1-1-2013

A Three Layered Framework for Annual Indoor Airflow CFD Simulation

Yue Wang
University of Pennsylvania, yulewang@gmail.com

A Three Layered Framework for Annual Indoor Airflow CFD Simulation

Abstract
Computational fluid dynamics (CFD) is one of the branches of fluid mechanics that uses numerical methods and algorithms to solve and analyze problems that involve fluid flows. Computers are used to perform the millions of calculations required to simulate the interaction of liquids and gases with surfaces defined by boundary conditions. Indoor airflow simulations are necessary for building emergency management, preliminary design of sustainable buildings, and real-time indoor environment control.

The simulation should also be informative since the airflow motion, temperature distribution, and contaminant concentration is important. However, CFD computation is usually time-consuming, and not suitable for simulating real-time indoor air movement. Many researchers are concentrating on both hardware utilization and CFD algorithms, to make simulation much faster. Fast flow simulations are important for some applications in the building industry, such as the conceptual design of indoor environment, or they are coupled with energy simulation to provide deep analysis on the performance of the buildings. Such application does not require the same high level of accuracy as traditional CFD simulation because it only requires conceptual or semi-accurate distributions of the flow but within a short computing time. However, year round simulation is needed rather than the analysis of two or three extreme cases in order to help the designer investigate the problem clearly. To meet these special needs, an efficient and informative fluid simulation method is needed to provide fast airflow simulation with an inevitable but nominal compromise in accuracy.

This research provides a comprehensive workflow for the designer to simulate and analyze the annual indoor environment. In addition to the hardware acceleration deployed, fast fluid simulation algorithm is developed, and a machine learning based interpolation is used to allow the simulation coverage to be conducted annually. The outcome of this research is a methodology that allows the annual simulation time similar to the one used to perform two or three extreme cases of simulation using current methods.

Degree Type
Dissertation

Degree Name
Doctor of Philosophy (PhD)

Graduate Group
Architecture

First Advisor
Ali M. Malkawi

Subject Categories
Computer Sciences | Environmental Engineering | Mechanical Engineering

A THREE LAYERED FRAMEWORK FOR ANNUAL INDOOR AIRFLOW CFD SIMULATION

Yue Wang

A DISSERTATION

in

Architecture

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2013

Supervisor of Dissertation

_____

Ali M. Malkawi

Professor of Architecture

Graduate Group Chairperson

_____

David Leatherbarrow

Professor of Architecture; Chair, Graduate Group in Architecture

Dissertation Committee

Ali M. Malkawi, Professor of Architecture

Paulo E. Arratia, Associate Professor of Mechanical Engineering and Applied Mechanics

Yun Kyu Yi, Assistant Professor of Architecture

To Prof. Malkawi

who mentors me to walk through my PhD journey

Acknowledgment

I could never find adequate words to express my gratitude to my advisor Professor Ali Malkawi. With his vision into the research frontier, unreserved support, constructive advice, broad knowledge, diligence, and great sense of humor, he has been my role model since my first day of graduate study. I appreciate his profound belief in my work and abilities. I would like to extend my sincere thanks to my committee members for their invaluable advice and suggestions: Prof. Paulo E. Arratia and Dr. Yun Kyu Yi.

This research has received help from scholars in Purdue University. Some part of the FFD simulations in this research was performed using a program developed by Wangda Zuo and Qinyan Chen with their permission.

ABSTRACT

A THREE LAYERED FRAMEWORK FOR ANNUAL INDOOR AIRFLOW CFD SIMULATION

Yue Wang

Ali Malkawi

Computational fluid dynamics (CFD) is one of the branches of fluid mechanics that uses numerical methods and algorithms to solve and analyze problems that involve fluid flows. Computers are used to perform the millions of calculations required to simulate the interaction of liquids and gases with surfaces defined by boundary conditions. Indoor airflow simulations are necessary for building emergency management, preliminary design of sustainable buildings, and real-time indoor environment control.

The simulation should also be informative since the airflow motion, temperature distribution, and contaminant concentration is important. However, CFD computation is usually time-consuming, and not suitable for simulating real-time indoor air movement. Many researchers are concentrating on both hardware utilization and CFD algorithms, to make simulation much faster. Fast flow simulations are important for some applications in the building industry, such as the conceptual design of indoor environment, or they are coupled with energy simulation to provide deep analysis on

the performance of the buildings. Such application does not require the same high level of accuracy as traditional CFD simulation because it only requires conceptual or semi-accurate distributions of the flow but within a short computing time. However, year round simulation is needed rather than the analysis of two or three extreme cases in order to help the designer investigate the problem clearly. To meet these special needs, an efficient and informative fluid simulation method is needed to provide fast airflow simulation with an inevitable but nominal compromise in accuracy.

This research provides a comprehensive workflow for the designer to simulate and analyze the annual indoor environment. In addition to the hardware acceleration deployed, fast fluid simulation algorithm is developed, and a machine learning based interpolation is used to allow the simulation coverage to be conducted annually. The outcome of this research is a methodology that allows the annual simulation time similar to the one used to perform two or three extreme cases of simulation using current methods.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

1.1 Motivation

Annual hourly simulation is important in many applications in the building simulation field. The building envelope is affected according to the outdoor condition and internal load, and both change rapidly throughout a year. The amount of solar energy received is the main cause of the daily and yearly temperature variations. These temperature variations also create forces that drive the atmosphere in its endless motions, which, in turn, affects outdoor humidity level. An indoor air control system relies on outdoor temperature and humidity values to calculates the enthalpy from it, and thus calibrates the following hour's control strategies. Moreover, occupant activity varies according to daily and weekly schedules. As a result, the hour-to-hour indoor condition can span a great range. This is the reason why most simulation algorithms in building simulation, such as energy simulation, tend to be performed annually.

Computational Fluid Dynamics (CFD) is one of the branches of fluid mechanics that uses numerical methods and algorithms to solve and analyze problems that involve fluid flows. CFD is widely used in building simulation studies. Computers are used to perform the millions of calculations required to simulate the interaction of liquids and gases with surfaces defined by boundary conditions.

Fast indoor airflow simulations are necessary for designing building emergency management systems, the preliminary design of sustainable buildings, and modeling real-time indoor environment control. The simulation should also be informative since the airflow motion, temperature distribution, and contaminant concentration are important. However, CFD computation is usually time-consuming and to simulate annual indoor air movement is not practical. Thus, only one or two extreme cases (such as the hottest hour in the summer and coldest hour in the winter) will be simulated by CFD in most design analysis.

Extreme cases will yield extreme results. Most realistic hourly results throughout the year will reside in much closer boundaries. Thus most assumptions and conclusions drawn from those extreme cases of simulations will be exaggerated. Moreover, while doing coupling simulation with other types of simulation (for instance, energy simulation or particle simulation) with hourly time step, a few extreme cases can hardly suffice. Research shows that CFD should be performed hourly when doing coupling simulation with energy simulation [1–7].

The motivation of this research is to accelerate CFD simulation, making it capable to perform year-round simulation.

1.2 Problem statement

Given a geometry with yearly boundary conditions, it is important to speed

up the traditional simulation strategies to cover annual simulation (8760 static state case hours) in a fairly reasonable amount of time and with acceptable precision.

This research will provide a way to utilize various methods and algorithms in order to speed up the simulation, and gives a tentative solution to make annual simulations possible.

1.3 Background

The traditional CFD workflow for solving annual hourly simulation will work as follows:

- Iterate all potential annual hourly boundary conditions, and solve them one by one.

- For each of the simulation cases, CFD will convert the boundary conditions into an equation set.

- When the final equation sets are formulated, an iterative solver will be used to solve the equation set.

In this traditional simulation strategy, there are three obvious different layers from bottom to top:

- CFD will first try to discrete the equations in the model part into several linear equations, and try to use an iterative method to solve these systems.

- Differential equations are used to describe the physics of the real world. Traditionally Navier-Stokes equations are applied to determine indoor air movement.

- Among the 8760 hourly cases, many cases are highly similar because the inputs do not change that much. Thus, reusing the calculated results as often as possible might help to reduce unnecessary calculation significantly.

Since this workflow will take months or even years to simulate a annual hourly case, the implementation of each layer should be improved in order to make it more efficient. Most research conducted has focused on only one layer at a time and it is assumed that some speed can be gained by exploring multiple layers simultaneously. However, there are limitations in each layer, and one can only obtain a certain speed in each layer. Thereby, it is better to speed up all three layers simultaneously, and try to combine the speed-ups in all layers together. There is a significant amount of research that contributes to understanding each of the three layers; however, the most urgent challenge is to solve and validate these models for building simu-

lation by integrating all these layers together to maximize the simulation performance.

This research provides a tentative method to accelerate each layer. The low-level mathematical procedures will be accelerated using the GPGPU method. Traditional Navier-Stokes equations will be replaced by the FFD model to speed up the convergence. And finally, strategy planning algorithm will be developed to minimize the simulation by case reuse.

All three layer optimizations will be designed as optional plugins. Each one is a standalone simulation optimization strategy, but it should also be integrated with other methods to make the framework as general-purposed as possible.

1.4 Dissertation outline

In the following chapters, each layer of the annual CFD simulation problem will be analyzed in detail. Based on these discussions, this research will propose a method to solve the problem.

In Chapter 2, a GPGPU acceleration method is introduced. An OpenCL routine is provided with case studies to validate its efficiency and accuracy.

In Chapter 3, a time-averaged FFD model is developed. The efficiency of the model is tested using a case study.

In Chapter 4, a strategy planning algorithm is provided to minimize the simulation by case reuse. A machine learning algorithm is used to predict case differences, which is used by a minimum spanning tree algorithm to plan the optimal simulation strategy. An interpolation method is developed to eliminate simulation for most annual cases. Both the machine learning algorithm and the interpolation method is also validated.

Finally, Chapter 5 summarizes the contribution of this research and discusses future work.

Chapter 2 GPU Accelerations

2.1 Introduction

Since air distribution within a room is usually highly turbulent, the Reynolds-averaged Navier-Stokes (RANS) equation and accompanying turbulent model are used instead of direct simulation of the traditional Navier-Stokes equation. The K-epsilon model is one of the most common RANS turbulence models. The model is widely used in building science research, especially indoor air quality and thermo distribution simulation [8–12].

The K-epsilon model is a industry standard for airflow simulation in building performance analysis, and most researches published in the building simulation field used this model as the primary engine. However, even with this time-averaged approach, the Navier-Stokes equation is notable for the length of time it takes to solve. Though researchers continuously improve the numerical algorithm to improve the efficiency, CFD is non-linear and can only be used to solve limited simple building simulations problems on modern CPU's.

The traditional CFD program divides the space into grids and applies the Navier Stokes equation, turbulent models, boundary conditions and so on to each grid, then solves the problem using Finite Difference Method,

Finite Volume Method, or Finite Element Method. Then the program will try to solve the linear system using the conjugate gradient method. The process is quite time consuming in general.

Several enhancements have been incorporated into the solving strategies of CFD codes recently. A strongly-implicit-procedure (SIP) solver is added to the iterative solution of implicit approximations of multidimensional partial differential equations [13–15]. This method is suitable for solving systems of linear equations resulting from a discretisation of partial differential equations (PDEs). It is a widely used method in fluid mechanics. The results show two to three speed ups over the original sparse matrix iterative solver. Also, Habich implemented the Bouzidi bounce-back with interpolation to meet the numerical requirements of continuous surfaces [16].

However, any improvements related to a pure numerical mathematical analysis view is quite limited. Those new solvers might solve certain matrix types faster, but they might perform even worse on other types of matrices since they rely heavily on the form of the matrix. Moreover, these methods would only give limited speed ups.

Recent advancement in the semi-conductors development make it possible that the power of parallel computing can be utilized to solve this bottleneck completely or at least partially [17–18]. The CPU clock speed in consumer products increases every year followed by Moore's Law, so

that it is possible to perform CFD more efficiently when new CPU chips are available. Moore's Law describes a long-term trend in the history of computing hardware, in which the number of transistors that can be placed inexpensively on an integrated circuit has doubled approximately every two years [19]. For the past 30 years, the CPU transistor counts against dates of introduction closely following Moore's Law [20].

However, the exponential processor transistor growth predicted by the Moore's Law does not always translate into exponentially greater practical computing performance. For example, the higher transistor density in multi-core CPUs doesn't greatly increase speed on many consumer applications that are not parallelized [21–22]. In light of the costs for manufacture, the performance per watt, and heat generation are concerned, consumer CPU products developed by AMD and Intel switched from speed improvements over one processor core, to multi core processors, which means, building a single thread application that runs on one CPU core will not see dramatical speed benefits in coming years [23–26]. Major CPU vendors' focus switch from improving clock speed, to integrating more cores to the processors. This means that a non-highly paralleled CFD program will not benefit much from the recent hardware development achievements. Remarkable engineering achievements in graphics processing should also be taken into account when designing the real-time CFD program, or the computation resources in personal computers might be wasted in vain. Moreover, traditional CFD's algorithms should still be dramatically improved to meet the performance

requirements.

For this reason, in recent years, several research projects explored new ways to speed up the simulation [27–29]. GPGPU (General Purposed Graphics Processing Unit) method is one of the tentative approaches. The GPU has attracted attention for numerical computing because its structure is highly parallelized and optimized to achieve high performance for image processing. GPU can perform floating-point calculations, which can be quickly translated into shading thanks to the accelerated hardware. Moreover, GPU speed has improved dramatically over the past five years, and the acceleration is now much greater than CPUs [30].

After realizing the power to do general-purpose computation on a GPU (a.k.a. GPGPU), several companies published their specifications and implementations of computation framework on their own GPUs. Compute Unified Device Architecture (CUDA) was developed by NVIDIA cooperation in 2007 [30]. AMD offers a similar SDK for their ATI-based GPUs and that SDK and technology is called Stream SDK (formerly CTM, Close to Metal), designed to compete directly with Nvidia's CUDA [31]. DirectCompute was developed by Microsoft to take advantage of the massively parallel processing power of a modern graphics processing unit (GPU) to accelerate PC application performance in Microsoft Windows Vista or Windows 7 [32]. Several research projects focused on utilizing the power of GPU to do CFD simulations using the GPGPU frameworks such as CUDA [27–29, 33–36]. Unfortunately, there

are still many problems related to the CUDA framework. Portability is one of the most important issues. Another is that CUDA programs can not be run on a traditional CPU.

In 2009, Apple Inc. developed a new technology called OpenCL which harnesses the power of GPU calculation to general purposed numerical calculations. With the support of AMD, Intel, and NVIDIA, Apple proposed OpenCL to the Khronos Group (creators of OpenGL, a cross-platform computer graphics API) as the basis for a new standard. Demonstrating the strength of the proposal, OpenCL was expanded to include digital signal processors (DSPs) and other specialized processor architectures. It was ratified as a royalty-free open standard in December 2008 [37]. On August 28, 2009, Apple released Mac OS X Snow Leopard, which contains a full implementation of OpenCL [38]. AMD and NVIDIA are closely following this step, releasing several OpenCL implementations in beta [39].

With the availability of these tools, it is now possible to apply the OpenCL computing model to many simulation scenarios (i.e. human evacuation, shadow simulation) to speed up performance. GPU is effective at computing many similar floating point calculations simultaneously [40–41]. Therefore if in a mathematical model, all the elements, or agents, share the same governing equation, it is possible to speed up the performance by parallel computing. For example, in Helbing's human evacuation model [42–43], all the pedestrians share the same social force equation so parallel computing

is possible by sharing and creating numerous threads with each thread predicting the movement of one agent. In this research, CFD is used as an example to introduce how the GPGPU method can provide benefits to building simulation since in a CFD computation, all the elements (grid cells) usually share the same governing equation, the Navier Stokes equation.

There are several ways to simplify the Navier-Stokes equation numerical solving method to make CFD calculations faster. One of the most popular ways is Fast Fluid Dynamics, which breaks the Navier-Stokes equations into several sub-equations and solves them one by one. The FFD scheme was originally proposed for computer visualization and computer games [44–47].

FFD creates a test bed to carry out experiments of fluid simulation on GPU. FFD's algorithm is a simple 4-step solver and can be written within one or two hundred lines of code (LOC), compared to CFD's millions of LOC. The structure of all FFD procedures are simple iterations over all the grids, which makes parallelization possible. However, traditional CFD does not iterate through all the grid elements, but solves non-linear equation set instead. This makes writing a FFD solver and converting the code to a GPU version is practically possible. There are many open source software libraries and applications widely available for download on various websites.

Some of the recent researches apply FFD to building simulation [48–49]. The results showed that the FFD is about 50 times faster than the CFD.

The FFD could correctly predict the laminar flow, such as a laminar flow in a lid-driven cavity at $Re = 100$. However this research also showed the FFD has some problems in computing turbulent flows due to the lack of turbulence treatments. Although the FFD can capture the major pattern of the flow, it cannot compute the flow profile as accurately as the CFD does. Researchers tried to improve the FFD by adding some simple turbulence treatments, but no general improvement was found.

The performance benefit of GPGPU computing gives hope to solve CFD problems more efficiently [28–29]. The major technical difficulty is that most mature CFD programs all have a large source code base. For example, OpenFOAM [50], an open source CFD toolbox, has millions of lines of code. When compiled into binaries, the program consumes about 200 megabytes of disk space. It is not practical to convert such large portions of code into GPU source code since GPU code and CPU code are different in many ways, and converting CPU code to GPU code might take even more time than writing the CPU code itself, inasmuch as GPU programming requires detailed knowledge of the hardware, and proficient GPGPU programming and debugging experience (GPU programming tools are not as advanced as CPU versions). Therefore these factors make GPU programming much more challenging than CPU programming. Moreover, CFD software programs do not use simple C programming paradigm, instead, they usually have very advanced software architecture, and use generic programming and object-oriented programming extensively to make the code maintainable.

