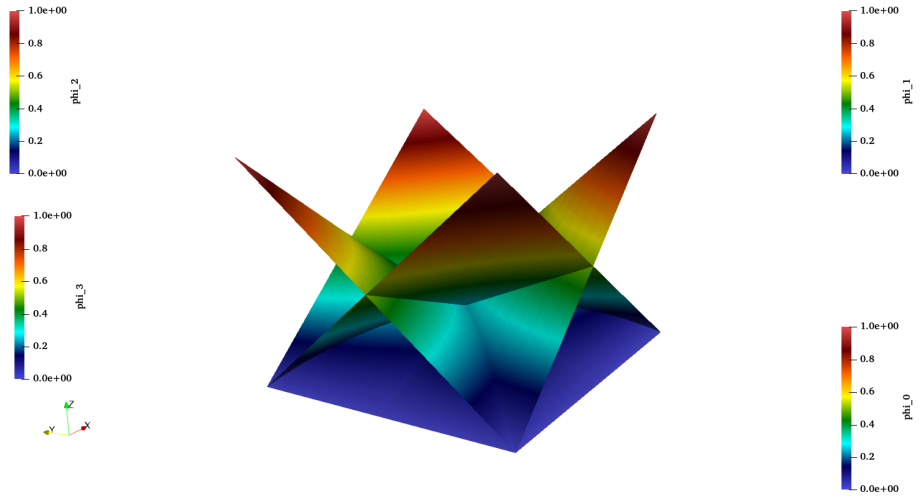


Figure 2.17: 2D basis for square mesh in deal.II.

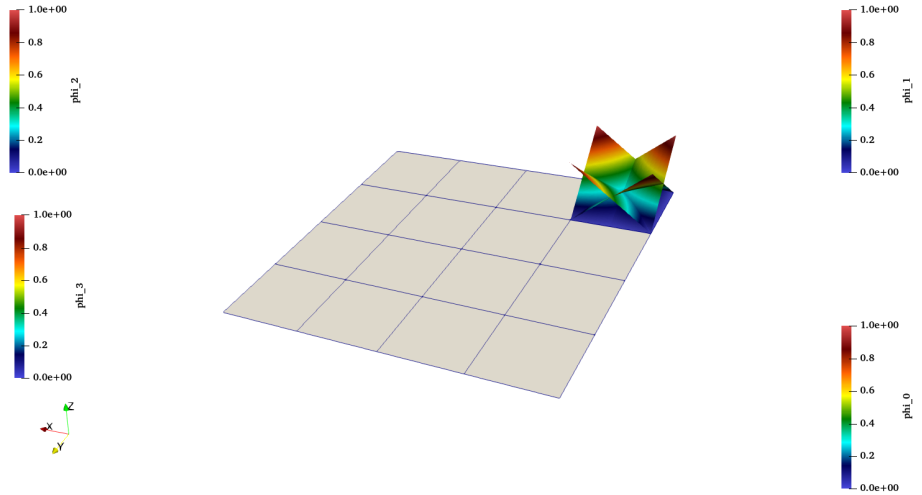


Figure 2.18: 2D basis for square mesh with mesh in deal.II.

2.3.6 Compute the error

Lemma 2. *(Céa's lemma) Let V be a Hilbert space, V_h be a linear subspace of V. Let the bilinear form $a(\cdot, \cdot)$ and the linear form $f(\cdot)$ satisfy the conditions of the Lax-Milgram lemma . Let $u \in V$ be the solution to the variational problem, and $u_h \in V_h$ satisfy the*

equation

$$a(u_h, \varphi) = f(\varphi) \quad \forall \varphi \in V_h.$$

Then, there exist a constant C independent of V_h, such that

$$||u - u_h|| \leq C \inf_{\varphi \in V_h} ||u - \varphi||$$

where $\|\cdot\|$ is the norm of V.

According to Céa's lemma, the accuracy of a numerical solution depends essentially on choosing function spaces which are capable of approximating the solution u well. For polynomials, the order of approximation is determined by the smoothness of the solution. Céa's lemma holds for Poisson's equation when using the finite element method.

2.4 Abstract form of Finite Element Method

Lemma 1.1 (The Lax-Milgram Lemma). Assume the following

1. V is a Hilbert space;

2. $a(,)$ is continuous on V , i.e. there exists $C > 0$ such that

$$|a(u,v)| \leq C||u||_V||v||_V \quad \forall u, v \in V;$$

3. $a(,)$ is V–coercive on V , i.e. there exists $\alpha > 0$ such that

$$|a(u,v)| \geq \alpha||u||_V^2 \quad \forall u \in V;$$

4. $l()$ is continuous on V , i.e. there exists $M > 0$ such that

$$|l(u)| \leq M||v||_V \quad \forall v \in V;$$

Then, there exists a unique solution $u \in V$ to $a(u,v) = l(v) \quad \forall v \in V$ such that

$$||u||_V \leq \frac{M}{\alpha}||l||_{V'}$$

Let us consider the general problem

$$\begin{cases} Lu = f & \text{in } \Omega \subset \mathbb{R}^d \\ u = u_0 & \text{on } \Gamma_D \subset \partial\Omega \\ \nabla_n u = g & \text{on } \Gamma_N \subset \partial\Omega \end{cases}$$

This is the strong form. Now let write the weak form for that let V be a Hilbert space with inner product $< .,. >$, then

$$< Lu, v >=< f, v >$$

Define

$$\begin{cases} a(u,v) =< Lu, v > \\ f(v) =< f, v > \end{cases}$$

where a is a bilinear form(not necessarily an inner product) and f is a linear form(a functional):

$$\begin{cases} a : V \times \hat{V} \to \mathbb{R} \\ F : \hat{V} \to \mathbb{R} \end{cases}$$

The vartional problem becomes: find $u \in V$ such that

$$a(u,v) = f(v) \quad \forall v \in \hat{V}$$

For a discrete solution: find $u_h \in \hat{V}_h$

$$a(u_h, v) = f(v) \quad \forall v \in \hat{V}_h$$

Let $\{\varphi_i\}_{i=1}^n$ be a basis for V_h. Take an Anzats

$$u_h(x) = \sum_{j=1}^{n} u_j \varphi_j(x)$$

Insert this to the vartional form, we get

$$\left(\sum_{j=1}^{n} u_j \varphi_j \hat{\varphi}_i \right) = F(\hat{\varphi}_i) \quad i = 1, 2, ..., n$$

$$\sum_{j=1}^{n} u_j a(\varphi_j \hat{\varphi}_i) = F(\hat{\varphi}_i) \quad i = 1, 2, ..., n$$

As previous, u_h is computed by solving a linear system

$$\sum_{j=1}^{n} A_{ij}u_j = b_i \quad i = 1,2,....,n$$

$$Au = b$$

where $A_{ij} = a(\varphi_j, \hat{\varphi}_i), b_i = F(\hat{\varphi}_i)$

2.5 Galerkin Orthogonality

Given

$$\begin{cases} a(u,v) = F(v) & \forall v \in V \\ a(u_h,v) = F(v) & \forall v \in V_h \subset V \end{cases}$$

Using these results and the linearity of the bilinear form, we get

$$a(u-u_h,v) = a(u,v) - a(u_h,v) = F(v) - F(v) = 0 \quad \forall v \in V_h$$

or written symbolically Using these results and the linearity of the bilinear form, we get

$$u - u_h \perp_a V_h$$

This property is called Galerkin orthogonality. The error $e = u - u_h$ is orthogonal to the test space V_h. Thus, u_h is the best approximation of u in V_h as shown in Figure (2.19).

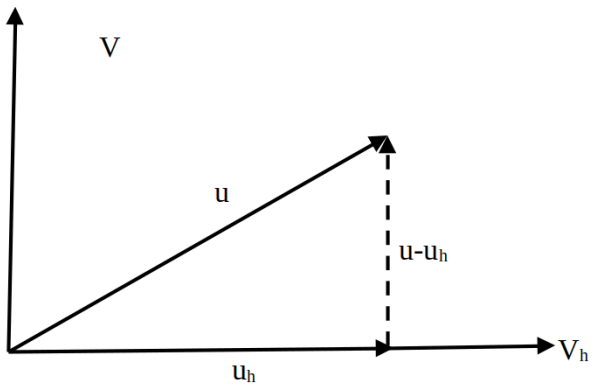


Figure 2.19: Galerkin orthogonality.

Here we conclude the mathematical description of finite element method. The reader is then required to read Chapter 5 and view the whole code in the Github `https://github.com/heena008/Diffusion-Equation-with-MsFEM.io`.

3 Multiscale Finite Element Method

"Divide each difficulty into as many parts as is feasible and necessary to resolve it."

- Rene Descartes

3.1 Introduction Multiscale Modeling

Multiscale modeling involves modeling two or more scales coupled bidirectionally, with a finer scale enclosing a coarser scale. Multiscale models are based on the idea of reducing global degrees of freedom while preserving important features of flow and transport processes.

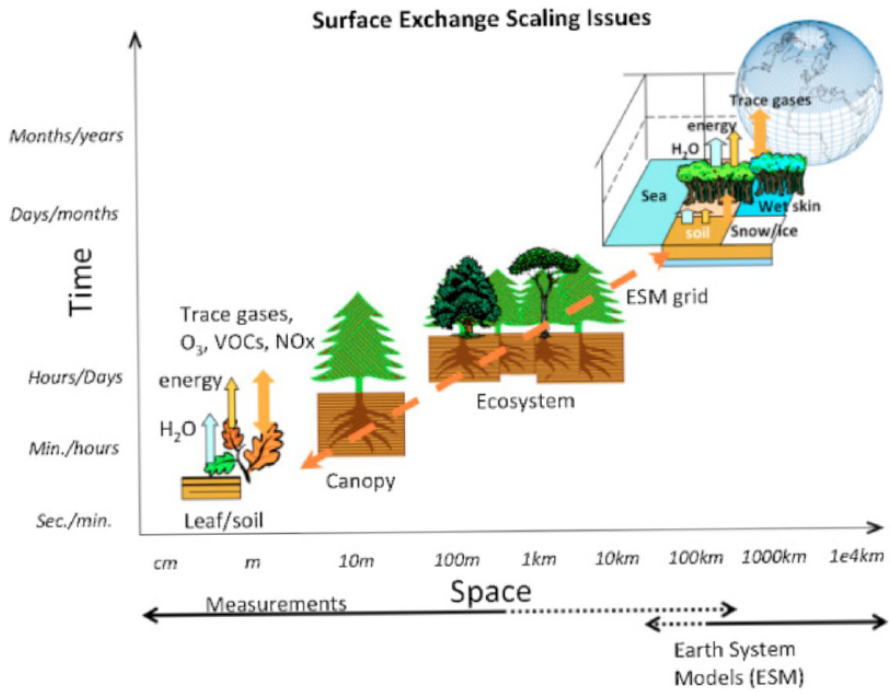


Figure 3.1: Canopy representation at various scales. Figure is courtesy of Laurens Ganzeveld [41].

There are scaling issues for surface exchange outlined in Figure (3.1), but the idea here is to show canopy representation at different scales.

Our focus will be on problems of various scales. A transition from a finer to a coarser scale is called upscaling, while a transition from a coarser to a finer scale is called downscaling.

A classic upscaling strategy is the asymptotic expansion (homogenization) and volume averaging. An effective parameter in the upscale equations usually compensates for the loss of fine-scale information due to averaging. A typical methodology for downscaling specifies boundary conditions at the boundaries of a coarse-grid block and solves a fine-grid problem in each domain. Boundary conditions were either obtained directly from coarse-scale problems or rescaled from coarse-scale results using fine-scale material parameters. Alternatively, fine-grid boundary conditions can be specified along the boundaries of the downscaling domain.

3.1.1 Introduction to Multiscale in canopy

- Canopy dimensions are in meters and time steps are in seconds.
- Climate models that have a mesh size in km and time steps in minutes cannot resolve canopy at given computational cost.
- Hence Canopy is parameterized and their coupling to climate models does not represent the subgrid process correctly.

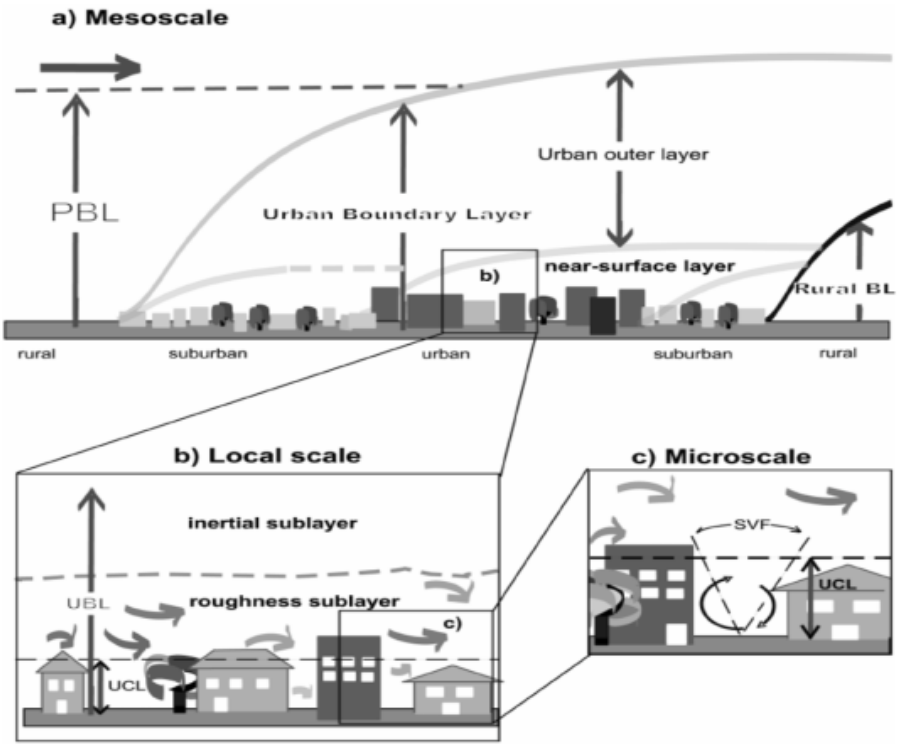


Figure 3.2: Scales in Canopy [33].

Figure (3.2) shows that canopy resolution ranges from microscale, where flow around two buildings is considered, to mesoscale, where time steps are in seconds. The flow around

neighboring buildings can also be resolved to some extent on a local scale. Mesoscale, however, when the entire city is considered, the mesh would be kilometers and the time steps would be minutes. There is not computationally feasible to considering a fine mesh that resolves scale and flow. In climate models, canopy features are taken into account through parametrization, and the way they are coupled to larger scales by averaging does not represent them accurately.

3.2 Homogenization theory

Homogenization theory studies the limiting behavior $u_\varepsilon \to u_0$ as $\varepsilon \to 0$. Weak convergence and averaging procedures share many features with this procedure. The content in these section is from the reference book [44]. Consider the second-order elliptic equation.

$$-\frac{\partial}{\partial x_i}\left(a_{ij}(x/\varepsilon)\frac{\partial}{\partial x_j}\right)u_\varepsilon + a_0(x/\varepsilon)u_\varepsilon = f, \quad u_{\varepsilon|\partial\Omega}$$

where x is also called the macroscopic variable, (x/ε) the microscopic variable. The matrix $a_\varepsilon = a_{ij}(x/\varepsilon)$ diffusion coefficient can be any second order tensor that is bounded and positive definite, $a_{ij}(y)$ and $a_0(y)$ are 1-periodic in both variables of y, $\xi \in \mathbb{R}^N$ is any vector, that satisfy $a_{ij}(y)\xi_i\xi_j \geq \alpha\xi_i\xi_j$, with $\alpha > 0$, $a_0 > \alpha_0 > 0$, and bounded.

Here we have used the Einstein summation notation; that is a repeated index means summation with respect to that index.
Homogenization theory studies the limiting behavior $u_\varepsilon \to u_0$ as $u \to 0$.
The main task is to find the homogenized coefficients, a^*_{ij} and a^*_0, and the homogenized equation for the limiting solution u.

$$-\frac{\partial}{\partial x_i}\left(a^*_{ij}\frac{\partial}{\partial x_j}\right)u_0 + a^*_0 u_0 = f, \quad u_{0|\partial\Omega} = 0$$

We define the bilinear form

$$a^\varepsilon(u,v) = \int_\Omega a^\varepsilon_{ij}(x)\frac{\partial u}{\partial x_j}\frac{\partial v}{\partial x_i}\,dx + \int_\Omega a^\varepsilon_0 uv\,dx.$$

The elliptic problem can also be formulated as a variational problem: find $u_\varepsilon \in H^1_0$

$$a^\varepsilon(u_\varepsilon, v) = (f,v), \quad \text{for all} \quad v \in H^1_0(\Omega),$$

where (f,v) is the usual L^2 inner product, $\int_\Omega fv\,dx$.

Consider the domain Ω with oscillation ε as shown in Figure (3.3).

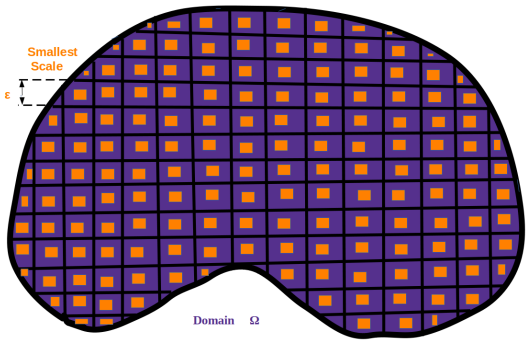


Figure 3.3: Domain Ω with oscillation ε.

Let multiscale differential or integral equation be given by

$$F_\varepsilon(u_\varepsilon) = 0 \tag{3.1}$$

where F_ε represents the differential equations with initial and boundary conditions.

Analytically, we are interested in the following the limit process

$$\lim_{\varepsilon \to -0} u_\varepsilon = \bar{u}, \quad \bar{F}(\bar{u}) = 0 \tag{3.2}$$

The type of convergence could be weak or strong and in different norms depending on the application. F represents an effective equation for large scale.

One-dimensional problem

Let $\Omega = (x_0, x_1)$ and take $a_0 = 0$. We have

$$-\frac{d}{dx}\left(a(x/\varepsilon)\frac{du_\varepsilon}{dx}\right) = f, \quad in \quad \Omega,$$

where $u_\varepsilon(x_0) = u_\varepsilon(x_1) = 0$, and $a(y) > \alpha_0 > 0$ is y-periodic with period y_0.
By taking $v = u_\varepsilon$ in the variational problem, we have

$$||u_\varepsilon||_{1,\Omega} \leq C.$$

Therefore one can extract a subsequence, still denoted by u_ε, such that

$$u_\varepsilon \to u \quad in \quad H_0^1(\Omega) \quad weakly.$$

Next, we introduce

$$\xi_\varepsilon = a_\varepsilon \frac{du_\varepsilon}{dx}$$

Because a_ε is bounded, and $\frac{du_\varepsilon}{dx}$ is bounded in $L^2(\Omega)$, so ξ_ε is bounded in $L^2(\Omega)$. Moreover, because $-\frac{d\xi_\varepsilon}{dx} = f$, we have $\xi_\varepsilon \in H^1(\Omega)$. Thus we get

$$\xi_\varepsilon \to \xi \quad in \quad L^2(\Omega) \quad stronly.$$

so that

$$\frac{1}{a_\varepsilon}\xi_\varepsilon \to m(1/a)\xi \quad in \quad L^2(\Omega) \quad weakly.$$

Furthermore, we note that $\xi_\varepsilon/a_\varepsilon = du_\varepsilon/dx$. Therefore, we arrive at

$$\frac{du_0}{dx} = m(1/a)\xi.$$

On the other hand, $-d\xi_\varepsilon/dx = f$ implies $-d\xi/dx = f$. This gives

$$-\frac{d}{dx}\Big(\frac{1}{m(1/a)}\frac{du_0}{dx}\Big) = f$$

This is the correct homogenized equation for u.
Note that $a^* = 1/m(1/a)$ is the harmonic average of a_ε.
It is in general not equal to the arithmetic average $\bar{a_\varepsilon} = m(a)$ [44]. It shows that climate models are over diffused due to incorrect averaging.

Multiscale asymptotic expansions.

It is not possible to generalize the above analysis to multidimensions. A multiscale expansion technique is presented in this subsection for obtaining homogenized equations. We find for $u_\varepsilon(x)$ in the form of asymptotic expansion

$$u_\varepsilon(x) = u_0(x,x/\varepsilon) + \varepsilon u_1(x,x/\varepsilon) + \varepsilon^2 u_2(x,x/\varepsilon) + ...,$$

where the functions $u_j(x,y)$ are periodic in y with period 1.

Denote by S_ε the second-order elliptic operator

$$S_\varepsilon(x) = -\frac{\partial}{\partial x_i}\left(a_{ij}(x/\varepsilon)\frac{\partial}{\partial x_j}\right)$$

Differentiating a function $\varphi(x,x/\varepsilon)$ with respect to x, we have

$$\frac{\partial}{\partial x_j} = \frac{\partial}{\partial x_j} + \frac{1}{\varepsilon}\frac{\partial}{\partial y_j}$$

where y is evaluated at $y = x/\varepsilon$. With this notation, we can expand S_ε as follows,

$$S_\varepsilon(x) = \varepsilon^{-2}S_1 + \varepsilon^{-1}S_2 + \varepsilon^0 S_3, \tag{3.3}$$

where

$$\begin{aligned} S_1 &= -\frac{\partial}{\partial y_i}\left(a_{ij}(y)\frac{\partial}{\partial y_j}\right), \\ S_2 &= -\frac{\partial}{\partial y_i}\left(a_{ij}(y)\frac{\partial}{\partial x_j}\right) - \frac{\partial}{\partial x_i}\left(a_{ij}(y)\frac{\partial}{\partial y_j}\right), \\ S_3 &= -\frac{\partial}{\partial x_i}\left(a_{ij}(y)\frac{\partial}{\partial x_j}\right) + a_0. \end{aligned} \tag{3.4}$$

Substitute the expansions for u_ε and S_ε into $S_\varepsilon u_\varepsilon = f$, and equating the terms of the same power, we get

$$S_1 u_0 = 0, \tag{3.5}$$

$$S_1 u_1 + S_2 u_0 = 0, \tag{3.6}$$

$$S_1 u_2 + S_2 u_1 + S_3 u_0 = f. \tag{3.7}$$

Equation (3.5) can be written as

$$-\frac{\partial}{\partial y_i}\left(a_{ij}(y)\frac{\partial}{\partial y_j}\right) u_0(x,y) = 0,$$

where u_0 is periodic in y. $u_0(x,y)$ is independent of y; that is $u_0(x,y) = u_0(x)$. This simplifies equation (3.6) for u_1,

$$-\frac{\partial}{\partial y_i}\left(a_{ij}(y)\frac{\partial}{\partial y_j}\right) u_1 = \left(\frac{\partial}{\partial y_i}a_{ij}(y)\right)\frac{\partial u}{\partial x_j}(x).$$

Define $\chi^j = \chi^j(y)$ as the solution to the following cell problem

$$\frac{\partial}{\partial y_i}\left(a_{ij}(y)\frac{\partial}{\partial y_j}\right)\chi^j = -\frac{\partial}{\partial y_i}a_{ij}(y),$$

where χ^j is periodic in y. The general solution of (3.6) for u_1 is then given by

$$u_1(x,y) = \chi^j(y)\frac{\partial u}{\partial x_j}(x) + \tilde{u}_1(x).$$

Finally, the equation u_2 is given by

$$\frac{\partial}{\partial y_i}\left(a_{ij}(y)\frac{\partial}{\partial y_j}\right)u_2 = S_2u_1 + S_3u_0 - f. \tag{3.8}$$

The solvability condition implies that the right-hand side of equation (3.8) must have mean zero in y over one periodic cell $Y = [0,1]\times[0,1]$; that is

$$\int_Y (S_2u_1 + S_3u_0 - f)\,dy = 0.$$

This solvability condition for second-order elliptic PDEs with periodic boundary condition requires that the right-hand side of equation (3.8) have mean zero with respect to the fast variable y. This solvability condition gives rise to the homogenized equation for u:

$$-\frac{\partial}{\partial x_i}\left(a^*_{ij}\frac{\partial}{\partial x_j}\right)u + m(a_0)u = f, \tag{3.9}$$

where $m(a_0) = (1/|Y|)\int_Y a_0(y)dy$ and

$$a^*_{ij} = \frac{1}{|Y|}\left(\int_Y (a_{ij} - a_{ik}\frac{\partial \chi^j}{\partial y_k})dy\right) \tag{3.10}$$

The homogenized coefficients are often difficult to compute when the periodic cell problem requires very fine discretization. It is useful to know the bounds for the homogenized coefficients in this case. It is difficult to find accurate bounds for heterogeneities. A number of works have been published that compute bounds and determine optimal microstructures. For homogenized solutions, one can avoid solving the cell problems if there are tight bounds [44]. Hence we cannot use homogenized process for our problem.

3.3 Multiscale Finite Element Method

A MsFEM consists of two major parts: multiscale basis functions and a global numerical formulation that combines these multiscale basis functions. Multiscale features of the

solution are captured by basis functions. They contain information about scales defined by scales smaller (and larger) than the local numerical scale defined by the basis functions, which are important for multiscale features of the solution. Global formulations provide an accurate approximation of the solution by coupling these basis functions.

3.4 Objective of Multiscale Finite Element Method

- To resolve $H > \varepsilon$, we solve some coarse scale global problems as shown in Figure (3.4).
- By considering the microstructure, significant features of the solution are captured (to resolve $h < \varepsilon$).
- On a fine scale, the problem is fully resolved, but it is not fully coupled.
- The global problem is the reduced degree of freedom system.

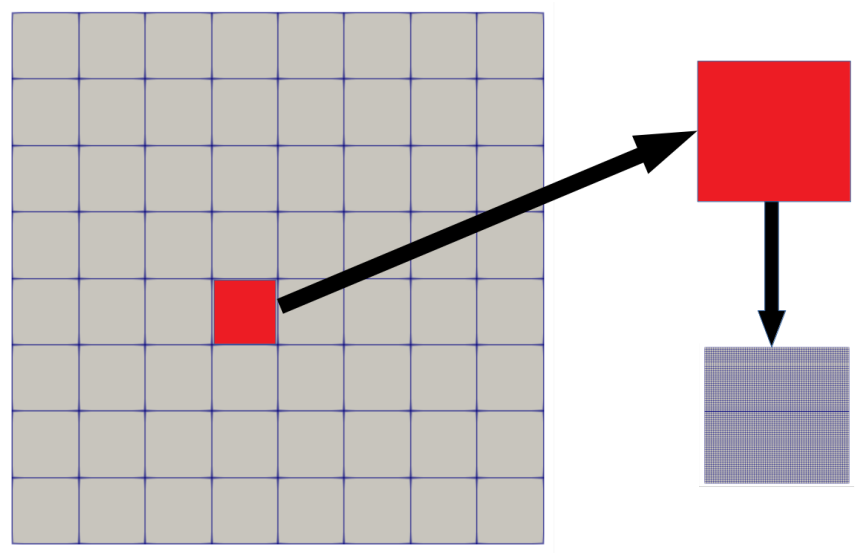


Figure 3.4: Objective of Multiscale Method.

3.5 Steps of Multiscale Finite Element Method

Given a second-order diffusion equation in a domain $\Omega \in \mathbb{R}^d$ with $d \in \{2,3\}$. The aim is to find $u : \Omega \to \mathbb{R}$ solution of the following equation.

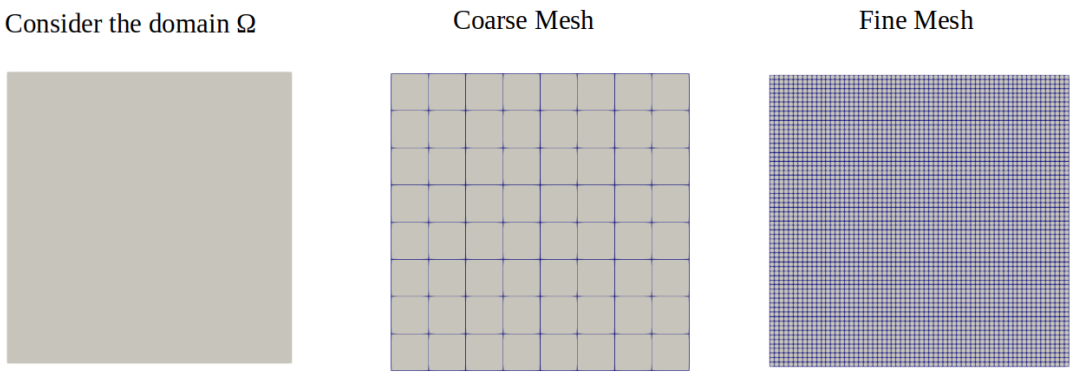


Figure 3.5: Mesh in Multiscale Finite Element Method.

$$-\nabla \cdot (a_\varepsilon(x)\nabla u) = f \quad \text{in } \Omega \tag{3.11}$$

where f is a given function and $a_\varepsilon(x)$ is the diffusion tensor. Additionally, $a_\varepsilon(x)$ is assumed to be positive definite and symmetric. There are multiple scales within $a_\varepsilon(x)$. The next step is to discuss some basic multiscale basis functions and global formulations.

3.5.1 Localization

To begin, discretize the global domain Ω into a coarse mesh. Let $\mathcal{T}_H$ represent the usual partition of Ω into finite elements (triangles, quadrilaterals, etc) and let N be the total number of vertices in $\mathcal{T}_H$. This partition is called the coarse grid. In the middle of the Figure (3.5) shows a partition $\mathcal{T}_H$ of size H. The finite element is a quadrilateral in two dimensions. As shown in Figure (3.6) we have a coarse grid and we denote K as the coarse block consist of four vertices on which the coarse-grid nodal values are calculated. On the partition $\mathcal{T}_H(K)$ on this coarse mesh there exist a regular mesh $\mathcal{T}_h$ that consists of fine elements for each fine block η, each with height and width h in Figure (3.6).

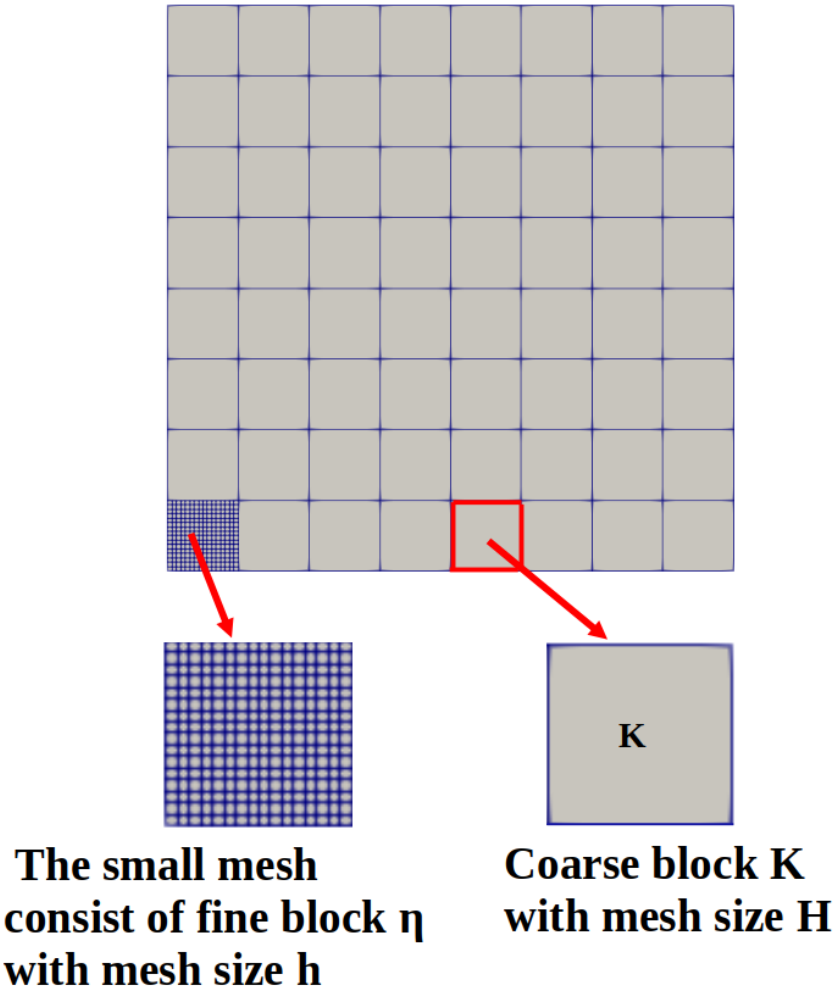


Figure 3.6: Coarse Mesh and fine mesh in Multiscale Finite Element Method.

3.5.2 Basis functions

Basis functions: Basis functions are constructed to capture the multiscale features of the solution. Let x_i be the interior nodes of the mesh $\mathcal{T}_H$ and φ_i^0 be the nodal basis of the standard finite element space $V_H = span\{\phi_i^0\}$ in Figure (3.6). Let us assume V_H to be consists of piecewise linear functions. Let us denote $\omega_i = supp(\varphi_i^0)$ and define φ_i with support in ω_i as following

$$-\nabla\left(a\left(\frac{x}{\varepsilon}\right)\nabla\varphi_i\right) = 0 \quad \text{in} \quad K, \varphi_i = \phi_i^0 \quad \text{on} \quad \partial K, \quad \forall K \in \mathcal{T}_H, \quad K \subset \omega_i \qquad (3.12)$$

The multiscale basis functions $\varphi_i, i = 1,2,...,N$, are solutions obtained as the result of solving the above local problems given by equation (3.12). The multiscale basis function coincides with standard finite element basis functions at the boundary of coarse elements

K, and are oscillatory in the interior of each coarse-grid block. The local problem (3.12) defined on element $K \subset \omega_i$ is totally independent of that defined on the adjacent element $K' \subset \omega_i$. As a result, local problems can be solved simultaneously on different elements (blocks like K) of $\mathcal{T}_H$. Therefore, it reduces computing time and memory requirements. This phase is called **offline phase**. The local subproblems (3.12) are given appropriate boundary conditions and solved on the fine scale $h < \varepsilon$ to determine the multiscale finite element. The finite-dimensional space $\mathcal{P}_H$ is spanned by φ_i.

$$\mathcal{P}_H = span\{\varphi_i\}$$

The multiscale modified basis function obtained as a result of (3.12) is shown in Figure (3.7) where one basis function is illustrated on coarse mesh $\mathcal{T}_H$. We see here the fine scale oscillation added to the solution in form of wiggles on the basis. Figure (3.8) shows all four basis functions on coarse mesh obtained by solving fine scale local problems.

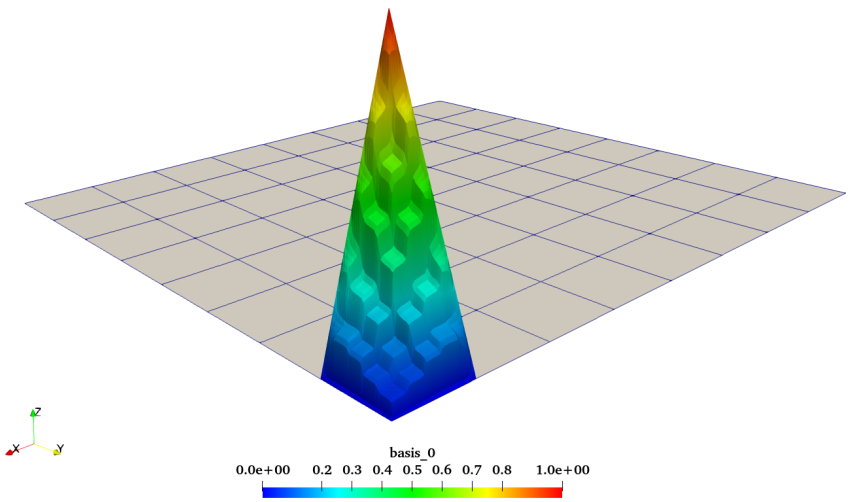


Figure 3.7: One modified Basis in 2 dimension.

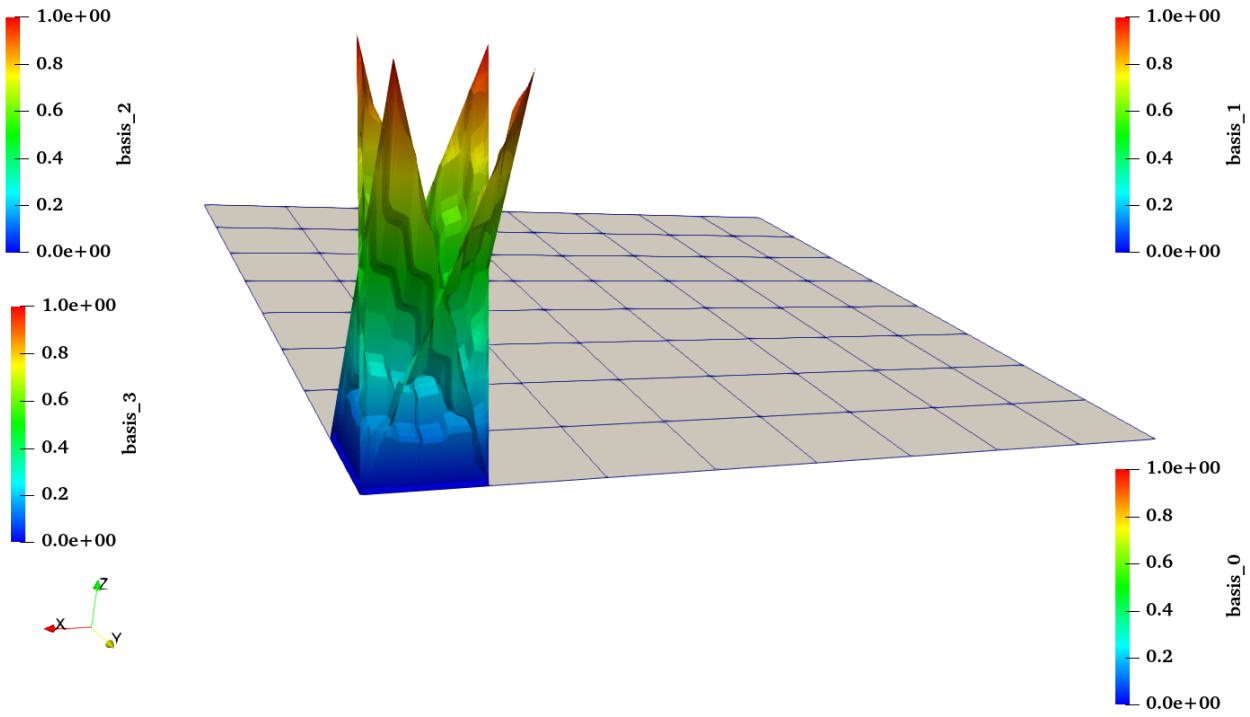


Figure 3.8: Four modified Basis in 2 dimension on coarse mesh.

3.5.3 Global coarse-grid problem

Global formulation: The next step is to discuss the global formulation of MsFEM. This is called **online phase**. Using multiscale basis functions to represent the fine-scale solution reduces the computation dimension.
The approximation u_H of the solution is :

$$u_H(x) = \sum_i u_H^i \varphi_i(x) \quad \forall x \in \Omega \tag{3.13}$$

with u_H^i the solution value at the nodal point x_i of the coarse mesh. When u_H is substituted into the fine-scale equation, the resulting system is projected onto the coarse-dimensional space to find u_H^i. To do this, multiply the resulting fine-scale equation with the coarse-scale test function.

In the case of Galerkin finite element methods, when the basis functions are conforming $(\mathcal{P}_H \subset H_0^1(\Omega))$, the coarse-scale problem is: find $u_H \in \mathcal{P}_H$ such that

$$\sum_{K \in \mathcal{T}_H} \int_K a_\varepsilon \nabla u_H \cdot \nabla v_H = \int_\Omega f v_H \quad \forall v_H \in \mathcal{P}_H \tag{3.14}$$

where v_H is the coarse-scale test functions. Therefore, the resulting linear equation system determines the solution on the coarse grid by determining the values of the solution at the

nodes of the coarse-grid block.

Substituting $u_H(x)$ (3.13) in the equation (3.14) leads to a linear system of equations involving nodal values u_H^i. The matrix system obtained is as follows:

$$Au_{nodal} = b \tag{3.15}$$

where

$$A = (A_{ij}), A_{ij} = \sum_{K \in \mathcal{T}_H} \int_K a_\varepsilon \nabla \varphi_i \cdot \nabla \varphi_j^0, \tag{3.16}$$

$u_{nodal} = (u_1, ..., u_i, ...)$ are the nodal values of the coarse-scale solution
$b = (b_i)$, $b_i = \int_\Omega f \varphi_i^0$
Boundary conditions are not discretized here. The stiffness matrix A is sparse, just like in standard finite element methods. In order to compute the stiffness matrix, we must compute the integrals for (A_{ij}) and (b_i). Calculating (A_{ij}) requires the evaluation of integrals on a fine grid. You can use a simple quadrature rule, for example, one point per fine grid cell. In this case, $\int_{K \in \mathcal{T}_H} a_\varepsilon \nabla \varphi_i \cdot \nabla \varphi_j^0 \approx \sum_{\eta \in \mathcal{T}_h \subset K \in \mathcal{T}_H} (a_\varepsilon \nabla \varphi_i)|_\eta \cdot \nabla \varphi_j^0$ where η is a fine grid block and $(a_\varepsilon \nabla \varphi_i)|_\eta$ is the value within a fine grid block η. We can reuse the pre-computed matrix A when the source term changes, but the matrix b must be recomputed.

A global formulation can generally be modified easily and various formulations based on finite volume, mixed finite element, discontinuous Galerkin finite element, and other methods can be derived.

Error Estimates in MsFEM

Theorem: Let $u^\varepsilon \in H^2(\Omega)$ solve the model problem and $u_H^\varepsilon \in P_H$ be the computed solution by the multiscale FEM.
Then if $H < \varepsilon$

$$||u^\varepsilon - u_H^\varepsilon||_{H^1(\Omega)} \leq CH(|u|_{H^2(\Omega)} + ||f||_{L^2(\Omega)}$$

If $H > \varepsilon$ and $u^0 \in H^2 \cap W^{1,\infty}$ is the solution to the homogenized problem then

$$||u^\varepsilon - u_H^\varepsilon||_{H^1(\Omega)} \leq C(H + \varepsilon)||f||_{L^2(\Omega)} + C\left(\sqrt{\frac{\varepsilon}{H}}\right)||u^0||_{W^{1,\infty}(\Omega)}$$

from [[44], Ch 6].

Following are the Algorithm for MsFEM.

Algorithm 2 Algorithm for Multiscale Finite Element Method.

1. Setup mesh (coarse).
2. Setup system and constraints.
3. For each coarse grid block K
   - For each vertex i
   - Solve equation (3.12) φ_i
   - End for.

   End do
4. Assemble system and right hand side for coarse mesh
5. Solve for coarse problem u

3.5.4 Assembly of stiffness matrix.

In order to assemble the stiffness matrix, fine-scale basis functions can be used to represent multiscale basis functions. In code development, this is particularly useful. Let's suppose that multiscale basis function (in discrete form) φ_i can be written as

$$\varphi_i = d_{ij}\varphi_j^{0,f}$$

where $D = (d_{ij})$ is a matrix and $\varphi_j^{0,f}$ are fine-scale finite element basis functions (e.g., piecewise linear functions). The ith row of this matrix contains the fine-scale representation of the ith multiscale basis function. Substitute this expression into the formula for the stiffness matrix d_{ij} in equation (3.13), we have

$$(A_{ij}) = \int_\Omega a_\varepsilon \nabla\varphi_i \cdot \nabla\varphi_j dx = d_{il}\int_\Omega a_\varepsilon \nabla\varphi_l^{0,f} \cdot \nabla\varphi_m^{0,f} dx\, d_{jm}.$$

The stiffness matrix for the fine-scale problem is denoted by $A^f = (a_{lm}^f)$, $a_{lm}^f = \int_\Omega a_\varepsilon \nabla\varphi_l^{0,f} \cdot \nabla\varphi_m^{0,f} dx$, we have

$$A = DA^f D^T$$

Similarly, for the right-hand side, we have $b = \int_\Omega \varphi_i dx = Db^f$, where $b^f = (b_i^f)$, $b_i^f = \int_\Omega f\varphi_i^{0,f} dx$. This is used in the assembly of the stiffness matrix.

One-dimensional example.

A one-dimensional stiffness matrix and basis functions can almost explicitly be computed (see 3.15). Let's consider a simple example

$$-(a_\varepsilon(x)u')' = f \tag{3.17}$$

$u(0) = u(1) = 0$, where refers to the spatial derivative. We assume that the interval $[0,1]$ is divided into N segments $0 = x_0 < x_1 < x_2 < .... < x_i < x_{i+1} < ... < x_N = 1$. The multiscale basis function for the node i is given by

$$(a_\varepsilon(x)\varphi_i')' = 0$$

with the support in $[x_{i-1}, x_{i+1}]$. The boundary conditions for the basis function φ_i are defined as $\varphi_i(x_{i-1}) = 0, \quad \varphi_i(x_i) = 1$ in the interval $[x_{i-1}, x_i]$. The boundary conditions for the basis function φ_i are defined as $\varphi_i(x_i) = 1, \quad \varphi_i(x_{i+1}) = 0$ in the interval $[x_i, x_{i+1}]$. We compute $a_\varepsilon\varphi_i'$. It is seen from Equation (3.17) that $a_\varepsilon\varphi_i'$ is constant with different constant in interval $[x_{i-1}, x_i]$ and $[x_i, x_{i+1}]$.
For the interval $[x_{i-1}, x_i]$ the constant can be computed as $\varphi_i = const/a_\varepsilon(x)$ and then integrating over the interval we get

$$a_\varepsilon(x)\varphi_i' = \frac{1}{\int_{x_{i-1}}^{x_i} \frac{dx}{a_\varepsilon(x)}}$$

and for $[x_i, x_{i+1}]$.

$$a_\varepsilon(x)\varphi_i' = -\frac{1}{\int_{x_i}^{x_{i+1}} \frac{dx}{a_\varepsilon(x)}}$$

The stiffness matrix A (3.16) is given by

$$\begin{aligned} a_{ij} &= \int_{x_{i-1}}^{x_i} a_\varepsilon\varphi_i'(\varphi_j^0)'dx + \int_{x_i}^{x_{i+1}} a_\varepsilon\varphi_i'(\varphi_j^0)'dx \\ &= \frac{1}{\int_{x_{i-1}}^{x_i} \frac{dx}{a_\varepsilon(x)}} \int_{x_i}^{x_i} (\varphi_j^0)'dx - \frac{1}{\int_{x_i}^{x_{i+1}} \frac{dx}{a_\varepsilon(x)}} \int_{x_i}^{x_{i+1}} (\varphi_j^0)'dx \end{aligned}$$

Taking the values $\int_{x_{i-1}}^{x_i} (\varphi_{i-1}^0)'dx = -1$, $\int_{x_{i-1}}^{x_i} (\varphi_i^0)'dx = 1$, $\int_{x_i}^{x_{i+1}} (\varphi_{i+1}^0)'dx = -1$, $\int_{x_i}^{x_{i+1}} (\varphi_{i+1}^0)'dx = 1$, we have

$$a_{i,i-1} = -\frac{1}{\int_{x_{i-1}}^{x_i} \frac{dx}{a_\varepsilon(x)}}, \quad a_{ii} = \frac{1}{\int_{x_{i-1}}^{x_i} \frac{dx}{a_\varepsilon(x)}} + \frac{1}{\int_{x_i}^{x_{i+1}} \frac{dx}{a_\varepsilon(x)}}, \quad a_{i,i+1} = -\frac{1}{\int_{x_i}^{x_{i+1}} \frac{dx}{a_\varepsilon(x)}}$$

As result we get the stiffness matrix has a tridiagonal form and the linear system is (3.15), where $b_i = \int_0^1 f\varphi_i^0 dx$.

Figure (3.9) and (3.10) illustrates the solution and a few multiscale basis functions.

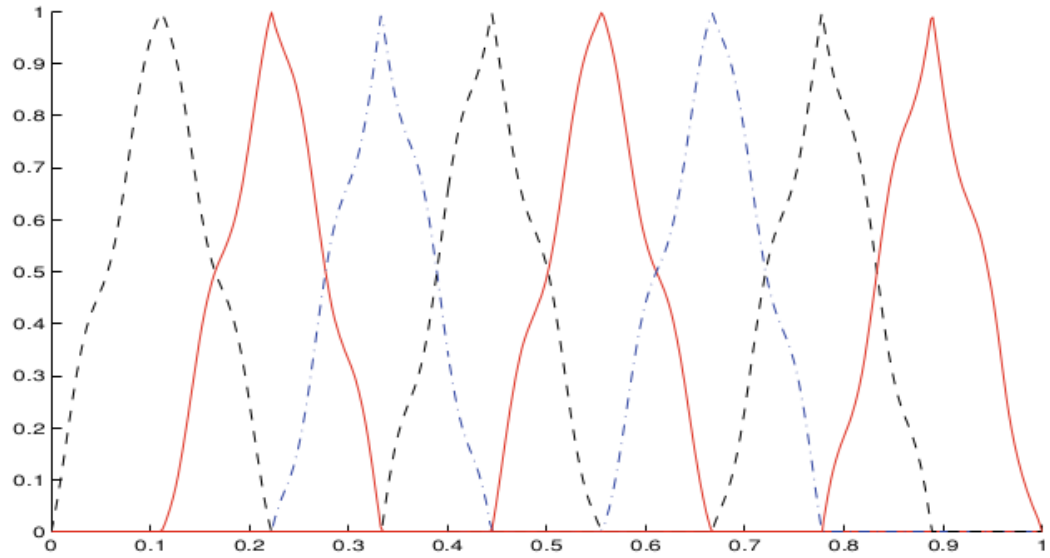


Figure 3.9: One-dimensional multiscale basis function. Figure courtesy of [12].

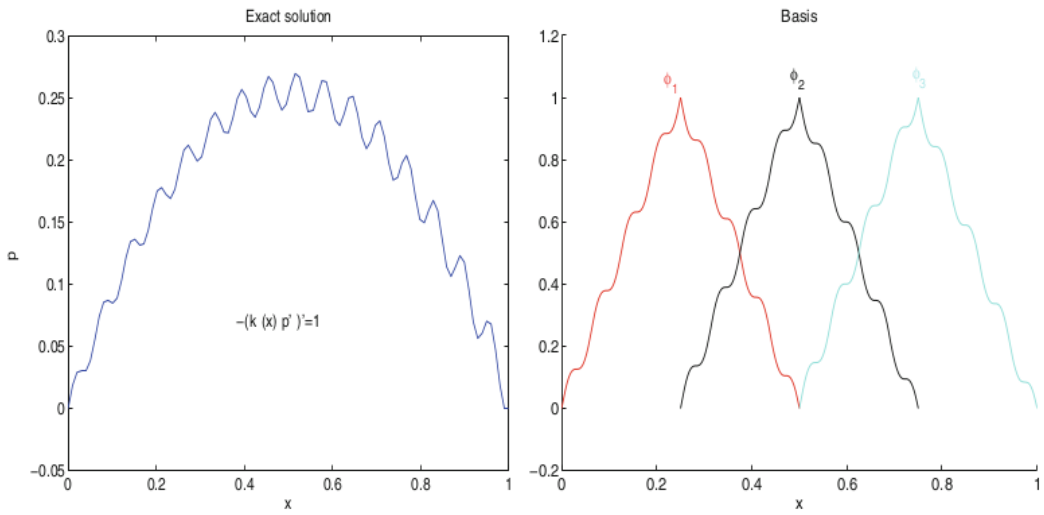


Figure 3.10: One-dimensional basis functions and the solution. Figure courtesy of [44].

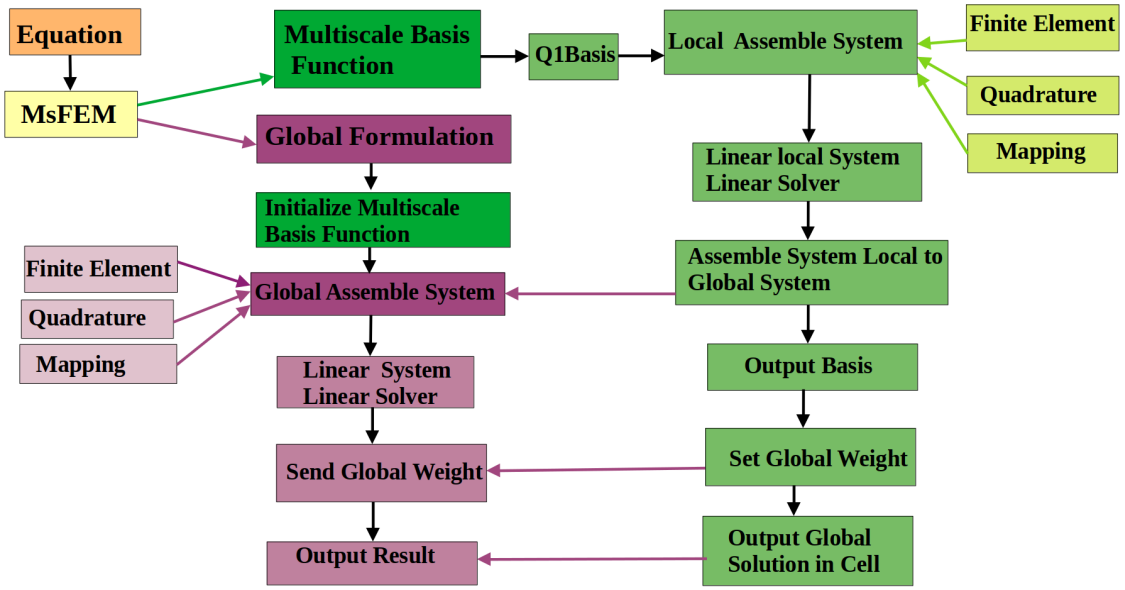


Figure 3.11: Multiscale Finite Element Method Flow Chart.

In Figure (3.11) is a Flow chart illustrating the implementation of the multiscale Finite Element Method. It is apparent from the purple coloration of the global formulation that it is the same as the standard finite element code in Chapter 2, while the green coloration denotes the Multiscale Basis Function solving local problems shown in section (3.5.2), and the Assemble system local to global is exactly what's explained in (3.5.4). The reader is then required to read Chapter 5 and view the whole code in the Github `https://github.com/heena008/Diffusion-Equation-with-MsFEM.io`.

Basis in 2 dimension

The Figures (3.12 and [3.13) shows the basis in 2 dimesion is obtained from solving stationary diffusion equation using the Multiscale finite element method. In Figure (3.14) we compare the basis of standard Finite element method and Multiscale finite element method we see that for Standard Finite element basis does not capture fine scale feature for which one needs very high resolution. The basis of the multiscale finite element method captures fine scale features and upscales them using a modified basis function.

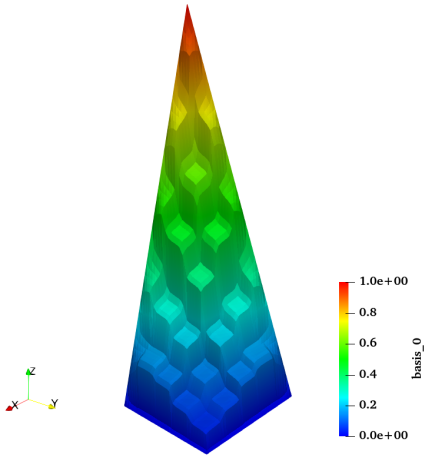


Figure 3.12: Basis in 2 dimension for Multiscale finite element method.

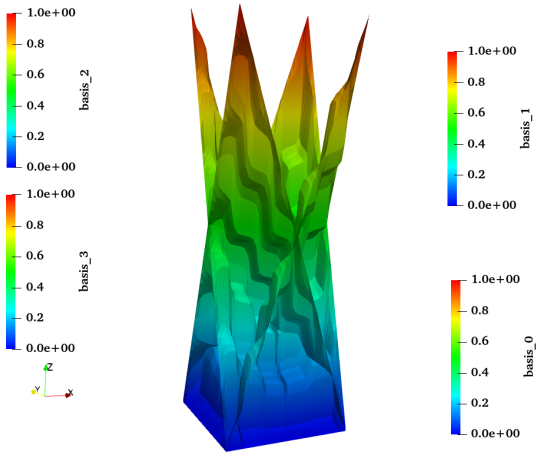


Figure 3.13: All four basis of two-dimensional basis functions for Multiscale finite element method.

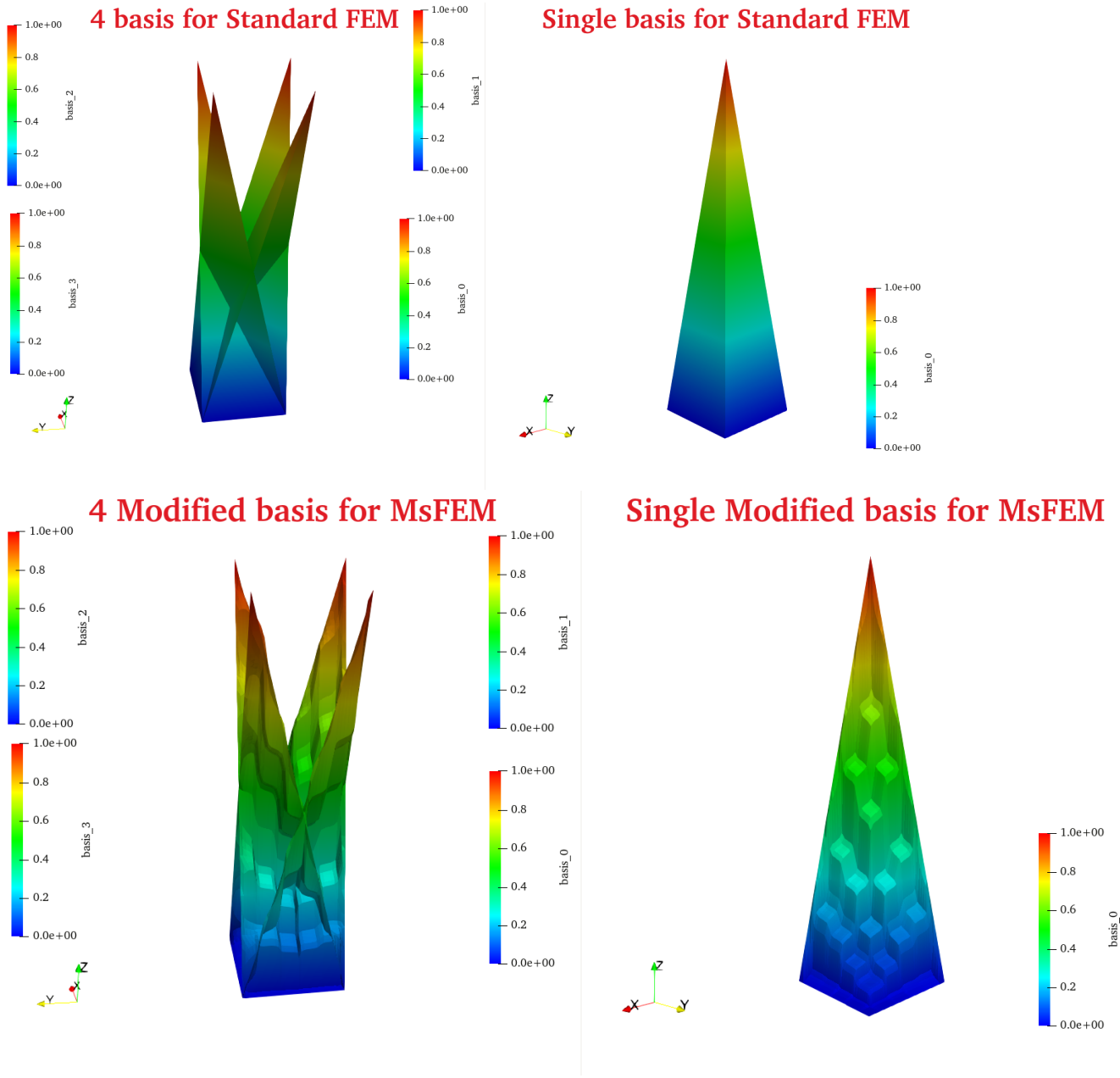


Figure 3.14: Two dimensional basis functions for Finite Element Method and Multiscale Finite Element Method.

Here we conclude Chapter 3 and move towards an advection-diffusion multiscale finite element solution in the next chapter.

4 Advection-Diffusion Equation

"It is more important to have beauty in one's equations than to have them fit experiment."

- Paul Dirac

4.1 Semi-Lagrangian Multiscale Finite Element

In 2 and 3 dimensions, consider the following equation for advection and diffusion.

$$\begin{aligned} \partial_t u + c_\delta \cdot \nabla u = \nabla \cdot (a_\varepsilon \nabla u) + f \quad & \text{in } \mathbb{T}^d \times [0,T] \sim [0,1]^d \times [0,T] \\ u(x,0) = u_0(x) & \end{aligned} \tag{4.1}$$

and

$$\begin{aligned} \partial_t u + \nabla \cdot (c_\delta u) = \nabla \cdot (a_\varepsilon \nabla u) + f \quad & \text{in } \mathbb{T}^d [0,1]^d \times [0,T] \\ u(x,0) = u_0(x) & \end{aligned} \tag{4.2}$$

- where velocity is given by c_δ .
- This diffusion matrix $a_\varepsilon(x,t)$ represents the canopy in our case in a positive definite way (on a uniform ε scale and point-wise in x).
- (4.1) does not conserve the tracer u if $f = 0$, but (4.2) conserves it if $f = 0$.
- u_0 is the initial condition
- $\mathbb{T}^d$ is the d-dimensional torus which is topologically equivalent to $[0,1]^d$.
- Bold letters for the vectors and tensors independent of the dimension.
- Indices $\delta > 0$ and $\varepsilon > 0$ represents large variations on small scales that are not resolved on coarse scales $H > 0$ of the multiscale method.
- $h \ll H$ is the local scale that can resolve the variations in the coefficients.

Further details about the assumption and mathematical analysis where done in work [39, 40]

A standard MsFEM method works for elliptic and parabolic stationary problems. Figure (4.1) illustrates how high advection creates artificial boundary layers, since in

multiscale finite elements flow blocks at coarse cell boundaries, resulting in blocked flow information.

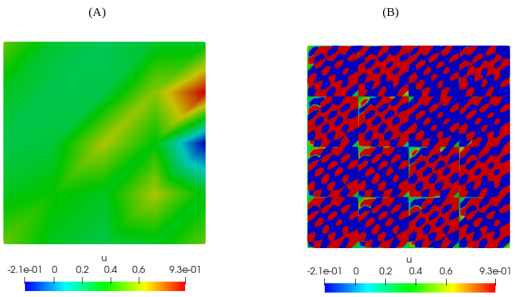


Figure 4.1: Solution (A) is for FEM and (B) is for MsFEM.

A Lagrangian framework was proposed in [39] to overcome this problem. Higher dimensions and high velocity do not work with it. Since the semi-Lagrangian method is unconditionally stable, it is used in [39] which would be used in our problem.

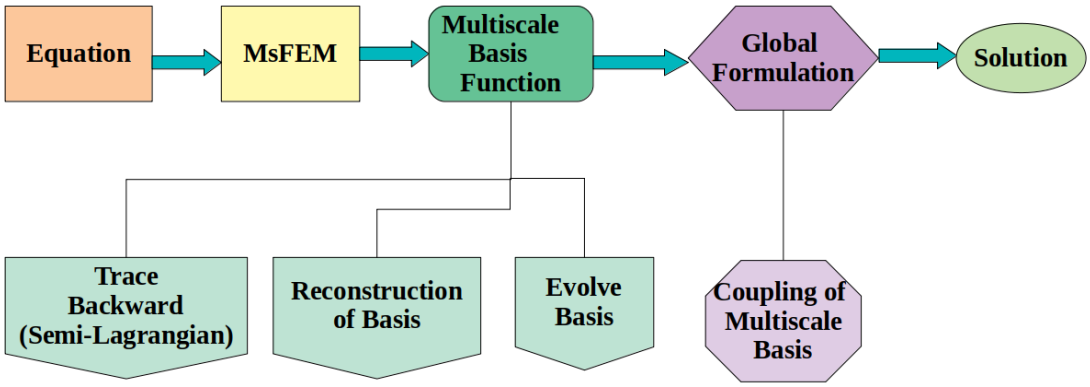


Figure 4.2: Semi-Lagrangian algorithm for high advection.

Figure (4.2) is the flow chart for the high advection problem. It is an extension of the flow chart for the elliptic problem presented in Chapter 3.

Weak formulation is obtained by multiply equation (4.1) by φ^{ms} on both side and integrate over $\mathbb{T}_d$ we get

$$\int_{\mathbb{T}^d} \varphi^{ms} \cdot \partial_t u \, dx + \int_{\mathbb{T}^d} c_\delta \varphi^{ms} \cdot \nabla u \, dx = \int_{\mathbb{T}^d} \nabla \varphi^{ms} \cdot (a_\varepsilon \nabla u) \, dx + \int_{\mathbb{T}^d} \varphi^{ms} \cdot f \, dx \qquad (4.3)$$

The global time step in 2D/3D.

With multiscale basis functions in conformal finite element settings, we approximate the global solution at each time step using a spatially coarse subspace $V^H(t) \subset H^1(\mathbb{T}^d)$ in

which the solution u is sought (almost everywhere). The definition of a finite-dimensional space is as follows:

$$V^H(t) = span\{\varphi_j^{H,ms}(.,t)|\ j = 1,.....,N_H\} \tag{4.4}$$

Let the solution $u^H(x,t)$ in terms of the basis at time $t \in [0,T]$ as follows

$$u^H(x,t) = \sum_{j=0}^{N_H} u_j^H(t)\varphi_j^{H,ms}(x,t) \tag{4.5}$$

Replacing the above equation in equation (4.3) with φ_i^{ms} basis function we get

$$\begin{aligned}
&\therefore \int_{\mathbb{T}^d} \varphi_i^{H,ms}(x,t)\cdot\partial_t\left(\sum_{j=0}^{N_H} u_j^H(t)\varphi_j^{H,ms}(x,t)\right)dx \\
&+\int_{\mathbb{T}^d} c_\delta(x,t)\varphi_i^{H,ms}(x,t)\cdot\nabla\left(\sum_{j=0}^{N_H} u_j^H(t)\varphi_j^{H,ms}(x,t)\right)dx \\
&=\int_{\mathbb{T}^d} \nabla\varphi_i^{H,ms}(x,t)\cdot a_\varepsilon(x,t)\nabla\left(\sum_{j=0}^{N_H} u_j^H(t)\varphi_j^{H,ms}(x,t)\right)dx+\int_{\mathbb{T}^d}\varphi_i^{H,ms}(x,t)\cdot f\,dx
\end{aligned} \tag{4.6}$$

Now use chain rule in first term

$$\begin{aligned}
&\therefore \frac{d}{dt}\sum_{j=0}^{N_H}\int_{\mathbb{T}^d}\varphi_i^{H,ms}(x,t)\cdot u_j^H(t)\varphi_j^{H,ms}(x,t)\,dx+\sum_{j=0}^{N_H}u_j^H(t)\int_{\mathbb{T}^d}\varphi_i^{H,ms}(x,t)\cdot\partial_t\varphi_j^{H,ms}(x,t)\,dx \\
&=\sum_{j=0}^{N_H}u_j^H(t)\int_{\mathbb{T}^d}\nabla\varphi_i^{H,ms}(x,t)\cdot a_\varepsilon(x,t)\nabla\varphi_j^{H,ms}(x,t)\,dx \\
&-\sum_{j=0}^{N_H}u_j^H(t)\int_{\mathbb{T}^d}c_\delta(x,t)\varphi_i^{H,ms}(x,t)\cdot\nabla\varphi_j^{H,ms}(x,t)\,dx+\int_{\mathbb{T}^d}\varphi_i^{H,ms}(x,t)\cdot f\,dx
\end{aligned} \tag{4.7}$$

Rearrange the terms

$$\begin{aligned}
&\therefore \sum_{j=0}^{N_H}\frac{d}{dt}u_j^H(t)\left(\int_{\mathbb{T}^d}\varphi_i^{H,ms}(x,t)\cdot\varphi_j^{H,ms}(x,t)\,dx\right)+\sum_{j=0}^{N_H}u_j^H(t)\left(\int_{\mathbb{T}^d}\varphi_i^{H,ms}(x,t)\cdot\partial_t\varphi_j^{H,ms}(x,t)\,dx\right) \\
&=\sum_{j=0}^{N_H}u_j^H(t)\int_{\mathbb{T}^d}\nabla\varphi_i^{H,ms}(x,t)\cdot a_\varepsilon(x,t)\nabla\varphi_j^{H,ms}(x,t)\,dx \\
&-\sum_{j=0}^{N_H}\int_{\mathbb{T}^d}c_\delta(x,t)\varphi_i^{H,ms}(x,t)\cdot\nabla\varphi_j^{H,ms}(x,t)\,dx+\int_{\mathbb{T}^d}\varphi_i^{H,ms}(x,t)\cdot f\,dx
\end{aligned} \tag{4.8}$$

$$\begin{aligned} M(t)\frac{d}{dt}u^H(t) + N(t)u^H(t) &= A(t)u^H(t) + f^H(t) \\ u^H(0) &= u^{H,0} \end{aligned} \tag{4.9}$$

where

$$\begin{aligned} A_{ij}(t) = &\int_{\mathbb{T}^d} \nabla\varphi_i^{H,ms}(x,t)\cdot a_\varepsilon(x,t)\nabla\varphi_j^{H,ms}(x,t)\,dx \\ &- \int_{\mathbb{T}^d} c_\delta(x,t)\varphi_i^{H,ms}(x,t)\cdot\nabla\varphi_j^{H,ms}(x,t)\,dx \end{aligned}$$

The mass matrix is given by

$$M_{ij}(t) = \int_{T^d} \varphi_i^{H,ms}(x,t)\cdot\varphi_j^{H,ms}(x,t)\,dx$$

$f^H(t)$ contains forcing and boundary conditions and the initial condition $u^{H,0}$ is the projection of $u^0 \in L^2(\mathbb{T}^d)$ onto $V^H(0)$. Note that equation (4.9) contains a derivative of the mass matrix:

$$N_{ij}(t) = \int_{\mathbb{T}^d} \varphi_i^{H,ms}(x,t)\cdot\partial_t\varphi_j^{H,ms}(x,t)\,dx$$

Discretization is first done in space then time as basis are time dependent. The implicit Euler method is used for time discretization.

$$\begin{aligned} \frac{M(t^{n+1})u^{n+1} - M(t^n)u^n}{\Delta t} + [(1-\theta)N(t^n)u^n(t^n) + \theta N(t^{n+1})u^{n+1}(t^{n+1})] = [(1-\theta)A(t^n)u^n(t^n) \\ + \theta A(t^{n+1})u^{n+1}(t^{n+1})] + [(1-\theta)f^n(t^n) + \theta f^{n+1}(t^{n+1})] \end{aligned} \tag{4.10}$$

For $\theta = 1$ we have implicit Euler Scheme.

$$\frac{M(t^{n+1})u^{n+1} - M(t^n)u^n}{\Delta t} + [N(t^{n+1})u^{n+1}(t^{n+1})] = [A(t^{n+1})u^{n+1}(t^{n+1})] + [f^{n+1}(t^{n+1})] \tag{4.11}$$

$$M(t^{n+1})u^{n+1} = M(t^n)u^n + \Delta t[A(t^{n+1})u^{n+1}(t^{n+1}) - N(t^{n+1})u^{n+1}(t^{n+1}) + f^{n+1}(t^{n+1})] \tag{4.12}$$

For $\theta = 0$ we have explicit Euler Scheme.

$$\frac{M(t^{n+1})u^{n+1} - M(t^n)u^n}{\Delta t} + [N(t^n)u^n(t^n)] = [A(t^n)u^n(t^n)] + [f^n(t^n)] \tag{4.13}$$

$$M(t^{n+1})u^{n+1} = M(t^n)u^n + \Delta t[A(t^n)u^n(t^n) - N(t^n)u^n(t^n) + f^n(t^n)] \tag{4.14}$$

For $\theta = 0.5$ we have Crank-Nicolson.

$$\frac{M(t^{n+1})u^{n+1} - M(t^n)u^n}{\Delta t} + 0.5[N(t^n)u^n(t^n) + N(t^{n+1})u^{n+1}(t^{n+1})] = 0.5[A(t^n)u^n(t^n) + A(t^{n+1})u^{n+1}(t^{n+1})] + 0.5[f^n(t^n) + f^{n+1}(t^{n+1})] \tag{4.15}$$

$$M(t^{n+1})u^{n+1} = M(t^n)u^n + 0.5\Delta t[A(t^n)u^n(t^n) + A(t^{n+1})u^{n+1}(t^{n+1}) - N(t^n)u^n(t^n) + N(t^{n+1})u^{n+1}(t^{n+1}) + f^n(t^n) + f^{n+1}(t^{n+1})] \tag{4.16}$$

4.2 The Reconstruction Mesh

Trace back

From time t_{n+1}, trace back an Eulerian cell $K \in \mathcal{T}_H$ where the basis and solution are unknown to t_n. We know the solution u_n on this distorted cell, but we don't know the multiscale basis $\tilde{\varphi}_i, i = 1,2$. Backward Euler method is used to solve ordinary differential equations [40].

- Trace back the local cell $t_n \to t_{n-1}$ (parallel).

- This gives an initial value problem.

$$\frac{d}{dt}\hat{x}_l(t) = -c_\delta(\hat{x}_l(t), -t), t \in [-t_{n+1}, -t_n]$$

$$\hat{x}_l(-t_{n+1}) = x_l$$

- It is solved by backward Euler.

for each x_l and then take $\tilde{x}_l = \tilde{x}_l(-t_n)$, see Figure (4.3) for an illustration. The process is then parallelized.

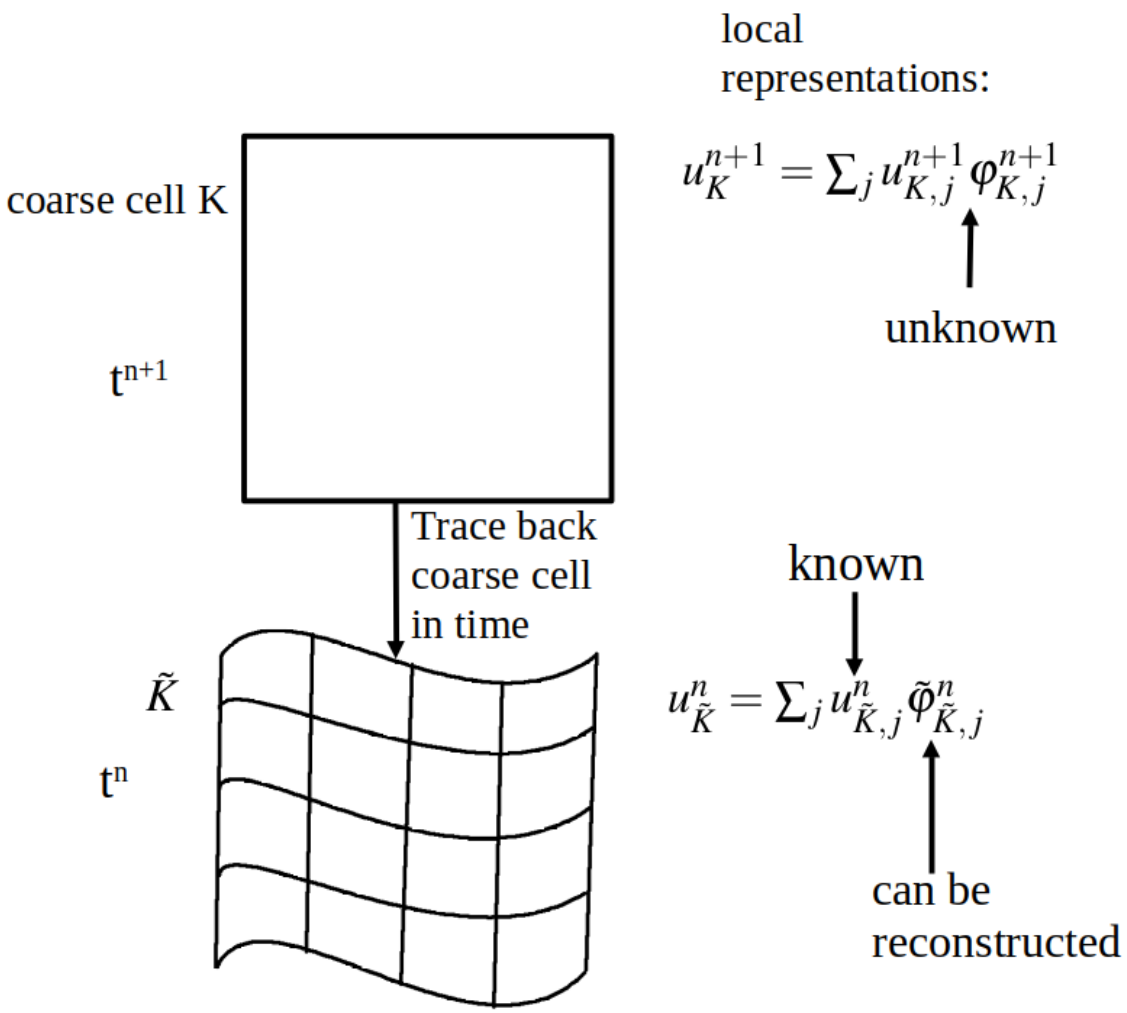


Figure 4.3: Eulerian coarse cell with its fine mesh is traced back with one time step where global solution is taken to reconstruct a basis.

