UNIVERSITÉ DE GENÈVE
Département d'informatique

FACULTÉ DES SCIENCES
Professeur B. Chopard

From a lattice Boltzmann model to a parallel and reusable implementation of a virtual river

THÈSE

présentée à la Faculté des sciences de Université de Genève
pour obtenir le grade de Docteur ès sciences, mention informatique

par

Alexandre DUPUIS

de
Cologny (GE)

Thèse N° 3356

GENÈVE
2002

La Faculté des sciences, sur le préavis de Messieurs B. CHOPARD, professeur adjoint et directeur de thèse (Département d'informatique), S. SUCCI, professeur (ICA, CNR - Rome, Italie), P. KUONEN, docteur (Haute Ecole Valaisanne - Sion) et S. MARCHAND-MAILLET, docteur (Département d'informatique) autorise l'impression de la présente thèse, sans exprimer d'opinion sur les propositions qui y sont énoncées.

Genève, le 20 juin 2002

Thèse -3356-

Le Doyen, Jacques Weber

"When Gluck attacks Hollywood nobody feels good"
Couleur 3

Contents

Abstract

This thesis proposes a numerical model for simulating sediment transport phenomena. We call it a virtual river model. It is actually composed of a fluid and of sediments.

The so-called lattice Boltzmann models describe the fluid. These models are defined for the document to be self-contained. This description is a good starting point for beginners but can also serves as a reference document for advanced readers. Boundary and flow settlement conditions are reviewed and a new boundary condition, which is mass conserving, is proposed and compared to existing ones conditions. To reach high Reynolds number, we use the so-called Smagorinsky model of turbulence which is clearly presented in the present lattice Boltzmann context. Mesh refinement techniques are reviewed and a more general algorithm is proposed.

We define rules describing the behavior of the sediments. Then, we focus our attention on two applications: scour formation under submarine pipelines and meanders in rivers. Results of the first application are in agreement with the literature. We give an explanation at the mesoscopic level of the meandering process. Our preliminary results confirm our theory.

To produce reusable and efficient implementations of our virtual rivers, we propose a parallel object-oriented environment for lattice based simulations called PELABS. Its performances are measured and analyzed. Due to its object-oriented architecture, we exhibit that, at worst, it is two times slower than a procedural implementation. Moreover, under certain conditions, it can be faster.

Acknowledgment

During this work, I was in contact with people who helped me in many ways. Let me try to thank them without, I hope, forgetting anyone.

My first thanks go to Bastien Chopard who offered me a motivating and fruitful environment to produce the present work. I will also keep a nice souvenir of the hikes we did in the Alpes. I thank: Paul Albuquerque for having patiently read and proposed corrections on the draft and also for the numerous coffees he lost after our dart games, Stefan Marconi for our many interesting discussions and for teaching me the backgammon rules (although today the student has surpassed the master), Alexandre Masselot for introducing me to the rudiments of lattice Boltzmann models, Patrick Roth who among others specially came to the EPFL four years ago to have a simple but unforgettable coffee, Lori Petrucci who broke my car antenna, Stéphane Marchand-Maillet who kindly accepted to be the local member of my jury, Shelby Perreira for his tasty chicken tandoori, Jean-Luc Falcone for teaching me the Jony Cash song and all the other members of the CUI.

I also thank Pierre Kuonen who helped me to start this project during my stay at the EPFL and for being the national member of my jury. I really do appreciate the presence of that Sauro Succi, who come specially from Rome, for accepting to be the international member of my jury.

For their suggestions, I thank: Franco Bagnoli and his colleagues for suggesting to focus our attention on the scour formation under submarine pipelines and Drona Kandhai for motivating to spend times on mesh refinement.

Despite they are far away from science, I would like to thank some of my friends: Vincent, Fred, Valérie, Philippe, Sandrine, Laurence, Sofian and Joséphine. They were present when I really needed them and they kept me in touch with the economical reality of Geneva.

Finally, I thank all my family from Geneva and Valais and especially my father for giving me the opportunity of being here, and my wonderful Viviane whose radiant smile made and, I hope, will make my life going well.

Résumé

Ces quelques pages, écrites dans la langue de Molière, résument le contenu de cette thèse de doctorat. Nous avons pris le parti de présenter les idées et résultats de manière à effectuer un survol. Le lecteur intéressé par des références aux publications ainsi que par des explications plus précises est invité à se rendre aux chapitres correspondant.

Introduction

Ce document résume le travail effectué dans un but: modéliser numériquement une rivière en utilisant des modèles de Boltzmann sur réseaux. L'accomplissement de ce dessein passe par l'étude de différents aspects de ces modèles ainsi qu'à leurs implémentations. La suite de ce résumé leurs est consacrée.

Fluides de Boltzmann

Description

Un fluide de Boltzmann évolue sur une grille (ou réseau) régulière, d-dimensionnelle discrétisant le domaine réel. Les sites $\mathbf{r}$ de cette grille sont reliés par z liens et ont, par conséquent, z voisins. On définit un modèle $DdQ(z+1)$ comme un modèle opérant sur une grille d-dimensionnelle, comportant z voisins et une position de repos.

Ce fluide est constitué de pseudo-particules (ou champs) se déplaçant sur les liens du réseau. Lorsque celles-ci se rencontrent sur un site, une collision à lieu. Si l'opérateur de collision est bien choisi, on montre que les propriétés d'un fluide (i.e. les équations de Navier-Stokes et de continuité) sont reproduites.

Conditions de bords

La dynamique présentée ci-dessus n'est plus valable pour un site de bord. En effet, pour ces derniers, certains champs sont manquant car ils représenteraient des informations venant du solide. Des conditions spéciales, nommées conditions de bords, sont nécessaires.

Dans ce document, nous avons répertorié et disséqué les conditions de bords les plus utilisées. Cela nous a permis de mettre en lumière certains défauts comme la non conservation de la masse lors de l'application de certaines d'entre elles. Pour y remédier, nous proposons une nouvelle condition qui conserve la masse et permet d'appliquer la dynamique jusqu'au bord.

Etablissement de l'écoulement

L'établissement de l'écoulement du fluide est un tâche difficile. En effet, les pseudo-particules sont initialement disposées dans un état d'équilibre. Cette configuration représente un fluide au repos. Notons que cet état de repos est choisi car il est le seul état stable commun à toutes simulations. Ainsi, le fluide doit adéquatement être accélérer afin de produire l'écoulement désiré.

Les différentes techniques existantes sont passées en revue. Leurs défauts et qualités sont mis en évidence.

Modèle de turbulence

Lorsque la viscosité ν imposée par les paramètres du système est trop faible, des instabilités numériques peuvent apparaître. On peut voir ces instabilités comme des problèmes liés à la discrétisation. En effet, on montre que plus la turbulence augmente plus les effets des petits tourbillons sont importants. On comprend intuitivement que des instabilité numériques apparaissent si ces petits tourbillons ne peuvent être modélisés (i.e. ils sont plus petits qu'une maille du réseau).

Au lieu d'augmenter indéfiniment la taille de la grille, on peut utiliser un modèle de turbulence qui se superpose au modèle de Boltzmann. Parmi les modèles connus, nous choisissons d'utiliser celui de Smagorinsky. L'idée principale est de considérer un accroissement local de la viscosité nommé ν_t qui dépend essentiellement de l'activité locale du fluide.

Pour illustrer l'utilisation du modèle de Smagorinsky, la figure 1 montre le rapport ν_t/ν lorsqu'on considère un écoulement turbulent autour d'un cylindre. Donc, plus le rapport est grand, plus la viscosité est localement augmentée.

Quelques simulations

Afin de valider les différents concepts présentés, un ensemble d'exemples simples ont été considérés tout au long de la présentation du modèle de fluides de Boltzmann.

De plus, deux simulations plus compliquées sont considérées. La première est l'écoulement d'un fluide au-dessus d'une marche. Les propriétés de l'écoulement, largement étudiés dans la littérature, sont raisonnablement reproduites. Puis, nous mesurons la force agissant sur un cylindre que l'on oppose à un écoulement

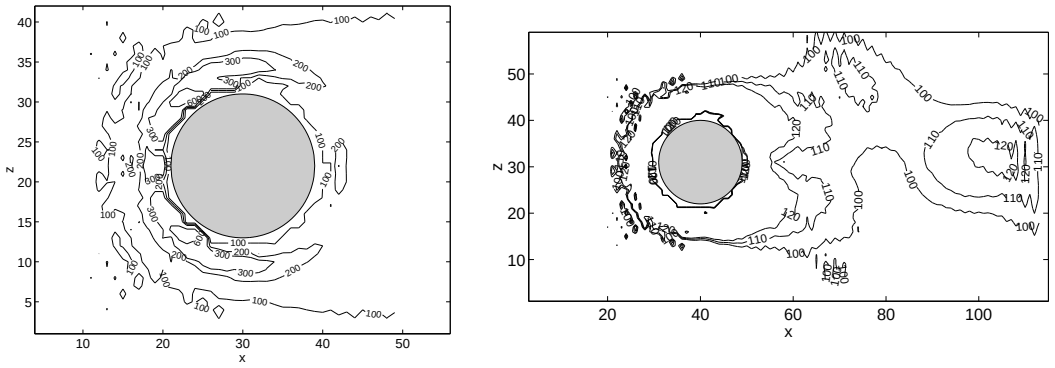


Figure 1: Rapport ν_t/ν d'un écoulement de Boltzmann couplé à un modèle de turbulence de Smagorinsky. On considère un modèle D3Q19 autour d'un cylindre de diamètre $D = 20$. Le domaine torique est composé de $N_x \times N_y \times N_z = 140 \times 5 \times 60$ sites. Les graphes gauche et droit montrent deux vues différentes à des échelles différentes correspondant au même écoulement. L'écoulement est turbulent ($Re = 10^5$) et il s'écoule de gauche à droite.

peu, et de plus en plus turbulent. A nouveau, les propriétés attendues sont reproduites.

Raffinement de grille

Motivations

Lorsque l'on simule l'écoulement d'un fluide de Boltzmann, on est à la recherche de résultats précis et obtenus en un temps minimal. On se trouve ainsi confronté à une contradiction compte tenu du fait que la précision des résultats décroît avec la discrétisation mais que, inversement, le temps de calcul croît avec la discrétisation. Pour remédier à ce problème, on peut, où une grande précision est souhaitée, raffiner la grille et considérer une grille plus grossière lorsque la précision influence moins les résultats. Un gain de temps est ainsi à espérer.

Différents modèles

Nous avons répertorié deux modèles permettant de raffiner localement une grille. Ils sont présentés en détails, leur algorithme est clairement expliqué et leurs défauts sont exhibés. L'un fait une supposition erronée et induit un algorithme simpliste. L'autre présente une singularité numérique.

Nous proposons une nouvelle approche qui ne présente aucune singularité numérique. Une validation considérant un écoulement simple indique que les résultats obtenus sont essentiellement aussi précis que ceux obtenus par des

méthodes existantes. Notre méthode est la plus précise autour de la singularité numérique.