OpenFOAM, for example, heavily depends on C++ features such as operator overloading, class inheritance, and templates. This makes it extremely difficult to convert to OpenCL or CUDA code without losing the programming flexibility provided by the software before. As a result, converting software programs such as this requires rewriting the code to C first, which makes the program source code footstep size increased by several times in the end.

Although various researchers use FFD as a simple testbed to show the possibility of running fluid simulation on GPU, no research has been published in the Building Simulation field that uses CFD that includes a fully fledged RANS model running on top of GPU, and there is also no available code ready to be used for production due to the above technical difficulties.

One of the goals of this research is to find a generic method to accelerate CFD algorithm, not the inaccurate FFD algorithm, via the OpenCL GPGPU framework, in order to accelerate the computation without losing accuracy of the CFD computation.

2.2 Methodology

Given the fact that it's not practical to transit a large code base from CPU to GPU architecture due to the aforementioned technical difficulties mentioned, and CFD programs usually contain a large source code footprint

with complicated software design and implementation, this research adopts an eclectic method instead of translating the entire RANS solver to OpenCL. Traditional performance tuning that is widely used in computer software engineering is performed by this research [51]. In software engineering, the performance tuning should follow these steps and this research strictly follows them:

1. Assess the problem and establish numeric values that categorize acceptable behavior.

2. Measure the performance of the system before modification.

3. Identify the part of the system that is critical for improving the performance. This is called the bottleneck.

4. Modify that part of the system to remove the bottleneck.

5. Measure the performance of the system after modification.

In the first step, OpenFOAM is used as the initial code base [50]. The execution binary files are compiled, and case files are simulated using the engine as the reference. Several unit testing procedures are written in this research to ensure that whenever the source codes are optimized, the simulation results should be similar as the reference implementation results (with small tolerance left for rounding error of floating point calculations).

Secondly, this research adopts a time profiling tool to measure the performance of the unmodified program. In this case, Apple's Xcode Instruments tool is used for timing [52]. This is a software built on top of Sun's DTrace utilities, and it is capable to time various system performance functions including the cumulative execution time consumption for each of the functions in the source code with great precision [53–54].

Thirdly, the function that consumes most of the running time, the bottleneck, is found. Later, this research will show that one small function, the conjugate gradient solver, takes most of the running time. This behavior is well known in numerical analysis and can be explained since it is the dominant procedure in finite methods.

Fourth, the bottleneck function is rewritten in OpenCL with exactly the same mathematical algorithm, with all the iteration running in parallel. Even though conjugate gradient is a small function, the conversion takes great effort. The reasons to that are explained later.

Lastly, with the unit test performed to ensure CPU version and GPU one give same results for simulation cases, this research compares the execution time of traditional programs and GPU counterparts to show the performance improvements.

2.3 Speeding up simulation using OpenCL

Though in theory it is possible to convert the entire program into a OpenCL version and run it in full speed, this is not practical. Given the fact that GPU programming is much more difficult than traditional programming, it is not possible to transit that large code base into an OpenCL program.

For the CFD-related problem, most computations happen in only a few pieces of code. For example, in OpenFOAM, there are thousands of C++ APIs and functions such as `fvm::`, `fvc::`, interpolation, matrix solvers, turbulent models, etc. However, benchmarking using profiling tools (such as Instruments on Mac OS X based on Sun's dtrace utility) shows that one of the procedures, the conjugate gradient method, takes more than 90 percent of the overall running time. Other procedures' running time can be neglected.

This behavior is easy to explain as shown in Figure 2.1. Traditional CFD programs divide the space into grids and apply the Navier Stokes equation, turbulent models, boundary conditions and so on to each grid, and then solve the problem using Finite Difference Method, Finite Volume Method, or Finite Element Method. The most common way to solve the Navier-Stokes equation in industry is to use the finite volume method. Most commercial or open source CFD codes have a finite volume solver built in, such that all the differential equations will be turned into a linear form when this method

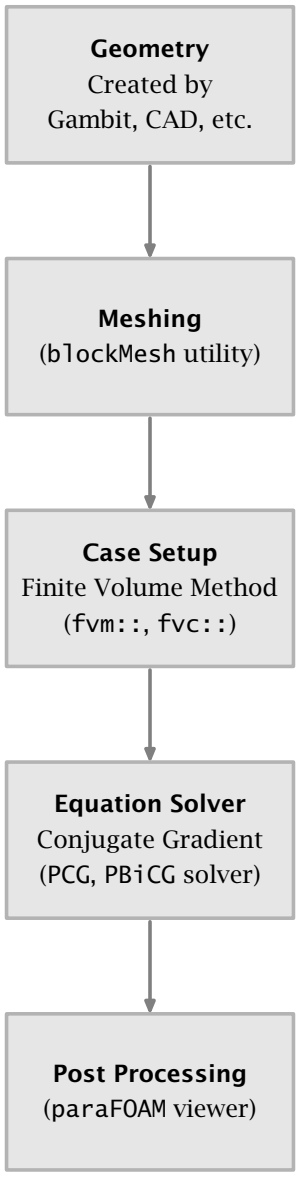


Figure 2.1 OpenFOAM simulation workflow
(All the computations are performed by CPU)

is applied. Then the program will try to solve the linear system using the conjugate gradient method.

For example, Dean and Glowinski listed all the procedures it requires to discretize the Navier-Stokes equation into several linear equations using the finite element approach [55]. Although the process is complicated, most operations are just one-pass scalar-vector multiplication or vector-vector addition, which is relatively simple and requires little time. However, solving the linear equation takes more time. Since the Navier-Stokes equation is non-linear, although it can be written in a linearized (matrix manipulation) fashion, both the left side and right side of the linear equation will have unknown variables. It requires many iterations to be converged, while each iteration is also a conjugate gradient iteration. The ADFC project implements the Navier-Stokes solver using finite element method in C++, and preliminary benchmarks show similar results (90 percent of the running time is devoted to conjugate gradient method, implemented in the source code file `gradiente_conjugado.c`) as of OpenFOAM [55–56].

In mathematics, the conjugate gradient method is an algorithm for the numerical solution of particular systems of linear equations, namely those whose matrix is symmetric and positive-definite [57]. The conjugate gradient method is an iterative method, so it can be applied to sparse systems that are too large to be handled by direct methods such as the Cholesky decomposition. Such systems often arise when numerically solving partial differential equations [58].

The conjugate gradient method for solving $Ax = b$ where A is a real,

positive-definite matrix can be written in a small piece of code [59]. For example, the conjugate gradient method source code in OpenFOAM, `PCG.C` and `PBiCG.C` is only 200 lines each with comments, or 70 lines with comments stripped. This helps make GPU acceleration possible.

Each step of the Conjugate Gradient method only requires one of the following three operations [59]:

- `saxby` operation is used to compute $\mathbf{s} = a\mathbf{x} + b\mathbf{y}$;
- `product` operation is used to compute the product of a sparse matrix times a full vector;
- `dot_product` operation is used to calculate the dot product of two vectors x and y.

With the following three parallel operations, the conjugate gradient method can be implemented efficiently on a multi-core GPU:

- The parallel version of `saxby` creates the number of threads that equal to the number of elements in the vector, and for each GPU thread i, it computes $s_i = ax_i + by_i$.
- The parallel version of `product` uses CRS (Compressed Row Storage) form [60] to reduce the matrix storage (so the 0s are not stored in the

memory) as well as to increase the efficiency (reduce the $O(n^2)$ problem into a $O(n)$ problem). For each GPU thread i, it calculates the i-th element of the resulting vector.

- The parallel version of `dot_product` requires simultaneously adding all the $x_i y_i$ values together in order to utilize the parallel computing power. There is extensive research on this topic and the most mature strategy is to utilize the parallel prefix reduce method [61].

Finally, the original `PCG.C` and `PBiCG.C` source code were replaced with the newly written OpenCL parallelized `OpenCLCG.C`. The changed code is hosted on Google code.

The change of the code is massive and takes a great amount of human labor because many technical difficulties are encountered during the modification:

- GPU Programming is, by nature, much more complicated than traditional programming in nature. Most programming interfaces and instructions are low-level, and they require knowledge of the hardware to get maximized performance, However, few technical details are published by the GPU vendors. There are also few mature tools available which can execute runtime debugging and profiling compared to traditional CPU capacities.

- OpenFOAM (and most other mature CFD codes) use C++ programming language to abstract higher level parameters (vectors or tensors) as classes, and use advanced C++ features (operator overloading, templates, generic libraries) for rapid development. However, none of these features are available in OpenCL. This procedure should convert to a clean C implementation with additional utility function and glue code to convert between C data structures and C++ classes.

- The converted C programs should be converted to GPU version. Each iteration should be abstracted as thread, and all the variables should be manually allocated, deallocated, copied, sent and received, in proper place and time. After the conversion, the code is almost expanded by ten times, and many of its parts are solely memory objects handling code.

As shown in Figure 2.2, the original source code with comments purged was about 70 lines and is now expanded to more than 600 lines, which includes the three kernels described before as well as many important routines for controlling the GPU to create memory allocations, performing the calculation, as well as sending the data between GPU and CPU forward and backward. Since all solvers and models are directly based on these matrix solvers, the modification makes it possible to accelerate any solvers so all the cases that were supported by the program can be simulated without any problem.

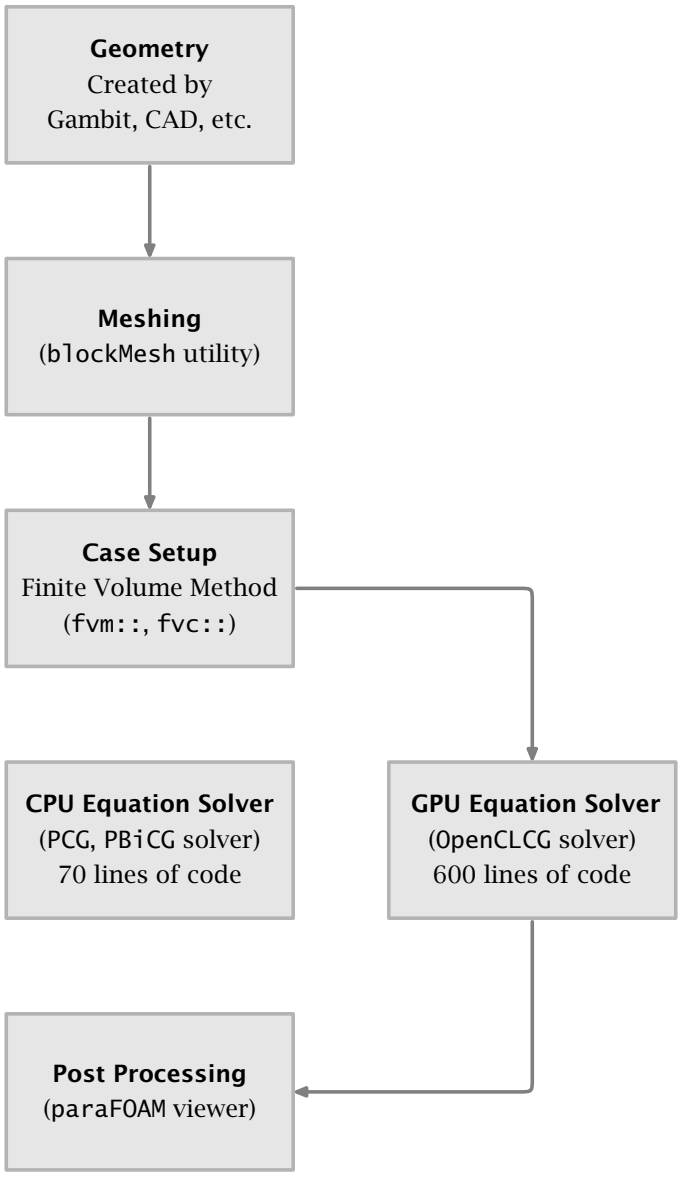


Figure 2.2 OpenFOAM simulation workflow

2.4 Case studies

2.4.1 Cavity case benchmark

This research is benchmarked on an Intel Xeon CPU. The frequency of the processor is 3.60GHz. A case is used with space divided into 500×500 grids, and let the OpenFOAM program to simulate one time step. Different GPU cards are used to test the program.

For the CPU case, this research used the unmodified OpenFOAM solver to simulate the case. The conjugate gradient solver requires 81.1 seconds to accomplish the work while other procedures required 3.55 seconds. This confirms our previous statement that conjugate gradient is the bottleneck.

This research attempted to simulate the case using different GPUs with the `OpenCLCG.C` solver. GeForce 9400M card uses 32.04 seconds on the conjugate gradient procedure. GeForce 9800 GTX card uses 8.03 seconds. Quardro FX 5800 card uses 2.81 seconds. Thereby, it is 28.86 times faster.

The performance of cards varies because each card uses different technology and configurations. For example, the Quadro card uses 512-bit GDDR3 memory, and the memory bandwidth is two times faster than GeForce 9800 GTX. The 240 processing cores inside Quadro FX 5800, compared to 16 in GeForce 9400M, and can perform much more threads concur-

rently. It is expected that with the latest generation or cards, such as Fermi, the performance can be even further improved.

As demonstrated with the benchmark Figure 2.3, instead of using 81.1 seconds, the conjugate gradient now takes about 2.81 seconds to finish, even faster than other procedures, which takes 3.55 seconds to finish. Thereby, the conjugate gradient method procedure is no longer the bottleneck.

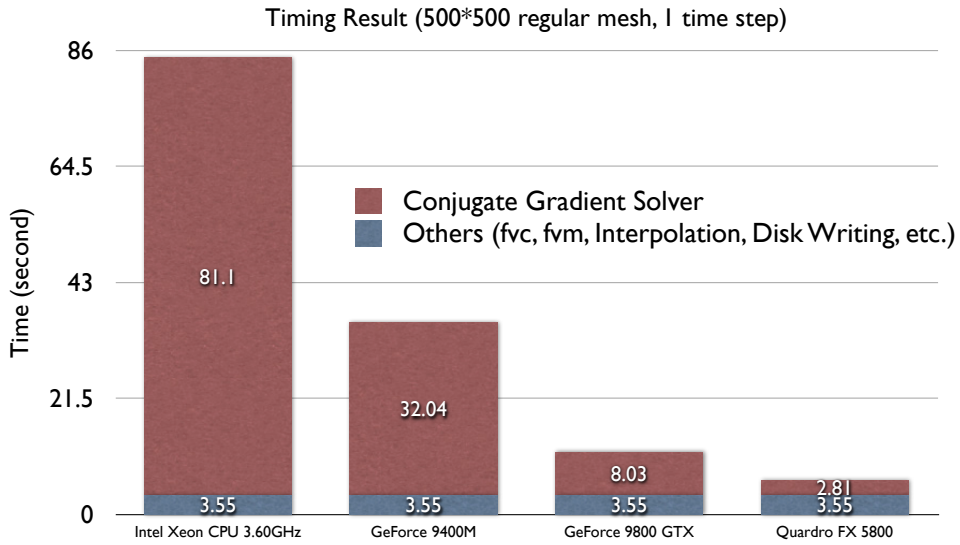


Figure 2.3 OpenCL Benchmark for 500 by 500 regular mesh

2.4.2 Pure conjugate gradient benchmark

The performance speed up of the GPU code is heavily dependent on the

problem size. Problems with different sizes of banded matrixes (band size is 4) are solved by the conjugate gradient solver using Quardro FX 5800 card and the execution times in relation to problem sizes are listed as Figure 2.4.

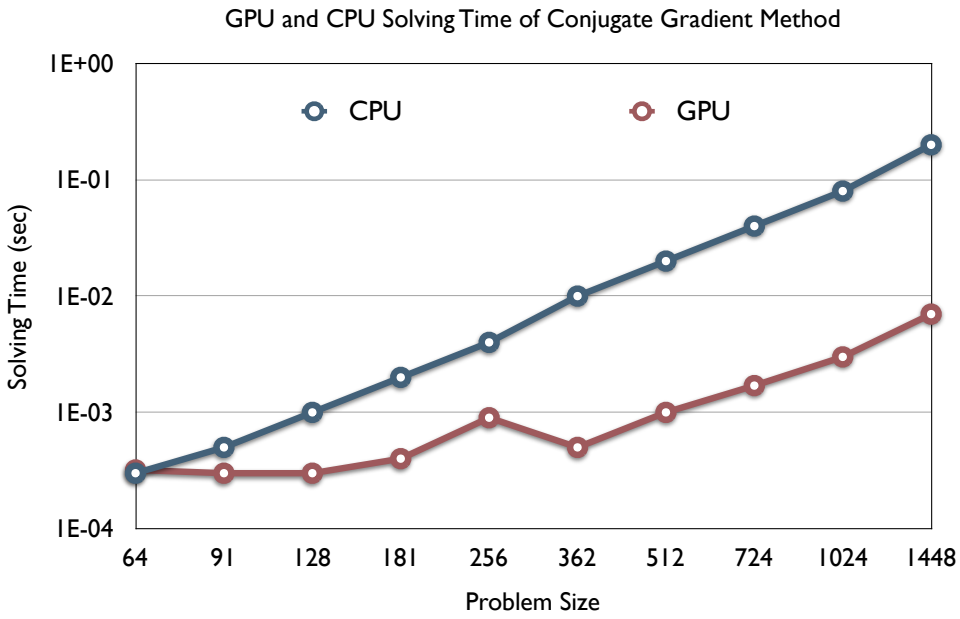


Figure 2.4 GPU and CPU solving time of conjugate gradient method

For each single iteration, the execution time of the CPU procedure grows linearly with the linear system problem size. This is because the number of floating points calculation (FLOP) is a linear function of matrix and vector sizes. However, for GPU the linear relation only holds when the problem size is relatively large (in this case, $N > 256$, which is called the turning point). For small problem sizes, GPU execution time is not linear. This

is because when OpenCL procedure is started, it takes some time to do initialization, kernel compilation, data loading and pushing, etc. For small problem sizes, GPU might be even slower than CPU, while for larger problems, these overheads can be neglected. Not until the problem size reached certain limits (called "turning point" in the following discussion) can the performance grow linearly.