Basis Reconstruction

You must compute the solution u_n on $\tilde{K}$ by tracing the point x_l of $K \in \mathcal{T}_H$ back to its origin $\tilde{x}_l$ in the coarse distorted cell $\tilde{K}$. A solution is reconstructed from previous time step t_n by looping all coarse edges of distorted cell.

Evolve Basis

Now reconstructed basis on each coarse cell obtained from previous time step $\tilde{K}$ is a conformal basis does not belong on the coarse Eulerian grid $\mathcal{T}_H$ that we initially fixed. Now we solve evolution problem on $\tilde{K}$ to get the i-th basis at t_{n+1} on $\mathcal{T}_H$.

$$\begin{aligned} \frac{d}{dt}\varphi_{K,i} + (\tilde{\nabla}\cdot\tilde{c}_\delta)\varphi_{\Gamma,i} &= \tilde{\nabla}\cdot(\tilde{a_\varepsilon}\tilde{\nabla}\varphi_{K,i}) \quad in \quad \tilde{K}\times[t_n,t_{n+1}] \\ \varphi_{K,i}(.,t)|_{\tilde{\Gamma}_l} &= \tilde{\varphi}_{\tilde{\Gamma}_l,i}(.,t), \quad t\in[t_n,t_{n+1}] \\ \varphi_{K,i}(\tilde{x},t^n) &= \tilde{\varphi}_{\tilde{K},i}(\tilde{x}) \end{aligned} \tag{4.17}$$

Local reconstruction step is semi-Lagrangian and the global step is completely Eulerian. The final step $\varphi_{K,i}(\tilde{x},t^{n+1})$ on $\tilde{K}$ can be again be mapped onto the Eulerian element $K\in\mathcal{T}_H$ to obtain the desired basis function $\varphi_{K,i}(\tilde{x},t^{n+1}) \sim \varphi_{K,i}^{n+1}(x)$ at the next time step. We conclude Chapter 4 with all of the mathematical theory we need for the next chapters.

5 Software Concepts

"The invisible pieces of code that form the gears and cogs of the modern machine age, algorithms have given the world everything from social media feeds to search engines and satellite navigation to music recommendation systems."

- Hannah Fry

5.1 Introduction

Mathematical models in science and engineering are solved with scientific computing. A number of libraries have been developed to solve partial differential equations. In the last few decades, many finite element-based libraries have been developed to solve complex problems using robust algorithms that require minimal code while still allowing for maximum mathematical control. There are several open-source libraries, including DUNE [13], GetDp [14], GetFEM [31], FreeFEM [18], FENICS Project [25, 24], Sundance (C++) [27], Analysa (C++) [8] and Feel++ (C++) [29, 30]. They either rely on a domain-specific language (Python, the freefem language, etc.) to describe PDEs, or they are geometry-dependent, or they do not express mathematics well, and use programming details to hide mathematics. Readers unfamiliar with C++ concepts should read Appendix Section IV.

5.2 The deal.II workflow

In deal.II, one can create fully functional finite element models using C++.

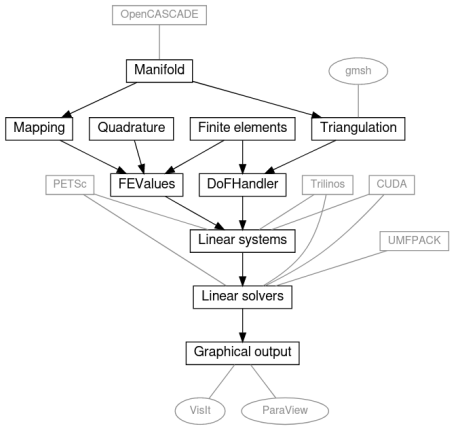


Figure 5.1: Finite Element setup in deal.II. Figure courtesy deal.II website [7].

A C++ library aimed at solving partial differential equations is used in the present study, deal.II, the Differential Equations Analysis Library. There are a number of applications that can benefit from the powerful high-performance computing features of this advanced mathematical library. Besides having control over the mathematical implementation, different preconditioned solvers are available, and detailed documentation is available. As shown in Figure (5.1) here are the main components of the FEM you need to follow.

- Triangulation: Geometry creation and mesh generation are performed by triangulation.
- DoFHandler : The module assigns degrees of freedom to each triangulation cell according to the finite element space described by a finite element object.
- Finite elements: It allows access to shape functions and includes all shape functions functionality.
- Quadrature: It provides quadrature point locations for the unit cell $[0, 1]^d$, d being dimension 1, 2, or 3 as well as the quadrature point's weight.
- Mapping : This function maps a point between a unit cell and a physical cell.
- FEValues: It is used for assembling matrices or vectors. It connects elements, quadrature, and mappings.
- Linear System: The classes in this module deal with linear algebra, i.e., matrices, vectors, and linear systems
- Linear Solver: This item contains linear solvers with iterative, direct, and eigenvalue algorithms.
- Output: The output is divided into three types. Besides creating meshes, it can also output matrices in a graphical format, which will be used to visualize data.

Code folders workflow

In the advection diffusion folder as shown in Figure (5.2) , there are four folders out of which two main subfolders for code, namely:

1. The **include** folder contains files with a .hpp format, which contains various header files and code that can be used by files in another folder, titled source.
    - Predefined header files define functions that you can include with a preprocessor directive *#include* followed by < >. As an example, the <iostream> predefined header file in C++ defines input-output functions.
    - A user-defined header file is one that is created by the user and is included by

using *#include* in the program followed by "".

2. The **source** folder contains files of the format .cc, which are source code files containing C++ programs.
3. The remaining two folders namely **doc** and **documentation** are used for generating documentation for the code. There is also README.md file for the instructions of the code, and a LICENSE file for the software license that tells others what they can and cannot do with your source code.

It also contains a CMakeLists.txt file for compiling the code, which means all the header files from the **include** folder are compiled with the .cc files in the **source** folder and an object file is created (the code is transformed into machine language). Object files are then run and the output is converted from machine language to program output.

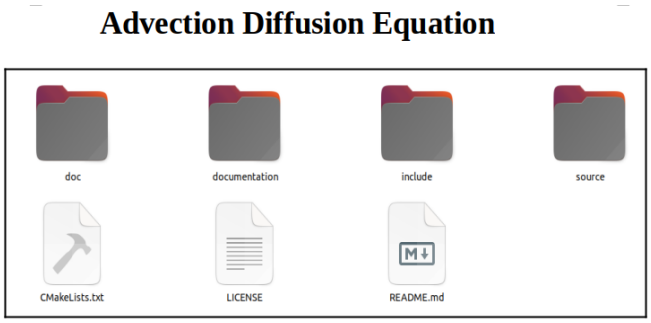


Figure 5.2: Main folder of code with this sub-folders.

The **source** folder contains two files as shown in Figure (5.3) . One is the main code file named **AdvectionDiffusionProblem.cc** and the other is the (**CMAKELists.txt**) file.

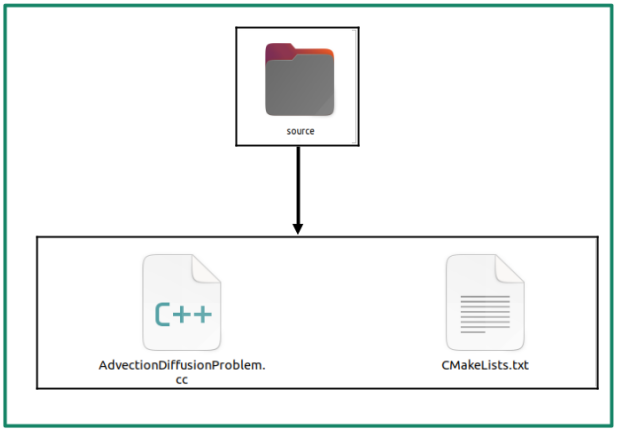


Figure 5.3: Sub-folder source that constant C++ files.

Some of the header files in the **include** folder as shown in Figure (5.4) will be discussed in more detail in the later sections. In deal.II, the advection diffusion equation is solved

using the standard finite method and the multiscale finite method.

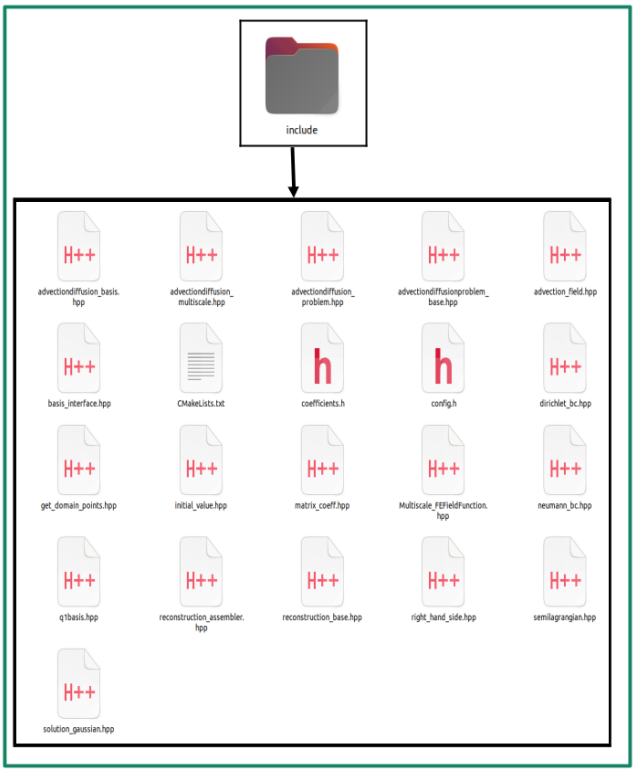


Figure 5.4: Sub-folder include that constant header files.

Now let understand the file **AdvectionDiffusionProblem.cc** code.

```
1  // User defind Headers
2
3  // File that contains code for Multiscale finite element method implementation
4  #include "advectiondiffusion_multiscale.hpp"
5  // File that contains code for Finite element method implementation
6  #include "advectiondiffusion_problem.hpp"
7  //-------------------------------------------------------------//
8  //C++ Predefined Headers
9  //File include standard library in C++
10 #include <cstdlib>
11 //In C++, fstreams is a library that creates files, writes to files, and reads data from files.
12 #include <fstream>
13 //In C++, iostream is a library that declares objects that control reading from and writing to the standard streams.
14 #include <iostream>
15 //-------------------------------------------------------------//
16 int main()
17 {
18    bool is_periodic = false;
19    unsigned int n_refine = 3;
20    const int dim =2;
21
22    // Solution for Low resolution Standard Finite Element Method
23    Timedependent_AdvectionDiffusionProblem::AdvectionDiffusionProblem<dim>
24    advectiondiffusion_problem_2d_coarse(n_refine, is_periodic);
```

```
25    advectiondiffusion_problem_2d_coarse.run();
26
27    // Solution for High resolution Standard Finite Element Method
28    Timedependent_AdvectionDiffusionProblem::AdvectionDiffusionProblem<dim> advectiondiffusion_problem(7, is_periodic);
29    advectiondiffusion_problem.run ();
30
31    // Solution for Low resolution Multiscale Finite Element Method
32    using ReconstructionType = Timedependent_AdvectionDiffusionProblem::BasicReconstructor<dim>;
33
34    using BasisType = Timedependent_AdvectionDiffusionProblem::SemiLagrangeBasis<dim, ReconstructionType>;
35
36    Timedependent_AdvectionDiffusionProblem::AdvectionDiffusionProblemMultiscale<dim, BasisType>
37    advectiondiffusion_ms_problem_2d(n_refine, is_periodic);
38    advectiondiffusion_ms_problem_2d.run();
39
40    return 0;
41  }
```

Listing 1: The advection-diffusion **AdvectionDiffusionProblem.cc** source file.

Let us now understand the code in Listing (1).

- The green sentences in the code are comments that start with // and /* */ . This helps us understand what the following line does in the code.
- In code, the violet represents variables or parameters in C++.
- The red is the user-defined header file.
- The blue lines in the code are C++ objects and functions in deal.II.
- Line 4 and 6 in the code represent adding user-defined header files to the code. There are two header files that contain implementation code for multiscale and standard finite element method solutions. Lines 1, 3 and 5 are comments regarding lines 4 and 6.
- Lines 9 to 14 are needed to include C++ header files. Here, the green lines represent the purpose of lines 10, 12 and 14.
- A program's start is always called **main** in Line 16. Line 20 is for the periodic boundary condition set to false, i.e. it is not periodic. The refinement level is indicated by line 21. Here we have chosen 3 for coarse mesh. Line 22 is for code dimension. Here it is set to 2.
- Line 24 to 27 is for Standard finite element coarse mesh here Namespace (refer Appendix IV [5]) Timedependent_AdvectionDiffusionProblem follows with class (refers Appendix IV [2]) AdvectionDiffusionProblem, which has dimensions to be considered, followed by object advectiondiffusion_problem which takes into account dimension and periodic boundary condition. Then at line 27 **run** function is used on object to get the program result.

- Line 29 to 32 does the same thing, but with a finer resolution mesh, since 7 represents how many refinements the code takes.
- The lines 38 to 42 are for the multiscale finite element method, while the lines 35 to 36 are for the first basis reconstruction type. Basically, it is the same way to call the function and object as above, except the object runs multiscale finite element code.

Remark: From now the deal.II related terms will be colored as deal.II and C++ related terms will be colored as C++. Also, the stationary diffusion equation solution discussed in Chapter 1 and Chapter 2 will follow the same implementation style except the time-dependent term and the advection term are zero. The workflow folder style for the diffusion equation solution is also similar to above.

Algorithm (3) is used to solve the advection-diffusion equation using a standard finite element method.

Algorithm 3 Algorithm for Time dependent Finite Element Method in deal.II.

Step1 Triangulation : Generate mesh

Step2 Degree of Freedom : Setup and associate DoFs on the mesh

Step3 Initialize with initial condition

Assemble the system :

time_step =0,

Choose theta (1 = implicit time stepping, 0 = explicit time stepping and 0.5 for Crank-Nicolson time stepping)

while (time$\leq$ Time_Maximum)

Finite Element : Define type of finite element

Quadrature : Define quadrature rule

FEValues : Mapping from reference to real cell.

Loop over cells

Loop over DoFs

Step4 Linear System: Collect Ax=b from previous step

Linear Solver : Compute $U = A^{-1}b$ with direct solver or use iterative solver

Step5 Output : Output solution

Solve for **n** time steps with time =time +time_step as updated time

End while loop

Now we will see the file **advectiondiffusion_problem.hpp** in the **include** folder and see how the deal.II function works step by step as shown in Algorithm (3).

Listing (2) is **advectiondiffusion_problem.hpp** header file that contains different types of constructor (refer Appendix IV [2]) is used in order to initialize the object and then after the program is complied it gets destroyed destructors is called in order to delete the function. This process is done to save memory. The text in green are the comments that gives the explanation of all the class created, followed by public class (refer Appendix IV [2]) function that can be access anywhere in the code and private class (refer Appendix IV [2]) that can be access only in certain part of code. An object(refer Appendix IV [2]) is created when a class is instantiated (i.e. a class is defined). Objects can be accessed using the dot ('.') operator with their data members and member functions. To access a member function with the name printName() on an object named obj, you will need obj.printName().

```
template <int dim> class AdvectionDiffusionProblem {
 public:
 /*!
 * Standard constructor disabled.
 */
 AdvectionDiffusionProblem() = delete;
 /*!
 * Default constructor.
 */
 AdvectionDiffusionProblem(unsigned int n_refine, bool is_periodic);
 /*!
 * Destructor.
 */
 ~AdvectionDiffusionProblem();
 /*!
 * @brief Run function of the object.
 *
 * Run the computation after object is built. Implements theping loop.
 */
 void run();

 private:
 /*!
 * @brief Set up the grid with a certain number of refinements
 * with either peridic or non-periodic bounday conditions.
 *
 * Generate a triangulation of \f$[0,1]^{\rm{dim}}\f$ with edges/faces
 * numbered form \f$1,\dots,2\rm{dim}\f$.
 */
 void make_grid();

 /*!
 * @brief Setup sparsity pattern and system matrix.
 *
 * Compute sparsity pattern and reserve memory for the sparse system matrix
 * and a number of right-hand side vectors. Also build a constraint object
 * to take care of Dirichlet boundary conditions.
 */
 void setup_system();
```

```
/*!
* @brief Assemble the system matrix and the right hand side at currtent time.
*
* Assembly routine to build the time-dependent matrix and rhs.
* Neumann boundary conditions will be put on edges/faces
* with odd number. Constraints are applied here.
*/
void assemble_system (double current_time);
/*!
* @brief Iterative solver.
*
* Parallel sparse direct solver through Amesos package.
*/
void solve_direct();
/*!
* @brief Iterative solver.
*
* CG-based solver with preconditioning.
*/
void solve_iterative();
/*!
* @brief Write results to disk.
*
* Write results to disk in vtu-format.
*/
void output_results(Vector& vector_out) const;
  /*!
* @brief Write results to disk.
*
* Calculate error.
*/
void compute_errors();
/*!
* triangulation
*/
Triangulation<dim> triangulation;

FE_Q<dim> fe;
DoFHandler<dim> dof_handler;

/*!
* Time-dependent matrix coefficient (diffusion).
*/
Coefficients::MatrixCoeff<dim> matrix_coeff;

/*!
* Time-dependent vector coefficient (velocity).
*/
Coefficients::AdvectionField<dim> advection_field;

/*!
* Time-dependent scalar coefficient (forcing).
*/
Coefficients::RightHandSide<dim> right_hand_side;

/*!
```

```
 * Time-dependent scalar coefficient (boundary flux).
 */
Coefficients::NeumannBC<dim> neumann_bc;

AffineConstraints<double> constraints;

SparsityPattern sparsity_pattern;

SparseMatrix system_matrix;

  Vector solution;
Vector old_solution;
Vector system_rhs;

double time;
double time_step;
unsigned int timestep_number;
/*!
 * parameter to determine the "implicitness" of the method.
 * Zero is fully implicit and one is (almost explicit).
 */
const double theta;
/*!
 * Final simulation time.
 */
const double T_max;
/*!
 * Number of initial refinements.
 */
unsigned int n_refine;
/*!
 * If this flag is true then periodic boundary conditions
 * are used.
 */
bool is_periodic;

ConvergenceTable convergence_table;

};
```

Listing 2: The advection-diffusion problem class definition and object functions in **advectiondiffusion_problem.hpp** header file.

Note : Following are the points the reader should keep in their mind.

- A reader should note that for the diffusion equation solution with standard finite element method some of the terms would be removed and can be found in file of the Github link described in Chapter 1.
- All the codes follow same structure, first, we define a class with a constructor and destructor, functions and declare the object functions, then we initialize the function we defined in the class, and finally we delete the destructor.
- A programmer can designate that a new class should inherit the members of an

existing class instead of writing entirely new data members and member functions. The **base** class is the existing one, and the **derived** class is the newly created one.

- The use of C++ feature called pointer "*" for dynamic memory allocation of object. This pointer requires an address "&"in order to access a cell.

Listing (3) shows initialization of all the object function defined above.

```
template <int dim>
AdvectionDiffusionProblem<dim>::AdvectionDiffusionProblem(unsigned int n_refine,
bool is_periodic)
: triangulation()
, fe(1)
, dof_handler(triangulation)
, time(0.0)
, time_step(1. / 100)
, timestep_number(0)
/*
* theta=1 is implicit Euler,
* theta=0 is explicit Euler,
* theta=0.5 is Crank−Nicolson
*/
, theta(1.0)
, T_max(0.5)
, n_refine(n_refine)
, is_periodic(is_periodic)
{}
```

Listing 3: The advection-diffusion problem initialization of object functions in **advectiondiffusion_problem.hpp** header file.

The Listing (4) destroys the construction once the program has been complied.

```
template <int dim>
AdvectionDiffusionProblem<dim>::~AdvectionDiffusionProblem()
{
 system_matrix.clear();
 constraints.clear();
 dof_handler.clear();
}
```

Listing 4: The advection-diffusion problem destruction **advectiondiffusion_problem.hpp** header file.

Figure (5.5) shows inter connection between deal.II classes. When explaining the code, it would be helpful to understand the dependencies between the classes.

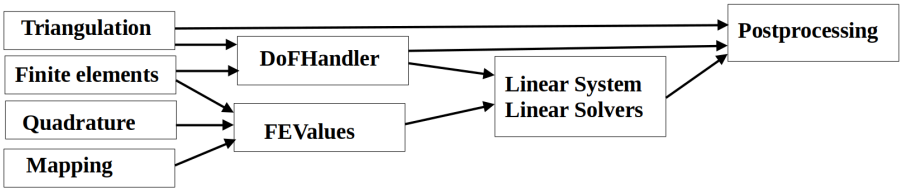


Figure 5.5: Connection between the most important classes in deal.II.

Now we dive into steps of deal.II in detail using **advectiondiffusion_problem.hpp** header file in order to understand Figure (5.5) and Algorithm (3).

5.3 Triangulation

The triangulation function provides line segments (1D), quadrilaterals (2D), and hexahedra (3D). As with almost the entire library, the triangulation class is structured so that the space dimension is determined by a template parameter [9]. In the result, users can write codes independent of space dimensions. This enables them to develop and test a program in 2D, then run it in 3D. Listing (6) illustrates how a Triangulation object is created. This function contains a template parameter that specifies the dimension. Depending on the value of the template parameter **dim**, a hypercube $[0,1]^{dim}$ (i.e., the unit line, unit square, or unit cube) is generated and refined uniformly twice into 4^{dim} mesh cells. It should be noted that the dimension is a compile-time constant.

5.3.1 Refinement

In deal.II, regular refinement (bisection) leads to 2, 4, or 8 children per cell in 1D, 2D, and 3D, respectively. As shown in Figure (5.6) in 2D the first refinement is $4^1 = 4$ cells in the left figure. In the middle Figure it is $4^2 = 16$ cells and in the last Figure $4^3 = 64$ cells.

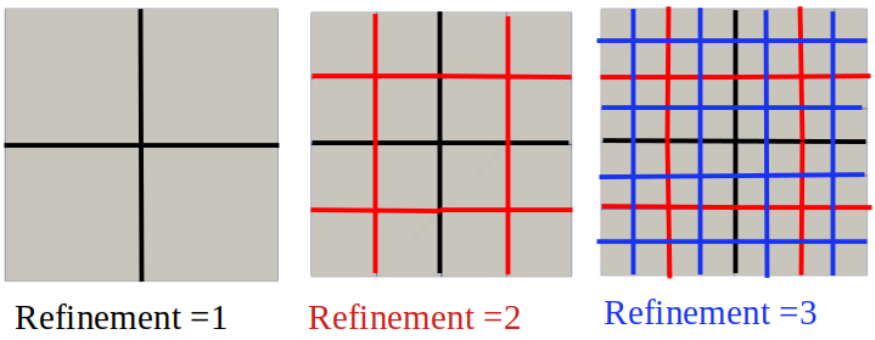


Figure 5.6: Mesh refinement in deal.II.

A triangulation of cells is therefore a binary tree, quad-tree, or oct-tree [32], with terminal nodes corresponding to active cells without children. As shown in Figure (5.7) **0.0** is a parent cell with four children namely 1.0,1.1, 1.2 and 1.3. Thereafter, 1.3 is the parent of four children 2.0,2.1,2.3, and 2.4. Non-terminal nodes refer to cells that are inactive, that is, cells that have children but are not part of the mesh hierarchy. Listing (5) shows code to output the number of all active cells in a triangulation using triangulation.n_active_cells().

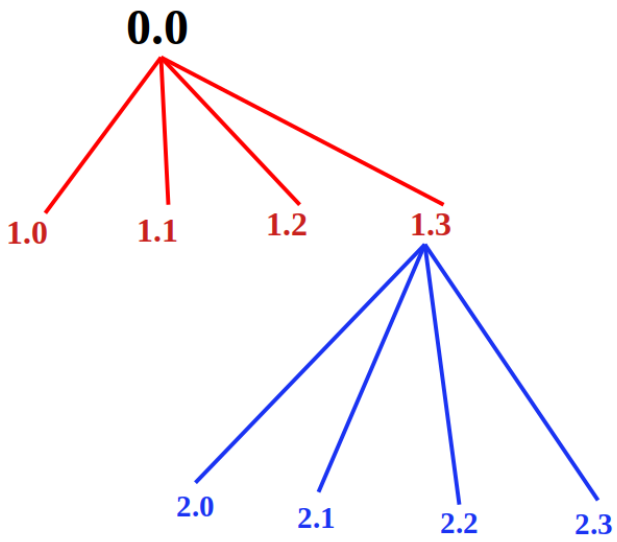


Figure 5.7: Quad-tree of cells.

```
std::cout << "Number of active cells: "<< triangulation.n_active_cells()<< std::endl;
```

Listing 5: Active cells in triangulation.

Figure (5.8) shows a refined 2D mesh along with its tree of cells.

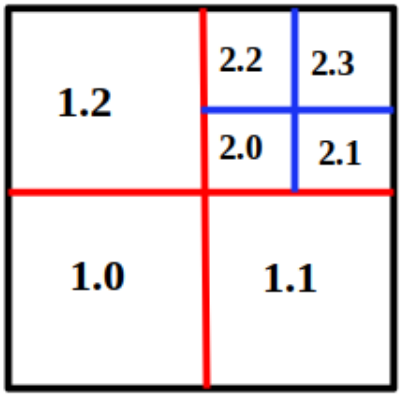


Figure 5.8: Mesh refinement in deal.II.

5.3.2 Mesh generation

The GridGenerator class (for class refer Appendix IV [2]) provides functions to automatically generate the most common and simplest geometries. Gmsh, Lagrit, and Cubit are programs that support grid data input formats. In the triangulation class, meshes can be improved or modified when they are created or loaded.
In triangulation, a domain is created, followed by mesh generation. First one needs to define the predefined headers of deal.II for mesh generation to visualization as follows.

```
#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/grid_tools.h>
#include <deal.II/grid/tria.h>
#include <deal.II/grid/tria_accessor.h>
#include <deal.II/grid/tria_iterator.h>
```

Then in order to generate C++ output, we call the predefined C++ header. To use math functions, we use the C++ header that includes math functions.

```
#include <fstream>
#include <iostream>h>
#include <cmath>
```

Lastly, import deal.II by placing all the functions and classes in a namespace deal.II, so that they don't conflict with other libraries you may want to use together.

```
using namespace dealii;
```

The triangulation object is first defined.

```
Triangulation<dim> triangulation;
GridGenerator::hyper_cube(triangulation, 0,1,true);
```

Listing 6: Mesh of arbitrary dimension.

The following template (for template refer Appendix IV [7]) code in Listing (6) creates a square domain with the help of function hyper_cube. Here void in C++ means it does not return a value. In the previous section, dim is 2 and n_refine was defined as 3. It would be used here as $4^3 = 64$.
In deal.II mesh cells is accessible by iterators rather than indices. For this purpose, the Triangulation class provides Standard Template Library (STL) (for STL refer Appendix IV [8]) like iterators that "point" to objects describing cells, faces, edges, and

other objects in a mesh. Listing (7) provides iterator that goes to each cell in triangulation find the periodic faces and pairs them using GridTools::PeriodicFacePair structure (for structure refer Appendix IV [9]).

```
for (unsigned int d = 0; d < dim; ++d)
{
  GridTools::collect_periodic_faces(triangulation,/*b_id1*/ 2 * (d + 1) - 2, /*b_id2*/ 2 * (d + 1) - 1,
  /*direction*/ d, periodicity_vector);
}
```

Listing 7: Iterator for periodic faces.

The periodic condition is applied from top to bottom (top is 2 edge and bottom is 3 edge in deal.II) and left to right (left is 0 edge and right is 1 edge in deal.II) as seen from Figure (5.9) and in the Listing (8).

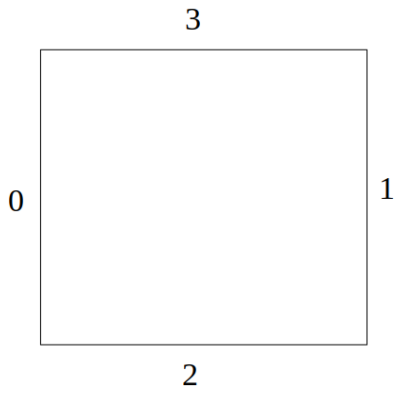


Figure 5.9: Periodic edges in 2D.

```
template <int dim>
void AdvectionDiffusionProblem<dim>::make_grid()
{
  GridGenerator::hyper_cube(triangulation, 0,1,true);

  if (is_periodic)
  {
    std::vector<GridTools::PeriodicFacePair<typename Triangulation<dim>::cell_iterator>> periodicity_vector;

    for (unsigned int d = 0; d < dim; ++d)
    {
      GridTools::collect_periodic_faces(triangulation, /*b_id1*/ 2 * (d + 1) - 2,
                        /*b_id2*/ 2 * (d + 1) - 1, /*direction*/ d, periodicity_vector);
    }

    triangulation.add_periodicity(periodicity_vector);
  } // if
```

```
  triangulation.refine_global(n_refine);

  std::cout << "Number of active cells: " << triangulation.n_active_cells() << std::endl;
}
```

Listing 8: Applying periodic faces on the boundaries.

Now we need to call the make_grid function in run to generate the grid. The following lines from 7 to 10 in Listing (9) are required to visually represent grids in paraview software.

```
1  template <int dim>
2  void
3  AdvectionDiffusionProblem<dim>::run()
4  {
5
6    make_grid();
7    std::ofstream out(filename);
8    GridOut grid_out;
9    grid_out.write_vtk(triangulation, out);
10   std::cout << " written to " << filename << std::endl << std::endl;
11 }
```

Listing 9: Grid output of triangulation.

Finally we can see the grid as follows as shown in Figure (5.10)

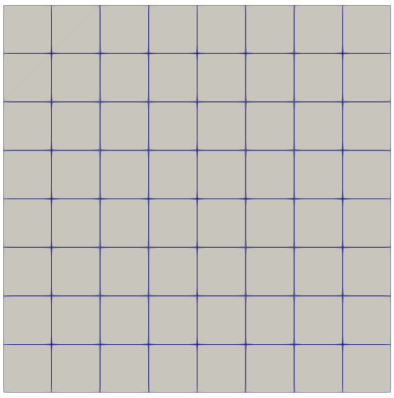


Figure 5.10: Grid generation.

Note : This is part of the big code in **advectiondiffusion_problem.hpp** header file. Lines 7 to 10 in Listing (9) are used to write the output. Here it is just to visualize the grid, they are not present in Appendix IV [61]. To keep it simple, only the triangulation part is shown. Later, we will discuss parallelization in more detail.

5.4 Degree of Freedom Handler

Handling degrees of freedom is the responsibility of the Degree of Freedom Handler (DoFHandler) class. Class DoFHandler provides a global enumeration of degrees of freedom, given a Triangulation object describing the mesh and a Finite Element object describing how degrees of freedom are associated with vertices, faces, and cells. These classes are built on triangulation and finite element classes.

Figure (5.11) shows such enumerations for the mesh presented in Figure (5.8).

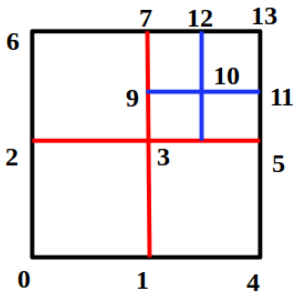


Figure 5.11: Degree of freedom in deal.II.

Listing (10) shows a typical code fragment for initializing a DoFHandler based on the triangulation of Listing (6). Due to the fact that we are using the FE_Q class and set the polynomial degree to 1, or bilinear elements, we get one degree of freedom for each vertex. As we generate output, let's also check the degree of freedom:

```
DoFHandler<dim>    dof_handler;
FE_Q<dim>    fe;
dof_handler(triangulation);
dof_handler.distribute_dofs(fe);

std::cout << std::endl<< "===========================================" << std::endl
<< "Number of active cells: " << triangulation.n_active_cells()<< std::endl
<< "Number of degrees of freedom: " << dof_handler.n_dofs()<< std::endl<< std::endl;
```

Listing 10: Initialization of a DoFHandler for Q1 elements.

There is one DoF for each vertex. Since we have a 8×8 grid, the number of DoFs should be 9×9, or 81. Now we apply Dirichlet boundary conditions as shown in Listing (11).

```
constraints.clear();
DoFTools::make_hanging_node_constraints(dof_handler, constraints);

/*
* Set up Dirichlet boundary conditions.
*/
```

```
const Coefficients::DirichletBC<dim> dirichlet_bs;
for (unsigned int i = 0; i<dim; ++i)
{
  VectorTools::interpolate_boundary_values(dof_handler, /*boundary id*/ 2*i, // only even boundary id dirichlet_bs,constraints);
}

constraints.close();
```

Listing 11: Set Dirichlet boundary conditions.

5.5 Finite Element

The finite element that describes the shape functions on the reference cell (which is always unit interval $[0,1]$, square $[0,1]^2$ or cube $[0,1]^3$ in deal.II, depending on your space dimension). Here we used an object of type FE_Q<dim>, which denotes the usual Lagrange elements that define shape functions by interpolation on support points. The simplest one is FE_Q<dim>(1), which uses a single polynomial degree. Since they are linear in each of the two coordinates of the reference cell, they are often called bilinear in 2D [7].

Listing (12) shows FE_Q<dim> class that describes Lagrange elements. The constructor takes one argument, which is a polynomial degree representing the element, in this case one (a bi-linear element); there is one degree of freedom at each vertex, while none on lines or inside the quadrilateral.

```
FE_Q<dim>    fe;
```

Listing 12: Finite Element.

5.6 Quadrature

A quadrature formula is provided in deal.II, along with the base class quadrature. The quadrature formula provides two essential pieces of information: the location and weight of the quadrature points in the unit cell.

Quadrature formulas in arbitrary dimensions are represented by this base class. This class stores quadrature points and weights in the coordinate system of a reference cell and represents quadrature points and weights on the unit line segment $[0,1]$ in 1 dimension, on the unit square or unit triangle in 2 dimensions, as well as unit tetrahedrons, cubes, pyramids, and wedges in 3 dimensions.

Integration formulae are denoted by derived classes. There is a Q prefix on their names.

The quadrature formulas of quadrilaterals and hexahedra (or, more precisely, the unit square and unit cube, since we work on reference cells) are typically tensor products of one-dimensional formulas.

In the following Listing (13) we see that the Gauss formula with two quadrature points in each direction is applied for the evaluation of the integrals in each element.

```
QGauss<dim> quadrature_formula(fe.degree + 1);
QGauss<dim − 1> face_quadrature_formula(fe.degree + 1);
```

Listing 13: Quadrature formula for the evaluation of the integrals.

5.7 Mapping

Matrixes and vectors are typically assembled by looping over all cells, and computing the contribution of each cell to the global matrix and right hand side by quadrature. It is important to understand now that we need to know the values of the shape functions at quadrature points on the real cell. Finite element shape functions and quadrature points, however, can only be defined on the reference cell. Since they are of little use to us, we rarely query finite element shape functions or quadrature points from them.

Instead, we need a way to map this data from the reference cell to the real cell. This can be done with classes derived from the Mapping class, although one is not often forced to use them directly: many functions in the library take a mapping object as an argument, but if it is not given, they resort to the standard bilinear Q1 mapping which is the case in our code.

5.8 FEValues

In this module, one can construct matrices and vectors. They connect finite elements, quadrature objects, and mappings. As integration happens at quadrature points on the real cell, the values and gradients of finite element shape functions at these points must be known. This information is coordinated by the FEValues class. FEFaceValues provides similar functionality to FEValues for integrations on faces (for example, on the boundary or between cells). In addition, the FESubfaceValues class can integrate on parts of faces if the neighboring cell has been refined and the current cell shares only a part of its face with the neighboring cell.

Listing (14) gives an overview of the assembly part we discussed in chapter 1. FEFaceValues initialize the cell and assemble system by taking quadrature point for all

parts of integral to be evaluated with respect to Q1 finite element and degrees of freedom on the cell.

```
FEValues<dim>  fe_values (fe, quadrature_formula,update_values | update_gradients |
update_quadrature_points | update_JxW_values);

FEFaceValues<dim>  fe_face_values(fe,face_quadrature_formula, update_values | update_quadrature_points |
update_normal_vectors |update_JxW_values);// for Neumaan boundary condition to evaluate boundary condition

fe_values.reinit (cell);

// Now actually fill with values.
matrix_coeff.value_list(fe_values.get_quadrature_points (),matrix_coeff_values);
advection_field.value_list(fe_values.get_quadrature_points(),advection_field_values);
right_hand_side.value_list(fe_values.get_quadrature_points(),rhs_values);

/*
* Integration over cell.
*/
for (unsigned int q_index=0; q_index<n_q_points; ++q_index)
{
  for (unsigned int i=0; i<dofs_per_cell; ++i)
  {
    for (unsigned int j=0; j<dofs_per_cell; ++j)
    {

      cell_diffusion_matrix(i,j) += fe_values.shape_grad(i,q_index) * matrix_coeff_values[q_index] *
        fe_values.shape_grad(j,q_index) *fe_values.JxW(q_index);
      cell_advection_matrix(i,j) += fe_values.shape_grad(i,q_index) *advection_field_values[q_index]
      *fe_values.JxW(q_index);
      cell_mass_matrix(i,j) += fe_values.shape_value(i,q_index) *fe_values.shape_value(j,q_index) *
      fe_values.JxW(q_index);
    } // end ++j

    cell_rhs(i) += fe_values.shape_value(i,q_index) *rhs_values[q_index] *fe_values.JxW(q_index);
  } // end ++i
} // end ++q_index

/*
* Boundary integral for Neumann values for odd boundary_id.
*/
for (unsigned int face_number = 0; face_number < GeometryInfo<dim>::faces_per_cell;
++face_number)
{
  if (cell->face(face_number)->at_boundary() &&
  (
  (cell->face(face_number)->boundary_id() == 1) ||(cell->face(face_number)->boundary_id() == 3) ||
  (cell->face(face_number)->boundary_id() == 5)
  )
  )
  {
    fe_face_values.reinit(cell, face_number);

    /*
    * Fill in values at this particular face.
```

```
    */
    neumann_bc.value_list(fe_face_values.get_quadrature_points(),neumann_values);

    for (unsigned int q_face_point = 0; q_face_point < n_face_q_points; ++q_face_point)
    {
      for (unsigned int i = 0; i < dofs_per_cell; ++i)
      {
        cell_rhs(i) += time_step *neumann_values[q_face_point] * // g(x_q)
        fe_face_values.shape_value(i, q_face_point) * // phi_i(x_q)fe_face_values.JxW(q_face_point);    // dS
      } // end ++i
    } // end ++q_face_point
  } // end if
} // end ++face_number
```

Listing 14: FEValues in deal.II.

5.9 Linear System

Linear System consists of classes that involve linear algebra, such as matrices, vectors, and linear systems. Listing (15) gives the object definition for linear algebra classes like SparsityPattern and SparseMatrix<double> followed by applying dirichlet boundary condition and initializing different matrices.

```
SparsityPattern    sparsity_pattern;
SparseMatrix<double> mass_matrix;
SparseMatrix<double>    diffusion_matrix;
SparseMatrix<double>    advection_matrix;
SparseMatrix<double>    system_matrix;

DynamicSparsityPattern dsp(dof_handler.n_dofs());
DoFTools::make_sparsity_pattern (dof_handler, dsp, constraints,
/*keep_constrained_dofs = */ true, // for time stepping this is essential to be true
sparsity_pattern.copy_from(dsp);

system_matrix.reinit (sparsity_pattern);
diffusion_matrix.reinit (sparsity_pattern);
advection_matrix.reinit (sparsity_pattern);
mass_matrix.reinit(sparsity_pattern);
```

Listing 15: Linear system.

5.10 Linear Solver

Several control classes are included, such as the iterative and direct solvers, the eigenvalue solvers, and the direct solvers. deal.II defines matrix and vector classes that operate on these objects.

5.10.1 Direct Solver

The linear system with sparse LU decomposition provided by UMFPACK(Unsymmetric MultiFrontal PACKage). It often works well for 2D problems despite high degrees of freedom. It is a set of routines used to solve nonsymmetric sparse linear systems, Ax = b, using the Unsymmetric-pattern MultiFrontal method and direct sparse LU factorization. Sparsity patterns and entries in matrices may be symmetric or asymmetric. A SparseDirectUMFPACK class provides the deal.II interface to UMFPACK, which is very easy to use and allows us to solve our linear system in just a few lines of code.

The code to solve the linear system is simple: First, we allocate an object **A_direct** of the right type. The initialization call below provides a SparseDirectUMFPACK object with a matrix and initiates the LU-decomposition. In this program, most computation occurs here.

As a result of the decomposition, we can use **A_direct** as the inverse of our system matrix. We can compute its solution by multiplying with the right hand side vector as shown in Listing (16).

```
SparseDirectUMFPACK A_direct;
solution = system_rhs;
A_direct.solve(system_matrix, solution);
A_direct.vmult(solution, system_rhs);
```

Listing 16: Direct Solver.

5.10.2 Iterative Solver

A discretized equation can be solved with the **solve_iterative()** function. Due to the large system, we use a Conjugate Gradient algorithm such as Gauss elimination or LU decomposition instead of direct solvers.

```
template <int dim>
void AdvectionDiffusionProblem<dim>::solve_iterative ()
```

To begin with, we need an object that tells when to terminate the CG algorithm. The SolverControl object is used to accomplish this. As a stopping criterion, we say: stop after 1000 iterations (which is much more than needed for 81 variables; see the results section to find out how many were actually used) and terminate when the residual norm is below 10^{-12}. As a result, the second criterion will stop iteration.

```
SolverControl solver_control(1000, 1e-12);
```

After that, we need the solver itself. The SolverCG class takes a template parameter that specifies the type of vectors. However, the empty angle brackets indicate that we simply use the default (which is **Vector<double>**).

```
SolverCG<> solver(solver_control);
```

Let's solve the system now. As its fourth argument, the CG solver accepts a preconditioner. With a relaxation factor of 1.2, we will use SSOR (symmetric successive overrelaxation). SSOR steps are performed by the SparseMatrix class. We need to package its address together with the invert matrix, the relaxation factor, and the matrix on which the SSOR step should act into one object. PreconditionSSOR does this for us. (The PreconditionSSOR class takes a template argument that indicates what matrix type it should work on. The default value is SparseMatrix<double>, which is exactly what we want, so we use that and do not specify anything in the angle brackets.)

```
PreconditionSSOR<> preconditioner;
preconditioner.initialize(system_matrix, 1.2);

solver.solve (system_matrix,solution,system_rhs,preconditioner);
```

5.11 Output

There are three types of graphical output generated by deal.II as follows:

1. Grid output: Meshes, without any data vectors associated with them, can also be written in multiple formats. The GridOut class handles this, it output a triangulation to a file in different formats.
2. Visualization of data: In deal.II, the DataOutBase class supports a large number of popular visualization formats, such as OpenDX, gmv, or gnuplot. DataOutBase is a base class for outputting data on meshes of general form. As a set of patches, output data is written to the output stream in the format expected by the visualization tool. A patch represents a single logical cell of a mesh, which can be subdivided number of times to represent higher order polynomials defined on this cell.
3. A graphical representation of matrices can also be generated by deal.II through the MatrixOut class. DataOutBase handles MatrixOut output.

Listing (33) gives code to get solution output.

```
DataOut<dim> data_out;
data_out.attach_dof_handler (dof_handler);
data_out.add_data_vector (solution, "solution");
data_out.build_patches ();
```

Listing 17: Output.

5.12 Compute the error

Let $f : \Omega \to \mathbb{R}^c$ be a finite element function with c components where component c is denoted by f_c and $\hat{f}$ be the reference function (the fe_function and exact_solution arguments to integrate_difference()). Let $e_c = \hat{f}_c - f_c$ be the difference or error between the two. Further, let $w : \Omega \to \mathbb{R}^c$ be the weight function of integrate_difference(), which is assumed to be equal to one if not supplied. Finally, let p be the exponent argument (for L_p-norms) [7]. Let E_k the local error computed by integrate_difference() on cell K, whereas E is the global error computed by compute_global_error().
Note that integrals are approximated by quadrature in the usual way:

$$\int_A f(x)\,dx \approx \sum_q f(w_q)w_q.$$

Similarly for suprema over a cell T:

$$\sup_{x\in T} |f(x)|\,dx \approx max_q |f(x_q)|.$$

L^2 error

The square of the function is integrated and the square root of the result is computed on each cell [7]:

$$E = \sqrt{\sum_K E_k^2} = \sqrt{\int_\Omega \sum_k e_c^2 w_c}$$

H^1 error

The square of this norm is the square of the L2_norm plus the gradient term [7]:

$$E = \sqrt{\sum_K E_k^2} = \sqrt{\int_\Omega \sum_k (e_c^2 + (\nabla e_c)^2) w_c}$$

L^∞ error

The square of the function is integrated and the square root of the result is computed on each cell [7]:

$$E = \max_k E_k = \sup_\Omega \max_c |e_c| w_c$$

Listing (35) gives the computing error based on the formulas discussed above. The TableHandler writes the table as text for the output of the solution.

```
Vector<float> difference_per_cell(triangulation.n_active_cells());
VectorTools::integrate_difference(dof_handler,solution,ZeroFunction<dim>(),difference_per_cell,
QGauss<dim>(fe.degree + 1),VectorTools::L2_norm);

const double L2_error = VectorTools::compute_global_error(triangulation,difference_per_cell,VectorTools::L2_norm);

const unsigned int n_active_cells = triangulation.n_active_cells();
const unsigned int n_dofs = dof_handler.n_dofs();
VectorTools::integrate_difference(dof_handler,solution,ZeroFunction<dim>(),difference_per_cell,
QGauss<dim>(fe.degree + 1),VectorTools::H1_norm);

const double H1_error = VectorTools::compute_global_error(triangulation, difference_per_cell,VectorTools::H1_norm);

VectorTools::integrate_difference(dof_handler,solution,ZeroFunction<dim>(),
difference_per_cell,QGauss<dim>(fe.degree + 1),VectorTools::Linfty_norm);
const double Linfty_error =VectorTools::compute_global_error(triangulation,difference_per_cell, VectorTools::Linfty_norm);
pcout << " Number of active cells: "<< n_active_cells<< std::endl
<< " Number of degrees of freedom: "<< n_dofs<< std::endl;
convergence_table.add_value("cells", n_active_cells);
convergence_table.add_value("dofs", n_dofs);
convergence_table.add_value("L2", L2_error);
convergence_table.add_value("H1", H1_error);
convergence_table.add_value("Linfty", Linfty_error);
convergence_table.set_precision("L2", 3);
convergence_table.set_precision("H1", 3);
convergence_table.set_precision("Linfty", 3);
convergence_table.set_scientific("L2", true);
convergence_table.set_scientific("H1", true);
convergence_table.set_scientific("Linfty", true);
convergence_table.set_tex_caption("cells", "\\# cells");
convergence_table.set_tex_caption("dofs", "\\# dofs");
convergence_table.set_tex_caption("L2", "L^2−error");
convergence_table.set_tex_caption("H1", "H^1−error");
convergence_table.set_tex_caption("Linfty", "L^\\infty−error");

convergence_table.set_tex_format("cells", "r");
convergence_table.set_tex_format("dofs", "r");

std::cout << std::endl;
convergence_table.write_text(std::cout);

std::ofstream error_table_file("tex−conv−table.tex");
convergence_table.write_tex(error_table_file);
```

```
deallog << " Error in the L2 norm : " << L2_error << std::endl;
deallog << " Error in the H1 norm : " << H1_error <<std::endl;
deallog << " Error in the Linfty norm : " << H1_error<< std::endl;
```

Listing 18: Compute error.

Flow chart (5.12) shows the summary of the whole code which we discussed in part above. The reader should see the bottom box that is the legend for the flow chart colors. The vertical arrow shows the steps to call the function in an orderly manner. In some cases, the function on top appears in the bottom function. It is explicitly mentioned, or sometimes it uses the C++ set ↔ get feature to define set functions in one part of code. It is called using get on the function in another part code.

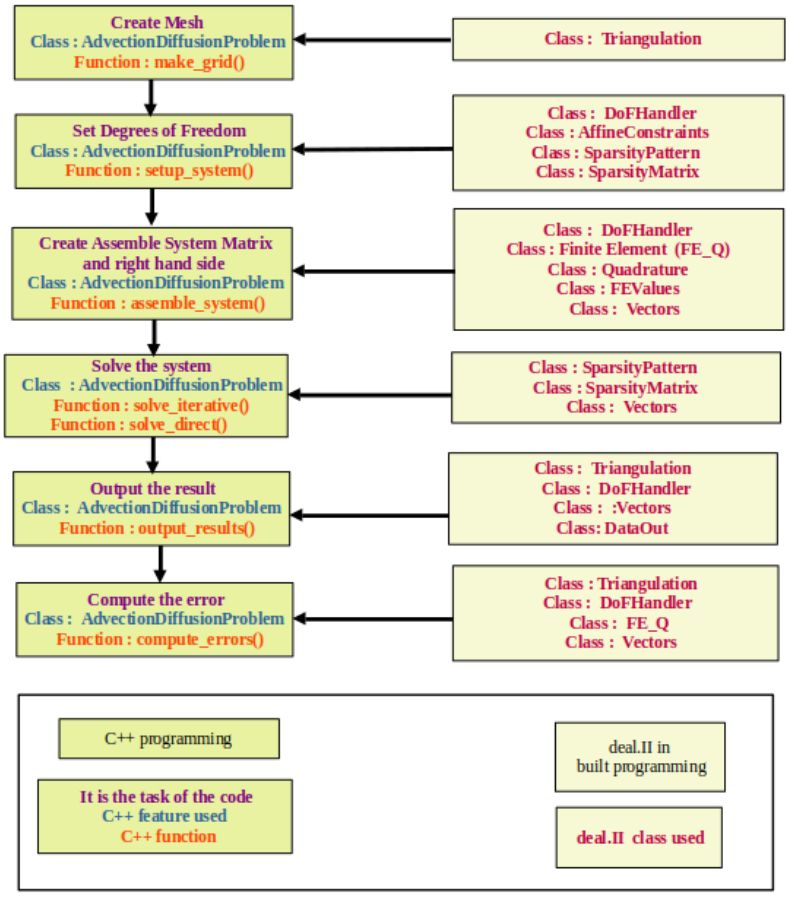


Figure 5.12: Flow chart of advection diffusion equation solution code in **advectiondiffusion_problem.hpp** file: left is the C++ programming code and right is deal.II classes used in the code.

Figure (5.13) gives the summary of the boundary conditions implemented in the code of the solution of the advection diffusion equation. The boundary conditions are individual header files containing functions implemented using deal.II class. The boundary conditions, diffusion coefficient and advection are then called in the assembly part of the

code in **advectiondiffusion_problem.hpp** file by using there respective object functions.

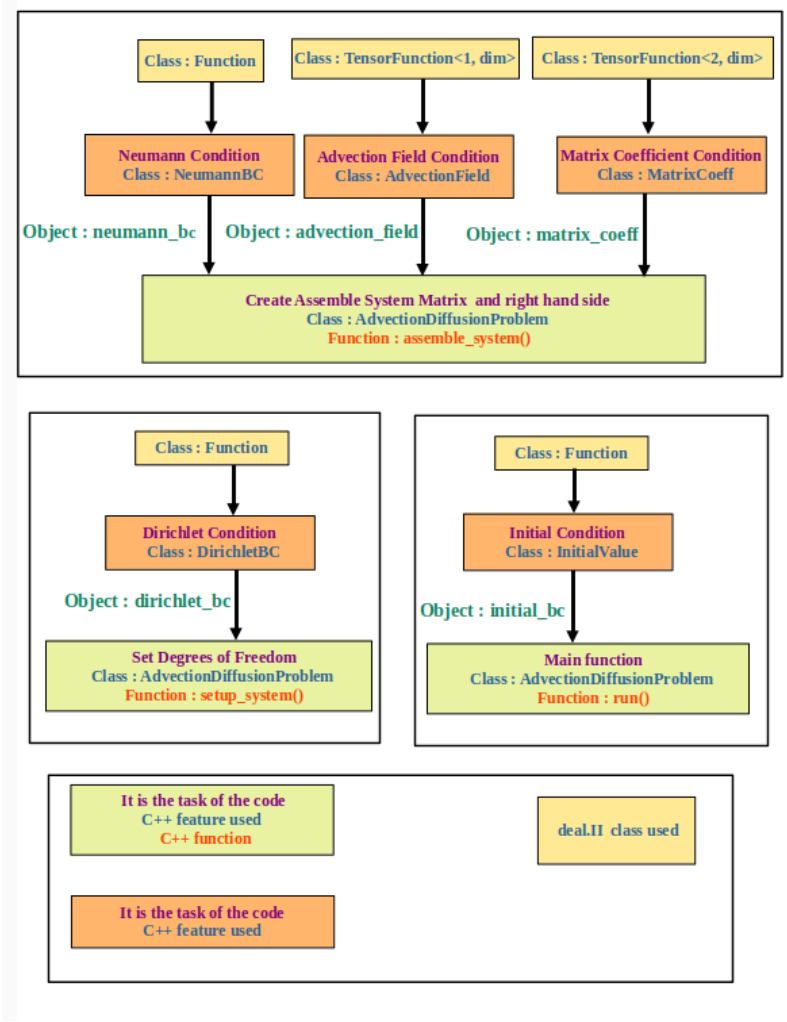


Figure 5.13: Flow chart of the boundary conditions connected to main code in **advectiondiffusion_problem.hpp** file code in **assemble system** with object functions.

Listing (19) gives diffusion coefficient defined as **MatrixCoeff** class and we have defined an object function **matrix_coeff** for it in the main file. Furthermore, it is used in the assembly part of the code.

Note : All the coefficients and Neumann boundary condition are used to assemble system part of code only Dirichlet boundary condition goes in set degree of freedom part of the code.

```
#ifndef INCLUDE_MATRIX_COEFF_HPP_
#define INCLUDE_MATRIX_COEFF_HPP_

// Deal.ii
#include <deal.II/base/tensor_function.h>

// STL
#include <cmath>
```

```
#include <fstream>

// My Headers
#include "coefficients.h"

namespace Coefficients {
using namespace dealii;

  /*!
  * @class MatrixCoeff
  * @brief Diffusion coefficient.
  *
  * Class implements a matrix valued diffusion coefficient.
  * This coefficient must be positive definite.
  */

template <int dim> class MatrixCoeff : public TensorFunction<2, dim>
  {
    public:  MatrixCoeff() : TensorFunction<2, dim>() {}

    virtual Tensor<2, dim> value(const Point<dim> &point) const override;
    virtual void value_list(const std::vector<Point<dim>> &points,
    std::vector<Tensor<2, dim>> &values) const override;

    const bool is_transient = false;
    private:
    const int k = 240;
    const double scale_factor = 0.9999;
  };
template <int dim>
Tensor<2, dim> MatrixCoeff<dim>::value(const Point<dim> &p) const
  {
    Tensor<2, dim> value;
    value.clear();
    const double t = this->get_time();
    for (unsigned int d = 0; d < dim; ++d)
    {
      using numbers::PI;
      value[d][d] =1*(1- sin(2 * PI*2* p(d)/0.05));
    }
    return value;
  }
template <int dim>
void MatrixCoeff<dim>::value_list(const std::vector<Point<dim>> &points,
  std::vector<Tensor<2, dim>> &values) const
  {
    Assert(points.size() == values.size(),ExcDimensionMismatch(points.size(), values.size()));

    for (unsigned int p = 0; p < points.size(); ++p)
    {
      values[p].clear();
      for (unsigned int d = 0; d < dim; ++d)
      {
        using numbers::PI;
        values[p][d][d] =1*(1- sin(2 * PI *2* points[p](d)/0.05));
      }
    }
```

```
  }

} // end namespace Coefficients
#endif /* INCLUDE_MATRIX_COEFF_HPP_ */
```

Listing 19: Diffusion Coefficient.

Lisiting (20) gives the code for implementation of diffusion coefficients in the assemble system step of the main code. As we can see, matrix coefficients that are tensors take quadrature values as input, is used to evaluate the integrals defined on a finite element at each degree of freedom.

```
template <int dim>
void AdvectionDiffusionProblem<dim>::assemble_system(double current_time)
{
  TimerOutput::Scope t(computing_timer, "assembly");

  const QGauss<dim> quadrature_formula(fe.degree + 1);
  const QGauss<dim - 1> face_quadrature_formula(fe.degree + 1);

  FEValues<dim> fe_values(fe,quadrature_formula,update_values |
  update_gradients |update_quadrature_points | update_JxW_values);

  const unsigned int dofs_per_cell = fe.dofs_per_cell;
  const unsigned int n_q_points = quadrature_formula.size();
  const unsigned int n_face_q_points = face_quadrature_formula.size();

  FullMatrix<double> cell_matrix(dofs_per_cell, dofs_per_cell);
  Vector<double> cell_rhs(dofs_per_cell);

  std::vector<types::global_dof_index> local_dof_indices(dofs_per_cell);

  std::vector<Tensor<2, dim>> matrix_coeff_values_old(n_q_points);
  std::vector<Tensor<2, dim>> matrix_coeff_values(n_q_points);

  for (const auto &cell : dof_handler.active_cell_iterators())
  {
    if (cell->is_locally_owned())
    {
      cell_matrix = 0;
      cell_rhs = 0;

      fe_values.reinit(cell);
      cell->get_dof_indices(local_dof_indices);
      /*
      * Values at current time.
      */
      matrix_coeff.set_time(current_time);

      matrix_coeff.value_list(fe_values.get_quadrature_points(),matrix_coeff_values);

      /*
      * Values at previous time.
      */
```

```
        matrix_coeff.set_time(current_time − time_step);

        matrix_coeff.value_list(fe_values.get_quadrature_points(),matrix_coeff_values_old);
  for (unsigned int q_index = 0; q_index < n_q_points; ++q_index)
  {
    for (unsigned int i = 0; i < dofs_per_cell; ++i)
    {
      for (unsigned int j = 0; j < dofs_per_cell; ++j)
      {
        // Diffusion is on rhs. Careful with signs here.
        cell_matrix(i, j) += (fe_values.shape_value(i, q_index) ∗
        fe_values.shape_value(j, q_index) + time_step ∗ (theta) ∗(fe_values.shape_grad(i, q_index) ∗
        matrix_coeff_values[q_index] ∗fe_values.shape_grad(j, q_index) +
        fe_values.shape_value(i, q_index) ∗advection_field_values[q_index] ∗fe_values.shape_grad(j, q_index))) ∗
        fe_values.JxW(q_index);
      }
    }
  }

}
```

Listing 20: Diffusion Coefficient implemented in assemble system.

Here we conclude the steps of solving advection diffusion equation with standard finite element method using deal.II. The reader can see the whole code in Appendix IV [61]. Now for the larger problems with high degree of freedoms the code needs to be parallelized.

5.13 Parallelization Concept

Parallelization is mainly used for high degrees of freedom, especially for canopy applications that have millions of degrees of freedom. In order to modify code for parallel computing we need to know some parallelization concepts.

5.13.1 Motivation

A single core can often handle a 2-dimensional problem of higher complexity. The real world, however, is three-dimensional, and applications are very high in demand. A single processor cannot solve these 3-dimensional problems due to their degree of freedom. The code is adapted to multiple cores in order to speed up simulation time. A Linux machine and a Marin cluster (`https://www.cen.uni-hamburg.de/facilities/cen-it/compute.html`) are used to run the code.

In deal.II, two kinds of parallelization strategies, distributed mesh and shared mesh, are used for optimal efficiency as shown in Figure (5.14).

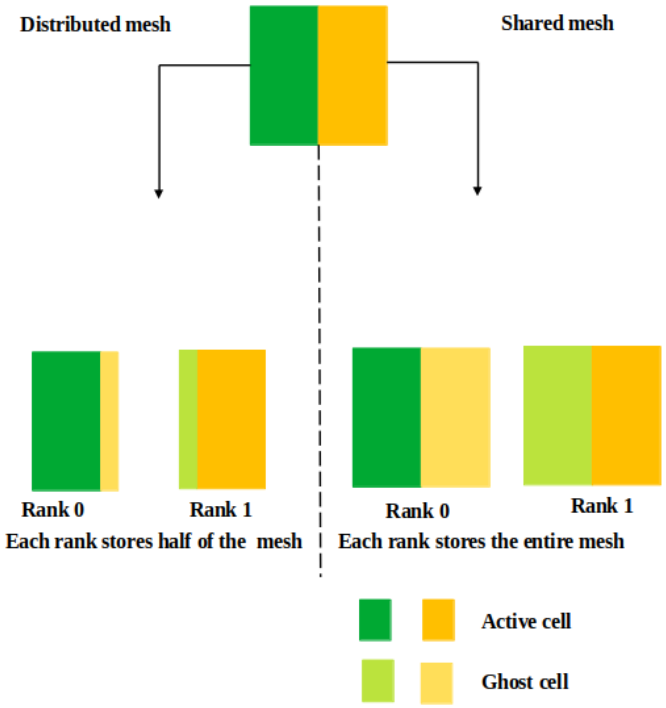


Figure 5.14: Distributed and Shared mesh.

- MPI stands for Message Passing Interface, and is an Application Program Interface that defines a parallel computing model where each parallel process has its own memory, and data must be explicitly shared between processes by passing messages. It enables programs to scale beyond the processors and shared memory of a single computing server to combine the processors and memory of multiple computing servers. MPI ranks with fully distributed mesh partitions only store a portion of the mesh, vectors, matrices, etc. MPI ranks do not have access to the entire mesh. In contrast, an MPI rank owns some elements, nodes, degrees of freedom, etc that it can read and modify. MPI ranks must communicate with neighboring ranks in finite element methods. In a "ghost" layer, information exchange occurs between elements owned by specific MPI ranks and their direct neighbors. Ghost layers cannot be written to, but can be read by MPI ranks. Ghost elements are set by the MPI ranks that own them. As a result of distributed mesh partitioning, a lot of memory can be saved by storing only a portion of the mesh, vectors, matrices, etc.
- The shared mesh partition uses a more straightforward approach. Although each rank only writes to its own partition, it keeps a copy of the entire mesh, so it is aware of all the information in the domain. A distributed mesh with a large ghost layer covers all nodes and elements that are not owned locally. On each rank, it uses more memory than the distributed mesh. The mesh should not cause any problems if

it is not too large to be a bottleneck. Figure (5.14) illustrates the difference between distributed mesh and shared mesh.

In order to make communication between processors we will use MPI.

5.13.2 Distributed Triangulation

On any particular processor, a deal.II mesh has different kinds of cells see the Figure (5.15):

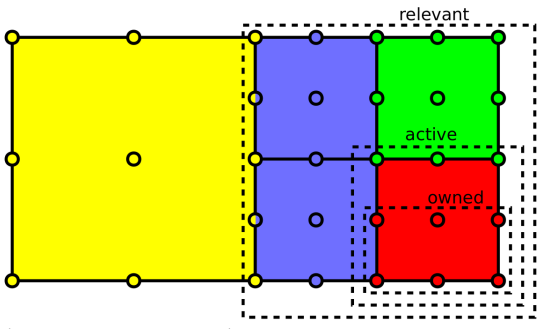


Figure 5.15: Types of cells in parallelization [19].

- An active cell is one that does not have children. The entire domain is covered by active cells. Active cells are leaves of the global distributed forest that forms the mesh if they belong to a part of it owned by the current processor. Then it is referred to as a locally owned active cell.
- Ghost cells are active cells represents leaves of the distributed forest that are adjacent to active cells that are locally owned.
- A cell that is artificial is an active cell that is neither locally owned nor ghostly. In deal.II, hanging nodes will never be stored more than once per face or edge, and all common coarse mesh cells will be stored. Every algorithm inside deal.II skips artificial cells, even if they correspond to leaves of the distributed forest.
- Cells that have children are considered non-active. Between coarse mesh cells and active cells, deal.II stores all intermediate cells.

In deal.II MPI parallelization is done using external library line p4est, PETSC and Trillions (see Figure (5.16)) as follows:

- For mesh parallelization deal.II is built on p4est. The p4est library decomposes this "real" mesh into pieces that are stored by the various processes. The p4est data structure stores the entire mesh as a parallel forest. Parallel forests consist of quad-trees (in 2d) or oct-trees (in 3d), which represent the refinement structure from

parent cells to their four (2d) or eight (3d) children. Parallel forests are internally represented by (distributed) linear arrays of cells that correspond to depth-first traversals of trees, with each process storing a section of this linear array. The Triangulation class is based on this, and further degrees of freedom are built on it.

- The parallel functionality of PETSc and Trilinos is based on the Message Passing Interface (MPI). In MPI, communication is based on collective communication: if one process wants something from another, the other must accept this communication. It contains several subroutines that help with assembly and the degree of freedom of vectors and matrices. It also contains a number of parallel solvers that can be used to solve the system in parallel.

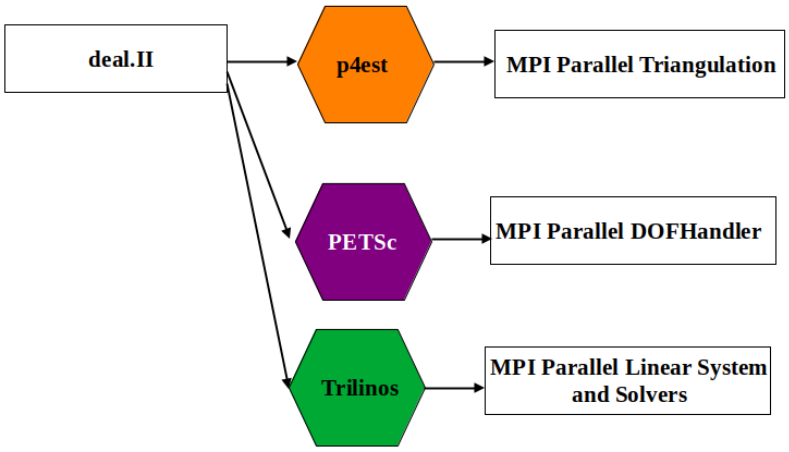


Figure 5.16: MPI parallelization of code the color code hexagon is for external parallel library which deal.II. uses for parallelization.

5.13.3 New Feature implementation

In deal.II, **Konrad Simon** developed a new feature to identify the processor to which the point belongs. This is when the mesh is traced back using semi-Lagrangian methods. It is possible to find the MPI ranks of cells containing specified points by using the newly added member function. find_point_owner_rank() of parallel::distributed::Triangulation as shown in Listing (21). This functionality is based on p4est (>version 2.2) and is communication-free. This implementation was merged and released in deal.II version 9.4 in 24 June 2022.

```
std::vector<types::subdomain_id> Triangulation<dim, spacedim>::find_point_owner_rank(const std::vector<Point<dim>> &points)
```

Listing 21: New Feature find point owner rank.

5.14 Parallel code implementation of advection-diffusion equation solution with FEM method in deal.II

Here are the steps for FEM parallelization

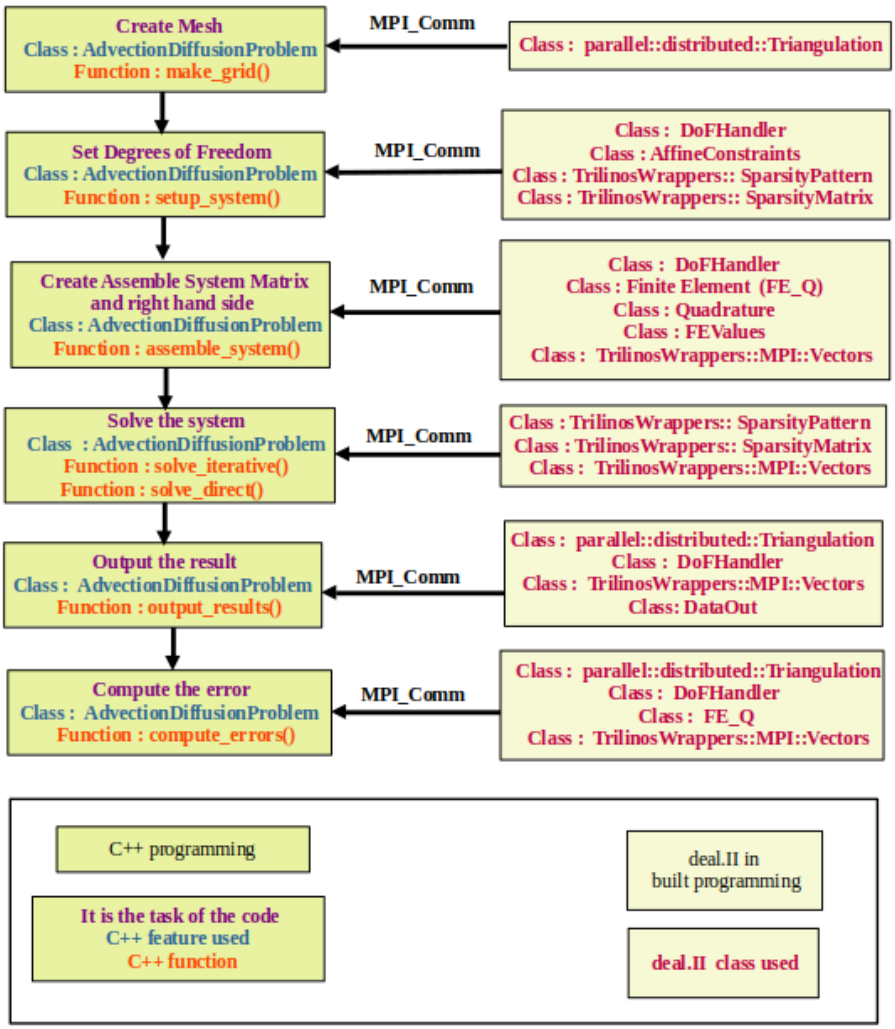


Figure 5.17: Flow chart of advection diffusion equation solution parallel code in **advectiondiffusion_problem.hpp** file.

Figure (5.17) gives the summary of the parallel code implementation of the solution of the advection diffusion equation. The reader can see the **advectiondiffusion_problem.hpp** header file the whole implementation in Github `https://github.com/heena008/Advection-Diffusion_MsFEM.io`.

For **AdvectionDiffusionProblem** class we have defined before in previous section adding an extra prefix to existing deal.II in order to run in parallel. Following lines describe the changes needed to the existing file.

- **MPI_Comm mpi_communicator** is used for communication between different processors.
- **ConditionalOStream pcout** is used in parallel output.
- **TimerOutput computing_timer** is used for time measurements of different subsections in a program.
- parallel::distributed:: is used as prefix to deal.II **triangulation**.
- TrilinosWrappers::MPI:: is used as prefix to deal.II **Vectors**.
- TrilinosWrappers:: is used as prefix to deal.II **SparsityPattern**.
- TrilinosWrappers:: is used as prefix to deal.II **SparseMatrix** .

Listing (22) gives the changes discussed above, along with a few constructors, destructors, functions and the object functions defined in the **AdvectionDiffusionProblem** class.

```
template <int dim> class AdvectionDiffusionProblem {
public:
  /*!
  * Standard constructor disabled.
  */
  AdvectionDiffusionProblem() = delete;
  /*!
  * Default constructor.
  */
  AdvectionDiffusionProblem(unsigned int n_refine, bool is_periodic);
  /*!
  * Destructor.
  */
  ~AdvectionDiffusionProblem();
  /*!
  * @brief Write results to disk.
  *
  * Write results to disk in vtu-format.
  */
  void output_results(TrilinosWrappers::MPI::Vector& vector_out) const;

  void compute_errors();

  MPI_Comm mpi_communicator;
  /*!
  * Distributed triangulation
  */
  parallel::distributed::Triangulation<dim> triangulation;

  FE_Q<dim> fe;
  DoFHandler<dim> dof_handler;
```

```
  /*!
  * Index Set
  */
  IndexSet locally_owned_dofs;
  IndexSet locally_relevant_dofs;

  TrilinosWrappers::SparsityPattern sparsity_pattern;

  TrilinosWrappers::SparseMatrix system_matrix;

  TrilinosWrappers::MPI::Vector solution;
  TrilinosWrappers::MPI::Vector old_solution;
  TrilinosWrappers::MPI::Vector system_rhs;

  ConditionalOStream pcout;
  TimerOutput computing_timer;

};
```

Listing 22: Parallel Advection diffusion class define constructor and destructor.

Listing (23) shows the initialization of an object function. This is a list of all processors assigned to this job by the batch scheduling system, such as **MPI_COMM_WORLD**. It is further taken as a variable in **mpi_communicator**, which is then used in deal.II class steps, computing time, and parallelizing.

```
template <int dim>
AdvectionDiffusionProblem<dim>::AdvectionDiffusionProblem(unsigned int n_refine,
bool is_periodic)
: mpi_communicator(MPI_COMM_WORLD)
, triangulation(mpi_communicator,
typename Triangulation<dim>::MeshSmoothing(
Triangulation<dim>::smoothing_on_refinement |
Triangulation<dim>::smoothing_on_coarsening))
, fe(1)
, dof_handler(triangulation)
, pcout(std::cout, (Utilities::MPI::this_mpi_process(mpi_communicator) == 0))
, computing_timer(mpi_communicator,
pcout,
TimerOutput::summary,
TimerOutput::wall_times)
{}
```

Listing 23: Parallel advection diffusion initialize object function.

5.14.1 Step 1 : Create Mesh

Set up mesh in different processors using parallel::distributed::Triangulation.

- In parallel distributed mode, the parallel::distributed::Triangulation object only

stores a subset of cells on each processor. Specifically, each MPI process owns a number of cells in the global mesh.

- It contains only the cells it owns locally, as well as one layer of ghost cells around them. It also contains a number of artificial cells. These cells ensure that each processor has a mesh that includes all coarse level cells. This mesh respects the invariant that neighboring cells cannot differ by more than one refinement level.

Listing (24) gives the lines in code required to achieve parallel triangulation with deal.II. Here make_grid() function is used in AdvectionDiffusionProblem<dim> class for the **main code** and deal.II function parallel::distributed::Triangulation is used to create a domain.

```
template <int dim>
void AdvectionDiffusionProblem<dim>::make_grid()
{
  TimerOutput::Scope t(computing_timer, "mesh generation");
    parallel::distributed::Triangulation triangulation;
```

Listing 24: Parallel Distributed Triangulation.

5.14.2 Step 2 : Set Degrees of Freedom

Distribute degrees of freedom with the DOFHandler.