Accélération de l'établissement de l'écoulement

On s'aperçoit aisément que l'établissement de l'écoulement est d'autant plus long que la grille est grande. Ainsi, établir l'écoulement sur une grille grossière que l'on utilise pour initialiser celui d'une grille plus fine permet de gagner du temps.

Un algorithme permettant d'accélérer le processus d'établissement de l'écoulement consiste à considérer une cascade de grille allant d'une grille très grossière à une grille très fine. Cette dernière est la grille de l'application, i.e. celle sur laquelle on désire établir l'écoulement.

Nous utilisons cet algorithme dans un écoulement simple et observons une accélération de 100%.

Un modèle de rivières virtuelles

Description

Afin de créer ce que nous nommons *rivière virtuelle*, nous ajoutons au fluide décrit ci-dessus des sédiments. Ils évoluent sur le même réseau que les particules de fluides et leur comportement est dicté par un ensemble de règles qui sont:

1. **Erosion:** selon une probabilité d'érosion, une particule est éjectée verticalement, d'un site.

2. **Transport:** en fonction de la vitesse du fluide et d'une vitesse de chute de sédiments, les particules en mouvement sont envoyées dans le site correspondant.

3. **Déposition:** un sédiment, touchant le sol ou un dépôt de sédiments, se dépose en se solidifiant.

4. **Eboulement:** une règle d'éboulement à lieu si localement l'angle entre des piles de sédiments voisines est plus grand que l'angle de repos maximum fixé.

Notons que l'application des règles 1 et 2, revient à éroder les sédiments seulement si le fluide est suffisamment puissant.

Applications

Nous avons principalement focalisé notre attention sur deux applications. La première est la formation d'une excavation sous un tuyau sous-marin et la deuxiè-

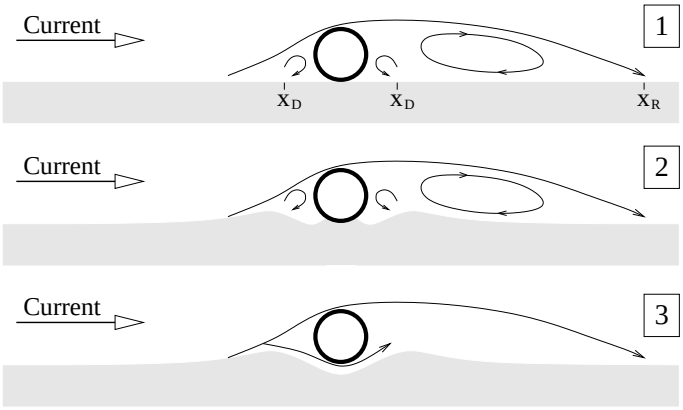


Figure 2: Trois étapes principales du démarrage du processus de formation d'une excavation sous un tuyau sous-marin soumis à un courant unidirectionnel. (1) A cause du courant, trois tourbillons apparaissent dans le voisinage du tuyau. (2) Les deux petits tourbillons, avant et après le tuyau, commencent tous deux à creuser un trou. (3) Après un certain temps, les trous se rencontrent sous le tuyau et le processus d'excavation proprement dit démarre. Les quantités x_D et x_R dénotent des points de réattachement.

me est la formation et la progression des méandres de rivières. Nous présentons maintenant ces applications et les résultats obtenus.

Excavation sous un tuyau sous-marin

La figure 2 explique le démarrage du processus de formation d'une excavation sous un tuyau sous-marin.

Lorsque les trous creusés de chaque côté du tuyau se rejoignent (dernière étape de la figure 2), la pression est grande entre le tuyau et le sol, étant donné que la profondeur de l'excavation est petite. Un forte pression implique une vitesse localement accélérée et donc une forte érosion. Cette dernière, comme la pression, va réduire plus la profondeur augmente. Ainsi, après un certain temps, la profondeur se stabilise et atteint un état stable.

Nous présentons maintenant les résultats obtenus considérant une géométrie initiale présentée dans la figure 3.

Les tourbillons obtenus par nos simulations son mis en évidence dans la figure 4. Ils sont semblables à ceux de la figure 2.

Les étapes caractéristiques de l'évolution du processus de formation de l'excavation sont présentées dans la figure 5.

Dans la figure 6, l'évolution temporelle du processus d'excavation est mesurée et comparée avec succès, avec le comportement exponentiel attendu.

On montre que la profondeur du trou est une fonction croissante du diamètre du tuyau. De plus, une fois l'excavation stabilisée, le tuyau peut s'y déposer et

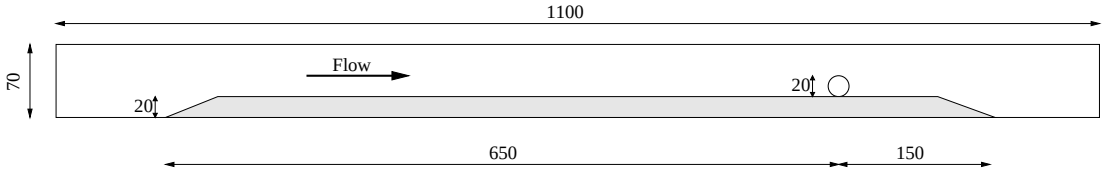


Figure 3: Géométrie initiale de l'expérience numérique. Les surfaces blanche et grise représentent des surfaces contenant du fluide et des sédiments respectivement. Les distances sont exprimées en unité de réseau.

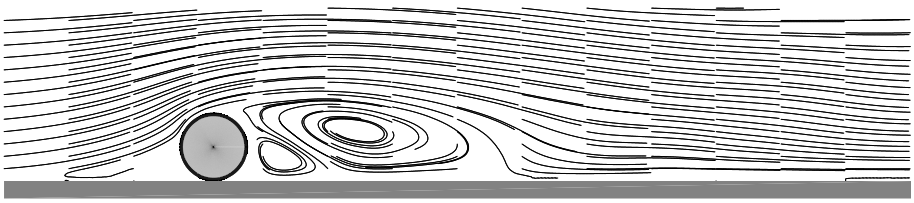


Figure 4: Tourbillons autour d'un tuyau sous-marin déposé sur un lit plat.

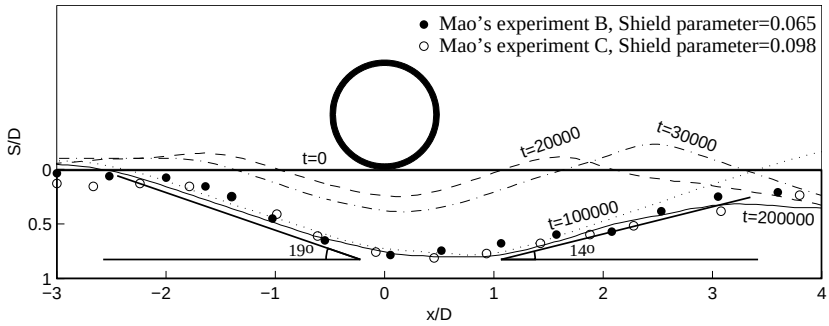


Figure 5: Etapes caractéristiques de l'évolution du processus de formation de l'excavation où S dénote la profondeur de l'excavation, D le diamètre du tuyau et x la distance horizontale à partir de la position du tuyau. Les profiles d'érosion sont présentés à plusieurs étapes de temps. Ils sont en accord avec les résultats d'expériences en laboratoire présentés par des cercles noirs et blancs.

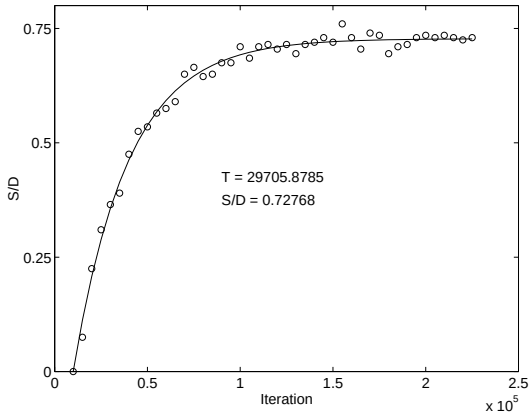


Figure 6: Evolution temporelle de la profondeur de l'excavation. La profondeur simulée est présentée par des cercles tandis que la ligne est la meilleure approximation par une exponentielle.

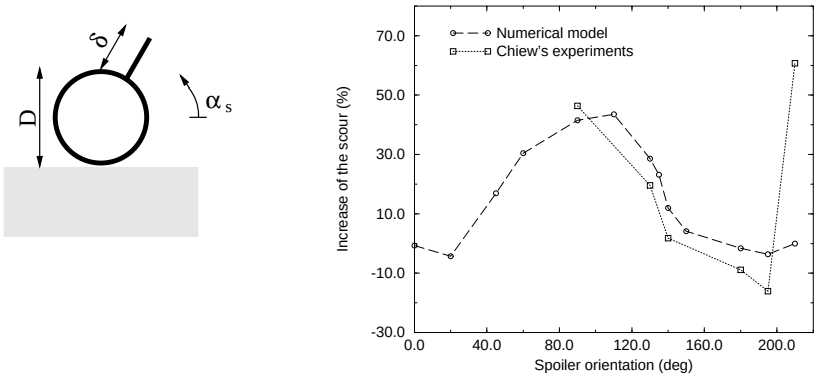
s'y faire enterrer. Ainsi, pour protéger le tuyau au mieux, on est intéressé à augmenter la profondeur du trou sans augmenter le diamètre. Cela peut se faire en attachant un aileron sur le tuyau. Notons que l'accroissement maximum est atteint lorsque ledit aileron trône verticalement (90^o) sur le tuyau. Nos simulations à ce sujet sont présentées dans la figure 7.

Formation et progression des méandres de rivières

La deuxième application que nous considérons est la formation et la progression des méandres de rivières. Ces derniers ont été passablement étudiés et un ensemble de lois empiriques a vu le jour. Par exemple, la relation entre la largeur de la rivière et son amplitude suit une loi de puissance. Ce qui implique que le ruisselet de laboratoire de quelques centimètres de large a les mêmes proportions que le fleuve Mississipi large de plusieurs kilomètres!

Nous proposons une explication des mécanismes mesoscopiques (à petites échelles, mais pas trop petites) gérant l'évolution de tels phénomènes. Notre explication semble être novatrice bien que certaines de ces parties soient présentes dans diverses publications.

Notre étude est aujourd'hui dans un état préliminaire. En effet, nous partons d'une rivière de forme sinusoïdale et la laissons évoluer. Nous observons un déplacement latéral de la rivière conforme à la réalité et à notre explication. Ces résultats sont présentés dans la figure 8.



a) b)

Figure 7: a) Tuyau sous-marin avec un aileron. L'angle de l'aileron est noté α_s, le diamètre du tuyau est D et la longueur de l'aileron est $\delta = D/2$. b) Accroissement de la profondeur de l'excavation en fonction de l'angle de l'aileron α_s. Les résultats obtenus par nos simulations sont dessinées avec des cercles et ceux d'expériences en laboratoire par des carrés.