In Zuo's research, the turning point is a hundred thousand while in this research it is 256 [28–29]. This is because the two simulation procedures are different. Zuo was executing full FFD in GPU, while this research only performs a CG on GPU. FFD has only four equations that takes short simulation time, but putting the entire program running on top of GPU requires much more memory objects, kernels, and other resources. Thus, the initialization, data transport, etc, take significantly longer. While in this research, the conjugate gradient resource requirement is more light weighted, but the actual calculation is more intense. In this case, the turning point is lower.

Although not sufficient for real time dynamic fluid simulation of large buildings, the performance improvement has several potential applications in building simulation. One possible area is the external coupling method of CFD and energy simulation. With the estimated speed up presented before, the annual case simulation is possible, and its total running time can be reduced to within a couple of hours, comparable to the time used for Energy Simulation, while a traditional CPU based method will take months.

2.4.3 Hot room case

The previous two cases compare the performance of conjugate gradient alone and CFD as a whole to show speed-up over the traditional code implementation. However, the accuracy of the OpenCL code is still open to examination. To illustrate that the OpenCL enhanced version can meet the precision requirement of building simulation study, in this case, the original *hot room* case provided by official OpenFOAM distribution is used as testing example [62]. The performance improvement is similar (25.4x) to the previous experiments. Here the research only focuses on numerical precision.

In this case, a room with dimensions of 10x6x2 meters and a box that will represent a heat source 1x1x0.5 meters in dimensions was tested. The temperature of walls, ceiling and floor is set to 300 K and the temperature of the heater to 500 K. The standard k-ϵ model is used and steady case is simulated. Refer to [50] manual and case file for its setup.

After the iteration has been completed, the temperature values along the x direction at the intersection line of plain $y = 0.7H$ and plain $z = 0.5D$ are extracted from the result file and compared. CPU version's result and GPU version's are almost the same, with relative difference lower than 0.1 percent. This is because the precision of the GPU used in this study is sufficient to perform 32bit floating point scientific calculations. The small

difference between CPU and GPU results, however, are related to rounding errors during the numerical evaluations.

2.5 Conclusions

As can be seen from the first case analysis, the method presented in this research can increase the performance of CFD computation to great extent, with much less effort than transitioning the entire code base. It is expected that this method can be applied to many other CFD codes as well because most CFD codes uses finite methods.

According to the second case, the turning point of this method is lower than previous studies of GPGPU fluid simulation, which means for almost any building simulation problem ranging from coarse grid to fine grid, GPU is always faster.

The most important feature of the method, as can be seen from the third case, is high precision. Its precision is of the same level as CPU calculation. While FFD uses different equations and algorithms, this method follows the same governing equations and numerical methods, which makes it highly accurate and thus can be applied to research and engineering work.

It is still possible to port other portions of the source code into OpenCL to eliminate more bottlenecks, and thus makes the program even faster.

It is also possible to use preconditioned method to further optimize the speed of conjugate gradient. However, these are beyond the scope of this research.

This research demonstrated a way to speed up the CFD program without losing any features or functionalities.

Chapter 3 FFD Accelerations

3.1 Introduction

The foundation of computational fluid dynamics is based on the Navier-Stokes equations. The Reynolds-averaged Navier-Stokes (RANS) equations are time-averaged equations of motion for fluid flow. They are primarily used while dealing with turbulent flows. The K-epsilon model is one of the most common RANS turbulence models [63]. It is a two-equation model, which means, it includes two extra transport equations to represent the turbulent properties of the flow [64]. This allows a two-equation model to account for history effects such as convection and diffusion of turbulent energy. The K-epsilon model should be a reference implementation for us since it is the most widely used solver in building science. When a new solver is developed, the results to the K-epsilon model should be compared to validate whether the new solver is accurate enough [65].

There are three major alternatives to the traditional method, all used heavily in computer animation. They are the Lattice Boltzmann Method, Smooth Particle Hydrodynamics, and Fast Fluid Dynamics.

Instead of solving the Navier-Stokes equations, the Lattice Boltzmann Methods (LBM) simulates the flow of a Newtonian fluid with collision models

such as Bhatnagar-Gross-Krook (BGK) [66–68]. Due to its particulate nature and local dynamics, LBM has several advantages over other conventional CFD methods, especially in dealing with complex boundaries, incorporation of microscopic interactions, and parallelization of the algorithm [69]. An important advantage of the Lattice Boltzmann Model allows efficient parallelization of the simulations. Benchmark shows that LBM on parallel machines with fast internet connection have close to linear speed-up when more processing units are added [70–73].

In general, room airflows are not fully turbulent. Most room airflows are locally turbulent [74]. Although measurements indicate that the flow in the main body of ventilated rooms may be transitional, airflow at diffuser outlets tends to be turbulent [75]. In this case, it is hard to use the Lattice Boltzmann Method to catch the turbulent property of indoor airflow, which is not good at simulating low Reynolds number problems.

Another alternative is Smoothed Particle Hydrodynamics (SPH). SPH was developed by L. B. Lucy and R. A. Gingold for the simulation of astrophysical problems, the method is general enough to be used in any kind of fluid simulation [76–77]. J. J. Monaghan and S. A. Munzel introduce SPH in detail [78–79] . The use of particles instead of a stationary grid simplifies these two equations substantially. Because the number of particles is constant and each particle has a constant mass, mass conservation is guaranteed and the equation of mass conservation can be omitted completely.

There are three problems that make SPH difficult to apply to building airflow simulations. First, in order to simulate realistic chaotic fluids, extremely high amounts of particles should be introduced to the system. Second, there are significant difficulties when introducing the traditional turbulent models into the kernel function, since turbulent models usually take advantage of time averages behavior, while simulating the particle movement over a certain amount of time and taking the average behavior will use much more computing power. Third, and most importantly, particles in the geometry are always moving similar to the fluid. It is not possible, for example, to calculate a steady-state case, such as the traditional one does. This is because the traditional CFD uses Euler view and SPH uses Lagrangian view.

The research thus will use the third approach, Fast Fluid Dynamics, which simplifies the Navier-Stokes equation numerical solving method to make CFD calculations faster, as the foundation. In Stam's research, for the first time, an unconditionally stable model which still produces complex fluid-like flows was proposed [44]. The unconditionally stable behavior increases the FFD model's performance because it is possible to use arbitrary long time step to perform the simulation. The FFD scheme was originally proposed for computer visualization and computer games [45–46].

FFD had a successful application in computer graphics, which intends to create an appealing effect. The accuracy of FFD for the engineering

application is open to question. Some of the recent researches apply FFD to building simulation. Zuo and Chen validate FFD for room airflow [48–49]. Even though FFD can correctly predict the laminar flow, it has some problems in computing turbulent flows due to the lack of turbulence treatments. Zuo and Chen continue to make several explorations and discovered many ways to improve their model [80–81].

The FFD algorithm is improved by increasing its speed and accuracy [82]. Enhancement of the computing speed can be realized by modifying the time-splitting method. Using the new FFD model for several indoor airflows, the results show a significant reduction in computing time and great improvements in accuracy. The method is further improved by reducing the numerical viscosity that is caused by a linear interpolation in its semi-Lagrangian solver [83]. The results show that the hybrid interpolation can significantly improve the accuracy of the FFD with a small amount of extra computing time.

In Stam's FFD model, the simulation will almost never reach a steady-state since the simulation numerical scheme does not ensure steady-state convergence. This research requires steady-state solution. In order to solve this problem, the time-averaged method is reintroduced in this research. Flows are simulated for a period of time, and averaged temperature and velocity are treated as steady state results.

A case study will be shown in the next section to validate real-time convergence behavior of the time-averaged FFD model.

3.2 Case study

A standalone fluid simulation system is written to predict indoor air movement. The system adopts Stem's FFD algorithms. Each iteration is composed of four computational steps [44]. After each iteration, a time-averaged residual is calculated. When the residual is below the threshold, the computation will be halted and the solution can be treated as the steady-state solution.

The system assumes the simulated case is a 2D rectangular room illustrated in Figure 3.1, with one inflow vent placed on the left wall and one outflow vent placed on the right one. The user can change the geometries of the system. In this application, the room width and room height can be adjusted via the interface. Also, the position of the two vents can be changed via the slider. Finally, software users can also change inflow temperature and speed.

The study employs fast fluid simulation algorithm developed by Jos Stam and validated by Qinyan Chen to perform the calculation. The system first conducts auto meshing according to the user input, then a steady case is simulated on the fly. A vector map of the indoor air speed and a gradient map of indoor air temperature distribution is plotted and visualized to the

Figure 3.1 Simple Room Case Demonstration

user. Because the simulation engine is robust, the visualization is displayed realtime just as users change the parameters.

The screenshot of the application is illustrated in Figure 3.2.

It can be seen from the software that the algorithm provides instant response to the value change. The convergence criteria are set to meet the averaged residual method mentioned in this chapter. The calculation is close to real-time.

3.3 Conclusions

This chapter first analyzed the historic research of improving the simulation performance by incorporating turbulent models. Stating those models are still too slow to reach real-time or faster-than-real-time simulation, this research incorporates Stem's Fast Fluid Dynamics method to use the steady-

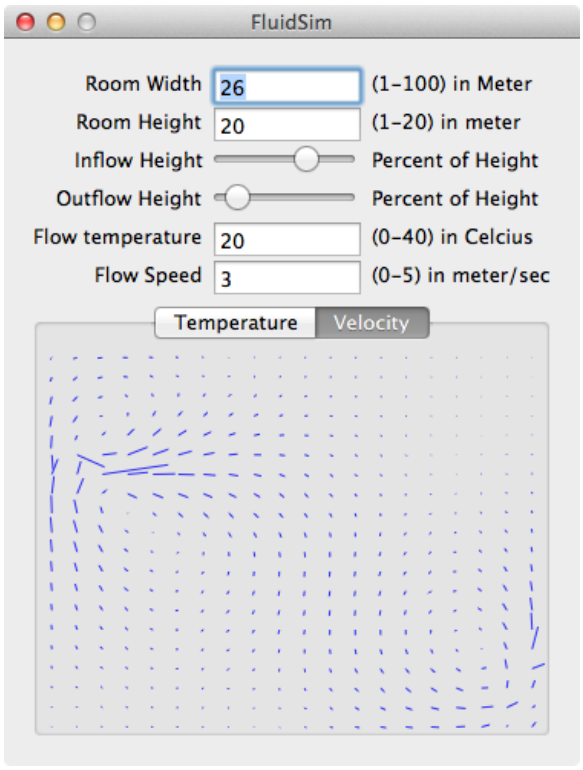


Figure 3.2 Screenshot of the FluidSim program

state fluid numerical solving scheme, and Zuo and Chen's turbulent model for FFD.

However, although Stem's stable fluid algorithm is unconditionally stable in terms of numerical scheme, it does not ensure steady-state solution because of fluctuations in the fluid. By re-introducing the time-averaged method, this research for the first time is able to simulate indoor airflow towards steady-state solutions.

As can be seen from the case study, this solution can always give steady-state solution for indoor air movement cases, and is highly efficient. The

estimated speed-up over the traditional CFD code is estimated to be 30 times to 50 times.

It is also possible to couple the FFD with GPGPU computing method introduced in the previous chapter to increase the two performance speed-up factors by two-fold. When multiplying the performance improvements introduced in this and the previous chapter, it is possible to get thousands-fold speed-up over traditional CFD code, thus making annual simulation possible. It is estimated that for a small room with a relatively fine grid (30 by 30 by 30) ventilation simulation, it usually takes several hours to several days to finish the annual hourly simulation, which is acceptable in some circumstances.

To achieve more speed up over the previous improvements, good optimization strategies should be applied to let the simulation engine only simulate what it needs to simulate, and reusing all the existing cases to perform new cases results without simulation, or at least eliminates all the unneeded iterations. In the next chapter, an optimal graph-based machine learning algorithm is presented to demonstrate the optimal simulation strategy, with the intent to further shorten the running time of annual hourly case simulation to within several hours.

Chapter 4 Strategy Planning Algorithm

4.1 Introduction

Since annual hourly simulations have more than 8000 hourly cases, to calculate each case independently may take longer simulation time. It is possible to reuse previous calculated results to generate new results, thus considerably shortening the simulation execution time. Among many methods, machine learning algorithms, which use existing results (training and testing set) to train the model which is used to predict future results, seem to be a good option.

Notably, artificial neural network (ANN) has been widely utilized. ANN has the solid ability of self-learning and self-organization. It can employ the prior acquired knowledge to respond to the new information rapidly and automatically. In the area of fluid flow, the applications of the ANN include the simulation of fluid flow and transport [84], the calculation of coefficients of heat transfer in fluid-particle systems [85], the prediction of flow field using a hybrid scheme of ANN [86], the dynamics simulation of a steam generator in a nuclear reactor [87], and the computation of the friction factor in pipeline flow [88].

An attractive attribute of machine learning models is their fast speed of predicted computation. Once an ANN model has been trained, tested and

validated, it can provide almost real-time parametric and sensitivity analysis. The typical time taken for an ANN model to execute one run is generally several orders of magnitude smaller than that required for running a CFD model [89]. Many researchers are trying to find a numerical method based on ANN to solve Navier-Stokes equations [90–94]. Most of the work showed that the trained ANN model predicts the behavior accurately. The results demonstrate the power and robustness of ANNs for obtaining fast responses to changing input conditions. However, although many studies focused on using artificial neural network to solve the flow transport problem, most researches, as listed above, do not use the Navier-Stokes equation as their training foundation equation. Most researches have a limited focus field such as flow and contaminant transport (GFCT) simulations, simulating Biot number or heat transfer coefficient. Those who use Navier-Stoke equations, usually focus on very low (less than 10) Re numbers, which are thus not applicable to the building simulation field.

Recent studies also use meshless methods as an alternative to grid-based flow computation [95]. Recent work has indicated that accurate results may be obtained with meshless methods as compared with grid-based methods [96–97]. In 2005, Zhi Shang combined ANN and meshless simulation together to simulate vapor-water two-phase flows in a tube with uniform and non-uniform heating. It does not generate mesh in the calculation domain except for some random points, nor does it solve the algebraic equations [98]. Again, it's challenging for machine learning algorithms to catch the

chaos properties of turbulence. Moreover, some machine learning algorithms would not solve the performance problems. Usually training all the parameters requires 1,000,000 iterations, which might take a considerable amount of time, and which takes much larger computation expanses than an annual result simulation. Thus, indoor air is chaotic and machine learning algorithms cannot catch the pattern very well.

CFD simulation is always time-consuming and it is even harder to cover annual simulation (8760 static state case) in a fairly reasonable amount of time and acceptable precision. Another way to reuse the result of existing cases, is to use CFD to converge new case simulation using existing cases' results as the starting point, and in some conditions the convergence will take much fewer iterations. However, randomly using existing cases may not achieve the desired performance gain, which is shown in the latter case study in Section 4.2; thus, such reusing should be based on scheduling strategies. This research will discuss a method to speed up annual CFD simulation by using scheduling strategies. Two tools are designed in this research to produce the optimal scheduling strategies.

1. A convergence speed predictor, discussed in Section 4.2, to predict the number of iterations needed for using one case's result as the starting point to simulate the other one.

2. A planning algorithm, discussed in Section 4.3, using the convergence

speed calculated by the predictor, to schedule an optimal strategy for annual hourly CFD simulation by reusing previously calculated results.

4.2 Predicting the convergence speed

Though machine learning methods might not capture the chaotic property of indoor air flow well, it can capture some of the other properties well, such as the difference of inputs of two cases and the iteration steps needed from one case as a string point to converge to the other case. There is a strong relationship between the input variance and the output variance, as well as strong relationships between the output variance and the iterations required for convergence from one case to another. This provides the foundation to design a convergence speed predictor used by an efficient algorithm discussed in Section 4.3 to reuse as many existing simulated cases as possible.

This research utilized a case study to investigate the possibility to design a convergence speed predictor new algorithm. It is a one-room case with a very simple ventilation system. The research used EnergyPlus to produce annual supply air information as well as the weather information as boundary conditions for CFD simulation. The goal of this case study is to explore the properties and relationships of input parameters, output parameters, and convergence speed.

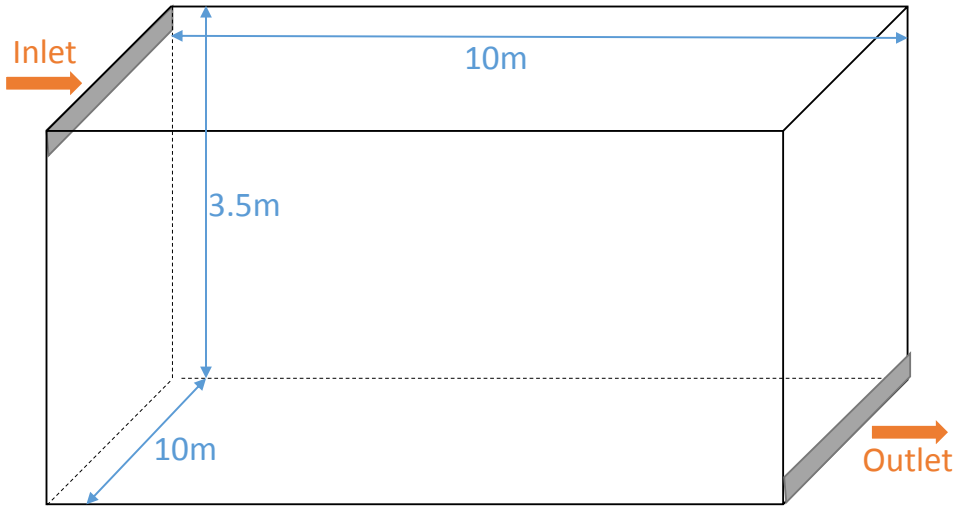


Figure 4.1 Reusable case study settings

Figure 4.1 describes the case, which has a 10m by 10m by 3.5m room. Its location is set to Los Angeles, whose weather data will be queried and used by the simulation. The wall has a U-value of $0.350 \mathrm{W/m^2K}$ and the roof has a U-value of $0.250\ \mathrm{W/m^2K}$. The internal gains in the room are from lighting and human body warmth. VAV box with reheat is assigned to condition the room. The outdoor temperature (as x axis) and indoor ventilation air speed (as y axis) of each hour from energy simulation is displayed as 8760 points in Figure 4.2. These points served as inputs to independent CFD cases.