- We assign DoF indices to all the degrees of freedom defined on all the cells we own locally and, in the case of subdomains that are owned by different processors, that are not owned by the neighboring processor, to the degrees of freedom defined on them.
- Each processor then exchanges how many degrees of freedom it owns locally and shifts its own index so that all degrees of freedom on all subdomains are uniquely identified by an index between zero and DoFHandler::n_dofs() (returns the global number of degrees of freedom, accumulated across all processors). Once this step is completed, each process's degrees of freedom will form a contiguous range, which can be obtained by calling DoFHandler::locally_owned_dofs().
- All degrees of freedom are assigned unique indices by the DoFHandler::distribute_dofs() function, which then loops over all ghost cells and communicates with neighboring processors to ensure that these ghost cells have global indexes of degrees of freedom that correspond to those assigned by their neighbors.

Listing (25) gives the lines in code that are required to achieve parallel degrees of freedom with deal.II.

```
dof_handler.distribute_dofs(fe);
```

Listing 25: Parallel Degrees of Freedom.

Extract the locally owned DOF with the DOFHandler.

Next, two index sets are extracted from the following lines.

- These include one that indicates the degrees of freedom owned by the current processor. This information will be used to initialize the solution and right-hand side vectors, as well as the system matrix, which will indicate which elements to store on the current processor and which to expect to be stored elsewhere.
- And another that indicates which degrees of freedom are locally relevant. For example, we need these degrees of freedom to estimate the error on the local cells if they live on cells that the current processor owns, or on the layer of ghost cells around these cells.

Listing (26) gives the lines in code that are required to distribute parallel degrees of freedom into different categories in deal.II.

```
locally_owned_dofs = dof_handler.locally_owned_dofs();
DoFTools::extract_locally_relevant_dofs(dof_handler, locally_relevant_dofs);
```

Listing 26: Parallel Degrees of Freedom distributions of cells.

Initialize solution and right hand side.

Let's now initialize the solution and the right-hand side vectors. Solution vectors do not only store elements we own, but also ghost entries, as previously mentioned; on the other hand, the right hand side vector does not need to have entries owned by the current processor because we will only ever write into it, never read from it on cells owned locally. Listing (27) gives the lines in code required to initialize solution and right hand side in deal.II.

```
solution.reinit(locally_owned_dofs, locally_relevant_dofs, mpi_communicator);
old_solution.reinit(locally_owned_dofs, locally_relevant_dofs, mpi_communicator);
system_rhs.reinit(locally_owned_dofs, mpi_communicator);
```

Listing 27: Parallel Degrees of Freedom initialize solution and right hand side.

Compute boundary value constraints.

Our next step is to compute boundary value constraints, which we then combine into one object.

- Parallel processing must be based on the mantra that no processor can store all information about the entire universe. We must therefore specify for which degrees of freedom the AffineConstraints object can store constraints, and for which it should not expect to store any.
- For our case, we need to care about local degrees of freedom, so we pass this to the AffineConstraints::reinit function. AffineConstraints will allocate a length equal to the largest DoF index it has seen if you forget to pass this argument.

```
constraints.clear();
constraints.reinit(locally_relevant_dofs);
DoFTools::make_hanging_node_constraints(dof_handler, constraints);

const Coefficients::DirichletBC<dim> dirichlet_bc;
  for (unsigned int i = 0; i < dim; ++i)
  {
    VectorTools::interpolate_boundary_values(dof_handler,/*boundary id*/2*i, // only even boundary id
    dirichlet_bc, constraints);
  }

  constraints.close();
```

Listing 28: Parallel affine constraints.

Initializes the matrix and sparsity pattern.

The last part of this function initializes the matrix and sparsity pattern.
As an intermediate, we use SparsityPattern to initialize the system matrix.

- DoFTools::make_sparsity_pattern fills the sparsity pattern based on its size and what DoFs we need to add items to (this function ignores all cells that are not locally owned).
- In this way, each processor will have a complete picture of all entries that will exist in that part of the finite element matrix that it will own after we call the function that exchanges sparsity pattern entries.

- Finally, we need to initialize the matrix with the sparsity pattern.

```
sparsity_pattern.reinit(locally_owned_dofs, mpi_communicator);
DoFTools::make_sparsity_pattern(dof_handler,sparsity_pattern,constraints,
/* keep_constrained_dofs */ true,Utilities::MPI::this_mpi_process(mpi_communicator));
sparsity_pattern.compress();

system_matrix.reinit(sparsity_pattern);

solution.reinit(locally_owned_dofs, locally_relevant_dofs, mpi_communicator);

old_solution.reinit(locally_owned_dofs, locally_relevant_dofs,
mpi_communicator);
system_rhs.reinit(locally_owned_dofs, mpi_communicator);
```

Listing 29: Parallel initializes the matrix and sparsity pattern.

5.14.3 Step 3 : Assemble the system matrix and right hand side

A similar function to the one we have seen before assembles the linear system. Here are some points to keep in mind:

- The assembly must only loop over cells that are owned locally. This can be tested in several ways; for example, we could compare the **subdomain_id** of a cell with the information from the triangulation **(cell->subdomain_id() == triangulation.locally_owned_subdomain())**, or skip all cells where the condition **cell->is_ghost()** || **cell->is_artificial()** is true. To determine whether a cell belongs to a local processor, simply ask it.
- In order to copy local contributions into the global matrix, boundaries and constraints must be distributed. Therefore, we cannot copy every local contribution to the global matrix first, and then handle constraints and boundary values later. Since the parallel vector classes have been assembled into a matrix, arbitrary elements cannot be accessed once they have been assembled in part because they may simply no longer reside on the current processor but have been shipped to a different machine.

```
const QGauss<dim> quadrature_formula(fe.degree + 1);
const QGauss<dim − 1> face_quadrature_formula(fe.degree + 1);

FEValues<dim> fe_values(fe,quadrature_formula,
update_values | update_gradients |update_quadrature_points | update_JxW_values);

FEFaceValues<dim> fe_face_values(fe,face_quadrature_formula,update_values | update_quadrature_points |
 update_normal_vectors |update_JxW_values); // for Neumaan boundary condition to evaluate
// boundary condition
```

```
for (unsigned int q_index = 0; q_index < n_q_points; ++q_index)
{ for (unsigned int i = 0; i < dofs_per_cell; ++i)
  { for (unsigned int j = 0; j < dofs_per_cell; ++j)
    {
      // Diffusion is on rhs. Careful with signs here.
      cell_matrix(i, j) +=(fe_values.shape_value(i, q_index) *fe_values.shape_value(j, q_index) +
      time_step * (theta) *(fe_values.shape_grad(i, q_index) *matrix_coeff_values[q_index] *
      fe_values.shape_grad(j, q_index) +fe_values.shape_value(i, q_index) * advection_field_values[q_index] *
      fe_values.shape_grad(j, q_index))) *fe_values.JxW(q_index);
      // Careful with signs also here.
      cell_rhs(i) += (fe_values.shape_value(i, q_index) *fe_values.shape_value(j, q_index) -
      time_step * (1 - theta) * (fe_values.shape_grad(i, q_index) *matrix_coeff_values_old[q_index] *
      fe_values.shape_grad(j, q_index) +fe_values.shape_value(i, q_index) *advection_field_values_old[q_index] *
      fe_values.shape_grad(j, q_index))) *fe_values.JxW(q_index) *old_solution(local_dof_indices[j]);
    } // end ++j

    cell_rhs(i) += time_step * fe_values.shape_value(i, q_index) * ((1 - theta) * rhs_values_old[q_index] +
    (theta)*rhs_values[q_index]) * fe_values.JxW(q_index);
  } // end ++i
} // end ++q_index
```

Listing 30: Parallel assemble the system matrix and right hand side.

The assembling in Listing (30) is basically a local operation. A synchronization between all processors is therefore required to form a "global" linear system. By invoking **compress()**, this can be accomplished see Listing (35).

```
/*For the "global" linear system to work, all processors must be synchronized.
/* By calling compress(), this can be accomplished. More information about compress() can be found in
/* @ref GlossCompress "Compressing distributed objects". */
system_matrix.compress(VectorOperation::add);
system_rhs.compress(VectorOperation::add);
```

Listing 31: Parallel compress.

5.14.4 Step 4 : Solve the system

Solve the system with different parallel solvers and preconditioner.

There are two key things to consider when solving linear systems on tens of thousands of processors:

- Solvers and preconditioners are provided by the deal.II wrappers of PETSc and **Trilinos**. When building massively parallel linear solvers, scalability of preconditioners is more of a concern than communication between processors.
- A vector should ultimately store not only the degrees of freedom that the current processor owns, but also all other locally relevant degrees of freedom. Conversely,

the solver itself needs a vector without overlap between processors. To solve the linear system, we create a vector that has these properties at the beginning of the function, and we assign it to the vector we want at the end. In this final step, all ghost elements are also copied.

```
template <int dim>
void AdvectionDiffusionProblem<dim>::solve_direct()
{
  TimerOutput::Scope t(computing_timer,"parallel sparse direct solver (MUMPS)");

  TrilinosWrappers::MPI::Vector completely_distributed_solution(locally_owned_dofs, mpi_communicator);

  SolverControl solver_control;
  TrilinosWrappers::SolverDirect solver(solver_control);
  solver.initialize(system_matrix);

  solver.solve(system_matrix, completely_distributed_solution, system_rhs);

  pcout << " Solved in with direct solver." << std::endl;

  constraints.distribute(completely_distributed_solution);

  solution = completely_distributed_solution;
}

template <int dim>
void AdvectionDiffusionProblem<dim>::solve_iterative()
{
  TimerOutput::Scope t(computing_timer, "iterative solver");

  TrilinosWrappers::MPI::Vector completely_distributed_solution(locally_owned_dofs, mpi_communicator);

  SolverControl solver_control( /* max number of iterations */ dof_handler.n_dofs(),
  /* tolerance */ 1e-7, /* print log history */ true);

  TrilinosWrappers::SolverGMRES gmres_solver(solver_control);

  TrilinosWrappers::PreconditionIdentity preconditioner;
  TrilinosWrappers::PreconditionIdentity::AdditionalData data;

  preconditioner.initialize(system_matrix, data);

  gmres_solver.solve(system_matrix,completely_distributed_solution, system_rhs, preconditioner);

  pcout << " Solved in " << solver_control.last_step() << " iterations (theta = " << theta << ")." << std::endl;

  constraints.distribute(completely_distributed_solution);

  solution = completely_distributed_solution;
}
```

Listing 32: Parallel solve the system.

5.14.5 Step 5 : Output the result

Collect output from all processor. High-performance, parallel **MPI-IO (I=Input O=Output)** routines are used to write to a small, fixed number of visualization files. Additionally, a .pvtu record is generated, which can be viewed directly in visualization tools such as paraview and visit.

- The data vector that stores the subdomain of each cell is attached along with the solution vector (the one that contains entries for all locally relevant elements, not only those owned locally). It's a bit tricky because not every processor knows about every cell.
- DataOut will ignore all entries that correspond to cells that are not owned by the current processor in the vector we attach (locally owned cells, ghost cells, and artificial cells).
- Consequently, it doesn't matter what values we write into these vector entries: we simply fill them with the current **MPI** process number (i.e. the **subdomain_id**); this sets the values we care about correctly, namely those that correspond to locally owned cells, while providing the wrong values for all other elements but these are ignored anyway.

```
template <int dim>
void AdvectionDiffusionProblem<dim>::output_results(TrilinosWrappers::MPI::Vector &vector_out) const
  {
    std::string filename = (dim == 2 ? "solution-std_2d" : "solution-std_3d");
    DataOut<dim> data_out;
    data_out.attach_dof_handler(dof_handler);
    data_out.add_data_vector(vector_out, "u");

    Vector<float> subdomain(triangulation.n_active_cells());
    for (unsigned int i = 0; i < subdomain.size(); ++i)
    {
      subdomain(i) = triangulation.locally_owned_subdomain();
    }
    data_out.add_data_vector(subdomain, "subdomain");

    data_out.build_patches();
```

Listing 33: Parallel output the result.

The final step is to write the data to disk. **MPI-IO** allows us to write up to total number of time steps .vtu files in parallel. In addition, a .pvtu record is generated, which groups the .vtu files that have been written.

```
if (Utilities::MPI::this_mpi_process(mpi_communicator) == 0)
```

```
{
  std::vector<std::string> file_list;
  for (unsigned int i = 0; i < Utilities::MPI::n_mpi_processes(mpi_communicator) ++i)
  {
    file_list.push_back(filename + "_refinements−" + Utilities::int_to_string(n_refine, 1) +"." +
     "theta−" + Utilities::to_string(theta, 4) + "." + "time_step−" +
     Utilities::int_to_string(timestep_number, 4) + "." + Utilities::int_to_string(i, 4) + ".vtu");
  }

  std::string filename_master(filename);

  filename_master += "_refinements−" + Utilities::int_to_string(n_refine, 1) + "." +
  "theta−" + Utilities::to_string(theta, 4) + "." + "time_step−" +
  Utilities::int_to_string(timestep_number, 4) + ".pvtu";

  std::ofstream master_output(filename_master);
  data_out.write_pvtu_record(master_output, file_list);
}
```

Listing 34: Write the output the result.

5.14.6 Step 6 : Compute the error

As part of parallel::TriangulationBase, entries in **cellwise_error** that do not correspond to locally owned cells are assumed to be 0.0 and a parallel reduction is performed using MPI to calculate the global error.

Next, we calculate the error in the L2 norm, HI norm and Linfty norm on each cell using a function from the library. The DoF handler object, the vector holding the numerical solution nodal values, the continuous solution as a function object, the vector into which the error norms of each cell will be placed, a quadrature rule by which they will be computed, and the type of norms that will be used must be passed to it. Using a Gauss formula with three points in each space direction, we compute the L2 norm.

Finally, we need the global L2 norm, HI norm and Linfty norm. The square root of the norm on each cell can be obtained by summing the squares of the norms on those cells. Each cell's l2 norm (lower case l) corresponds to the following:

```
template <int dim>
void AdvectionDiffusionProblem<dim>::compute_errors()
{
    Vector<float> difference_per_cell(triangulation.n_active_cells());
    VectorTools::integrate_difference(dof_handler, solution, ZeroFunction<dim>(),
    difference_per_cell, QGauss<dim>(fe.degree + 1),VectorTools::L2_norm);

    const double L2_error = VectorTools::compute_global_error(triangulation, difference_per_cell,VectorTools::L2_norm);

    const unsigned int n_active_cells = triangulation.n_active_cells();
```

```
    const unsigned int n_dofs = dof_handler.n_dofs();
    VectorTools::integrate_difference(dof_handler,solution, ZeroFunction<dim>(),
    difference_per_cell,QGauss<dim>(fe.degree + 1), VectorTools::H1_norm);

    const double H1_error = VectorTools::compute_global_error(triangulation,
    difference_per_cell, VectorTools::H1_norm);

    VectorTools::integrate_difference(dof_handler, solution,
    ZeroFunction<dim>(), difference_per_cell, QGauss<dim>(fe.degree + 1),
    VectorTools::Linfty_norm);
    const double Linfty_error = VectorTools::compute_global_error(triangulation,
    difference_per_cell,VectorTools::Linfty_norm);
}
```

Listing 35: Parallel compute the error.

Note : In the example we discussed above for advection diffusion equation parallel code, the code for the diffusion equation solution with the standard finite element method is more simple. To understand clear code, readers can follow the steps above and the Flow charts (5.18) and (5.19) below.

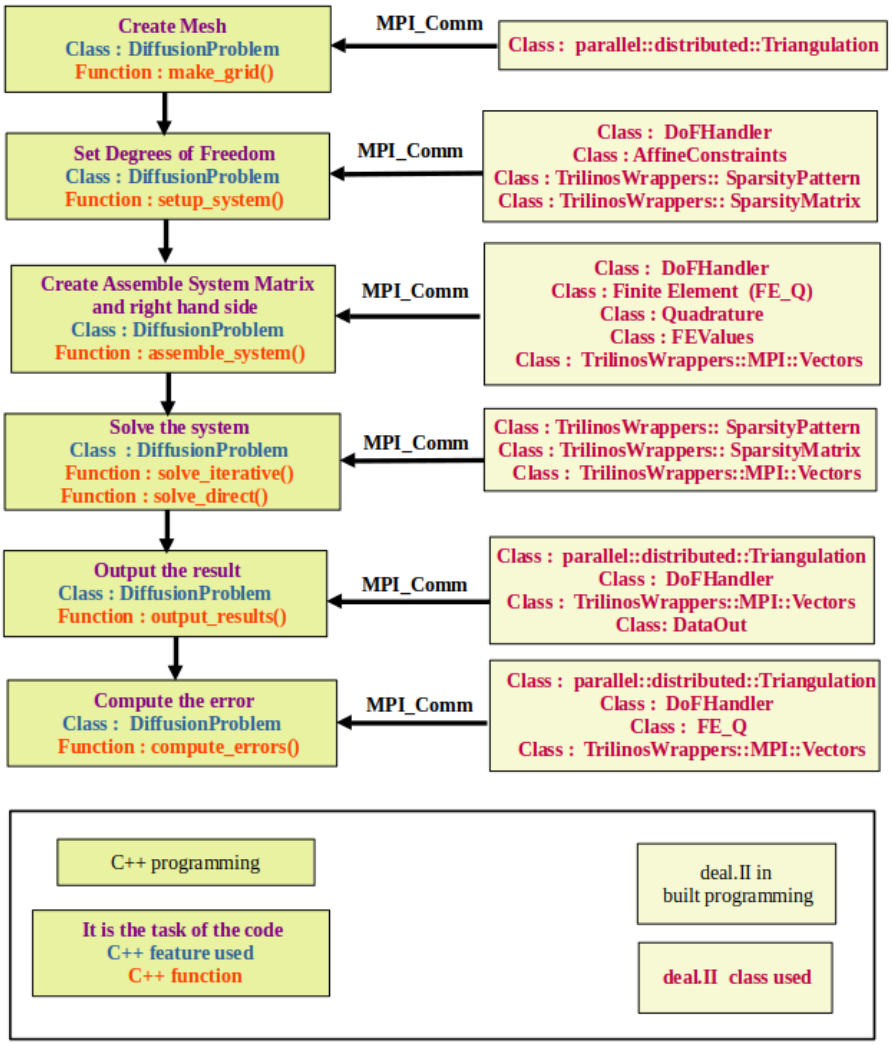


Figure 5.18: Flow chart of diffusion equation finite element solution code main code in **diffusion_problem.hpp** file.

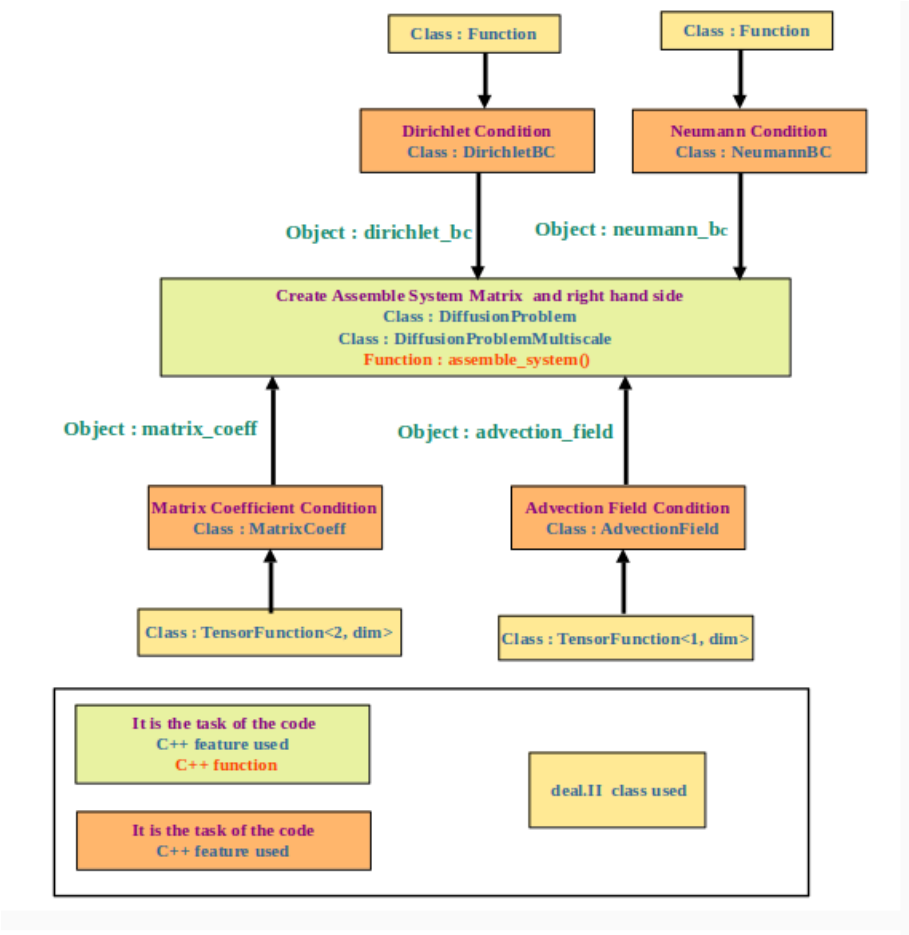


Figure 5.19: Flow chart of boundary condition code connected to main file **diffusion_problem.hpp** file.

5.15 Implementation of diffusion equation solution with MsFEM method in deal.II

The diffusion equation solution with multiscale finite element code follows the method explained in Chapter 3. We discussed the steps for solving the problem in a Flow chart (3.11). Now to make it simple we split the code into two major parts or code files.

- The first part is called the global formulation, which is the **main code**. See the Flow chart (5.20). It follows all the steps we discussed for the advection-diffusion equation.
- Multiscale basis function which is the **basis code**, as shown in the Flow chart (5.22). Further Flow chart (5.23) explains connection between **main code** and **basis code**.

5.15.1 Global formulation code

Now for global formulation in Flow chart (5.20) the reader knows most steps we discussed in the advection-diffusion finite element solution. Boundary condition in Flow chart (5.19) and coefficients are similar to those discussed in the previous section.

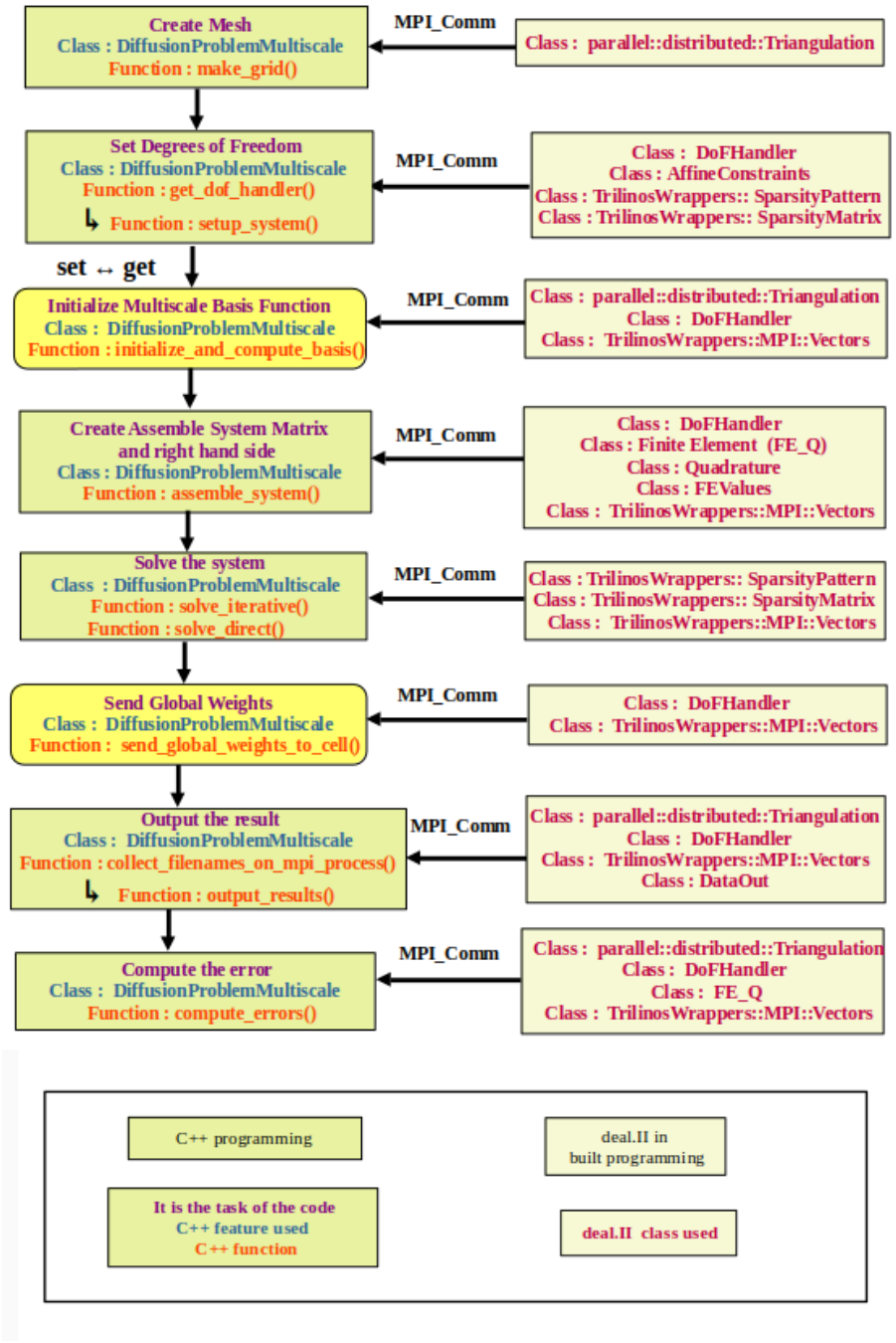


Figure 5.20: Flow chart of diffusion equation multiscale finite element solution parallel main code.

The multiscale basis is implemented in header file **diffusion_basis.hpp** are introduced in **diffusionproblem_multiscale** file to get access to multiscale basis in **main code**. The

Listing (36) shows the code with three new functions added to the existing code, followed by a function for local mesh refinement and map to connect the local basis and global cell.

```
template <int dim>
class DiffusionProblemMultiscale
{
  public:
  /*!
  * Constructor.
  */
  DiffusionProblemMultiscale(unsigned int n_refine, unsigned int n_refine_local);
  /*!
  * @brief Run function of the object.
  *
  * Run the computation after object is built.
  */
  void run();

  private:
  /*!
  * @brief Set up the grid with a certain number of refinements.
  *
  * Generate a triangulation of \f$[0,1]^{\rm{dim}}\f$ with edges/faces
  * numbered form \f$1,\dots,2\rm{dim}\f$.
  */
  void make_grid();

  /*!
  * Set all relevant data to local basis object and initialize the basis
  * fully. Then compute.
  */
  void initialize_and_compute_basis();

  /*!
  * @brief Setup sparsity pattern and system matrix.
  *
  * Compute sparsity pattern and reserve memory for the sparse system matrix
  * and a number of right-hand side vectors. Also build a constraint object
  * to take care of Dirichlet boundary conditions.
  */
  void setup_system();

  /*!
  * @brief Assemble the system matrix and the static right hand side.
  *
  * Assembly routine to build the time-independent (static) part.
  * Neumann boundary conditions will be put on edges/faces
  * with odd number. Constraints are not applied here yet.
  */
  void assemble_system();

  /*!
  * @brief Iterative solver.
  *
  * CG-based solver with AMG-preconditioning.
```

```
  */
  void solve_iterative();

  /*!
  * @brief Send coarse weights to corresponding local cell.
  *
  * After the coarse (global) weights have been computed they
  * must be set to the local basis object and stored there.
  * This is necessary to write the local multiscale solution.
  */
  void send_global_weights_to_cell();

  /*!
  * @brief Write coarse solution to disk.
  *
  * Write results for coarse solution to disk in vtu-format.
  */
  void output_result() const;

  /*!
  * Collect local file names on all mpi processes to write
  * the global pvtu-record.
  */
  std::vector<std::string> collect_filenames_on_mpi_process() const;

  /*!
  * Number of local refinements.
  */
  const unsigned int n_refine_local;

  /*!
  * STL Vector holding basis functions for each coarse cell.
  */
  std::map<CellId, DiffusionProblemBasis<dim>> cell_basis_map;
};
```

Listing 36: Diffusion multiscale basis class.

We will now focus on the three new steps for connecting global formulation to multiscale scale basis in **main code**.

Initialize the basis

In the following Listing (37) is used to pair multiscale basis from local cell to global cell using C++ std:: map and then it is solved on each basis using **run()** function from basis code for multiscale basis function.

```
 template <int dim>
void
DiffusionProblemMultiscale<dim>::initialize_and_compute_basis()
{
  TimerOutput::Scope t(computing_timer,"basis initialization and computation");
```

```
  typename Triangulation<dim>::active_cell_iterator cell = dof_handler.begin_active(),endc = dof_handler.end();
  for (; cell != endc; ++cell)
  {
    if (cell->is_locally_owned())
    {
      DiffusionProblemBasis<dim> current_cell_problem(n_refine_local, cell,triangulation.locally_owned_subdomain(),
      mpi_communicator);
      CellId current_cell_id(cell->id());

      std::pair< typename std::map<CellId, DiffusionProblemBasis<dim>>::iterator, bool> result;
      result = cell_basis_map.insert(std::make_pair(cell->id(), current_cell_problem));

      Assert(result.second, ExcMessage( "Insertion of local basis problem into std::map failed. "
        "Problem with copy constructor?"));
    }
  } // end ++cell
  /*
  * Now each node possesses a set of basis objects.
  * We need to compute them on each node and do so in
  * a locally threaded way.
  */
  typename std::map<CellId, DiffusionProblemBasis<dim>>::iterator
  it_basis = cell_basis_map.begin(), it_endbasis = cell_basis_map.end();
  for (; it_basis != it_endbasis; ++it_basis)
  {
    (it_basis->second).run();
  }

}
```

Listing 37: Compute the error for parallel code.

Send global weights to cell

Listing (38) shows code to connect global degrees of freedom to local degrees of freedom for each global cell. Further **extracted_weights** vector is created in order use to add with local basis solution to global solution.

```
template <int dim>
void DiffusionProblemMultiscale<dim>::send_global_weights_to_cell()
  {
    // For each cell we get dofs_per_cell values
    const unsigned int dofs_per_cell = fe.dofs_per_cell;
    std::vector<types::global_dof_index> local_dof_indices(dofs_per_cell);

    // active cell iterator
    typename DoFHandler<dim>::active_cell_iterator cell =dof_handler.begin_active(), endc = dof_handler.end();
    for (; cell != endc; ++cell)
    {
      if (cell->is_locally_owned())
      {
```

```
            cell->get_dof_indices(local_dof_indices);
            std::vector<double> extracted_weights(dofs_per_cell, 0);
            solution.extract_subvector_to(local_dof_indices,extracted_weights);

            typename std::map<CellId, DiffusionProblemBasis<dim>>::iterator
            it_basis = cell_basis_map.find(cell->id());
            (it_basis->second).set_global_weights(extracted_weights);
          }
        } // end ++cell
    }
```

Listing 38: Send global weights to cell.

Collect filenames on MPI process

Listing (39) shows the code for connecting all multiscale basis from each cell in given processor to global cell. It uses C++ feature set and get so **set_filename_global** and **get_filename_global** is defined in basis code and **get_filename_global** is used in main code to make file list of all local basis on global cell.

```
template <int dim>
std::vector<std::string> DiffusionProblemMultiscale<dim>::collect_filenames_on_mpi_process() const
{
  std::vector<std::string> filename_list;

  typename std::map<CellId, DiffusionProblemBasis<dim>>::const_iterator
  it_basis = cell_basis_map.begin(),
  it_endbasis = cell_basis_map.end();

  for (; it_basis != it_endbasis; ++it_basis)
  {
    filename_list.push_back((it_basis->second).get_filename_global());
  }

  return filename_list;
}
```

Listing 39: Collection of solution from all processor.

5.15.2 Multiscale basis function code

Multiscale basis function code follows similar steps, but here it is necessary to define the Q1 basis function first in a header file named **q1bais.hpp**, which is then applied to **advectiondiffusion_basis.hpp** file. The header file of Q1 basis file is then called in **advectiondiffusion_basis.hpp** file and is used by defining object function **basis_q1** for class **Q1Basis**.

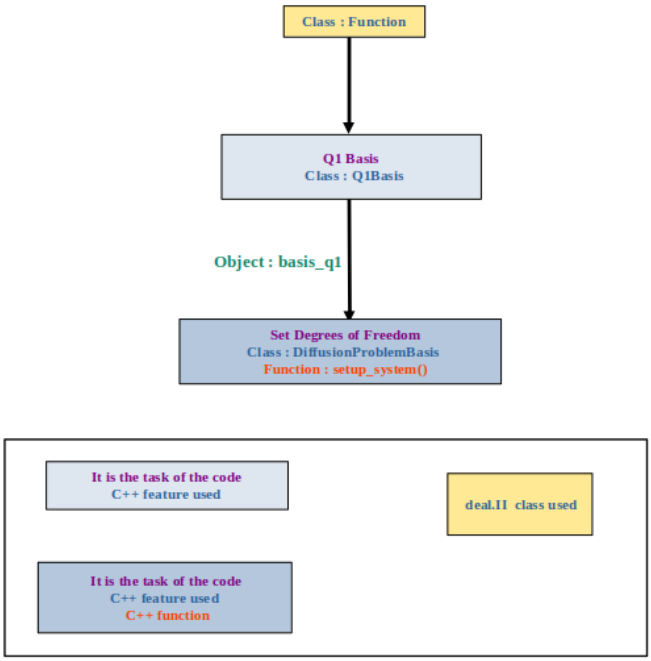


Figure 5.21: Flow chart of Q1 Basis function in multiscale basis code.

Next, we will explain following steps for solving multiscale basis functions in **basis code** file as shown in Flow chart (5.22).

- the **run** function is created and all the following function are called in it. As in C++ run is used to call the function.
- In **make_grid** function, with the help of Triangulation domain is created using the coordinates of global cell and then grid is generated.
- In **setup_system** , with DOFHandler degrees of freedom are added. At the boundary Q1 Basis is added with object function **basis_q1**.

  The local system is assembled in **assemble_system** using the same process we discussed previously.
- In **assemble_global_element_matrix** is used to take all the local contribution from above **assemble_system()** function and multiplying be test function on left and trial function on right to form global assemble system with local assemble system contribution. It was discussed in Chapter 3 in section (3.5.4). This part will then be called in the global formulation file.
- In **set_filename_global** collects local solution from all the processor add to global cell. This would be called in the main code.
- In **solve_iterative** or **solve_direct** is used to solve local the system to local solution.
- In **set_global_weights** global solution is obtained by multiplying global weights

and local solution from above step.

- In **output_basis** is used to output local basis.
- In **output_global_solution_in_cell** is used to obtained global solution on each cell. This is also called in the **main code** in the global formulation

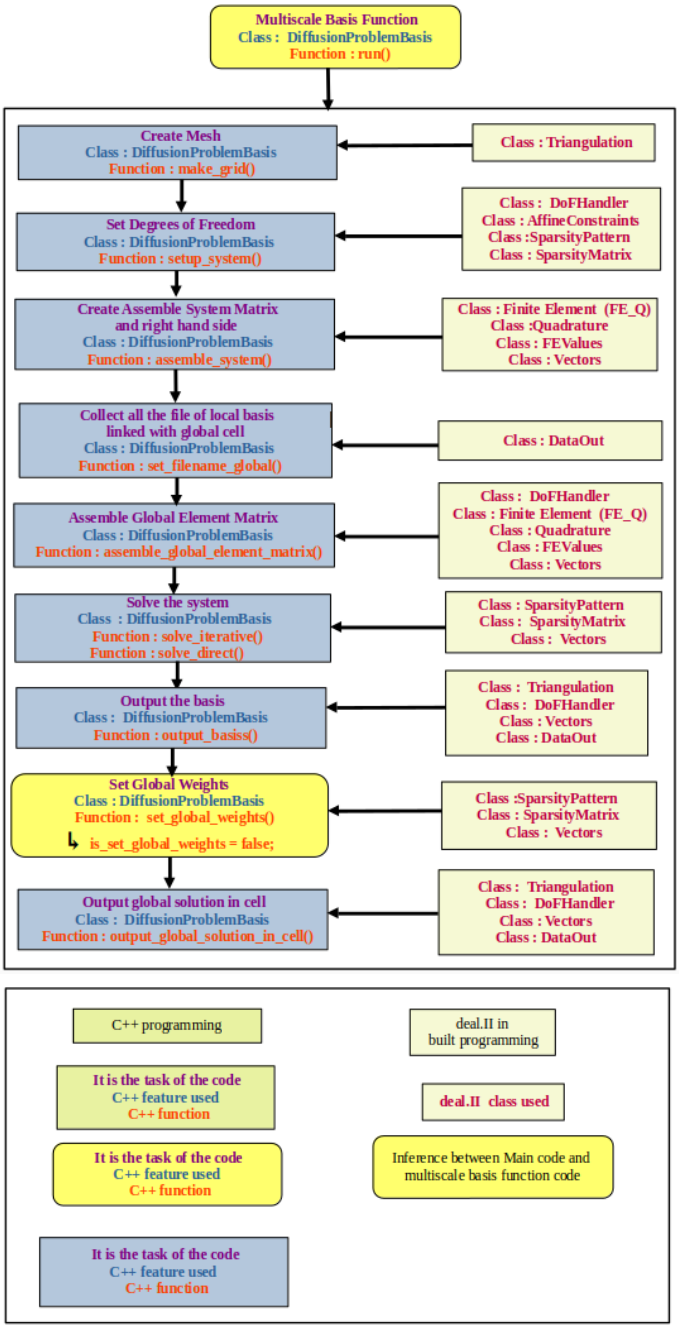


Figure 5.22: Flow chart of diffusion equation multiscale finite element basis code.

5.16 Connection between main code and basis code

Now we will see all the interconnection between the main code file and the basis code file as shown in the Flow chart (5.23).

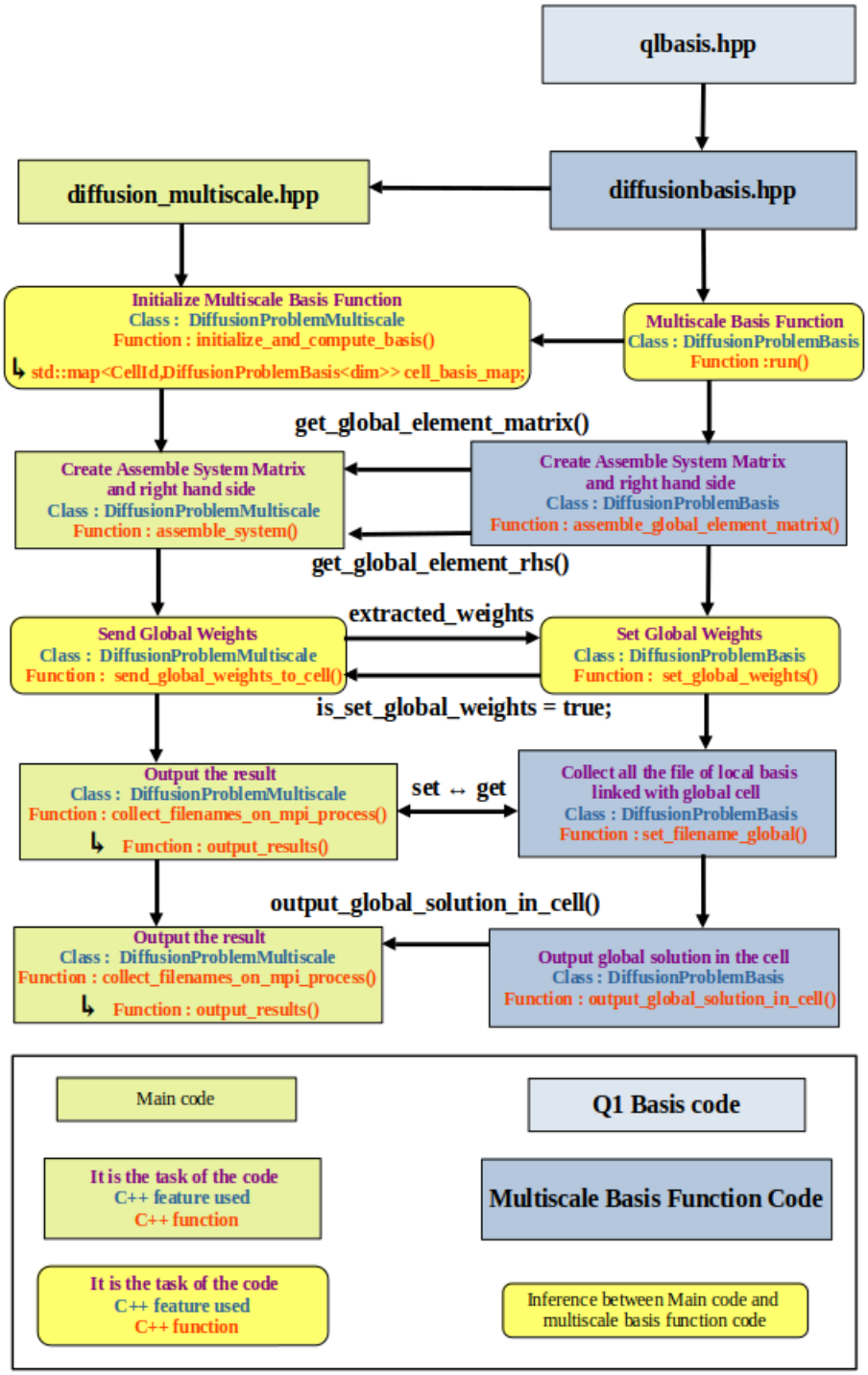


Figure 5.23: Flow chart of interface between diffusion equation multiscale finite element solution main code and multiscale basis code.

- The first inter connection is when we use the **run** function from basis code in

initialize_and_compute_basis function in main code to calculate basis on every global cell.

- From **assemble_global_element_matrix** in basis code we have global assemble matrix and global right hand side by calling **get_global_element_matrix** and **get_global_element_rhs** defined in the basis code, the system is assemble in the main code.
- In **set_global_weights** in the basis code extract local solution. Local solution is then multiplied with global weights to get a global solution. In the main code file, **extracted_weights** is defined as the input variable for **set_global_weights** function, which is applied to all local bases in a domain for the purpose to get a global solution. The theory was shown in section (3.5.2).
- In **set_filename_global** in basis code is used for collection of local solution at all processors and add then to global cell and it is called in main code as **set_filename_global** to apply it on all basis and collect it in one file.
- In **collect_filenames_on_mpi_process** collects local file names of fine mesh solution in basis code on all mpi processes to write the global pvtu-record. The C++ set $\leftrightarrow$ get is use set function is in basis code and it is called with get in main code.
- In the main code, **output_global_solution_in_cell** is called on multiscale basis to get global solution for each processor.

5.17 Implementation of advection-diffusion equation solution with Semi-Lagrangian Multiscale Finite Element in deal.II

Algorithm 4 Pseudo code for reconstruction basis algorithm in the Semi-Lagrangian Multiscale Finite Element Method in deal.II.

Step 1 : Construct a coarse grid and initialize it with multiscale basis function.
begin A coarse mesh $\mathcal{T}_H$ is to be initialized.

Step 2 : For every cell $K \in \mathcal{T}_H$ a fine mesh $\mathcal{T}_h^K$ is to be initialized.
Step 3 :
for $K \in \mathcal{T}_H$ (Online Phase). **in parallel**. **do**
Reconstruct the basis $u_0(x)|_K$.
end for
/* The time stepping begins. The solution at t_{n+1} is computed. */
Step 4 :
for $n = 0$ to $n \leq N_{steps}$ **do**
for $K \in \mathcal{T}_H$ **in parallel**. **do**
Basis Reconstruction :
(i) **Step 4a :** Each node in K is trace back one time step from t_{n+1} to t_n.
(ii) **Step 4b :** Basis $u_n(x)|_{\tilde{K}}$ from equation (4.10) to be reconstructed.
(iii) **Step 4c :**Propagate the boundary conditions of the optimal basis forward onto K.
Global Formulation: Assemble the global (coarse) system matrices
Now do a global backward Euler time step.
end for
Step 5 : Postprocess the solution.
Return
end for
end

An overview of all the files and their interconnections can be seen in Figure (5.24). Code will be divided into four main parts as follows :

1. The **main code** that resembles the global formulation. It is code for coarse mesh.
2. The multiscale basis function that is the **basis code**. It is code for local fine mesh.
3. The **interface code** for the interface between codes and
4. The semi-Lagrangian multiscale basis reconstruction code that is the **basis reconstruction code**. Reconstructed code is for the distorted cell we get after

tracing back the mesh. The mathematical derivation was discussed in Chapter 4.

The reader can see the whole implementation in Github `https://github.com/heena008/Advection-Diffusion_MsFEM.io`.

The Figure (5.24) shows the connection between all the files required for the entire solution Semi-Lagrangian basis based multiscale finite element code. In the code workflow section we discussed previously, we had seen this file in the **include** folder.

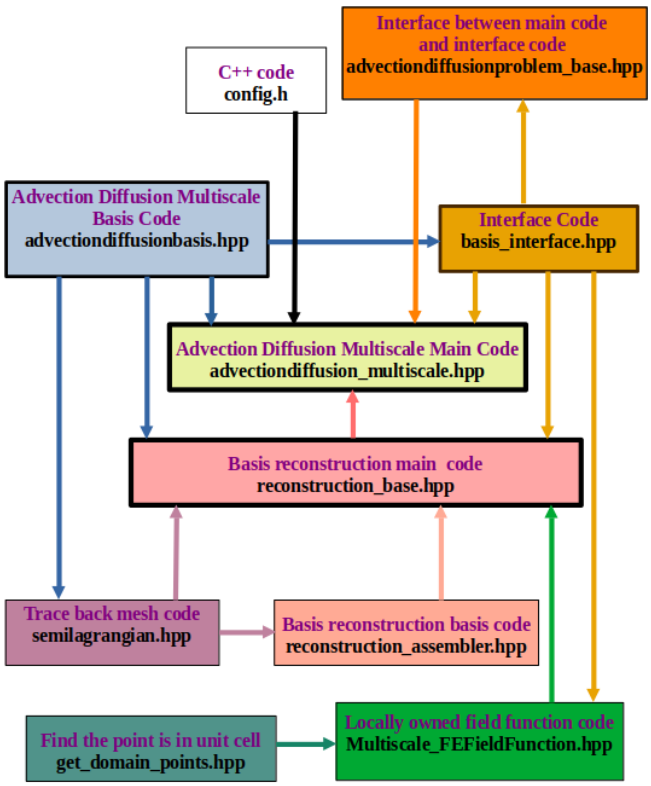


Figure 5.24: Flow chart of advection-diffusion equation multiscale finite element solution code header files connections.

Now we will connect steps of Algorithm (4) and Figure (5.24) in order to understand the whole code.

5.17.1 Step 1 : construct a coarse grid

This part come in global formulation of code.

Advection Diffusion equation multiscale finite element solution main code connections

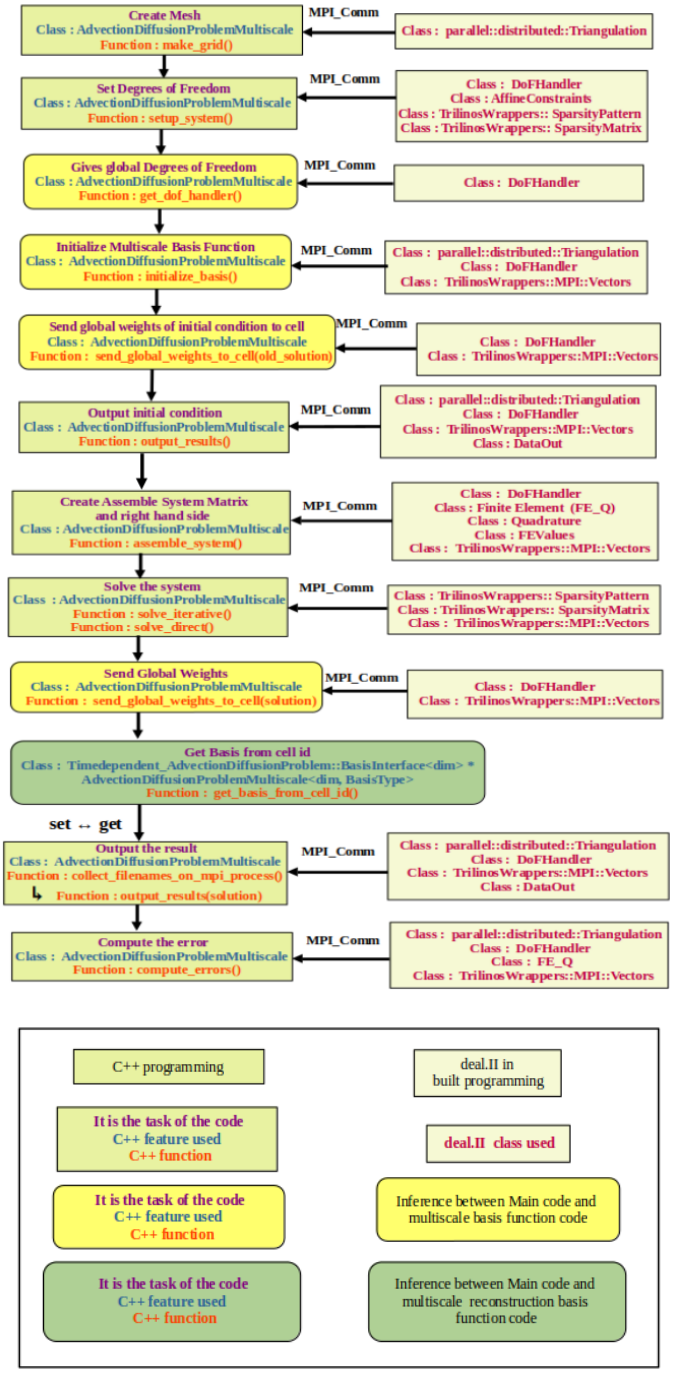


Figure 5.25: Flow chart of advection-diffusion equation multiscale finite element solution code main code.

We create **advectiondiffsuion_multiscale.hpp** code and define all the functions as shown in Flow chart (5.25) for global formulation. The code follows the same structure and steps of deal.II as **advectiondiffsuion_problem.hpp** with Flow chart (5.17) file discussed

before for advection diffusion equation solution with standard finite element method. The reader has already learned this concept. There are some new edition as follows:

- The template requires one more basis parameter **BasisType** compared to previous codes. It is for holding basis functions for each coarse cell.
- For **interface code** and **reconstruct basis code**, some additional functions are defined in the main code.
- The function **output_result** gets global solution in this cell as vtu by calling **output_global_solution_in_cell** function in **reconstruct basis code**.
- There are two functions in code: **send_global_weights_to_cell** and **output_result**. The first is to initialize this system and the second is to update it after the system has been solved.

The reader can see the full code in the link mentioned above. In the next sections, only the new function will be explained, and the old implementation will be referenced.

In the following sections, we will explain in detail the constructor, destructor, and new functions found in Listing (40). The function **get_basis_from_cell_id** defined in **main code** is used to get the reconstructed basis from **reconstruction basis code** in **main code**. The **get_dof_handler** is define in main code to get degrees of freedom in **main code** and then is used **interface code**.

```
namespace Timedependent_AdvectionDiffusionProblem {
  using namespace dealii;

template <int dim, class BasisType>
class AdvectionDiffusionProblemMultiscale : public AdvectionDiffusionBase<dim>
  {
    public:
    AdvectionDiffusionProblemMultiscale() = delete;
    AdvectionDiffusionProblemMultiscale(unsigned int n_refine, bool is_periodic);
    AdvectionDiffusionProblemMultiscale(const AdvectionDiffusionProblemMultiscale<dim, BasisType> &other) = delete;

    AdvectionDiffusionProblemMultiscale<dim, BasisType> &
    operator=(const AdvectionDiffusionProblemMultiscale<dim, BasisType> &other) = delete;

    ~AdvectionDiffusionProblemMultiscale();

    virtual const DoFHandler<dim> & get_dof_handler() const override;

    virtual Timedependent_AdvectionDiffusionProblem::BasisInterface<dim> *get_basis_from_cell_id(CellId cell_id) override;

    void run();

    private:
```

```
    std::map<CellId, BasisType> cell_basis_map;
  };

template <int dim, class BasisType>
Timedependent_AdvectionDiffusionProblem::BasisInterface<dim>
 *AdvectionDiffusionProblemMultiscale<dim, BasisType>::get_basis_from_cell_id(CellId cell_id)
  {
    return &(cell_basis_map.find(cell_id)->second);
  }
template <int dim, class BasisType>
const DoFHandler<dim> & AdvectionDiffusionProblemMultiscale<dim, BasisType>::get_dof_handler() const
{
  return dof_handler;
}
```

Listing 40: The advection-diffusion multiscale code in **advectiondiffsuion_multiscale.hpp** header file.

Connect main code to boundary conditions

In Flow chart (5.26), the boundary condition and coefficients implementations and their connection to the main code. In previous sessions, we introduced similar concepts.

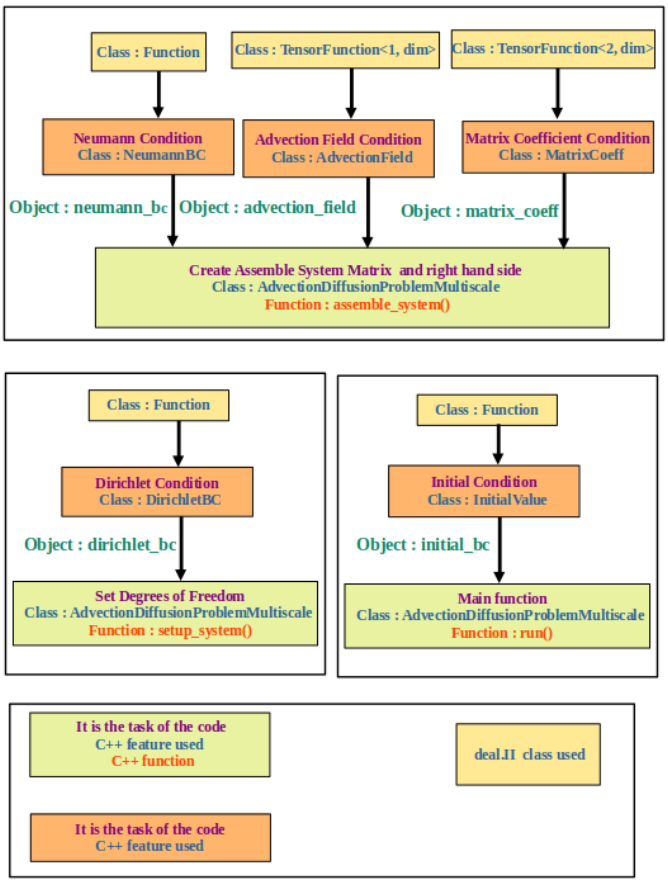


Figure 5.26: Flow chart of advection-diffusion equation multiscale finite element solution **main code** in **advectiondiffsuion_multiscale.hpp** header file and boundaries condition in other header files connections.

Connect main code to C++ config file

In Flow chart (5.27), the C++ **config.h** shows connectionof **config.h** file to the main code in **advectiondiffsuion_multiscale.hpp** header file.

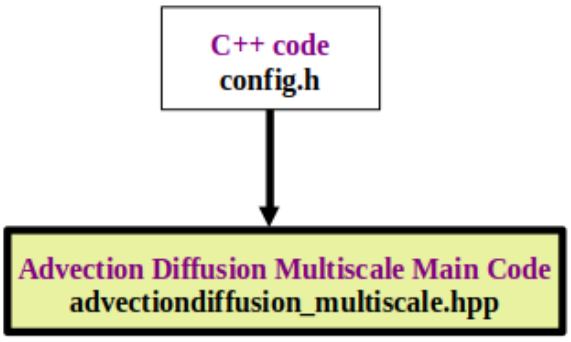


Figure 5.27: Flow chart of advection-diffusion equation multiscale finite element solution code in **advectiondiffsuion_multiscale.hpp** header file and C++ config file code in **config.h** header file.

Listing (41) is the implementation of numerical limits in C++ programming language for real numbers using advanced library called Standard Template Library STL is in **config.h** file. This file appears in all other header files as well.

```
#ifndef INCLUDE_CONFIG_H_
#define INCLUDE_CONFIG_H_
#include <deal.II/base/config.h>
#include <algorithm>
#include <cmath>
#include <iomanip>
#include <limits>
#include <type_traits>

#if !(defined(DEAL_II_WITH_TRILINOS))
#error DEAL_II_WITH_TRILINOS required
#endif

namespace Timedependent_AdvectionDiffusionProblem {
namespace Timedependent_AdvectionDiffusionProblemUtilities {
    /*!
    * Numeric epsilon for types::REAL. Interface to C++ STL.
    */
    static const double double_eps = std::numeric_limits<double>::epsilon();

    /*!
    * Numeric minimum for types::REAL. Interface to C++ STL.
    */
    static const double double_min = std::numeric_limits<double>::min();

    /*!
    * Numeric maximum for types::REAL. Interface to C++ STL.
    */
    static const double double_max = std::numeric_limits<double>::max();

    /*!
    * Function to compare two non-integer values. Return a bool. Interface to C++
    * STL.
    */
template <class T>
typename std::enable_if<!std::numeric_limits<T>::is_integer, bool>::type
    is_approx(T x, T y, int ulp = 2) {
      /* Machine epsilon has to be scaled to the magnitude of the values used and multiplied by the desired precision
      * in ULPs (units in the last place) */
      return std::abs(x - y) <= std::numeric_limits<T>::epsilon() * std::abs(x + y) * ulp
      /* unless the result is subnormal. */ || std::abs(x - y) < std::numeric_limits<T>::min();
    }
  } // namespace Timedependent_AdvectionDiffusionProblemUtilities
} // namespace Timedependent_AdvectionDiffusionProblem

#endif /* INCLUDE_CONFIG_H_ */
```

Listing 41: C++ config file in **config.h** header file.

Connect main code to three main other part of code

The Figure (5.28) shows all the connections between **main code** and 3 other important parts of code: basis code, interface code and reconstruction basis code. Now we introduce and explain all the connections to the **main code**.

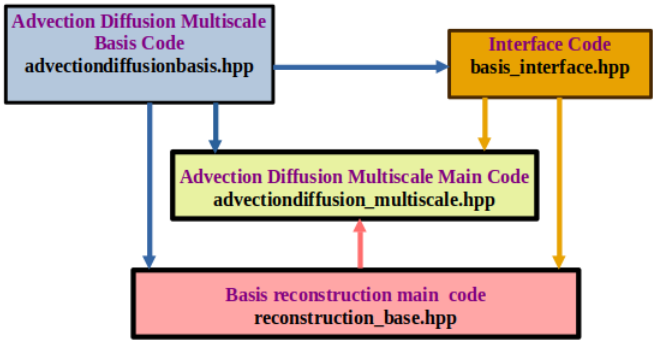


Figure 5.28: Flow chart of advection-diffusion equation multiscale finite element main code connections with other 3 main codes: basis code, interface code and reconstruction basis code.

1. The **main code** and **basis code** are connected using run() function of **basis code** in **main code** to get local multiscale basis and local solution.
2. The **main code** and **interface code** are connected using get_basis_from_cell_id pointer in **main code** to get reconstructed basis from local cell to global cell.
3. The **main code** and **reconstruction basis code** are connected using initial_reconstruction() to initialize the reconstructed basis.

5.17.2 Step 2 : For every cell $K \in \mathcal{T}_H$ a fine mesh $\mathcal{T}_h^K$ is to be initialized

This part is for local mesh with multiscale basis function.

Connect main code to multiscale basis code

The next step is to create the multiscale basis file, as we did with diffusion equation solutions using multiscale finite element methods.

We follow the same steps we did in the diffusion equation solution with MsFEM. We first create the Q1 Basis file as **q1bais.hpp** and add as header in **advectiondiffusion_basis.hpp**. Further Q1 Basis is called as object function **basis_q1** in file **advectiondiffusion_basis.hpp** as shown in Flow chart (5.29).

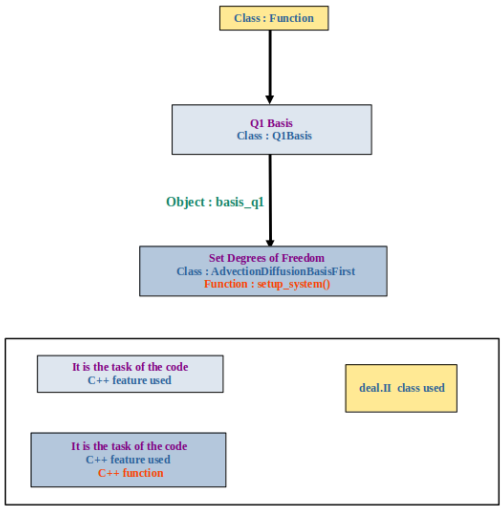


Figure 5.29: Flow chart of advection-diffusion equation multiscale finite element solution Q1 basis code.

Also, the following code of lines are added to map basis to cell in **main code**.

```
std::map<CellId, BasisType> cell_basis_map;
```

Listing 42: Cell basis map.

We follow the same steps for **advectiondiffusion_basis.hpp** as we did for **diffusion_basis.hpp** header file following the Flow chart (5.22) with the following changes.

1. The class name changes form **DiffusionProblemBasis** to **AdvectionDiffusionBasisFirst**
2. The code is spitted into two functions namely **initialize** and **make_time_step** which are used in main code and reconstruction code in order to used the deal.II class in there respective function.
3. For **initialize** we have following sub function for local cell.
   - For local mesh **make_grid** function is used.
   - For degree of freedom **setup_system** function is used.
   - Multiscale basis function is initialized and set with appropriate filename link to global cell with **set_filename_global** function.
   - and for given timesteps before the output. Initial basis output is generated with **output_basis**.
4. For **make_time_step** the time step update steps are created with the following sub function to be updated with time.

- The system is assembled with **assemble_system(time)** and is updated every time.
- Then local system is solved using **solve_direct** or **solve_iterative** function,
- Followed by global assembly with **assemble_global_element_data()** for global matrix and global right hand side. The get function is used to obtain these function at each time steps.
- The **compute_time_derivative** compute time derivative of basis at current time step using backward difference.
- Then with **set_filename_global** defines the global filename for pvtu-file in global output. The function **get_basis_info_string** is to gives basis filename, which then used in **set_filename_global** to add local basis to global cell.
- The **output_basis** is called to write basis results to disk in vtu-format.
- Global weight is generated with function **set_global_weights** and global assemble system is written in basis code but used in main code functions.
- In order to get global solution for each processor **output_global_solution_in_cell** is called to get multiscale basis on each global cell in **main code**.

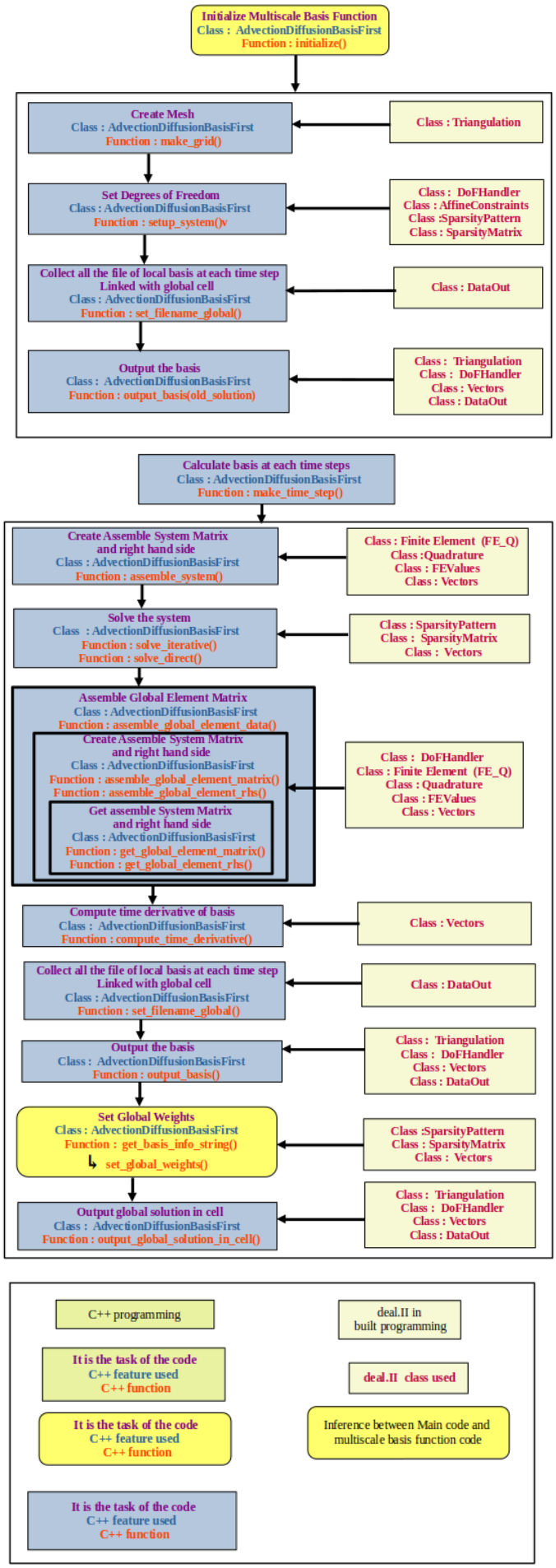


Figure 5.30: Flow chart of advection-diffusion equation multiscale **basis code**.

In Flow chart (5.31), the code in two yellow backgrounds for the functions **initialize_basis** and **set_global_weights** is already explained in the diffusion equation solution only the class name is changed as mentioned above.

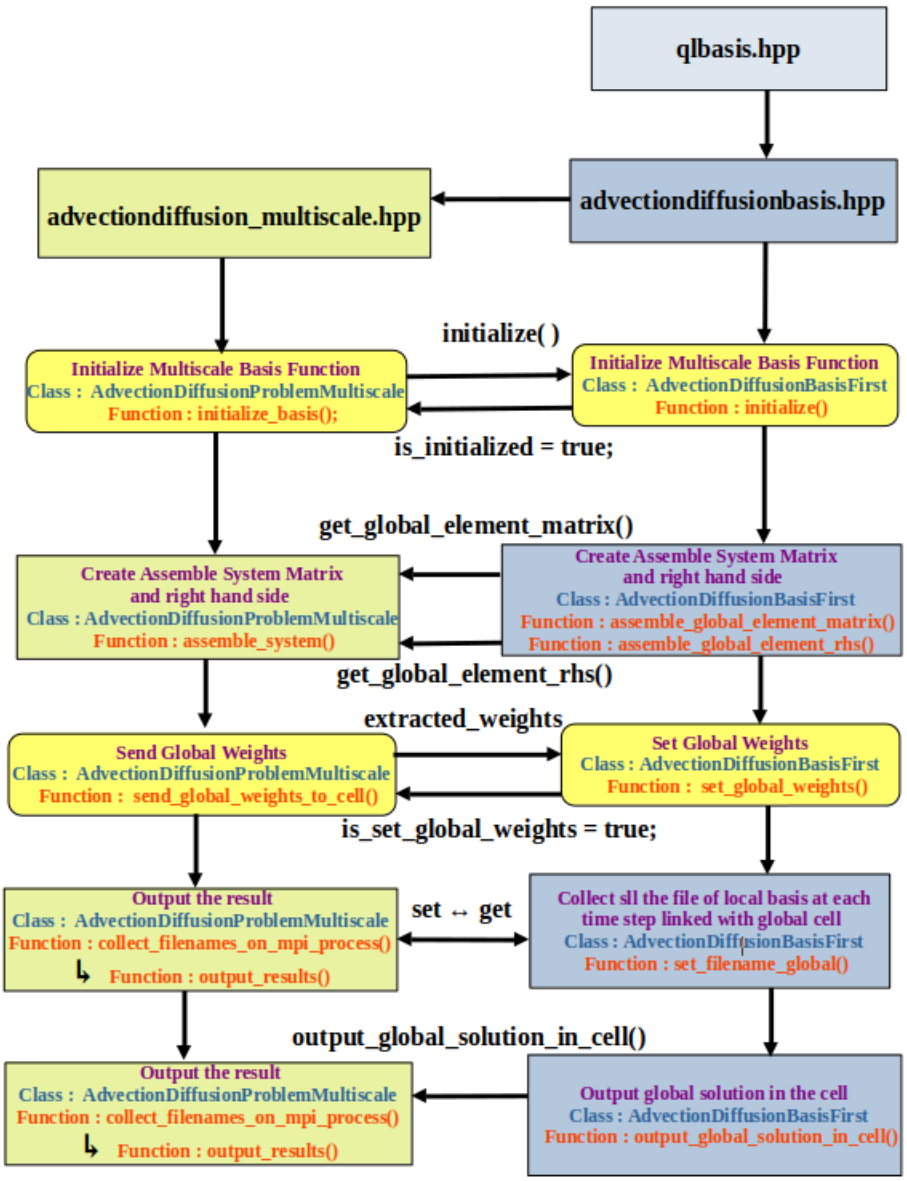


Figure 5.31: Flow chart of advection-diffusion main code connection with multiscale basis function code.

The **basis code** explained above would be used in following header files see Figure (5.32) as follows:

- The **main code** is connected to **basis code** with run function.
- The **basis code** is sent to **interface code** for initialization of local cell.
- The **basis code** passes to **semilagragian.hpp** header file to get advection field for trace back mesh to generate distorted mesh.

- The **basis code** is sent to **reconstruction basis code** to get known solution for distorted cell.

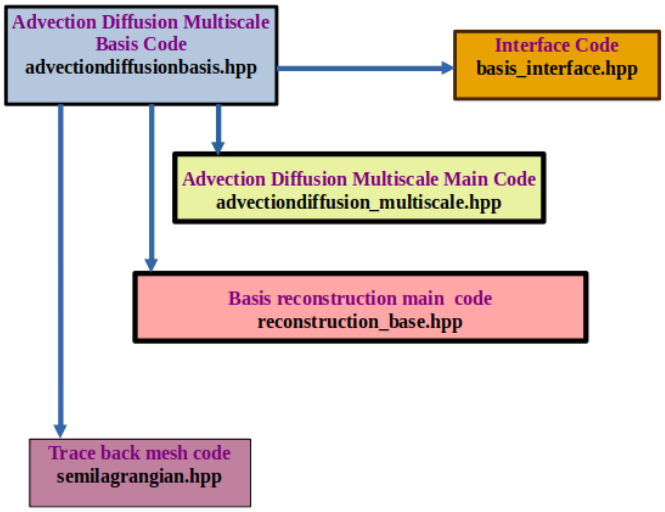


Figure 5.32: Flow chart of advection-diffusion multiscale basis code connections with other header files.

5.17.3 Step 3 : For $K \in \mathcal{T}_H$ (Online Phase), Reconstruct the basis $u_0(x)|_K$

Here first **interface code** in **basis_interface.hpp** header file would be introduced. Flow chart (5.33) shows the function for **interface code**.

The functions in the **interface code** works as follows:

- The **get_filename_global** function in **interface code** returns filename for local pvtu record that would be used in **collect_filenames_on_mpi_process** function in **main code** to gather all the semi-lagrangian basis solution together.
- The **get_global_dof_handler** function read-only access to global distributed DOFHandler this is used in **Muiltscale_FEFieldFunction.hpp** header file to access DOF for distributed mesh. Note that the **Muiltscale_FEFieldFunction.hpp** file will be explained in detail later.
- The **get_global_cell** function gets reference to global cell as pointer "*" to CellAcessor which then applied to local basis in **Muiltscale_FEFieldFunction.hpp** file.
- The **get_global_cell_dof_accessor** function gets reference "&" to global cell as pointer "*" to DoFCellAcessor. It gives all the information about global cell degree of freedom.

- The **get_other_local_basis** function get constant reference "&" to other local cell basis from locally owned **cell_id** function. It gives basis from the cell.
- The **get_local_field_function** function get constant reference "&" to locally owned field function object it is declared here, and then defined as reference "&" to the local cell basis in **Muiltscale_FEFieldFunction.hpp** header file and is called **reconstruction_base.hpp** header file.
- The **initial_reconstruction** function initializes reconstructed basis. Here it would just be declared but implemented in **reconstructed basis code**.
- The **get_basis_info_string** function gets strings information to added filename for reconstructed basis. Here it would just be declared but implemented in **reconstructed basis code**.

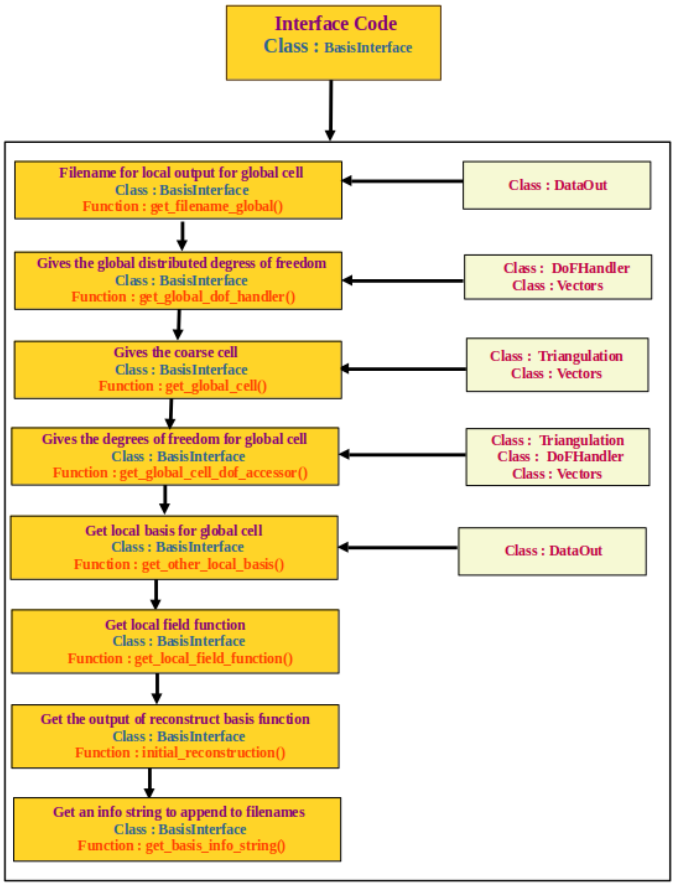


Figure 5.33: Flow chart of Interface code in **basis_interface.hpp** header file.