Programmation des modèles de Boltzmann sur réseaux

Programmation traditionnelle

Traditionnellement, les modèles de Boltzmann sur réseaux se programme procéduralement en considérant des tableaux multidimensionnels modélisant le réseau. Plusieurs remarques émergent de ce genre d'implémentation:

- une modification peut souvent entraîner beaucoup de modifications majeures, ailleurs;
- un tableau est une structure régulière qui ne permet pas de considérer une grille qui pourrait elle-même être irrégulière, comme par exemple celle considérée pour la simulation de la figure 8 (le terrain loin de la rivière n'est pas modélisé);
- la réutilisabilité est moins aisée lors d'une programmation procédurale;
- la parallélisation est essentiellement à repenser lors de la considération de chaque nouvelle géométrie.

Ces remarques et réflexions nous ont conduits à l'élaboration d'un environnement orienté-objet pour des simulations utilisant des modèles de Boltzmann

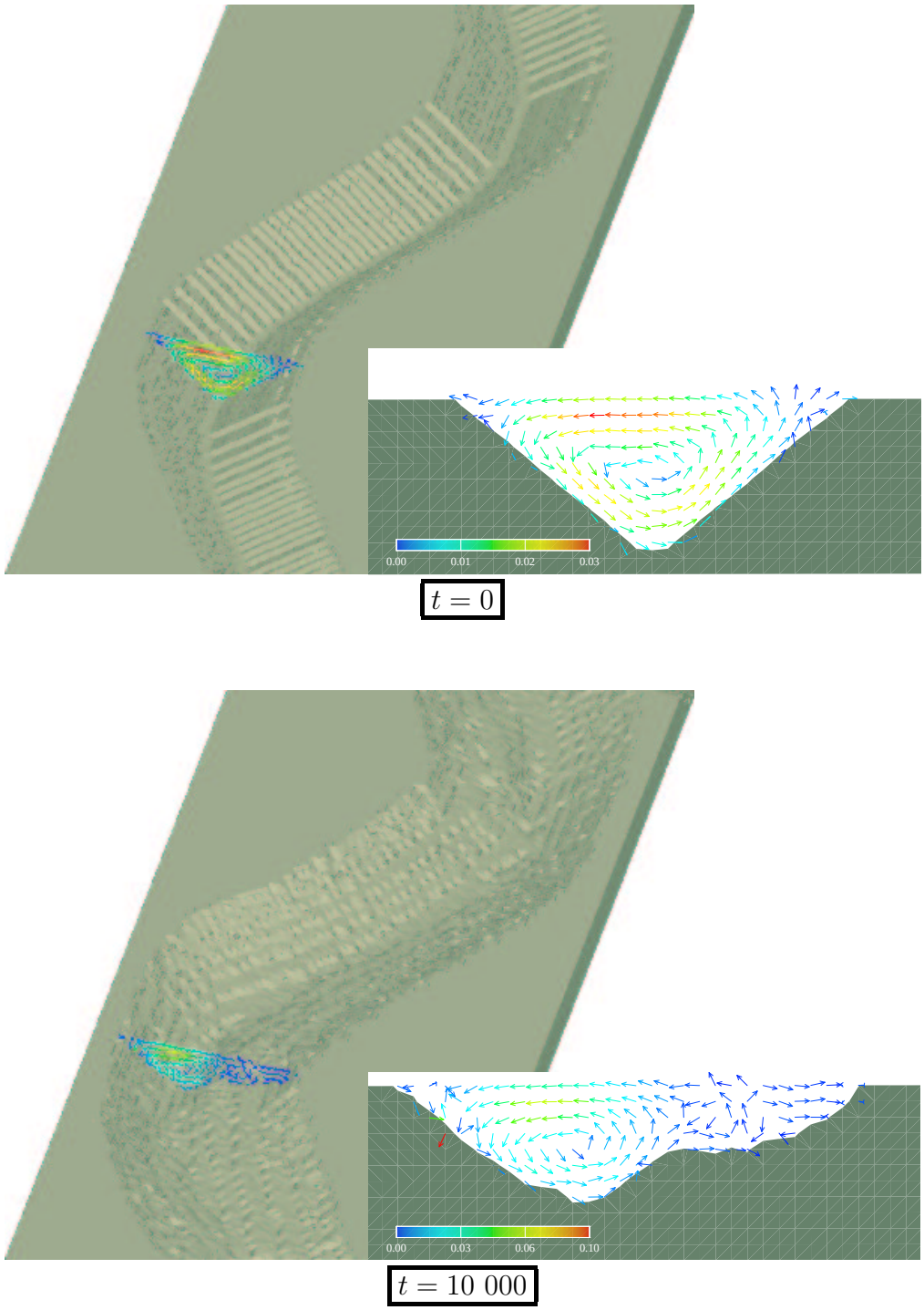


Figure 8: Résultats préliminaires de nos simulations d'évolution de méandres. La simulation démarre à $t = 0$, en considérant un méandre sinusoïdal. Les flèches représentent l'écoulement secondaire. Après 10 000 itérations, le méandre a progressé et a formé un nouvel écoulement secondaire décalé.

sur réseaux. Cet environnement est nommé PELABS (*Parallel Environment for LAttice Based Simulations*).

L'environnement PELABS

Brièvement, l'idée est de considérer un vecteur comme structure de données principale. Les cellules de ce vecteur sont les sites de la grille considérée. Ils sont composés de leurs voisinages (pointeurs sur des cellules du vecteur) et d'autres informations tel que les particules de fluide.

Ce vecteur est donc un déploiement d'une structure plus compliquée, multidimensionnelle. Cette manière de faire, permet de discerner les méthodes liées à une grille (création, affichage, parallélisation) du traitement de son contenu. La réutilisabilté et l'encapsulation en est grandement améliorée.

La parallélisation est programmée une unique fois de manière générale. En effet, indépendamment de son nombre de dimension, une grille est considérée comme un graphe. Ce dernier est décomposé par un algorithme de partitionnement de graphe et est judicieusement réparti aux processeurs de la machine parallèle.

Performances numériques de PELABS

Les performances de PELABS sont étudiées avec attention. Un modèle théorique est proposé. Il concorde bien avec des mesures réelles.

De plus, nous comparons numériquement une implémentation traditionnelle et une implémentation avec PELABS. Nous observons au pire que l'implémentation avec PELABS va jusqu'à deux fois plus lentement qu'une implémentation traditionnelle. Par contre, si les irrégularités de la grille sont suffisamment importante, l'implémentation PELABS devient plus rapide.

Finalement, nous analysons le comportement des performances lorsque les cellules dans le vecteur sont déplacées. Par ce biais, on diminue les défauts de cache. Nous observons que l'ordre naturelle, en ligne, est très proche de l'optimum. Toutefois, une amélioration sensible peut être obtenue en utilisant à nouveau un algorithme de partitionnement de graphe.

Conclusion

Nous montrons tout au long de ce document, que les modèles de Boltzmann sur réseaux sont appropriés au traitement efficace de problèmes ardus où d'autres méthodes peinent, ou voire échouent. Bien que ces modèles numériques ne soient pas complètement maîtrisés, les simulations effectuées nous permettent de leur prédire un avenir prometteur.

Chapter 1

Introduction

This chapter proposes an introduction and only deals with general concepts without going into details. States of the art related to the different aspects of this thesis, are presented in appropriate chapters.

1.1 Motivation

Reflecting the two components of this work, this section is decomposed into a model and an implementation subsection. The first one explains why we need new numerical models while the second one gives the motivation to consider current software engineering concepts in this computational physics context.

1.1.1 Model

Today, with the increasing power of computers, one has more and more recourse to them to simulate physical phenomena. This implies a virtual description of the phenomena which has to be as close as possible to reality. Compared to scale models, computer simulations present many advantages (e.g. they are easy to modify and handle, and are also cheaper). Also, these simulations and their numerical models bring a better understanding, allow us to make predictions and, sometimes, offer an explanation of the mechanisms.

Among the large number of physical processes, some are currently well simulated: some fluid flows [1], the first few seconds of crack formations [2] or even traffic flow [3]. But some others still need to be improved or considered. In this work, we will focus our attention on a subset which is the transport of sediments by a fluid. Sediment transport is basically composed of two main parts: a flowing fluid and sediments transported by the fluid.

Fluid flow simulations have interested researchers for a long time. This infatuation is essentially due to aeronautical and to car industry applications. Hence at present, one easily and accurately simulates the flow around aircraft wings

but also, in our context, the flow of a river. In these two examples, the fluid boundaries are fixed (at least initially). However, some applications require to consider moving boundaries which are, most of the time, not adequately managed by traditional numerical models. As an example, let us consider the wind flow around a dune, which is by nature dynamic. Thus, after a while, the dune moves and acts as a new boundary for the fluid. This example, rather simple, is difficult to simulate with current models.

Sediment transport represents a set of interesting applications. To fix the ideas, let us consider some examples: scour formation around submarine structures (see figure 5.1 for an illustration), formation and progression of meanders in rivers or the problems linked with the delta of rivers. Roughly speaking, the process acts as follows. A fluid flowing over sediments erodes some of them because of eddies which may appear sufficiently close to the fluid-sediment interface. Some of these eddies are sometimes strong enough to pick-up sediments. Eroded sediments are then transported in various ways (saltation, creeping and suspension [4]) according to several parameters such as sediment weight or local fluid velocity.

Despite their apparent simplicity, the previous examples are usually not so well simulated and understood by existing numerical models [5, 6]. These models compute not so intuitively some transport rates which are function of the flow. On the other hand, some more intuitive models have been proposed [7, 8]. Their main disadvantage is the small number of sediments they can deal with. The computational time of these methods is really high at the scale we wish to work at. Hence, more adapted numerical models are consequently needed.

1.1.2 Implementation

In the present computational physics context, computers are most of the time used as tools. Procedural languages (e.g. Fortran 90) are employed to program these computers. The main advantages of using that kind of languages are their simplicity and the efficiency of their compilers. Programs are quickly built and used for a short period of time. A subsequent use usually requires to write a new code or to manually adapt the old one. This is obviously a source of bugs and, after few iterations, increases the development time. Moreover, sharing and maintaining code is made more difficult by using this kind of implementation techniques.

On the other hand, much of computer science progress was done since the first programming languages were created (see figure 6.1). The aim of this progress was to propose better tools producing better codes. At the present time, computational physicists delay the use of these new concepts such as object oriented programming essentially for efficiency reasons.

Hence, there is a need to use high level computer science concepts even for numerical applications which are greedy in CPU time. The expected decrease of

performances should be quantified.

1.2 A mesoscopic approach: cellular automata

Differential equations, which is the description used by most numerical approaches, model the variations in space or time of quantities such as velocity. These quantities describe the mean behavior of molecules on small pieces of the domain. In this case, one looks at the phenomena of the so-called macroscopic scale as only high level informations are available.

The opposite scale consists in describing the molecular behavior of the phenomena. This scale is called microscopic and implies more accurate results. At this scale, one deals with quantities which are not available at the macroscopic scale. Obviously, a microscopic computation needs much more computational time. Indeed, it is rather common at this scale to simulate some nanoseconds of a physical process using several months of CPU time! Thus, the size of the applications considered is reduced.