1. Highly condensed input distribution:

   Typically, annual building simulation gives limited variables to corre-

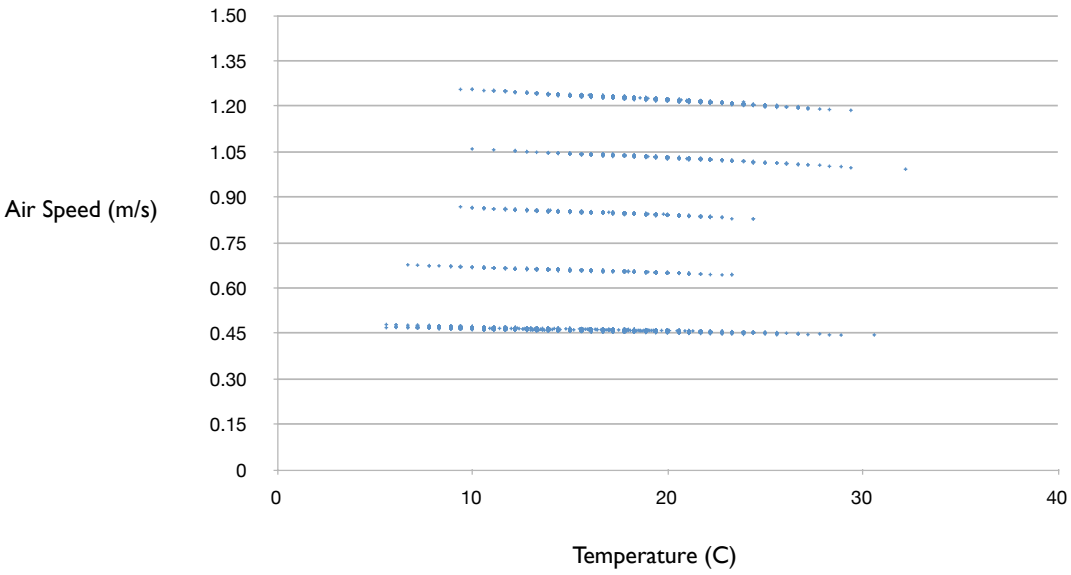


Figure 4.2 Outdoor temperature and indoor ventilation air speed, as output by energy simulation

sponding CFD simulation. The geometry is always the same, and boundary conditions are coarse and are simple functions of limited variables. Such variables form low dimensional input space. In this example, there are two variables (inflow air speed and outdoor temperature) and thus two dimensional space. Complex cases will have higher dimensions, but are usually still limited to less than five to ten dimensions.

Moreover, each dimension has very close boundaries. Building simulation has low variable variances. All temperatures fall within living conditions for human beings. Velocities are within one or two meters per second, contrary to other engineering fields (such as aerospace). In this case, all results reside within a rectangular shape from point

(278 K, 0.45 m/s) to (308 K, 1.2 m/s).

A annual simulation is composed of 8760 discrete steady state cases, each representing an hour's input setting in the year. There are almost ten thousand points residing in this low dimensional, small space, which leads to a very condensed distribution. Even if all the inputs are distributed randomly, for most of the given points, it is easy to find a point that is very close to it. If the inputs have certain patterns similar to the result given by this energy simulation, whose results form several almost-horizontal lines (since the inner condition change according to a fairly fixed schedule), such an observation can be even more obvious to find.

It is also clear that many of the cases are highly similar. For example, if today's noon weather is similar to tomorrow's noon weather, chances that indoor air control system will develop similar control strategies for the two cases are high, not including all other uncertainties. Thus, it is expected that the inputs of the cases are highly identical.

The following three observations are based on simulating four cases randomly selected: A, B, C, and D as listed in Table 4.1.

2. Output similarity based on input similarity:

If the input variables of two CFD cases are highly similar, the steady-state solutions tend to have high similarity between them. The greater

Case ID	Outdoor temperature	Inflow air speed
A	5C	0.6m/s
B	6C	0.6m/s
C	30C	0.6m/s
D	30C	0.7m/s

Table 4.1 Four selected cases in annual simulation

the two inputs differ, the greater the outputs deviate. For example, case $A = (5\ \text{C}, 0.6\ \text{m/s})$ and case $B = (6\ \text{C}, 0.6\ \text{m/s})$ differ only by 1C, thus their temperature field and velocity field are highly similar (less than 1 percent difference). However, case $B = (6\ \text{C}, 0.6\ \text{m/s})$ and case $C = (30\ \text{C}, 0.6\ \text{m/s})$ differs 24C, thus their outputs differ by around 10 percent. (Note: the percentage differences of temperature and velocity field are derived by averaging the result in each grid.)

3. Input has different impact on output:

Different parameters usually have different levels of impact on the output. Output is highly sensitive to some of the input variables. Considering the experiment, the outdoor temperature does not affect the indoor airflow too much while the slightest change in inflow velocity has a great impact on final results. For example, case $B = (6\ \text{C}, 0.6\ \text{m/s})$ and case $C = (30\ \text{C}, 0.6\ \text{m/s})$ differ around 10 percent. However, case $C = (30\ \text{C}, 0.6\ \text{m/s})$ and case $D = (30\ \text{C}, 0.7\ \text{m/s})$ differ greater than 50 percent. Thus, this case is more sensitive to the flow speed than to

outdoor temperature.

4. Iterations depend on output similarities:

   Usually the number of iterations needed for using case A's result as the starting point to simulate B, heavily depends on whether case A's result is similar to B's result or not. This property is easy to explain. When using case A's result as the initial starting point and to start the conjugate gradient method iteration, the converging speed from A to B is closely related to their solution distance.

   In this example, each case takes roughly 2000 iterations to converge without basing them on any pre-calculated result as a starting point. However, if two case's outputs are similar, it requires fewer iterations. As described before, since case $A = (5\ \mathrm{C}, 0.6\ \mathrm{m/s})$ and case $B = (6\ \mathrm{C}, 0.6\ \mathrm{m/s})$ are highly similar (less than 1 percent difference), It takes just 22 steps to complete the B case simulation when using A's result as the starting point. On the contrary, according to a previous analysis, case $C = (30\ \mathrm{C}, 0.6\ \mathrm{m/s})$ and case $D = (30\ \mathrm{C}, 0.7\ \mathrm{m/s})$ differ greatly (greater than 50 percent difference). While using case C's result as the starting point to converge to case D, it takes 2145 iteration steps, almost the same iteration steps required for simulation without the bases of existing results. Case B and case C's similarity lies in between (around 10 percent difference), and it takes 364 steps from B to C. Thus, iterations depend on output similarities. Thus, randomly using existing cases

(such as using C to simulate D) may not be favorable, and can even hurt the performance. One should always choose the optimal starting existing case for each of the case.

Cases	Input Difference	Output Difference	Iterations
A-B	(1C, 0m/s)	1 percent	22
B-C	(24C, 0m/s)	10 percent	364
C-D	(0C, 0.1m/s)	50 percent	2145

Table 4.2 Summary of relationship between input difference and iterations needed to converge from one case to another

The results of the previous three observations are summarized Table 4.2. The results reveal that the three quantities (input difference of two cases, output difference of two cases, and the number of iterations needed for using one case's result as a starting point to simulate the other one) are inherently equivalent. Thus, *the output difference can be used as an indicator of convergence speed*.

From the four observations in the case study, the following conclusions can be drawn:

1. Case reusing is possible since many cases are similar, according to the first observation.

2. The relationships discovered in observation two, three, and four, give insight to construct a learning algorithm to capture the strong correlation

with input difference and iteration steps needed, via the linkage of their output difference, rather than simply using the input-output pair as a testing and training set.

The two findings reveal that although it is difficult to generate a model to formulate the case output as a function of case input, it is much easier to formulate the difference of case outputs as a function of the difference of case inputs. Thus, rather than training using the input and output data, in this research, the difference of the inputs and the difference of the outputs are used as training set and testing set. Moreover, according to the observations above, the predicted difference's magnitude can be used as a strong indicator of the convergence speed.

Based on the machine learning model, a new scheduling algorithm will be developed. The following section describes the algorithm structure in its details to speed up by reusing as many existing cases as possible.

4.3 Minimal simulation by case reuse

There are large amount of similar cases for annual simulation. The output similarities of cases can be directly predicted by the input similarities. Then the optimal simulation strategies, which yields the list of simulation ordering to ensure the fastest converging speed between every two nearby cases, will turn into a graph-based problem. This research uses the output difference

of two cases as the indicator to tell if the two cases are similar enough (which is closely related to the number of iterations it takes according to the observations in Section 4.2). There are 8760 cases, and every two nearby cases have a weight which is the indicator defined before. The concept of minimum spanning tree algorithms in computer science is used to determine the optimal simulation strategy path efficiently.

The algorithm utilized several steps that include:

1. Simulate a few cases to produce the differences of input and output used in the algorithm. A machine learning algorithm learns the cases' difference and generates a model for that. Thus, in the future when given the input difference, the model will be able to predict the output differences.

2. Assign the edge values as the predicted difference in the solution graph.

3. The method also uses minimum spanning tree algorithm to calculate the best simulation strategies and perform a walk on the graph.

4. Interpolate most of the nodes' results without simulation, and converge other nodes' results.

4.3.1 Predicting output differences

The simulation procedure will begin with random cases of simulation as a starting point, while at the same time training the machine learning algorithm for preparation of predicting results for other cases in the future. Twenty to thirty cases will be simulated to calculate the difference of the input (400 samples).

For instance, given two sets of inputs, $A = (19\ \mathrm{C}, 1.5\ \mathrm{m/s}, 9\ \mathrm{C})$ and $B = (12\ \mathrm{C}, 2\ \mathrm{m/s}, 10\ \mathrm{C})$, the difference of the two inputs in both cases can be written as $A - B = (7\ \mathrm{C}, 0.5\ \mathrm{m/s}, 1\ \mathrm{C})$. Then, the difference of the output **v1** and **v2** is calculated. The final result is given as a percentage of the deviation. After the model is trained and given two inputs, it can *predict* the difference of the output. Because of the assumptions used in this research, this model can be very robust.

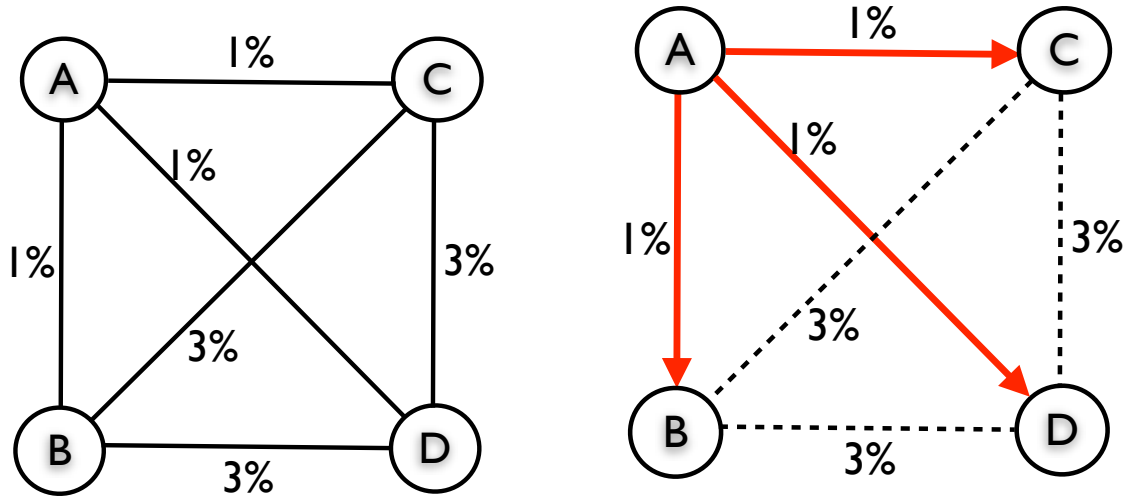
For instance, four cases A, B, C and D are chosen after the machine learning algorithm predicts all of the output differences between each pair of cases. Six relationships between each other can be seen: outputs of A and C have 1 percent difference; A and B have 1 percent; A and D have 1 percent; B and C have 3 percent; B and D have 3 percent; and C and D have 3 percent.

4.3.2 Assigning edge values in a graph

A graph is an abstract representation of a set of objects where some pairs of the objects are connected by links. The interconnected objects are represented by mathematical abstractions called vertices, and the links that connect some pairs of vertices are called edges. Typically, a graph is depicted in diagrammatic form as a set of dots for the vertices, joined by lines or curves for the edges. An undirected graph is one in which edges have no orientation.

In the undirected graph, the nodes represent all the cases and there are edges to link any two of the nodes. The weight assigned to the edges are the output differences between two nodes predicted by the machine learning model. Additionally, the output differences in the annual CFD simulation implies the number of iterations from one case to another. The algorithm should ensure that the traversal cumulative edge sum of all the nodes is minimal. There are a number of ways to calculate the optimal route to include all the nodes (the 8760 cases). This is a very classical problem in computer science called minimum spanning tree.

In the above example, an undirected graph with four nodes shown in Figure 4.3a will be constructed, representing the four cases A, B, C and D. The percentage on each edge is the output differences between these four cases, calculated by the machine learning algorithm.



(a) A weighted graph whose edges indicate case difference

(b) The minimum spanning tree for the weighted graph

Figure 4.3 A weighted graph and its minimum spanning tree

4.3.3 Strategy planning module

The goal of using a strategy planning module is to visit all of the 8760 nodes (cases) by choosing the minimum sum of edges. Thus, the strategy planning module was turned into a minimum spanning tree solver, which is efficiently solved by graph theory.

Given a connected, undirected graph, a spanning tree of that graph is a subgraph which is a tree that connects all the vertices together. A single graph can have many different spanning trees. Each edge can be assigned a weight, which is a number representing how unfavorable it is, and this can be used to assign a weight to a spanning tree by computing

the sum of the weights of the edges in that spanning tree. A minimum spanning tree or minimum weight spanning tree is then a spanning tree with weight less than or equal to the weight of every other spanning tree. More generally, any undirected graph (not necessarily connected) has a minimum spanning forest, which is a union of minimum spanning trees for its connected components [99].

The minimum spanning tree for the graph in Figure 4.3a can be obtained as the solid line tree in Figure 4.3b. In such a path, all four of these four nodes could be walked through from the starting point of A, determining that the traversal cumulative edge sum of the nodes is minimal. A better solution to this is not possible because at least three edges should be chosen in order to link four nodes, and choosing any other combination will have an edge value which is at least three, which is equal to the current total value; picking two other edges will result greater total value than the current one.

The planning strategy derived from the minimum spanning tree is optimal for the four cases because the minimum spanning tree has minimum total edge value to visit all the vertexes, indicating the total output differences of the selected path is minimal. According to the previous discussion, output difference of two cases, and the number of iterations needed for using one case's result as a starting point to simulate the other one is equivalent, so the total converging iterations required by the calculated planning strategy is the smallest.

The first algorithm for finding a minimum spanning tree was developed by Czech scientist Otakar Boruvka in 1926 [100]. Its purpose was an efficient electrical coverage of Moravia. There are now two algorithms commonly used, Prim's algorithm [101] and Kruskal's algorithm [102]. All three are greedy algorithms that run in polynomial time. This research uses Prim's algorithm because it is fairly simple to implement and has relatively high performance. For a highly complicated graph, the Prim's algorithm is necessary to judge the minimum spanning tree.

4.3.4 Multivariate interpolation

When the minimum spanning tree sequence is obtained, it is difficult to interpolate the new result based on existing results. The interpolation is not a single variable interpolation, because the input variables are a well formed vector containing ventilation speed, temperature and other boundary conditions. However, those values should be treated differently. As mentioned in the assumptions, some variables have less effect on the problem, while others have more. The weight of the interpolation should be assigned in a good way.

Staggeringly, a multivariate input interpolation problem can be reduced to a single variable by simply using the difference predicted by the machine learning algorithm. If case A and case C are predicted to differ x percent, and case C and case B are predicted to differ y percent, it is fairly reasonable to

interpolate C using case A and B by $(xA + yB)/(x + y)$. Thus, the predicted indicator not only gives precise information to form the minimum spanning tree, but also provides insight for the interpolation process.

However, the output result is multivariate too. Notably, the interpolation parameter for temperature field and vector field should be treated differently. During the machine learning model's training process, given two inputs, the model can be revised to produce the differences of different output parameters (such as temperature field, velocity field) as well; thus, this is one of the powerful aspects of the model. In the machine learning process, independent differences based on different parameters are learned and predicted. Though the combination of these difference values are used in the graph minimum spanning tree algorithm calculation, independent values are used in interpolation. Thus, the predicted value by the machine learning algorithm is a multivariate variable based on temperature and the velocity field.

4.4 Validation

4.4.1 Introduction

The following case is constructed in order to validate the previous planning algorithm and interpolation method.