Listing (43) gives the code of **Basis Interface class** where first constructor and destructor are created and some of functions are initialized that where part of **basis code**. We also have several functions with prefix virtual void it is written in **Basis Interface class** and implemented in **basis code** and **reconstruction basis code**. One can see some key functions from local basis code in **advectiondiffusion basis.hpp** file and from **main code**

file in the below code.

```
template <int dim>
class BasisInterface
{
  public:
  BasisInterface() = delete;

  BasisInterface(typename Triangulation<dim>::active_cell_iterator &global_cell,
  bool is_first_cell, unsigned int local_subdomain,
  MPI_Comm mpi_communicator, AdvectionDiffusionBase<dim>& global_problem);
  /*!
  * Copy constructor is necessary since cell_id-basis pairs will
  * be copied into a basis std::map. Copying is only possible as
  * long as large objects are not initialized.
  */
  BasisInterface(const BasisInterface<dim> &X);
  /*!
  * Destructor must be virtual.
  */
  virtual ~BasisInterface()=0;
  /*!
  * Initialization function of the object. Must be called before first time
  * step update.
  */
  virtual void initialize() = 0;
  /*!
  * Make a global time step.
  */
  virtual void make_time_step()= 0;
  /*!
  * Write out global solution in this cell as vtu.
  */
  virtual void output_global_solution_in_cell() const = 0;
  /*!
  * Return the multiscale element matrix produced
  * from local basis functions.
  */
  virtual const FullMatrix<double> & get_global_element_matrix(bool current_time_flag) const = 0;
  /*!
  * Get the right hand-side that was locally assembled
  * to speed up the global assembly.
  */
  virtual const Vector<double> &get_global_element_rhs(bool current_time_flag) const = 0;
  /*!
  * Return filename for local pvtu record.
  */
  virtual const std::string &get_filename_global() const final;
  /*!
  * Get reference to global cell as pointer to DoFCellAcessor.
  */
  virtual typename DoFHandler<dim>::active_cell_iterator get_global_cell_dof_accessor() final;
  /*!
  * Get reference to global cell as pointer to CellAcessor.
  */
  virtual typename Triangulation<dim>::active_cell_iterator get_global_cell() final;
```

```
/*!
* Get global cell id.
*/
virtual CellId get_global_cell_id() const final;
/*!
* For some basis objects an initial reconstruction must be done. The
* default implementation in the base class does nothing bit it could be
* reimplemented in derived classes.
*/
virtual void initial_reconstruction();
/*!
* @brief Set global weights.
*
* The coarse weights of the global solution determine
* the local multiscale solution. They must be computed
* and then set locally to write an output.
*/
virtual void set_global_weights(const std::vector<double> &global_weights)= 0;
/*!
* Get an info string to append to filenames.
*/
virtual const std::string get_basis_info_string() = 0;
/*!
* Read-only access to global distributed DoFHandler.
*/
virtual const DoFHandler<dim> & get_global_dof_handler() const final;
/*!
* Get a const reference to locally owned (classic) field function object.
*/
virtual std::shared_ptr<Functions::FEFieldFunction<dim>>
get_local_field_function();
/*!
* Get const reference to other local cell basis from locally owned cell_id
*/
virtual BasisInterface<dim> * get_other_local_basis(CellId other_local_cell_id) final;
protected:
/*!
* Current MPI communicator.
*/
MPI_Comm mpi_communicator;

std::string filename_global;
/*!
* Guard for global filename passing without proper value for pvtu-file in
* global output.
*/
bool is_set_filename_global;
/*!
* Reference to global cell.
*/
typename Triangulation<dim>::active_cell_iterator global_cell;
/*!
* Global cell id.
*/
const CellId global_cell_id;
/*!
* Bool indicating if global_cell is the first in global mesh. Relevant to
```

```
  * output.
  */
  const bool is_first_cell;

  const unsigned int local_subdomain;

  private:
  /*!
  * Pointer to global problem. This should not be accessible in derived
  * classes.
  */
  AdvectionDiffusionBase<dim> *global_problem_ptr;

};
```

Listing 43: Basis Interface class in **basis_interface.hpp** header file.

Connect main code to interface code

In order to connect **main code** to **interface code** we follow following steps:

- First a **advectiondiffusionproblem_base.hpp** header file is created as shown in Figure (5.34) and the class **AdvectionDiffusionBase** is defined in order to access global degree of freedom from global cell in main code.
- To access the reconstructed bases in the interface code **basis_interface.hpp** header file with class **BasisInterface** defined in it.

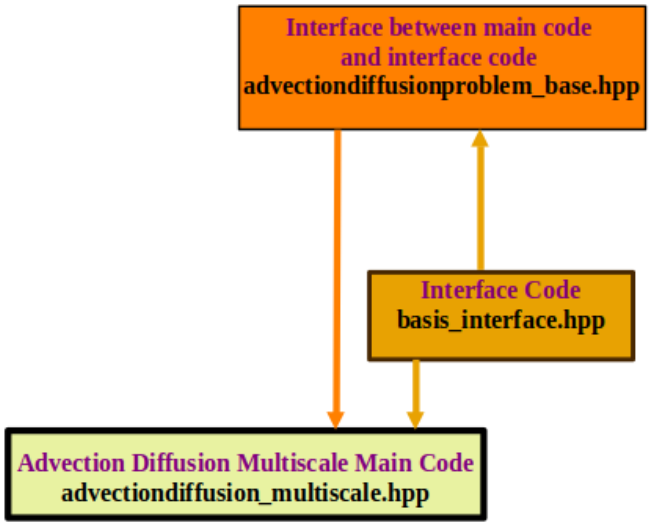


Figure 5.34: Flow chart of advection-diffusion equation main code connected to interface code.

Listing (44) shows how virtual function(refers Appendix IV [10]) is created to access global cell and reconstructed basis in **main code** and **interface code** respectively.

```
namespace Timedependent_AdvectionDiffusionProblem
  {
    template <int dim>
    class BasisInterface;

    template <int dim>
    class AdvectionDiffusionBase
    {
      public:
      AdvectionDiffusionBase() = default;

      virtual ~AdvectionDiffusionBase(){};

      virtual const DoFHandler<dim> & get_dof_handler() const = 0;

      virtual BasisInterface<dim> *get_basis_from_cell_id(CellId cell_id) = 0;
    };
  }
```

Listing 44: The advection diffusion base code in **advectiondiffusion_base.hpp** header file.

After that the file is added to the **main code** and **interface code**. In **main code** the object function **get_basis_from_cell_id** called to get reconstructed basis at each global cell.

Listing (45) gives reconstructed multiscale **basis code** corresponding to global cell in the form of C++ feature called pointer "*". This pointer requires an address "&" in order to access a cell.

```
template <int dim>
BasisInterface<dim> * BasisInterface<dim>::get_other_local_basis(CellId other_local_cell_id)
{
  return global_problem_ptr->get_basis_from_cell_id(other_local_cell_id);
}
```

Listing 45: Get local basis.

Listing (46) is code to get a reconstructed basis using C++ pointer "*" feature **get_basis_from_cell_id** in **main code** is accessed using address "&".

```
template <int dim, class BasisType>
Timedependent_AdvectionDiffusionProblem::BasisInterface<dim> *
AdvectionDiffusionProblemMultiscale<dim, BasisType>::get_basis_from_cell_id(CellId cell_id)
  {
    return &(cell_basis_map.find(cell_id)->second);
  }
```

Listing 46: Get basis from the cell id.

Figure (5.35) shown connection of **interface code** with other code. The local basis

function declared in **interface code** are define in **advectiondiffusion_basis.hpp** and **reconstruction_base.hpp** files.

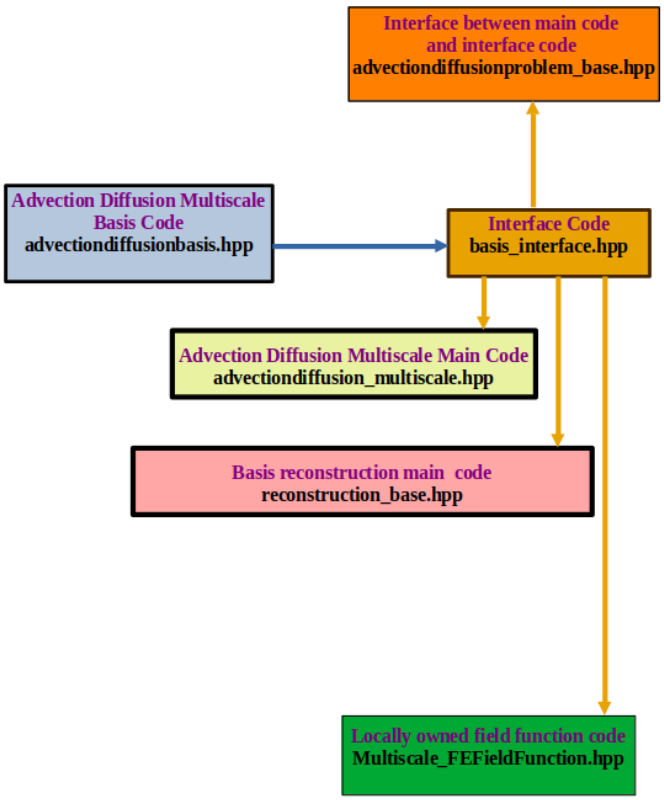


Figure 5.35: Flow chart of advection-diffusion equation multiscale finite element solution connection of interface code with other codes. .

5.17.4 Step 4 : For $n = 0$ to $n \leq N_{steps}$

Now are the steps above that are initialize would be used.

5.17.5 Step 4a : Each node in K is trace back one time step from t_{n+1} to t_n

To account for dominant advection, we combine semi-Lagrangian methods with multiscale methods. Reconstruction is based on the observation that the global solution at a previous timestep contains local information about the entire domain of dependence. A Eulerian multiscale basis can be constructed this way: we trace back at the time t^{n+1} an Eulerian cell $K \in \mathcal{T}_H$ with unknown solution and basis.are unknown to the previous time step t^n. This results in a distorted cell $\tilde{K}$, over which the solution u^n is known, but not the multiscale basis $\varphi_i, i = 1,2$.

Our goal is to trace back all nodes in $\mathcal{T}_H^K$ from time t^{n+1} to t^n in order to find the points

where information is transported. To accomplish this, one simply needs to solve an ODE with the time-reversed velocity field.

In the first line of code, the constructor, destructor and function are declared and we initialize the functions. Listing (47) gives the code to solve ODE with backward Euler step in **semilagrangian.hpp**, where the **time_step** is the global time step and **n_steps_local** is the local time step. Here we see that advection field as dependent variable that is traced back in order to get distorted cell at previous time step.

```
template <int dim>
Point<dim> SemiLagrangian<dim>:: operator()(const Point<dim> &in) const
{
  Assert(is_initialized, ExcNotInitialized());
  Assert(is_set_current_time, ExcNotInitialized());

  Point<dim> out(in);

  for (unsigned int j = 0; j < n_steps_local; ++j)
  {
    advection_field->set_time(current_time - j * time_step / n_steps_local);
    out = out - (time_step / n_steps_local) * advection_field->value(out);
  }
  return out;
}
```

Listing 47: Semi-Lagrangian trace back mesh **semilagrangian.hpp** header file.

Figure (5.36) shows how **semilagrangian.hpp** header file is connected to different files as follows:

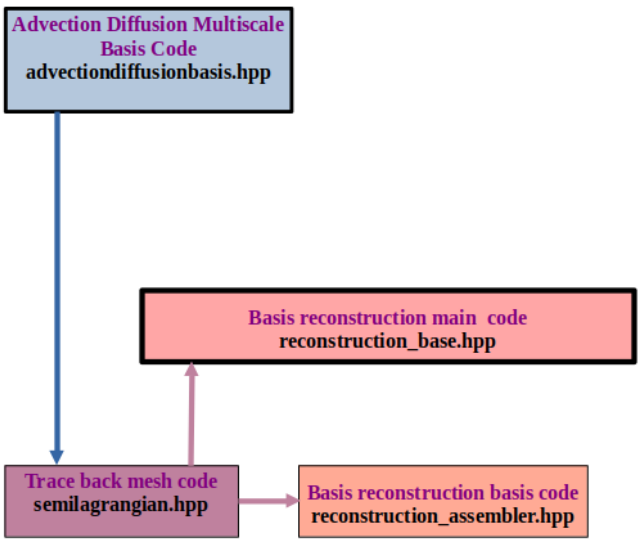


Figure 5.36: Flow chart of reconstruction basis code from basis code.

- The **basis code** in **advectiondiffusion_basis.hpp** header file has advection field

that is been used in current **semilagrangian.hpp** header file.

- The result of **semilagrangian.hpp** header file is the distorted cell that would be used to reconstruct the basis in **reconstruction basis code**.

Now in order to find whether the point belongs to unit cell or not, we create **get_domain_points.hpp** header file as shown in Listing (48).

```
template <int dim>
Point<dim> UnitCellPointFinder<dim>::value(const Point<dim> &p)
{
  //transform_real_to_unit_cell : Map the point p on the real cell to the corresponding point on the unit cell, and return its coordinates.
  Point<dim> unit_cell_point =mapping.transform_real_to_unit_cell(this_global_cell,p);
  // is_inside_unit_cell : Return true if the given point is inside the unit cell of the present space dimension.
  bool inside_unit_cell=GeometryInfo<dim>::is_inside_unit_cell(unit_cell_point);
  if(inside_unit_cell)
  return unit_cell_point;
}
```

Listing 48: Point finder in unit cell in **get_domain_points.hpp** header file.

Figure 5.37: Flow chart of find point in unit cell code connection with point belonging to MPI processor code.

Then we use this function by declaring an object function name **point_finder** for it and along with header file **semilagrangian.hpp** in the header file **Multiscale_FEFieldFunction.hpp** as shown in Figure (5.37). Now, the **Multiscale_FEFieldFunction.hpp** header file is the magic file as it solves the bottleneck of which MPI processor is associated with each semi-Lagragian trace back point. Thanks to **Konrad Simon** for implementing the **new** feature of **find_point_owner_rank** in section (5.13.3) to find the MPI rank of the processor in deal.II
Listing (49) shows the code that takes the object function **point_finder** to find the point in the unit cell. The result is value **p**. In order to find which process **p** belongs to, **find_point_owner_rank** is used. The result is allocated to shared pointer **local_field_function_ptr** that takes pointer for **get_local_field_function** function declared in **interface code**.

```
Coefficients::UnitCellPointFinder<dim> point_finder;
template <int dim, typename DoFHandlerType, bool is_periodic>
double MsFEFieldFunctionMPI<dim, DoFHandlerType, is_periodic>::value(
const Point<dim> & point,const unsigned int comp) const
```

```
{
  Assert(is_initialized, ExcNotInitialized());

  const Point<dim> p = point_finder(point);

  // Shared pointer
  std::shared_ptr<Functions::FEFieldFunction<dim>> local_field_function_ptr = local_basis_ptr->get_local_field_function();

  double value;

  typename DoFHandler<dim>::active_cell_iterator cell = this_global_cell;
  if (cell == global_dh->end()) cell = global_dh->begin_active();

  return value = local_field_function_ptr->find_point_owner_rank.value(point);

}
```

Listing 49: Find point owner rank in **Multiscale_FEFieldFunction.hpp** header file.

5.17.6 Step 4b : Basis $u_n(x)|_{\tilde{K}}$ from 4.10 to be reconstructed

We have two important header files for **reconstruct basis code** namely **reconstruct_base.hpp** header file for reconstruct basis to global cell and **reconstruct_assembler.hpp** header file to reconstruct basis. We will discuss **reconstruct_assembler.hpp** header file. The file now contains two important classes, namely **AdvectionDiffusionBasis_Reconstruction** and **BasicReconstructor** as shown in Flow chart (5.38).

The class **AdvectionDiffusionBasis_Reconstruction** in **reconstruct_assembler.hpp** header file has following function as shown in Flow chart (5.38). The function has the following role:

- The function **copy_triangulation** defines local cells.
- The function **initialize** is to initialize mesh with distorted cell with function **mesh_back_tracer**. We also initialize the solution, the solution time derivative, the matrix coefficient, and the right hand side of the equation.
- The function **trace_back_mesh** is called to transform local cell mesh to distorted cell mesh.
- The function **print_traced_mesh_info** to print mesh and solution information.
- The remaining functions **solution_vector**, **solution_vector_old**, **solution_vector_time_derivative**, **matrix_coeff**, **advection_field** and **right_hand_side** is to get pointers "*" for solution, solution old, solution time derivatives, matrix coefficients, advection field and right hand side.

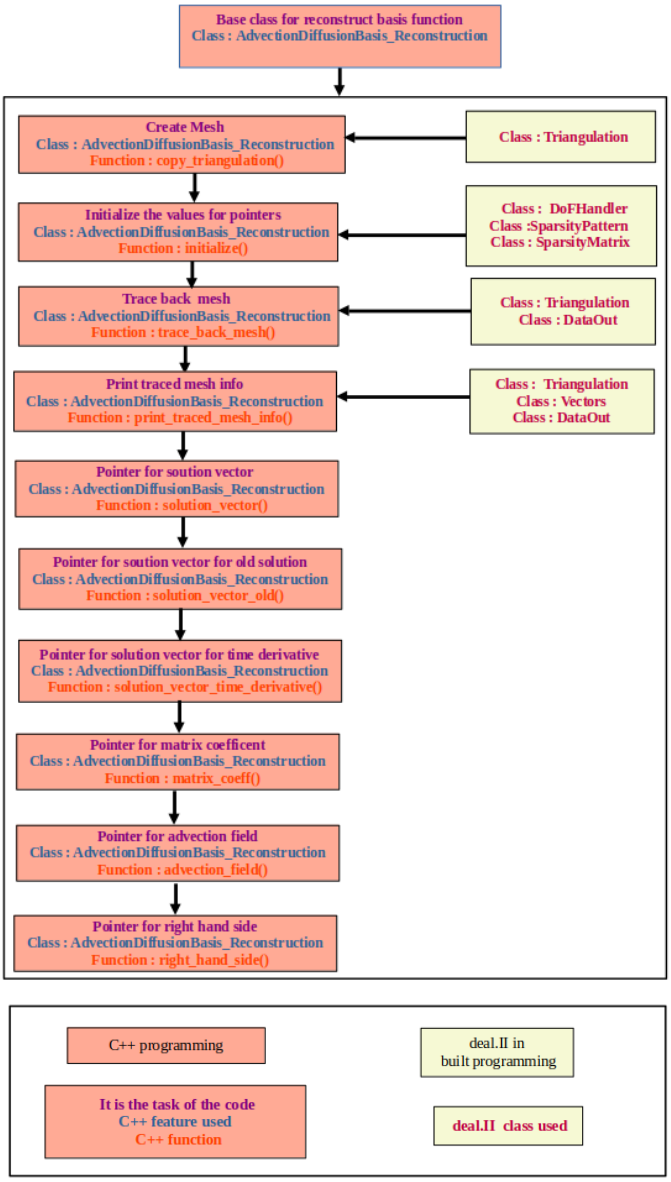


Figure 5.38: Flow chart of reconstruct basis code first part in **reconstruct_assembler.hpp** header file.

Listing (50) gives the code of **Basis Interface class** where first constructor and destructor are created and some of functions are initialized that where part of **basis code**.

```
template<int dim>
class AdvectionDiffusionBasis_Reconstruction {
  public:
  /*
  * Default Constructor.
  */
  AdvectionDiffusionBasis_Reconstruction() = delete;

  /*!
  * Constructor. Does not initialize fully.
  */
```

```
AdvectionDiffusionBasis_Reconstruction(BasisInterface<dim> & _local_basis,
MPI_Comm mpi_communicator,
const FE_Q<dim> & _fe,
Coefficients::MatrixCoeff<dim> & _matrix_coeff,
Coefficients::AdvectionField<dim> &_advection_field,
Coefficients::RightHandSide<dim> & _right_hand_side,
std::vector<Vector<double>> & _solution_vector,
std::vector<Vector<double>> & _solution_vector_old,
std::vector<Vector<double>> & _solution_vector_time_derivative,
bool _is_first_cell);
/*!
* Copy constructor is deleted.
*/
AdvectionDiffusionBasis_Reconstruction(const AdvectionDiffusionBasis_Reconstruction<dim> &X) = delete;
/*!
* Copy assignment is deleted.
*/
AdvectionDiffusionBasis_Reconstruction<dim> &
operator=(const AdvectionDiffusionBasis_Reconstruction<dim> &X) = delete;

/*!
* Destructor must be virtual.
*/
virtual ~AdvectionDiffusionBasis_Reconstruction()=0;
/*!
* Late initialization.
*/
void
initialize( const double _time_step,
const unsigned int _n_steps_local,
Coefficients::MatrixCoeff<dim> & _matrix_coeff,
Coefficients::AdvectionField<dim> &_advection_field,
Coefficients::RightHandSide<dim> & _right_hand_side,
std::vector<Vector<double>> & _solution_vector,
std::vector<Vector<double>> & _solution_vector_old,
std::vector<Vector<double>> &_solution_vector_time_derivative);

/*!
* Virtual function to implement a
* reconstruction for the initial basis. This is done from an initial
* condition. Does nothing by default.
*/
virtual void
basis_initial_reconstruction(){};
/*!
* Pure virtual function to implement a reconstruction step for the basis
* at current_time.
*/
virtual void
basis_reconstruction(const double current_time,
const double time_step,
const double theta) = 0;
/*!
* Get an info string to append to filenames.
*/
virtual const std::string
```

```
get_info_string() = 0;
/*!
* Interface to copy a triangulation into the object.
*/
void
copy_triangulation(Triangulation<dim> &other_tria);
/*!
* Plot trace back mesh with info for diagnostic purposes. Only
* implemented in base class.
*/
virtual void
print_traced_mesh_info(const Vector<double> &solution,
const std::string & filename) final;
/*
* These members and functions should be available in derived classes.
*/
protected:

void
trace_back_mesh(const double current_time);

Coefficients::MatrixCoeff<dim> & matrix_coeff();

Coefficients::AdvectionField<dim> & advection_field();

Coefficients::RightHandSide<dim> & ight_hand_side();

std::vector<Vector<double>> & solution_vector();

std::vector<Vector<double>> & solution_vector_old();

std::vector<Vector<double>> & solution_vector_time_derivative();

MPI_Comm mpi_communicator;

Triangulation<dim> local_tria;

DoFHandler<dim> dof_handler;

const FE_Q<dim> fe;

private:

Coefficients::MatrixCoeff<dim> *matrix_coeff_ptr;

/*!
* Reference to time-dependent vector coefficient (velocity).
*/
Coefficients::AdvectionField<dim> *advection_field_ptr;

/*!
* Reference to time-dependent scalar coefficient (forcing).
*/
Coefficients::RightHandSide<dim> *right_hand_side_ptr;

std::vector<Vector<double>> *solution_vector_ptr;
```

```
  std::vector<Vector<double>> *solution_vector_old_ptr;

  std::vector<Vector<double>> *solution_vector_time_derivative_ptr;

  bool is_initialized;

  protected:

  bool is_first_cell;

  private:

  Coefficients::SemiLagrangian<dim> mesh_back_tracer;

  const unsigned int n_refine_local =4;

  protected:
  BasisInterface<dim> *local_basis_ptr;
};
```

Listing 50: AdvectionDiffusionBasis_Reconstruction class in **reconstruction_assembler.hpp** header file.

Now we go through **BasicReconstructor** class **reconstruct_assembler.hpp** header file. Flow chart (5.39) gives overview of the functions in the class. The implication of these functions is as follows:

- The **BasicReconstructor** is defined with public constructor for class **AdvectionDiffusionBasis_Reconstruction** to use all the functions defined in it.
- The function **initialize** is to initialize global cell and local cell with all the functions **solution_vector**, **solution_vector_old**, **solution_vector_time_derivative**, **matrix_coeff**, **advection_field** and **right_hand_side** is to get pointers for solution, solution old, solution time derivatives, matrix coefficients, advection field and right hand side. Q1Basis and global degrees of freedom is initialized with function **setup_system**.
- In the **assemble_system(time)** function, the local system is assembled with the this pointer of the C++ and updated every time.
- Then **compute_time_derivative** is set with this pointer and is updated every time.
- Followed by global assembly with **assemble_global_element_matrix** with help of this pointer.
- The **assemble_global_element_rhs** is also created with the help of this pointer.
- The **basis_reconstruction** updates the assembled local system every time steps.
- The **get_info_string** gives the filename output for reconstructed basis at every time

steps.

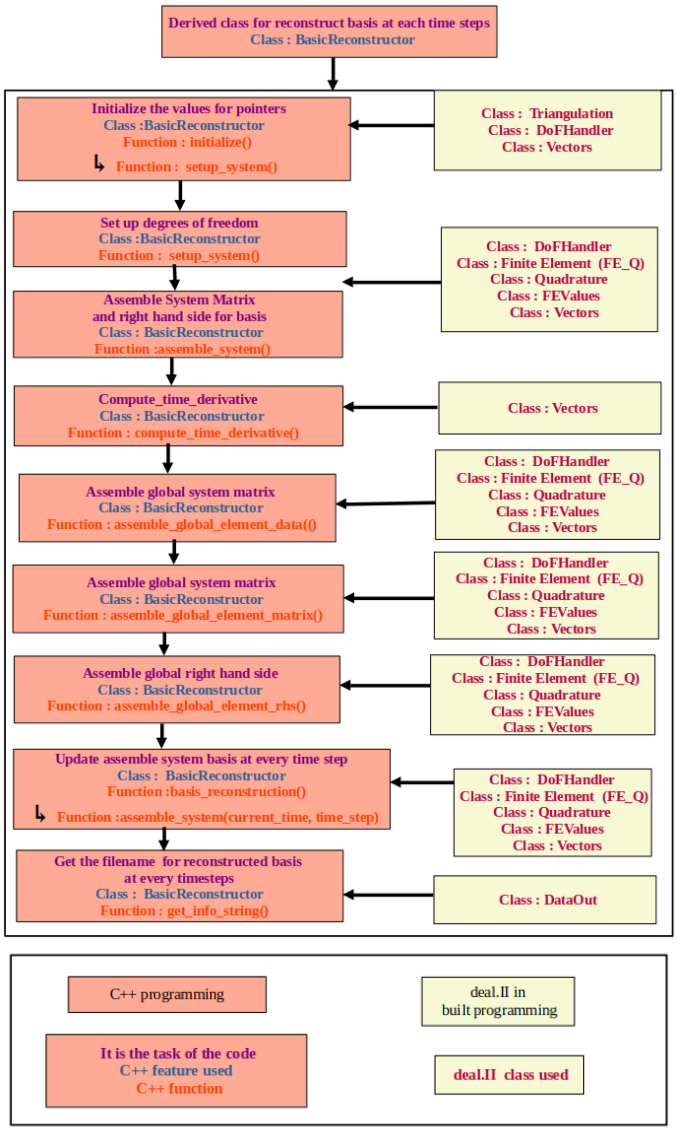


Figure 5.39: Flow chart of reconstruct basis code second part in **reconstruct_assembler.hpp** header file.

5.17.7 Step 4c : Propagate the boundary conditions of the optimal basis forward onto K

Global Formulation: Assemble the global (coarse) system matrices with reconstructed basis function. We follow the steps for **reconstruct basis code** in **reconstruction_base.hpp** header file following the Flow chart (5.40) with the corresponding changes from **basis code**.

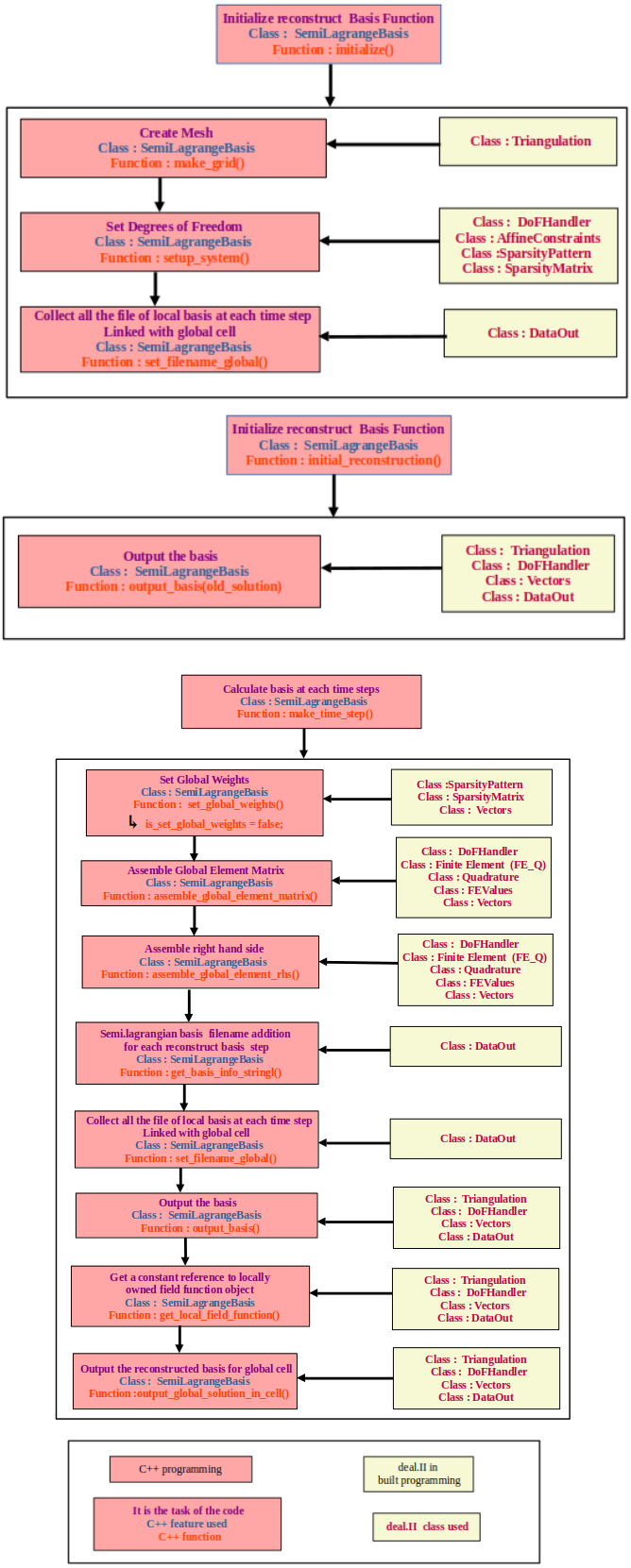


Figure 5.40: Flow chart of reconstruct basis code for global cell in **reconstruction_base.hpp** header file.

Figure (5.41) gives the overview of how the different files would be connected in

reconstruction process.

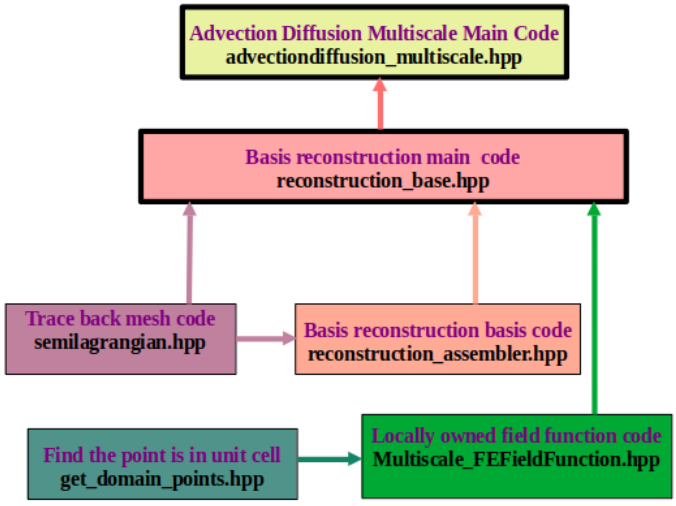


Figure 5.41: Flow chart of reconstruction multiscale basis code connection with main code.

Now we will walk through the steps in class **SemiLagrangianBasis** and its connection to **main code**.

1. Create **reconstruction_basis.hpp** header file with class **SemiLagrangianBasis**. It has one more parameter **ReconstructionType** for including **reconstruct_assembler.hpp** header file functions in **reconstruction_basis.hpp** header file.
2. The code is spitted into three functions instead of two. These functions are **initialize**, **initialize_reconstruction** and **make_time_step** which are used in **main code** and **reconstruction basis code** to used the deal.II class in there respective function.
3. For **initialize** we have the following sub function for local cell.
    - For local mesh copy triangulation from **basis code** in **make_grid** in **reconstruction basis code**.
    - For degree of freedom **setup_system** function reinitialize solution vectors, old solution vectors and global right hand side.
    - The function **get_local_field_function** was declared in **interface code** to get local pointer "*" for locally owned field function object in **MsFEFieldFunctionMPI** function of **Multiscale_FEFieldFunctionMPI.hpp** header file.
    - Multiscale basis function is initialized and set with appropriate filename link to global cell with **set_filename_global** function. Initial reconstruction for the

basis can now be done since the initial global solution is locally set.

4. and the function **initial_reconstruction** initial reconstructed basis output is generated with **output_basis** for solution at previous time step.
5. For **make_time_step** the time step update steps are created with the following sub function to be updated with time.
    - Global weight is generated with function **set_global_weights** is updated in **main code** functions.
    - Followed by global assembly with for global matrix which is obtained from function **get_global_element_matrix**. This part is connected with the **main code**.
    - and global right hand side is obtained from function **get_global_element_matrix** respectively. This part is connected with the **main code**.
    - The filename is added as semi-Lagrangian basis at each reconstructed basis step with **get_basis_info_string** function. This part is initialize in **interface code**.
    - Then filename is updated with **set_filename_global** at every time steps.
    - The **output_basis** is called for the new updated solution in **main code** and function **global_cell_id** to write local reconstructed basis for global cell.
    - The **get_local_field_function** is return constant reference to locally owned field function object, defined in **Muiltscale_FEFieldFunction.hpp** header file.
    - The **output_global_solution_in_cell** is called get global solution for each processor. It is called to get multiscale basis on each global cell in **main code**.

Listing (51) gives the implementation of function in deal.II.

```
template <int dim, class ReconstructionType>
void SemiLagrangeBasis<dim, ReconstructionType>::initialize()
{
  Timer timer;
  timer.restart();

  make_grid();

  setup_system();

  if (verbose)
```

```
  {
    std::cout << " Initializing local basis in global cell id "
    << this->global_cell_id.to_string() << " (subdomain "<< this->local_subdomain << " "
    << triangulation.n_active_cells() << " active fine cells "
    << dof_handler.n_dofs() << " subgrid dofs) ....";
  }
  if (verbose_all)
  std::cout << std::endl;

  {
    /*
    * The global solution must be set in this case for an initial
    * reconstruction.
    */
    AffineConstraints<double> constraints_fake;
    constraints_fake.clear();
    DoFTools::make_hanging_node_constraints(dof_handler, constraints_fake);
    constraints_fake.close();

    VectorTools::project(dof_handler, constraints_fake, QGauss<dim>(fe.degree + 1),
    initial_value,global_solution);

    local_field_function_ptr.reset(new Functions::FEFieldFunction<dim>(dof_handler, global_solution));

    is_set_global_solution = true;
  }

  basis_reconstructor.initialize(this->global_cell, time_step, /* n_steps_local */ 1,
  matrix_coeff, advection_field, right_hand_side, solution_vector,
  solution_vector_old, solution_vector_time_derivative);

  /*
  * Must be set with appropriate name for this timestep before output.
  */
  set_filename_global();

  if (verbose)
  {
    timer.stop();
    std::cout << " done in " << timer.cpu_time() << " seconds." << std::endl;
  }
}

template <int dim, class ReconstructionType>
void SemiLagrangeBasis<dim, ReconstructionType>::initial_reconstruction()
{
  /*
  * Initial reconstruction for the basis can now be done since the initial
  * global solution is locally set.
  */
  time = 0.0;
  basis_reconstructor.basis_initial_reconstruction();

  if (this->is_first_cell)
  {
    output_basis(solution_vector_old);
  }
```

```
}
template <int dim, class ReconstructionType>
void SemiLagrangeBasis<dim, ReconstructionType>::make_time_step()
{
  Timer timer;
  if (verbose)
  {
    timer.restart();
    std::cout << " Time step for local basis in global cell id "
    << this->global_cell_id.to_string() << " (subdomain " << this->local_subdomain << " "
    << triangulation.n_active_cells() << " active fine cells " << dof_handler.n_dofs() << " subgrid dofs) ....";
  }
  if (verbose_all)
  std::cout << std::endl;

  time += time_step;
  ++timestep_number;

  // reset
  is_set_global_weights = false;
  this->is_set_filename_global = false;
  is_solved = false;

  {
    basis_reconstructor.basis_reconstruction(/* current time = */ time,
    /* dt = */ time_step, /* theta = */ theta);
    is_solved = true;
  }

  basis_reconstructor.assemble_global_element_data(global_element_matrix,
  global_element_matrix_old, global_element_rhs, global_element_rhs_old,
  time_step, theta);

  /*
  * Must be set with appropriate name for this timestep before output.
  */
  set_filename_global();

  if (output_first_basis && this->is_first_cell)
  output_basis(solution_vector);

  if (verbose)
  {
    timer.stop();

    std::cout << " done in " << timer.cpu_time() << " seconds." << std::endl;
  }
}
```

Listing 51: Flow chart of reconstruct basis code for global cell in **reconstruction_base.hpp** header file.

5.17.8 Step 5 : Postprocess the solution

Listing (52) gives the code for output basis. In the first few lines of code, the local reconstructed basis is gathered, and then all the basis for MPI processes is gathered using a for loop. After that pvtu files are generated for collecting solution of global cell and then for collecting output for local cell.

```
template <int dim, class BasisType>
void AdvectionDiffusionProblemMultiscale<dim, BasisType>::output_results( TrilinosWrappers::MPI::Vector &vector_out) const
{
  // write local fine solution
  typename std::map<CellId, BasisType>::const_iterator it_basis = cell_basis_map.begin(),
  it_endbasis =cell_basis_map.end();

  for (; it_basis != it_endbasis; ++it_basis)
  {
    (it_basis->second).output_global_solution_in_cell();
  }

  // Gather local filenames
  std::vector<std::string> filenames_on_cell;
  {
    std::vector<std::vector<std::string>> filename_list_list = Utilities::MPI::gather(mpi_communicator,
    collect_filenames_on_mpi_process(), /* root_process = */ 0);

    for (unsigned int i = 0; i < filename_list_list.size(); ++i)
    for (unsigned int j = 0; j < filename_list_list[i].size(); ++j)
    filenames_on_cell.emplace_back(filename_list_list[i][j]);
  }

  std::string filename = (dim == 2 ? "solution-ms_2d" : "solution-ms_3d");
  DataOut<dim> data_out;
  data_out.attach_dof_handler(dof_handler);
  data_out.add_data_vector(vector_out, "u");

  Vector<float> subdomain(triangulation.n_active_cells());
  for (unsigned int i = 0; i < subdomain.size(); ++i)
  {
    subdomain(i) = triangulation.locally_owned_subdomain();
  }
  data_out.add_data_vector(subdomain, "subdomain");

  // Postprocess
  // std::unique_ptr<Q_PostProcessor> postprocessor(
  // new Q_PostProcessor(parameter_filename));
  // data_out.add_data_vector(locally_relevant_solution, *postprocessor);

  data_out.build_patches();

  std::string filename_local_coarse(filename);
  filename_local_coarse += "_coarse_refinements-" + Utilities::int_to_string(n_refine, 2) + "." +
  "theta-" + Utilities::to_string(theta, 4) + "." + "time_step-" +
  Utilities::int_to_string(timestep_number, 4) + "." +
```

```
  Utilities::int_to_string(triangulation.locally_owned_subdomain(), 4) + ".vtu";

  std::ofstream output(filename_local_coarse.c_str());
  data_out.write_vtu(output);

  /*
  * Write a pvtu-record to collect all files for each time step.
  */
  if (Utilities::MPI::this_mpi_process(mpi_communicator) == 0)
  {
    std::vector<std::string> all_local_filenames_coarse;
    for (unsigned int i = 0;
    i < Utilities::MPI::n_mpi_processes(mpi_communicator); ++i)
    {
      all_local_filenames_coarse.push_back(
      filename + "_coarse_refinements-" + Utilities::int_to_string(n_refine, 2) + "." + "theta-" +
      Utilities::to_string(theta, 4) + "." + "time_step-" + Utilities::int_to_string(timestep_number, 4) + "." +
      Utilities::int_to_string(i, 4) + ".vtu");
    }

    std::string filename_master(filename);

    filename_master += "_coarse_refinements-" + Utilities::int_to_string(n_refine, 2) + "." + "theta-" +
    Utilities::to_string(theta, 4) + "." + "time_step-" + Utilities::int_to_string(timestep_number, 4) + ".pvtu";

    std::ofstream master_output(filename_master);
    data_out.write_pvtu_record(master_output, all_local_filenames_coarse);
  }

  // pvtu-record for all local fine outputs
  if (Utilities::MPI::this_mpi_process(mpi_communicator) == 0)
  {
    std::string filename_master = filename;
    filename_master += "_fine_refinements-" + Utilities::int_to_string(n_refine, 2) + "." + "theta-" +
    Utilities::to_string(theta, 4) + "." + "time_step-" + Utilities::int_to_string(timestep_number, 4) + ".pvtu";

    std::ofstream master_output(filename_master);
    data_out.write_pvtu_record(master_output, filenames_on_cell);
  }
}
```

Listing 52: Output the solution in main code.

The error is computed similar as for advection-diffusion problem. Here we use **main code** to find error. Here we conclude Chapter 5 with lots of implementation and Flow chart descriptions.

6 Numerical result

"I have had my results for a long time: but I do not yet know how I am to arrive at them."

- Carl Friedrich Gauss

6.1 Numerical Experiments.

All implementations were performed in C++ using deal.II 9.4.0 [7] . Now apply this methods to two model problems of increasing complexity. The first model problem involves a periodically oscillating coefficient function. We calculate quantitative results in terms of explicit L^2, L^∞ and H^1 errors. The second model problem involves diffusion with the Neumann condition on odd boundaries and the Dirichlet boundary on even boundaries as shown in Figure (6.1). Here, the exact solution is unavailable. We qualitatively evaluate our MsFEM approximations and Low resolution FEM results by comparing them with finite element computations on a highly resolved grid.

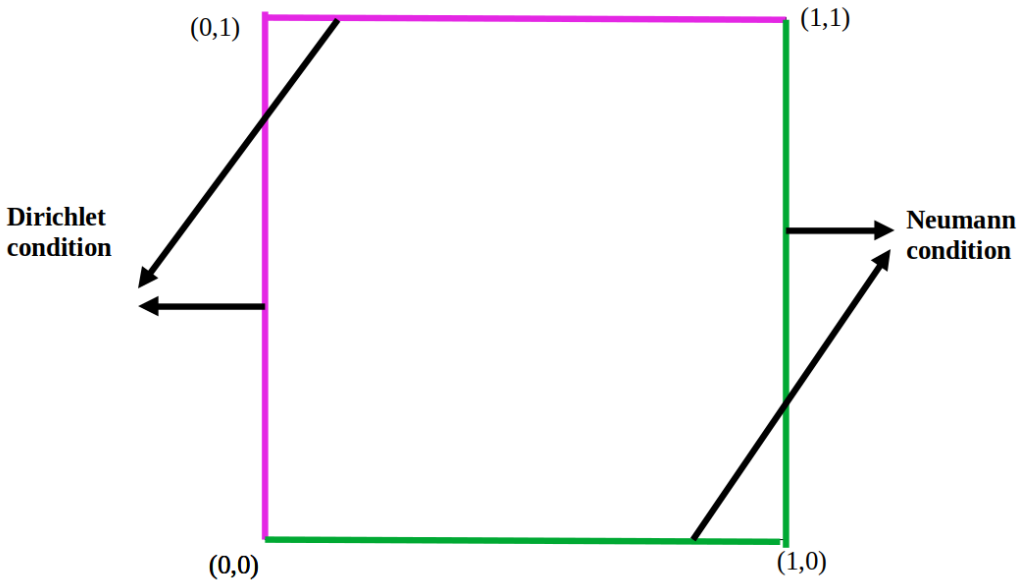


Figure 6.1: Domain with boundary condition.

The computations were performed on a Linux based Marin cluster from 10 to 80 CPUs.

6.1.1 Test case 1

Let $\Omega = [0,1]$ $\varepsilon = 0.05$. We define Poisson's Equation

$$-\nabla \cdot (a_\varepsilon \nabla u) = f \quad \text{in} \quad \Omega$$

Dirichlet condition is

$$u = 0 \quad \text{on} \quad \partial\Omega$$

where

Diffusion coefficient

$$a_\varepsilon(x,t) = \left(I_2 + \begin{bmatrix} sin(4\pi x/\varepsilon) & 0 \\ 0 & sin(4\pi y/\varepsilon) \end{bmatrix} \right)$$

and

$$f = -\nabla \cdot (A_\varepsilon \nabla u) \approx sin(2\pi x) sin(2\pi y)$$

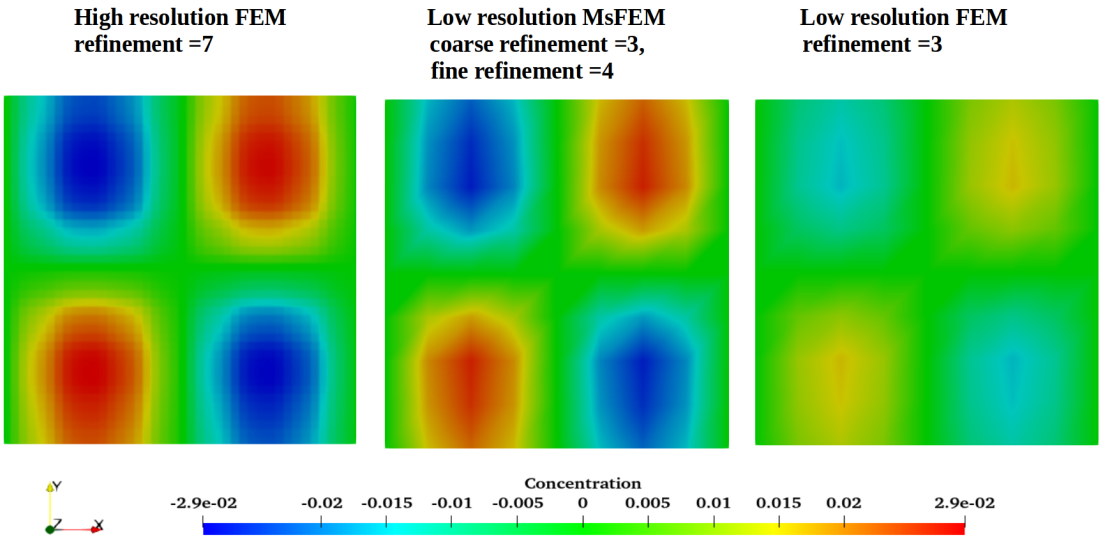


Figure 6.2: Test case 1 Solution of High resolution FEM, Low resolution FEM and Low resolution MsFEM for stationary diffusion equation.

According to Figure (6.2), the high-resolution FEM captures all small scales because its mesh resolution is high, whereas the low-resolution FEM does not provide accurate results. Low-resolution MsFEM gives acceptable results.

Test case 1: Error Table

The error is calculated using the difference between the reference solution, which is a high resolution FEM solution, and the low resolution FEM and MsFEM solutions.

Relative Error Analysis			
Simulation Type	L^2 Error	H^1 Error	L^∞ Error
Low resolution FEM	0.0080	0.1011	0.0146
Low resolution MsFEM	0.0019	0.0555	0.0041

Table 6.1: Test case 1 Error in simulation with Low resolution FEM and Low resolution MsFEM.

It can be seen from the above Table (6.1) that low-resolution MsFEM has less error then low-resolution FEM.

Test case 1: Wall times with respect to MPI rank

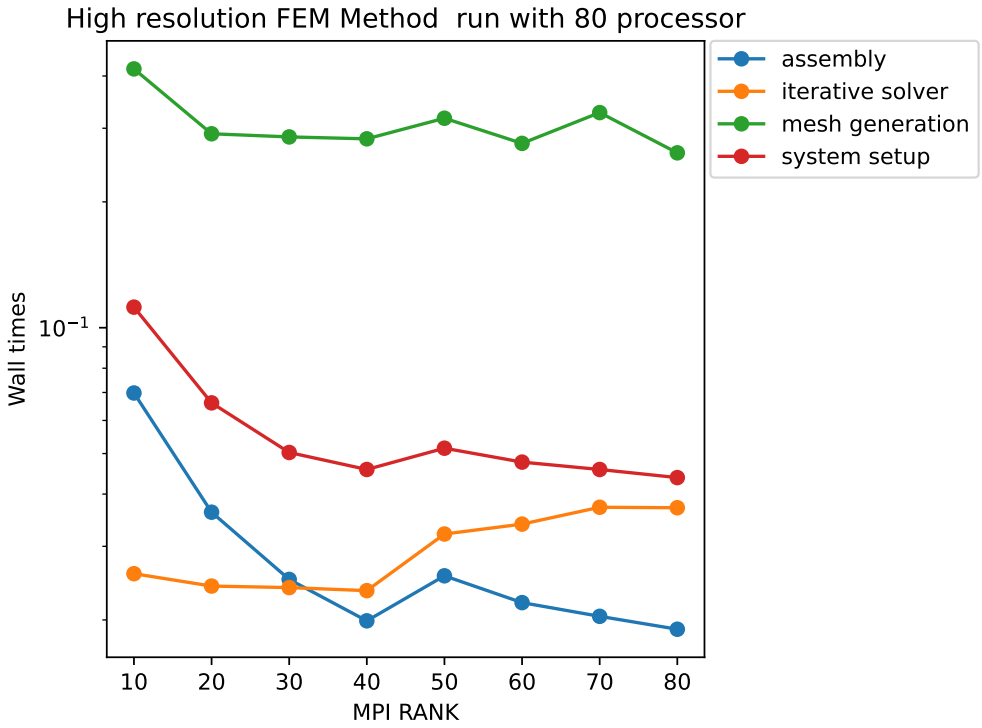


Figure 6.3: Test case 1 Wall times with respect to MPI rank for High resolution FEM method.

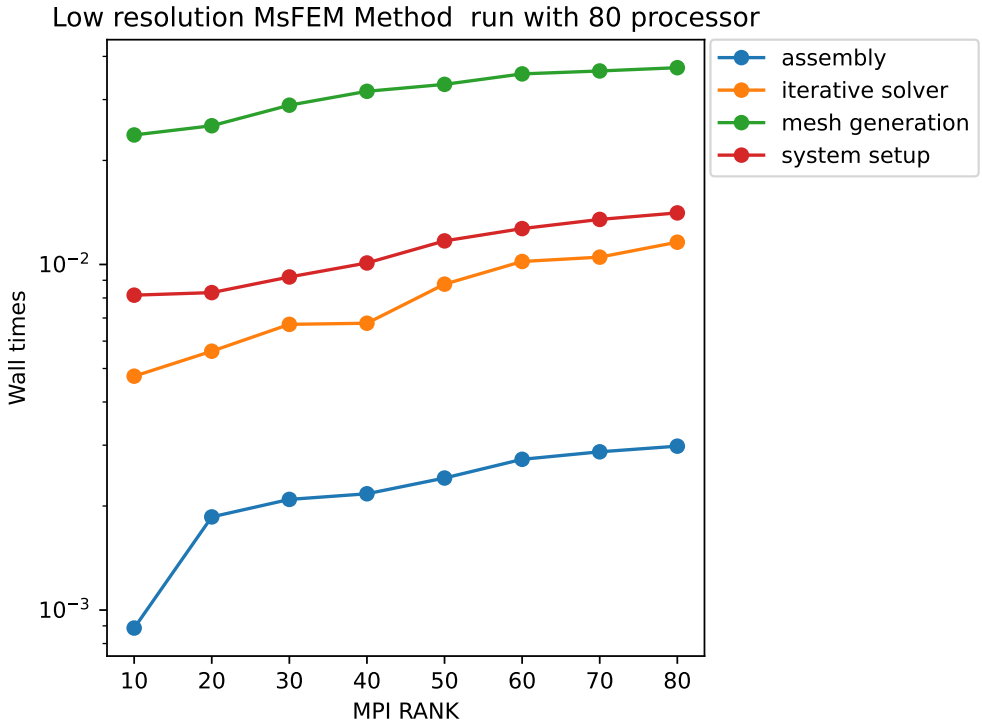


Figure 6.4: Test case 1 Wall times with respect to MPI rank for Low resolution MsFEM method.

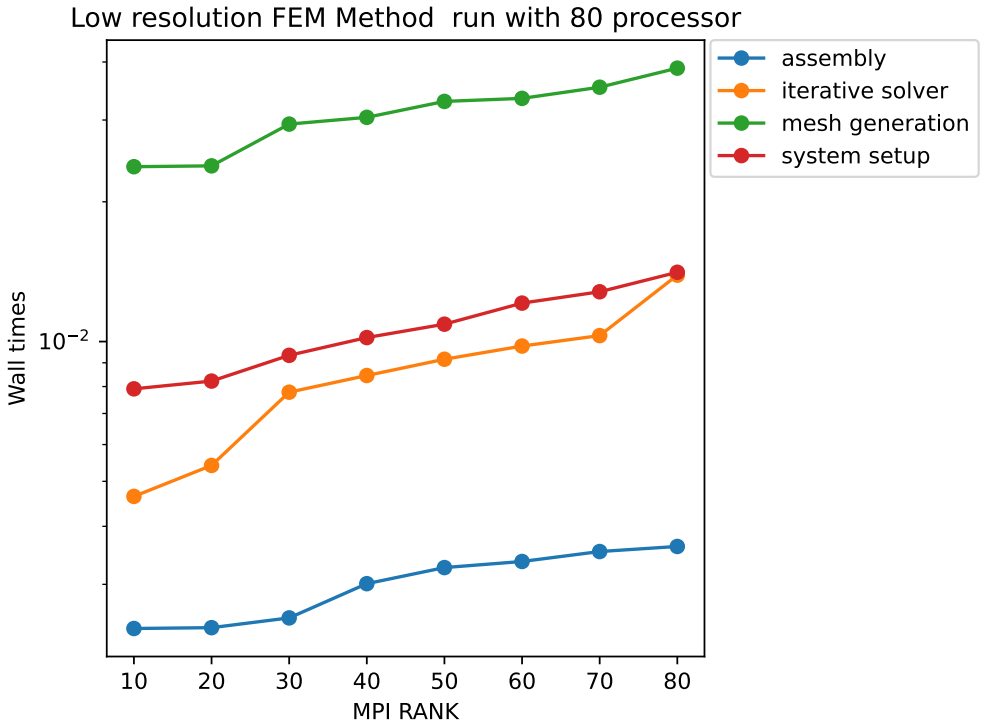


Figure 6.5: Test case 1 Wall times with respect to MPI rank for Low resolution FEM method.

In Figures [(6.3),(6.4) and (6.5)] here we run 3 different simulations for test case 1 with 81 degrees of freedom for low-resolution FEM. For high-resolution FEM, there are 16641 degrees of freedom while for multiscale finite element method, there are 81 degrees of freedom. Mesh generation in high-resolution FEM takes the maximum time; however, it decreases with the number of processors, except for the 50 and 70 MPI ranks. For low-resolution FEM and low-resolution MsFEM, there are 81 degrees of freedom. Because of the size of the problem, mesh generation wall times does not vary much. For the remaining steps, such as assembly, the iterative solver and system setup decrease except at 50 MPI ranks in high-resolution FEM. The number of iterative solvers increases in low-resolution FEM, but setup and assembly does not increases much. In the case of a low-resolution MsFEM assembly, the system setup and iterative solver wall times increase then becomes almost constant. The number of degrees of freedom is less than 10,000 in Low Resolution FEM and MsFEM are not capable of showing good scaling. The idea here is to compare three cases of parallel computing.

Test case 1: Memory Consumption

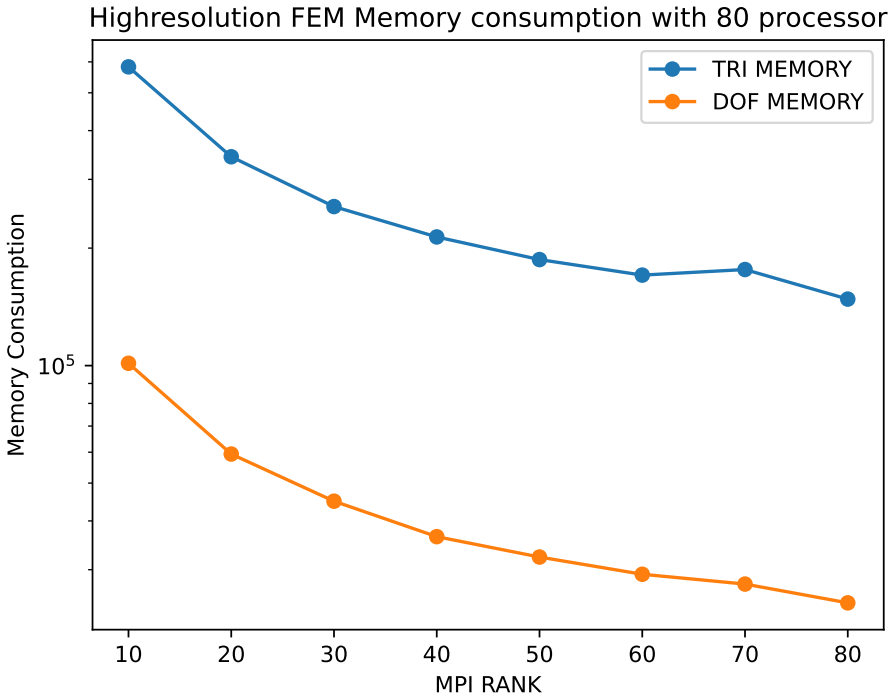


Figure 6.6: Test case 1 Memory consumption for High resolution FEM method.

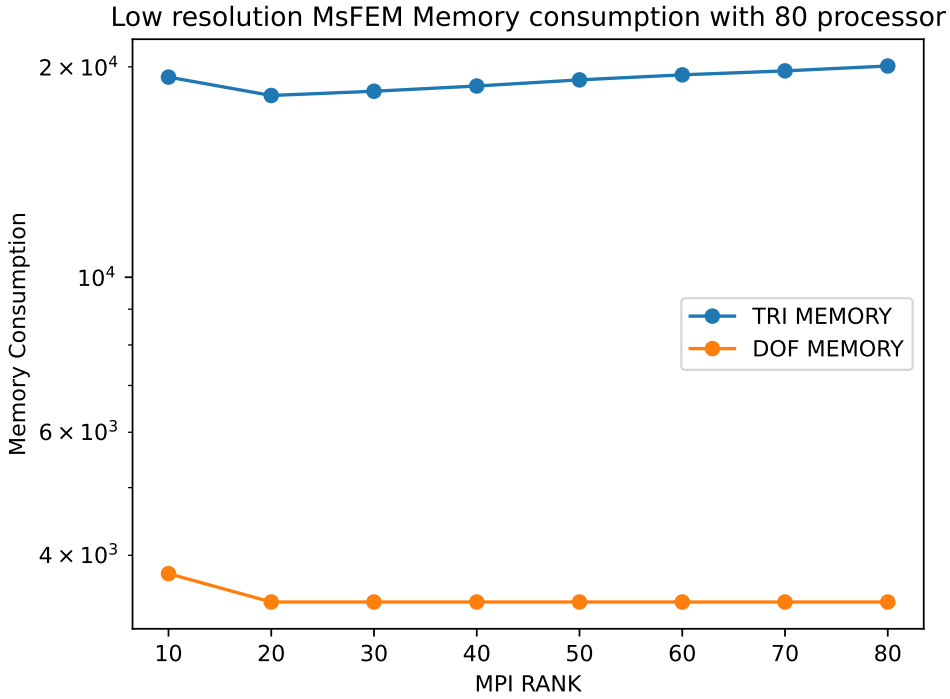


Figure 6.7: Test case 1 Memory consumption for Low resolution MsFEM method.

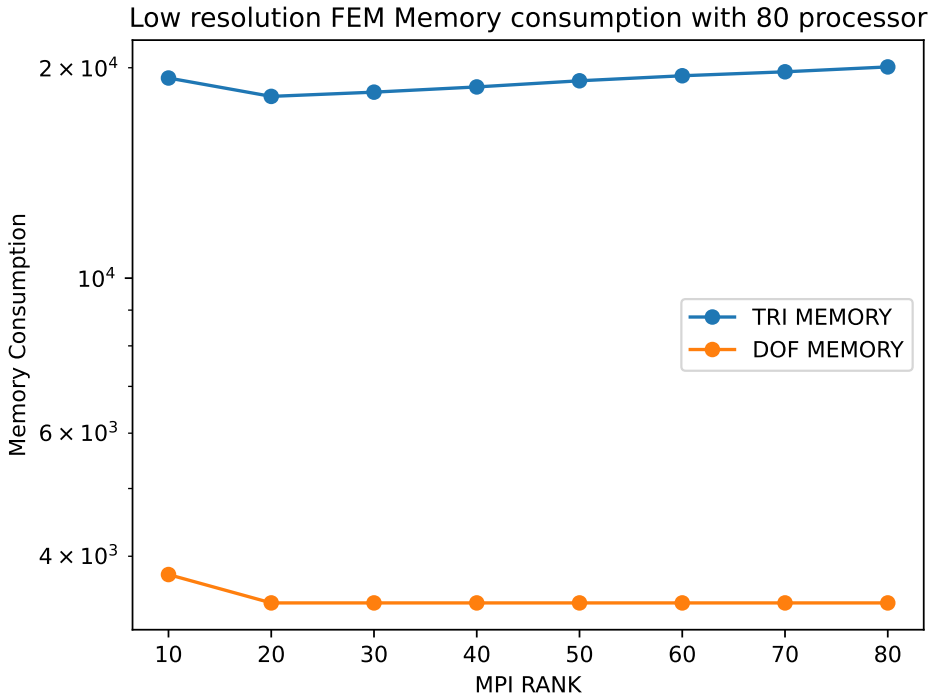


Figure 6.8: Test case 1 Memory consumption for Low resolution FEM method.

The deal.II function calculates memory consumption. Here TRI_MEMORY gives the maximum memory used for triangulation that is mesh generation by the processor. And

DOF_MEMORY gives maximum memory used for degree of freedom by the given process. For example the output below
Running with PETSc on 10 MPI rank(s)...
FEM Coarse TRI MEMORY — 14519
FEM Coarse TRI MEMORY — 19271
FEM Coarse TRI MEMORY — 15951
FEM Coarse TRI MEMORY — 16015
FEM Coarse TRI MEMORY — 15983
FEM Coarse TRI MEMORY — 14519
FEM Coarse TRI MEMORY — 17943
FEM Coarse TRI MEMORY — 17943
FEM Coarse TRI MEMORY — **19335**
FEM Coarse TRI MEMORY — 15951
FEM Coarse DOF MEMORY — 3106
FEM Coarse DOF MEMORY — **3766**
FEM Coarse DOF MEMORY — 3514
FEM Coarse DOF MEMORY — 3170
FEM Coarse DOF MEMORY — 2834
FEM Coarse DOF MEMORY — 3514
FEM Coarse DOF MEMORY — 3170
FEM Coarse DOF MEMORY — 2834
FEM Coarse DOF MEMORY — 3766
FEM Coarse DOF MEMORY — 3106
Here TRI_MEMORY is taken 19335 and DOF_MEMORY is 3766. The goal is to determine which methods consume the most memory. Memory usage at high resolutions decreases as processors increase. Low-resolution FEM and MsFEM consume very few CPUs, and problem resolution is small. It is seen from Figures [(6.6), (6.7) and (6.8)] for high-resolution FEM, the degrees of freedom decrease. It remains constant for low-resolution FEM and MsFEM.

Test case 1: Wall times with respect to DOF

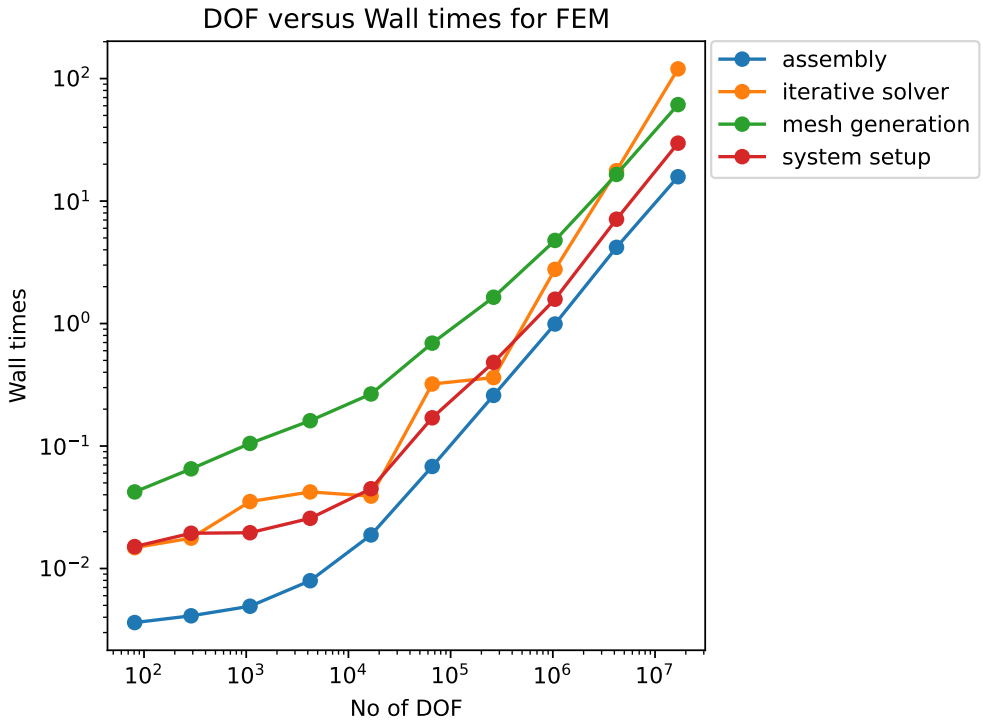


Figure 6.9: Test case 1 Wall times with respect to DOF for FEM method.

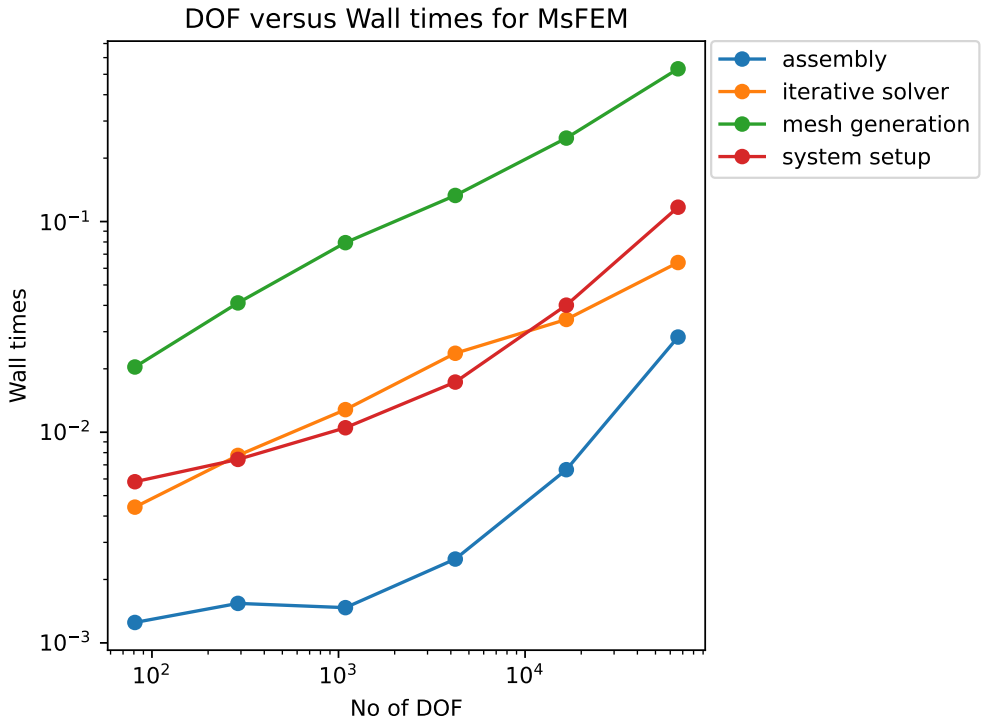


Figure 6.10: Test case 1 Wall times with respect to DOF for MsFEM method.

In Figures [(6.9) and (6.10)] here we run 2 different simulations for test case 1 with 80 MPI rank for FEM and MsFEM. Mesh generation in high-resolution FEM takes the maximum time; however, it increases with the number of degrees of freedom. For MsFEM, wall times increase with respect to DOF. For the remaining steps, such as assembly, the iterative solver, and system setup, there is an increase in wall times for FEM and for low-resolution MsFEM.

Test case 1: Memory Consumption with respect to DOF

Figures [(6.11) and (6.12)] shows memory consumption for FEM and MsFEM methods. Memory usage for FEM increases as the number of DOFs increases. MsFEM consumes very small memory and problem resolution is small.

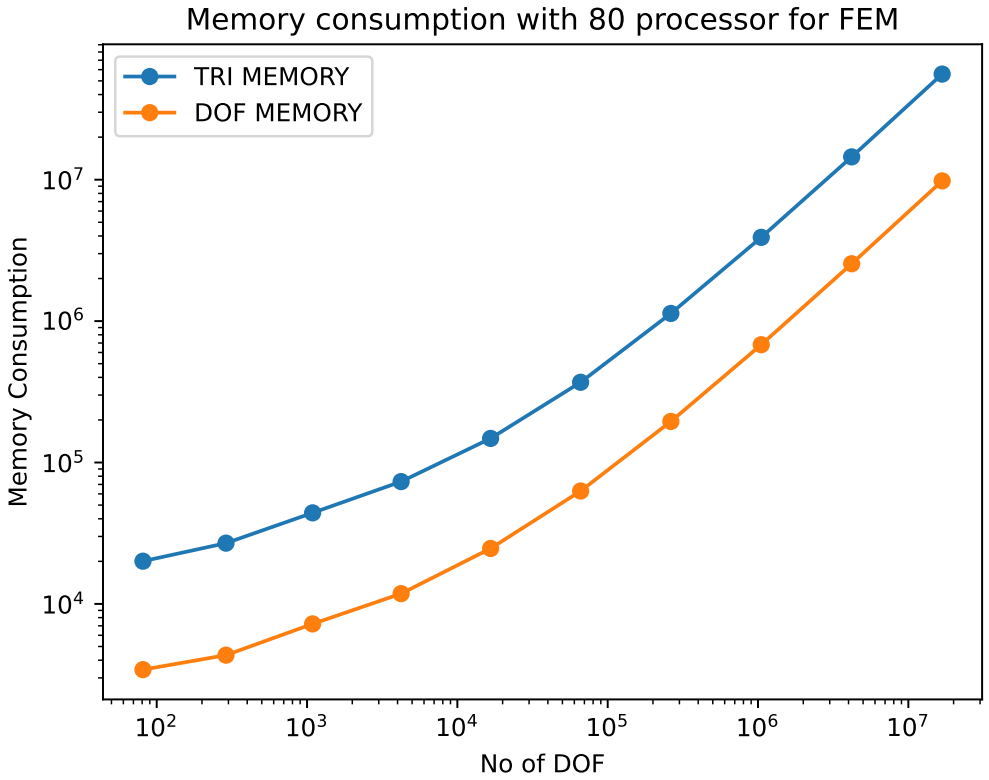


Figure 6.11: Test case 1 Memory Consumption with respect to DOF for FEM method.

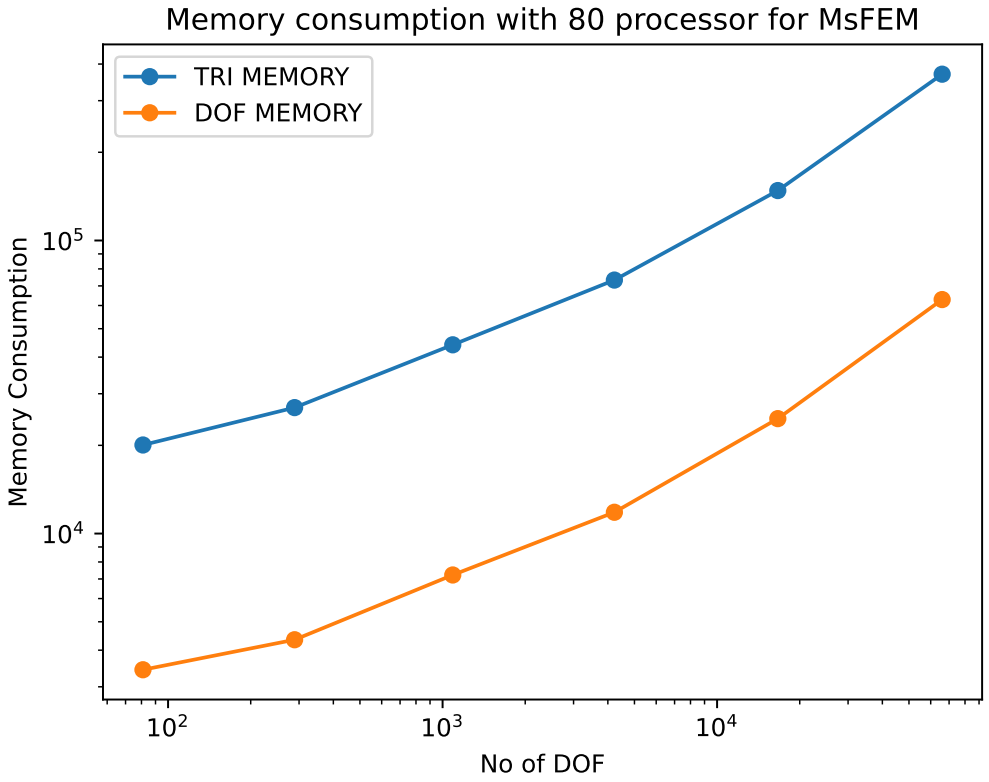


Figure 6.12: Test case 1 Memory Consumption with respect to DOF for MsFEM method.

6.1.2 Test case 2

Let $\Omega = [0,1]$ and $\varepsilon = 0.6666$ and k=23 . We define Poisson's Equation

$$-\nabla \cdot (a_\varepsilon \nabla u) = f \quad \text{in} \quad \Omega$$

Dirichlet condition is

$$u = (x-0.5)^2 + (y-0.5)^2 \quad \text{on} \quad left \quad and \quad bottom \quad \partial\Omega$$

Neumann Condition is

$$\nabla u.n = cos(2\pi x) * cos(2\pi y) \quad \text{on} \quad right \quad and \quad top \quad \partial\Omega$$

where

Diffusion coefficient

$$a_\varepsilon(x,t) = \left(I_2 - \varepsilon \begin{bmatrix} 0.5 * sin(2\pi kx) + 0.5 * sin(2\pi ky) & 0 \\ 0 & 0.5 * sin(2\pi kx) + 0.5 * sin(2\pi ky) \end{bmatrix} \right)$$

and

$$f = 2$$

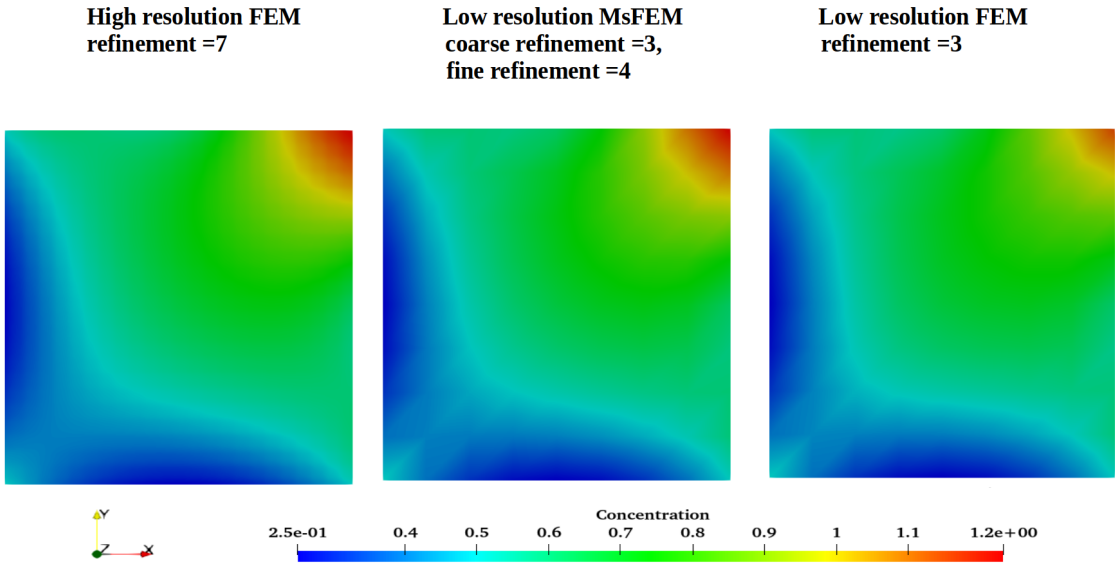


Figure 6.13: Test case 2 Solution of High resolution FEM, Low resolution FEM and Low resolution MsFEM for stationary diffusion equation.

It is obvious that a reference solution with high resolution yields the best results, but computation power and memory are increased.. As shown Figure (6.13) Low-resolution FEM does not provide accurate results. With low computation and memory demands, the multiscale finite element MsFEM exhibits excellent performance. It captures the subgrid scale features in the solution.

Test case 2: Error Table

Here the error is calculated with the difference in the reference solution which is high resolution FEM.