Depending on the application and on the expected results, one is interested to view the system at an intermediate scale: the mesoscopic scale. It can be defined as a scale which refers to objects larger than an atom, but smaller than anything that can be manipulated with human hands.

At this mesoscopic scale, let us define a new simulation tool which consists of pseudo-particles moving in a fictitious world obtained by a discretization of the real world, and interacting according to a set of rules. Informally, we call it cellular automata (CA). See [9] for a precise description.

Some CA models have been successfully employed to simulate various processes. Let us quote some characteristic examples: traffic flow on highways [3] and in urban environment [10], or reaction-diffusion processes [9]. Next, CA models have been extended to simulate more complex processes such as radio wave propagation [11] or fluid flows [12]. These new models are called lattice Boltzmann (LB) models [12]. They are introduced and used in the following chapters.

We will see that CA and LB models use regular and uniform grids, usually referred to as lattices. This geometrical structure is often represented, in a computer, as regular arrays on which various functions are defined. Because one is interested to build efficient (in terms of speed) programs, software engineering concepts such as reusability or maintenance are often neglected. For example, a modification of the lattice topology can, in traditional implementation, implies many changes. Hence, we believe that an efficient but also a software engineering oriented implementation is needed.

1.3 A virtual river model

To describe erosion-deposition problems such as moving dunes and meandering rivers, we are interested to use cellular automata and lattice Boltzmann models. By considering a flow composed of water and of sediments, we will define what we call *a virtual river*. Water and sediment particles move in the same space and interact via a set of rules adequately chosen.

To satisfy software engineering concepts, the implementation of a virtual river, and of any other model using a lattice, may be programmed by considering a vector of cells rather than a multidimensional array. Such a vector implementation implies to add neighborhood information to the cells. The major advantage is that this vector data structure is usable for any lattice topology and dimension. Object oriented features allow us to deal with the latter requirements.

1.4 Organization of the thesis

This thesis is organized as follows. Following the introduction, chapter 2 discusses fluid dynamics. Some fluid dynamics essentials are introduced and the main traditional numerical models are presented. The chapter ends with a model comparison.

Chapter 3 is devoted to an introduction to lattice Boltzmann (LB) models. After the presentation of the basic model, existing boundary conditions are discussed. Then a new condition called *mass conserving boundary condition* is proposed and analyzed. The various existing flow settlement conditions are presented. Their characteristics are highlighted. Before closing the chapter with some validations concerning difficult flows, the Smagorinsky subgrid model is presented in the context of LB models.

Chapter 4 deals with mesh refinement. It presents the problematic and introduces the existing models. After having underlined their shortcomings, a new model is presented and validated. A specific application for speeding up the flow settlement process closes the chapter.

Chapter 5 introduces the numerical model describing the virtual river. The model is composed of water and sediments. The rules describing the behavior of sediments are defined (transport, deposition, erosion and toppling). Recall that water was the main subject of the previous chapter. Using this model, two applications are taken into account: scour formation under submarine pipelines and meanders in rivers. The first one is used as an application for calibration and validation essentially because the process is well known. In the second application, our numerical model acts as a tool to better understand the phenomenon of meandering rivers.

Chapter 6 discusses the computer implementation of lattice Boltzmann models. After reviewing basic concepts of computer science such as object oriented

and parallel programming, the usual parallelization of lattice Boltzmann is presented. Then, we precisely define our parallel and reusable environment called PELABS (**P**arallel **E**nvironment for **LA**ttice **B**ased **S**imulations). It is parallelized using a graph partitioning technique to decompose the domain. Its numerical performances are studied and a theoretical performance model is given. Then, a better use of the cache memory is proposed by employing again a graph partitioning technique.

Finally, in chapter 7 we draw some conclusions and highlight the work which remains to be performed.

Chapter 2

Basic concepts in hydrodynamics

2.1 Fluid dynamics essentials

2.1.1 Introduction

The main goal of the present section is to give the essentials of fluid dynamics in order to have a self-consistent document.

First, we recall the main equations of fluid dynamics. They are informally presented so as to understand the basic phenomena. More details and rigor can be found in specialized books, see for instance [13, 14].

Second, we review some aspects of turbulence by skimming through Kolmogorov's theory. This will be used to explain two empirical laws which are useful to overview the state of the art of turbulence theory. This presentation does not cover the wide turbulence theory. But it rather gives answers to basic questions. Among others, more information can be found in the books [15, 1] and in the complete review [16].

2.1.2 Governing equations

Consider a volume element $dxdydz$. The mass flux through this box leads to the first basic equation of fluid dynamics, the mass-conservation or continuity equation. It yields to

$$\frac{\partial(\rho U_x)}{\partial x} + \frac{\partial(\rho U_y)}{\partial y} + \frac{\partial(\rho U_z)}{\partial z} = \nabla \cdot (\rho \mathbf{U}) = \frac{\partial \rho}{\partial t} \tag{2.1}$$

where ρ is the fluid density and $\mathbf{U}$ is the fluid velocity. As we usually consider incompressible fluids (ρ is constant in space and time), equation (2.1) leads to its well-known form

$$\nabla \cdot \mathbf{U} = 0. \tag{2.2}$$

The end of this subsection is devoted to derive the momentum equation. If the fluid is inviscid, the momentum equation that results is called the Euler

equation. On the other hand, when the fluid is viscous, the equation is known as the Navier-Stokes equation. We begin with inviscid flows.

The force acting on a control volume of infinitesimal size is composed of a pressure difference on the faces of this volume and a body force due to gravity. This force can be expressed as

$$\mathbf{F} = \nabla p + \mathbf{G} \tag{2.3}$$

where ∇p is the pressure gradient $(\frac{\partial p}{\partial x}; \frac{\partial p}{\partial y}; \frac{\partial p}{\partial z})$ and $\mathbf{G}$ is a gravitational vector. Newton's second law of motion indicates that $\mathbf{F}$ is equal to the rate of change of momentum following a fluid particle. It leads to the Euler equation

$$\rho \frac{D\mathbf{U}}{Dt} = -\nabla p + \mathbf{G}. \tag{2.4}$$

where $\frac{D\mathbf{U}}{Dt}$ is the total derivative expressed as $\frac{\partial \mathbf{U}}{\partial t} + \mathbf{U} \cdot \nabla \mathbf{U}$.

We now consider the effect of viscosity. It is specific to a fluid and can be interpreted as the attachment rate between fluid particles. For example, honey particles are highly attached together, the viscosity is high. Water or air particles are less linked to each other, their viscosity is consequently smaller.

The attachment rate intuitively introduced before can be defined more precisely here. In the case of an incompressible flow, one can define a (viscous) shear stress as

$$\tau_{\alpha\beta} = \mu \frac{\partial U_\beta}{\partial x_\alpha} \tag{2.5}$$

where α and β are spatial components, μ is called the dynamic viscosity of the fluid. Note that a fluid which obeys equation (2.5) is called Newtonian. The force acting on a fluid element due to a shear stress is the variation of the stress in all the directions. It implies that the viscous force is given by

$$\mu\left(\frac{\partial \tau_{x\alpha}}{\partial x} + \frac{\partial \tau_{y\alpha}}{\partial y} + \frac{\partial \tau_{z\alpha}}{\partial z}\right) = \mu\left(\frac{\partial^2 U_\alpha}{\partial x^2} + \frac{\partial^2 U_\alpha}{\partial y^2} + \frac{\partial^2 U_\alpha}{\partial z^2}\right) = \mu \nabla^2 U_\alpha.$$

The viscous force can be added to equation (2.4) to give the Navier-Stokes equation

$$\frac{D\mathbf{U}}{Dt} = \frac{\partial \mathbf{U}}{\partial t} + \mathbf{U} \cdot \nabla \mathbf{U} = -\frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{U} + \frac{1}{\rho}\mathbf{G} \tag{2.6}$$

where ν is the kinematic viscosity related to the dynamic viscosity by $\nu = \mu/\rho$. In what follows, the kinematic viscosity will be simply called viscosity.

The Euler and especially the Navier-Stokes equations are nonlinear, partial differential equations, and no general solutions exist. Under particular conditions only, analytical solutions can be found. Numerical techniques must be used for other cases. Traditional numerical techniques are presented in section 2.2 while techniques based on lattice Boltzmann models make up section 3.5.

2.1.3 Turbulence

A flow classificator: the Reynolds number

A flow is, among others, characterized by a single non-dimensional parameter: the Reynolds number Re. It is defined by

$$Re = \frac{\mathcal{U}L}{\nu} \tag{2.7}$$

where $\mathcal{U}$ is the characteristic fluid velocity and L is the characteristic length scale of the flow.

A low Reynolds number indicates that the fluid velocity is slow or the length scale is small or the fluid viscosity is high. Hence the flow is smooth. On the other hand, a high Reynolds number expresses that the fluid velocity is fast or the length scale is big or the fluid viscosity is small.

A laminar flow is defined to be a flow with a small Reynolds number whereas a turbulent flow is a flow at high Reynolds number. The Reynolds number separating these two regimes is called the critical Reynolds number. For example, the critical Reynolds number of a flow around a cylinder is around 40 (see figure 2.1) and around 2300 for a channel flow.

A laminar flow is typically smooth and stationary in time. It means that after the flow settles $\mathbf{U}(\mathbf{r}, t) = \mathbf{U}(\mathbf{r}, t + 1)$. A turbulent flow presents some instabilities. Higher is the Reynolds number bigger are these instabilities. They appear far from the obstacle. Hence a layer called boundary layer is formed around the obstacle. The thickness of the boundary layer tends to zero as the Reynolds number tends to infinity. In this boundary layer the viscosity plays a more important role than in the bulk. The Van Dyke's book [17] proposes a wide variety of real experiments, some pictures of a flow around cylinders have been selected. They present a good example of a Reynolds number variation. These pictures also exhibit the apparition of turbulent instabilities. They are presented in figure 2.1.

The random nature of turbulence

In a turbulent flow, the velocity $\mathbf{U}(\mathbf{r}, t)$ is often called a random variable. In order to better understand the word *random*, consider a fluid-flow experiment that can be repeated many times under a specified set of conditions. Consider also an event A such as $A \equiv \mathbf{U}(\mathbf{r}, t) < 10m/s$. If the event A inevitably occurs, then A is certain. If the event A cannot occur, then it is impossible. The third possibility is that A may occur. In this case the event A is random. $\mathbf{U}(\mathbf{r}, t)$ is therefore a random variable.