Figure 4.4 shows a mechanically ventilated room with floor heating that generated a mixed convection flow. The flow pattern was influenced by both the inertial force and buoyancy force. The case was accompanied with experimental data from [103]. The model room of 1.04 m by 1.04 m was supplied with air horizontally injected into an empty room at a speed. The inlet height and outlet height were 0.018 m and 0.024 m, respectively. The temperature of the floor was 25.5 C and that of the other walls was outdoor temperature. The measured data showed that the flow pattern was two-dimensional. Thus, the FFD (Fast Fluid Dynamics) simulations used two-dimensional meshes. Three different meshes (20 by 20, 60 by 60, and 120 by 120) were used and the results showed that the mesh with 60 by 60 was sufficiently fine. All cases simulated reached convergence.

4.4.2 Regression and prediction

The annual simulation is performed hourly (8760 cases). As described in the algorithm, a small amount of cases should be directly simulated to train the machine learning model, which is used to predict the distance between each of the two cases in the input space. This example adopts the simplest machine learning method to fit the data — Linear Regression. Of course, more sophisticated learning methods, such as supported vector machine [104], principle component analysis [105], or least angle regression [106], may be used for more complicated tasks.

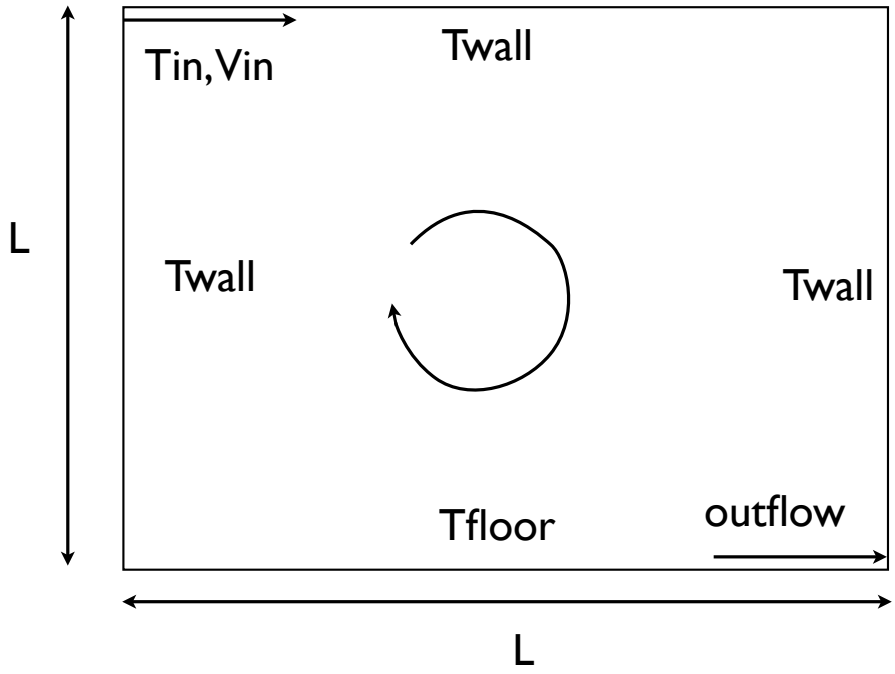


Figure 4.4 Mixed convection case settings

Figure 4.5 shows the predicted difference estimation vs the real difference after using 20 random cases' differences as training data. It can be seen from the figure that the predicted difference is fairly accurate for all the small difference value boundaries, while for larger differences, it tends to have slight noises. The overall residual standard error is 0.03128. Using the strategy planning algorithm, on the other hand, only small differences (most of them are within 5 percent) predicted will be used in actual calculation, as will be illustrated in the following sections. Larger (however inaccurate it is) values will all be discarded in the planning strategy.

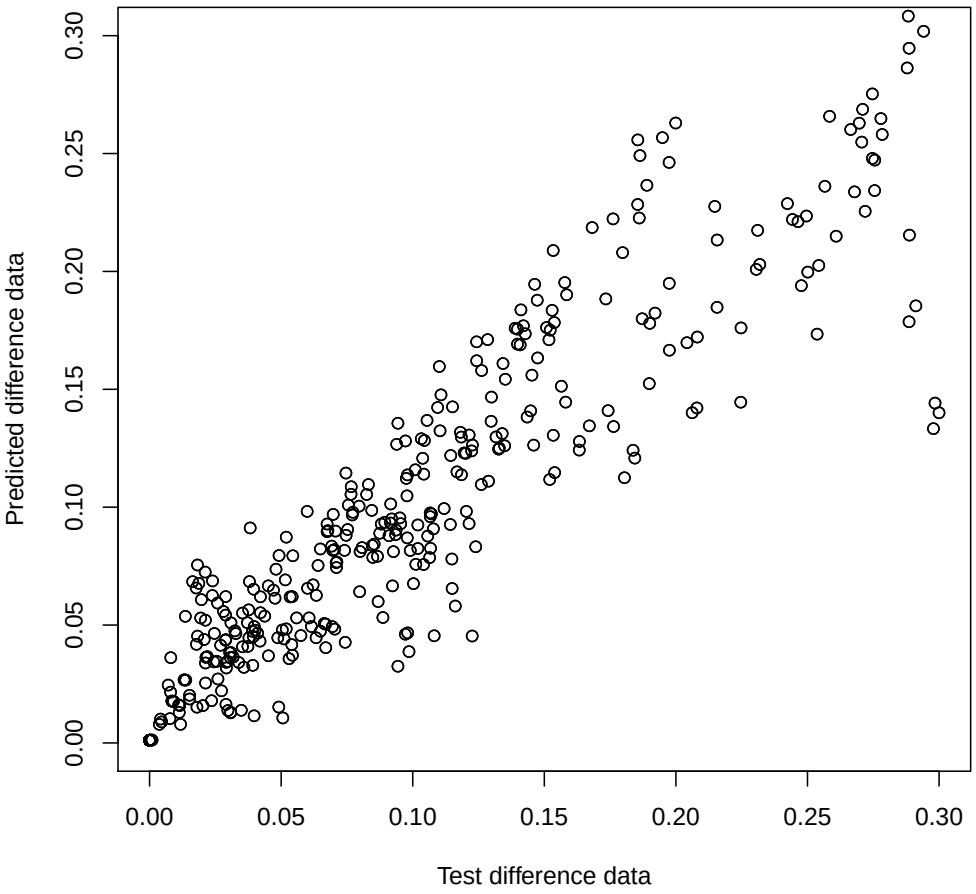


Figure 4.5 Predicted Difference vs. Real Difference

Figure 4.6 shows diagnostic plots for the regression:

1. The points in the Residuals vs. Fitted plot shown in Figure 4.6a are randomly scattered with no particular pattern.

2. The points in the Normal Q-Q plot shown in Figure 4.6b are almost on the line, indicating that the residuals follow a normal distribution.

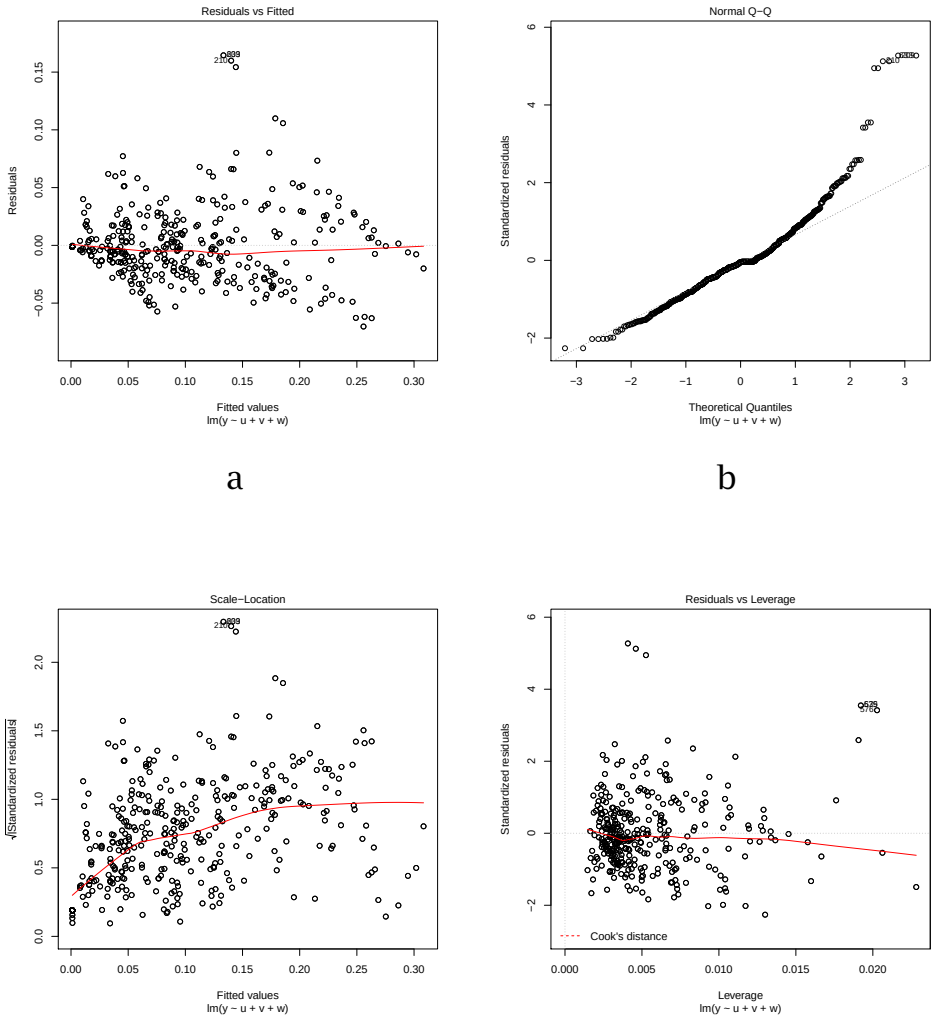


Figure 4.6 Diagnostic plots for the linear regression

3. In both the Scale-Location plot shown in Figure 4.6c and the Residuals vs. Leverage plots shown in Figure 4.6d, the points are in a group with none of them occurring too far from the center.

Thus, in this case, the linear regression can predict accurate output difference of two cases based on input difference.

4.4.3 The strategy graph

After the machine learning algorithm predicts the output differences based on input differences between every two cases, an optimal strategy based on the previously discovered minimum spanning tree algorithm is used to predict the best simulation strategy.

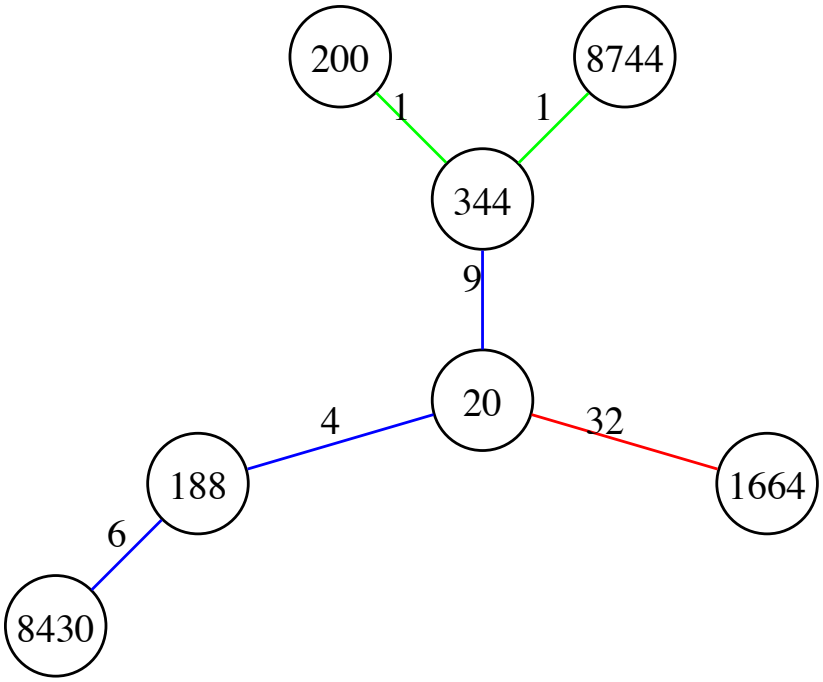


Figure 4.7 Magnified small fraction of the minimum spanning tree

Figure 4.7 is a small part from the calculated minimum spanning tree. The points linked by edges whose values are larger than an upper threshold (for

instance, 10 percent) need to be fully simulated. The points linked by edges whose values are lesser than a lower threshold (for instance, 5 percent) can be interpolated without simulation. The other points indicate that a fast convergence simulation can be performed after the interpolation procedure. The entire minimum spanning tree shows that most of the points are able to be interpolated; thus, the simulation time is saved greatly. The statistics of the simulation are provided in the next subsection.

A unique `id` number is assigned to each of the simulated cases, based on their hour index in the year. For example, if the `id` equals to 5, this means the case is for Jan 1 at 5AM in the morning. The simulation uses realistic outdoor temperature, simulated indoor air flow speed and temperature simulated by energy calculation.

The simulation is then performed according to the calculated strategy. A simple breath first search (BFS) walk can traverse the whole tree efficiently, interpolating all the small edges directly, converging the interpolated middle weighted edges, and then using the brutal force method to do the large weighted edges.

4.4.4 Solving cases that can be directly interpolated

The algorithm will interpolate all the small edges in the graph directly without simulation. For example, case `id` 200 and case `id` 8744 are found

by the algorithm linked by case `id` 344. Their outdoor temperature, inflow air speed and ventilation temperature are shown in Table 4.3.

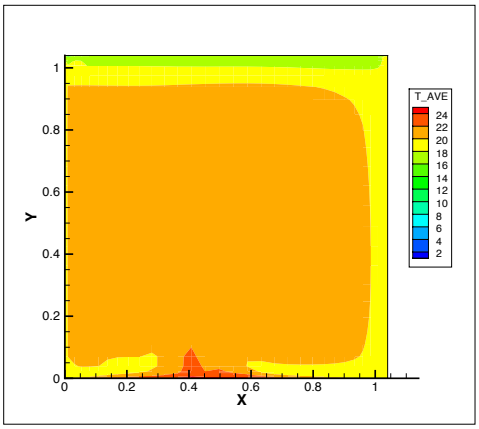
case `id`	**outdoor temperature**	**inflow speed**	**inflow temperature**
200	7.8C	0.674m/s	18.00102C
8744	11.1C	0.666m/s	18.10021C
344	9.4C	0.670m/s	18.00061C

Table 4.3 Input parameters for case `id` 200, 8744 and 344

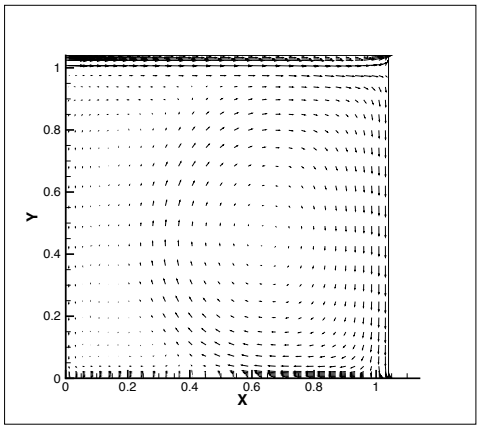
Figure 4.8 shows the temperature and velocity distribution result for case `id` 200 and case `id` 8744.

It is pretty easy to see that case `id` 200 and case `id` 8744 are really similar. The interpolated result of case `id` 344 based on case `id` 200 and case `id` 8744, together with the actual result simulated by the fluid solver, are shown in Figure 4.9.

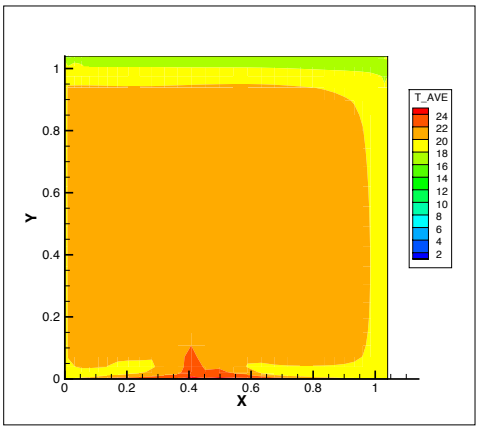
The interpolated and real results are highly identical. Using the difference ration calculation method in Appendix A, the average difference between the real simulation result and the interpolation result is lower than 1 percent, which can be neglected. This suggests in such scenarios a fluid simulation is not required for case 344; direct interpolation result will suffice.