Error Analysis			
Simulation Type	L^2 Error	H^1 Error	L^∞ Error
Low resolution FEM	0.0159	0.073	0.14
Low resolution MsFEM	0.0025	0.034	0.056

Table 6.2: Test case 2 Error in simulation with Low resolution FEM, High resolution FEM and Low resolution MsFEM.

In the above Table (6.2), the error is calculated using high-resolution FEM as the reference solution. The low-resolution MsFEM has the least error, whereas the low-resolution FEM has the most error.

Test case 2: Wall times with respect to MPI rank

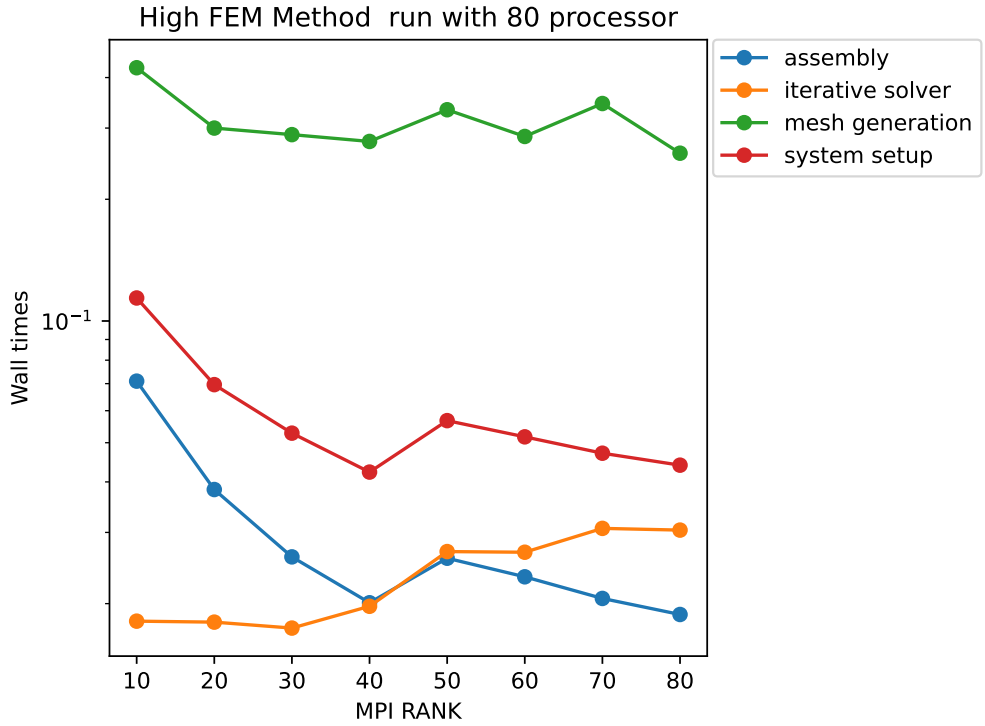


Figure 6.14: Test case 2 Wall times with respect to MPI rank for High resolution FEM method.

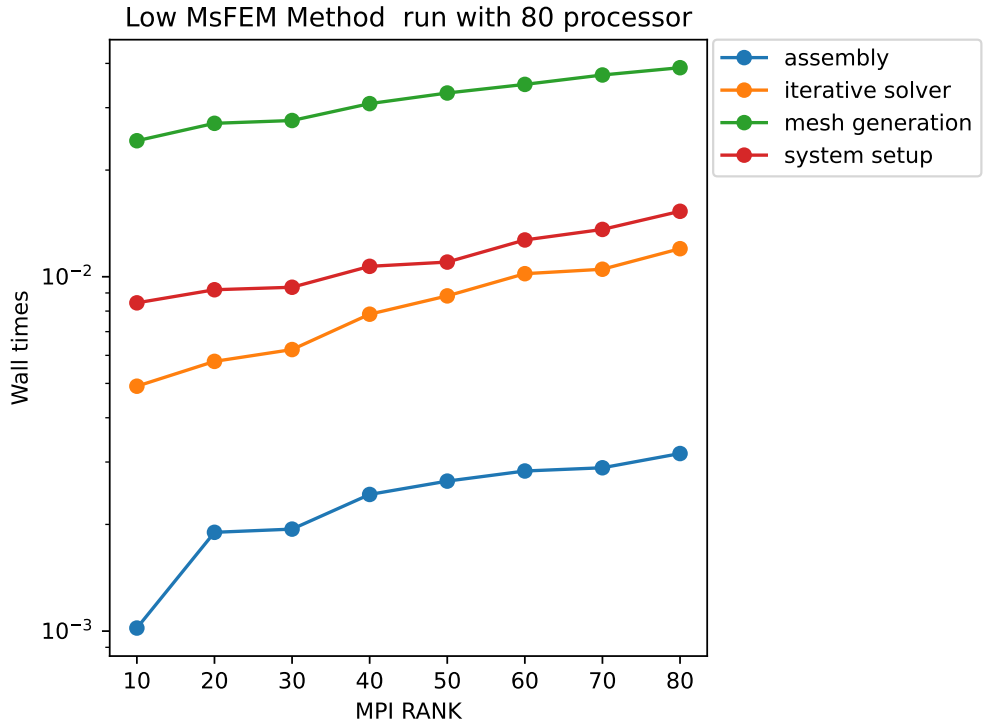


Figure 6.15: Test case 2 Wall times with respect to MPI rank for Low resolution MsFEM method.

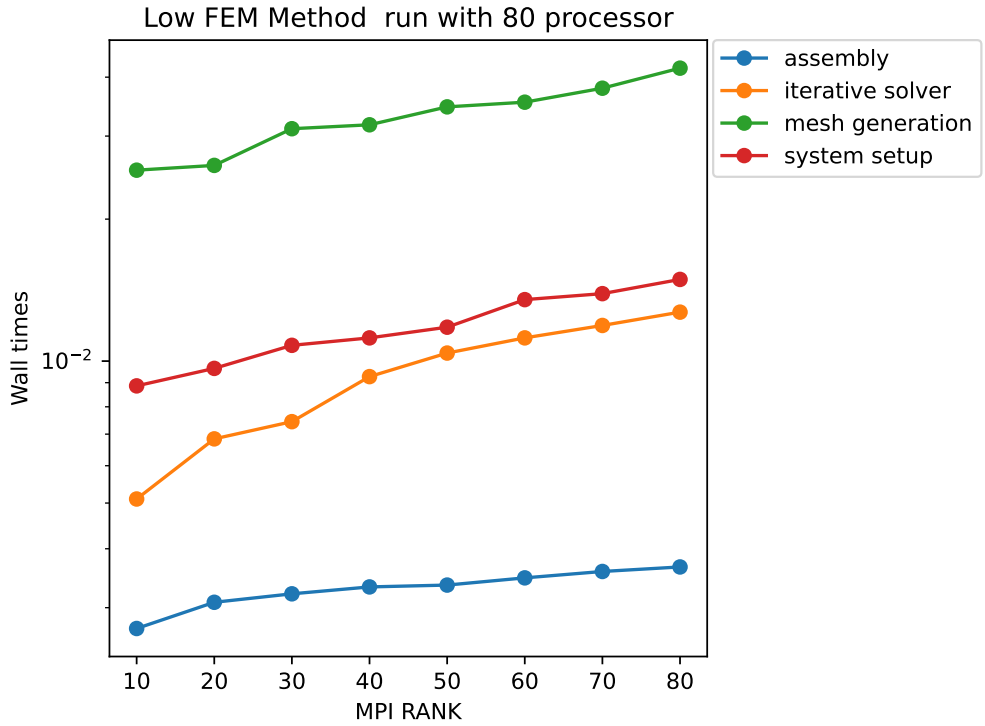


Figure 6.16: Test case 2 Wall times with respect to MPI rank for Low resolution FEM method.

In Figure [(6.14), (6.15) and (6.16)] here we run 3 different simulations for test case 2 with 81 degrees of freedom for low-resolution FEM. The high-resolution FEM has 16641 degrees of freedom, while the multiscale FEM has 81 degrees of freedom. Mesh generation in the high-resolution FEM takes maximum time; however, it decreases with the number of processors, except for the 50 and 70 MPI ranks. For low-resolution FEM and low-resolution MsFEM, there are 81 degrees of freedom. Because of the size of the problem, mesh generation does not vary much. For the remaining steps, such as assembly, the iterative solver and system setup decrease mostly in high-resolution FEM. Low-resolution FEM increases iterative solvers, setup and assembly. In the case of a low-resolution MsFEM assembly, the system setup and iterative solver increase with MPI rank.

Test case 2: Memory Consumption

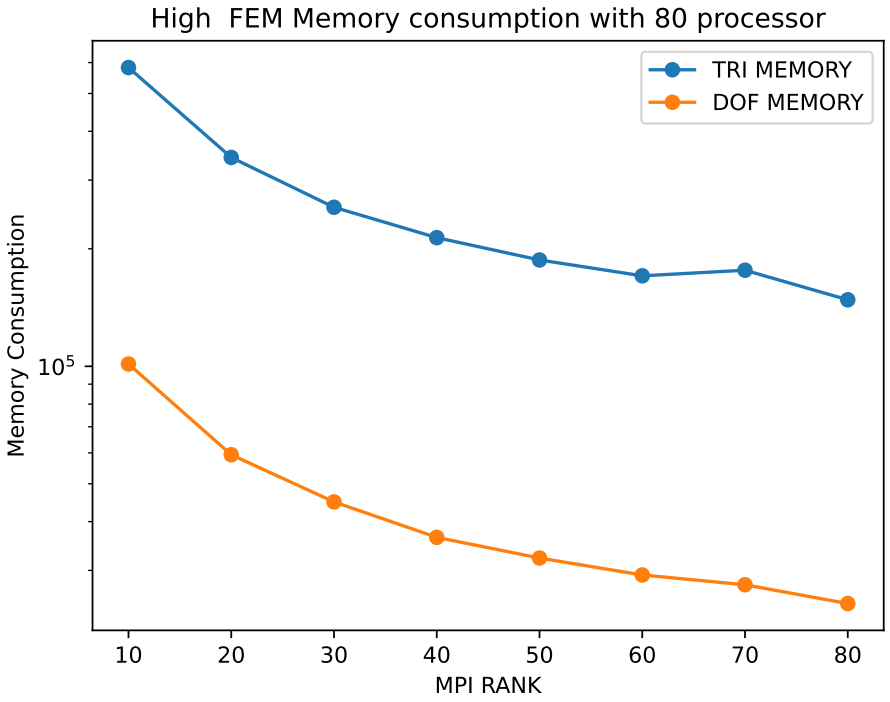


Figure 6.17: Test case 2 Memory consumption for High resolution FEM.

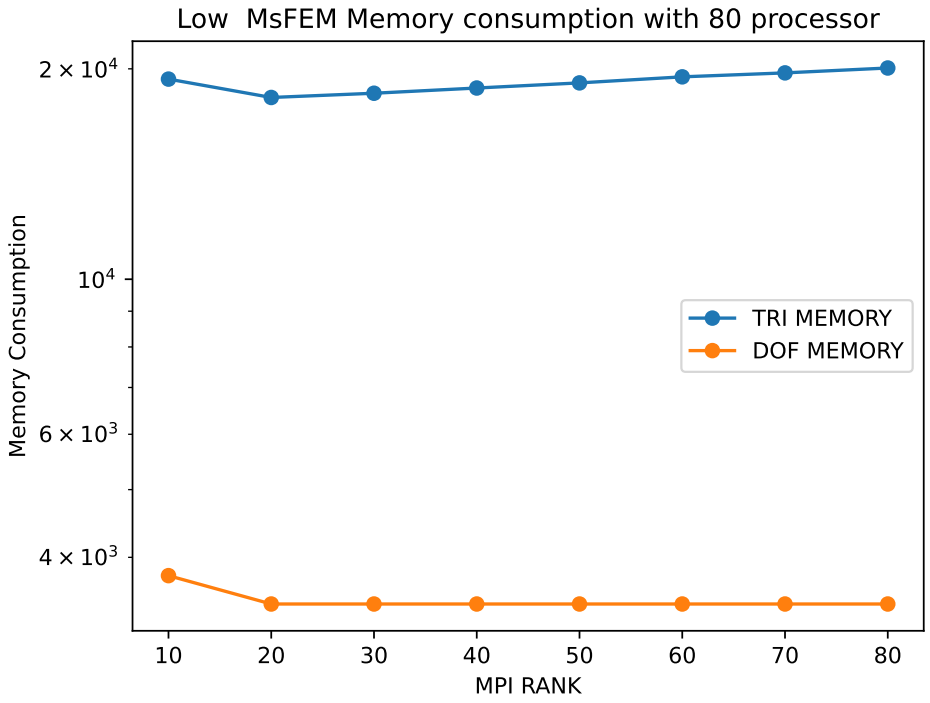


Figure 6.18: Test case 2 Memory consumption for Low resolution MsFEM.

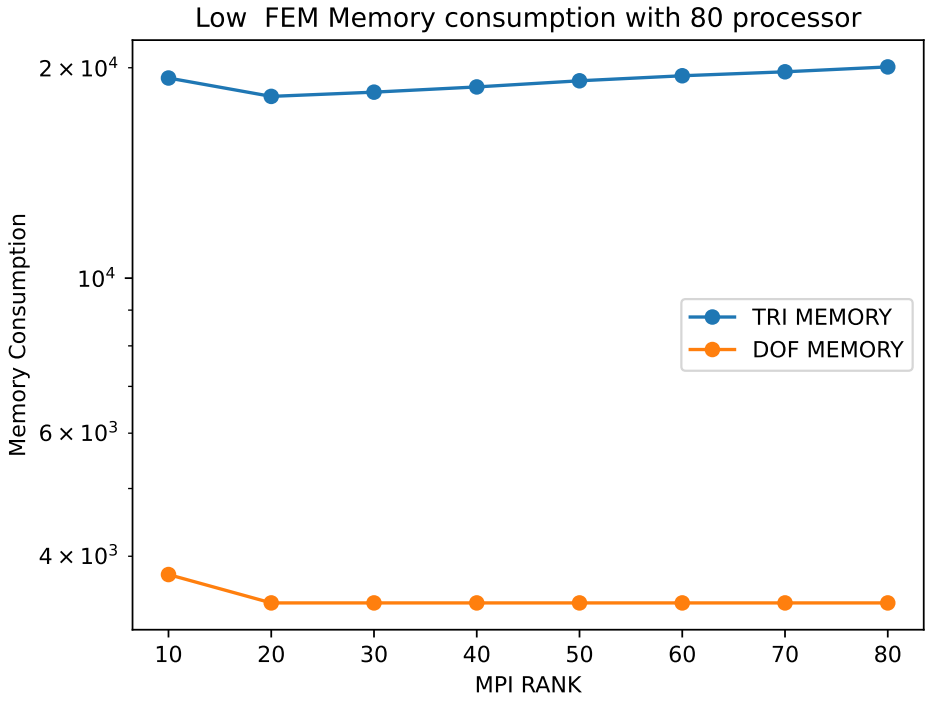


Figure 6.19: Test case 2 Memory consumption for Low resolution FEM.

As shown in Figure [(6.17), (6.18) and (6.19)], low-resolution FEM and MsFEM consume less memory than high-resolution FEM.

Test case 2: Wall times with respect to DOF

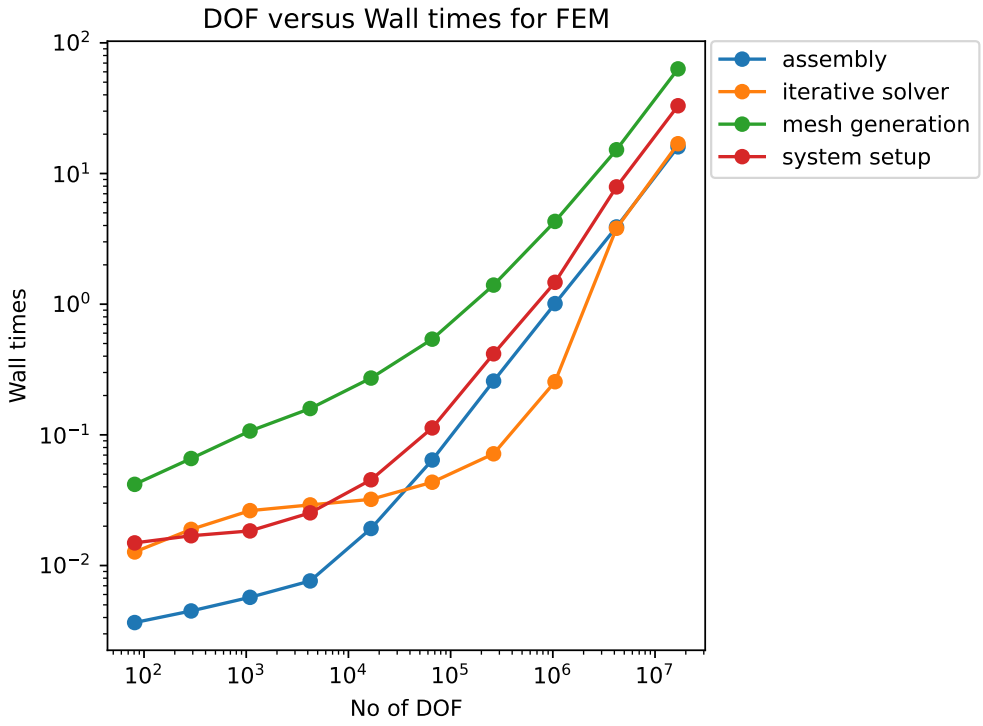


Figure 6.20: Test case 2 Wall times with respect to DOF for FEM method.

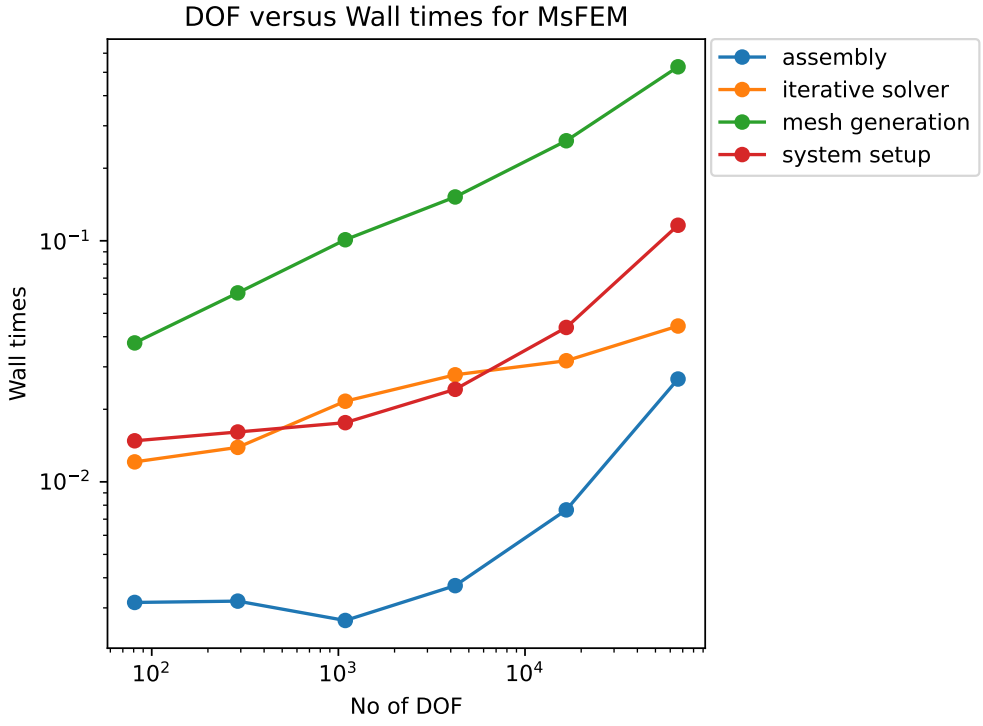


Figure 6.21: Test case 2 Wall times with respect to DOF for MsFEM method.

In Figures [(6.20) and (6.21)] here we run 2 different simulations for test case 2 with 80 MPI rank for low-resolution FEM. Mesh generation in high-resolution FEM takes the most time; however, it increases with the number of degrees of freedom. For MsFEM, wall times increase with respect to DOF. For the remaining steps, such as assembly, the iterative solver, and system setup, wall times increase for FEM. In the case of a low-resolution MsFEM assembly, system setup and iteration increase with wall time.

Test case 2: Memory Consumption with respect to DOF

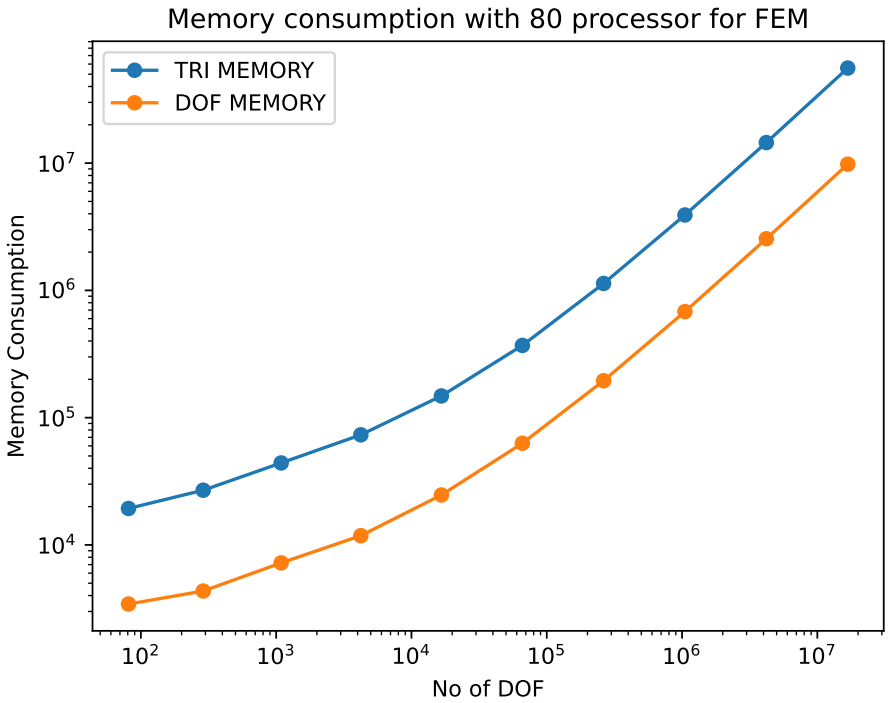


Figure 6.22: Test case 2 Memory Consumption with respect to DOF for FEM method.

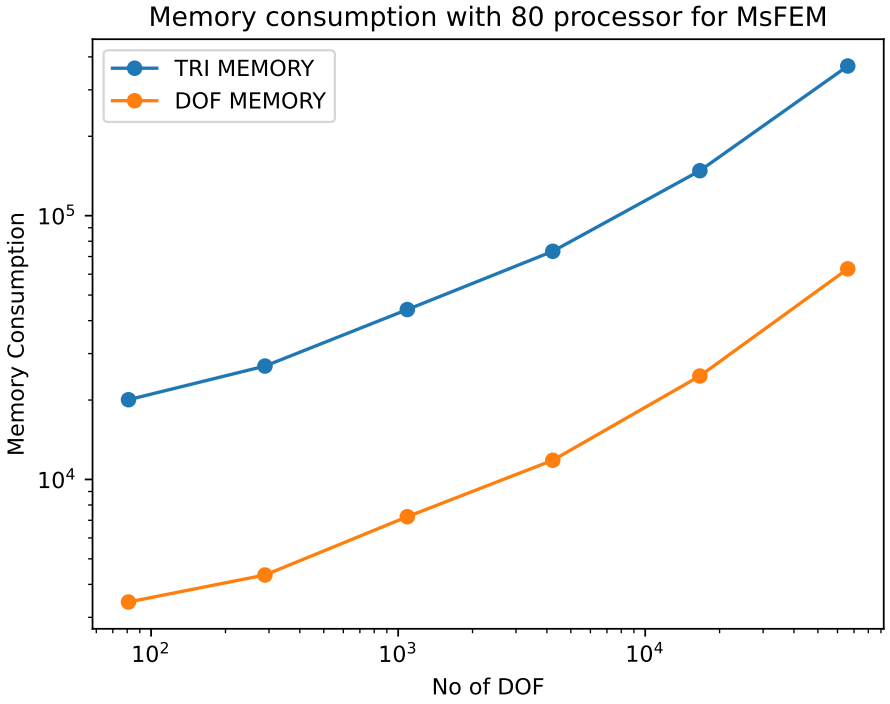


Figure 6.23: Test case 2 Memory Consumption with respect to DOF for MsFEM method.

As shown in Figures [(6.22) and (6.23)] , memory consumption increases with increase in DOF in FEM and MsFEM increases with increase in DOF in MsFEM. Both in Test 1 and Test 2 the memory is maximum memory and not total memory hence the figure is not linear.

6.1.3 Test case 3

Here the domain is square. Model problem advection-diffusion involve periodic boundaries on left-right and top-bottom and Dirichlet boundary conditions on all boundaries.

$$\partial_t u + c_\delta \cdot \nabla u = \nabla \cdot (a_\varepsilon \nabla u) + f$$

Dirichlet condition is

$$u = \frac{1}{2\sqrt{(2\pi)^2}} \sum_{i=1}^{2} exp\{-\frac{1}{2}(x - r_i)^T (x - r_i)\} \partial\Omega$$

where $r_i = [\frac{i}{3}, \frac{1}{2}]$, i=1,2

Initial condition at $t = 0$

$$u(x,y) = \frac{1}{2\sqrt{(2\pi)^2}} \sum_{i=1}^{2} exp\{-\frac{1}{2}(x-r_i)^T(x-r_i)\}$$

where $r_i = [\frac{i}{3}, \frac{1}{2}]$, i=1,2
Velocity

$$c_\delta(x,t) = 2 \begin{bmatrix} sin(30\pi x)cos(30\pi y) \\ -cos(30\pi x)sin(30\pi y) \end{bmatrix}$$

Diffusion coefficient

$$a_\varepsilon(x,t) = 0.001 * \left(1 - 0.999 \begin{bmatrix} sin(60\pi x) & 0 \\ 0 & sin(60\pi y) \end{bmatrix}\right)$$

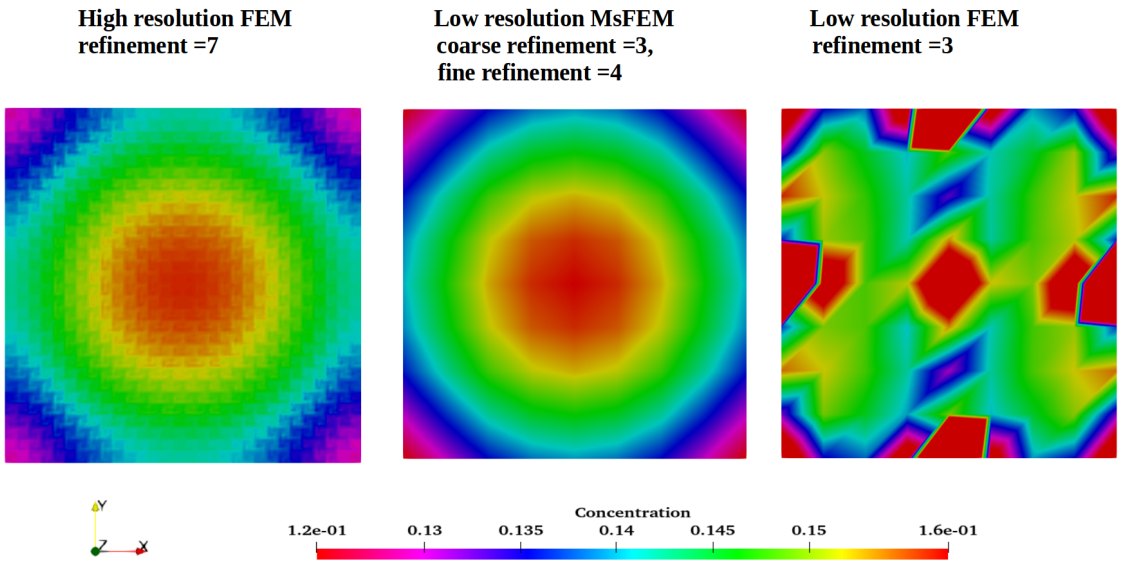


Figure 6.24: Test case 3 Solution of High resolution FEM, Low resolution FEM and Low resolution MsFEM for advection-diffusion equation.

In the given square domain with high advection it is found in Figure (6.24) that low resolution MsFEM shows better results when compared to high resolution FEM compared to low resolution FEM. The coarse line of mesh are seen in low resolution MsFEM. The low resolution FEM solution does not converge due to high advection.

Test case 3: Error Table

Here the error is calculated with the difference in the reference solution which is high resolution FEM.

Error Analysis			
Simulation Type	L^2 Error	H^1 Error	L^∞ Error
Low resolution FEM	0.0085	0.3920	0.0693
Low resolution MsFEM	0.0003	0.031	0.0004

Table 6.3: Test case 3 Error in simulation with Low resolution FEM, High resolution FEM and Low resolution MsFEM.

According to Table (6.3), low resolution FEM has more error compared to low resolution MsFEM.

Test 3 Wall Times with Direct and Iterative solvers

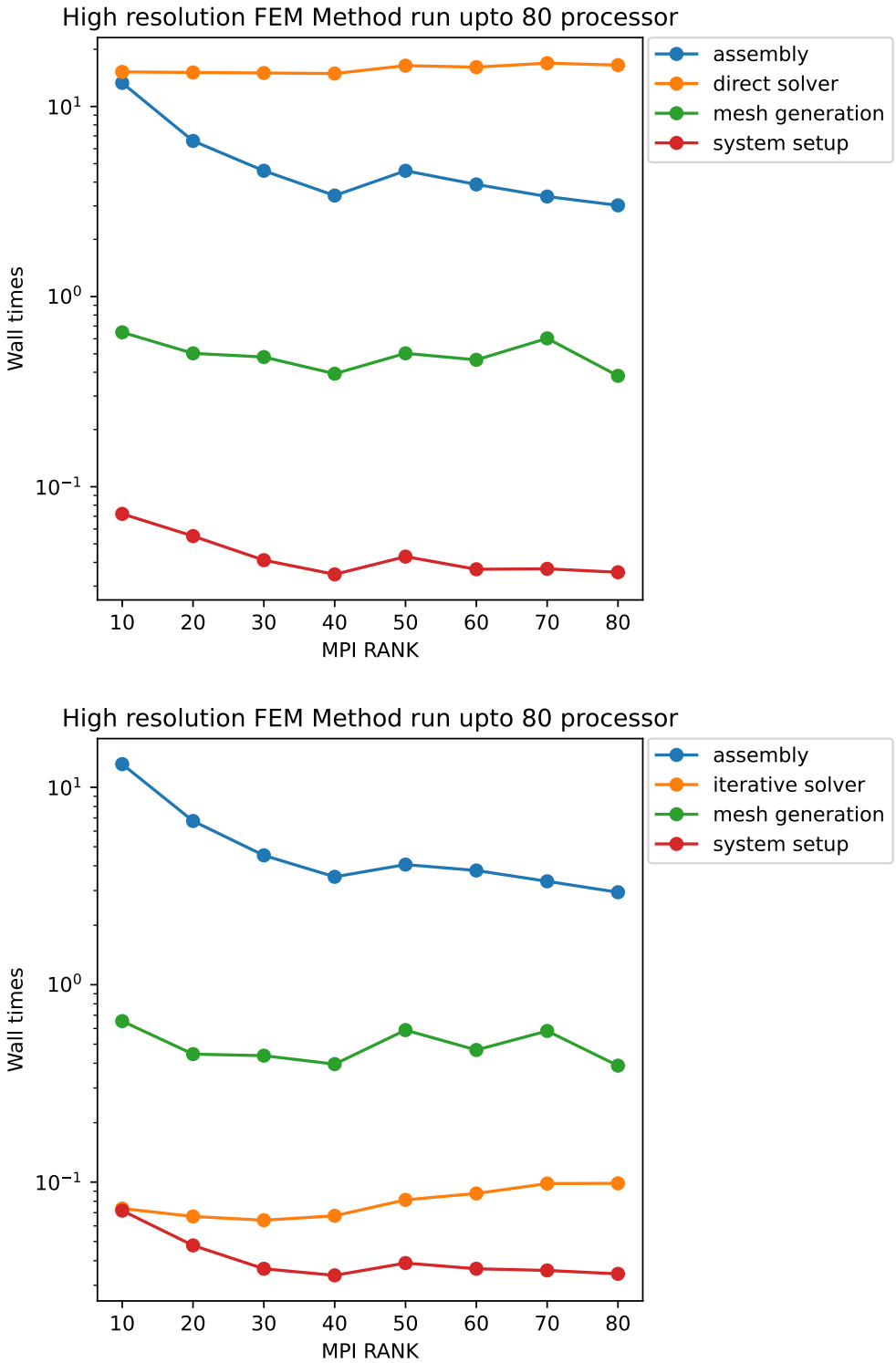


Figure 6.25: Test Case 3 Wall times with respect to MPI rank for High resolution FEM for direct and iterative solver.

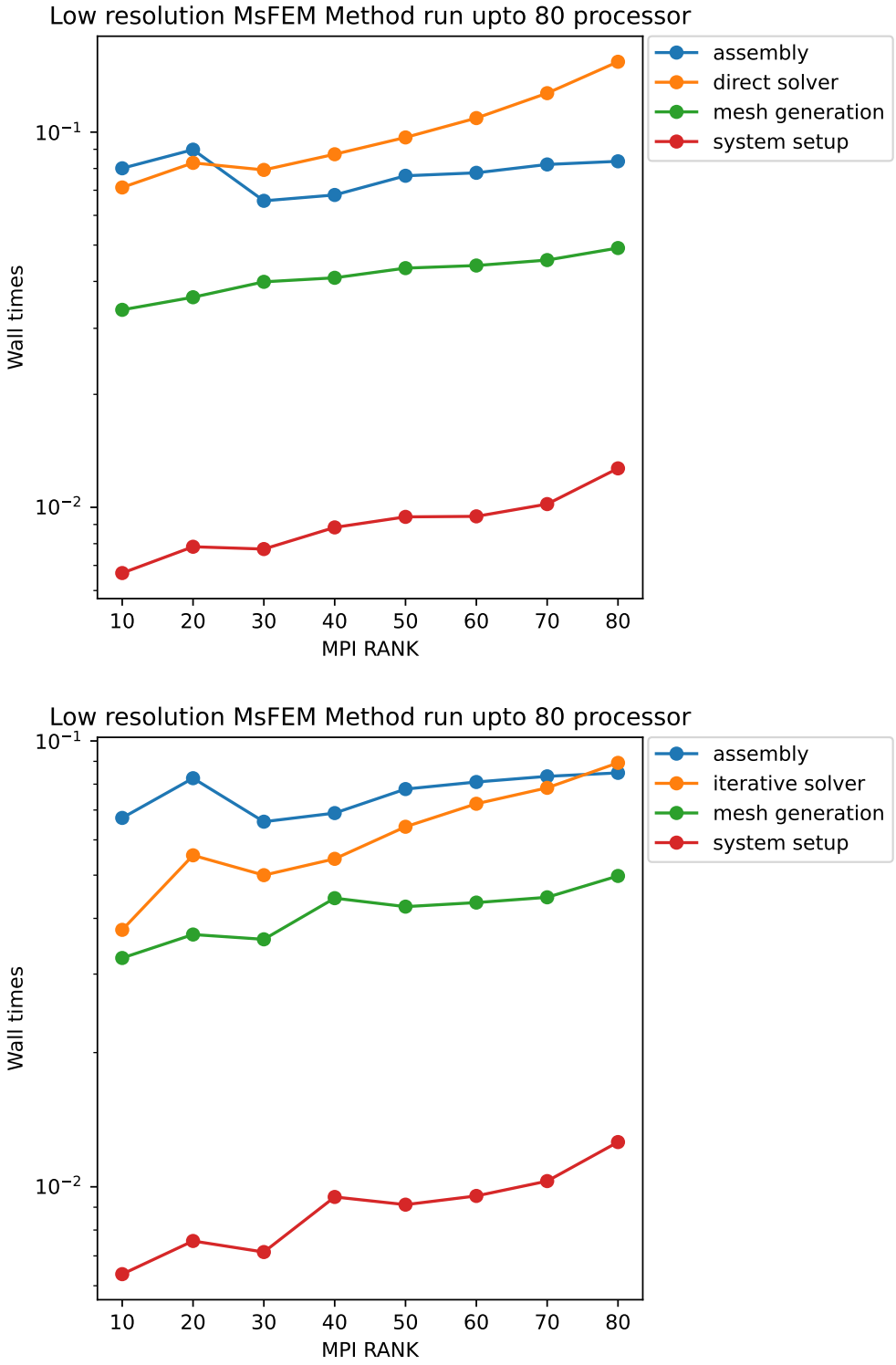


Figure 6.26: Test Case 3 Wall times with respect to MPI rank for Low resolution MsFEM for direct and iterative solver.

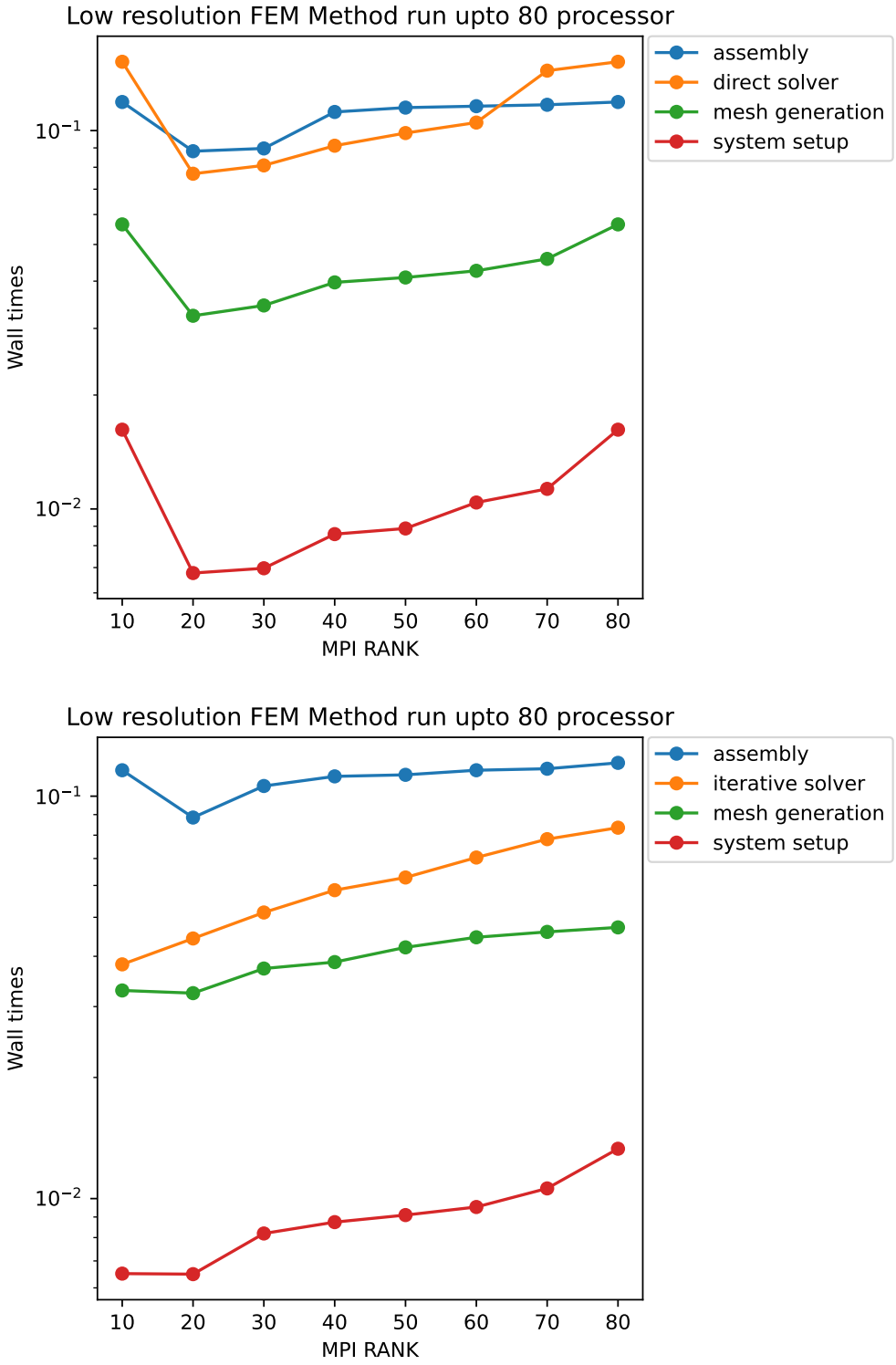


Figure 6.27: Test Case 3 Wall times with respect to MPI rank for Low resolution FEM for direct and iterative solver.

In Figures [(6.25), (6.26) and (6.27)] three different simulations for test case 3 with 81 degrees of freedom for low resolution FEM, 16641 degrees of freedom for high resolution FEM and multiscale finite element method with degrees of freedom 81 are run with direct and iterative solvers. In low resolution MsFEM and high resolution FEM solutions, the

direct solver requires the longest time, whereas assembly consumes the longest time in low resolution MsFEM. For high resolution assembly, it takes the longest with an iterative algorithms. Assembly and iterative solver assembly take the most time for low resolution MsFEM and low resolution FEM.

Test Case 3 Memory Consumption

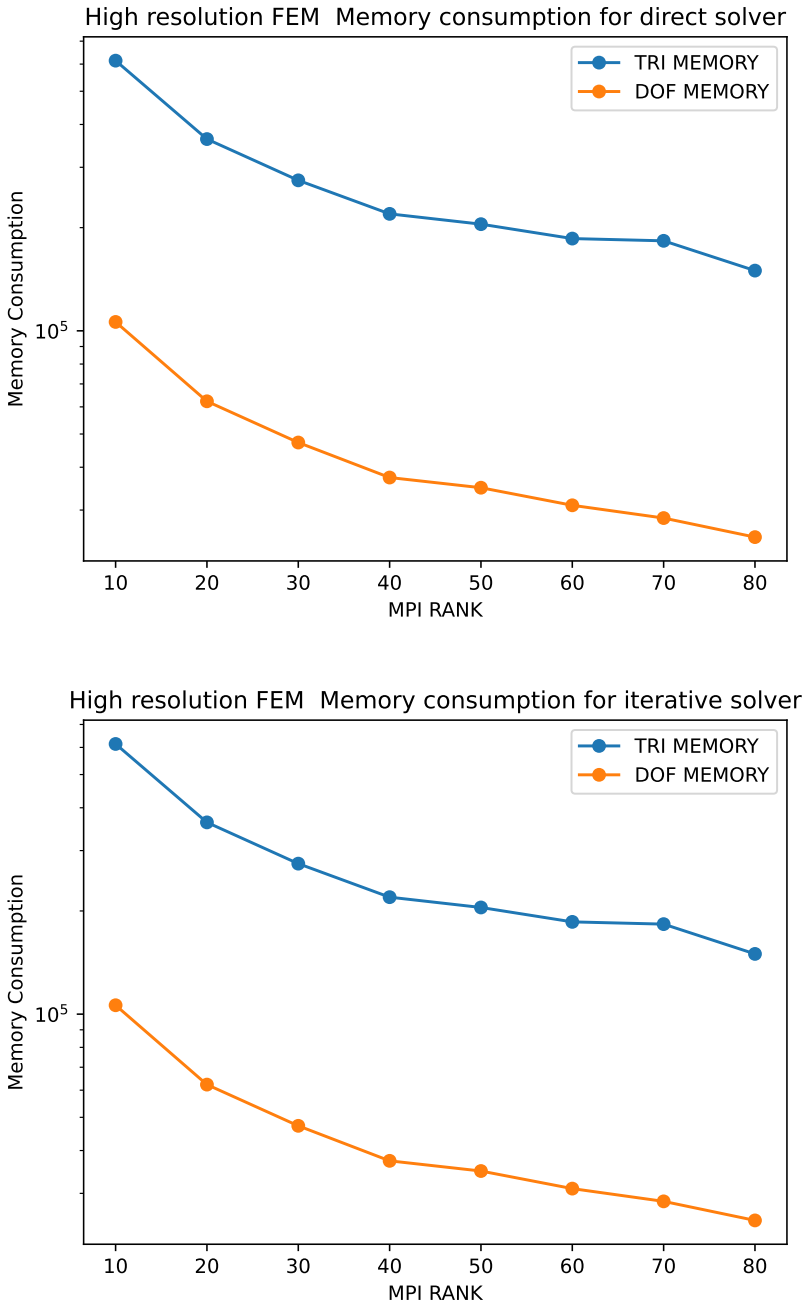


Figure 6.28: Test Case 3 Memory consumption for High resolution FEM method for direct and iterative solver.

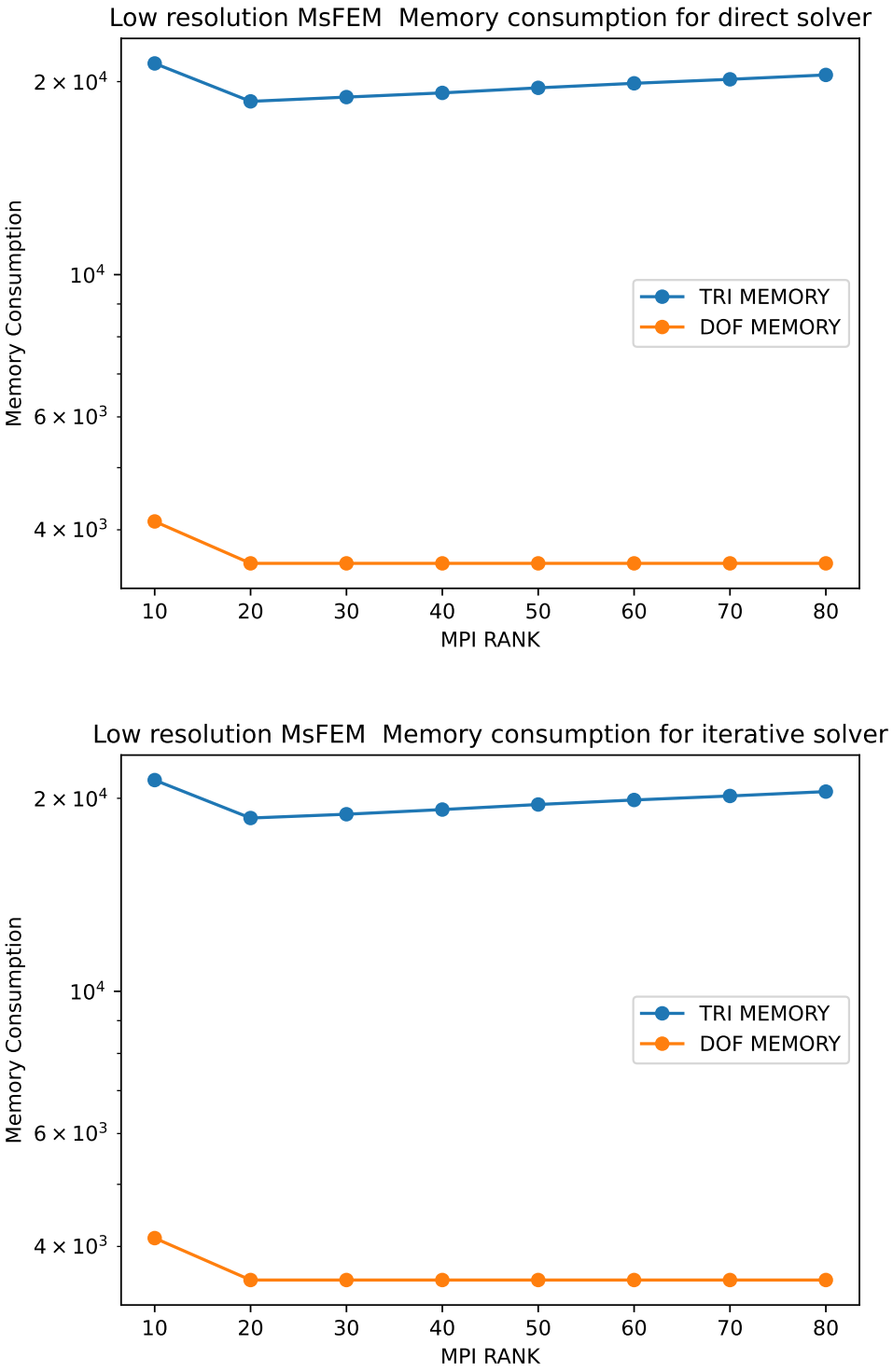


Figure 6.29: Test Case 3 Memory consumption for Low resolution MsFEM method for direct and iterative solver.

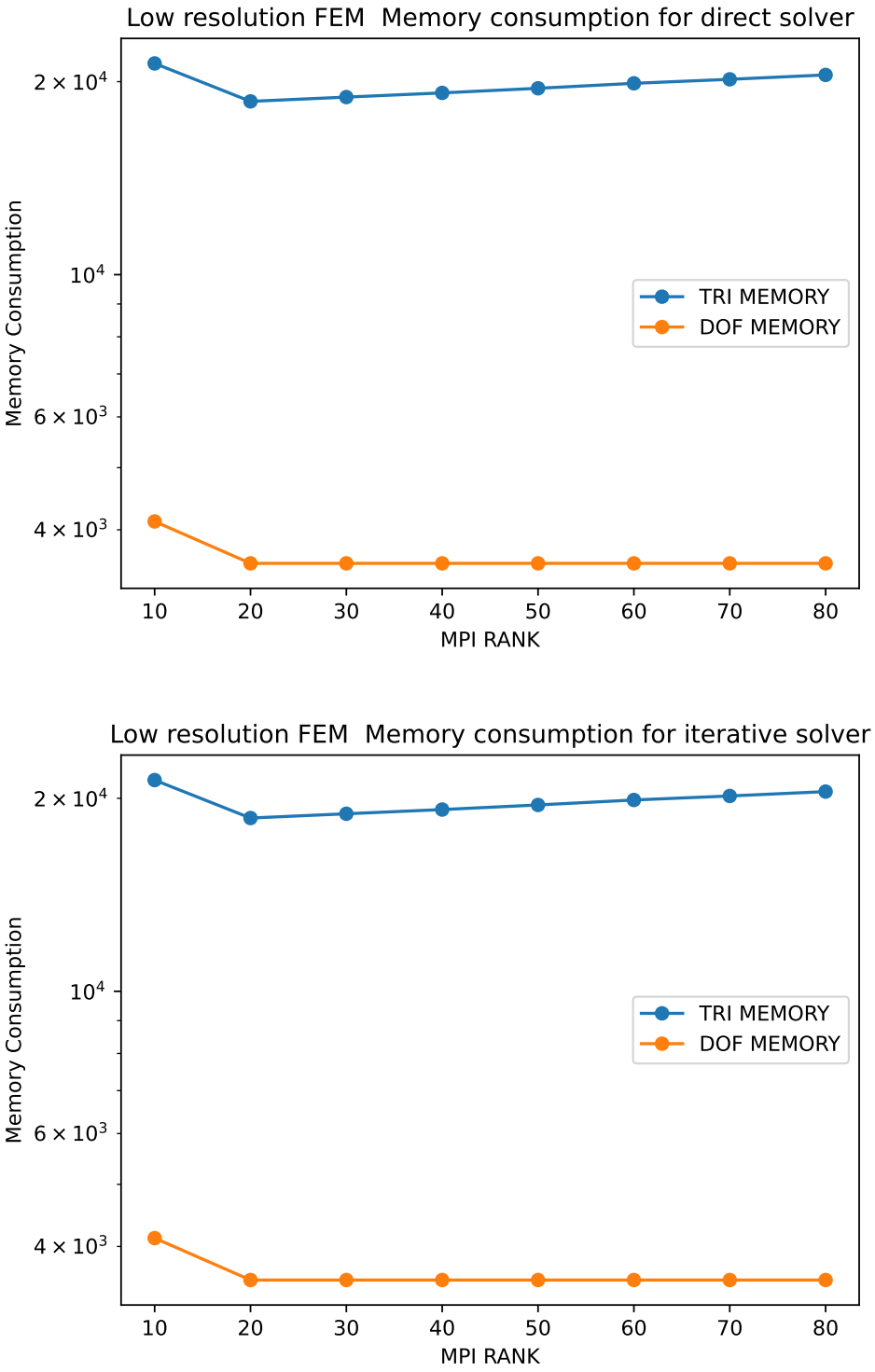


Figure 6.30: Test Case 3 Memory consumption for Low resolution FEM method for direct and iterative solver.

According to Figures [(6.28), (6.29) and (6.30)], memory consumption decreases as MPI rank increases for high-resolution FEM. For low-resolution FEM and MsFEM, there is a slow increase in memory consumption, but it decreases for high-resolution FEM due to the problem size.

6.1.4 Test case 4

Here the domain is square. The advection-diffusion problem involves a non-periodically oscillating coefficient function with the Neumann condition on odd boundaries and the Dirichlet boundary on even boundaries

$$\partial_t u + c_\delta \cdot \nabla u = \nabla \cdot (a_\varepsilon \nabla u) + f$$

Dirichlet condition is

$$u = (x-0.5)^2 + (y-0.5)^2 \quad \text{on} \quad left \quad and \quad bottom \quad \partial\Omega$$

Neumann Condition is

$$\nabla u.n = 0 \quad \text{on} \quad right \quad and \quad top \quad \partial\Omega$$

Initial condition

$$u_0(x) = (x-0.5)^2 + (y-0.5)^2$$

Velocity

$$c_\delta(x,t) = \begin{bmatrix} 10 \\ 10 \end{bmatrix}$$

Diffusion coefficient

$$a_\varepsilon(x,t) = 0.1 * \left(I_2 - 0.9999 \begin{bmatrix} sin(60\pi x) & 0 \\ 0 & sin(60\pi y) \end{bmatrix} \right)$$

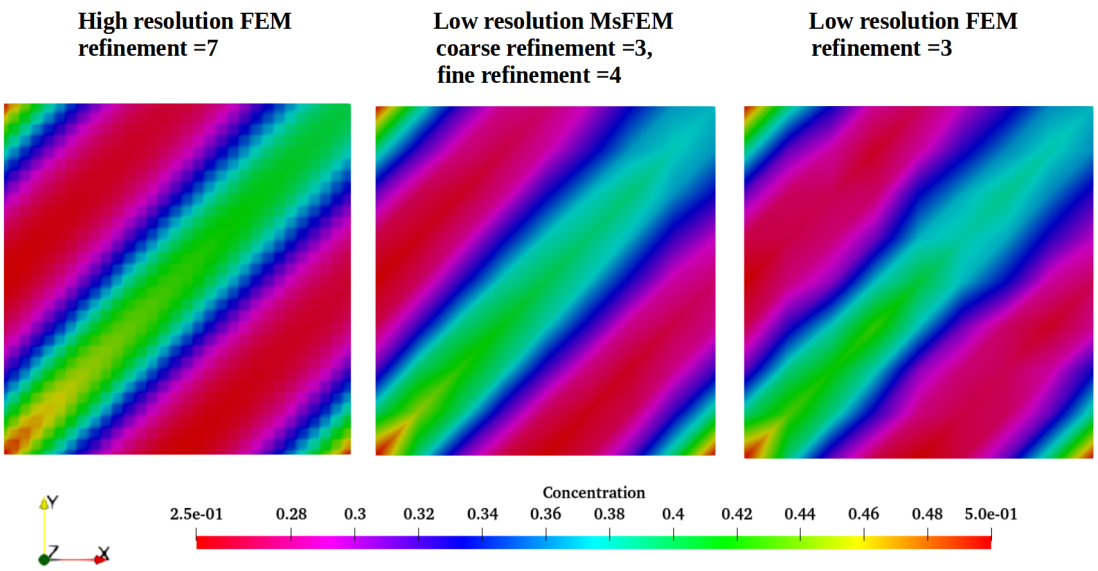


Figure 6.31: Test case 4 Solution of High resolution FEM, Low resolution FEM and Low resolution MsFEM for advection-diffusion equation.

In the given square domain with high advection it is found in Figure (6.31) that low resolution MsFEM shows better results when compared to high resolution FEM compared to low resolution FEM.

Test case 4: Error Table

Here the error is calculated with the difference in the reference solution which is a high resolution FEM.

Error Analysis			
Simulation Type	L^2 Error	H^1 Error	L^∞ Error
Low resolution FEM	0.0052	0.2646	0.0367
Low resolution MsFEM	0.0038	0.2618	0.036

Table 6.4: Test case 4 Error in simulation with Low resolution FEM, High resolution FEM and Low resolution MsFEM.

In the above Table (6.4), the error is calculated using a high resolution FEM solution as the reference solution. The low-resolution MsFEM has less error in L^2 and H^1, compare to the low-resolution FEM.

Test Case 4 Wall Times for direct and iterative solvers

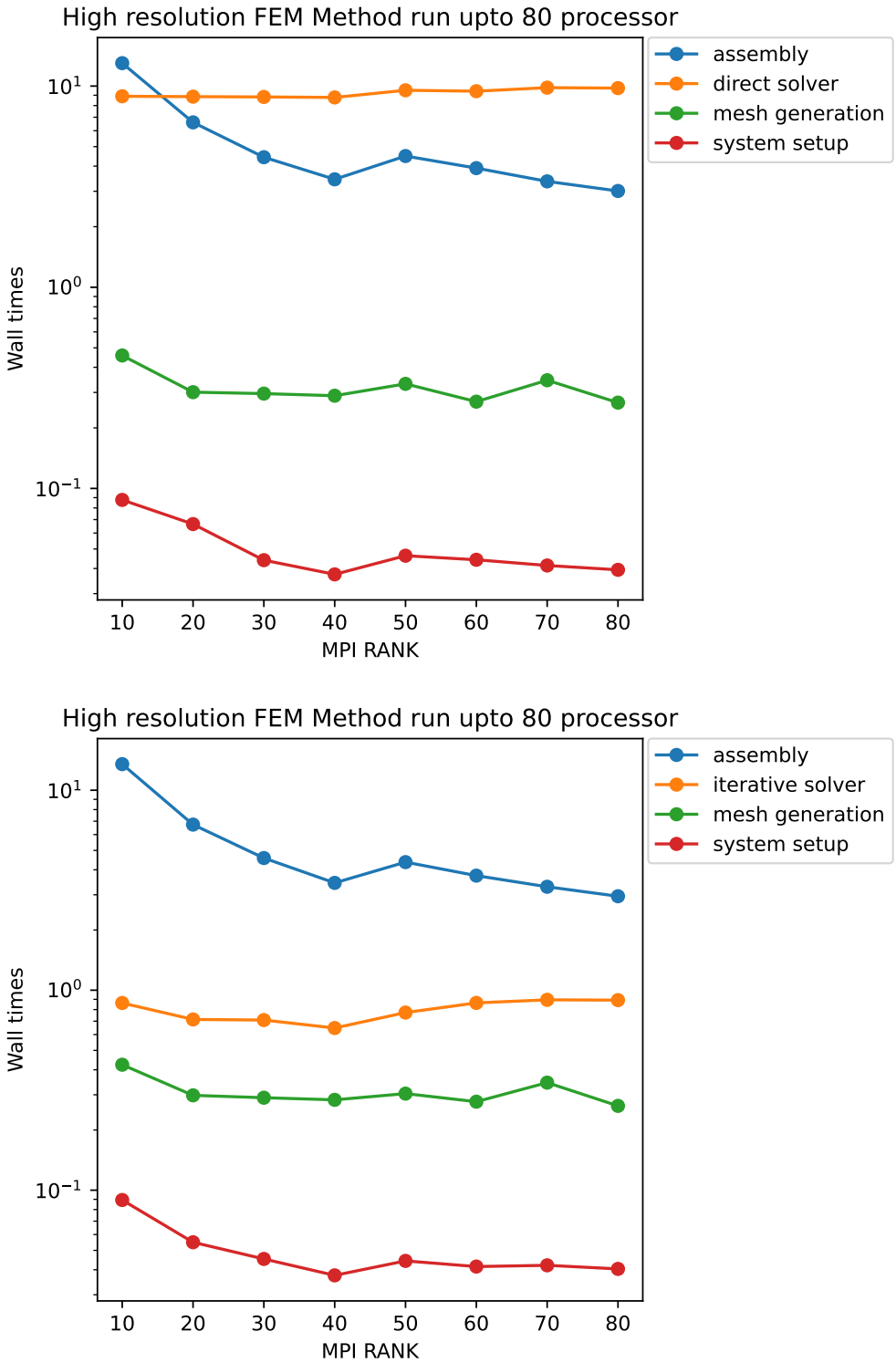


Figure 6.32: Test Case 4 Wall times with respect to MPI rank for High resolution FEM method.

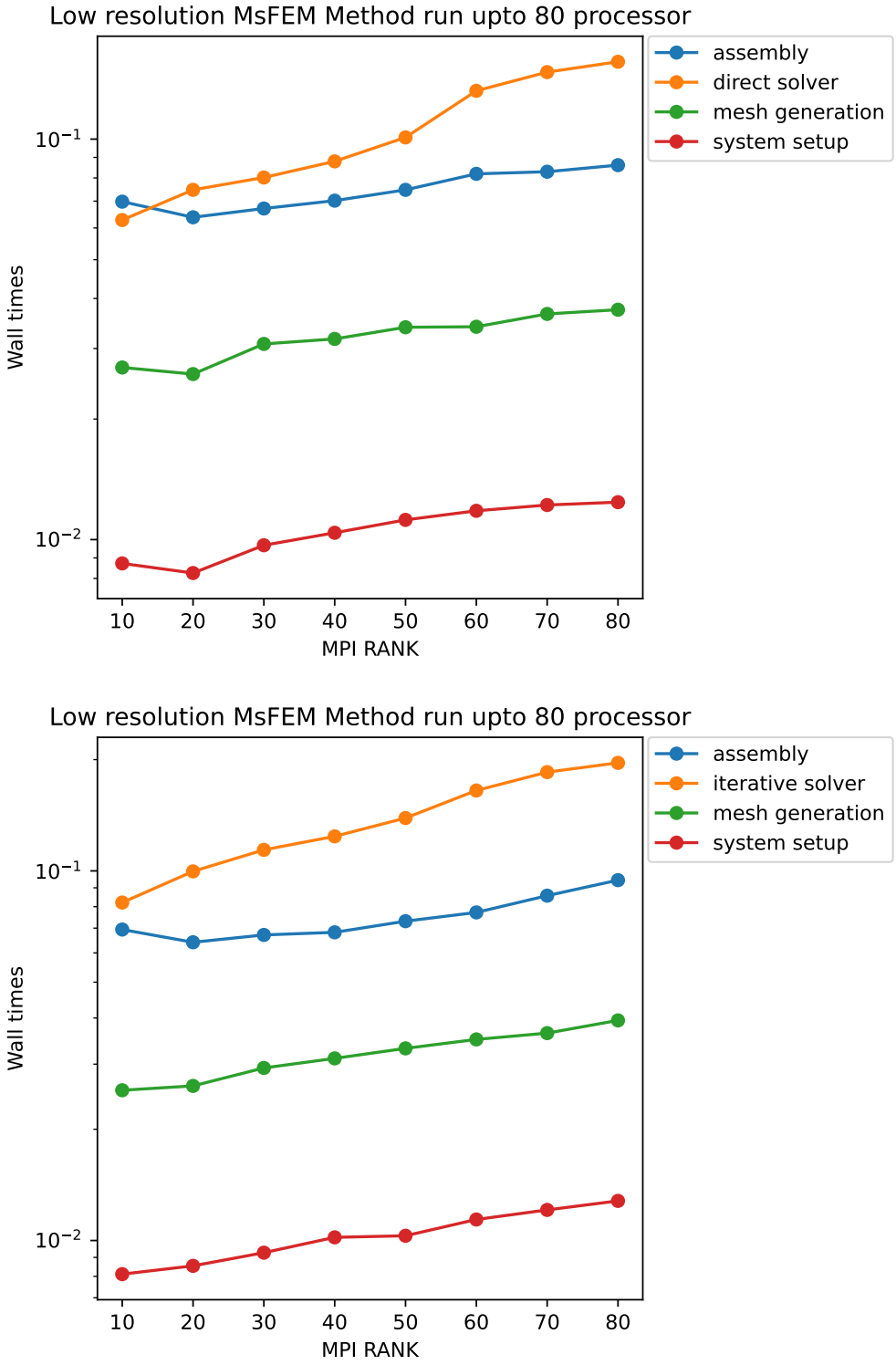


Figure 6.33: Test Case 4 Wall times with respect to MPI rank for Low resolution MsFEM method.

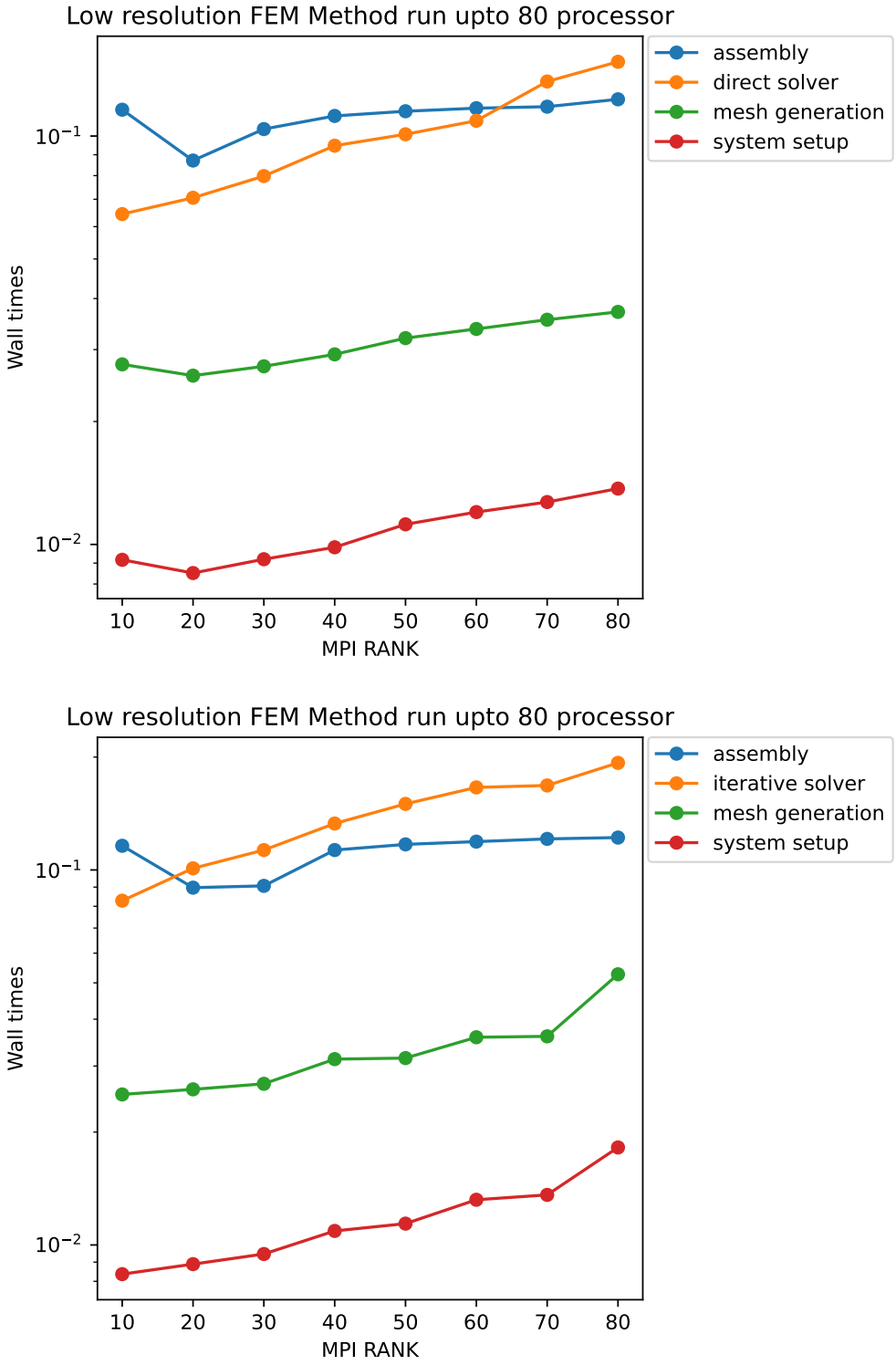


Figure 6.34: Test Case 4 Wall times with respect to MPI rank for Low resolution FEM method.

In Figures [(6.32), (6.33) and (6.34)] are plots of all three simulation runs for the MPI run versus wall time. It is clear that the direct solver has the maximum wall time in high-resolution FEM, whereas assembly requires more time with an iterative algorithm solution. For low resolution MsFEM, direct solvers and iterative solvers require more

time than assembly, mesh generation, and system setup. In the case of low resolution FEM assemble a system takes longer for the direct algorithm solution but for iterative algorithm solution iterative solver takes the longest time.

Test Case 4 Memory Consumption

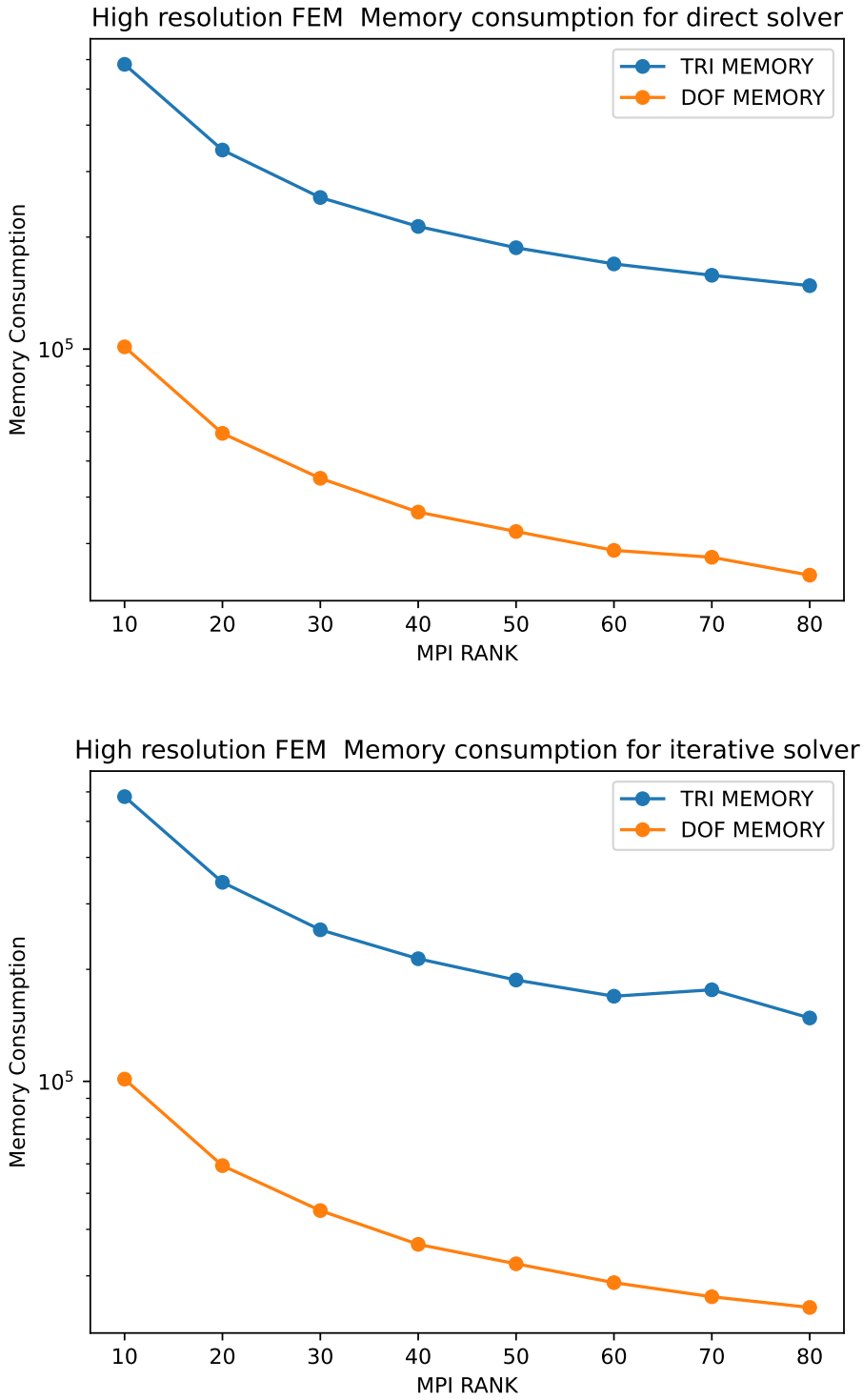


Figure 6.35: Test Case 4 Memory consumption for High resolution FEM method.

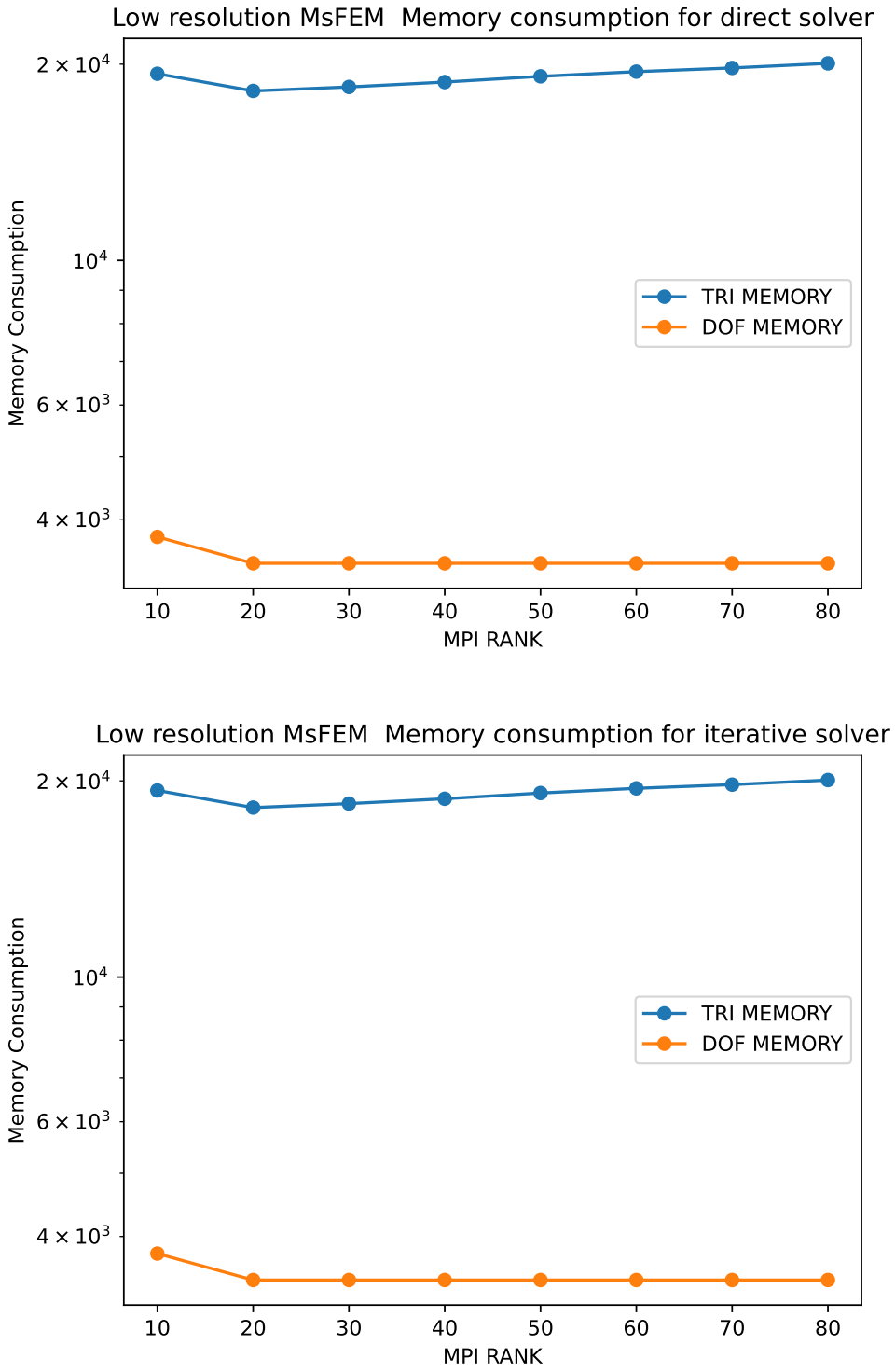


Figure 6.36: Test Case 4 Memory consumption for Low resolution MsFEM method.