The equations of motion are deterministic. But their solutions are random. Is there an inconsistency? No, the answer is that in any turbulent flow there are, unavoidably, perturbations in initial conditions, boundary conditions, and

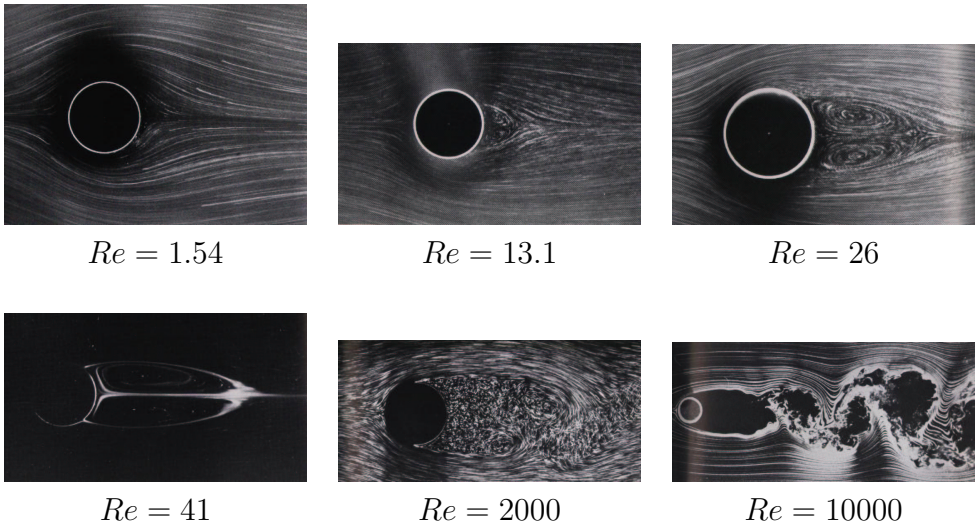


Figure 2.1: Real experiment of a flow past a cylinder at various Reynolds numbers. Instabilities appear if the Reynolds number is sufficiently high. These pictures have been selected in [17].

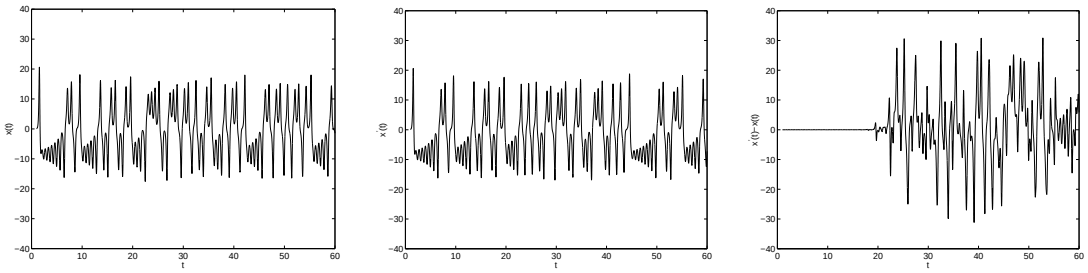
material properties. And that turbulent flow fields display an acute sensitivity to such perturbations especially at high Reynolds number.

To illustrate the importance of these perturbations let us consider the Lorenz system of ordinary differential equations

$$\begin{aligned} \dot{x} &= c_1(y - x) \\ \dot{y} &= c_2 x - y - xz \\ \dot{z} &= -c_3 z + xy \end{aligned} \tag{2.8}$$

where the coefficients are $c_1 = 10$, $c_2 = 28$ and $c_3 = 8/3$. For the initial condition $[x(0), y(0), z(0)] = [0.1, 0.1, 0.1]$, figure 2.2(a) shows the time history $x(t)$ obtained from the numerical integration of equations 2.8. The results obtained, denoted by $x^{'}(t)$, with the slightly different initial condition $[x(0), y(0), z(0)] = [0.100001, 0.1, 0.1]$ is shown in figure 2.2(b). It may be observed that $x(t)$ and $x^{'}(t)$ are initially indistinguishable, but by $t = 25$ they are quite different. This observation is made clearer in figure 2.2(c) showing the difference $x^{'}(t) - x(t)$.

A consequence of this extreme sensitivity to initial conditions is that the state of the system cannot be predicted. This example serves to demonstrate that a simple set of deterministic equations, much simpler than the Navier-Stokes equation, can exhibit a high sensitivity to initial conditions, and hence unpredictability. Note also that for the fixed values $c_1 = 10$ and $c_3 = 8/3$. the behavior of the Lorenz system depends on c_2. If c_2 is less than a critical value $c_2^* \approx 24.74$, then



(a) (b) (c)

Figure 2.2: Evolution of the Lorenz differential equation (2.8) (a) $x(t)$ from the initial condition $[x(0), y(0), z(0)] = [0.1, 0.1, 0.1]$; (b) $x^{'}(t)$ from the slightly different initial condition $[x(0), y(0), z(0)] = [0.100001, 0.1, 0.1]$; (c) the difference $x^{'}(t) - x(t)$.

the system goes to a stable fixed point, i.e., the variables $[x(t), y(t), z(t)]$ tend asymptotically to fixed values. However, for $c_2 > c_2^*$ (e.g. $c_2 = 28$ as in figure 2.2) chaotic behavior ensues. Again, there is a similarity to the Navier-Stokes equations. More details can be found in [1] from which this discussion is largely inspired.

Two empirical laws

Before going through the terms of the laws, consider the so-called longitudinal structure function. It is defined as

$$S_p(l) = \langle (\delta U_{\|}(\mathbf{r}, l, t))^p \rangle = \langle (U_x(\mathbf{r} + \mathbf{l}, t) - U_x(\mathbf{r}, t))^p \rangle \cdot \frac{\mathbf{l}}{l} \tag{2.9}$$

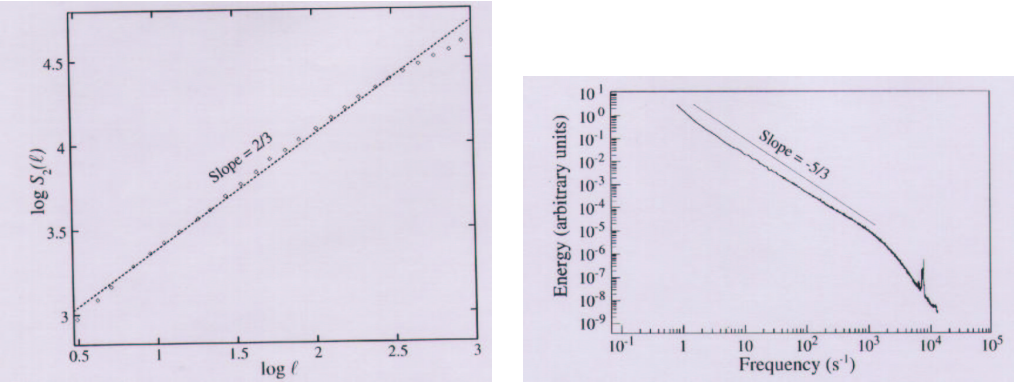
where $\langle\rangle$ is the time average, $p > 0$, U_x is the x velocity component, $\mathbf{r}$ is an arbitrary position, $\mathbf{l}$ is a longitudinal spatial increment and l its length.

Real experiments done so far allowed to exhibit two interesting empirical laws:

> **Two third law**. *In a turbulent flow at very high Reynolds number, $S_2(l)$ behaves as power of law of exponent* $2/3$ (i.e. $S_2(l) \propto l^{2/3}$).
> **Law of finite energy dissipation**. *If in an experiment on turbulent flow, all the control parameters are kept the same, except for the viscosity, which is lowered as much as possible, the energy dissipation ϵ has a finite limit.*

The energy dissipation is the amount of energy which is dissipated by the flow essentially by the small eddies, see below for details.

Figure 2.3(a) shows an observation of the two-third (empirical) law. Experiments [15] allow to observe the expected exponent in the inertial range (this notion will become clear in the following lines).



(a) (b)

Figure 2.3: These measurements are base on real experiment data in a wind tunnel. The Reynolds number is approximatively 10^6. (a) $S_2(l)$ of wind tunnel data. We observe a $2/3$ exponent. Remark also that l varies only in the inertial range. (b) Energy spectrum of wind tunnel data. The slope $-5/3$ appears in the inertial range. These two pictures have been selected from [15].

An often used measurement is the energy spectrum. It is defined as

$$E(\kappa) = \hat{U}(\kappa)^2 \tag{2.10}$$

where $\hat{}$ denotes the Fourier transform. One can demonstrate that the first empirical law is equivalent to the fact that the energy spectrum behaves as a power law of exponent $-5/3$. This is illustrated in figure 2.3(b).

The second empirical law can be illustrated by the drag force experiment acting on an obstacle. Consider a flow around an obstacle. Let $\mathbf{U}$ be the fluid velocity, ρ the fluid density and S the surface of the obstacle. The force acting on the obstacle is given by $\frac{1}{2}C_D\rho S\mathbf{U}^2$ where $C_D(Re)$ is a drag coefficient. The latter is dimensionless and allows us to compare experiments for which parameters are different. Expressing the kinetic energy dissipated when the obstacle moves against the flow, one can show that $\epsilon \propto C_D$ where ϵ is the energy dissipation.

Figure 2.4 presents the drag force coefficient around a cylinder versus the Reynolds number. We observe that for very small Reynolds number the drag coefficient goes as Re^{-1}. At high Reynolds number the drag coefficient stays approximately constant, except for an accident which occurs around Reynolds of a few hundred thousands. Thus taking $C_D(Re)$ to be constant is a reasonable approximation for large Re. Hence the energy dissipation has a finite limit for viscosity tending to zero.

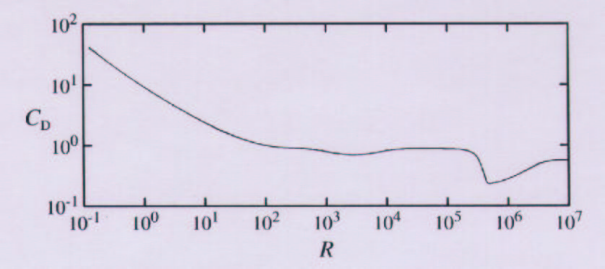


Figure 2.4: Drag force coefficient acting on a cylinder versus the Reynolds number of uniform flow over a cylinder. This picture has been selected from [15].

The Kolmogorov theory shortly visited

In 1941, Kolmogorov proposed his famous theory about turbulence [18, 19, 20] (K41). These are in agreement with the previous empirical laws and lead to other predictions. The K41 is partially and informally presented here. The reader interested in a complete and rigorous presentation can find some pointers in [15, 16].

Figure 2.5 presents the scales and the energy balance of turbulent flows. This phenomenon is usually represented by eddies successively breaking into smaller ones until they reach the dissipation scale. This process is often referenced as the cascade of Richardson[1].

The energy is introduced in the system between the biggest scale l_0 and $l_{EI} = l_0/6$ (empirically determined). In this context, a scale is the diameter of an eddy. l_0 corresponds to the size of the biggest eddy. For example, if a propeller fan of diameter l_0 would be used to accelerate the fluid it would generate eddies of size l_0. From l_{EI} and $l_{DI} = 60\eta$ (empirically established), eddies break into smaller ones and therefore energy is transfered to smaller scales. Finally from l_{DI} and $\eta = (\frac{\nu}{\epsilon})^{\frac{1}{4}}$, energy is dissipated into heat for example. η is the Kolmogorov dissipation scale, ν is the viscosity and ϵ is the energy dissipation.