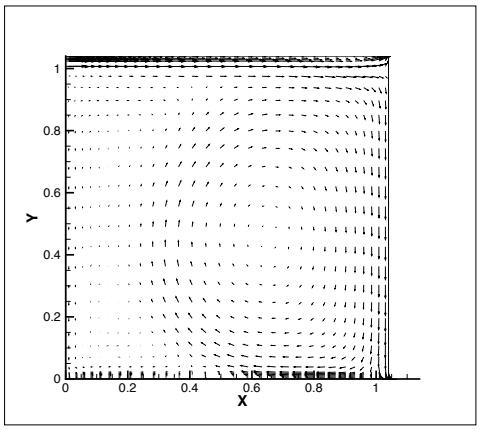
temperature (case 200)



velocity (case 200)



temperature (case 8744)

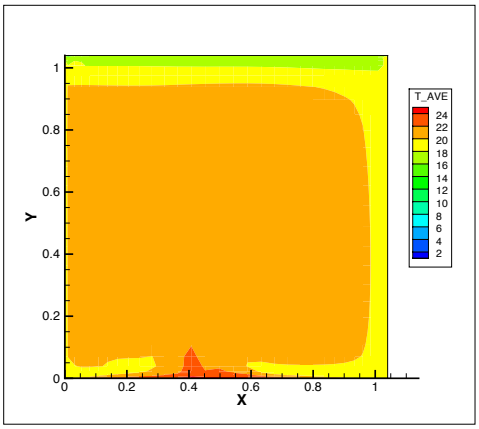


velocity (case 8744)

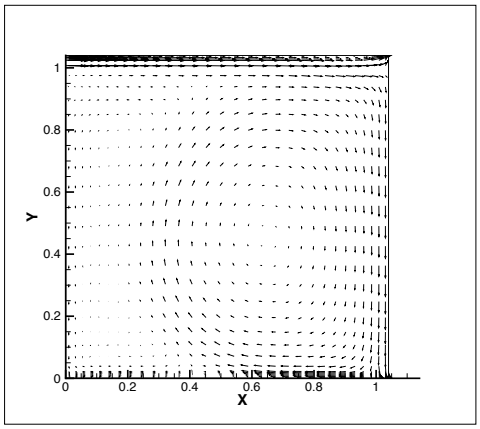
Figure 4.8 Temperature and velocity distribution for case `id` 200 and case `id` 8744

4.4.5 Simulate certain cases after interpolation

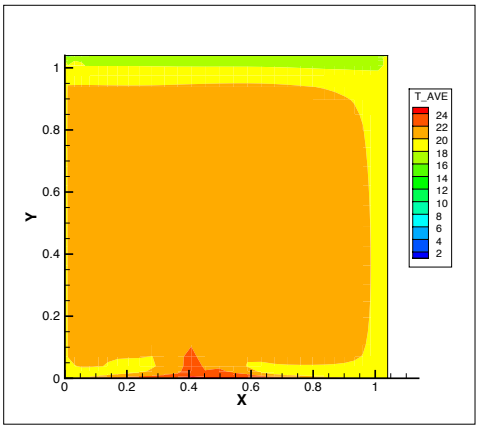
The previous experiment does not conclude any two cases can be interpolated. Below is an example case using three sets of simulation results, which cannot be directly interpolated, and the algorithm needs to use fluid solver



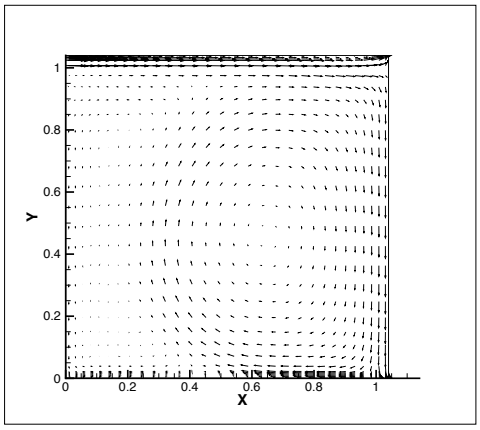
temperature (interpolated without simulation)



velocity (interpolated without simulation)



temperature (simulated by FFD)



velocity (simulated by FFD)

Figure 4.9 Temperature and velocity distribution for case `id` 344

to converge the interpolated result.

Case `id` 8430 and case `id` 20 are found by the algorithm linked by case `id` 188. Their outdoor temperature, inflow air speed and ventilation temperature are shown in Table 4.4.

case id	outdoor temperature	inflow speed	inflow temperature
8430	10.6C	0.466m/s	21.26164C
20	14.4C	0.465m/s	18.15218C
188	12.8C	0.465m/s	19.72866C

Table 4.4 Input parameters for case id 8430, 20 and 188

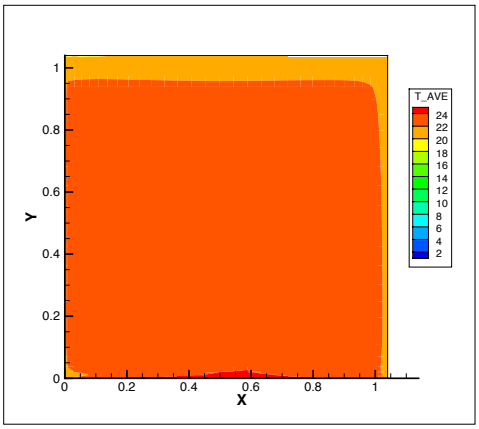
Figure 4.10 shows the temperature and velocity distribution result for case id 8430 and case id 20.

It is evident that case id 20 and case id 7430 are not similar. The temperature distribution and velocity field are different as can be seen on the figures. The interpolated result of case id 188 based on case id 200 and case id 8744, together with the actual result simulated by the fluid solver, are shown in Figure 4.11.

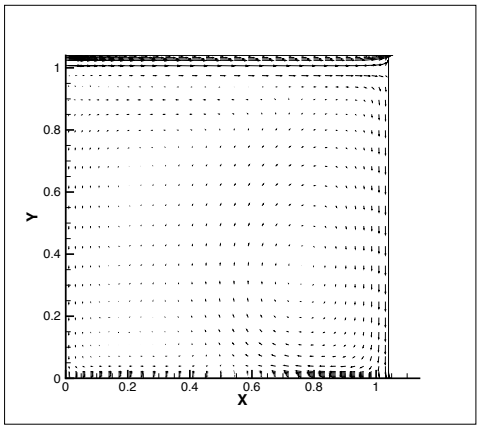
The algorithm will perform the fluid solver to converge the interpolated result to exact the solution. The averaged difference between the simulated result and the interpolated result is about 13 percent using the calculation method described in Appendix A. Converging from the interpolated result takes much fewer iterations (about 3 times faster) than doing from case 20 or case 8430.

4.4.6 Perform simulation on the remaining cases

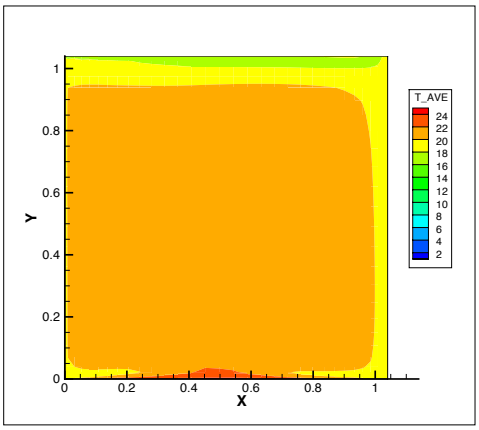
Certain cases are inherently far away from other cases. For instance, this can be seen in the previously mentioned case whose id is 20. Case 1664 is



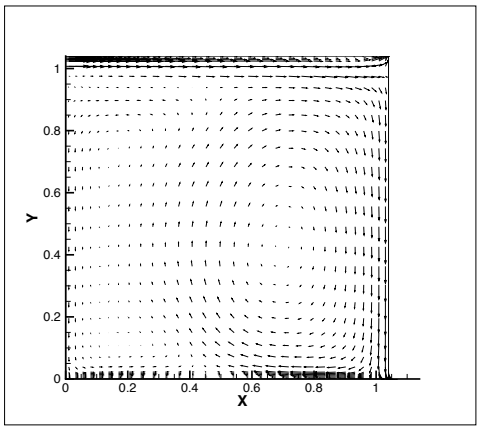
temperature (case 8430)



velocity (case 8430)



temperature (case 20)

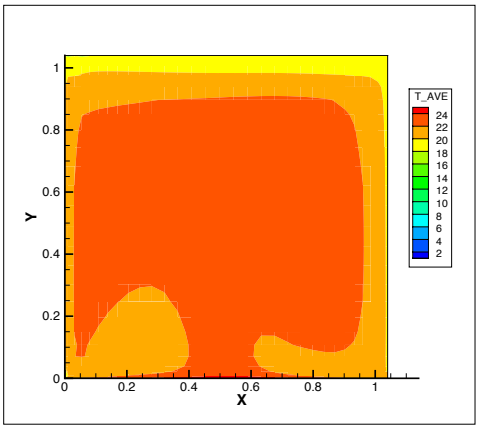


velocity (case 20)

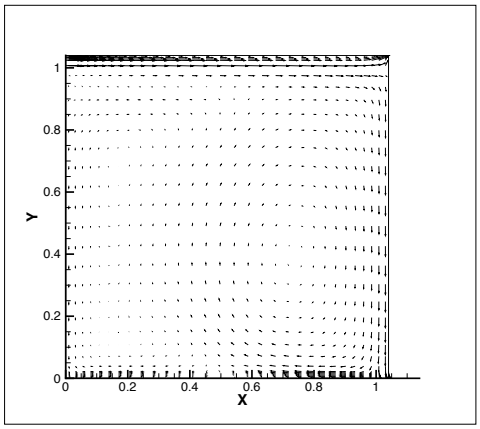
Figure 4.10 Temperature and velocity distribution for case `id` 8430 and case `id` 20

directly linked with case 20. Their outdoor temperature, inflow air speed and ventilation temperature are shown in Table 4.5.

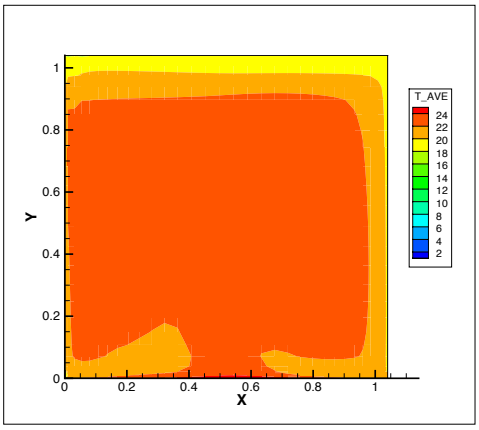
Since such cases are not similar by prediction, the only thing the solver can do is to converge from case 20 to case 1664. Figure 4.12 shows the temperature and velocity distribution result for case `id` 1664 and `id` 20



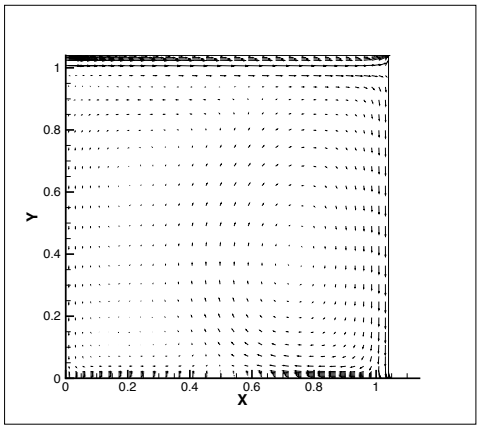
temperature (interpolated without simulation)



velocity (interpolated without simulation)



temperature (simulated by FFD)



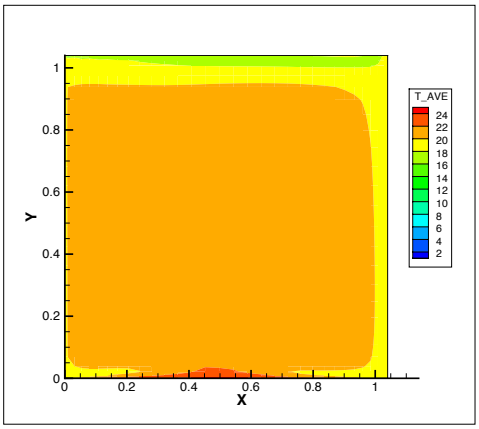
velocity (simulated by FFD)

Figure 4.11 Temperature and velocity distribution for case `id` 188

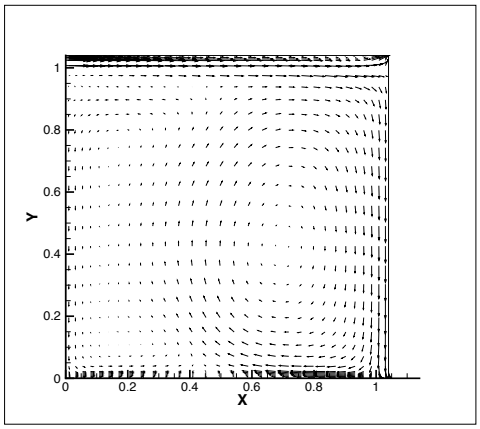
case `id`	outdoor temperature	inflow speed	inflow temperature
20	14.4C	0.465m/s	18.15218C
1664	14.4C	0.65947m/s	18C

Table 4.5 Input parameters for case `id` 20 and 1664

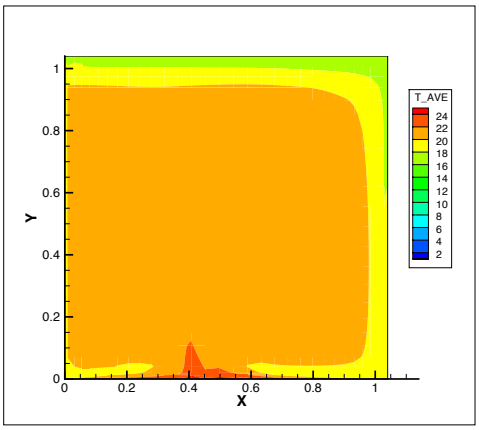
simulated by the FFD solver.



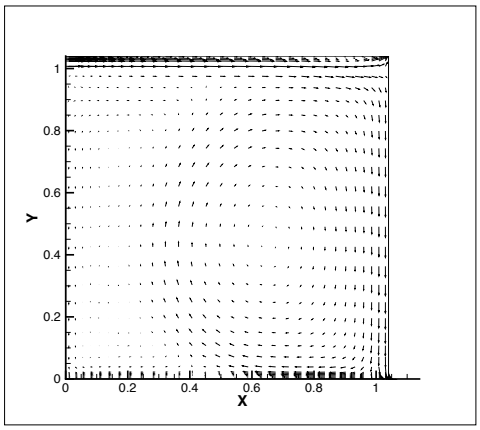
temperature (case 20)



velocity (case 20)



temperature (case 1664)



velocity (case 1664)

Figure 4.12 Temperature and velocity distribution for case `id` 20 and case `id` 1664

Case 1664 and case 20 were predicted to differ by 34 percent using the calculation method described in Appendix A. By reusing case 20 rather than starting from zero vectors as the initial state, the simulation only saves about 1/6 of the iterations, which is a quite limited speed-up.

4.5 Discussion

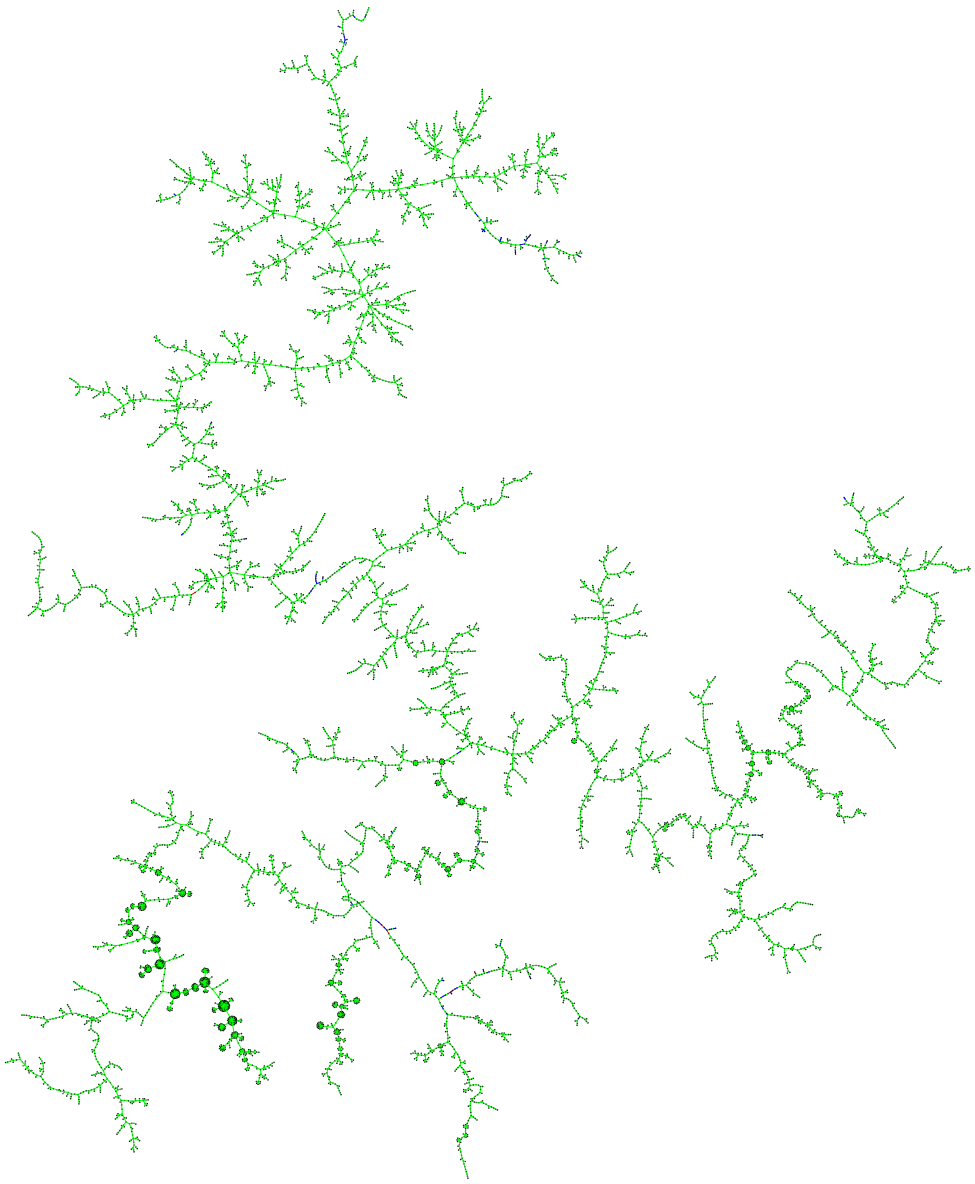


Figure 4.13 minimum spanning tree for 8760 cases

The minimum spanning tree of 8760 cases shown in Figure 4.13 illustrates the following:

First, it shows that it is important to conduct annual simulations rather than only extreme cases. The spanning tree shows that there are at least 100 cases that are extreme cases. The 100 cases differ from each other by greater than 16 percent. Thus, when using the extreme case's results to represent a problem, greater errors might be expected.