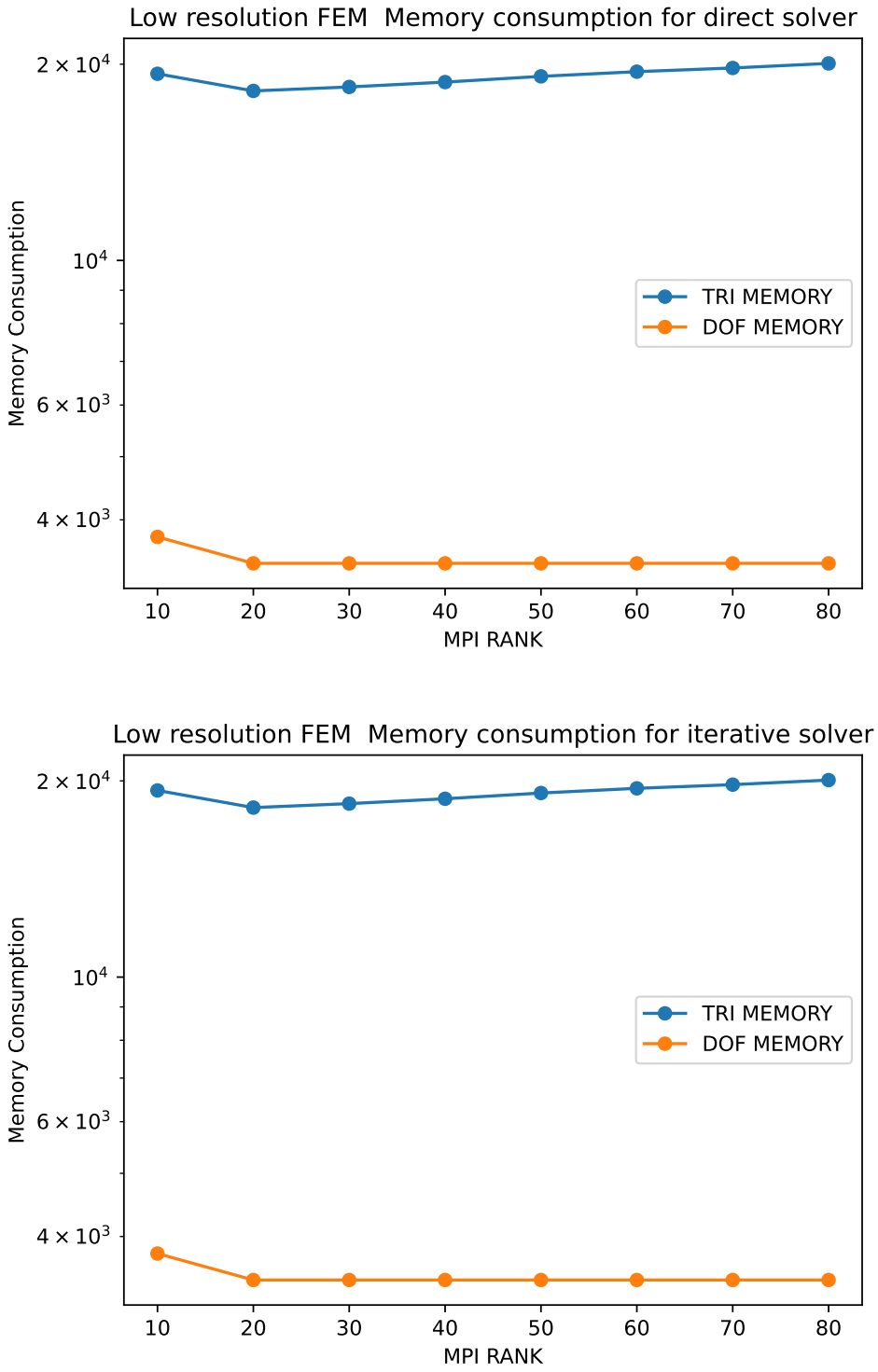


Figure 6.37: Test Case 4 Memory consumption for Low resolution FEM method.

Figure [(6.35), (6.36) and (6.37)] shows that memory consumption decreases with an increase in MPI rank in high-resolution FEM. There is a slow increase in memory consumption, which then becomes constant for low-resolution FEM and MsFEM, but the memory consumption is not as large as that of high-resolution FEM because of the size of the problem.

6.1.5 Test case 3D

Here the domain is a cube. The advection-diffusion problem is solved. Dirichlet boundaries are present at every boundary and periodic boundary conditions are present on the left-right and top-bottom sides.

$$\partial_t u + c_\delta \cdot \nabla u = \nabla \cdot (a_\varepsilon \nabla u) + f$$

Dirichlet condition is

$$u = (x-0.5)^2 + (y-0.5)^2$$

Initial condition

$$u_0(x) = (x-0.5) + (y-0.5)^2$$

Velocity

$$c_\delta(x,y,z) = \begin{bmatrix} 10 \\ 10 \\ 10 \end{bmatrix}$$

Diffusion coefficient

$$a_\varepsilon(x,t) = 0.1 * \left(I_2 - 0.9999 \begin{bmatrix} sin(240\pi x_1) & 0 \\ 0 & sin(240\pi x_2) \end{bmatrix} \right)$$

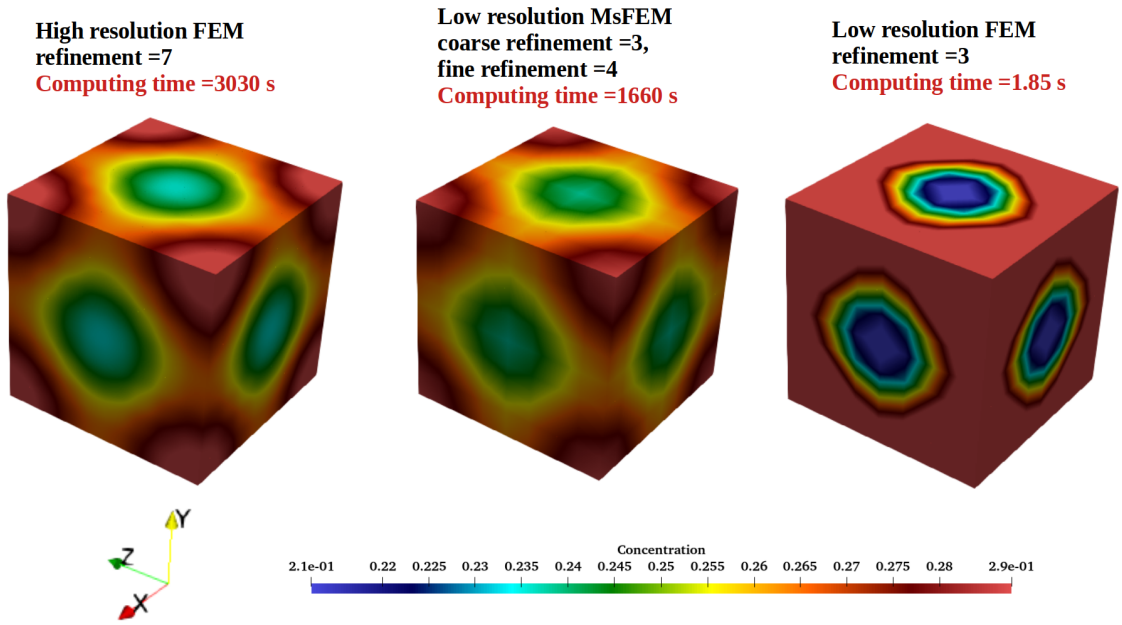


Figure 6.38: Solution for advection-diffusion Equation 3 dimensional.

In the given cube domain with high advection, it is found in Figure (6.38) that low resolution MsFEM shows better results when compared to high- resolution FEM compared to low resolution FEM.

Test case 3D: Error Table

Here the error is calculated with the difference in the reference solution which is high resolution FEM.

Error Analysis			
Simulation Type	L^2 Error	H^1 Error	L^∞ Error
Low resolution FEM	0.0042	0.1785	0.0984
Low resolution MsFEM	0.0001	0.0084	0.0093

Table 6.5: Test 3D Error in simulation with Low resolution FEM, High resolution FEM and Low resolution MsFEM.

According to Table (6.5), low resolution FEM has more error compared to low resolution MsFEM.

Test case 3D : Wall time distribution

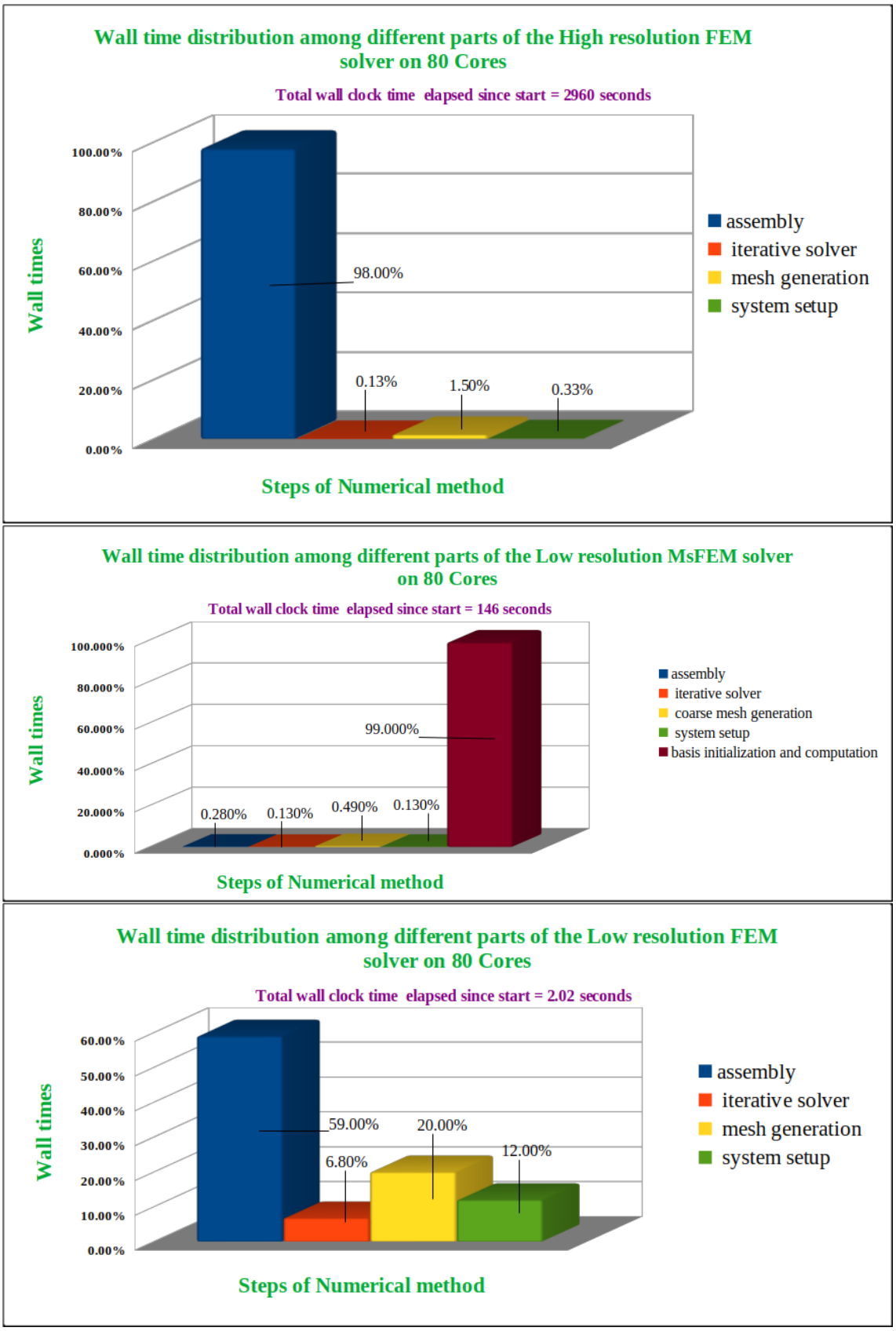


Figure 6.39: Test case 12 : Wall time distribution for solution of High resolution FEM, Low resolution FEM and Low resolution MsFEM for advection-diffusion equation.

It is found that the basis initialization and computation takes the maximum time for low resolution MsFEM, and assembly takes the most computation for high resolution FEM and for low resolution FEM. The total wall clock time elapsed since start is 2960 seconds for high resolution FEM, 146 seconds for low resolution MsFEM and 2.02 seconds for FEM. We see that MsFEM method gives good result with low resolution. As Chapter 6 draws to a close, you can see how the seeds sowed in Chapter 5 have blossomed.

7 Canopy Parameterization

"Marie Curie is my hero. Few people have accomplished something so rare - changing science. And as hard as that is, she had to do it against the tide of the culture at the time - the prejudice against her as a foreigner, because she was born in Poland and worked in France. And the prejudice against her as a woman."

- Alan Alda

7.1 Introduction

Consider the following advection-diffusion equation for two and three dimensions as shown in Figure (7.1):

$$\partial_t u + c_\delta \cdot \nabla u = \nabla \cdot (a_\varepsilon \nabla u) + f \quad \text{in } \mathbb{T}^d \times [0,T] \sim [0,1]^d \times [0,T]$$

Timedependent | Advection | Diffusion | Source

$$u(x,0) = u_0(x)$$

Figure 7.1: The advection-diffusion Equation.

where c_δ is the wind velocity. a_ε is a diffusion coefficient, f and u_0 are smooth external forcing and initial conditions.

As shown in Figure (7.2) we parameterize the building with a maximum diffusion coefficient inside the structure and an outside linear function. The velocity is set to zero inside the building, grows exponentially in a small transition layer near the building to a preset profile in the horizontal direction, and grows logarithmically above the building in the vertical direction, as expected for a boundary layer profile.

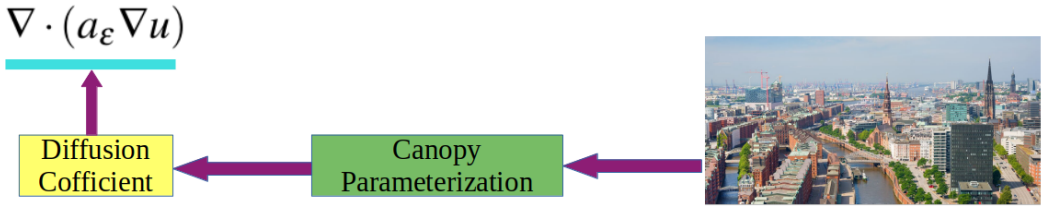


Figure 7.2: Canopy Parameterization.

This leads to canopy parameterization added to previous algorithms as follows: Algorithm (5) canopy parameterization is part of assemble system whereas in Algorithm (6) is part of local assemble system which is then added to global assembly.

Algorithm 5 Algorithm for Canopy parameterization in Time dependent Finite Element Method.

1. Setup mesh
2. Setup system and constraints $for(n = 0 \quad to \quad n = N_{steps})$
3. $while(time \leq T_{max})$
   - Assemble system

     Canopy parameterization
   - Solve for u^{n+1}
   - Set n = n+1

   end while loop

Visualize the output of solution.

Algorithm 6 Algorithm for Canopy parameterization in Time dependent Multiscale Finite Element Method.

1. Setup mesh (coarse).
2. Setup system and constraints.
3. Compute Ms-basis at t = 0 $for(n = 0 \quad to \quad n = N_{steps})$
4. $while(time \leq T_{max})$
5. compute basis at t^{n+1}
   - Assemble system

     Canopy parameterization as part of local system
   - Solve for u^{n+1}
   - Set n = n+1

   end while loop

Visualize the output of solution.

Implementation of canopy parameterization

Let us see how canopies are implemented in the code.

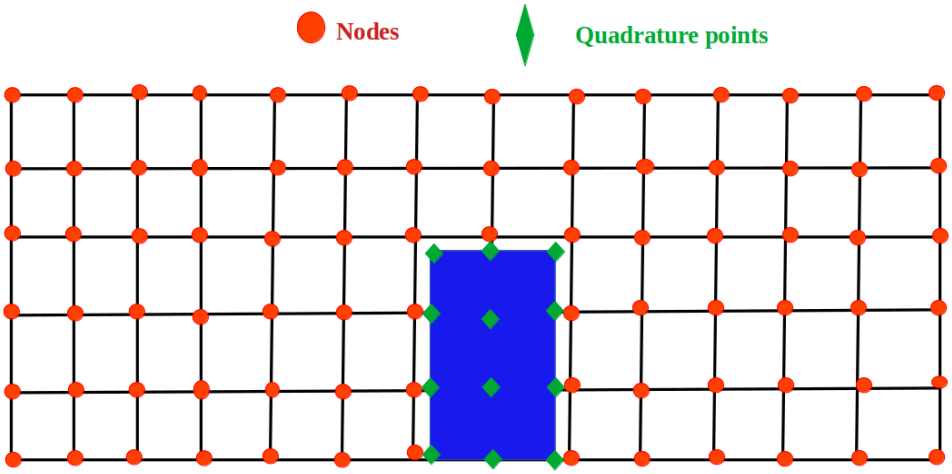


Figure 7.3: Implementation of building.

As shown in Figure (7.3) we have a mesh with nodes in red circle. The green diamonds are the quadrature point used to represent the blue rectangle that is a single building. Numerical evaluation of integrals is based on the quadrature point. The canopy here is a single building which is implemented as diffusion coefficient, which is the maximum value, which is implemented as QGauss quadrature rule in deal.II as shown below in Listing (53). The whole process takes place at the assembly of the code.

```
1  template <int dim>
2  void AdvectionDiffusionProblem<dim>::assemble_system(double current_time)
3    {
4      TimerOutput::Scope t(computing_timer, "assembly");
5
6      const QGauss<dim> quadrature_formula(fe.degree + 1);
7      const QGauss<dim - 1> face_quadrature_formula(fe.degree + 1);
8
9      FEValues<dim> fe_values(fe,quadrature_formula,update_values | update_gradients |
10     update_quadrature_points | update_JxW_values);
11     FEFaceValues<dim> fe_face_values(fe,face_quadrature_formula,
12     update_values | update_quadrature_points | update_normal_vectors |update_JxW_values);
13      // for Neumaan boundary condition to evaluate boundary condition
14
15     const unsigned int dofs_per_cell = fe.dofs_per_cell;
16     const unsigned int n_q_points = quadrature_formula.size();
17     const unsigned int n_face_q_points = face_quadrature_formula.size();
18     FullMatrix<double> cell_matrix(dofs_per_cell, dofs_per_cell);
19     Vector<double> cell_rhs(dofs_per_cell);
20
21     std::vector<types::global_dof_index> local_dof_indices(dofs_per_cell);
```

```
22
23          // Canopy parameterization
24      std::vector<double> matrix_coeff_values_old(n_q_points);
25      std::vector<double> matrix_coeff_values(n_q_points);
26
27      std::vector<Tensor<1, dim>> advection_field_values_old(n_q_points);
28      std::vector<Tensor<1, dim>> advection_field_values(n_q_points);
29
30      std::vector<double> rhs_values_old(n_q_points);
31      std::vector<double> rhs_values(n_q_points);
32
33      std::vector<double> neumann_values_old(n_face_q_points);
34      std::vector<double> neumann_values(n_face_q_points);
35
36    for (const auto &cell : dof_handler.active_cell_iterators())
37    {
38      if (cell->is_locally_owned())
39      {
40        cell_matrix = 0; cell_rhs = 0;
41
42        fe_values.reinit(cell); cell->get_dof_indices(local_dof_indices);
43        /* * Values at current time. */
44        matrix_coeff.set_time(current_time);
45        advection_field.set_time(current_time);
46        right_hand_side.set_time(current_time);
47        advection_field.value_list(fe_values.get_quadrature_points(), advection_field_values);
48        matrix_coeff.value_list(fe_values.get_quadrature_points(), matrix_coeff_values);
49        right_hand_side.value_list(fe_values.get_quadrature_points(), rhs_values);
50        /*
51        * Values at previous time.
52          */
53        matrix_coeff.set_time(current_time - time_step); advection_field.set_time(current_time - time_step);
54        right_hand_side.set_time(current_time - time_step);
55        advection_field.value_list(fe_values.get_quadrature_points(), advection_field_values_old);
56        matrix_coeff.value_list(fe_values.get_quadrature_points(), matrix_coeff_values_old);
57        right_hand_side.value_list(fe_values.get_quadrature_points(), rhs_values_old);
58        /*
59          * Integration over cell.
60            */
61        for (unsigned int q_index = 0; q_index < n_q_points; ++q_index)
62        {
63          for (unsigned int i = 0; i < dofs_per_cell; ++i)
64          {
65            for (unsigned int j = 0; j < dofs_per_cell; ++j)
66            {
67              // Diffusion is on rhs. Careful with signs here.
68              cell_matrix(i, j) +=(fe_values.shape_value(i, q_index) *
69              fe_values.shape_value(j, q_index) +time_step * (theta) *
70              (fe_values.shape_grad(i, q_index) *matrix_coeff_values[q_index] *
71              fe_values.shape_grad(j, q_index) +fe_values.shape_value(i, q_index) *
72              advection_field_values[q_index] *fe_values.shape_grad(j, q_index))) *
73              fe_values.JxW(q_index);
74              // Careful with signs also here.
75              cell_rhs(i) += (fe_values.shape_value(i, q_index) *fe_values.shape_value(j, q_index) -
76              time_step * (1 - theta) *(fe_values.shape_grad(i, q_index) *
77              matrix_coeff_values_old[q_index] *fe_values.shape_grad(j, q_index) +
78              fe_values.shape_value(i, q_index) *advection_field_values_old[q_index] *
```

```
79                fe_values.shape_grad(j, q_index))) *fe_values.JxW(q_index) *
80                old_solution(local_dof_indices[j]);
81              } // end ++j
82              cell_rhs(i) += time_step * fe_values.shape_value(i, q_index) *
83              ((1 - theta) * rhs_values_old[q_index] +(theta)*rhs_values[q_index]) *
84              fe_values.JxW(q_index);
85            } // end ++i
86          } // end ++q_index
87
88          if (!is_periodic)
89          {
90        /* Boundary integral for Neumann values for odd boundary_id in non-periodic case.*/
91
92            for (unsigned int face_number = 0;face_number < GeometryInfo<dim>::faces_per_cell; ++face_number)
93            {
94              if (cell->face(face_number)->at_boundary() && ((cell->face(face_number)->boundary_id() == 1) ||
95              (cell->face(face_number)->boundary_id() == 3) ||
96              (cell->face(face_number)->boundary_id() == 5)))
97              {
98                fe_face_values.reinit(cell, face_number);
99          /* Fill in values at this particular face at current time.*/
100               neumann_bc.set_time(current_time);neumann_bc.value_list(fe_face_values.get_quadrature_points(), neumann_values);
101         /* Fill in values at this particular face at previous time.*/
102               neumann_bc.set_time(current_time - time_step);neumann_bc.value_list(
103               fe_face_values.get_quadrature_points(),neumann_values_old);
104
105         for (unsigned int q_face_point = 0;q_face_point < n_face_q_points;++q_face_point)
106         {
107           for (unsigned int i = 0; i < dofs_per_cell; ++i)
108             {
109              cell_rhs(i) +=time_step *((1 - theta) *
110              neumann_values_old[q_face_point] * // g(x_q, t_n) = A*grad_u at t_n
111                          +(theta)*neumann_values[q_face_point]) * // g(x_q, t_{n+1}) = = // A*grad_u at t_{n+1}
112                        fe_face_values.shape_value(i, q_face_point) *//phi_i(x_q)
113                        fe_face_values.JxW(q_face_point); // dS
114                  } // end ++i
115              } // end ++q_face_point
116            } // end if
117          } // end ++face_number
118        }
119        constraints.distribute_local_to_global( cell_matrix,cell_rhs,local_dof_indices,
120        system_matrix,system_rhs, /* use_inhomogeneities_for_rhs */ true);
121      } // if
122    } // ++cell
123  system_matrix.compress(VectorOperation::add);
124  system_rhs.compress(VectorOperation::add);
125  }
```

Listing 53: Canopy parametrization in deal.II.

For MsFEM method, it is implemented in **basis** code and **reconstruction basis** code. The following test cases are implemented for the high resolution standard finite element method with refinement = 8.

7.2 Test Case 5

Domain is $-50 \leq x \leq 150$ and $0 \leq y \leq 50$. In Figure (7.4) each of the boundary conditions is color coded. We see the details of the values and their implementation in the code as follows.

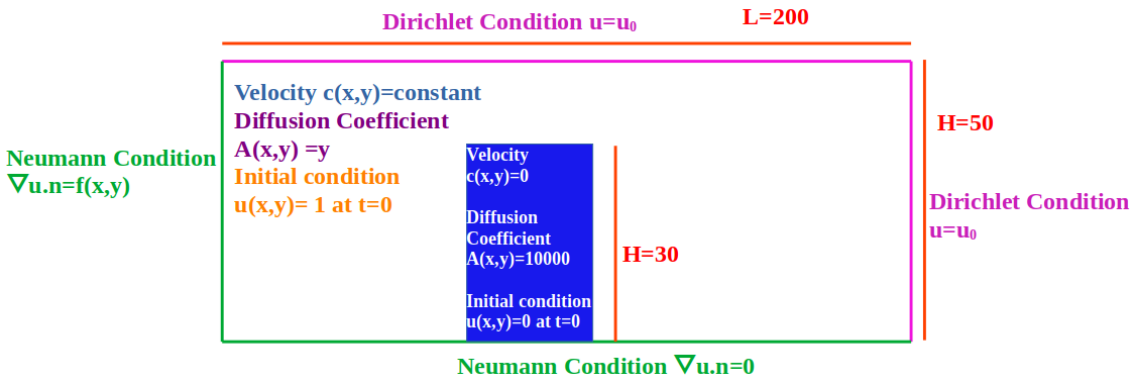


Figure 7.4: Sketch of rectangle building with conditions.

1. Dirichlet condition

$$u_0 = \begin{cases} 0 & 40 \leq x \leq 60, \quad and \quad y \leq 30 \quad \textbf{Building} \\ 1 & else, \end{cases}$$

```
template <int dim>
double DirichletBC<dim>::value(const Point<dim>& p, const unsigned int /*component*/) const
{
  double return_value=0;
  if (p[0]>=40 && p[0]<=60 && p[1]<=30)
  return_value =0;
  else
  return_value =1;
  return return_value;
}
```

Listing 54: Dirichlet Condition for Test Case 5.

2. Initial condition at time t = 0

$$u(x,y) = \begin{cases} 0 & 40 \leq x \leq 60, \quad and \quad y \leq 30 \quad \textbf{Building} \\ 1 & else, \end{cases}$$

```
template <int dim>
double InitialValue<dim>::value(const Point<dim>& p, const unsigned int /*component*/)const
```

```
{
  double return_value=0;
  if (p[0]>=40 && p[0]<=60 &&p[1]<=30)
  return_value =0;
  else
  return_value =1;
  return return_value;
}
```

Listing 55: Initial Condition for Test Case 5.

3. Neumann condition

$$g = A * exp(-B * (x-25)^2)$$

where $A = B = 1$

```
template <int dim>
double NeumannBC<dim>::value(const Point<dim> &p,
const unsigned int /*component*/) const
{
  double A = 1; double B = 1;const double x_minus_mui = p[1] - 25;
  double sum= A*std::exp(-B * (x_minus_mui)* (x_minus_mui));
  return sum;
}
```

Listing 56: Neumann Condition for Test Case 5.

4. Velocity

$$c(x,y) = \begin{cases} \begin{pmatrix} 0 \\ 0 \end{pmatrix} & -50 \le x \le 150, \quad y = 0 \quad \textbf{Bottom Boundary} \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} & 40 \le x \le 60, \quad and \quad y \le 30 \quad \textbf{Building} \\ \begin{pmatrix} c_h * exp(-(1-(y/30))) \\ 0 \end{pmatrix} & y \le 30 \quad \textbf{Around the Building} \text{ where } u_h \text{ is reference velocity} \\ \begin{pmatrix} (c*/\kappa) \cdot log((y+y_0)/y_0) \\ 0 \end{pmatrix} & \text{else} \quad \textbf{Above the Building} \text{ where } c* \text{ is friction velocity, } \kappa \text{ is von Kármán constant and } z_0 \text{ is roughness length} \end{cases}$$

where $c_h = 4$ is the wind velocity at $h = 30$, $y_0 = 0.5$ is the surface roughness , k is

the von Kármán constant $(k = 0.4)$,and $c^* = 0.35$ is the friction velocity.

```
template <int dim>
Tensor<1, dim> AdvectionField<dim>::value(const Point<dim> &p) const
{
  Tensor<1, dim> value;
  value.clear();

  double c_h = 4;
  double c_star = 0.35;
  double k = 0.4;
  double z0=0.5;
  if (p[0]>=-50 && p[0]<=150 && p[1]==0)
  {
    value[0]=0;
    value[1] =0;
  }
  else if (p[0]>=40 && p[0]<=60 &&p[1]<=30)
  {
    value[0]=0;
    value[1] =0;
  }
  else if ( p[1]<=30)
  {
    value[0]=c_h*exp(-(1-(p[1]/h)));
    value[1] =0;
  }
  else
   {
    value[0]=(c_star/k)*log((p[1]+z0)/z0);
    value[1] =0;
  }
  return value;
}
```

Listing 57: Velocity for Test Case 5.

5. Diffusion coefficient

$$a(x,y) = \begin{cases} 10000 & \textbf{when } 40 \le x \le 60, \quad \textit{and} \quad y \le 30 \quad \textbf{Building} \\ y & \textbf{else} \end{cases}$$

```
template <int dim>
Tensor<2, dim> MatrixCoeff<dim>::value(const Point<dim> &p) const
{
  Tensor<2, dim> value;
  value.clear();
  for (unsigned int d = 0; d < dim; ++d)
  {
    if (p[0]>=40 && p[0]<=60 &&p[1]<=30)
    {
```

```
        value[d][d] =10000;
      }
      else
       {
        value[d][d] =p[1];
      }
   }
   return value;
}
```

Listing 58: Diffusion coefficient for Test Case 5.

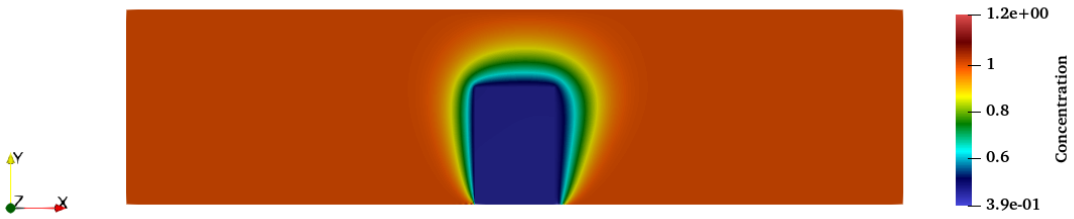


Figure 7.5: Rectangle building.

Here, the domain is 200 m long and 50 m tall. Here, we use quadrature points to implement different initial and boundary conditions. The initial condition is a non-zero value outside the building and zero inside the building. The velocity is zero inside the building, exponential near the building, and logarithmic above the building. The diffusion inside the building is very high and linear function outside the facility is very high. We can see the rectangle building in Figure (7.5) due to high diffusion, and trace transport around it.

7.3 Test Case 6

Domain is $-50 \leq x \leq 150$ and $0 \leq y \leq 50$. Now we consider boundary conditions for triangle building.

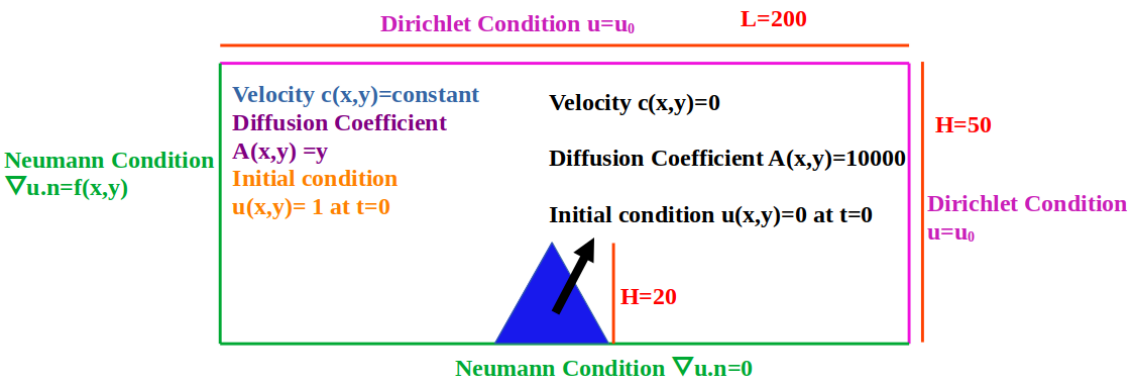


Figure 7.6: Sketch of triangle building with conditions.

1. Dirichlet condition

$$u_0 = \begin{cases} 0 & 40 \leq x \leq 60, \quad y \leq x-40 \quad and \quad y \leq 60-x \quad \textbf{Building} \\ \\ 1 & else, \end{cases}$$

2. Initial condition at time t=0

$$u(x,y) = \begin{cases} 0 & 40 \leq x \leq 60, \quad y \leq x-40 \quad and \quad y \leq 60-x \quad \textbf{Building} \\ \\ 1 & else, \end{cases}$$

3. Neumann condition

$$g = A * exp(-B * (x-25)^2)$$

where $A = B = 1$

4. Velocity

$$c(x,y) = \begin{cases} \begin{pmatrix} 0 \\ 0 \end{pmatrix} & -50 \leq x \leq 150, \quad y = 0 \quad \textbf{Bottom Boundary} \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \textbf{when } 40 \leq x \leq 60, \quad y \leq x - 40 \\ & \textit{and} \quad y \leq 60 - x \quad \textbf{Building} \\ \begin{pmatrix} c_h \cdot exp(-(1-(y/20))) \\ 0 \end{pmatrix} & y \leq 20 \quad \textbf{Around the Building} \text{ where } c_h \text{ is reference velocity} \\ \begin{pmatrix} (c*/\kappa) \cdot log((y+y_0)/y_0) \\ 0 \end{pmatrix} & \text{else} \quad \textbf{Above the Building} \text{ where } c* \text{ is friction velocity, } \kappa \text{ is von Kármán constant and } z_0 \text{ is roughness length} \end{cases}$$

where $c_h = 4$ is the wind velocity at $h = 20$, $y_0 = 0.5$ is the surface roughness , k is the von Kármán constant $(k = 0.4)$,and $c^* = 0.35$ is the friction velocity.

5. Diffusion coefficient

$$a(x,y) = \begin{cases} 10000 & \textbf{when } 40 \leq x \leq 60, \quad y \leq x - 40 \quad \textit{and} \quad y \leq 60 - x \quad \textbf{Building} \\ y & \textbf{else} \end{cases}$$

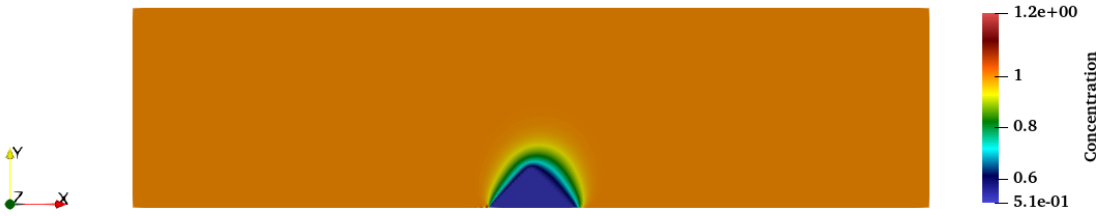


Figure 7.7: Triangular building.

In Test Case 6, a domain size and concept similar to Test Case 5 is implemented for the initial and boundary condition to get a triangle shape building. Tracer transport through the canopy is shown in Figure (7.7).

7.4 Test Case 7

Domain is $-50 \leq x \leq 150$ and $0 \leq y \leq 50$

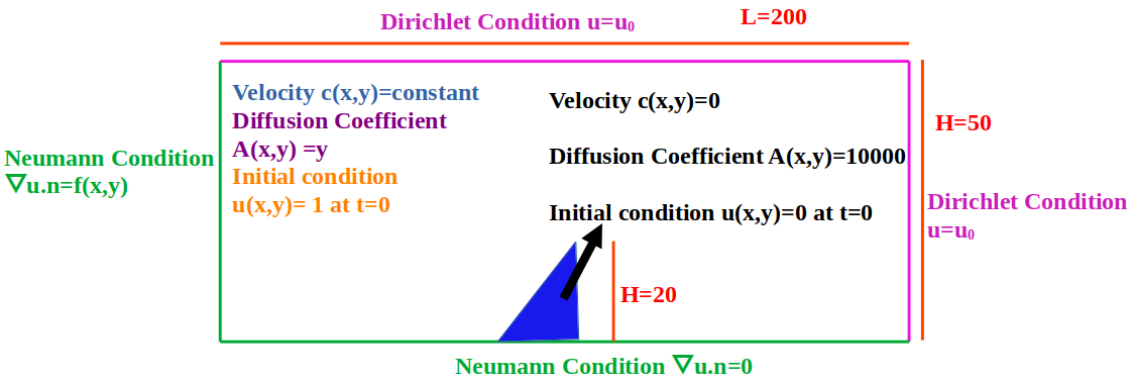


Figure 7.8: Sketch of Hip Triangular building with conditions.

1. Dirichlet condition

$$u_0 = \begin{cases} 0 & 40 \leq x \leq 60, \quad y \leq 60 \quad and \quad y \leq x - 40 \quad \textbf{Building} \\ 1 & else, \end{cases}$$

2. Initial condition at time t=0

$$u(x,y) = \begin{cases} 0 & 40 \leq x \leq 60, \quad y \leq 60 \quad and \quad y \leq x - 40 \quad \textbf{Building} \\ 1 & else, \end{cases}$$

3. Neumann condition

$$g = A * exp(-B * (x-25)^2)$$

where $A = B = 1$

4. Velocity

$$c(x,y) = \begin{cases} \begin{pmatrix} 0 \\ 0 \end{pmatrix} & -50 \le x \le 150, \quad y = 0 \quad \textbf{Bottom Boundary} \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} & 40 \le x \le 60, \quad y \le 60 \quad \textit{and} \quad y \le x - 40 \quad \textbf{Building} \\ \begin{pmatrix} c_h * exp(-(1-(y/20))) \\ 0 \end{pmatrix} & y \le 20 \quad \textbf{Around the Building} \text{ where } c_h \text{ is reference velocity} \\ \begin{pmatrix} (c*/\kappa) \cdot log((y+y_0)/y_0) \\ 0 \end{pmatrix} & \text{else} \quad \textbf{Above the Building} \text{ where } c* \text{ is friction velocity, } \kappa \text{ is von Kármán constant and } z_0 \text{ is roughness length} \end{cases}$$

where $c_h = 4$ is the wind velocity at $h = 20$,$y_0 = 0.5$ is the surface roughness , k is the von Kármán constant $(k = 0.4)$,and $c_* = 0.35$ is the friction velocity.

5. Diffusion Coefficient

$$a(x,y) = \begin{cases} 10000 & 40 \le x \le 60, \quad y \le 60 \quad \textit{and} \quad y \le x - 40 \quad \textbf{Building} \\ y & \textbf{else} \end{cases}$$

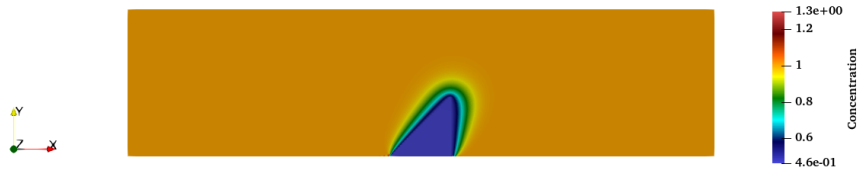


Figure 7.9: Hip Triangular building.

As in previous test cases, a similar domain size and concept is implemented for the initial and boundary conditions to obtain a triangle shape building. In Figure (7.9) tracer transports through the canopy.

7.5 Test Case 8

Domain is $-50 \leq x \leq 150$ and $0 \leq y \leq 50$

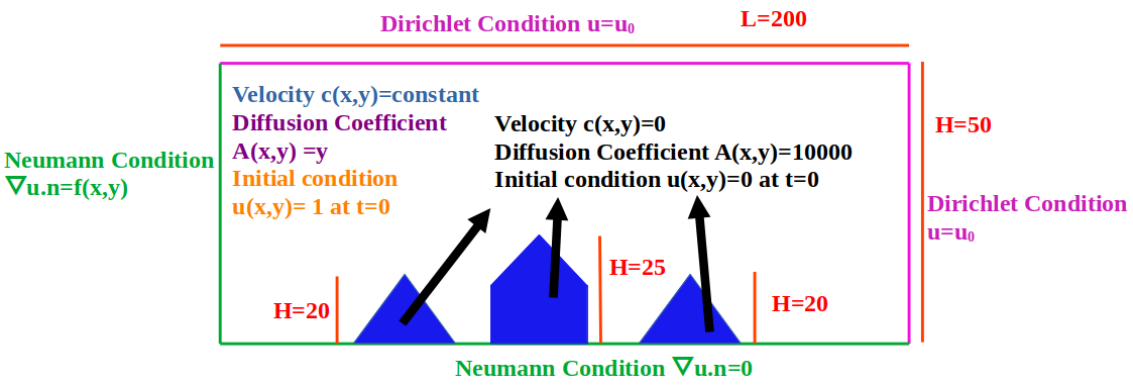


Figure 7.10: Sketch of three building with conditions.

1. Dirichlet condition

$$u_0 = \begin{cases} 0 \quad 10 \le x \le 30, \quad y \le x-10 \quad and \quad y \le 30-x & \textbf{Left Building} \\ 0 \quad 40 \le x \le 60, \quad y \le x-30 \quad and \quad y \le 70-x & \textbf{Middle Building} \\ 0 \quad 70 \le x \le 90, \quad y \le x-70 \quad and \quad y \le 90-x & \textbf{Last Building} \\ 1 \quad else, \end{cases}$$

2. Initial condition at time t=0

$$u(x,y) = \begin{cases} 0 \quad 10 \le x \le 30, \quad y \le x-10 \quad and \quad y \le 30-x & \textbf{Left Building} \\ 0 \quad 40 \le x \le 60, \quad y \le x-30 \quad and \quad y \le 70-x & \textbf{Middle Building} \\ 0 \quad 70 \le x \le 90, \quad y \le x-70 \quad and \quad y \le 90-x & \textbf{Last Building} \\ 1 \quad else, \end{cases}$$

3. Neumann condition

$$g = A * exp(-B * (x-25)^2)$$

where $A = B = 1$

4. Velocity

$$c(x,y) = \begin{cases} \begin{pmatrix} 0 \\ 0 \end{pmatrix} & -50 \le x \le 150, \quad y = 0 \quad \textbf{Bottom Boundary} \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} & 10 \le x \le 30, \quad y \le x - 10 \quad \textit{and} \quad y \le 30 - x \quad \textbf{Left Building} \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} & 40 \le x \le 60, \quad y \le x - 30 \quad \textit{and} \quad y \le 70 - x \quad \textbf{Middle Building} \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} & 70 \le x \le 90, \quad y \le x - 70 \quad \textit{and} \quad y \le 90 - x \quad \textbf{Last Building} \\ \begin{pmatrix} c_h * exp(-(1-(y/20))) \\ 0 \end{pmatrix} & y \le 20 \quad \textbf{Around the Building} \text{ where } c_h \text{ is reference velocity} \\ \begin{pmatrix} (c*/\kappa) \cdot log((y+y_0)/y_0) \\ 0 \end{pmatrix} & \text{else} \quad \textbf{Above the Building} \text{ where } c* \text{ is friction velocity, } \kappa \text{ is von Kármán constant and } z_0 \text{ is roughness length} \end{cases}$$

where $c_h = 4$ is the wind velocity at $h = 20$, $y_0 = 0.5$ is the surface roughness , k is the von Kármán constant $(k = 0.4)$,and $c_* = 0.35$ is the friction velocity.

5. Diffusion Coefficient

$$a(x,y) = \begin{cases} 10000 & 10 \le x \le 30, \quad y \le x - 10 \quad \textit{and} \quad y \le 30 - x \quad \textbf{Left Building} \\ 10000 & 40 \le x \le 60, \quad y \le x - 30 \quad \textit{and} \quad y \le 70 - x \quad \textbf{Middle Building} \\ 10000 & 70 \le x \le 90, \quad y \le x - 70 \quad \textit{and} \quad y \le 90 - x \quad \textbf{Last Building} \\ y & \textbf{else} \end{cases}$$

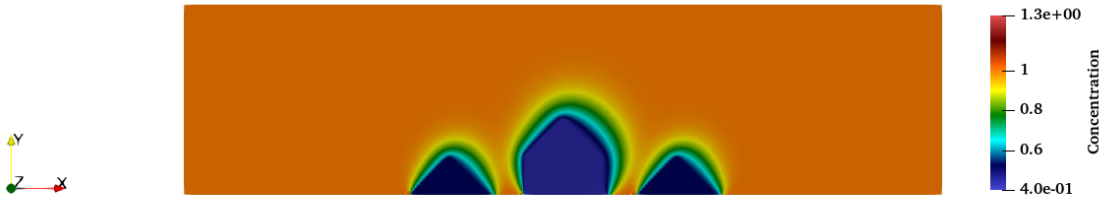


Figure 7.11: Mix Triangular buildings.

Here, we construct a heterogeneous canopies with three triangular canopies and implement the same initial and boundary conditions as in the previous test case. As shown in Figure (7.11) the trance transport occurs near the building. Chapter 7 concludes with complete implementation of all computational bits needed for the real-world application discussed in Chapter 8.

8 Application

"What interests me is the connection between maths and the real world."

- Terence Tao

8.1 Wind tunnel test cases

The "small" boundary layer wind tunnel "Blasius" at the meteorological institute of the University of Hamburg was used for all experiments. Approximately 16 m long, 1.5 m wide, and 1 m high, the wind tunnel's test section measures 4.5 m long, 1.5 m wide, and 1 m high. There is a 7 m boundary layer development section in the tunnel. There is an adjustable ceiling in the tunnel, as shown in Figure (8.1).

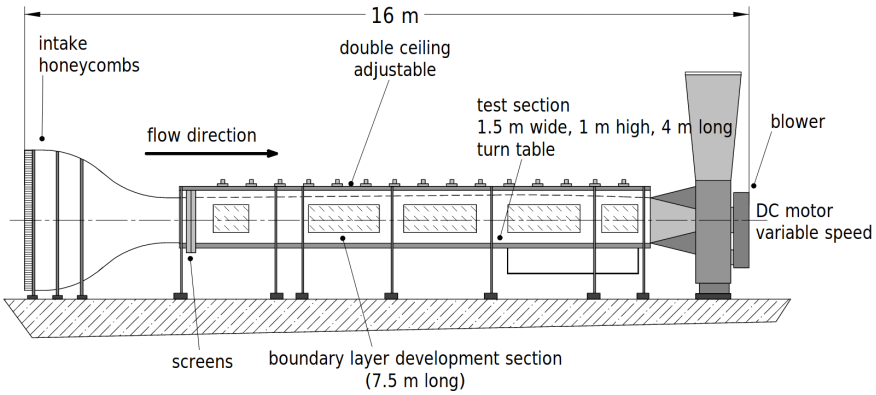


Figure 8.1: Sketch of the boundary layer wind tunnel "Blasius" at the Meteorological Institute of Hamburg University [34].

A wooden cuboid with a height of 30 m at full scale (0,06 at Model scale) was placed inside the tunnel's test section. Ethane (15 litres/hour release) is the tracer gas released in a wind tunnel. For all measurements, velocity and concentration were measured. A cuboid with a size of h = 30 m representing idealized urban roughness was positioned in the wind tunnel test section. The experiments were conducted by **Sylvio Freitas**. There were 2 setups with 0° in Figure (8.2) and 45° in Figure (8.3) rotation of the building for flow and concentration measurements.

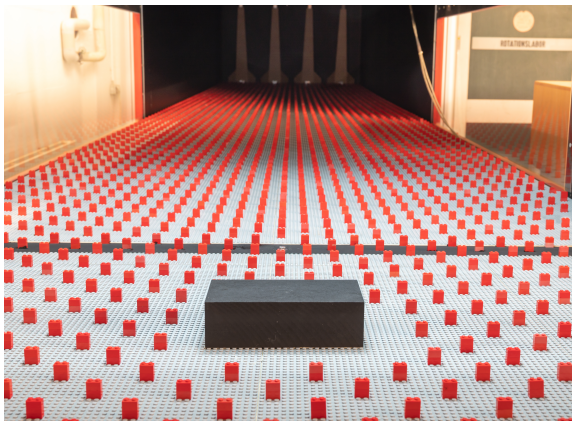


Figure 8.2: Wind tunnel setup with 0° rotation.

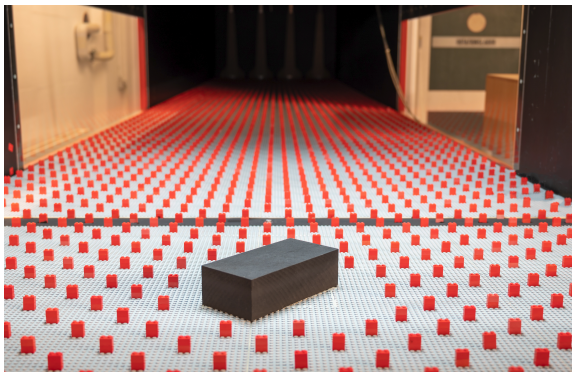


Figure 8.3: Wind tunnel setup with 45° rotation.

8.1.1 Setup

As shown in Figure (8.4) following details for wind tunnel measurement.

1. Main flow direction (wind tunnel) represented by U.
2. 0° (bold outline of building) and 45° (dashed outline) inflows.
3. Dimensions in full-scale.
4. Origin of coordinate system: P02: (X,Y) = (0,0) [m].
5. Flow measurement points (2 x 2D): P01–P09 (7 heights: z = 20, 30, 40, 60, 80, 100 and 120 m)
6. Location of the gas release source: P01: (X,Y) = (-236.5, 0) [m]
7. Concentration measurement points: P01c to P09 (5 heights above Y = 0 m axis: z = 20, 30, 40, 50 and 60 m), (2 heights above P03, P04, P07, P08: z = 20 and 40 m)

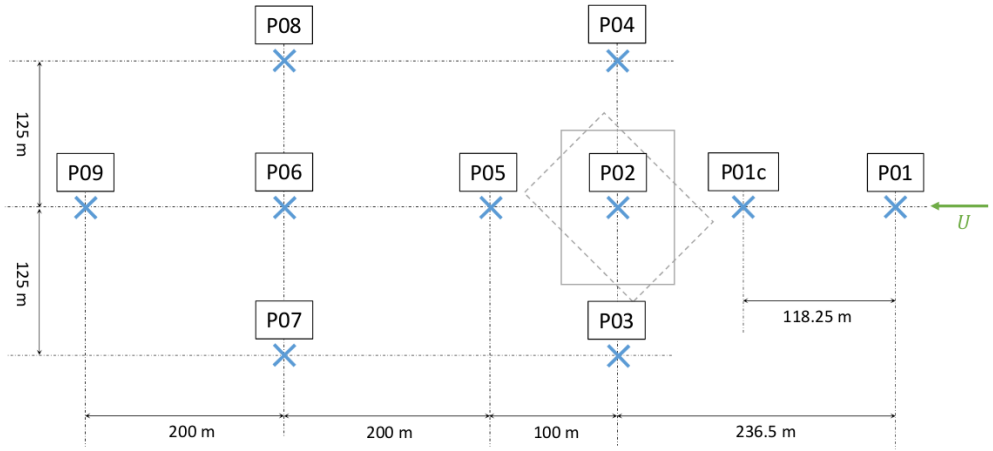


Figure 8.4: Setup for wind tunnel.

8.2 Test Case 9 : Building with 0° rotation

Consider the rectangular computational domain of size 800 × 500 × 120 as shown in Figure (8.5). In deal.II the hyper_rectangle is used to create this domain. A building of size 100 × 50 × 30 is constructed using the canopy parametrization described in the previous section.

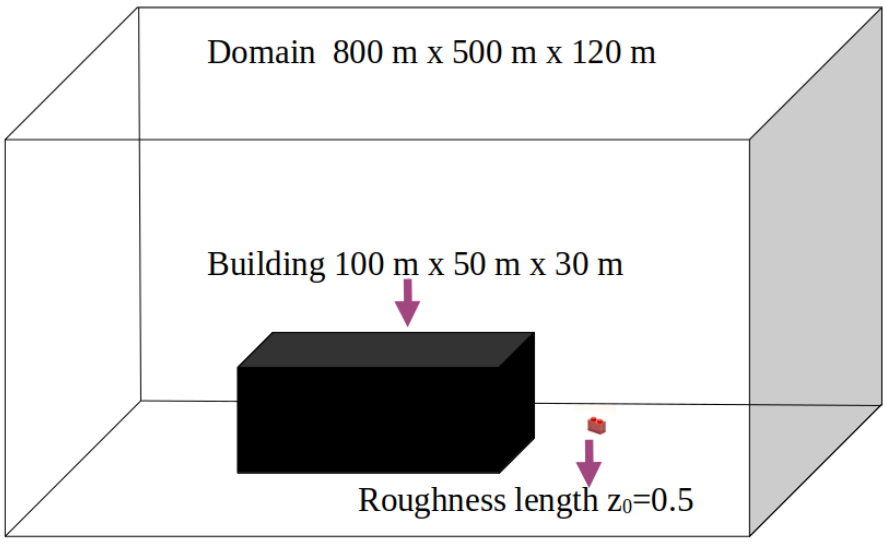


Figure 8.5: Computational Domain.

The Figure (8.6) shows the face numbers of deal.II. Face 0 is on the left-hand side of the domain, which is the inlet, and the face on the right-hand side of the domain is the outlet.

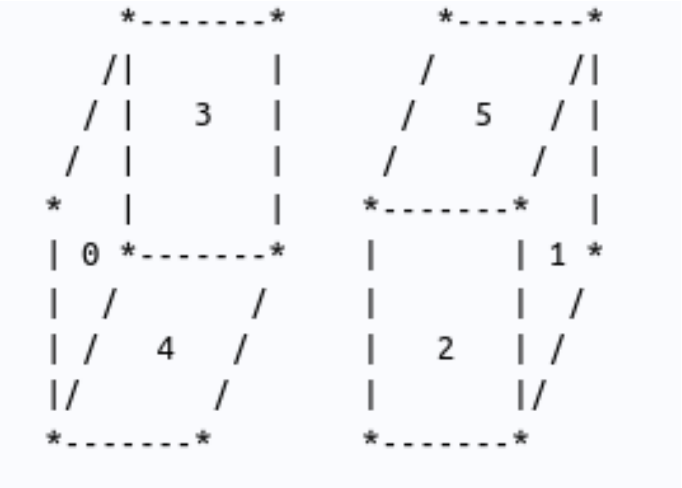


Figure 8.6: Face numbering in deal.II.

The high-resolution FEM test case 9 as shown in Figure (8.7) has 2146689 degree of freedom.

Figure 8.7: Domain Test 9 High resolution Mesh.

The low-resolution FEM and MsFEM test case 9 as shown in Figure (8.8) has 729 degrees of freedom.

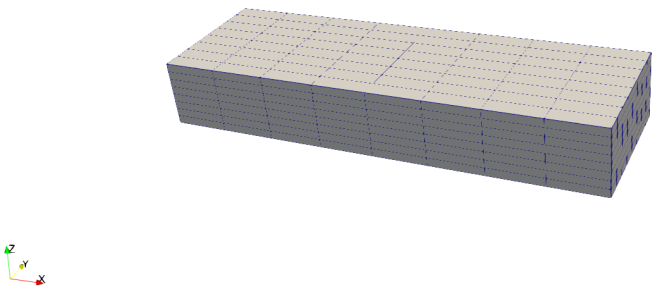


Figure 8.8: Domain Test 9 Low resolution Mesh.

The following are the initial and boundary conditions

Domain is $-300 \leq x \leq 500$, $-125 \leq y \leq 125$ and $0 \leq z \leq 120$

1. Dirichlet Condition on right and top boundaries

$$u_0 = \begin{cases} 0 \quad \textbf{when} \quad -25 \leq x \leq 25, \quad -50 \leq y \leq 50 \quad \textit{and} \quad 0 \leq z \leq 30 \quad \textbf{Building} \\ \\ 0.1 \quad \textbf{else} \end{cases}$$

2. Initial Condition at t = 0

$$u(x,y,z) = \begin{cases} 0 \quad \textbf{when} \quad -25 \leq x \leq 25, \quad -50 \leq y \leq 50 \quad \textit{and} \quad 0 \leq z \leq 30 \quad \textbf{Building} \\ \\ 0.1 \quad \textbf{else} \end{cases}$$

3. Neumann Condition at the Left hand boundary

$$g = exp(-(z-60)^2)$$

4. Diffusion Coefficient

$$a(x,y,z) = \begin{cases} 10000 \quad \textbf{when} \ -25 \leq x \leq 25, \quad -50 \leq y \leq 50 \quad \textit{and} \quad 0 \leq z \leq 30 \quad \textbf{Building} \\ \\ z \qquad\qquad \textbf{else} \end{cases}$$

5. Velocity

$$c(x,y,z) = \begin{cases} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} & \textbf{when } -25 \le x \le 25, -50 \le y \le 50 \\ & \textit{and} \quad 0 \le z \le 30 \ \textbf{Building} \\ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} & \textbf{when } -300 \le x \le 500, \quad \textit{and} \\ & z = 0 \ \textbf{Bottom Building} \\ \begin{pmatrix} c_h \cdot exp(-(1-(z/30))) \\ 0 \\ 0 \end{pmatrix} & \text{else} \quad \textbf{Around the Building} \text{ where } c_h \text{ is reference velocity} \\ \begin{pmatrix} (c*/\kappa) \cdot log((z+z_0)/z_0) \\ 0 \\ 0 \end{pmatrix} & z \ge 30 \quad \textbf{Above the Building} \text{ where } c* \text{ is friction velocity, } \kappa \text{ is von Kármán constant and } z_0 \text{ is roughness length} \end{cases}$$

where $c_h = 4$ is the wind velocity at $h = 30$, $y_0 = 0.5$ is the surface roughness, k is the von Kármán constant $(k = 0.4)$,and $c^* = 0.35$ is the friction velocity.

6. Right hand side

$$f = 0$$

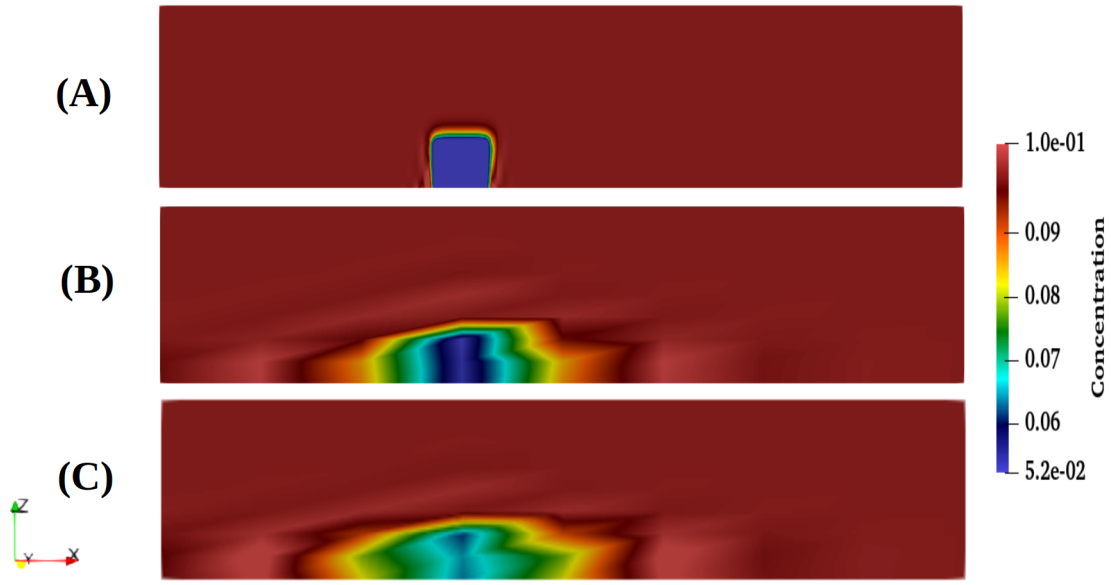


Figure 8.9: Test case 9 Wind tunnel test case single building with 0° degree rotation (A) High resolution FEM with refinement = 7 (B) Low resolution MsFEM with coarse refinement = 3, fine refinement = 4 and (C) Low resolution FEM with refinement = 3.

As shown in Figure (8.9) the cross-section at the y normal of the 3D domain, where (A) is a high resolution FEM that accurately shows the canopy structure, is our reference solution which requires high computation times, and (B) is the low resolution MsFEM that shows some similarity with the high resolution FEM, but the (C) low resolution FEM does not display an accurate representation of the canopy in the domain. Hence we can say with low computation MsFEM can be reasonable choice for representing canopy on large scale.

8.2.1 Test case 9 : Wall times distribution

For all heavy simulations, column-charts would be shown to display results unlike 2D case as the maximum number of nodes where needed to carry out one simulation. It is found in Figure (8.10) that the iterative solver takes the maximum time for low resolution FEM iterative solver and basis initialization and computation takes the maximum time, and for low resolution MsFEM, whereas assembly takes the most computation for high resolution FEM. The total wall clock time elapsed since start is 3220 seconds for high resolution FEM, 130 seconds for low resolution MsFEM and 243 seconds for FEM.

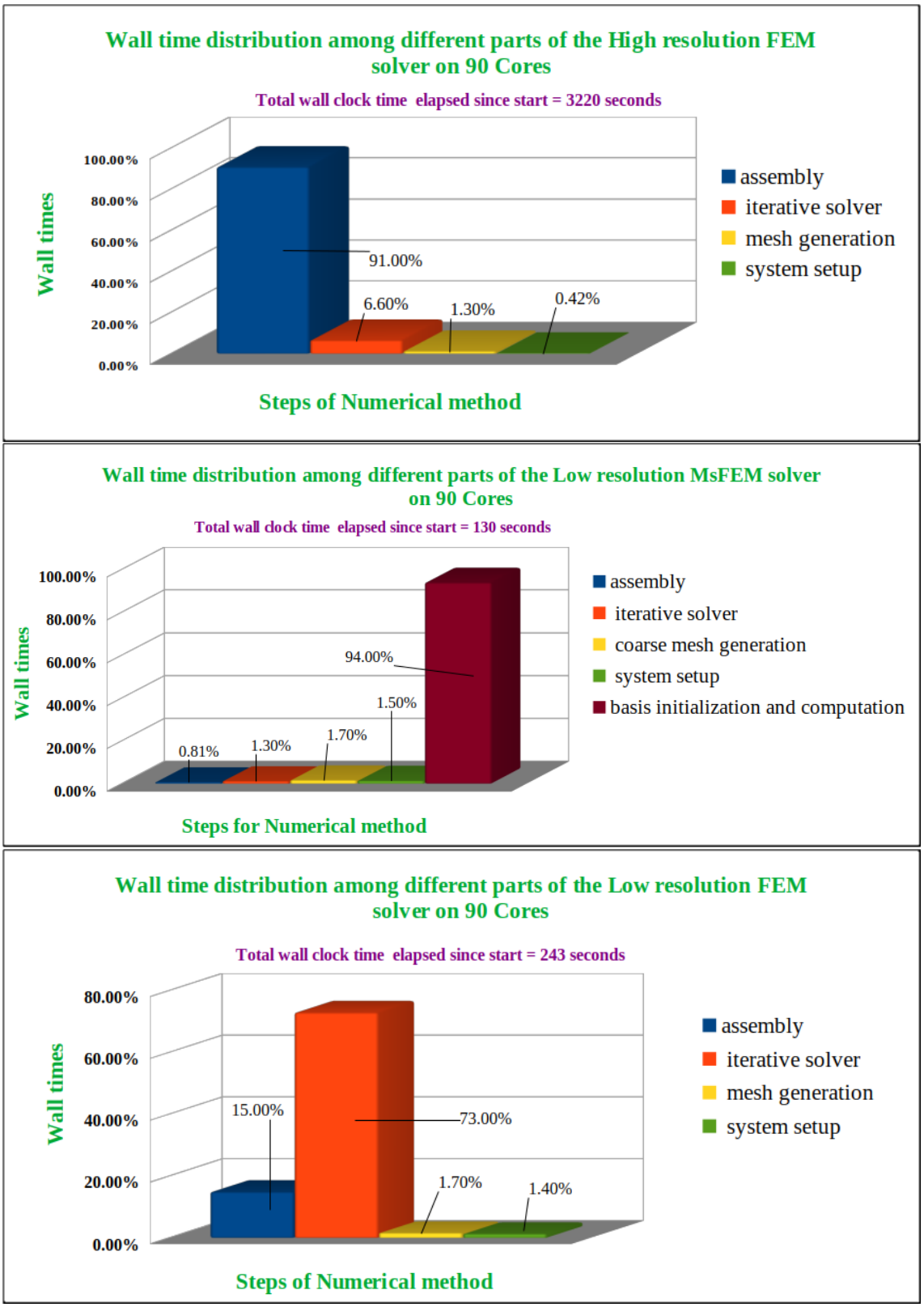


Figure 8.10: Test Case 9 Wall times distribution for High resolution FEM method, Low resolution MsFEM method and Low resolution FEM method.

8.2.2 Test case 9 : Error Table

Here, the error is calculated using the difference in the reference solution, which is a high-resolution FEM.

Error Analysis			
Simulation Type	L^2 Error	H^1 Error	L^∞ Error
Low resolution FEM	2.9	2.9	0.011
Low resolution MsFEM	2.7	2.7	0.0103

Table 8.1: Test case 9 Error in simulation with Low resolution FEM and Low resolution MsFEM.

Table (8.1) shows the error analysis of the low resolution FEM and low resolution MsFEM compared with the high resolution FEM. It is found that low resolution MsFEM is more accurate than low resolution FEM.

8.3 Test Case 10 : Building with 0° rotation and source term

There is no change in the domain set, but a point source has been added before the building.

A high-resolution FEM test case 10 has 2146689 degrees of freedom.

The low resolution FEM and MsFEM test case 10 has 729 degrees of freedom.

Following are initial and Boundary condition is same values as Test 9 except the right hand side

1. Dirichlet Condition on left and bottom boundary
2. Initial condition at t = 0
3. Neumann Condition on right hand side
4. Diffusion Coefficient
5. Velocity
6. Right hand side

$$f = \begin{cases} 1 & \textbf{when } -239.5 \leq x \leq -236.5, \quad y \leq 1 \quad \textit{and} \quad z \leq 1 \quad \textbf{point source} \\ 0 & \textbf{else} \end{cases}$$

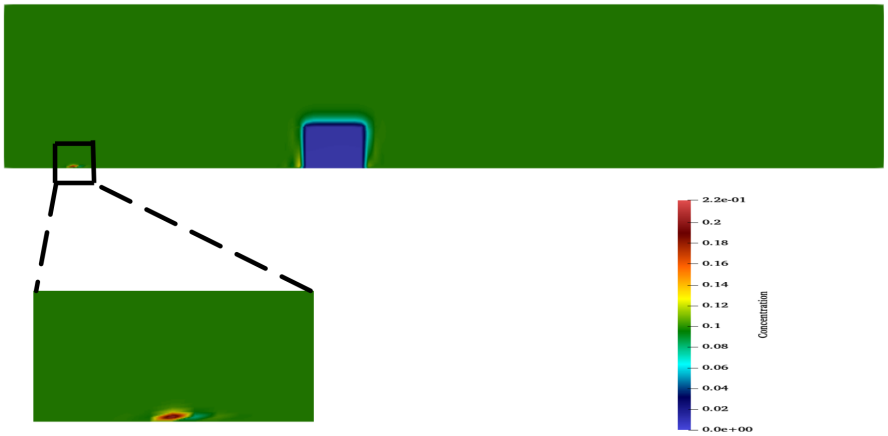


Figure 8.11: High resolution FEM solution with a point source for Test 10.

Figure (8.11) shows high resolution FEM solution with a point source.

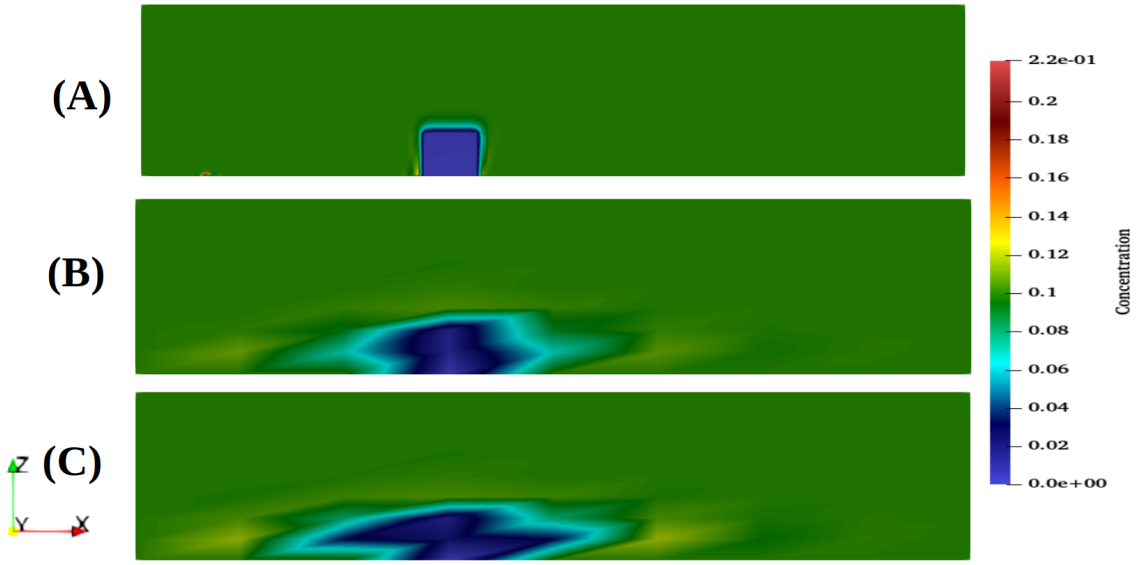


Figure 8.12: Test case 10 Wind tunnel test case single building with 0° degree rotation and point source (A) High resolution FEM with refinement = 7 (B) Low resolution MsFEM with coarse refinement = 3, fine refinement = 4 and (C) Low resolution FEM with refinement = 3.

Figure (8.12) shows a 2D cross-section of a 3D domain with a point source before the canopy at y normal is taken. Here, we observe that in (C) low resolution FEM, the canopy structure is distorted, and in (B), the low-resolution MsFEM canopy structure shows some similarity to the reference solution that is (A) high resolution FEM.

8.3.1 Test case 10 : Wall time distribution

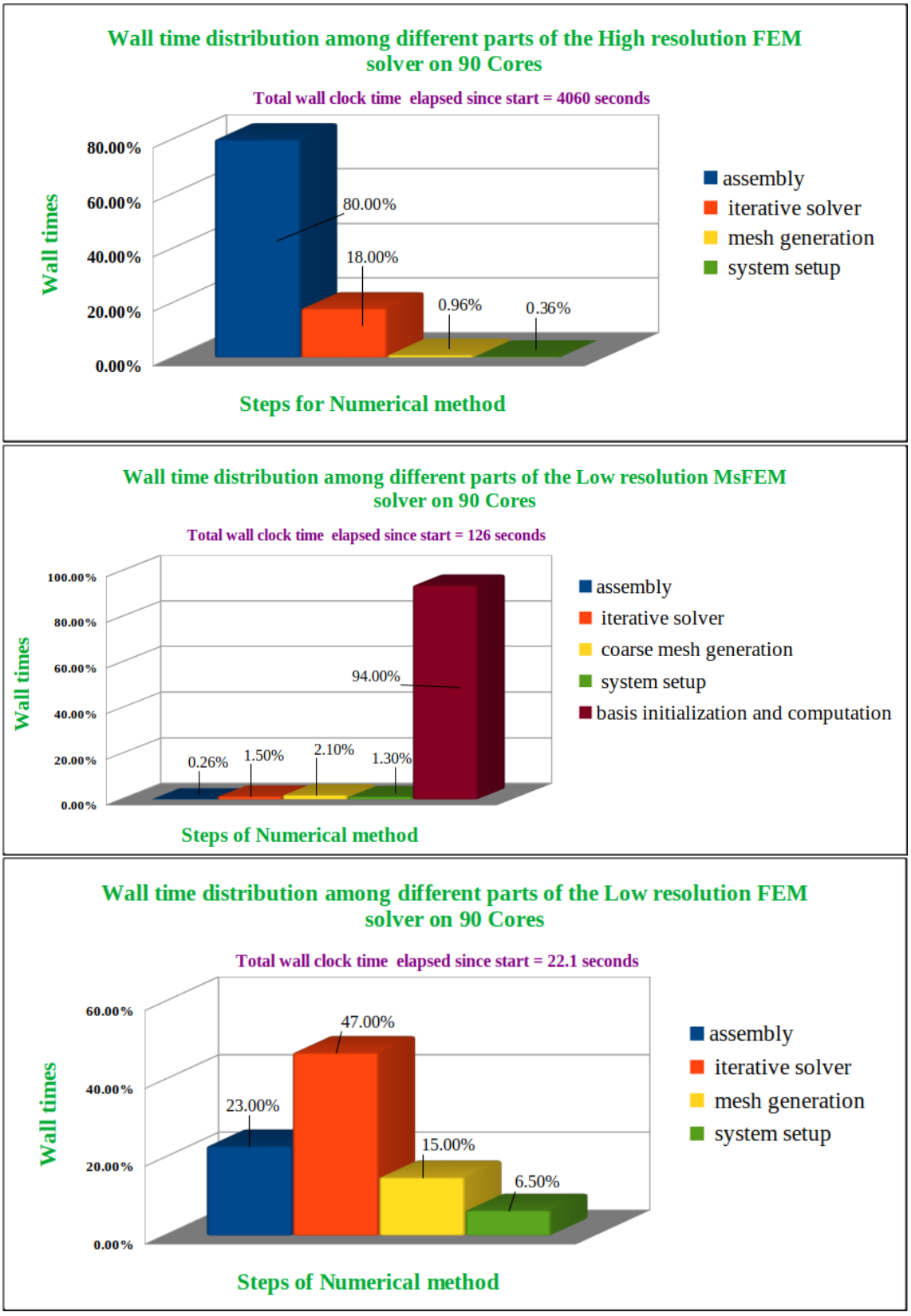


Figure 8.13: Test Case 10 Wall times distribution for High resolution FEM method, Low resolution MsFEM method and Low resolution FEM method.

It is found in Figure (8.13) that the iterative solver takes the maximum time for low resolution FEM, and basis initialization and computation for low resolution MsFEM, whereas assembly takes the most computation for high resolution FEM. The total wall clock time elapsed since start is 4060 seconds for high resolution FEM, 126 seconds for low resolution MsFEM and 22.1 seconds for FEM.

8.3.2 Test case 10 : Error Table

Here, the error is calculated using the difference in the reference solution, which is a high-resolution FEM.

Error Analysis			
Simulation Type	L^2 Error	H^1 Error	L^∞ Error
Low resolution FEM	5.9	6	0.0492
Low resolution MsFEM	4.4	4.5	0.0525

Table 8.2: Test case 10 Error in simulation with Low resolution FEM and Low resolution MsFEM.