The range called *energy containing range* is defined as the scales between l_0 and l_{EI}. The scales between l_{EI} and l_{DI} are in the inertial range. In this range, direct energy injection and energy dissipation are both negligible. Finally, the dissipation range is defined as the scales between l_{DI} and the smallest scale η.

The K41 theory is based on the following hypothesis:

[1] It seems that Leonardo Da Vinci had already observed this cascade and drawn similar sketches in his notes.

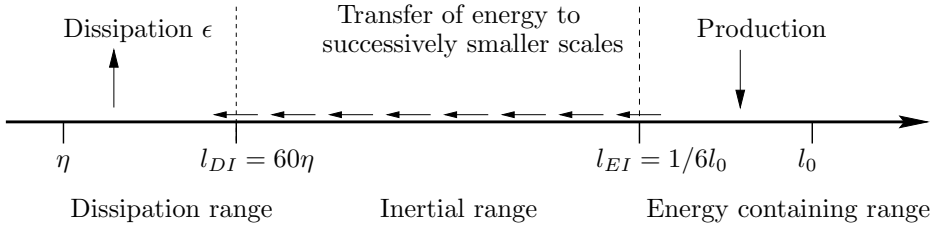


Figure 2.5: Scales and the energy balance of turbulent flows.

H1 *In the limit of infinite Reynolds number, all the symmetries of the Navier-Stokes equation, usually broken by the mechanism producing the turbulent flow, are restored in a statistical sense at small scales and away from boundaries.*

H2 *Under the same assumptions as in H1, the turbulence flow is self-similar at small scales, i.e. it possesses a unique scaling exponent h. Thus there exists a scaling exponent $h \in \mathbb{R}$ such that $\delta\mathbf{U}(\mathbf{r}, \lambda l) = \lambda^h \delta\mathbf{U}(\mathbf{r}, l), \forall \lambda \in \mathbb{R}_+$ for all $\mathbf{r}$ and all l small compared to the integral scale l_0.*

H3 *Under the same assumptions as in H1, the turbulent flow has a finite non vanishing mean rate of dissipation ϵ per unit mass.*

In [20], Kolmogorov found that an exact relation can be derived for $S_3(l)$. He assumed homogeneity (translation invariance), isotropy (rotation invariance) and hypothesis H3. Without any further assumptions he derived the following result from the Navier-Stokes equation.

> **The 4/5 law**. *In the limit of infinite Reynolds number, $S_3(l)$ of homogeneous isotropic turbulence, evaluated for increments l small compared to l_0, is given in terms of the mean energy dissipation per unit mass ϵ (assumed to remain finite and non vanishing) by*
>
> $$\langle (\delta U_{\parallel}(\mathbf{r}, l))^3 \rangle = -\frac{4}{5}\epsilon l. \tag{2.11}$$

The proof of this law is quite hard and not the aim of this section. However a proof can be found in [20, 15].

One of the main results of the K41 theory is the power law for the structure function of order p, $S_p(l)$. Kolmogorov argued that $S_p(l) \propto l^{p/3}$. This result gives a theoretical explanation of the first empirical law. Let us conclude this section by a proof of this result.

Hypothesis H2 implies that

$$\exists h \in \mathbb{R}_+ \quad | \quad \delta\mathbf{U}(\mathbf{r}, \lambda l) = \lambda^h \delta\mathbf{U}(\mathbf{r}, l).$$

The third structure function can be written as

$$S_3(\lambda l) = \langle(\delta U_\parallel(\lambda l))^3\rangle = \lambda^{3h}\langle(\delta U_\parallel(l))^3\rangle.$$

From the four-fifths law, one can express

$$S_3(\lambda l) = -\frac{4}{5}\epsilon\lambda l = \lambda(-\frac{4}{5}\epsilon l) = \lambda\langle(\delta U_\parallel(l))^3\rangle.$$

Finally, we conclude that

$$\lambda^{3h} = \lambda \Rightarrow h = \frac{1}{3}$$

and that

$$S_p(l) = S_p(l \cdot 1) = l^{p/3}S_p(1) = cl^{p/3}.$$

Since the K41 theory was proposed, many researchers have protested against it. The major critic is the simplicity of his hypotheses. Among others, Kolmogorov himself introduced important modifications in 1962 [21] taking into account spatial fluctuations in the turbulent energy dissipation. Anselmet [22] and others have criticized the linear exponent $\frac{p}{3}$ of the longitudinal structure function $S_p(l)$ for $p > 3$ and proposed some refinements. Let us also mention the notion of Extended Self Similarity (ESS) which extends hypotheses H1 and H2 in the case of smaller Reynolds number.

Discussion

Although the above presentation was succinct, it is clear that the K41 theory is definitely not complete and is largely supported by hypothesis. However an important part of his theory is still up to date. Probably because turbulence is a current subject of research and because a complete model describing the whole turbulent phenomenon is still needed.

To conclude, note that Nelkin who worked on turbulence for a long time wrote in 2000 [16] that: *Turbulence is often called the last major unsolved problem of classical physics.*

2.2 Computational fluid dynamics

2.2.1 Basic ideas

The most widely used class of fluid dynamics models is the so-called Computational Fluid Dynamics (CFD). The aim is to solve the partial differential equations constituting the Navier-Stokes equation. Their full numerical solutions are

possible only if the Reynolds number is low. In this case one talks about Direct Numerical Simulations (DNS) introduced in subsection 2.2.3. The Reynolds number reachable by the use of a DNS is rarely greater than the Reynolds number at which the transition to turbulence occurs, essentially because of the time needed to compute such a flow. Some DNS computation times are presented in table 2.1.

One has to deal with other kinds of models when the Reynolds number of the flow is high. The aim of these models is to compute one part of the flow and to simulate the other part. There exists currently two main models for solving high Reynolds number flows. First, the so-called Large Eddy Simulations (LES) models discussed in subsection 2.2.4. Second, the $k-\epsilon$ models which are presented in subsection 2.2.5. These two techniques attempt to simplify the inertial term $(\mathbf{U} \cdot \nabla \mathbf{U})$ in the Navier-Stokes equation which is difficult to solve.

Before going through an explanation of these CFD techniques, let us define a new concept: the eddy viscosity.

2.2.2 Eddy viscosity

In 1877, Boussinesq [23] observed that turbulence greatly (locally) increases the viscosity. So he introduced the concept of eddy viscosity ν_t in order to quantify this increase. The eddy viscosity is defined as

$$-\langle u_\alpha u_\beta \rangle = \nu_t \frac{\partial \tilde{U}_\alpha}{\partial x_\beta} \tag{2.12}$$

where $\langle\rangle$ is the time average, α and β are spatial components, u is the fluctuation of the velocity around the time averaged velocity $\tilde{U}$ and $\langle u_\alpha u_\beta \rangle$ is called Reynolds stress tensor component. This stress is an important quantity because it represents the main contribution to the residual which appears when one uses most numerical techniques. Roughly speaking, the Reynolds stress gives an indication on the flow fluctuation. One can note the similitude between equations (2.12) and (2.5) indicating why ν_t is considered as a viscosity.

The eddy viscosity is typically several orders of magnitude larger than the molecular viscosity. It is important to realize that ν_t is a representation of the action of turbulence on the mean flow and not a property of the fluid.

Hence, the eddy viscosity can be seen as an amount of viscosity produced by the unresolved scales. The effective viscosity (ν_{eff}) is locally the sum of the molecular viscosity (ν) and the eddy viscosity (ν_t).

2.2.3 Direct numerical simulation

A Direct Numerical Simulation (DNS) is a simulation where all scales of the flow are computed. It is computationally expensive but very accurate.

Re	Computation time
94	20 minutes
375	9 hours
1500	13 days
6000	20 months
24000	90 years
96000	5000 years

Table 2.1: DNS computation time of a typical flow at various Reynolds numbers. A single traditional computer has been used (in 2000). Values of this table were taken from [1].

The aim of this subsection is to give an insight into DNS. Only homogeneous turbulence (translation invariance) will be considered here. However references on DNS methods for inhomogeneous turbulence can be found in [1].

One of the DNS methods used for homogeneous turbulence is the so-called spectral method. It allows us to transform the Navier-Stokes equation in the Fourier space.

The higher the Reynolds number, the bigger the number of frequencies needed to solve all scales [1]. The increase in computation time is important. Table 2.1 presents the DNS computation time of a typical flow at various Reynolds numbers. A single traditional computer has been used. One observes (theoretically for most Re numbers) [1] that time scales as the Reynolds number to the cube. A DNS method can therefore be used for a low Reynolds number. But a DNS method is definitely not suitable if the Reynolds number increases too much.

To conclude this presentation let us mention the experiment of Le et al. [24]. They simulated the flow over a backward-facing step at a Reynolds number of 5000. They ran a DNS several months on a CRAY supercomputer. Their results were in very good agreement with laboratory experiments.

2.2.4 Large eddy simulation

A large eddy simulation (LES) is a turbulent flow simulation in which the large scale motions are explicitly resolved while the small scales motions are represented approximately by a model such as the Smagorinsky turbulence model.

The large scales are selected via a filtering operation. When the Navier-Stokes equation is filtered, the resulting equation for the large scale components of the velocity contains a term representing the effect of the small scales. This term is a stress tensor τ_{ij}^r (the subscript r is to underline the residual aspect of the tensor). In a sense, the effects of small scales are represented by this stress tensor.

One of the famous LES model is the one due to Smagorinsky [25]. He proposed

to define the residual stress tensor τ_{ij}^r as

$$\tau_{ij}^r = -2\nu_t \left(\frac{\partial \overline{U}_i}{\partial x_j} + \frac{\partial \overline{U}_j}{\partial x_i} \right) = -2\nu_t \overline{S}_{ij} \tag{2.13}$$

where ν_t is the turbulent viscosity, $\overline{U}_\alpha$ is the α component of the filtered velocity and $\overline{S}_{ij}$ is the strain tensor[2]. The turbulent viscosity is expressed as

$$\nu_t = (C_{smago}\Delta)^2 (2\overline{S}_{ij}\overline{S}_{ij})^2 \tag{2.14}$$

where C_{smago} is the Smagorinsky constant and Δ the filter width. The Smagorinsky constant C_{smago} is unfortunately not constant. It depends on simulations and to distance to obstacles but it is usually close to 0.1.

Hence, the new element (τ_{ij}^r) which appeared under the filtering operation is computable and the system closed.

In this subsection, we focused our attention on the Smagorinsky subgrid[3] model. This is not the only model. Pointers to others LES techniques can be found in [26].

2.2.5 $k - \epsilon$ model

The $k - \epsilon$ model is the most widely used complete turbulence model. k is the turbulent kinetic energy and ϵ is the rate of dissipation of turbulent kinetic energy. This model proposes to average the Navier-Stokes equation. The non-linear term becomes linear but an additional term, the so-called Reynolds stress, appears.