Also, there are 1000 cases differing from each other by greater than 10 percent. Thus, simple interpolation will not be accurate enough.

Finally, it is shows that more than half of the cases can be directly interpolated. Thus, tremendous speed-up can be achieved by interpolation which eliminates the need to simulate similar cases.

For instance, when zoomed in, Figure 4.14 shows that there are many points forming some clusters. It is assumed that the points within a cluster perform very similar results and the minor differences between these points could be neglected. Therefore, the clusters indicate that many cases can be predicted by simply using interpolation, and no fluid computation is needed.

If 8760 cases are calculated separately using the traditional way, the average difference would be 100 percent. Randomly or sequentially choosing a starting point from the simulated points would result in an average difference of 35 percent. In the planning strategy mentioned, as there are 8760 cases, there are $8760 - 1$ paths when the minimum spanning tree

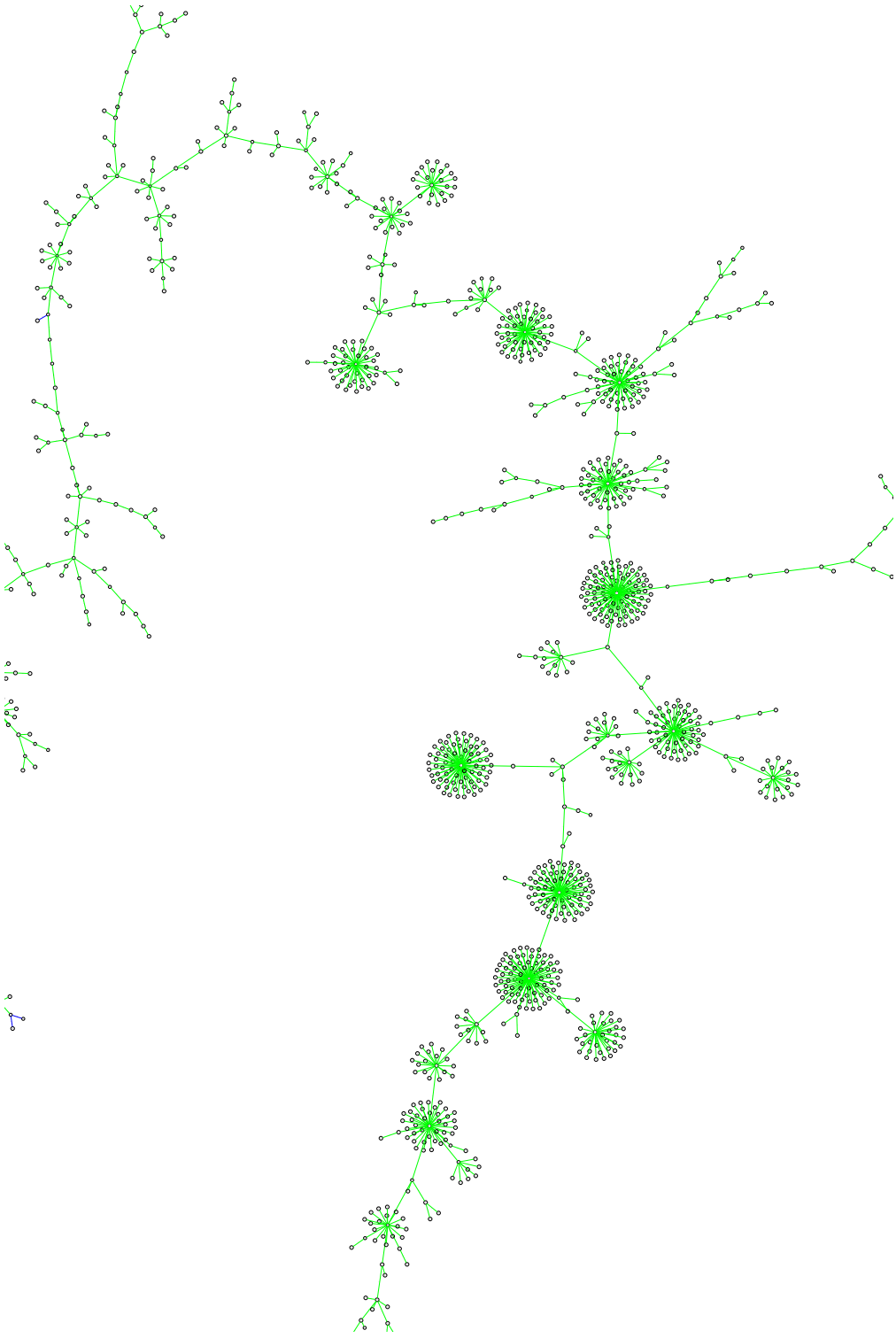


Figure 4.14 Similar cases forming clusters in the graph

is computed. For all the 8759 paths, 5423 paths have a distance below 1 percent, 7032 paths have a distance below 2 percent, 8340 paths have a distance below 3 percent, and only 81 paths have a distance greater than 5 percent. The average difference is much smaller than the traditional method or the random/sequential reusing method. See Figure 4.15.

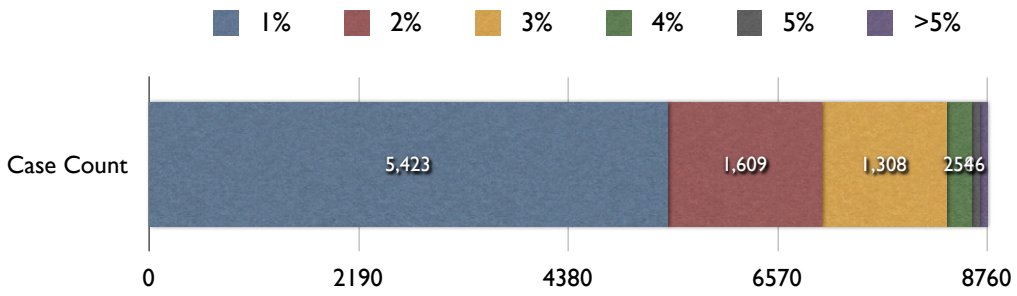


Figure 4.15 Distance statistics

In order to estimate the overall speed-up, this research uses Fluent [107] to perform the tests to estimate the converging speed in relation to case difference.

When simulating without using any previous case results, a typical simulation takes 428 iterations for this simple geometry using a 100×100 grid. On average, 15 iterations will be used for cases that differ by 1 percent, 53 iterations for 3 percent, 97 iterations for 5 percent, 185 iterations for 15 percent, and 368 iterations for 35 percent.

Thus, even without any interpolation, the minimum spanning tree based algorithm is 12.55 times faster than simulating without case reuse, or 10.79 times faster than simulating by reusing sequential or random case results.

By incorporating the multivariate interpolation method, the iteration required can be further limited; more than half of the 1 percent weighted edge can be done without fluid simulation, and for those cases simulated by fluid solver after interpolation (2 percent – 5 percent), much fewer iterations (about 2/3 of them) are also saved after interpolation. Thus, the overall speed-up is estimated to be 20 times faster than the traditional method.

The simulation algorithm will also be able to adjust itself during the run time to predict the actual distance (edge value) between each pair of the cases. When the simulation algorithm detects during actual simulation that two cases are not close or far away as predicted, it will be able to adjust the parameters in its simulation algorithms to correctly predict future cases. Online learning method, may also help when adjusting the parameter settings. When a few edges in the graph are changed, a very efficient linear time algorithm exists to update the minimum spanning tree [108].

4.6 Conclusions

With the algorithms introduced in this research, it is possible to form an optimized strategy to perform simulation by reusing previous cases instead of simulating similar cases over and over again.

The method can correctly choose similar cases as baseline, either using interpolation method to predict the case without any simulation, or by using nearest neighboring case as a starting point to do the simulation, thus dramatically reducing the simulation time.

This simulation strategy is self-adjustable. It will automatically adjust its own parameters based on existing cases, and thus make future predictions more accurate.

The algorithm can also be integrated with GPGPU optimization and FFD algorithm, and each optimization can be standalone. This gives great portability for the three-layered framework.

With the scheduling algorithm added to the three layered framework, this research is able to simulate annual hourly conditions as fast as executing two extreme cases using the traditional approach by multiplying the time factors of the three. Thus, annual approximate simulation can be possible within a short period of time, usually within an hour or several hours.

Chapter 5 Conclusions

5.1 Summary

In this research, three methods are used to allow great speed-up in CFD modeling. Table 5.1 shows the summary.

Method	Speed Ups
Reusing simulated cases	10 – 20x
Faster turbulent models	10 – 20x
GPGPU optimization	10 – 20x

Table 5.1 Three Layered framework. The estimated total speed-up is expected to be 1000 – 8000x

All these methods are integrated as a part of the framework developed.

This research employed techniques from the GPGPU computing world, modifying the low level mathematical procedures to speed up the simulation. This approach can and should be generic enough to be applied to different computational models (including FFD, k-epsilon, Large Eddy Simulation, etc) and achieve tremendous speed up over the traditional CPU method.

This research also incorporated an FFD model developed previously [81] with slight modification to perform steady case analysis. The time

averaged approach is reincorporated into the real-time dynamic simulation FFD model and have the averaged behavior of indoor airflow.

By utilizing the fact that many cases are similar in annual simulation, this research proposed to implement an efficient algorithm by reusing existing results to speed up the process.

With the three layered optimization strategies workflow, this research is able to simulate annual hourly conditions as fast as doing two extreme cases using traditional approaches by multiplying the time factors of the three. Thus, annual approximate simulation is made possible within a very short period of time, usually within an hour or several hours. All the methods are developed and can be applied separately. It is possible to just apply one or two methods if the simulation requires high accuracy by leaving only GPGPU computing and optimization strategies in this chapter. It is also possible to eliminate the GPGPU computing power if the casework size is very small, where the GPGPU method may be inefficient in such cases.

Also there are other variations to this method. For example, if the case problem size is relatively small, the GPGPU method may fail to achieve the expected speed. But it is possible to simulate the annual cases altogether since each hour's case's algorithm is exactly as other hour's case's, and thus can be highly parallelized to achieve great speed.

5.2 Precision and speed tradeoff

This research developed an efficient and informative fluid simulation framework to provide fast airflow simulation with an inevitable but nominal compromise in accuracy. Thus, there is a tradeoff between precision and speed in this research. Simulations with low precision will be faster so that it can give informative coarse result which can be used for conceptual design of indoor environment or realtime control of indoor environment. However, if the simulation is too coarse it will not meet the engineering precision requirement for certain applications such as modeling the airflow in a cleanroom [109].

The key aspect of this three layered framework is that it is able to adjust this tradeoff between precision and speed. All these three layer optimizations is designed as optional plugins. Each one is a standalone simulation optimization strategy, but also can be integrated with any other methods to make the framework as general-purposed as possible. Moreover, many parameters in the framework (such as the time-averaged residual threshold for terminating the FFD simulation, as well as the upper difference value threshold in the planning strategy for doing full fluid simulation) can be adjusted to balance between precision and simulation speed.

5.3 Limitation of this research

There are some potential shortcomings of this research, which can be improved in the future.

This strategy planning algorithm requires the input parameters for all hours in annual simulation to be present. In some applications, these data might be computed just in time, whereas a more sophisticated planning strategy might be needed.

Linear regression is sufficient for the simple case study presented in Chapter 4, where there is a close-to-linear relationship among output difference, input difference and converging speed. For more complicated cases, such linear relationship may not hold, and a more advanced machine learning method might be needed.

The strategy planning acceleration requires a confined input space, such as simulations of a fully mechanically controlled building enclosure. Only when there are many similar hourly solutions for the annual simulation will this method work. Notably, much work needs to be done to apply this strategy planning algorithm to simulation cases with sub-hourly variations or with large input space, such as those with natural ventilation or sun radiation.

- In natural ventilation simulation, wind gust speed and direction change from minute to minute, and indoor air is always in unsteady state. Thus, fine detailed transient simulation is required.

- In simulation cases that have solar radiation, the hour angle, the declination angle, and the latitude angle change each hour, which results in an input space with great variance.

Thus, for very complicated scenarios as these, this strategy planning algorithm may fail to achieve expected speed-up.

Interpolating a new outcome on previous results effectively assumes that there is a steady-state solution to the fluid state and that there are no historical effects from previous fluid states or from influencing conditions such as thermal mass. While for hourly simulations this may be acceptable, the application of this approach to sub-hourly simulations is quite limited, and requires more work.

Appendix A Vector Difference Definition

The difference calculation is of key importance in the thesis. It not only is the indicator and definition to determine whether two cases are far or near enough, but it also contributes to the accuracy and feasibility of the strategy planning algorithm.

The most trivial way to define the difference is via vector substraction. However, this method will be unfair for some corner cases, thus producing large error in the machine learning process.

Here's the traditional way to define the vector difference between case A and case B.

$$\Delta(A,B) = \text{average}(\delta(\mathbf{A},\mathbf{B})) = \sum_i \frac{1}{n}\delta(\mathbf{a}_i,\mathbf{b}_i) \qquad [A.1]$$

Here, $\mathbf{A}$ and $\mathbf{B}$ indicate the solution of case A and case B, where each solution is composed of an array of vectors. For example $\mathbf{a}_i$ indicates the i-th vector for case A.

The δ function could be calculated using the traditional method — vector difference. But vector difference will give the magnitude of the difference, so further scaling to percentage is required. Thus for example, the δ of a two-dimensional vector can be calculated as the following

$$\delta(\mathbf{a},\mathbf{b}) = \text{avg}\left(\frac{x_a - x_b}{x_b}, \frac{y_a - y_b}{y_b}\right) \quad [A.2]$$

where x_a is the x-coordinate of the **a** vector and y_a is the y-coordinate of the **a** vector. Vectors of higher dimension can be processed in the same way.

The avg function can be defined as any average function over scalars. This research uses the geometric means as its definition. Generally, if the numbers are $x_1, \ldots, x_n$, the geometric mean $\text{avg}(x_1, \ldots, x_n)$ satisfies

$$G = \text{avg}(x_1, \ldots, x_n) = \sqrt[n]{x_1 x_2 \cdots x_n} \quad [A.3]$$

and hence

$$\log G = \frac{1}{n}\sum_{i=1}^{n} \log x_i \quad [A.4]$$

For example, two-dimensional vector a is equal to $(2.0, 1.1)$, and another two-dimensional vector b is equal to $(2.1, 1.0)$. Then the difference of the two vector is

$$\delta(\mathbf{a},\mathbf{b}) = \text{avg}\left(\frac{0.1}{2.1}, \frac{0.1}{1.0}\right) = \text{avg}(0.05, 0.1) = 0.07 \quad [A.5]$$

Another example. A two dimentional vector a is equal to $(1.0, 1.1)$, and another two dimentional vector b is equal to $(2.1, 1.0)$. Then the difference of the two vector is

$$\delta(\mathbf{a},\mathbf{b}) = \text{avg}\left(\frac{1}{2}, \frac{0.1}{1.0}\right) = \text{avg}(0.5, 0.1) = 0.36 \quad [A.6]$$

Here vectors have larger difference, thus the calculated indicator is larger than the previous case. So this method seems to work fine.

The unfortunate aspect of this method is that the indicator will overflow in some cases, even if two vectors are quite similar.

For example, a two-dimensional vector a is equal to $(2.0, 0.1)$, and another two-dimensional vector b is equal to $(2.0, 0.00001)$. Then the difference of the two vectors is

$$\delta(\mathbf{a}, \mathbf{b}) = \text{avg}\left(\frac{0}{2}, \frac{0.99999}{0.00001}\right) = \text{avg}(0, 99999) = 70,000 \qquad \text{[A.7]}$$

Thus the previous indicator definition is inherently flawed. This method uses a better approach, which is described below.

For two-dimensional vector (the following definition can be easily extend to higher dimension as well) **a** and **b**, the δ function is defined as below:

$$\delta(\mathbf{a}, \mathbf{b}) = \text{avg}\left(\frac{\text{deg}}{180}, \left|\frac{|\mathbf{b}| - |\mathbf{a}|}{|\mathbf{b}| + |\mathbf{a}|}\right|\right) \qquad \text{[A.8]}$$

Where $|\mathbf{a}|$ and $|\mathbf{b}|$ is the length of vector **a** and **b** respectively. The deg is the degree which vector **a** and **b** formed in between.

Of course, to calculate the deg, $|\mathbf{a}|$ and $|\mathbf{b}|$, more mathematical calculation will be required:

$$|\mathbf{a}| = \sqrt{a_x^2 + a_y^2} \qquad |\mathbf{b}| = \sqrt{b_x^2 + b_y^2} \qquad \text{[A.9]}$$

and also needed to solve

$$\deg = \arccos\left(\frac{a_x * b_x + a_y * b_y}{|\mathbf{a}| \cdot |\mathbf{b}|}\right) \qquad [A.10]$$

Take the previous case (vector a is equal to $(2.0, 0.1)$, and another two-dimensional vector b is equal to $(2.0, 0.00001)$) as an example,

$$\delta(\mathbf{a}, \mathbf{b}) \approx \text{avg}\left(0, \frac{0}{4}\right) \approx \text{avg}(0, 0) \approx 0 \qquad [A.11]$$

It is easy to prove this method will have nice property that the calculation of δ will never overflow since

$$0 \leq \frac{\deg}{180} \leq 1 \qquad [A.12]$$

and

$$0 \leq \left|\frac{|\mathbf{b}| - |\mathbf{a}|}{|\mathbf{b}| + |\mathbf{a}|}\right| \leq 1 \qquad [A.13]$$

As a result, this new definition describes more precisely the difference of two vectors. The machine learning algorithm described in the thesis also shows much smaller relative error by using this definition.