Table (8.2) shows that the error in low resolution MsFEM is lower than that in low resolution FEM in L^2 and H^1. L^∞ error is similar in both cases.

8.3.3 Test case 10 : Wind tunnel Validation

The simulation data is normalized by multiplying the value of ethane gas passed which was 4.16 milliliters/second below the canopy. Above the canopy the simulation value was multiplied by 4.16 the result is then divided by height of the canopy. Figure (8.14) presents the simulation of wind tunnel validation compared with the numerical simulation with the source point at 2 measurement points before and above the building taken at different heights. It can be seen that some points at the building height match the wind tunnel measurements. The emission point occurs before the building. First, the graph on the top shows a reasonable agreement between the source and the building. The second graph on the bottom is the point above the building. Here wind tunnel measurements and numerical simulations is close at a 50m building height.

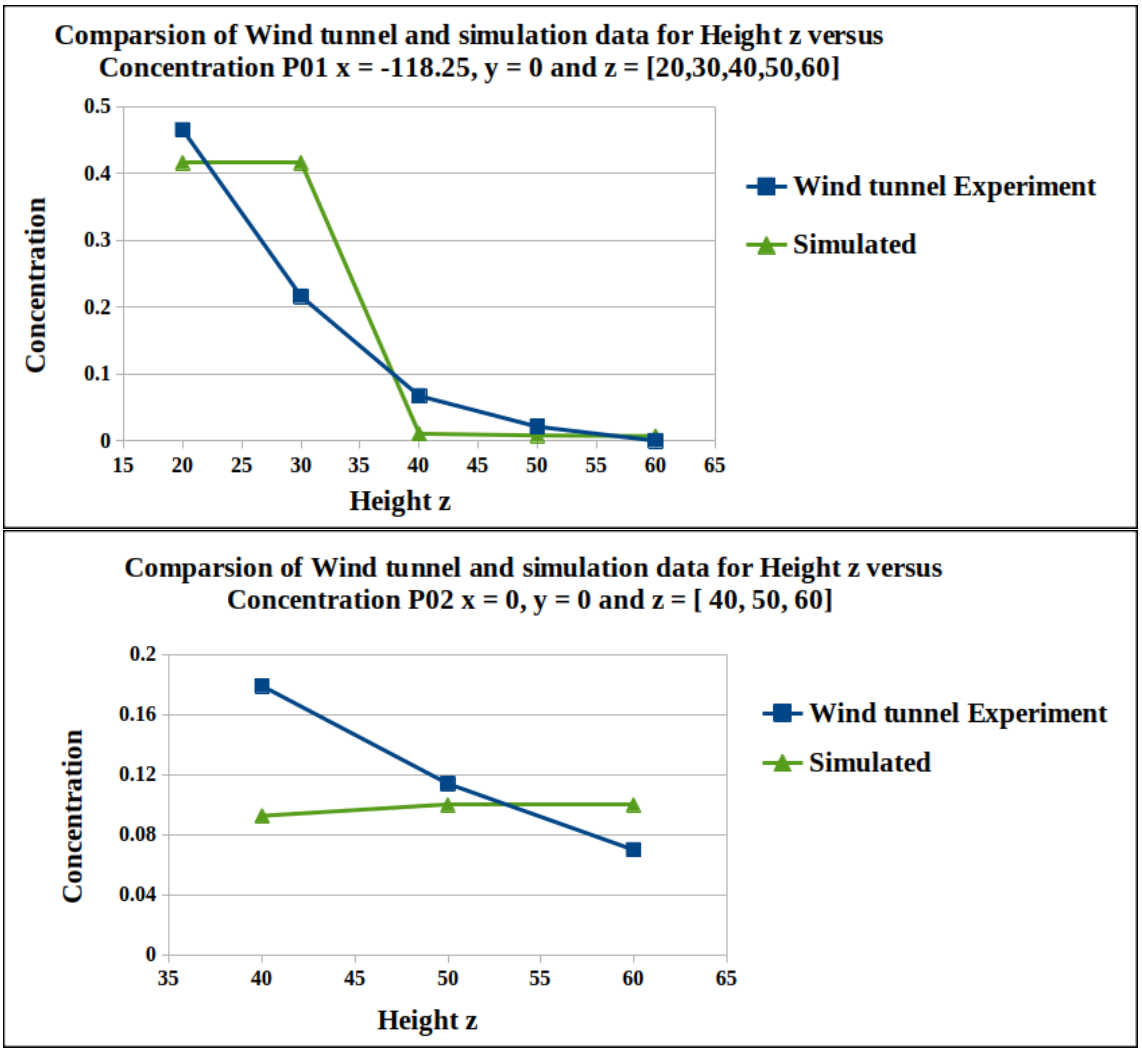


Figure 8.14: Wind tunnel Validation results for Test Case 10 for P01 and P02.

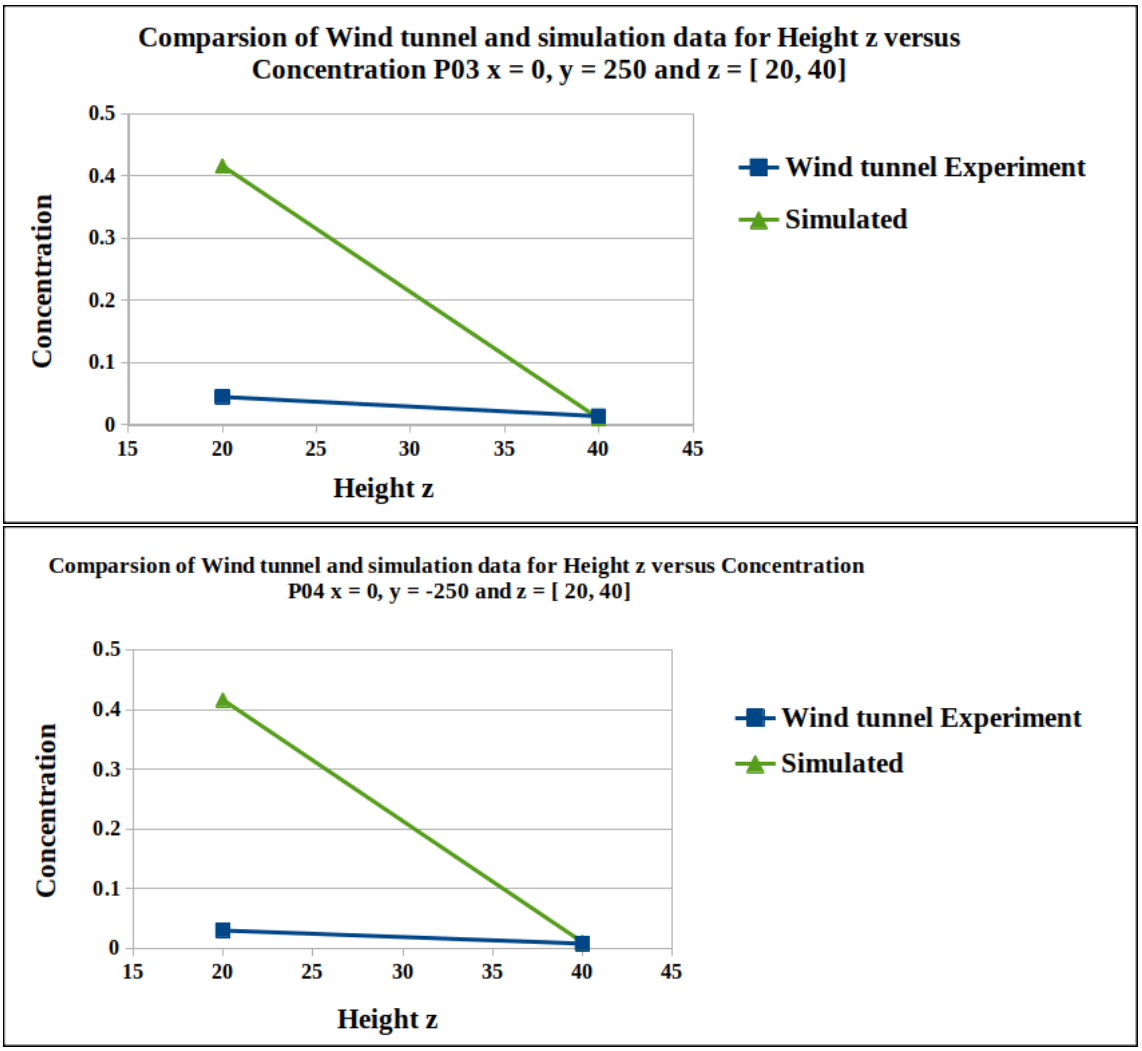


Figure 8.15: Wind tunnel Validation results for Test Case 10 for P03 and P04.

Figure (8.15) shows the results for points 3 and 4 at the height at the left and right corner of the domain. It shows that at 40m building height concentration data match the wind tunnel measurements and the numerical simulations.

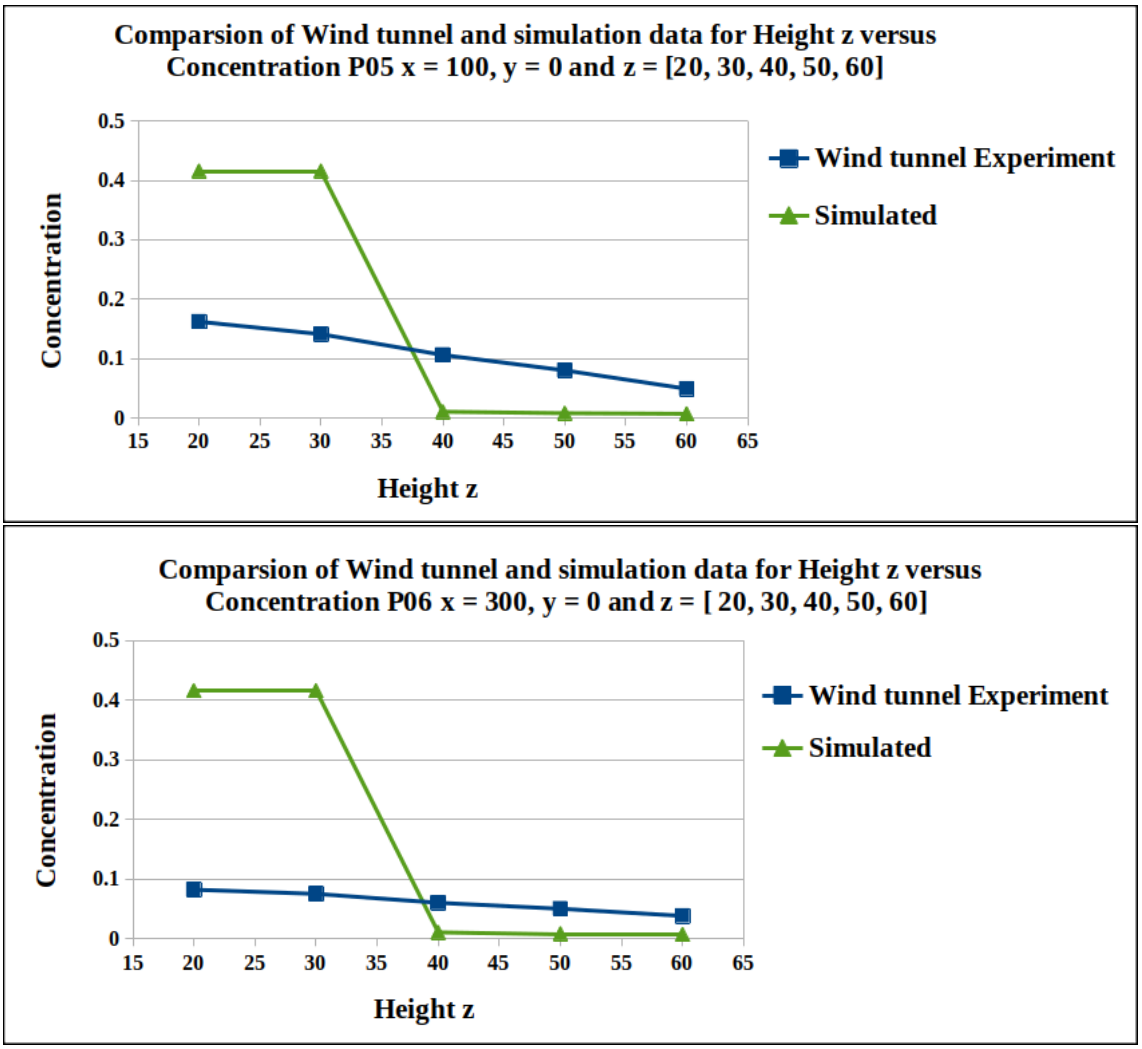


Figure 8.16: Wind tunnel Validation results for Test Case 10 for P05 and P06.

Figure (8.16) displays the results for points 5 and 6 at the domain heights. The points are located after the canopy. It shows that in the first graph on the top there is less agreement between wind tunnel measurements and numerical simulations. The remaining graph shows points measured after the building and on the left and right sides. As can be seen in the second graph on the bottom, measurements and simulations of high resolution FEM concentration data are close above building heights.

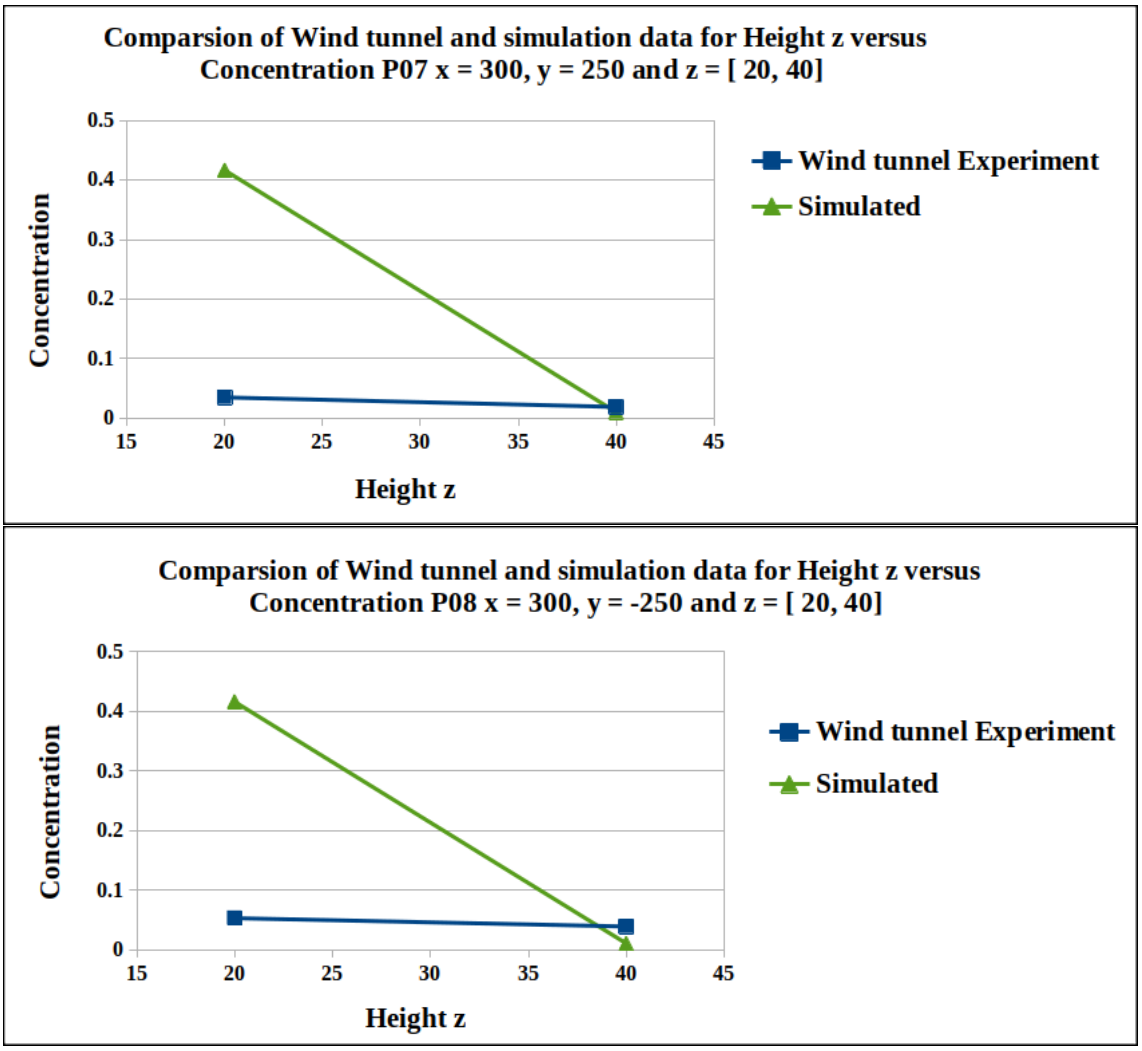


Figure 8.17: Wind tunnel Validation results for Test Case 10 for P07 and P08.

Figure (8.17) shows the results for points 7 and 8 at the height at the left and right corner of the domain. It shows that at 40m height concentration data is close between wind tunnel measurements and numerical simulations.

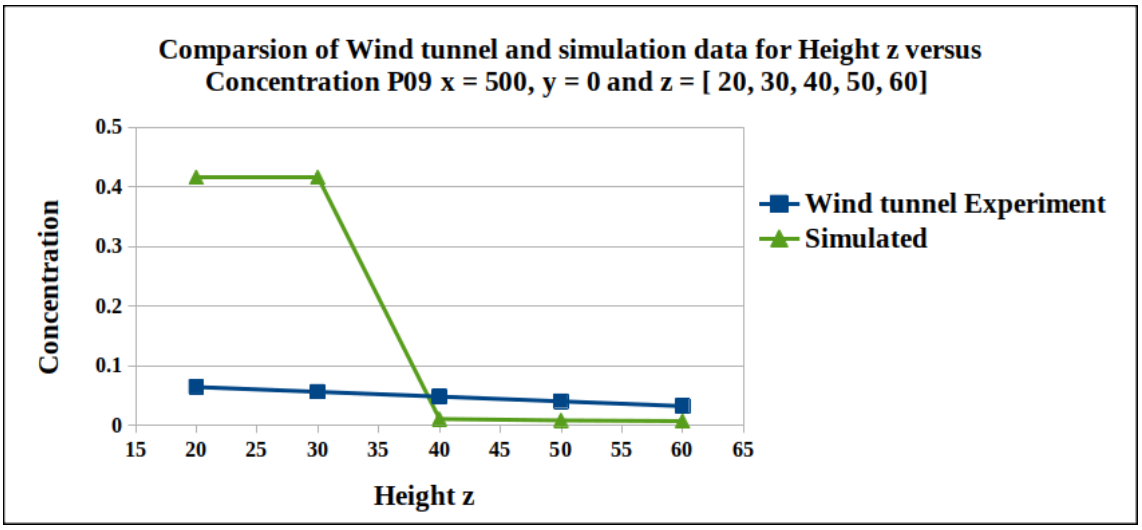


Figure 8.18: Wind tunnel Validation results for Test Case 10 for P09.

According to the graph (8.18), some points above the building height show some similarity between the simulated and wind tunnel measurements.

Generally, some points match and some do not because the wind tunnel data contains turbulence and simulated data does not. Simulated and wind tunnel data points do not match when close to domain boundaries.

8.3.4 Test 10a : Building with 0° rotation and full source term

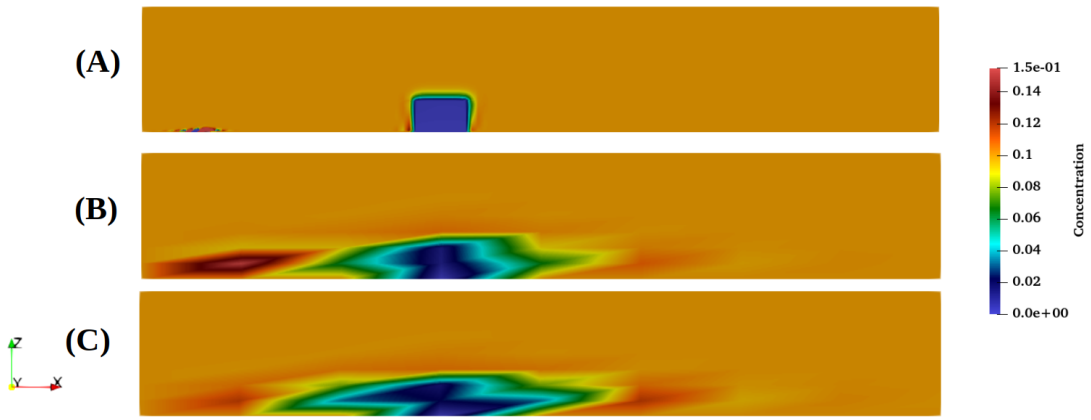


Figure 8.19: Test Case 10a for full source.

Figure (8.19) illustrates the 2D cross section of the 3D domain. Same as the previous test case with 100% source it is found that in Low resolution MsFEM is able to show the point source effect on a large scale. High resolution FEM can capture the point source

most accurately, whereas low resolution FEM shows some effect of the point source.

8.4 Test Case 11 : Building with 45° rotation

The domain setup is the same except the building is rotated to 45° as shown in Figure 8.3.

The high-resolution FEM test case 11 has 16974593 degrees of freedom.

There are 729 degrees of freedom in both the low resolution FEM and MsFEM.

1. New position

$$new_x = xcos(\theta) - ysin(\theta)$$

$$new_y = -xsin(\theta) + ycos(\theta)$$

where θ=45 is the rotation angle given in radians.

2. Dirichlet Condition on odd boundaries

$$u_0 = \begin{cases} 0 \quad \textbf{when } -55 \leq new_x \leq 55,\ -47 \leq new_y \leq 47\ and \quad 0 \leq z \leq 30 \quad \textbf{Building} \\ \\ 0.1 \quad \textbf{else} \end{cases}$$

3. Initial condition at t = 0

$$u(x,y,z) = \begin{cases} 0 \quad \textbf{when} \quad -55 \leq new_x \leq 55, \quad -47 \leq new_y \leq 47 \quad and \quad 0 \leq z \leq 30 \quad \textbf{Building} \\ \\ 0.1 \quad \textbf{else} \end{cases}$$

4. Neumann Condition on left hand side boundary

$$g = exp(-(z-60)^2)$$

5. Diffusion Coefficient

$$a(x,y,z) = \begin{cases} 3000 & \textbf{when } -55 \leq new_x \leq 55,\ -47 \leq new_y \leq 47 \textit{ and } 0 \leq z \leq 30 \textbf{ Building} \\ 4z & \textbf{else} \end{cases}$$

6. Velocity

$$c(x,y,z) = \begin{cases} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} & \textbf{when } -55 \leq new_x \leq 55, -47 \leq new_y \leq 47 \\ & \textit{and} \quad 0 \leq z \leq 30 \textbf{ Building} \\ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} & \textbf{when } -300 \leq x \leq 500, \textit{and} \\ & z = 0 \ \textbf{Bottom Building} \\ \begin{pmatrix} c_h \\ 0 \\ 0 \end{pmatrix} & \text{else} \quad \textbf{Around the Building} \text{ where } c_h \text{ is reference velocity} \\ \begin{pmatrix} (c*/\kappa) \cdot log((z+z_0)/z_0) \\ 0 \\ 0 \end{pmatrix} & z \geq 30 \quad \textbf{Above the Building} \text{ where } c* \text{ is friction velocity, } \kappa \text{ is von Kármán constant and } z_0 \text{ is roughness length} \end{cases}$$

where $c_h = 4$ is the wind velocity at $h = 30$, $y_0 = 0.5$ is the surface roughness, k is the von Kármán constant $(k = 0.4)$,and $c^* = 0.35$ is the friction velocity.

7. Right hand side

$$f = 0$$

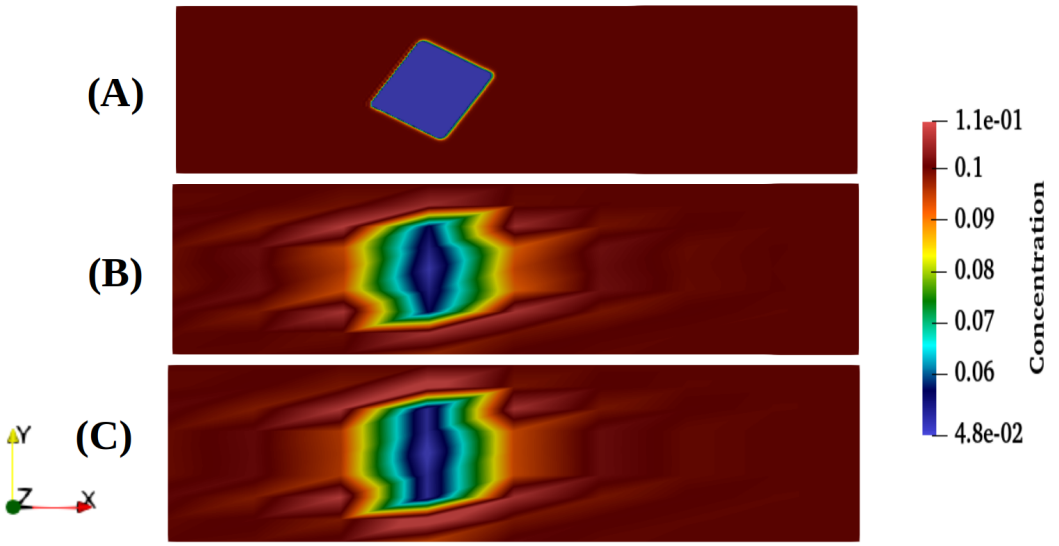


Figure 8.20: Test case 11 Wind tunnel test case single building with 45° degree rotation (A) High resolution FEM with refinement = 7 (B) Low resolution MsFEM with coarse refinement = 3, fine refinement = 4 and (C) Low resolution FEM with refinement = 3.

As shown in Figure (8.20) shows a 2D cross-section of a 3D domain the building with 45° rotations at the 10m z normal, (A) high-resolution FEM is the reference solution, whereas (B) low resolution MsFEM shows the canopy structure at a larger scale with 45° rotation. In (C) the low-resolution FEM cannot maintain 45° rotations.

8.4.1 Test case 11 : Wall time distribution

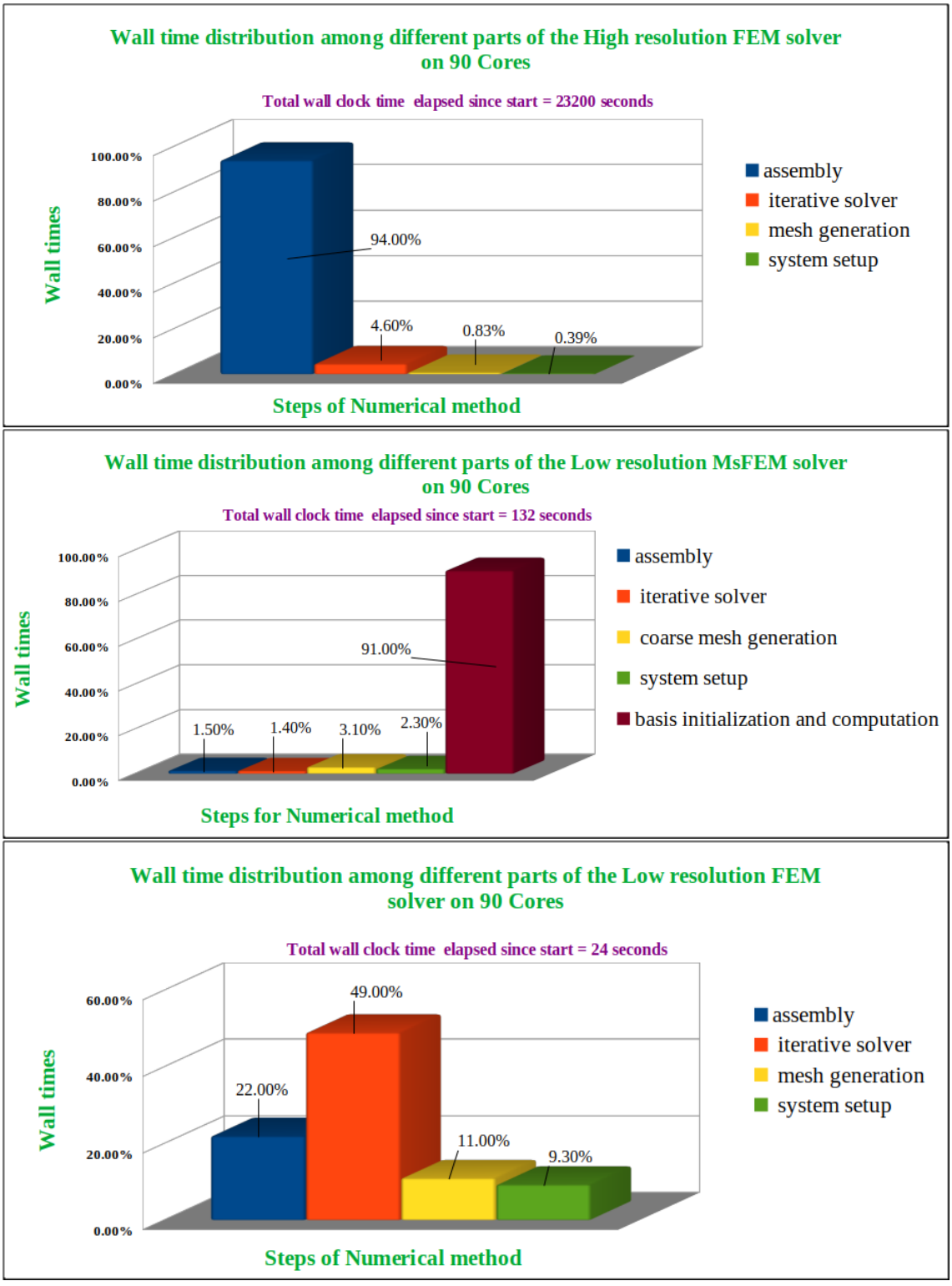


Figure 8.21: Test Case 11 Wall times distribution for High resolution FEM method, Low resolution MsFEM method and Low resolution FEM method.

Figure (8.21) presents the wall time distribution of the computation. It is found that the iterative solver takes the maximum time for low- resolution FEM, and for low resolution MsFEM basis initialization and computation takes maximum time, whereas assembly takes the most computation for high resolution FEM. The total wall clock time elapsed since start is 23200 seconds for high resolution FEM, 132 seconds for low resolution MsFEM and 24 seconds for FEM.

8.4.2 Test case 11 : Error Table

Here, the error is calculated using the difference in the reference solution, which is a high resolution FEM.

Error Analysis			
Simulation Type	L^2 Error	H^1 Error	L^∞ Error
Low resolution FEM	0.1	0.1	0.0055
Low resolution MsFEM	0.1	0.1	0.0077

Table 8.3: Test case 11 Error in simulation with Low resolution FEM and Low resolution MsFEM.

Table (8.3) shows contrary to Test 9, the error in low resolution MsFEM is higher than low resolution FEM. As the building is located diagonally in the coarse mesh, the quadrature points are less for low resolution MsFEM than high resolution FEM. Also in the MsFEM there quadrature points for the building are calculated as diffusion coefficient at subgrid mesh. For both the low and high resolution FEM these quadrature points are on coarse and fine mesh respectively. L^∞ error does not perform well for functions with jumps. Therefore, even though low resolution FEM has fewer quadrature points than high resolution FEM, overall their error calculation on each cell is close.

8.4.3 Test Case 11a : Building with 45° rotation with increase in MsFEM refinement

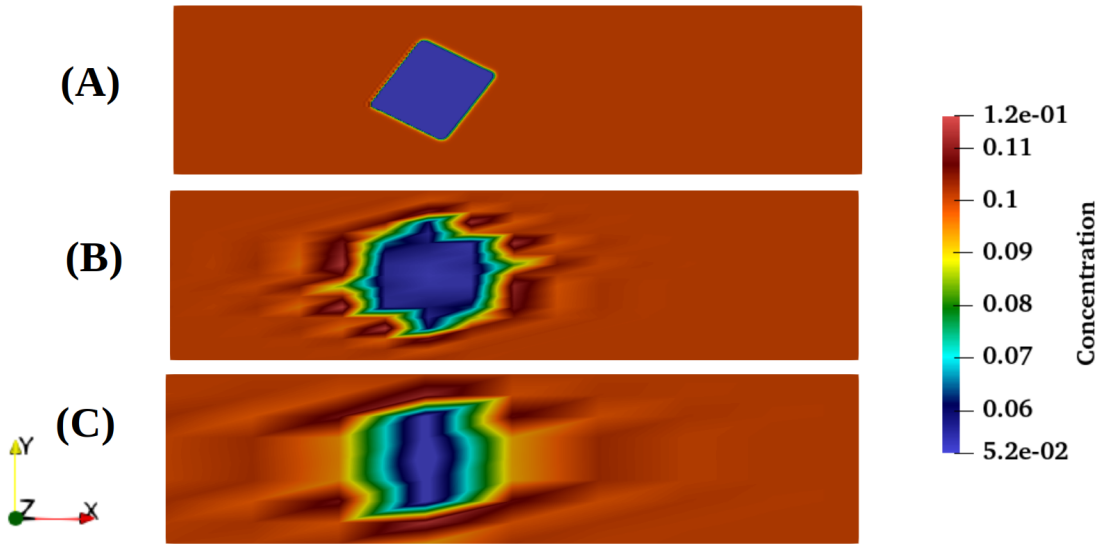


Figure 8.22: Test case 11a Wind tunnel test case single building with 45° degree rotation (A) High resolution FEM with refinement = 8 (B) Low resolution MsFEM with coarse refinement = 4, fine refinement = 4 and (C) Low resolution FEM with refinement = 3.

As shown in Figure (8.22) shows a 2D cross-section of a 3D domain of the building with 45° rotations at the 10m z normal, (A) high-resolution FEM is the reference solution, whereas (B) low resolution MsFEM with 4 refinement levels both on coarse and fine grid shows the canopy structure at a larger scale with 45° rotation and edges are also seen due to increase in refinement. In (C) the low-resolution FEM with 3 refinement levels cannot maintain 45° rotations.

8.4.4 Test Case 11b : Building with 45° rotation with decrease in diffusion

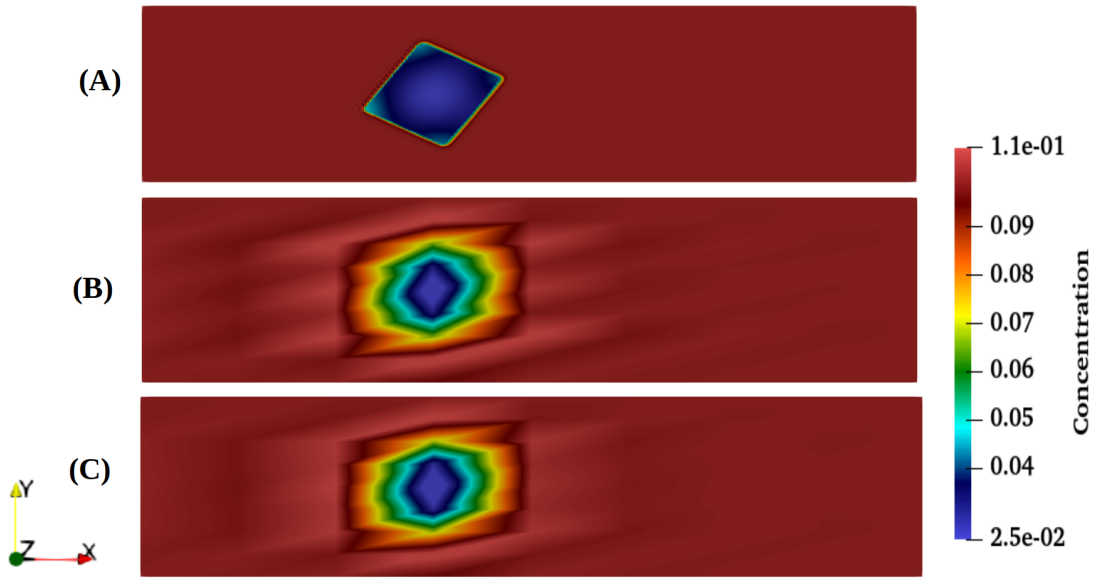


Figure 8.23: Test case 11b Wind tunnel test case single building with 45° degree rotation (A) High resolution FEM with refinement = 8 (B) Low resolution MsFEM with coarse refinement = 4, fine refinement = 3 and (C) Low resolution FEM with refinement = 3.

As shown in Figure (8.23) shows a 2D cross-section of a 3D domain of the building with 45° rotations and diffusion coefficients inside canopy is 1000, at the 1m z normal, (A) high-resolution FEM is the reference solution, but canopy edges are not clear whereas in (B) low resolution MsFEM and in (C), the low-resolution FEM shows similar canopy structure except the near canopy effect is better in low resolution MsFEM.

Overall, the canopy structure with rotation varies in mesh size and diffusion coefficient from case 11 to case 11a and case 11b, but it is more difficult to get a simulation in case 11 because of rotation than in case 9 and 10.

8.5 Test Case 12 : Building with 45° rotation and source term

As with test case 11, the setup is similar, but source is added before building.

In high-resolution FEM test case 11, there are 16974593 degrees of freedom.

Low-resolution FEMs and MsFEMs have 729 degrees of freedom.

1. New position

$$new_x = xcos(\theta) - ysin(\theta)$$

$$new_y = -xsin(\theta) + ycos(\theta)$$

where θ = 45 is the rotation angle given in radians.

2. Dirichlet Condition on right and top boundaries

$$u_0 = \begin{cases} 0 \quad \textbf{when} \quad -55 \leq new_x \leq 55, \quad -47 \leq new_y \leq 47 \quad and \quad 0 \leq z \leq 30 \quad \textbf{Building} \\ \\ 0.1 \quad \textbf{else} \end{cases}$$

3. Initial condition at t = 0

$$u(x,y,z) = \begin{cases} 0 \quad \textbf{when} \quad -55 \leq new_x \leq 55, \quad -47 \leq new_y \leq 47 \quad and \quad 0 \leq z \leq 30 \quad \textbf{Building} \\ \\ 0.1 \quad \textbf{else} \end{cases}$$

4. Neumann Condition on left hand side boundary

$$g = exp(-(z-60)^2)$$

5. Diffusion Coefficient

$$a(x,y,z) = \begin{cases} 3000 \quad \textbf{when} \; -55 \leq new_x \leq 55, \; -47 \leq new_y \leq 47 \; and \; 0 \leq z \leq 30 \; \textbf{Building} \\ \\ 4z \qquad \textbf{else} \end{cases}$$

6. Velocity

$$c(x,y,z) = \begin{cases} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} & \textbf{when } -55 \le new_x \le 55, -47 \le new_y \le 47 \\ & and \quad 0 \le z \le 30 \textbf{ Building} \\ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} & \textbf{when } -300 \le x \le 500, and \\ & z = 0 \textbf{ Bottom Building} \\ \begin{pmatrix} c_h \\ 0 \\ 0 \end{pmatrix} & \text{else } \textbf{Around the Building} \text{ where } c_h \text{ is reference velocity} \\ \begin{pmatrix} (c*/\kappa) \cdot log((z+z_0)/z_0) \\ 0 \\ 0 \end{pmatrix} & z \ge 30 \textbf{ Above the Building} \text{ where } c* \text{ is friction velocity, } \kappa \text{ is von Kármán constant and } z_0 \text{ is roughness length} \end{cases}$$

where $c_h = 4$ is the wind velocity at $h = 30$, $y_0 = 0.5$ is the surface roughness , k is the von Kármán constant $(k = 0.4)$,and $c^* = 0.35$ is the friction velocity.

7. Right hand side

$$f = \begin{cases} 1 & \textbf{when } -239.5 \le x \le -236.5, \quad y \le 1 \quad and \quad z \le 1 \quad \textbf{point source} \\ 0 & \textbf{else} \end{cases}$$

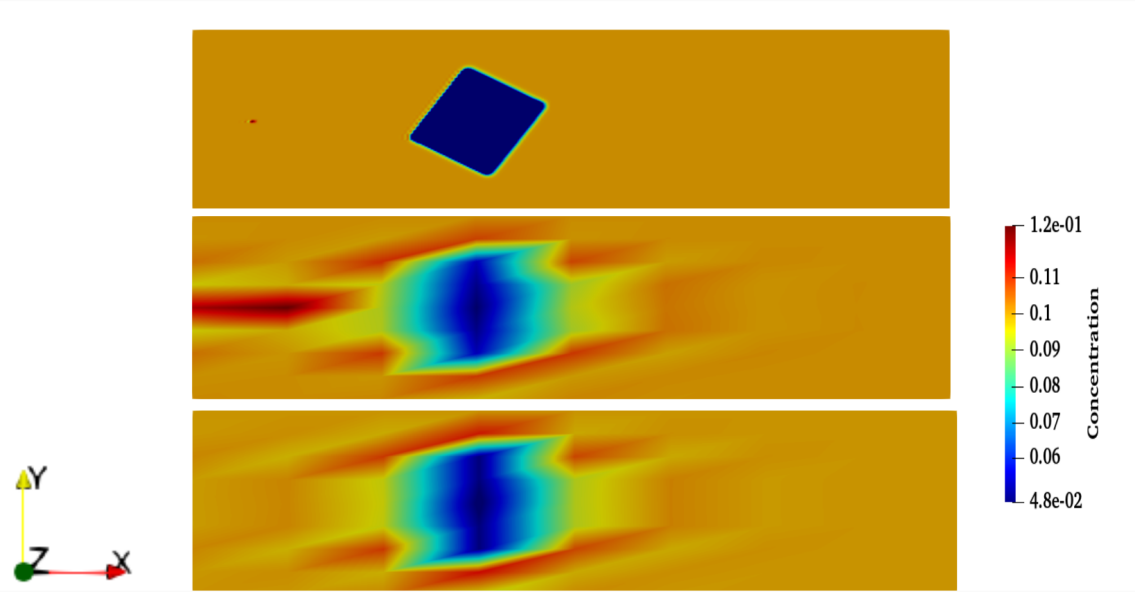


Figure 8.24: Test case 12 Wind tunnel test case single building with 45° degree rotation and source (A) High resolution FEM with refinement = 7 (B) Low resolution MsFEM with coarse refinement =4, fine refinement = 4 and (C) Low resolution FEM with refinement = 3.

In Figure (8.24),shows a 2D cross-section of a 3D domain the building with 45° rotations at the 1m z normal with a point source (A) High resolution FEM is the reference solution, and (B) low-resolution MsFEM shows canopy structure at large scale. The point source is also seen because the refinement was high at coarse scale. In (C) the low resolution FEM cannot maintain 45° rotations.

8.5.1 Test case 12 : Wall time distribution

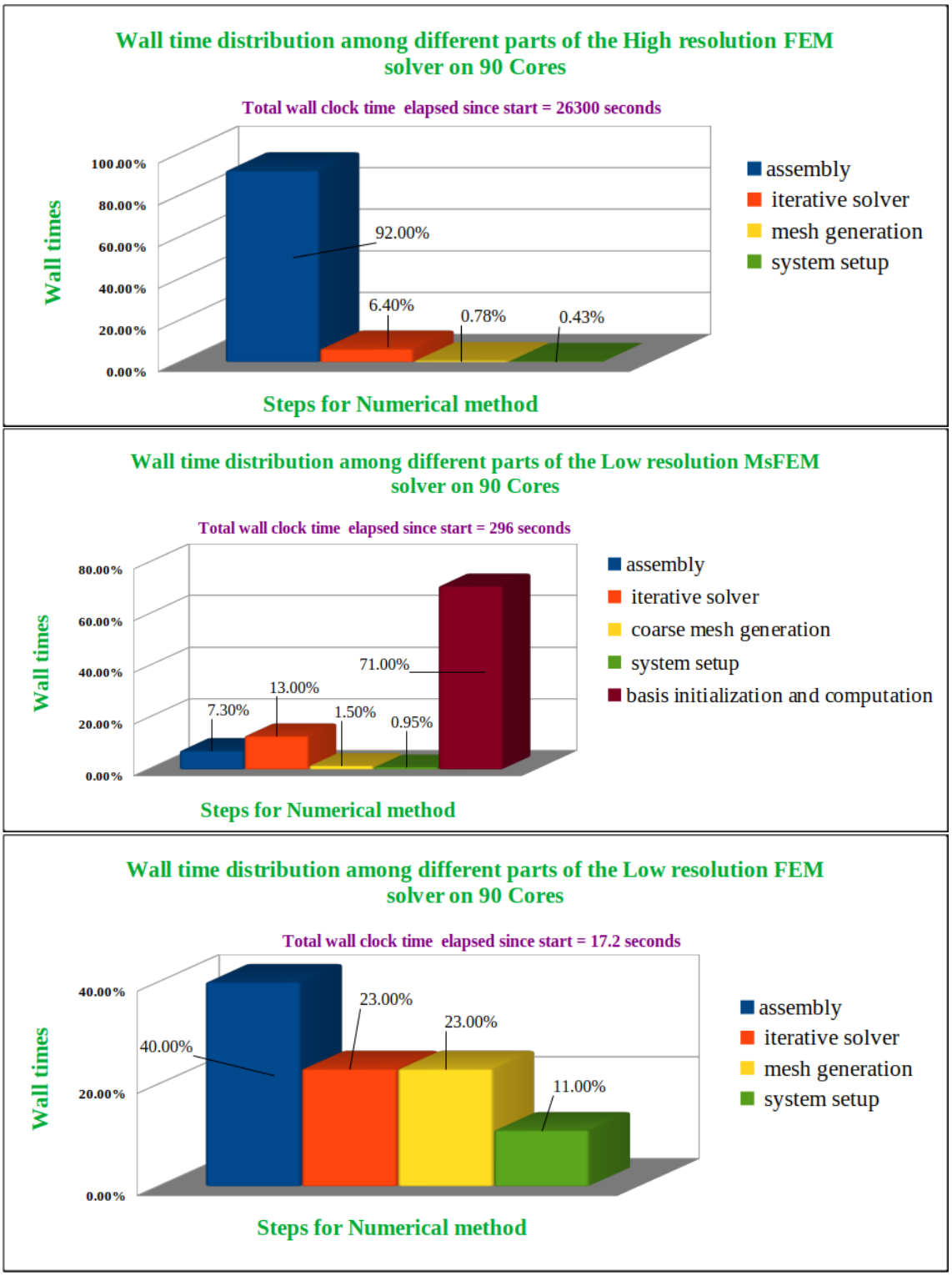


Figure 8.25: Test Case 12 Wall times distribution for High resolution FEM method, Low resolution MsFEM method and Low resolution FEM method.

Figure (8.25) presents the computation wall time distribution. It is found that the iterative solver takes the maximum time for low- resolution FEM, and for low resolution

MsFEM the basis initialization and computation takes maximum time, whereas assembly takes the most computation for high resolution FEM. The total wall clock time elapsed since start is 26300 seconds for high resolution FEM, 296 seconds for low resolution MsFEM and 17.2 seconds for FEM.

8.5.2 Test case 12 : Error Table

Here, the error is calculated using the difference in the reference solution, which is a high-resolution FEM.

Error Analysis			
Simulation Type	L^2 Error	H^1 Error	L^∞ Error
Low resolution FEM	0.1	0.1	0.0928
Low resolution MsFEM	0.1	0.1	0.0864

Table 8.4: Test case 12 Error in simulation with Low resolution FEM and Low resolution MsFEM.

Table (8.4) shows the error in low resolution MsFEM is less than low resolution FEM.

8.5.3 Test case 12 : Wind tunnel Validation

The simulation data is normalized by multiplying the value of ethane gas passed which was 4.16 milliliters/second below the canopy. Above the canopy the simulation value was multiplied by 4.16 the result is then divided by height of the canopy. A comparison between a simulation of wind tunnel validation and a numerical simulation is shown in Figure (8.26). At some points in the building height, the wind tunnel measurements match. Emission occurs before the building. In the graph on the top, we can see a some agreement between the source and the building. In the second graph on the bottom, you can see the point above the building. At a building height of 50m, wind tunnel measurements and numerical simulations agree.

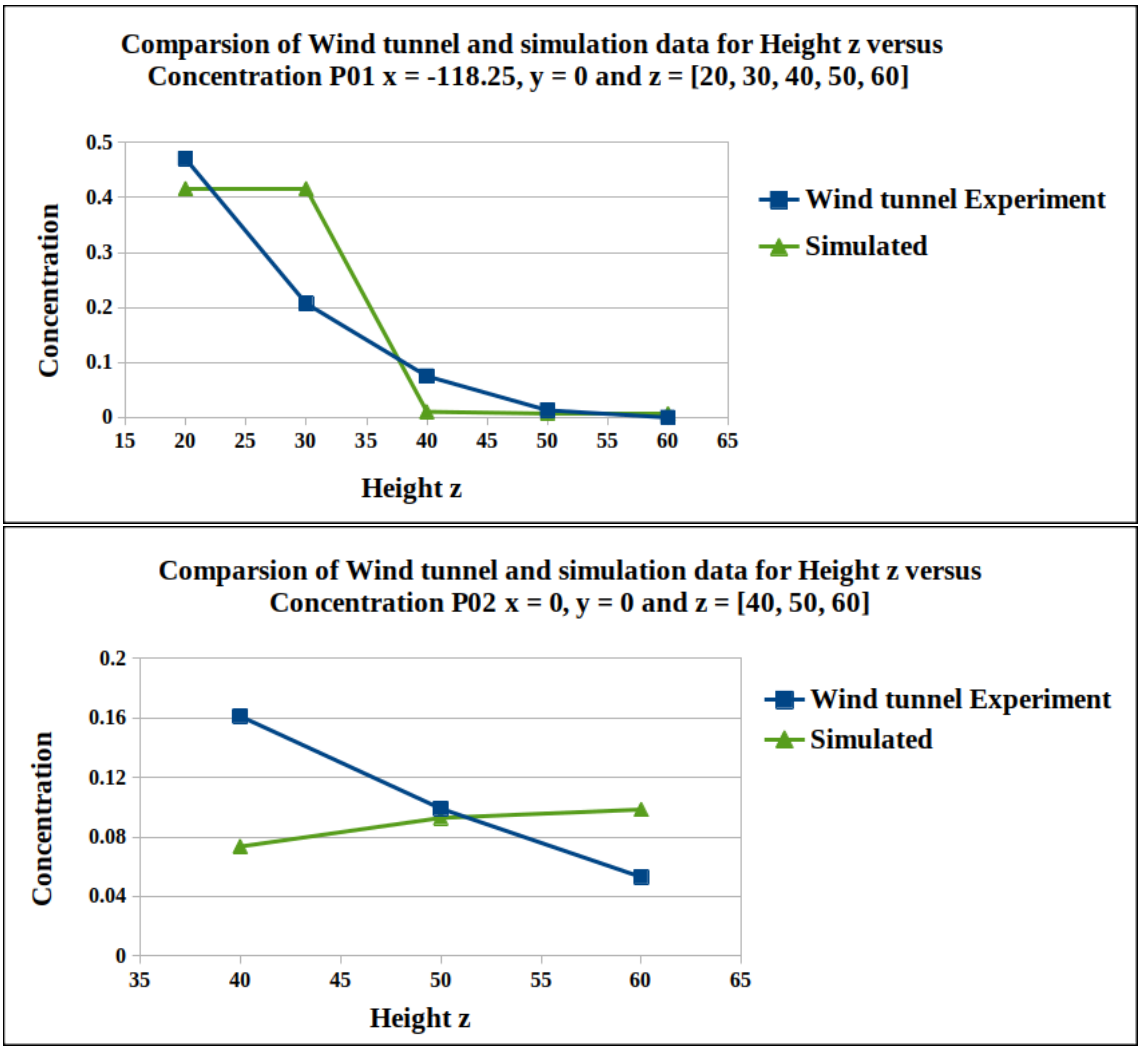


Figure 8.26: Wind tunnel Validation results for Test Case 12 for point P01 and P02.

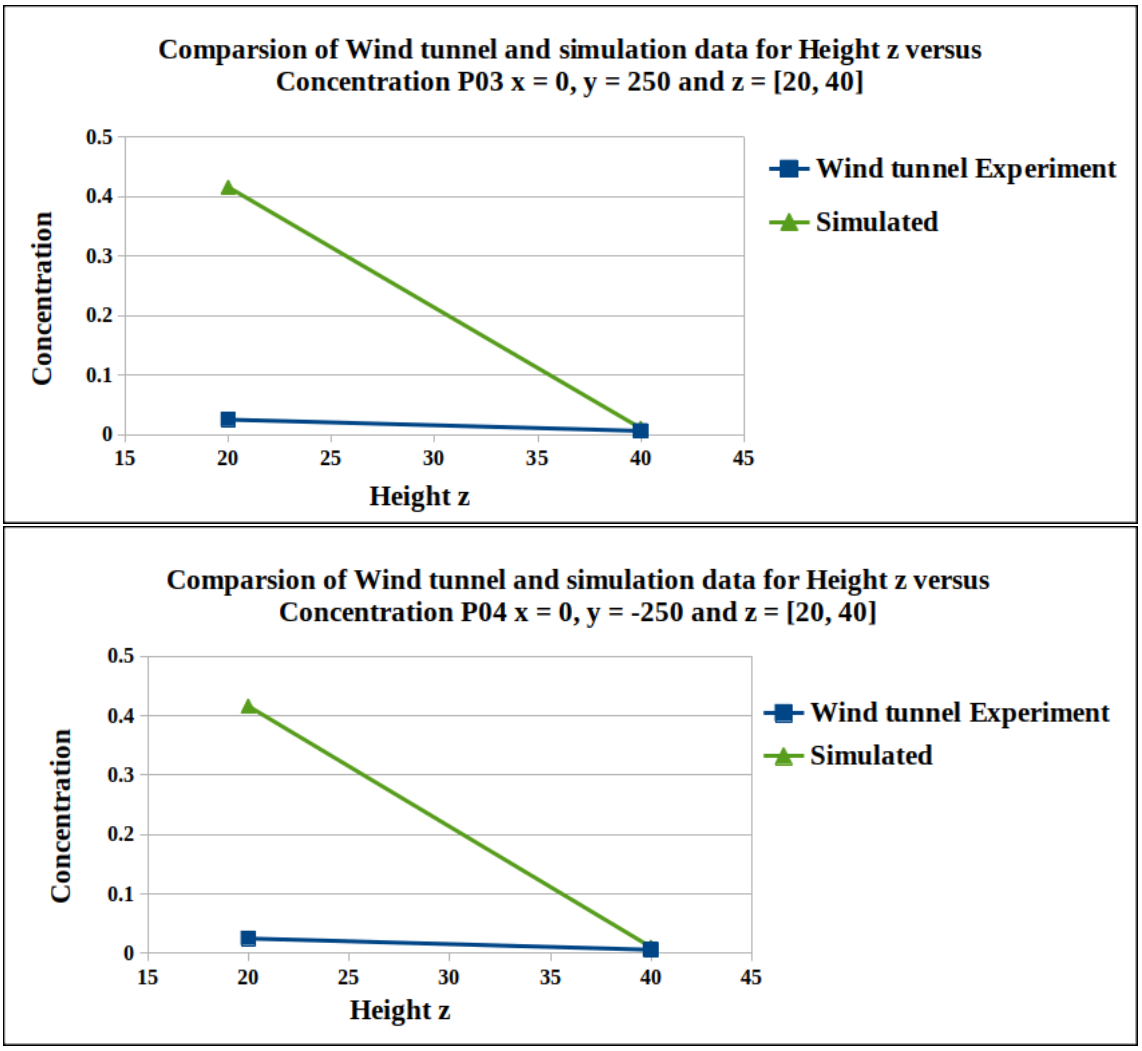


Figure 8.27: Wind tunnel Validation results for Test Case 12 for point P03and P04.

The results are shown in Figure (8.27) at the height in the left and right corners of the domain for points 3 and 4. The wind tunnel measurements and numerical simulations match the concentration data at 40m building height.

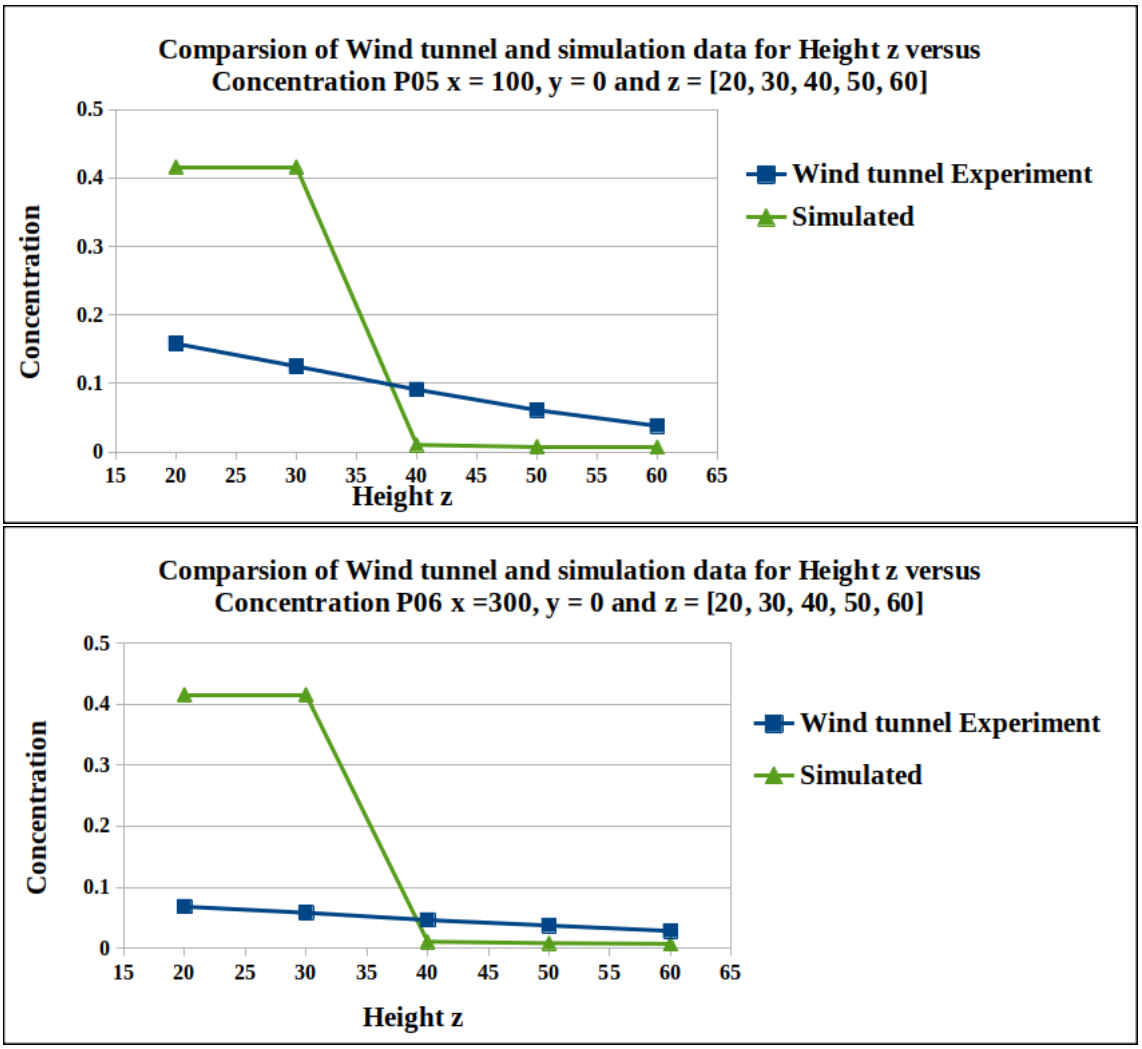


Figure 8.28: Wind tunnel Validation results for Test Case 12 for point P05 and P06.

Figure (8.28) shows the results for points 5 and 6 at the domain heights. Points are located after the canopy. According to the first graph at the top, numerical simulations and wind tunnel measurements are not always in agreement. On the left and right sides of the building, points were measured after the building. In the bottom graph, we can see that measurements and simulations of the high resolution FEM concentration data values are close above the building heights.

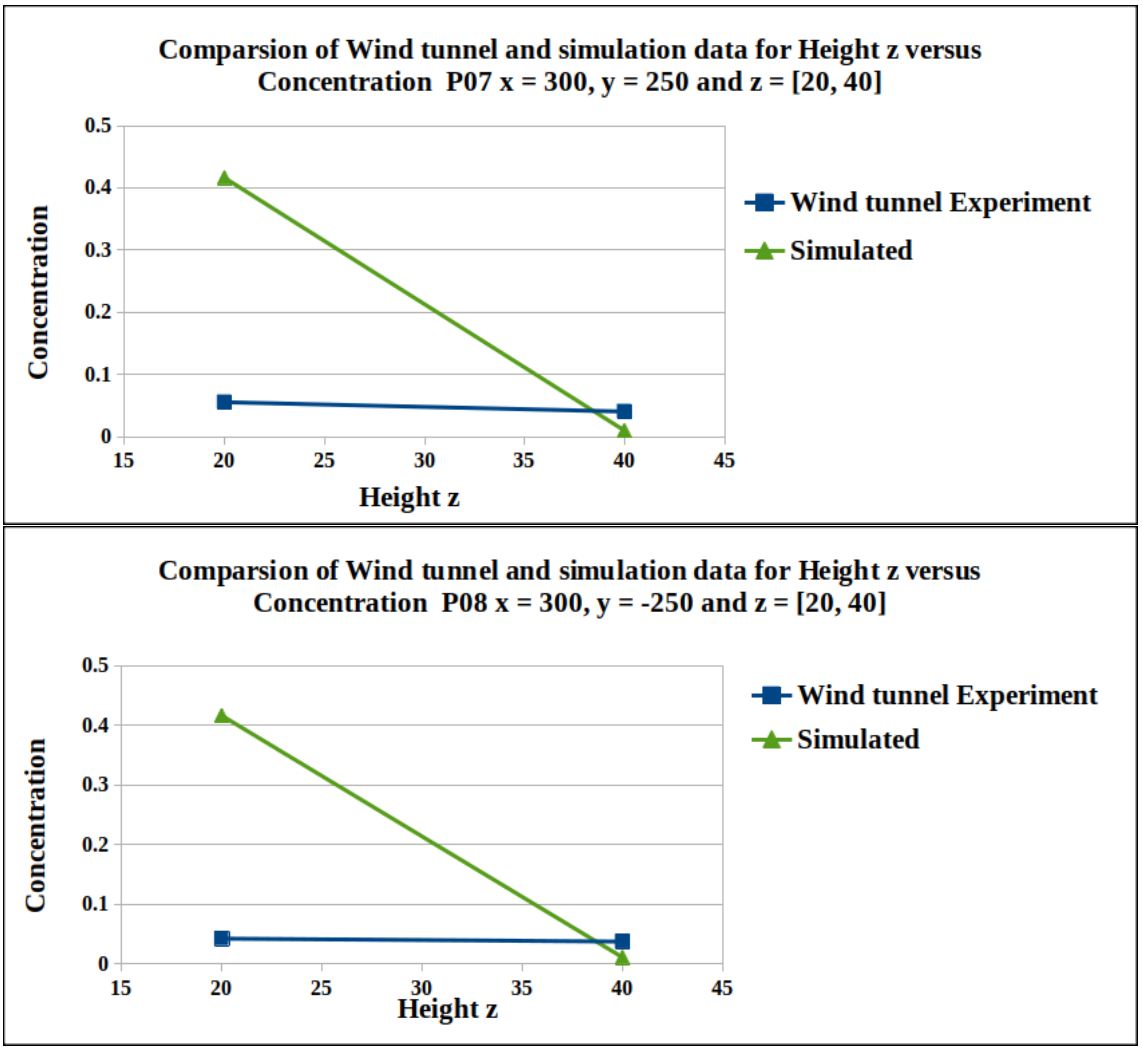


Figure 8.29: Wind tunnel Validation results for Test Case 12 for point P07 and P08.

In Figure (8.29), you can see the results at points 7 and 8 at the left and right corners of the domain. Data from wind tunnel measurements and numerical simulations are similar at 40 meters height.

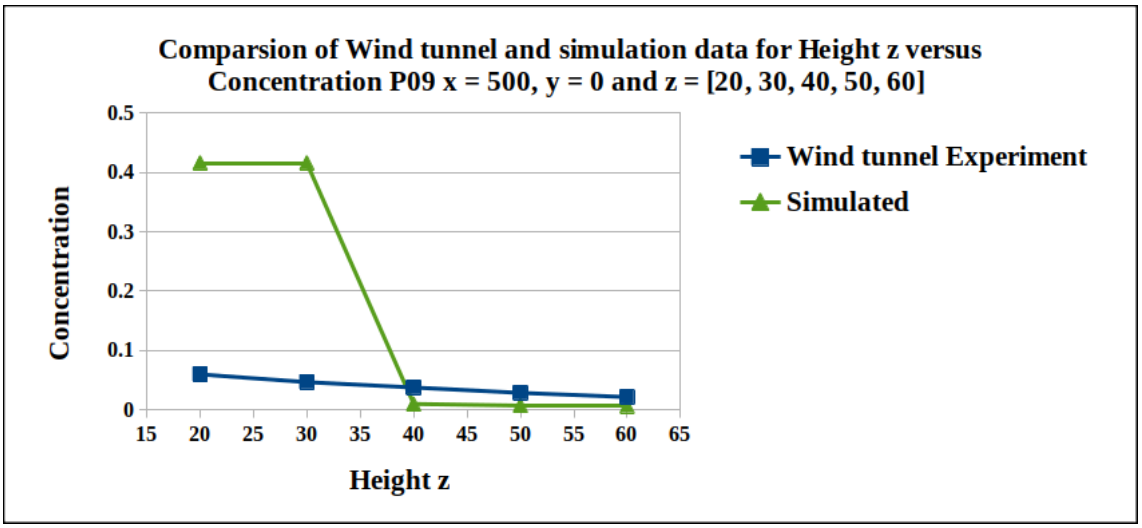


Figure 8.30: Wind tunnel Validation results for Test Case 12 for point P09.

The Figure (8.30) shows similarities between simulated and wind tunnel measurements above building height.

Generally, some points match and some do not because the wind tunnel data contains turbulence and simulated data does not. Simulated and wind tunnel data points do not match when close to domain boundaries.

8.5.4 Test Case 12a : Building with 45° rotation with increase in MsFEM refinement

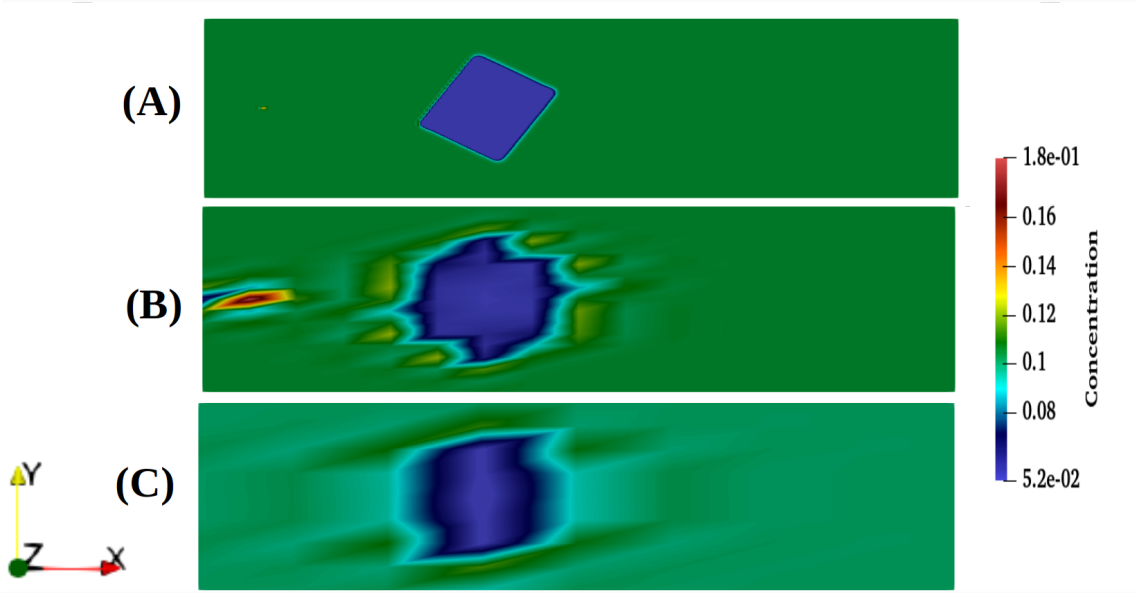


Figure 8.31: Test case 12a Wind tunnel test case single building with 45° degree rotation and source (A) High resolution FEM with refinement = 8 (B) Low resolution MsFEM with coarse refinement = 4, fine refinement = 4 and (C) Low resolution FEM with refinement = 3.

As shown in Figure (8.31) shows a 2D cross-section of a 3D domain of the building with 45° rotations at the 10m z normal, (A) high-resolution FEM is the reference solution, whereas in (B) low resolution MsFEM with 4 refinement levels both on coarse and fine grid shows the canopy structure at a larger scale with 45° rotation and edges are also seen due to increase in refinement. In (C) the low-resolution FEM with 3 refinement levels cannot maintain 45° rotations.

8.5.5 Test Case 12b : Building with 45° rotation with decrease in diffusion

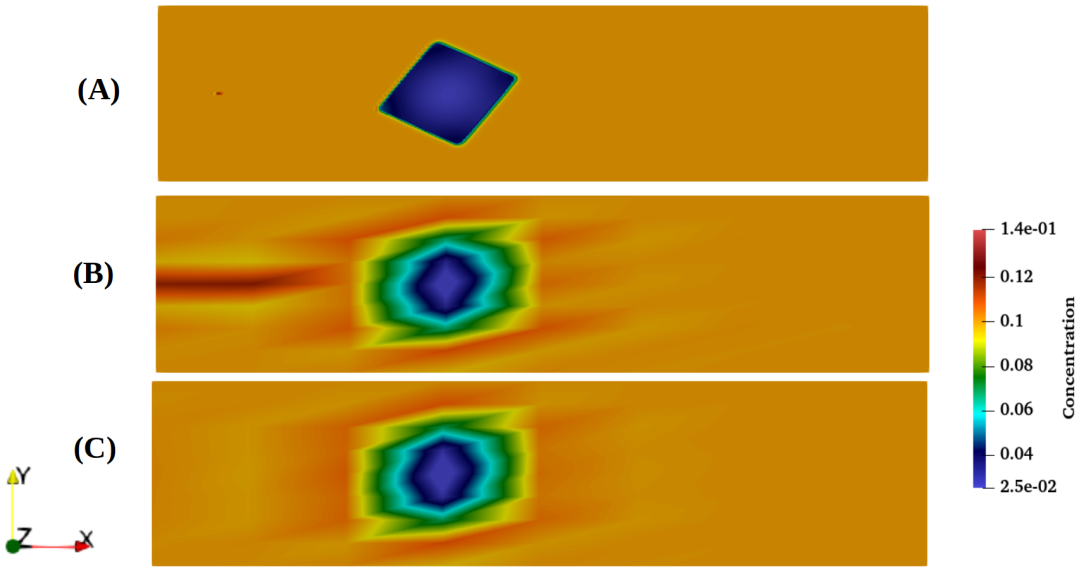


Figure 8.32: Test case 12b Wind tunnel test case single building with 45° degree rotation and source (A) High resolution FEM with refinement = 8 (B) Low resolution MsFEM with coarse refinement = 4, fine refinement = 3 and (C) Low resolution FEM with refinement = 3.

As shown in Figure (8.32) shows a 2D cross-section of a 3D domain of the building with 45° rotations and diffusion coefficients inside canopy is 1000, at the 1m z normal, (A) high-resolution FEM is the reference solution whereas (B) low resolution MsFEM and in (C), the low-resolution FEM shows similar canopy structure except the source is better seen in low resolution MsFEM.

Overall, the canopy structure with rotation varies in mesh size and diffusion coefficient from case 12 to case 12a and case 12b, but it is more difficult to get a simulation in case 12 because of rotation than in case 9 and 10.

8.6 Hamburg mesh

Mesh is obtained from derived, comprehensive digital terrain models with a grid width of 25 meters. For the Free and Hanseatic City of Hamburg (without the Hamburg Wadden Sea), a laser scan (Airborne Laser Scanning) was carried out in 2020. The data are available in field 310 (ETRS89/UTM), with heights of normal height zero (NHN).

The accuracy of a single measuring point lies in clearly defined areas, e.g. on road surfaces, at approx. ± 255 cm. In areas of shade (bridges), vegetation, particularly forest

and shrub areas and in highly inclined terrain, accuracy is lower.

By default, the Landesbetrieb Geoinformation und Vermessung (LGV) offers the following grid widths: Digital Ground Model (DGM), 25 (grid width 25 m). An annual update of this data is carried out via aerial surveys.

The high-resolution FEM for Hamburg Mesh has 1002433 degrees of freedom.

A low resolution FEM and MsFEM have 292 degrees of freedom.

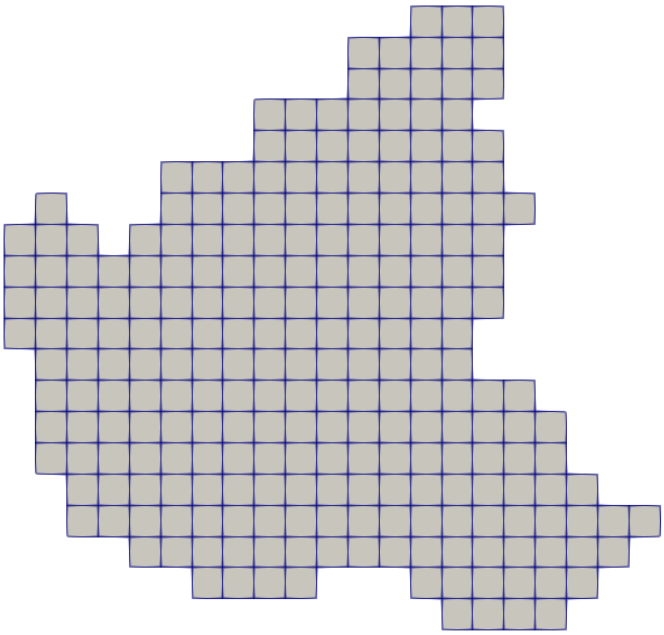


Figure 8.33: Hamburg mesh.

Here we consider 2 km $\times$ 4 km urban area.

1. Dirichlet Condition on even boundary

$$u_0 = \begin{cases} 0 \quad \textbf{when} \quad 570 \leq x \leq 572 \quad 5936 \leq y \leq 5940 \quad \textbf{Building} \\ \\ 1 \quad \textbf{else} \end{cases}$$

2. Initial condition at t = 0

$$u(x,y) = \begin{cases} 0 \quad \textbf{when} \quad 570 \leq x \leq 572 \quad 5936 \leq y \leq 5940 \quad \textbf{Building} \\ \\ 1 \quad \textbf{else} \end{cases}$$

3. Neumann Condition on right hand side

$$g = 0$$

4. Diffusion Coefficient

$$a(x,y) = \begin{cases} 10000 & \textbf{when} \quad 570 \leq x \leq 572, \quad 5936 \leq y \leq 5940 \quad \textbf{Building} \\ 0.001 * y & \textbf{else} \end{cases}$$

5. Velocity

$$c(x,y) = \begin{cases} 0 & \textbf{when} \quad 570 \leq x \leq 572 \quad 5936 \leq y \leq 5940 \quad \textbf{Building} \\ 4 & else \end{cases}$$

6. Right hand side

$$f = 0$$

It is seen from Figure (8.34) that canopy is well represented in high resolution FEM and low resolution MsFEM shows similarity with results to high resolution FEM whereas low resolution FEM does not represent the urban block well.