Without going into too much detail, one should be aware that the Reynolds stress tensor is related to the eddy viscosity and to the strain tensor (many models are presented in [1]). The latter is computable. A relation for the eddy viscosity is needed. The $k-\epsilon$ model defines the eddy viscosity as Ck^2/ϵ where C is a constant and where k and ϵ are given through evolution equations. Several variants exist. Some of their descriptions as well as pointers can be found in [1].

2.2.6 Model comparison

There are many kinds of flows and many problems to solve. First, for example one can mention a turbulent flow, at Reynolds number sufficiently low, for which the interest could be to compute the fluctuation of the velocity. In this case a DNS is probably adapted. A second example could be the determination of the center of a vortex at high Reynolds number. This is an example of adequate use

[2]Strain or stress? The confusion is easily done. In fact, a stress is a constraint and a strain is a deformation.

[3]This word is usually employed so as to express the fact that a LES model simulates actions happening at scales lower than the size of the grid.

	DNS	$k-\epsilon$	LES
Level of description	◯◯◯	◯	◯
Completeness	◯◯◯	◯◯◯	◯
Cost of use	◯	◯◯	◯◯◯
Range of applicability	◯	◯◯◯	◯◯
Accuracy	◯◯◯	◯	◯◯

Table 2.2: Appraisal criteria of three CFD models. The score of a model criteria is related to the number of symbols ◯ (between 1 and 3).

of LES. These examples highlight that there is no model perfectly adapted to applications but rather applications adapted to models.

Criteria indicating which model is preferable to select for a given application are needed. Pope defines five criteria for appraising models [1]. These criteria are

- level of description: indicates the quality of the flow description;
- completeness: points out if a model constituent equations are free from flow-dependent specifications;
- cost of use: describes the computational difficulty;
- range of applicability: indicates the kind of applications which can be addressed by the model;
- accuracy: gives an appreciation of the model accuracy.

The models presented in this section are given an evaluation according to the above criteria in table 2.2.

2.2.7 Limitations

A wide variety of flows can be addressed by CFD models. The large number of years spent in this direction has produced great ability and accuracy. For example, setting boundary conditions is usually a well-known operation.

However, there still exists flows for which CFD is not adapted. Let us mention two examples. First, the flow of a fluid transforming itself into a solid after a certain amount of time. This transformation could be typically due to temperature effects. The solid formed would be a new boundary for the flow. Second, the flow of mud or lava. These fluids are actually non Newtonian. The Navier-Stokes equations are consequently not suitable. These examples as well as many others are more easily treated with Lattice Boltzmann (LB) models. They are widely discussed in the next section.

To speed up an execution, one turns to parallel computing. Although it has not been shown here, one can mention the non parallel nature of CFD models. This is essentially due to the non locality of the process. Again, we will see in what follows that LB models are easily and efficiently parallelized.

Chapter 3

Lattice Boltzmann fluids

3.1 The lattice Boltzmann ancestor: the lattice gas automata

Before presenting Lattice Boltzmann (LB) models, let us devote some lines to an overview of their ancestors: the lattice gas automata. There is a large amount of literature on this subject. Pointers can be found in [9, 27].

Informally, the term *Lattice Gas Automata* (LGA) describes this tool quite well. It implements the dynamics of point particles colliding in a fully discrete space-time universe as illustrated, for instance, in figure 3.2. First, *Lattice* implies that one is working on a lattice which is d-dimensional and usually regular. Second, *Gas* suggests that a gas is moving on the lattice. The gas is actually represented by boolean particles. Third, *Automata* indicates that the gas evolves according to a set of rules. These are mainly defined to conserve mass (number of particles) and momentum (product of particle mass by particle velocity). These conservation rules are imposed in order to reproduce an hydrodynamical behavior, i.e. the Navier-Stokes equation (2.6).

More formally, a lattice is set of sites which are connected together by links. Their lengths are equal to $\Delta\mathbf{r}$ (which is the lattice spacing) for regular links, and depending on the lattice, equal to $\sqrt{2}\Delta\mathbf{r}$ or $\sqrt{3}\Delta\mathbf{r}$ for diagonal and longer links. A lattice forms a discrete and regular space. It is populated by fictitious particles $N_i^{in,out}(\mathbf{r},t) \in \{0;1\}$ where the subscript in, out indicates if a fictitious particle is going in or going out a lattice site $\mathbf{r}$ at discrete time t with velocity $\mathbf{v}_i$. Usually, i runs between 0 and z, where z is the number of links. By convention $\mathbf{v}_0 = 0$ and N_0 represents a particle at rest. A lattice topology is denoted by $DdQ(z+1)$ where d is the spatial dimension. Figure 3.1 presents the most common lattices.

An iteration of an LGA consists of a collision and a propagation step. Applying the automata rules makes up for the collision step. One has

$$N_i^{out}(\mathbf{r},t) = N_i^{in}(\mathbf{r},t) + \Omega_i(N_0^{in},\cdots,N_z^{in})$$

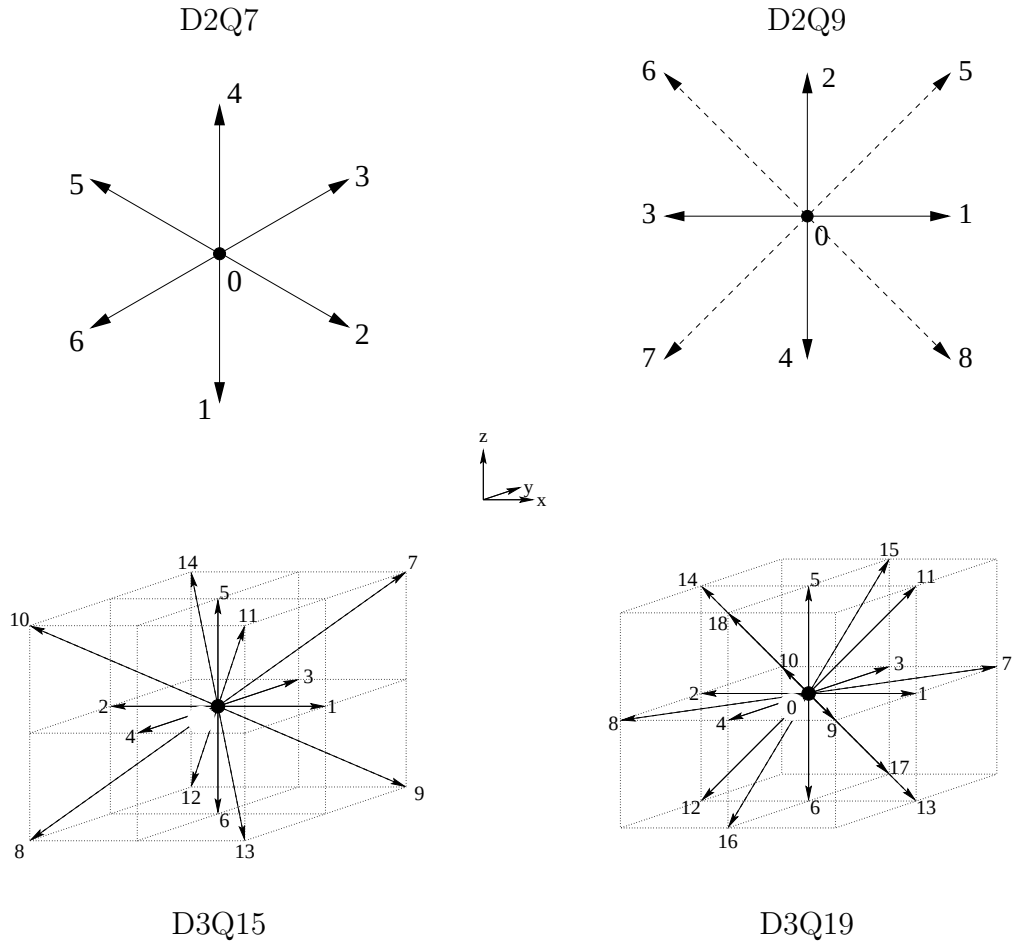


Figure 3.1: Sketches of the most common lattices. The directions i are numbered and correspond to the velocity vector $\mathbf{v}_i$. The D2Q5 model is the D2Q9 model without the diagonals. Links of size $\Delta\mathbf{r}$ are drawn as a solid line while the others are drawn as a dashed line.

where $\Omega_i(N_0^{in}, \cdots, N_z^{in})$ is the collision operator which depends on the model. Note that the collision process is local to a site.

The propagation step aims at letting the particles stream from site to site. It can be written as

$$N_i^{in}(\mathbf{r} + \Delta t\mathbf{v}_i, t + \Delta t) = N_i^{out}(\mathbf{r}, t).$$

where Δt is the time step of the simulation.

Hence, the LGA evolution can be expressed as

$$N_i^{in}(\mathbf{r} + \Delta t\mathbf{v}_i, t + \Delta t) = N_i^{in}(\mathbf{r}, t) + \Omega_i(N_0^{in}, \cdots, N_z^{in}) \tag{3.1}$$

The local density $\rho(\mathbf{r},t)$ is defined as

$$\rho(\mathbf{r},t) = \sum_{i=0}^{z} m_i N_i^{in,out}(\mathbf{r},t) \tag{3.2}$$

indifferently with *in* and *out* populations as the mass conservation is imposed by Ω. The local velocity $\mathbf{u}(\mathbf{r},t)$ is defined through the relation

$$\rho(\mathbf{r},t)\mathbf{u}(\mathbf{r},t) = \sum_{i=0}^{z} m_i N_i^{in,out}(\mathbf{r},t)\mathbf{v}_i \tag{3.3}$$

where m_i are weights associated to the lattice directions. A discussion about how the m_i's are set is postponed until the end of section 3.2.

Some LGA models are discussed in [28]. One of the first was the HPP model [29] which lives on a D2Q4 lattice (i.e. a D2Q5 lattice without rest particles). Among others, it allows us to simulate a gas with a very simple set of rules. We will see at the end of section 3.2 that an HPP model, that is to say its D2Q4 lattice is not isotropic. Indeed, it presents some unphysical anisotropy to rotation. Let us also mention an important class of LGA which are the FHP models [30, 31]. They are defined on an hexagonal lattice (D2Q7 and D2Q6) which is isotropic (for details see section 3.2). One can show, by averaging the dynamics, that the FHP set of rules reproduce, in some appropriate limit, a hydrodynamical behavior fluid [9, 32, 12] (i.e. the Navier-Stokes and the continuity equation).

The above discussion is now illustrated in figure 3.2, by presenting some typical iterations of an FHP model.

The collision operator $\Omega_i(N_0^{in},\cdots,N_z^{in})$ can be a long arithmetic Boolean expression. Its computation at each iteration and for every direction represent an important amount of time. Hence, one has usually recourse to a look-up table, which stores all possible configurations and their transformations (i.e. collisions). A look-up table is computed only once before the simulation. For example, the basic FHP model requires $2^6 = 64$ entries as there are two possible states and 6 links. Note that the arithmetic expression of the FHP collision operator generates 27 multiplications and 23 additions at each time step and for every directions (see [9] for details). We easily conclude that the use of a look-up table coding the LGA rules is efficient.