Bibliography

[1] E. Djunaedy, *External coupling between building energy simulation and computational fluid dynamics* (2005)

[2] C. O. Negrao, *Conflation of computational fluid dynamics and building thermal simulation*. (University of Strathclyde Glasgow, 1995).

[3] C. O. Negrão, *Integration of computational fluid dynamics with building thermal and mass flow simulation*, Energy and Buildings **27** (1998), no. 2, 155–165

[4] J. Srebric, Q. Chen, and L. Glicksman, *A coupled airflow and energy simulation program for indoor thermal environmental studies* Tech. Rep., Massachusetts Inst. of Tech., Cambridge, MA (US) (2000).

[5] I. Beausoleil-Morrison, *The adaptive conflation of computational fluid dynamics with whole-building thermal simulation*, Energy and Buildings **34** (2002), no. 9, 857–871

[6] Z. Zhai, Q. Chen, P. Haves, and J. H. Klems, *On approaches to couple energy simulation and computational fluid dynamics programs*, Building and Environment **37** (2002), no. 8, 857–864

[7] E. Djunaedy, J. Hensen, and M. Loomans, *Toward external coupling of building energy and airflow modeling programs*, ASHRAE Transactions **109** (2003), no. 2, 771–787

[8] W. Cheng, C.-H. Liu, and D. Y. Leung, *Computational formulation for the evaluation of street canyon ventilation and pollutant removal performance*, Atmospheric Environment **42** (2008), no. 40, 9041–9051

[9] H. Sun, L. Zhao, and Y. Zhang, *Evaluating rng ke models using piv data for airflow in animal buildings at different ventilation rates*, Transactions of ASHRAE **113** (2007), no. 1, 358–365

[10] Y. Tominaga, A. Mochida, S. Murakami, and S. Sawaki, *Comparison of performance of various revised k-epsilon models applied to cfd analysis of flowfield around a high-rise building*, Journal of architecture planning and environmental engineering (2002), no. 556, 47–54

[11] P. Neofytou *et al.*, *Cfd simulations of the wind environment around an airport terminal building*, Environmental Modelling & Software **21** (2006), no. 4, 520–524

[12] R. Panneer Selvam, *Computation of flow around texas tech building using k-epsilon and kato-launder k-epsilon turbulence model*, Engineering structures **18** (1996), no. 11, 856–860

[13] F. Deserno, G. Hager, F. Brechtefeld, and G. Wellein, *Basic optimization strategies for cfd-codes* Tech. Rep., Technical report, Regionales Rechenzentrum Erlangen (2002).

[14] H. L. Stone, *Iterative solution of implicit approximations of multidimensional partial differential equations*, SIAM Journal on Numerical Analysis **5** (1968), no. 3, 530–558

[15] S. A. ORSZAGT, *Analytical theories of turbulence*, J. Fluid Mech **41** (1970), no. part 2, 363–386

[16] J. Habich, *Improving computational efficiency of lattice boltzmann methods on complex geometries* (2006)

[17] M. Herlihy and N. Shavit *The art of multiprocessor programming*, in *Annual ACM Symposium on Principles of Distributed Computing: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing* (2006), pp. 1–2.

[18] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*. (Morgan Kaufmann, 2012).

[19] G. E. Moore and others, *Cramming more components onto integrated circuits* (McGraw-Hill, 1965).

[20] Wikipedia, *Transistor count — Wikipedia, the free encyclopedia* (2013a).

[21] G. M. Amdahl *Validity of the single processor approach to achieving large scale computing capabilities*, in *Proceedings of the April 18-20, 1967, spring joint computer conference* (ACM, 1967), pp. 483–485.

[22] D. P. Rodgers *Improvements in multiprocessor system design*, in *ACM SIGARCH Computer Architecture News* (IEEE Computer Society Press, 1985), pp. 225–231.

[23] Intel, *Intel multi-core processor architecture development backgrounder* Tech. Rep., Intel, Tech. Rep (2005).

[24] K. Quinn, J. Yang, and V. Turner, *The next evolution in enterprise computing: The convergence of multicore x86 processing and 64-bit operating systems* Tech. Rep., AMD (2005).

[25] N. Brookwood, *Amd fusion family of apus: Enabling a superior, immersive pc experience*, Insight **64** (2010), no. 1, 1–8

[26] J. Casazza, *First the tick, now the tock: Intel microarchitecture (nehalem)*, Intel Corporation (2009)

[27] A. Corrigan, F. Camelli, R. Löhner, and J. Wallin, *Running unstructured grid based cfd solvers on modern graphics hardware*, AIAA paper **4001** (2009), 22–25

[28] W. Zuo and Q. Chen, *Simulations of air distributions in buildings by ffd on gpu*, HVAC&R Research **16** (2010a), no. 6, 785–798

[29] W. Zuo and Q. Chen, *Fast and informative flow simulations in a building by using fast fluid dynamics model on graphics processing unit*, Building and environment **45** (2010b), no. 3, 747–757

[30] C. Nvidia, *Compute unified device architecture programming guide* (2007)

[31] A. AMD, *Stream computing-technical overview* Tech. Rep., AMD, Tech. Rep (2008).

[32] N. Thibieroz and C. Cebenoyan *Directcompute performance on dx11 hardware*, in *Proceedings of the Game Developers Conference* (2010).

[33] K. Kothapalli *et al. A performance prediction model for the cuda gpgpu platform*, in *High Performance Computing (HiPC), 2009 International Conference on* (IEEE, 2009), pp. 463–472.

[34] S. Lee, S.-J. Min, and R. Eigenmann, *Openmp to gpgpu: a compiler framework for automatic translation and optimization*, ACM Sigplan Notices **44** (2009), no. 4, 101–110

[35] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka *Bandwidth intensive 3-d fft kernel for gpus using cuda*, in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for* (IEEE, 2008), pp. 1–11.

[36] T. D. Han and T. S. Abdelrahman *hi cuda: a high-level directive-based language for gpu programming*, in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units* (ACM, 2009), pp. 52–61.

[37] A. Munshi, *Opencl 1.0 specification*, Khronos OpenCL Working Group (2009)

[38] Apple, *Technology brief: Opencl taking the graphics processor beyond graphics* Tech. Rep., (2009).

[39] A. A. P. Processing, *Sdk (formerly ati stream)*, Website. Online available at http://developer. amd. com/gpu/AMDAPPSDK/Pages/default. aspx (2005)

[40] N. Whitehead and A. Fit-Florea, *Precision & performance: Floating point and ieee 754 compliance for nvidia gpus*, rn (A+ B) **21** (2011), 1–1874919424

[41] S. Kestur, J. D. Davis, and O. Williams *Blas comparison on fpga, cpu and gpu*, in *VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on* (IEEE, 2010), pp. 288–293.

[42] D. Helbing and P. Molnar, *Social force model for pedestrian dynamics*, Physical review E **51** (1995), no. 5, 4282

[43] D. Helbing, I. Farkas, and T. Vicsek, *Simulating dynamical features of escape panic*, Nature **407** (2000), no. 6803, 487–490

[44] J. Stam *Stable fluids*, in *Proceedings of the 26th annual conference on computer graphics and interactive techniques* (ACM Press/Addison-Wesley Publishing Co., 1999), pp. 121–128.

[45] M. J. Harris and A. Lastra *Real-time cloud rendering*, in *Computer Graphics Forum* (Wiley Online Library, 2001), pp. 76–85.

[46] O.-Y. Song, H. Shin, and H.-S. Ko, *Stable but nondissipative water*, ACM Transactions on Graphics (TOG) **24** (2005), no. 1, 81–97

[47] M. Harris, *Fast fluid dynamics simulation on the gpu*, GPU gems **1** (2004), 637–665

[48] W. Zuo and Q. Chen, *Validation of fast fluid dynamics for room airflow*, IBPSA Building Simulation (2007)

[49] W. Zuo and Q. Chen, *Real-time or faster-than-real-time simulation of airflow in buildings*, Indoor Air **19** (2009), no. 1, 33–44

[50] H. Jasak, A. Jemcov, and Z. Tukovic *Openfoam: A c++ library for complex physics simulations*, in *International Workshop on Coupled Methods in Numerical Dynamics, IUC, Dubrovnik, Croatia* (2007), pp. 1–20.

[51] Wikipedia, *Performance tuning — Wikipedia, the free encyclopedia* (2013b).

[52] Apple, *Instruments user guide* Tech. Rep., (2013).

[53] T. Beauchamp and D. Weston, *Dtrace: The reverse engineerÃ-s unexpected swiss army knife*, Blackhat Europe (2008)

[54] M. Richard, *Solaris" Performance And Tools: Dtrace And Mdb Techniques For Solaris 10 And Opensolaris*. (Pearson Education India, 2007).

[55] M. D. Gunzburger and R. A. Nicolaides, *Incompressible computational fluid dynamics: trends and advances*. (Cambridge University Press, 1993).

[56] A. R. Gallego and L. M. Gonzalez, *Adfc navier-stokes solver* (2009).

[57] J. R. Shewchuk, *An introduction to the conjugate gradient method without the agonizing pain* (Carnegie Mellon University, Pittsburgh, PA, 1994).

[58] V. Rotkin and S. Toledo, *The design and implementation of a new out-of-core sparse cholesky factorization method*, ACM Transactions on Mathematical Software (TOMS) **30** (2004), no. 1, 19–46

[59] Wikipedia, *Conjugate gradient method — Wikipedia, the free encyclopedia* (2013c).

[60] R. Barrett *et al.*, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. (SIAM, Philadelphia, PA, 1994).

[61] M. Harris, *Optimizing parallel reduction in cuda*, NVIDIA Developer Technology **6** (2007)

[62] OpenFOAM, *Openfoam source code* (2013).

[63] D. C. Wilcox, *Turbulence modeling for CFD*, volume 2. (DCW industries La Canada, 1998).

[64] S. R. .P, *Modeling of turbulent flows and boundary layer, computational fluid dynamics* Tech. Rep., (2010).

[65] A. J. Lew, G. C. Buscaglia, and P. M. Carrica, *A note on the numerical treatment of the k-epsilon turbulence model**, International Journal of Computational Fluid Dynamics **14** (2001), no. 3, 201–209

[66] M. C. Sukop and D. T. Thorne Jr, *Lattice Boltzmann modeling: an introduction for geoscientists and engineers*. (Springer Publishing Company, Incorporated, 2007).

[67] S. Succi and J. M. Yeomans, *The lattice boltzmann equation for fluid dynamics and beyond*, Physics Today **55** (2002), no. 12, 58–60

[68] A. J. Wagner, *A practical introduction to the lattice boltzmann method*, Adt. notes for Statistical Mechanics **463** (2008), 663

[69] S. Chen and G. D. Doolen, *Lattice boltzmann method for fluid flows*, Annual review of fluid mechanics **30** (1998), no. 1, 329–364

[70] V. Heuveline and J. Latt, *The openlb project: An open source and object oriented implementation of lattice boltzmann methods*, International Journal of Modern Physics C **18** (2007), no. 4, 627-634

[71] V. Heuveline, M. Krause, and J. Latt, *Towards a hybrid parallelization of lattice boltzmann methods*, Computers and Mathematics with Applications (to appear)

[72] V. Heuveline and J.-P. Weiss, *A parallel implementation of a lattice boltzmann method on the clearspeed advance accelerator board* Tech. Rep. 07-03, (2007).

[73] D. Groen *et al.*, *Analysing and modelling the performance of the hemelb lattice-boltzmann simulation environment*, Journal of Computational Science (2013)

[74] S. Roy, R. M. Kelso, and A. J. Baker, *An efficient cfd algorithm for the prediction of contaminant dispersion in room air motion*, ASHRAE Transactions-American Society of Heating Refrigerating Airconditioning Engin **100** (1994), no. 2, 980–987

[75] P. Jones and G. Whittle, *Computational fluid dynamics for building air flow prediction-current status and capabilities*, Building and Environment **27** (1992), no. 3, 321–338

[76] L. B. Lucy, *A numerical approach to the testing of the fission hypothesis*, The astronomical journal **82** (1977), 1013–1024

[77] R. A. Gingold and J. J. Monaghan, *Smoothed particle hydrodynamics-theory and application to non-spherical stars*, Monthly notices of the royal astronomical society **181** (1977), 375–389

[78] J. J. Monaghan, *Smoothed particle hydrodynamics*, Annual review of astronomy and astrophysics **30** (1992), 543–574

[79] S. Minzel, *Smoothed particle hydrodynamics und ihre anwendung auf akkretionsscheiben* (1996)

[80] W. Zuo and Q. Chen, *Validation of a fast-fluid-dynamics model for predicting distribution of particles with low stokes number* Tech. Rep., Ernest Orlando Lawrence Berkeley National Laboratory, Berkeley, CA (US) (2011).

[81] W. Zuo, J. Hu, and Q. Chen, *Improvements in ffd modeling by using different numerical schemes*, Numerical Heat Transfer, Part B: Fundamentals **58** (2010), no. 1, 1–16

[82] W. Zuo and Q. Chen *Improvements on the fast fluid dynamics model for indoor airflow simulation*, in *4th National Conference of IBPSA-USA (SimBuild 2010), New York, NY* (2010).

[83] W. Zuo and M. Jin, *Reduction of numerical diffusion in the ffd model*, Engineering Applications of Computational Fluid Mechanics **6** (2012), no. 2, 234–247

[84] J. Morshed and J. J. Kaluarachchi, *Application of artificial neural network and genetic algorithm in flow and transport simulations*, Advances in Water Resources **22** (1998), no. 2, 145–158

[85] S. S. Sablani, *A neural network approach for non-iterative calculation of heat transfer coefficient in fluid–particle systems*, Chemical Engineering and Processing: Process Intensification **40** (2001), no. 4, 363–369

[86] R. Benning, T. Becker, and A. Delgado, *Initial studies of predicting flow fields with an ann hybrid*, Advances in Engineering Software **32** (2001), no. 12, 895–901

[87] R. Masini, E. Padovani, M. Ricotti, and E. Zio, *Dynamic simulation of a steam generator by neural networks*, Nuclear engineering and design **187** (1999), no. 2, 197–213

[88] W. H. Shayya and S. S. Sablani, *An artificial neural network for non-iterative calculation of the friction factor in pipeline flow*, Computers and electronics in agriculture **21** (1998), no. 3, 219–228

[89] A. Kelkar, R. Mahajan, and R. Sani, *Real time physiconeural solutions for mocvd*, Journal of heat transfer **118** (1996), no. 4, 814–821

[90] N. Smaoui and S. Al-Enezi, *Modelling the dynamics of nonlinear partial differential equations using neural networks*, Journal of Computational and Applied Mathematics **170** (2004), no. 1, 27–58

[91] N. Mai-Duy and T. Tran-Cong, *Numerical solution of navier–stokes equations using multiquadric radial basis function networks*, International Journal for Numerical Methods in Fluids **37** (2001), no. 1, 65–86

[92] P. N. Williams, *The artificial neural network solution to double diffusive convection equations using spectral methods*. (Rice University, 2001).

[93] T. Kozek and T. Roska, *A double time-scale cnn for solving two-dimensional navier-stokes equations*, International journal of circuit theory and applications **24** (1996), no. 1, 49–55

[94] N. Smaoui and A. A. Garrouch, *A new approach combining karhunen-loéve decomposition and artificial neural network for estimating tight gas sand permeability*, Journal of Petroleum Science and Engineering **18** (1997), no. 1, 101–112

[95] T. Belytschko *et al.*, *Meshless methods: an overview and recent developments*, Computer methods in applied mechanics and engineering **139** (1996), no. 1, 3–47

[96] A. Katz and A. Jameson, *Edge-based meshless methods for compressible flow simulations*, AIAA paper **699** (2008)

[97] Z. El Zahab, E. Divo, and A. Kassab, *A meshless cfd approach for evolutionary shape optimization of bypass grafts anastomoses*, Inverse Problems in Science and Engineering **17** (2009), no. 3, 411–435

[98] Z. Shang, *Application of artificial intelligence cfd based on neural network in vapor–water two-phase flow*, Engineering Applications of Artificial Intelligence **18** (2005), no. 6, 663–671

[99] Wikipedia, *Minimum spanning tree — Wikipedia, the free encyclopedia* (2013e).

[100] R. L. Graham and P. Hell, *On the history of the minimum spanning tree problem*, Annals of the History of Computing **7** (1985), no. 1, 43–57

[101] R. C. Prim, *Shortest connection networks and some generalizations*, Bell system technical journal **36** (1957), no. 6, 1389–1401

[102] J. B. Kruskal, *On the shortest spanning subtree of a graph and the traveling salesman problem*, Proceedings of the American Mathematical society **7** (1956), no. 1, 48–50

[103] D. Blay, S. Mergui, and C. Niculae, *Confined turbulent mixed convection in the presence of a horizontal buoyant wall jet*, ASME-PUBLICATIONS-HTD **213** (1993), 65–65

[104] C. Cortes and V. Vapnik, *Support vector machine*, Machine learning **20** (1995), no. 3, 273–297

[105] I. T. Jolliffe, *Principal component analysis*, volume 487. (Springer-Verlag New York, 1986).

[106] B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani, *Least angle regression*, The Annals of statistics **32** (2004), no. 2, 407–499

[107] ANSYS, *Fluent 6.2 manual* (2005)

[108] S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani, *Algorithms* (2006), Exercise 5.22

[109] W. Whyte, M. Hejab, W. Whyte, and G. Green, *Experimental and cfd airflow studies of a cleanroom with special respect to air supply inlets*, International Journal of Ventilation **9** (2010), no. 3, 197–209

Index