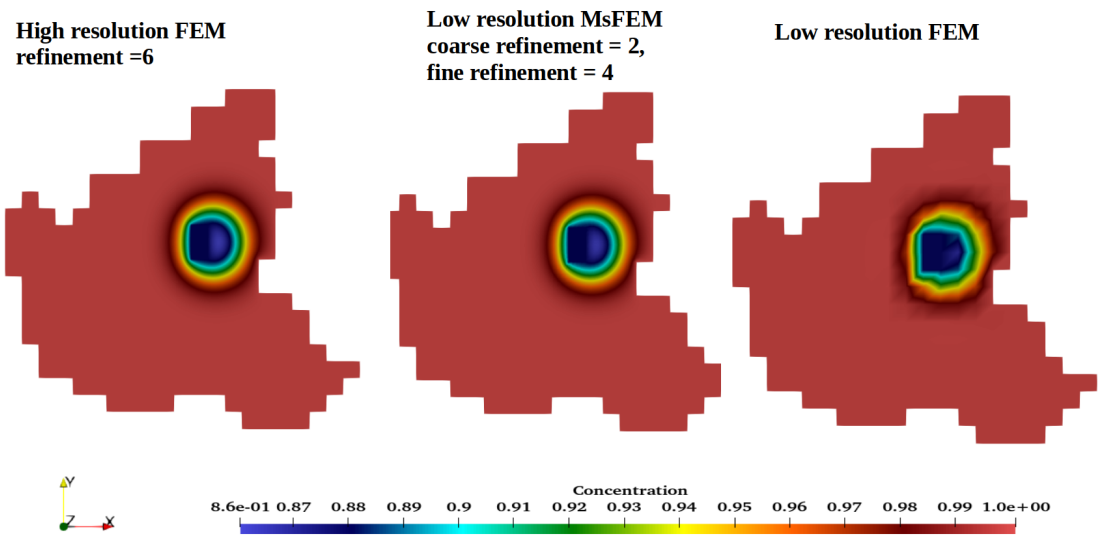


Figure 8.34: Result of Hamburg Mesh for 2 km urban block.

Here we consider 30 m $\times$ 30 m building in 2 km mesh.

1. Dirichlet Condition on even boundary

$$u_0 = \begin{cases} 0 \quad \textbf{when} \quad 570 \leq x \leq 570.030, \quad 5936 \leq y \leq 5936.030 \quad \textbf{Building} \\ \\ 1 \quad \textbf{else} \end{cases}$$

2. Initial condition at t = 0.

$$u(x,y) = \begin{cases} 0 \quad \textbf{when} \quad 570 \leq x \leq 570.030, \quad 5936 \leq y \leq 5936.030 \quad \textbf{Building} \\ \\ 1 \quad \textbf{else} \end{cases}$$

3. Neumann Condition on right hand side

$$g = 0$$

4. Diffusion Coefficient

$$a(x,y) = \begin{cases} 10e15 & \textbf{when} \quad 570 \leq x \leq 570.030, \quad 5936 \leq y \leq 5936.030 \quad \textbf{Building} \\ 0.001 * y & \textbf{else} \end{cases}$$

5. Velocity

$$c(x,y) = \begin{cases} 0 & \textbf{when} \quad 570 \leq x \leq 570.030, \quad 5936 \leq y \leq 5936.030 \quad \textbf{Building} \\ 4 & else \end{cases}$$

6. Right hand side

$$f = 0$$

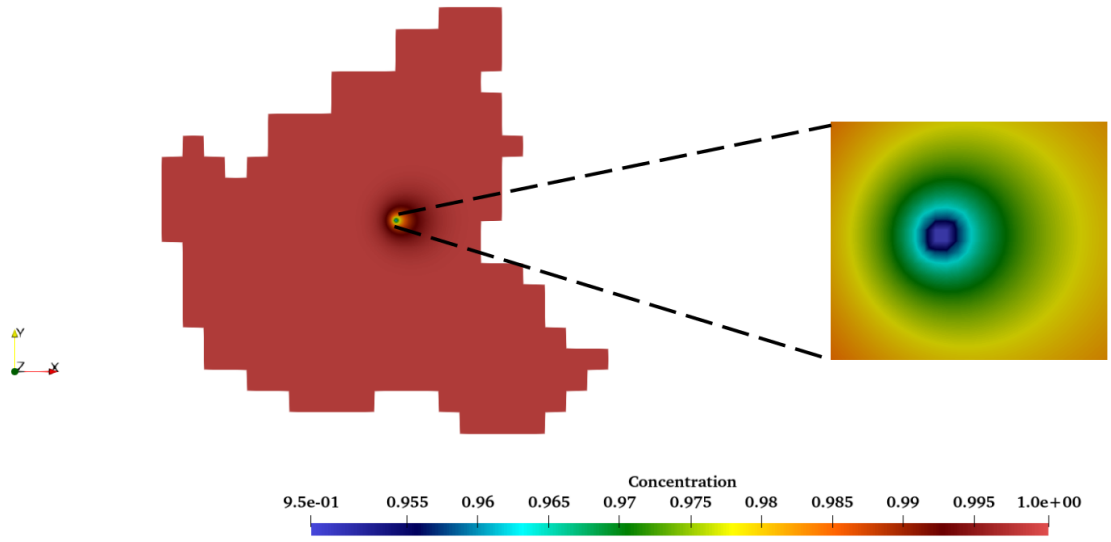


Figure 8.35: Result of Hamburg Mesh for 30 m building for High resolution FEM with the building.

Figure (8.35) shows a high resolution simulation with the building zoomed in. The building is 30 m highis in blue.

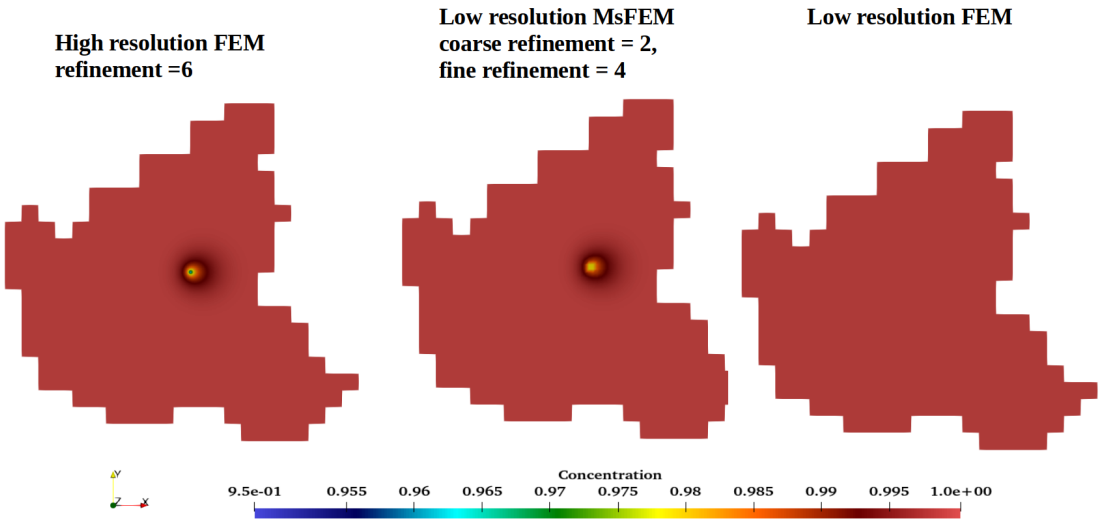


Figure 8.36: Result of Hamburg Mesh with 30 m high building.

It is seen from Figure (8.36) that canopy is well represented in high resolution FEM and low resolution MsFEM shows some agreement as yellow building structure with the results of high resolution FEM. Low resolution FEM does not represent canopy structure.

8.6.1 Hamburg mesh : Wall time distribution

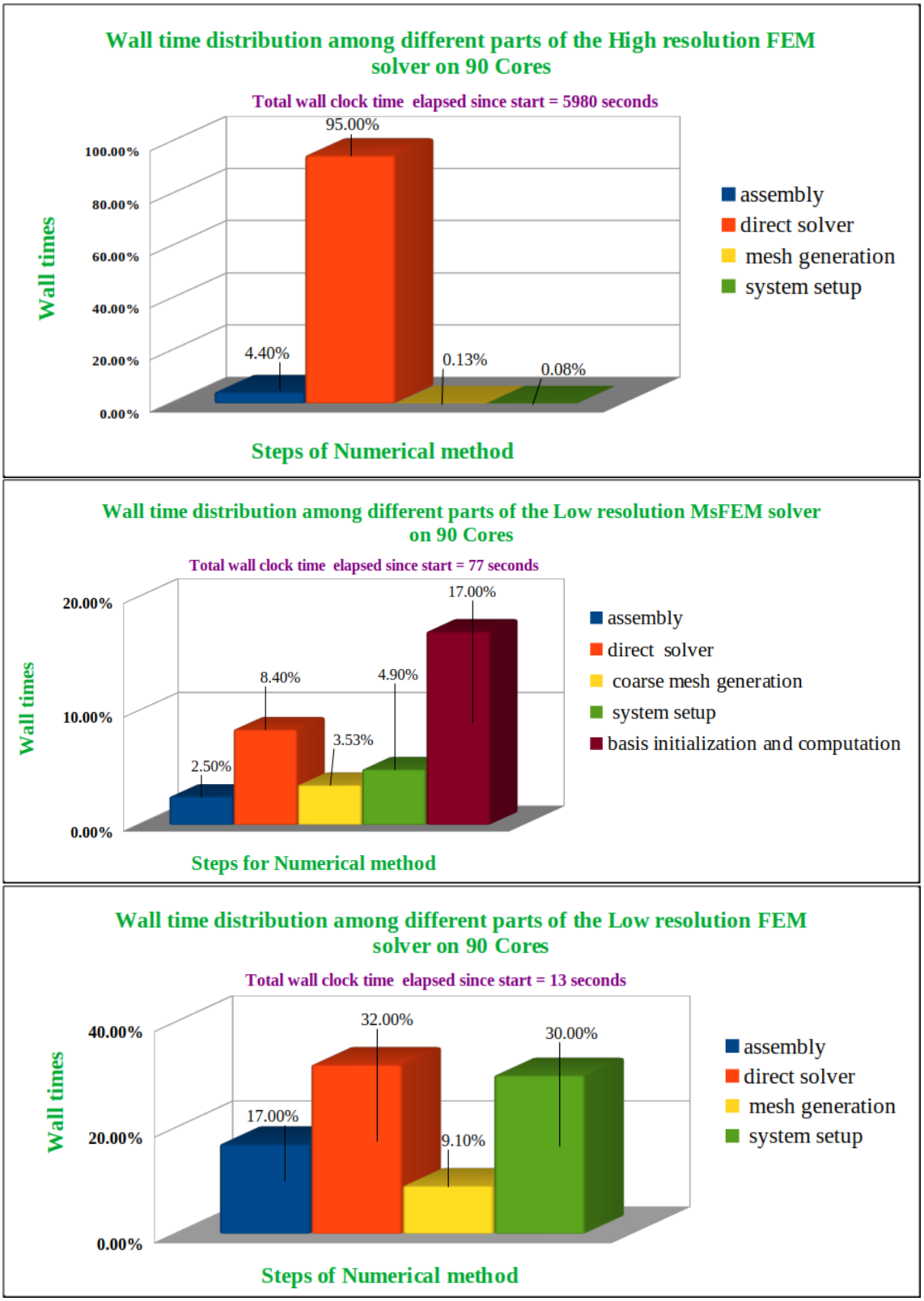


Figure 8.37: Wall time distribution for Hamburg Mesh.

Figure (8.37) presents the computation wall time distribution. It is found that the direct solver takes the maximum time for low- resolution FEM, and for high- resolution FEM. For low resolution MsFEM basis initialization and computation takes maximum times. The total wall clock time elapsed since start is 5980 seconds for high resolution FEM, 77 seconds for low resolution MsFEM and 13 seconds for FEM.

In all cases, high resolution FEM provides accurate results, but at high computation costs. In real-world climate or weather models, low precision MsFEM can be an appropriate choice since it is computationally less expensive but can show subgrid scale effects like canopy on larger scales. Chapter 8 concludes with the application part for readers and Chapter 9 begins with conclusion and future work.

9 Conclusion and Future Work

"I like crossing the imaginary boundaries people set up between different fields - it's very refreshing."

- Maryam Mirzakhani

Numerical simulations of climate models can be computationally expensive if they run at high resolution. Because of the physical component and grid resolution limitations, applying numerical methods to climate models is challenging. Developing a method that considers the subgrid scale while being computationally efficient is essential. We solve the Poisson's equation with standard FEM and MsFEM using deal.II 9.4.0 version. This study provides an overview of the upscaling approach with multiscale finite elements. A gradient-based adaptive mesh refinement method is accurate; however, with every new step, it becomes more computationally expensive, see Appendix Section III for preliminary results. For high-resolution FEM, these methods are impractical to implement in climate models. For representative subgrid scale features, we find accurate results with MsFEM because it uses a modified basis function that can be run in parallel. Therefore, it is computationally inexpensive compared with the high-resolution finite element method and adaptive mesh refinement. The Github code is now available

`https://github.com/heena008/Diffusion-Equation-with-MsFEM.io` and `https://github.com/heena008/Advection-Diffusion_MsFEM.io`

In conclusion, the multiscale finite element method gives accurate results when considering subgrid processes compared with the standard finite element method for advection-diffusion equations. If advection is the dominant in the problem, the algorithm is modified by semi-Lagrangian trace back. You can see that this process is computationally intensive because the basis are calculated in parallel. In the present study a lot of code development and parallel architecture for high performance computing is carried out with C++ based dealii library.

9.1 Research Contribution

9.1.1 Software development

1. The code has been developed from scratch in C++ library dealii.
2. The code template can therefore run for 2D and 3D cases easily.

3. It is fully parallelized and can work on a supercomputer.
4. The implicit, Crank-Nicolson and explicit schemes are implemented.
5. The High Order Finite Element Method is accurate but expensive.
6. Low resolution Finite Element Method does not show accurate result and Multiscale Finite Element Method often gives better results than Low resolution Finite Element Method.
7. Semi-Lagrangian trace back is computationally intensive, but Konrad Simon implemented a new feature to find points owned by the processor that was not previously developed, that solved the bottleneck of MPI computing of semi-Lagrangian trace back mesh.
8. Code works well with 16 millions of degrees of freedom, it can also work with billions. The whole architecture can be extended to global circulation models.
9. Canopy parameterization is key contribution of my work
10. It is open source code and can be extended to solve a variety of real world problems.

9.2 Key Findings

9.2.1 Canopy Modeling

The new achievement in this study from previous work done by [38, 39, 40] for an oscillating diffusion traveling with the flow is that it can also be used for stationary obstacles of subgrid scale. We apply this method to urban climate simulations. The combination of different parallel processes will decrease simulation time, making it easy to include subgrid processes such as canopies in the diffusion coefficient. These show the impact of canopies on a large scale.

Regional climate models usually use downscaling techniques that use external forcing at finer scales. Our approach, on the other hand, upscales fine scales and adds features to large scales as well. In this way, fine scale features can be included on a large scale. In terms of urban climate simulation, this algorithm is unique. Downscaling methods are computationally expensive.

The canopy exists at a subgrid scale in climate models but plays an influential role in the modulation of local—but large-scale climate. Canopy modeling has become increasingly significant as global temperatures rise above pre-existing limits. While numerous studies have been conducted in this area, they incur high computational costs. To account for canopy effects, models include canopy parametrization as a source term in flow or

transport equations. Researchers typically add a source or sink term to the equation to represent canopies — i.e., buildings are considered solid and surrounded by the fluid domain in the mesh, based on the different boundary conditions applied to the fluid and solid domains. They then solve the system using terrain-following coordinates or the immersed boundary method. Mesoscale models also account for external forcing of data from climate simulations. All these methods are computationally expensive when the models are run at high spatial resolution.

Our approach simplifies this complexity by representing the obstacles in the canopy with a (large) diffusion coefficient and employing MsFEM discretization for the subgrid-scale structure of the canopy layer. We tested our approach by conducting wind tunnel experiments with simple one-building setups and comparing the data with numerical solutions. While our experiments are idealized cases that demonstrate the general feasibility of this new method, a generalization promises to enable new ways to solve the so-called upscaling problem in which small-scale features—which are not representable numerically—must be represented in large-scale dynamics. Furthermore, the Hamburg mesh test case shows promising results for single building effects. Thus, we established the potential for future implementation in global climate models.

9.3 Future Work

In the future, work

- The adaptive feature can be implemented at a local scale by parameterizing the canopy so that the near field flow (the shape of flow in the region immediately next to an object) can be captured. See the Appendix Section III.

- Further Reduce Order Model techniques can be used to implement effective modes represented by a Full Order model [6].

- The code can be run and tested in supercomputers and also transformed to Domain Specific Language to couple with large models.

- The entire city of Hamburg is estimated to be underwater due to sea ice melt in 80 years. Due to my fascination with Antarctica, I want to implement this framework for sea ice modeling refer Figure (9.1) and (9.2).

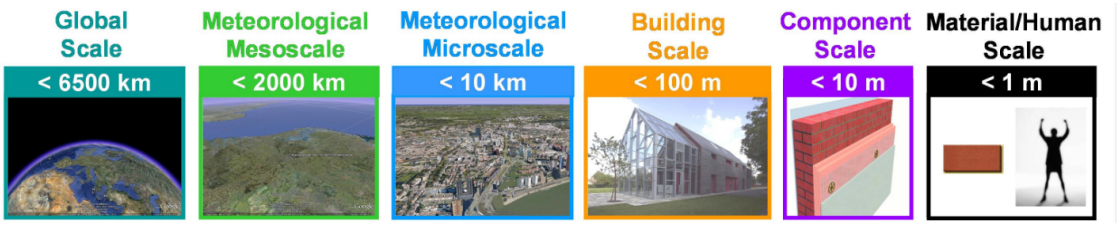


Figure 9.1: Scales in canopy [11].

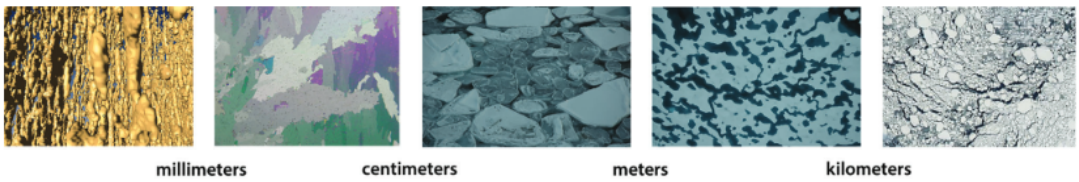


Figure 9.2: Arctic sea ice at 1 cm, 5 cm, 5 m, 100 m, 100 km [16].

- Also, I want to add turbulence to the model, to determine how it upscales on a large scale.

10 Appendix

I Mathematical Concept

I.1 Well-posedness (Hadamard 1923)

The concept of well-posedness is general and simple.

Definition 12. *Let $A : X \leftarrow Y$ be a mapping and X, Y topological spaces. Then, the problem*

$$Ax = y \tag{10.1}$$

is well posed if

- *for each $y \in Y$, Problem (12) has a solution x,*
- *the solution x is unique,*
- *the solution x depends continuously on the problem data.*

This is the first condition that is immediately evident. In many physical processes, there are no unique solutions to the second condition, which is also apparent, but difficult to achieve. Lastly, if the input data (right hand side, boundary values, initial conditions) vary just a little bit, so should the unique solution.

I.2 Create mathematical models

Consider the second order differential equation as follows:

$$A\frac{\partial^2 u}{\partial x^2} + B\frac{\partial^2 u}{\partial x \partial y} + C\frac{\partial^2 u}{\partial y^2} + D\frac{\partial u}{\partial x} + E\frac{\partial u}{\partial y} + Fu = G \tag{10.2}$$

where A, B, C, and F are functions of x and y. The above equation (10.2) can be classified based on this discriminant as follows according to the value of $L = B^2 - AC$.

$$\begin{aligned} L = 0 &\Leftrightarrow parabolic \\ L > 0 &\Leftrightarrow hyperbolic \\ L < 0 &\Leftrightarrow elliptic \end{aligned} \tag{10.3}$$

The heat equation (10.4) given below is an example of a parabolic equation.

$$\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2} = 0 \tag{10.4}$$

The wave equation (10.5) is a hyperbolic equation.

$$\frac{\partial^2 u}{\partial t^2} = c\Big(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\Big) \tag{10.5}$$

where c is a fixed non-negative real coefficient. The Laplace equation (10.6) is elliptic equation.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{10.6}$$

Types of problems govern by differential equation

1. **Initial Value Problems (IVP) :** An initial value problem is one in which the dependent variables and their derivatives are defined at the same time as the independent variables and at the same value at the initial time $t = 0$.

   In this case, $u(x,0)$ and $u_t(x,0)$ are the initial values of $u(x,0)$ at time $t = 0$.

   $$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}, \quad u(x,t_0) = f(x)$$

2. **Boundary Value Problems (BVP) :** Dependent variables and their derivatives are defined at the extremes of independent variables in boundary value problems. To solve a differential equation, boundary conditions are applied to boundary points.

Types of Boundary Conditions

There are three kinds of Boundary Conditions

1. **Dirichlet Boundary Conditions :** In the domain $(0,L)$, consider a stationary diffusion equation with Dirichlet boundary conditions.

   $$\frac{\partial^2 u}{\partial x^2} = f(x), \quad u(0) = u(L) = 0$$

   PDE specifies the dependent variables at different points in the domain based on the value of the given function u.

2. **Neumann Boundary Conditions :** As a rule, Neumann boundary conditions are

derivatives of solution values on the boundary of a given domain.

In the domain $(0, L)$, consider the following heat equation in one dimension.

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$$
$$\frac{\partial u}{\partial n} = 0$$
$$u(x,0) = u_0(x), x \in \mathbb{R}$$

3. **Mixed Boundary Conditions :**

   A mixed boundary condition is a linear combination of Dirichlet and Neumann boundary conditions.

   Here is the stationary heat equation in one dimension in domain $(0, L)$.

$$\frac{\partial^2 u}{\partial x^2} = f(x), \quad 0 < x < 1$$
$$u(0) = 0, \quad c(1)\frac{\partial u}{\partial x} = g_1$$

II What is finite element method?

This chapter introduces Galerkin methods for approximating partial differential equations (PDEs).

II.1 Introduction

Computing Finite Element Methods (FEMs) uses variational methods. FEM provides a convenient method for solving complex mathematical models in complex geometric domains. FEM can solve a wide range of PDE problems. Domain of interest is defined, and boundary conditions are specified.

FEM discretizes the domain of interest, where the PDE is defined, to obtain an approximate solution. Basis functions are combined linearly within each subdomain to accomplish this. As a result of repositioning the finite elements into their original positions, the assembly of subdomains results in a discrete set of equations equivalent to the original mathematical problem.

III Norms and Functional Spaces

Let us define some Functional Spaces, norms and inner products.

Vector space

Definition 13. *Vector space (over a field $F \in \mathbb{R}$). A vector space is a set V equipped with,*

- *addition $+ : V \times V \to V$*
- *multiplication $\cdot : F \times V \to V$*

Where + and $\cdot$ satisfy the following conditions

1. *+ is commutative: $v + u = u + v$*
2. *+ is associative: $u + (v + w) = (u + v) + w$*
3. *additive identity: $\exists \quad 0 \in V$ such that $v + 0 = 0 + v = v$*
4. *additive inverse: $\exists \quad -v \in V$ such that $v + (-v) = (-v) + v = 0$*
5. *$\cdot$ is distributive: $c \cdot (u + v) = c \cdot u + c \cdot v$*
6. *$\cdot$ is distributive: $(c + d) \cdot v = c \cdot v + d \cdot v$*
7. *$\cdot$ is associative: $c \cdot (d \cdot v) = (c \cdot d) \cdot v$*
8. *multiplicative identity: $1 \cdot v = v$*

or all $u, v, w \in V \quad and c, d \in \mathbb{R}$

Examples:

1. $V = \mathbb{R}^3$
2. $V = \mathbb{R}^N, [x_1, ..., x_N] + [y_1, ..., y_N] = [x_1 + y_1, ..., x_N + y_N] \quad and \quad \alpha[x_1, ..., x_N] = [\alpha x_1, ..., \alpha x_N]$
3. $V = \{v : [0,1] \to \mathbb{R} \quad | \quad v \quad is \quad continuous\}$

Definition 14. *Inner product space (over a field $F = \mathbb{R}$).*
An inner product space is a vector space with an inner product, a map,

$$\langle ., . \rangle \quad V \times V \to F,$$

satisfying the following conditions:

1. $$\langle v, w \rangle = \overline{\langle w, v \rangle} \quad \forall v, w \in V \quad (conjugate \quad symmetry)$$

2.
$$\left.\begin{array}{l} \langle \alpha v, w\rangle = \alpha\langle v, w\rangle \quad \forall v, w \in V \quad and \quad \alpha \in F \\ \langle u+v, w\rangle = \langle u, v\rangle + \langle v, w\rangle \quad \forall u, v, w \in V \end{array}\right\} \ linearity$$

3.
$$\langle v, v\rangle \geq 0 \quad with \quad \langle v, v\rangle = 0 \quad iff \quad v = 0 \quad (positive \quad definite)$$

Examples:

1. $V = \mathbb{R}^N \quad \langle v, w\rangle = \sum_{i=1}^{N} v_i w_i$
2. $V = l^2 \quad \langle v, w\rangle = \sum_{i=1}^{N} v_i w_i$
3. $V = C^\infty(\Omega) \quad \langle v, w\rangle = \int_\Omega vw dx$

l^2 is the space of all sequences (or infinite vectors) that satisfy $\sum_i v_i^2 < \infty$.

Definition 15. *Orthogonality*
Let V be an inner product space. Two vectors u, v ∈ V are said to be orthogonal if

$$\langle v, w\rangle = 0.$$

Examples:

1. $V = \mathbb{R}^3 \quad v = (1,2,3) \quad w = (3,0,-1)$
2. $V = \mathbb{P}^2 \quad u = 1, \quad v = x, \quad w = 0.5(3x^2 - 1)$ (Legendre polynomials)

Definition 16. *Normed vector space (over a field F)*
A normed vector space is a vector space with a norm, a map,

$$||.|| : V \to \mathbb{R},$$

satisfying the following conditions:

1. $||\alpha v|| = |\alpha||v||, \quad \forall v \in V \quad and \quad \forall \alpha \in F \quad (Positive \quad homogeneity)$
2. $||u+v|| \leq ||u|| + ||v||, \quad \forall u, v \in V \quad (triangle \quad inequality)$
3. $||v|| = 0 \Rightarrow v = 0 \quad (point \quad separation)$

Examples:

1. $V = \mathbb{R}^N \quad ||v||_p = (\sum_{i=1}^{N} v_i^p)^{\frac{1}{p}}, \quad 1 \leq p < \infty$
2. $V = C^\infty(\Omega) \quad ||v||_p = (\int_\Omega v^p dx)^{\frac{1}{p}}, \quad 1 \leq p < \infty$

Definition 17. *Cauchy sequence (on normed space)*
Let V be a normed space. A sequence $\{v_i\}_{i=1}^{\infty} \subset V$ is a Cauchy sequence if for all $\varepsilon > 0$

there exists a number $N > 0$, such that

$$||v_m - v_n|| < \varepsilon \quad \forall m,n > N$$

Examples:

1. The 2-norm: $||v||_2 = \sqrt{\sum_{i=0}^{n} v_i^2}$
2. The 1-norm: $||v||_1 = \sum_{i=0}^{n} |v_i|$
3. The inf-norm: $||v||_\infty = max|v_i|$
4. $V = \mathbb{R} \quad ||v|| = |v|, \quad v_n = \frac{1}{n}$
5. $V = C([0,1]), \quad ||v|| = ||v||_\infty, \quad v_n(x) = \sum_{i=0}^{n} \frac{x^i}{i!}$

Definition 18. *Completeness*

A (metric) space, V, is complete if all Cauchy sequences converge to a point in V.

Definition 19. *Banach space*

A Banach space is a complete normed vector space.

Definition 20. *Hilbert space*

A Hilbert space is a complete normed inner product space.

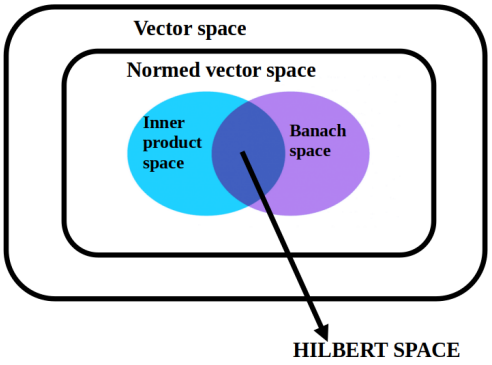


Figure 10.1: Venn diagram of different spaces.

Definition 21. *(Continuous) Dual space*

Let V be a normed vector space. The dual space V' (sometimes denoted V^*) is the space of all continuous, linear functionals on V:

$$V' = \{l : V \to R| \quad ||l|| < \infty\} \quad where, \quad ||l|| = \sup_{||v|| \leq 1} |l(v)|$$

Theorem 3. *Cauchy–Schwartz inequality*
Let V be an inner product space. Then

$$|\langle v,w\rangle| \leq ||v|| \cdot ||w|| \quad \forall v,w \in V$$

Theorem 4. *Banach fixed-point theorem*
Let V be a Banach space and let

$$T : V \to V$$

be a continuous mapping on V, that is,

$$\exists \quad M < 1 : ||T(v) - T(w)|| \leq M||v-w|| \quad \forall v,w \in V.$$

Then $\exists!$ $\quad \overline{v} \in V, such \quad that \quad T\overline{v} = \overline{v}$

Examples:

1. $V = R \quad Tv = \frac{v}{2}, \overline{v} = 0$
2. $V = R^+ \quad Tv = \frac{v+2/v}{2}, \quad \overline{v} = \sqrt{2}$

Theorem 5. *Riesz representation theorem*
Let H be a Hilbert space and let H' denote its dual space. Then for all $l \in H'$ there exists a unique element $l \in H$, such that

$$l(v) = \langle \hat{v}, v \rangle \quad \forall v \in H$$

Theorem 6. *Gauss theorem*
Let $\Omega \subset R^d$ be bounded and sufficiently regular with outward normal n. Then

$$\int_{\Omega} u\nabla \cdot v \, dx = \int_{\partial\Omega} v \cdot n \, dS$$

for a differentiable vector field $v : \Omega \to R^d$.
The following integration by parts formula can be seen as an expression of the product rule

$$\nabla(uv) = u \cdot \nabla v + v \cdot \nabla u$$

Theorem 7. *Integration by parts*
Let $\Omega \subset R^d$ be bounded and sufficiently regular with outward normal n. Then the

integration by parts formula holds

$$\int_\Omega \nabla u \cdot v \, dx + \int_\Omega u \nabla \cdot v \, dx = \int_{\partial\Omega} uv \cdot n \, dS$$

for a differentiable vector field $u : \Omega \to R$ and $v : \Omega \to R^d$.

Definition 22. *(Weak derivative). Suppose that*

$$\int_\Omega u \partial_{x_k} v \, dx = -\int_\Omega gv \, dx \quad \text{for all test functions } v$$

holds for some function $g : \Omega \to R$. Then g is called the k-th weak (partial) derivative of u. The integration by parts formula and its variations enables us to introduce a generalization of classical derivatives.
We say that a vector field $g : \Omega \to R^d$ is the weak gradient of u if

$$\int_\Omega u \nabla v \, dx = -\int_\Omega gv \, dx \quad \text{for all vector valued test functions } v$$

In this case we also just write $\nabla u = g$.
Similarly, we say that a function $g : \Omega \to R$ is the weak divergence of a vector field u if

$$\int_\Omega u \nabla v \, dx = -\int_\Omega gv \, dx \quad \text{for all scalar test functions } v$$

Also, in this case, we simply write $\nabla u = g$.

Classical and weak derivatives are the same for smooth functions. Weak derivatives can often be found even for functions that are not classically differentiable.

Sobolev spaces

Definition 23. *The Lebesgue space $L^2(\Omega)$ space*
Let Ω be an open subset of R^n, with piecewise smooth boundary, then $L^2(\Omega)$ is defined by

$$L^2(\Omega) = \left\{ u : \Omega \to R | \int_\Omega |u(x)|^2 \, dx < \infty \right\}$$

It is a Hilbert space with scalar product

$$\langle u,v\rangle_{L^2} = \int_\Omega uv\,dx$$

Examples:

1. $v(x) = \frac{1}{\sqrt{x}} \quad \Omega = (0,1), \quad v \notin L^2(\Omega)$
2. $v(x) = \frac{1}{x^{\frac{1}{4}}} \quad \Omega = (0,1), \quad v \in L^2(\Omega)$

Theorem 8. *L^2 with $\langle v,w\rangle = \int_\Omega vw\,dx$ is a Hilbert space.*

Definition 24. *Weak derivative (first order)*
Let $v \in L^2(\Omega)$. The weak derivative of v (if it exists), is a function $\frac{\partial v}{\partial x_i} \in L^2(\Omega)$ satisfying,

$$\int_\Omega \frac{\partial v}{\partial x_i}\varphi\,dx = -\int_\Omega v\frac{\partial \varphi}{\partial x_i}\,dx, \quad \forall \varphi \in C_0^\infty(\Omega)$$

Definition 25. *Weak derivative (general order)*
Let $v \in L^2(\Omega)$. The weak derivative of v (if it exists), is a function $\partial^\alpha v \in L^2(\Omega)$ satisfying,

$$\int_\Omega \partial^\alpha v\varphi\,dx = (-1)^{|\alpha|}\int_\Omega v\partial^\alpha v\,dx, \quad \forall \varphi \in C_0^\infty(\Omega)$$

where $\partial^\alpha \varphi = \frac{\partial^{|\alpha|}}{\partial^{\alpha_1}x_1\partial^{\alpha_2}x_2\ldots\partial^{\alpha_n}x_n}$

Lemma 3. *A weak derivative (if it exist), is unique.*

Lemma 4. *A (strong) derivative (if it exist), is a weak derivative.*

Theorem 9. *Poincaré inequality*
Let $v \in H_0^1(\Omega)$. Then ,

$$||v||_{L^2(\Omega)} \leq C|v|_{H^1(\Omega)'}$$

where C depends only on Ω.

Convergence in FEM

Theorem: For a sequence of "uniformly regular" meshes in 2-d or 3-d, there exists a constant C, independent of h and u such that

$$||u-u_h||_{H^1(\Omega)} \leq Ch||u||_{H^2(\Omega)}$$

(roughly, $||u||_{H^2(\Omega)} \approx ||\nabla\nabla u||_{L^2(\Omega)}$)

Remark : In the case of oscillating coefficients $A(\frac{x}{\varepsilon})$ the Q1 Finite Element method can converge only if $h << \varepsilon$

Adaptive Mesh refinement

An adaptive mesh refinement technique begins by imposing a coarse grid over the entire problem domain. The grid defines the cell spacing, or resolution, of computations in the domain. In order to solve the PDE, more grid points are introduced as the grid spacing becomes finer.

An error estimation algorithm estimates the interpolation error of a solution, resulting in the Kelly indicator.

$$\eta_K^2 = \sum_{F \in \partial K} h_K^{1/2} \int_{\partial F} \left[\left[a \frac{\partial u_h}{\partial n} \right] \right]^2$$

This is the error estimator for cell K. There is a jump in the function in square brackets at the face, and there is a factor called cF discussed below. Kelly et al. derived this form of the interface terms for their error estimator in the paper referenced above. As a result, the overall error estimate can be calculated as follows:

$$\eta^2 = \sum_K \eta_K^2$$

Adaptative mesh refinement is an adhoc algorithm. There is no global refinement that is mark those cells that have large errors for refinement, mark those that have particularly small errors for coarsening, and leave the rest alone. The eight here is taken as a heustric constant.

Algorithm 7 Algorithm for Adaptive Mesh Refinement Method.

1. for (unsigned int cycle = 0; cycle < 8; ++cycle)
2. if (cycle == 0)
   Setup mesh (coarse).
   refine grid(1);
   else
   refine grid()
   Kelly Error Estimator checker
3. - Setup system and constraints.
   - Assemble system
   - Solve for u^{n+1}
   - Set n=n+1

III.1 Test 1

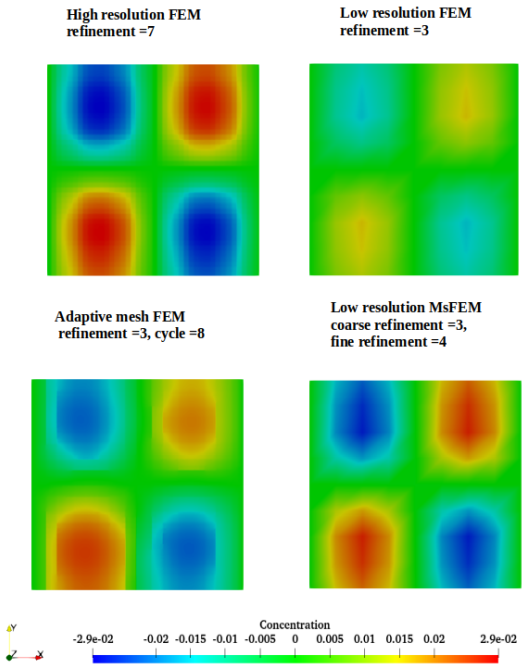


Figure 10.2: Test 1 Solution of High resolution FEM, Low resolution FEM, Low resolution MsFEM and Adaptive mesh refinement FEM.

Test case 1: Error Table

Here the error is calculated with the difference in the exact solution and approximate solution.

Error Analysis			
Simulation Type	L^2 Error	H^1 Error	L^∞ Error
Low resolution FEM	0.0080	0.1011	0.0146
Adaptive Mesh FEM	0.00304	0.0208	0.01156
Low resolution MsFEM	0.0019	0.0555	0.0041

Table 10.1: Test 1 Error in simulation of Low resolution FEM, Low resolution MsFEM and Adaptive mesh refinement FEM.

III.2 Test 2

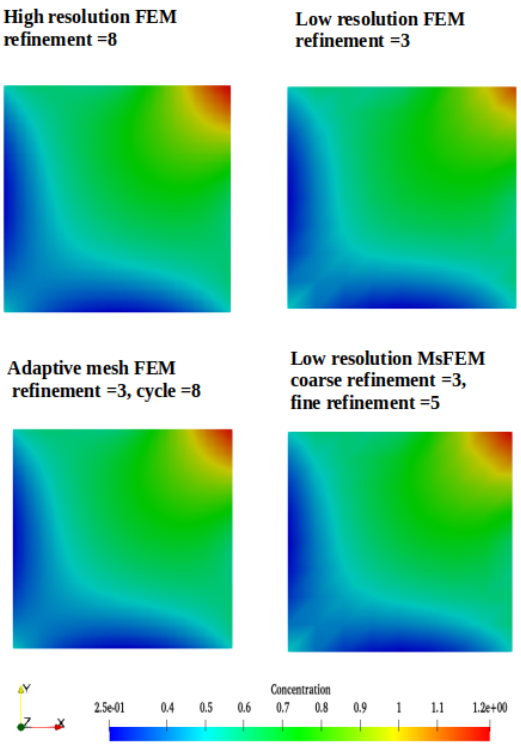


Figure 10.3: Solution of High resolution FEM, Low resolution FEM, Low resolution MsFEM and Adaptive mesh refinement FEM.

Test case 2: Error Table

Here the error is calculated with the difference in the reference solution which is high resolution FEM.

Error Analysis			
Simulation Type	L^2 Error	H^1 Error	L^∞ Error
Low resolution FEM	0.0159	0.073	0.14
Adaptive Mesh FEM	0.0097	0.019	0.027
Low resolution MsFEM	0.0025	0.034	0.056

Table 10.2: Test 2 Error in simulation of Low resolution FEM, Low resolution MsFEM and Adaptive mesh refinement FEM.

IV Glossary for C++ terms

The content is from [1] and [2]

1. **alias**:

Terms	**Description**
Type alias	Type alias is a name that refers to a previously defined type.
Alias template	Alias template is a name that refers to a family of types.

2. **Class**: A class is a user-defined data type with its own data members and member functions, which can be accessed and used by creating an instance of the class. The data members are the variables and the member functions are the functions used to manipulate these variables. They together define the properties and behavior of objects in a class.

   An **Object** belongs to a Class. An object is created when a class is instantiated (i.e. a class is defined).

   C++ classes are defined by class followed by their names. Class bodies are enclosed in curly brackets and terminated by semicolons.

```
class class_name {
  access_specifier_1:
```

```
    member1;
    access_specifier_2:
    member2;
    ...
} object_names;
```

Listing 59: Class.

An optional list of names for objects of a class can be found in object_names, where class_name is the class's identifier. Declaration bodies can contain data or function declarations, as well as access specifiers.

Declaring Objects: An object's specification is the only thing defined when a class is defined; memory and storage are not allocated. It is necessary to create objects to access and use the class data.

Constructor : Constructors are special member functions of classes, and they share the same name as their classes. The constructor is called whenever an object of the class is created by the compiler; it allocates memory and initializes class data members with default values or values passed by the user.

Destructors : Destructors are member functions that are called instantly when an object is destroyed. Compilers call the destructor automatically when an object goes out of scope, for example, when a function ends.

Accessing member data and member functions:

Objects can be accessed using the dot ('.') operator with their data members and member functions. To access a member function with the name printName() on an object named obj, you will need obj.printName().

- Class members who are private are only accessible to other class members (or their "friends").
- A protected member can be accessed by members of the same class (or their "friends"), but also by members of their derived classes.
- Finally, public members can access the object from anywhere.

Member Functions in Classes: There are 2 ways to define a member function:

- Inside class definition
- Outside class definition

We must use the scope resolution :: operator along with the class name and function name to define a member function outside of the class definition.

A dot (.) can be inserted between the name of the object and the name of any public member as if they were normal variables or functions. For private members, use the

scope resolution operator.

3. **Data Types**: When variables are declared, they use data-type to restrict the type of data they can store. This means that data types tell variables what type of data they can store. C++ allocates some memory for variables based on the data-type with which they are declared. There is a different amount of memory needed for each type of data.

   Data types in C++ is mainly divided into three types:

   a) **Primitive Data Types:** These types of data are built-in or predefined and can be used directly to declare variables. For example, int, char, float, bool, etc. The following primitive data types are available in C++:

      - Integer: The keyword int is used for integer data types. The range of integers typically ranges from -2147483648 to 2147483647 and requires 4 bytes of memory space.
      - Character: The character data type is used to store characters. The character data type is referred to as char. Characters typically take up one byte of memory and range from -128 to 127 or 0 to 255.
      - Boolean: Boolean data types store boolean or logical values. It is possible to store either true or false in a boolean variable. The keyword used for boolean data types is bool.
      - Floating Point: A floating point value is a value that can be stored as a single precision value or as a decimal value. Float is the keyword used for floating point data types. A float variable typically requires 4 bytes of memory.
      - Double Floating Point: This data type stores floating point or decimal values with double precision. This data type uses the keyword double as its keyword. Memory space for double variables is typically 8 bytes.
      - Valueless or Void: A void means that it has no value. The void datatype represents an entity without any value. For functions that return no value, void data types are us.
      - Wide Character: The wide character data type is also a character data type, but it has a larger size than the normal 8-bit data type. This type is represented by the wchar_*t* type. There are usually two or four bytes in it.

   Datatype Modifiers: A datatype modifier, as its name suggests, modifies the

amount of data a built-in data type can hold. C++ provides the following data type modifiers:

Data Type	Size in Bytes	Range
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
unsigned int	4	0 to 4,294,967,295
int	4	-2,147,483,648 to 2,147,483,647
long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295
long long int	8	$-(2^63)to(2^63)-1$
unsigned long long int	8	0 to 18,446,744,073,709,551,615
unsigned char	1	0 to 255
float	4	
double	8	
long double	12	
wchar_t	2 or 4	1 wide character

Note : Above values may vary from compiler to compiler. In above example, we have considered GCC 64 bit.

b) **Derived Data Types:** Derived data types are derived from primitive or built-in data types. There are four types of these:

- Function
- Array
- Pointer
- Reference

c) **Abstract or User-Defined Data Types:** Data types are defined by the user. In C++, for example, defining a class. The following user-defined datatypes are available in C++:

- Class
- Structure
- Union
- Enumeration
- Typedef defined DataType

4. **Float:** Floating point variables hold real numbers, such as 4320.0, -3.33, or 0.01226. There can be a variable number of digits before and after the decimal

point due to the floating part of the name floating point.

Terms	Description
float	single precision floating point type. Usually IEEE-754 32 bit floating point type.
double	double precision floating point type. Usually IEEE-754 64 bit floating point type.
long double	extended precision floating point type. Does not necessarily map to types mandated by IEEE-754. Usually 80-bit x87 floating point type on x86 and x86-64 architectures.

5. **Namespaces:** We can give namespace scope to named entities that otherwise would have global scope. A program can be organized by name into various logical scopes.
   Note: using and using namespaces are valid only within the block in which they are stated, or in the entire source code file if they are applied directly in global scope.

Terms	Description
using	The keyword using introduces a name into the current declarative region (such as a block), thus avoiding the need to qualify the name.
namespace new_name = current_name;	Existing namespaces can be aliased with new names.
std namespace	All the entities (variables, types, constants, and functions) of the standard C++ library are declared within the std namespace.

6. **Preprocessor:** The preprocessor directive is a line of code that precedes a hash sign (#). Preprocessor directives are not program statements. Preprocessors examine code before compilation and resolve all directives before regular statements generate any code.Pre] Types of directives and their functions.

Terms	Description
#define	When the preprocessor encounters this directive, it replaces any occurrence of identifier in the rest of the code by replacement.
#undef	undefined with the preprocessor directive.
##	The operator ## concatenates two arguments leaving no blank spaces between them
#ifdef	allows a section of a program to be compiled only if the macro that is specified as the parameter has been defined, no matter which its value is.
#ifndef	compiled if the specified identifier has not been previously defined.
#if, #else and #elif	directives serve to specify some condition to be met in order for the portion of code they surround to be compiled.
#error	This directive aborts the compilation process when it is found, generating a compilation error that can be specified as its parameter.
#include	When the preprocessor finds an #include directive it replaces it by the entire content of the specified header or file.

7. **Template:** A function template is a special function that operates with generic types. Our function template can be adapted to more than one type or class without having to repeat the entire code.

Terms	Description
Class templates	A class template defines a family of classes.
Template specialization	If we want to define a different implementation for a template when a specific type is passed as template parameter, we can declare a specialization of that template.

8. **Standard Template Library (STL)** STL provides common programming data structures and functions, such as arrays, stacks, and lists, as a set of C++ template classes.

   The STL includes the following components:

   a) Containers : The STL provides a range of containers, such as vector, list, map, set, and stack, for storing and manipulating data.

   b) Algorithms: STL provides algorithms for manipulating data in containers,

such as find, sort, and binary_search.

c) Iterators: An iterator is an object that traverses the elements of a container. For different types of containers, the STL provides iterators such as forward_iterator, bidirectional_iterator, and random_access_iterator.

d) Function Objects: Algorithms can use function objects, also called functors, as arguments. An algorithm can be customized by passing it a function.

e) Adapters: Adapters modify the behavior of other STL components. A container can be reversed by using the reverse_iterator adapter.

9. **Structure** Structs, short for C++ Structures, are user-defined data types available in C++. It allows users to combine data items of (possibly) different types under single name.

```
struct structure_name
{
  member data_type1 member_name1 ;
  .
  .
  .
  member data_typeN member_nameN;
} ;
```

Listing 60: Structure.

10. **Virtual Function in C++:** The term "virtual function" refers to a member function that is declared within a base class and is overridden by derived classes. Using a pointer or reference to a base class object, you can call a virtual function for that object and execute the version of the function that was implemented in the derived class.

- In virtual functions, the correct function is called for an object, regardless of the type of reference (or pointer) used to call it.
- Runtime polymorphism is achieved by using them.
- Virtual functions are declared in base classes.
- Run-time resolving of function calls takes place.

Rules for Virtual Functions

a) A virtual function cannot be static or a friend function of another class.

b) Run-time polymorphism can be achieved by accessing virtual functions through pointers or references of base class types.

c) Base and derived classes should share the same prototype for virtual functions.

d) In derived classes, they are overridden by the base class. In that case, the base class version of the function is used instead of overriding (or redefining) the virtual function.

e) Virtual destructors are allowed, but virtual constructors are not.

V Implementation of advection diffusion equation in deal.II

```
#ifndef INCLUDE_ADVECTIONDIFFUSION_PROBLEM_HPP_
#define INCLUDE_ADVECTIONDIFFUSION_PROBLEM_HPP_

#include <deal.II/base/quadrature_lib.h>
#include <deal.II/base/function.h>
#include <deal.II/base/tensor_function.h>
#include <deal.II/base/logstream.h>

#include <deal.II/lac/vector.h>
#include <deal.II/lac/full_matrix.h>
#include <deal.II/lac/sparse_matrix.h>
#include <deal.II/lac/dynamic_sparsity_pattern.h>
#include <deal.II/lac/solver_cg.h>
#include <deal.II/lac/precondition.h>
#include <deal.II/lac/affine_constraints.h>
#include <deal.II/base/convergence_table.h>
#include <deal.II/grid/tria.h>
#include <deal.II/grid/tria_accessor.h>
#include <deal.II/grid/tria_iterator.h>
#include <deal.II/grid/grid_generator.h>

#include <deal.II/dofs/dof_handler.h>
#include <deal.II/dofs/dof_accessor.h>
#include <deal.II/dofs/dof_tools.h>

#include <deal.II/fe/fe_q.h>
#include <deal.II/fe/fe_values.h>

#include <deal.II/numerics/vector_tools.h>
#include <deal.II/numerics/matrix_tools.h>
#include <deal.II/numerics/data_out.h>
#include <deal.II/lac/sparse_direct.h>

#include <deal.II/fe/fe_system.h>

#include <deal.II/base/timer.h>
// STL
#include <cmath>
#include <fstream>
#include <iostream> // std::cout, std::endl
#include <thread> // std::this_thread::sleep_for
#include <chrono> // std::chrono::seconds

// My Headers
#include "matrix_coeff.hpp"
```

```
46  #include "right_hand_side.hpp"
47  #include "neumann_bc.hpp"
48  #include "dirichlet_bc.hpp"
49  #include "initial_value.hpp"
50  #include "advection_field.hpp"
51
52  /*!
53  * @namespace TransientAdvectionDiffusionProblem
54  * @brief Contains implementation of the main object
55  * and all functions to solve a time dependent
56  * Dirichlet–Neumann problem on a unit square.
57  */
58  namespace Timedependent_AdvectionDiffusionProblem
59  {
60    using namespace dealii;
61
62    template <int dim>
63    class AdvectionDiffusionProblem
64    {
65      public:
66
67      AdvectionDiffusionProblem() = delete;
68      AdvectionDiffusionProblem(unsigned int n_refine, bool is_periodic);
69      void run ();
70
71      private:
72      void make_grid ();
73      void setup_system ();
74      void assemble_system ();
75      void solve_iterative ();
76      void solve_direct ();
77      void output_results () const;
78      void compute_errors();
79      // void timestep();
80
81      Triangulation<dim>    triangulation;
82      FE_Q<dim>    fe;
83      DoFHandler<dim>    dof_handler;
84
85      AffineConstraints<double>  constraints;
86
87      SparsityPattern    sparsity_pattern;
88      SparseMatrix<double> mass_matrix;
89      SparseMatrix<double>    diffusion_matrix;
90      SparseMatrix<double>    advection_matrix;
91      SparseMatrix<double>    system_matrix;
92
93      Vector<double>    solution;
94      Vector<double> old_solution;
95      Vector<double>    static_rhs; // only the time independent part
96      Vector<double>    system_rhs;
97
98      double time;
99      double time_step;
100     unsigned int timestep_number;
101
102     const double theta;
```

```
103     const double T_max;
104     ConvergenceTable convergence_table;
105
106   };
107
108
109   template <int dim>
110   AdvectionDiffusionProblem<dim>::AdvectionDiffusionProblem (unsigned int n_refine, bool is_periodic) :
111   fe (1),
112   dof_handler (triangulation),
113   time(0.0),
114   time_step(1. / 100),
115   timestep_number(0),
116   theta(1),
117   T_max(0.5)
118   {}
119
120
121   template <int dim>
122   void AdvectionDiffusionProblem<dim>::make_grid ()
123   {
124     GridGenerator::hyper_cube (triangulation, 0, 1, /* colorize */ true);
125
126     triangulation.refine_global (2);
127
128     std::cout << "Number of active cells: " << triangulation.n_active_cells()<< std::endl;
129   }
130
131
132   template <int dim>
133   void AdvectionDiffusionProblem<dim>::setup_system ()
134   {
135     dof_handler.distribute_dofs (fe);
136
137     std::cout << std::endl
138     << "===========================================" << std::endl
139     << "Number of active cells: " << triangulation.n_active_cells()
140     << std::endl
141     << "Number of degrees of freedom: " << dof_handler.n_dofs()<< std::endl << std::endl;
142
143
144     constraints.clear();
145     DoFTools::make_hanging_node_constraints(dof_handler, constraints);
146
147     /*
148     * Set up Dirichlet boundary conditions.
149     */
150     const Coefficients::DirichletBC<dim> dirichlet_bs;
151     for (unsigned int i = 0; i<dim; ++i)
152     {
153       VectorTools::interpolate_boundary_values(dof_handler,/*boundary id*/ 2*i, // only even boundary id
154       dirichlet_bs, constraints);
155     }
156
157     constraints.close();
158
159
```

```
160     DynamicSparsityPattern dsp(dof_handler.n_dofs());
161     DoFTools::make_sparsity_pattern (dof_handler,dsp,
162     constraints,/*keep_constrained_dofs = */ true); // for time stepping this is essenttial to be true
163     sparsity_pattern.copy_from(dsp);
164
165     system_matrix.reinit (sparsity_pattern);
166     diffusion_matrix.reinit (sparsity_pattern);
167     advection_matrix.reinit (sparsity_pattern);
168     mass_matrix.reinit(sparsity_pattern);
169
170     solution.reinit (dof_handler.n_dofs());
171     old_solution.reinit(dof_handler.n_dofs());
172     static_rhs.reinit (dof_handler.n_dofs());
173     system_rhs.reinit (dof_handler.n_dofs());
174   }
175
176
177   template <int dim>
178   void AdvectionDiffusionProblem<dim>::assemble_system ()
179   {
180   const QGauss<dim> quadrature_formula(fe.degree + 1);
181   const QGauss<dim - 1> face_quadrature_formula(fe.degree + 1);
182
183   FEValues<dim> fe_values(fe,quadrature_formula,update_values |
184   update_gradients |update_quadrature_points | update_JxW_values);
185
186   FEFaceValues<dim> fe_face_values(fe,face_quadrature_formula, update_values| update_quadrature_points |
187   update_normal_vectors | update_JxW_values); // for Neumaan boundary condition to evaluate
188   // boundary condition
189
190   const unsigned int dofs_per_cell = fe.dofs_per_cell;
191   const unsigned int n_q_points = quadrature_formula.size();
192   const unsigned int n_face_q_points = face_quadrature_formula.size();
193
194   FullMatrix<double> cell_matrix(dofs_per_cell, dofs_per_cell);
195   Vector<double> cell_rhs(dofs_per_cell);
196
197   std::vector<types::global_dof_index> local_dof_indices(dofs_per_cell);
198
199   std::vector<Tensor<2, dim>> matrix_coeff_values_old(n_q_points);
200   std::vector<Tensor<2, dim>> matrix_coeff_values(n_q_points);
201
202   std::vector<Tensor<1, dim>> advection_field_values_old(n_q_points);
203   std::vector<Tensor<1, dim>> advection_field_values(n_q_points);
204
205   std::vector<double> rhs_values_old(n_q_points);
206   std::vector<double> rhs_values(n_q_points);
207
208   std::vector<double> neumann_values_old(n_face_q_points);
209   std::vector<double> neumann_values(n_face_q_points);
210
211   for (const auto &cell : dof_handler.active_cell_iterators())
212   {
213     if (cell->is_locally_owned())
214     {
215       cell_matrix = 0;
216       cell_rhs = 0;
```

```
217
218         fe_values.reinit(cell);
219         cell->get_dof_indices(local_dof_indices);
220         /*
221         * Values at current time.
222         */
223         matrix_coeff.set_time(current_time);
224         advection_field.set_time(current_time);
225         right_hand_side.set_time(current_time);
226         advection_field.value_list(fe_values.get_quadrature_points(), advection_field_values);
227         matrix_coeff.value_list(fe_values.get_quadrature_points(), matrix_coeff_values);
228         right_hand_side.value_list(fe_values.get_quadrature_points(), rhs_values);
229
230         /*
231         * Values at previous time.
232         */
233         matrix_coeff.set_time(current_time - time_step);
234         advection_field.set_time(current_time - time_step);
235         right_hand_side.set_time(current_time - time_step);
236         advection_field.value_list(fe_values.get_quadrature_points(), advection_field_values_old);
237         matrix_coeff.value_list(fe_values.get_quadrature_points(), matrix_coeff_values_old);
238         right_hand_side.value_list(fe_values.get_quadrature_points(), rhs_values_old);
239
240         /*
241         * Integration over cell.
242         */
243         for (unsigned int q_index = 0; q_index < n_q_points; ++q_index)
244         {
245           for (unsigned int i = 0; i < dofs_per_cell; ++i)
246           {
247             for (unsigned int j = 0; j < dofs_per_cell; ++j)
248             {
249               // Diffusion is on rhs. Careful with signs here.
250               cell_matrix(i, j) +=
251               (fe_values.shape_value(i, q_index) *fe_values.shape_value(j, q_index) +
252               time_step * (theta) *(fe_values.shape_grad(i, q_index) *
253               matrix_coeff_values[q_index] *fe_values.shape_grad(j, q_index) +
254               fe_values.shape_value(i, q_index) *advection_field_values[q_index] *
255               fe_values.shape_grad(j, q_index))) *fe_values.JxW(q_index);
256               // Careful with signs also here.
257               cell_rhs(i) += (fe_values.shape_value(i, q_index) *
258               fe_values.shape_value(j, q_index) -time_step * (1 - theta) *
259               (fe_values.shape_grad(i, q_index) *matrix_coeff_values_old[q_index] *
260               fe_values.shape_grad(j, q_index) +fe_values.shape_value(i, q_index) *
261               advection_field_values_old[q_index] *fe_values.shape_grad(j, q_index))) *
262               fe_values.JxW(q_index) *old_solution(local_dof_indices[j]);
263             } // end ++j
264
265             cell_rhs(i) += time_step * fe_values.shape_value(i, q_index) *
266             ((1 - theta) * rhs_values_old[q_index] +
267             (theta)*rhs_values[q_index]) *fe_values.JxW(q_index);
268           } // end ++i
269         } // end ++q_index
270
271         if (!is_periodic)
272         {
273           /*
```

```
274           * Boundary integral for Neumann values for odd boundary_id in
275           * non-periodic case.
276           */
277           for (unsigned int face_number = 0;
278           face_number < GeometryInfo<dim>::faces_per_cell;
279           ++face_number)
280           {
281             if (cell->face(face_number)->at_boundary() &&
282             ((cell->face(face_number)->boundary_id() == 1) ||
283             (cell->face(face_number)->boundary_id() == 3) ||
284             (cell->face(face_number)->boundary_id() == 5)))
285             {
286               fe_face_values.reinit(cell, face_number);
287
288               /*
289               * Fill in values at this particular face at current
290               * time.
291               */
292               neumann_bc.set_time(current_time);
293               neumann_bc.value_list(fe_face_values.get_quadrature_points(), neumann_values);
294
295               /*
296               * Fill in values at this particular face at previous
297               * time.
298               */
299               neumann_bc.set_time(current_time - time_step);
300               neumann_bc.value_list(fe_face_values.get_quadrature_points(), neumann_values_old);
301
302               for (unsigned int q_face_point = 0;
303               q_face_point < n_face_q_points;
304               ++q_face_point)
305               {
306                 for (unsigned int i = 0; i < dofs_per_cell; ++i)
307                 {
308                   cell_rhs(i) +=time_step * ((1 - theta) *
309                   neumann_values_old[q_face_point] * // g(x_q, t_n) = A*grad_u
310                   // at t_n+(theta)*neumann_values
311                   [q_face_point]) * // g(x_q, t_{n+1}) = =// A*grad_u at t_{n+1}
312                   fe_face_values.shape_value(i, q_face_point) * // phi_i(x_q)
313                   fe_face_values.JxW(q_face_point); // dS
314                 } // end ++i
315               } // end ++q_face_point
316             } // end if
317           } // end ++face_number
318         }
319
320         constraints.distribute_local_to_global(cell_matrix,cell_rhs,
321         local_dof_indices, system_matrix, system_rhs,/* use_inhomogeneities_for_rhs */ true);
322       } // if
323     } // ++cell
324
325   system_matrix.compress(VectorOperation::add);
326   system_rhs.compress(VectorOperation::add);
327   }
328
329
330
```

```
331
332   template <int dim>
333   void AdvectionDiffusionProblem<dim>::solve_iterative ()
334   {
335     SolverControl solver_control (1000, 1e-12);
336     SolverCG<> solver (solver_control); // when program is runned CG iteration should not be zero//
337
338     PreconditionSSOR<> preconditioner;
339     preconditioner.initialize(system_matrix, 1.2);
340
341     solver.solve (system_matrix, solution, system_rhs, preconditioner);
342
343     constraints.distribute (solution);
344
345     std::cout << " " << solver_control.last_step()
346     << " CG iterations needed to obtain convergence."
347     << std::endl;
348
349   }
350
351   template <int dim>
352   void AdvectionDiffusionProblem<dim>::solve_direct ()
353   {
354
355     SparseDirectUMFPACK A_direct;
356
357     solution = system_rhs;
358     A_direct.solve(system_matrix, solution);
359     A_direct.vmult(solution, system_rhs);
360
361   }
362
363
364   template <int dim>
365   void AdvectionDiffusionProblem<dim>::output_results () const
366   {
367     DataOut<dim> data_out;
368     data_out.attach_dof_handler (dof_handler);
369     data_out.add_data_vector (solution, "solution");
370     data_out.build_patches ();
371
372     const std::string filename =
373     "solution-" + Utilities::int_to_string(timestep_number, 3) + ".vtk";
374     //std::ofstream output(filename);
375     std::ofstream output (dim == 2 ?"solution-2d.vtk"+filename :"solution-3d.vtk"+ filename);
376     data_out.write_vtk (output);
377
378   }
379
380 template <int dim>
381 void AdvectionDiffusionProblem<dim>::compute_errors()
382 {
383   Vector<float> difference_per_cell(triangulation.n_active_cells());
384   VectorTools::integrate_difference(dof_handler, solution,
385   ZeroFunction<dim>(), difference_per_cell,
386   QGauss<dim>(fe.degree + 1), VectorTools::L2_norm);
387
```

```
388     const double L2_error = VectorTools::compute_global_error(triangulation,
389     difference_per_cell, VectorTools::L2_norm);
390
391
392     const unsigned int n_active_cells = triangulation.n_active_cells();
393     const unsigned int n_dofs = dof_handler.n_dofs();
394     VectorTools::integrate_difference(dof_handler,
395     solution, ZeroFunction<dim>(),
396     difference_per_cell, QGauss<dim>(fe.degree + 1),
397     VectorTools::H1_norm);
398
399     const double H1_error = VectorTools::compute_global_error(triangulation,
400     difference_per_cell,VectorTools::H1_norm);
401
402     VectorTools::integrate_difference(dof_handler,
403     solution, ZeroFunction<dim>(),
404     difference_per_cell, QGauss<dim>(fe.degree + 1),
405     VectorTools::Linfty_norm);
406     const double Linfty_error =VectorTools::compute_global_error(triangulation,
407       difference_per_cell, VectorTools::Linfty_norm);
408     pcout << " Number of active cells: "
409     << n_active_cells<< std::endl
410     << " Number of degrees of freedom: " << n_dofs
411     << std::endl;
412     convergence_table.add_value("cells", n_active_cells);
413     convergence_table.add_value("dofs", n_dofs);
414     convergence_table.add_value("L2", L2_error);
415     convergence_table.add_value("H1", H1_error);
416     convergence_table.add_value("Linfty", Linfty_error);
417     convergence_table.set_precision("L2", 3);
418     convergence_table.set_precision("H1", 3);
419     convergence_table.set_precision("Linfty", 3);
420     convergence_table.set_scientific("L2", true);
421     convergence_table.set_scientific("H1", true);
422     convergence_table.set_scientific("Linfty", true);
423     convergence_table.set_tex_caption("cells", "\\# cells");
424     convergence_table.set_tex_caption("dofs", "\\# dofs");
425     convergence_table.set_tex_caption("L2", "L^2-error");
426     convergence_table.set_tex_caption("H1", "H^1-error");
427     convergence_table.set_tex_caption("Linfty", "L^\\infty-error");
428
429     convergence_table.set_tex_format("cells", "r");
430     convergence_table.set_tex_format("dofs", "r");
431
432     std::cout << std::endl;
433     convergence_table.write_text(std::cout);
434
435     std::ofstream error_table_file("tex-conv-table.tex");
436     convergence_table.write_tex(error_table_file);
437
438     deallog << " Error in the L2 norm : " << L2_error << std::endl;
439     deallog << " Error in the H1 norm : " << H1_error <<std::endl;
440     deallog << " Error in the Linfty norm : " << H1_error << std::endl;
441 }
442     template <int dim>
443     void AdvectionDiffusionProblem<dim>::run ()
444     {
```

```
445     std::cout << "Solving problem in " << dim << " space dimensions." << std::endl;
446
447     make_grid ();
448
449     setup_system ();
450
451     assemble_system ();
452
453     Vector<double> tmp;
454     Vector<double> forcing_terms;
455
456     tmp.reinit(solution.size());
457     system_rhs.reinit(solution.size());
458
459     // initialize with initial condition
460     VectorTools::project(dof_handler,
461     constraints, QGauss<dim>(fe.degree + 1),
462     Coefficients::InitialValue<dim>(),old_solution);
463
464     solution = old_solution;
465
466     // output initial condition
467     output_results();
468
469
470     while (time <= T_max)
471     {
472       time += time_step;
473       ++timestep_number;
474
475       std::cout << "Time step " << timestep_number << " at t=" << time<< std::endl;
476
477       // re-initializes the system_rhs = M*old_solution
478       mass_matrix.vmult(system_rhs, old_solution);
479
480       // system_rhs = M*old_solution - (1-theta)*dt*A*old_solution
481       diffusion_matrix.vmult(tmp, old_solution);
482       advection_matrix.vmult(tmp, old_solution);
483       system_rhs.add(-(1-theta) * time_step, tmp);
484
485       // system_rhs = M*old_solution - theta*dt*A*_old_solution + dt*static_rhs
486       system_rhs.add(time_step, static_rhs);
487
488       // Now create the full system matrix to get system_matrix = M + dt+theta*A
489       system_matrix.copy_from(mass_matrix);
490       system_matrix.add(theta * time_step, advection_matrix);
491       system_matrix.add(theta * time_step, diffusion_matrix);
492       // Now take care of constraints
493       constraints.condense(system_matrix, system_rhs);
494
495       // Now solve
496       solve_direct ();
497
498       output_results ();
499
500       // reinitialize
501       tmp.reinit(solution.size());
```

```
502         system_rhs.reinit(solution.size());
503         system_matrix.reinit (sparsity_pattern);
504
505         // hand over solution value for next time step
506         old_solution = solution;
507
508       }
509       deallog << "Solve" << std::endl;
510       compute_errors();
511     }
512
513 } // namespace Timedependent_AdvectionAdvectionDiffusionProblem
514
515 #endif /* INCLUDE_DIFFUSION_PROBLEM_HPP_ */
```

Listing 61: The advection-diffusion problem header file.

Bibliography

[1] C++ language. https://www.cplusplus.com.

[2] C++ programming language. https://www.geeksforgeeks.org/c-plus-plus/,.

[3] Canopies in the earth system. https://www.cliccs.uni-hamburg.de/en/research/theme-a/a3.html. Last accessed on Jun 15, 2019.

[4] Climate, climatic change, and society (CLICCS). http://www.cliccs.uni-hamburg.de/about-cliccs.html. Last accessed on Jun 15, 2019.

[5] (2018). 2018 revision of world urbanization prospects. Report, Population Division of the United Nations Department of Economic and Social Affairs (UN DESA).

[6] Alebrand, S. (2015). *Efficient Schemes for Parameterized Multiscale Problems*. Ph. D. thesis, Universität Stuttgart.

[7] Arndt, D., W. Bangerth, M. Feder, M. Fehling, R. Gassmöller, T. Heister, L. Heltai, M. Kronbichler, M. Maier, P. Munch, J.-P. Pelteret, S. Sticko, B. Turcksin, and D. Wells (2022). The `deal.II` library, version 9.4. *Journal of Numerical Mathematics 30*(3), 231–246.

[8] Bagheri, B. and R. Scott. (2003). Analysa. http://people.cs.uchicago.edu/ridg/al/aa.ps,.

[9] Bangerth, W. (2000, 8). Using modern features of c++ for adaptive finite element methods: Dimension-independent programming in deal.ii. Michel Deville;Robert Owens,IMACS – Department of Computer Science, Rutgers University, New Brunswick., booktitle.

[10] Beljaars, A., G. Balsamo, P. Bechtold, A. Bozzo, R. Forbes, R. J. Hogan, M. Köhler, J.-J. Morcrette, A. M. Tompkins, P. Viterbo, and N. Wedi (2018). The numerics of physical parametrization in the ecmwf model. *Frontiers in Earth Science 6*.

[11] Blocken, B., Y. Tominaga, and T. Stathopoulos (2013). Cfd simulation of micro-scale pollutant dispersion in the built environment. *Building and Environment 64*, 225–230.

[12] Chung, E., Y. Efendiev, and T. Hou (2023). *Multiscale Model Reduction: Multiscale Finite Element Methods and Their Generalizations*. Springer International Publishing.

[13] Dedner, A., R. Klöfkorn, M. Nolte, and M. Ohlberger (2010). A Generic Interface for Parallel and Adaptive Scientific Computing: Abstraction Principles and the DUNE-FEM Module. *Computing 90*(3–4), 165–196.

[14] Dular, P., C. Geuzaine, F. Henrotte, and W. Legros (1998, September). A general environment for the treatment of discrete problems and its application to the finite element method. *IEEE Transactions on Magnetics 34*(5), 3395–3398.

[15] Efendiev, Y. and J. Galvis (2012, 08). Coarse-grid multiscale model reduction techniques for flows in heterogeneous media and applications. *Lecture Notes in Computational Science and Engineering 83*.

[16] Golden, K. M., L. G. Bennetts, E. Cherkaev, I. Eisenman, D. Feltham, C. Horvat, E. Hunke, C. Jones, D. K. Perovich, P. Ponte-Castañeda, C. Strong, D. Sulsky, and A. J. Wells (2020, 9). Modeling sea ice. *Notices of the Ameican Mathemical Society 10*, 1535–1553.

[17] Graham, I. G., T. Y. Hou, O. Lakkis, and R. Scheichl (2012). *Numerical Analysis of Multiscale Problems*. Springer.

[18] Hecht, F. (2012). New development in freefem++. *J. Numer. Math. 20*(3-4), 251–265.

[19] Heister, T. (2015, August). Parallel computations. https://www.dealii.org /workshop-2015/slides/heister.pdf.

[20] Jandaghian, Z. and U. Berardi (2019, 01). Proper choice of urban canopy model for climate simulations.

[21] Johnson, C. (1987). *Numerical solution of partial differential equations by the finite element method*. Cambridge University Press, Cambridge England ; New York.

[22] K. Heinke Schlünzen, S. G. and A. Baklanov (2023). *Guidance on Measuring, Modelling and Monitoring the Canopy Layer Urban Heat Island (CL-UHI)*. World Meteorological Organization.

[23] Kanda, I., Y. Yamao, T. Ohara, and K. Uehara (2012, 04). An urban atmospheric diffusion model for traffic-related emission based on mass-conservation and advection-diffusion equations. *Environmental Modeling and Assessment 18*.

[24] Kirby, R. C. and A. Logg (2007). Efficient compilation of a class of variational forms. *ACM Transactions on Mathematical Software 33*(3).

[25] Kirby, R. C., A. Logg, L. R. Scott, and A. R. Terrel (2006). Topological optimization of the evaluation of finite element matrices. *SIAM Journal on Scientific Computing 28*(1), 224–240.

[26] Lamoureux, P. (2017, 01). *Computational Modeling in Heterogeneous Catalysis*.

[27] Long., K. Sundance: Rapid development of high-performance parallel finite-element solutions of partial differential equations. http://software.sandia.gov/sundance/,.

[28] Park, Y.-S. and K. Tha Paw U (2004). Numerical estimations of horizontal advection inside canopies. *Journal of Applied Meteorology 43*(10), 1530 – 1538.

[29] Prud'homme, C. (2006, 01). A domain specific embedded language in c++ for automatic differentiation, projection, integration and variational formulations. *Scientific Programming 14*, 81–110.

[30] Prud'homme, C. (2007, 12). Life, a modern and unified c++ implementation of finite-element and spectral-element methods in 1d, 2d and 3d. *PAMM 7*, 1010605 – 1010606.

[31] Renard, Y. and J. Pommier. Getfem++:generic and efficient c++ library for finite element methods elementary computations. http://getfem.org/. Last accessed on July 17, 2020.

[32] Rheinboldt, W. C. and C. K. Mesztenyi (1980). On a data structure for adaptive finite element mesh refinements. *ACM Trans. Math. Softw. 6*, 166–187.

[33] Rotach, M., R. Vogt, C. Bernhofer, E. Batchvarova, A. Christen, A. Clappier, B. Feddersen, S.-E. Gryning, G. Martucci, H. Mayer, V. Mitev, T. Oke, E. Parlow, H. Richner, M. Roth, Y.-A. Roulet, D. Ruffieux, J. Salmond, M. Schatzmann, and J. Voogt (2005, 07). Bubble - an urban boundary layer meteorology project. *Theoretical and Applied Climatology 81*, 231–261.

[34] Schultz, M., M. Schatzmann, and B. Leitl (2005, 01). Effect of roughness inhomogeneities on the development of the urban boundary layer. *International Journal of Environment and Pollution - INT J ENVIRON POLLUTION 25*.

[35] Shammas, M. and H. M. Imran (2021, 11). Causes, modeling and mitigation of urban heat island: A review. *Earth Sciences 10*, 244–264.

[36] Simon, K. (2020a, October). 3d high-performance multiscale simulations of urban canopies. Presentation at CLICCS Retreat University of Hamburg.

[37] Simon, K. (2020b, July). Numerical methods for pdes. Lecture notes at University of Hamburg.

[38] Simon, K. and J. Behrens (2019). A semi-lagrangian multiscale framework for advection-dominant problems. In *International Conference on Computational Science*. Springer, 393–409.

[39] Simon, K. and J. Behrens (2020). Multiscale finite elements for transient advection-diffusion equations through advection-induced coordinates. *Multiscale Modeling & Simulation 18*(2), 543–571.

[40] Simon, K. and J. Behrens (2021, 05). Semi-lagrangian subgrid reconstruction for advection-dominant multiscale problems with rough data. *Journal of Scientific Computing 87*.

[41] Suni, T., A. Guenther, H.-C. Hansson, M. Kulmala, M. Andreae, A. Arneth, P. Artaxo, E. Blyth, M. Brus, L. Ganzeveld, P. Kabat, N. Noblet-Ducoudré, M. Reichstein, A. Reissell, D. Rosenfeld, and S. Seneviratne (2015, 12). The significance of land-atmosphere interactions in the earth system - ileaps achievements and perspectives. *Anthropocene 12*.

[42] Wick, T. (2020, September). Numerical methods for partial differential equations. https://www.repo.uni-hannover.de/handle/123456789/9301.

[43] Wilby, R. L. (2003). Past and projected trends in london's urban heat island. *Weather 58*(7), 251–260.

[44] Yalchin Efendiev, T. Y. H. (2009). *Multiscale Finite Element Methods: Theory and Applications*, Volume 4. Springer.

Hiermit versichere ich an Eides statt, dass ich die vorliegende Dissertation mit dem Titel: „Multiscale Finite Element method application to Canopies in Earth System" selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Hamburg, 19 September 2024 _____ Unterschrift: _____