We saw that the LGA models deal with Boolean particles. Hence, to obtain a macroscopic value, such as the velocity, one has to average quantities in order to measure them with accuracy. The average is taken over time and/or over the spatial neighborhood of the given position. This is time consuming. Moreover, the reachable Reynolds numbers are quite low since the built-in viscosity of the model is rather large. Thus, higher Reynolds numbers can only be attained by an enlargement of the lattice. However, it is again time consuming.

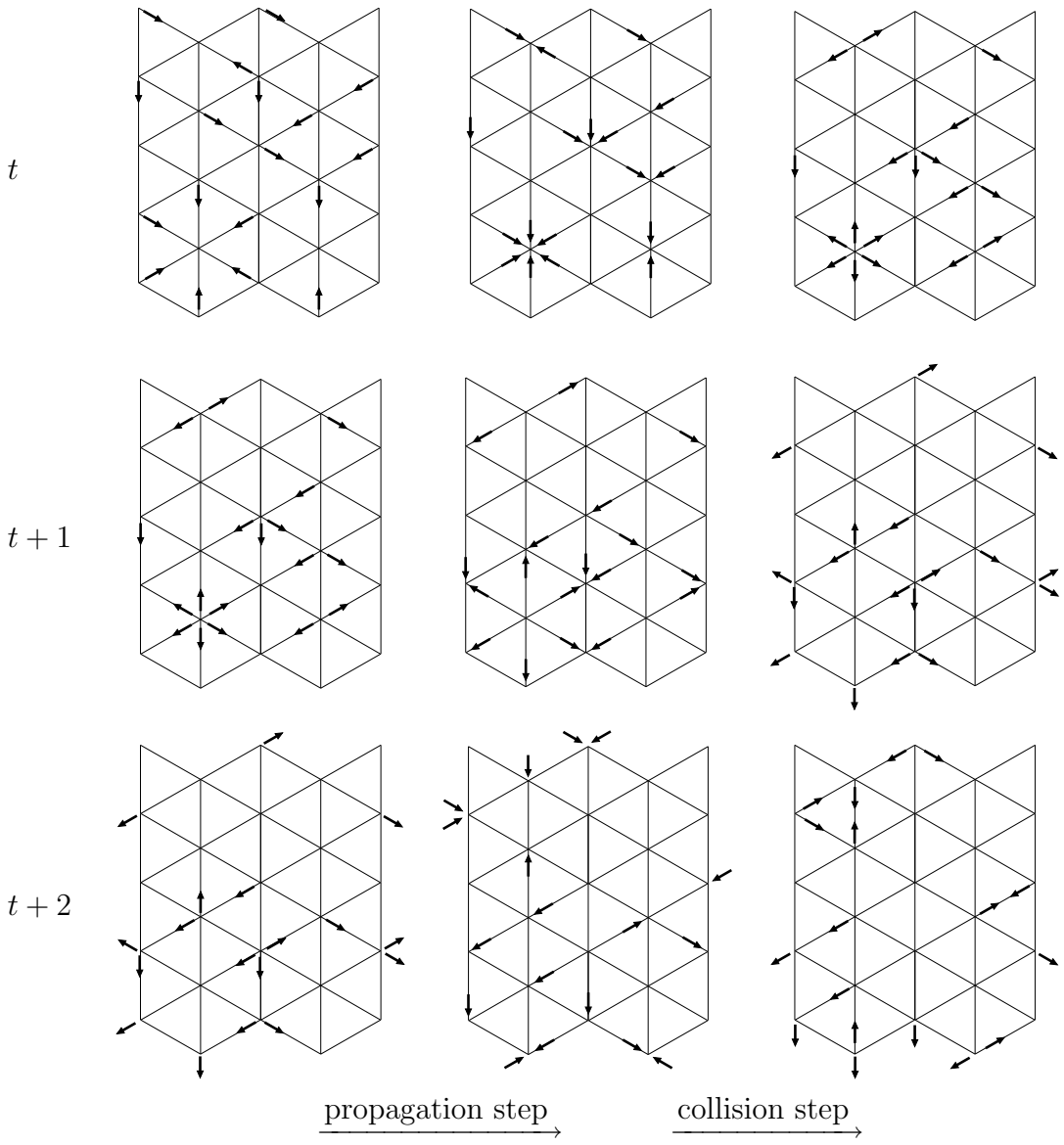


Figure 3.2: Three typical iterations of an FHP model on a D2Q6 lattice. The domain geometry is a 5×5 torus. Due to the fact that mass and momentum have to be conserved, there are only two non-trivial collision rules [30]. First, when two particles enter a site in front of each other. In this case, they are deflected by an angle of ± 60 degrees chosen with a probability of 0.5. Second, when 3 particles collide with an angle of 120 degrees between each other, they bounce back to where they come from. In all other cases, particles are unchanged as a collision preserving mass and momentum does not exist.

What happens if the average is done before the simulation rather than after? We will see in the next section how to implement this idea and that the gain is important in terms of accuracy and efficiency. The reachable Reynolds numbers are then significantly higher.

3.2 The lattice Boltzmann model

The concepts introduced in the previous section allow us to introduce the lattice Boltzmann models. Indeed, a formal averaging of equation (3.1) gives[1]

$$f_i^{in}(\mathbf{r} + \Delta t\mathbf{v}_i, t + \Delta t) = f_i^{in}(\mathbf{r}, t) + \langle \Omega_i(N_0^{in}, \cdots, N_z^{in}) \rangle \tag{3.4}$$

where $f_i^{in} = \langle N_i^{in} \rangle \in [0; 1]$ is the probability to have a fictitious particle with velocity $\mathbf{v}_i$ entering lattice site $\mathbf{r}$ at discrete time t. The quantity f_i is also usually called the particle density or density distribution. The term $\langle \Omega_i(N_0^{in}, \cdots, N_z^{in}) \rangle$ still needs to be expressed in function of the f_is. For that, we use the Boltzmann chaos hypothesis which proposes to neglect particle correlations. Let us assume that the Boltzmann hypothesis is valid for LGA models, i.e. $\langle \Omega(\cdots) \rangle = \Omega(\langle \cdots \rangle)$ (a discussion on its validity can be found in [9]). Hence, one can write down the evolution of a **lattice Boltzmann (LB) model** as

$$f_i^{in}(\mathbf{r} + \Delta t\mathbf{v}_i, t + \Delta t) = f_i^{in}(\mathbf{r}, t) + \Omega_i(f_0^{in}, \cdots, f_z^{in}). \tag{3.5}$$

The collision operator Ω_i is no longer a function of Boolean but of real-valued variables. Due to this change, the possible states of an LB model, and consequently their collisions, are much more numerous compared to the ones of an LGA. Note that the states of an LB model is not infinite because of the finite aspect of the real number representation on a computer. Anyway, a look-up table is not usable anymore. On the other hand, the collision operator is generally composed of a few hundred floating point operations. Their computation represents an important amount of time. Therefore, one needs another way to determine the post-collision state.

Higuera et al. [33, 34] proposed to linearize the collision operator around the so-called local equilibrium solution. This considerably reduces the complexity of the collision operator. Focusing our attention on a microdynamics, the lattice Boltzmann equation can be written as a relaxation equation [35]. This is the so-called LBGK method (the abbreviation BGK stands for Bhatnager, Gross and Krook [36] who first considered a collision term with a single relaxation time). It is expressed as

$$f_i^{in}(\mathbf{r} + \Delta t\mathbf{v}_i, t + \Delta t) = f_i^{in}(\mathbf{r}, t) + \frac{1}{\tau}(f_i^{eq}(\mathbf{r}, t) - f_i^{in}(\mathbf{r}, t))$$

[1] Average in the sense of statistical physics, i.e. over equivalent systems.

or equivalently as

$$f_i^{in}(\mathbf{r}+\Delta t\mathbf{v}_i, t+\Delta t) = \frac{1}{\tau} f_i^{eq}(\mathbf{r},t) + (1-\frac{1}{\tau}) f_i^{in}(\mathbf{r},t) \tag{3.6}$$

where the free parameter τ ($\tau > 0.5$, see below) is the relaxation time and $f_i^{eq}(\mathbf{r},t)$ is the local equilibrium which is a function of the density

$$\rho(\mathbf{r},t) = \sum_{i=0}^{z} m_i f_i^{in,out}(\mathbf{r},t) \tag{3.7}$$

and the fluid velocity $\mathbf{u}(\mathbf{r},t)$ defined through the momentum relation

$$\rho(\mathbf{r},t)\mathbf{u}(\mathbf{r},t) = \sum_{i=0}^{z} m_i f_i^{in,out}(\mathbf{r},t)\mathbf{v}_i. \tag{3.8}$$

where again, the m_i are the lattice weights whose determination is postponed till later. Note that in the following, the reference to time and space for the local density and velocity will usually be omitted.

Let us define two important dimensionless parameters. The Mach number as $M = \mathcal{U}/c_s$ where $\mathcal{U}$ is the characteristic flow velocity and c_s the speed of sound. Mach numbers are typically smaller than 0.3 when one considers fluid flow simulations. The Knudsen number $\epsilon = \Delta\mathbf{r}/\mathcal{L}$ where $\mathcal{L}$ is the characteristic length of the system.

By using the Chapman-Enskog expansion, the LBGK microdynamics (equation (3.6)), one can recover the governing fluid equations [37, 38, 39] if $\Delta\mathbf{r}$ and Δt are small enough and if the local equilibrium functions are appropriately chosen.

One shows [40] that error terms $O(\epsilon^2)$ and $O(\epsilon^2)+O(M^3)$ are attached to the continuity and the Navier-Stokes equation respectively. The error $O(\epsilon^2)$ is called the discretization error while the error $O(M^3)$ is called the compressibility error. Among others, they indicate that a small $\Delta\mathbf{r}$ leads to a small discretization error and that a small Δt, which implies a high speed of sound, is necessary to ensure a small Mach number.

An adequate choice of the local equilibrium functions is [37, 38, 39]

$$\begin{aligned} f_i^{eq} &= \rho\left[\frac{1}{C_2}\frac{c_s^2}{v_l^2} + \frac{1}{C_2}\frac{v_{i\alpha}u_\alpha}{v_l^2} + \frac{1}{2C_4 v_l^4}\sum_{\alpha\beta}\left(v_{i\alpha}v_{i\beta} - v_l^2\frac{C_4}{C_2}\delta_{\alpha\beta}\right)u_\alpha u_\beta\right] \\ f_0^{eq} &= \rho\left[1 - \frac{C_0}{C_2}\frac{c_s^2}{v_l^2} + \left(\frac{C_0}{2C_2} - \frac{C_2}{2C_4}\right)\frac{\mathbf{u}^2}{v_l^2}\right] \end{aligned} \tag{3.9}$$

where Greek indices are spatial directions and c_s is the speed of sound. Notice that the speed of sound is a free parameter. The Einstein convention is used for repeated spatial indices[2]. The constants C_0, C_2 and C_4 depend on the lattice and can be computed from equation (3.16), below.

[2] i.e. $v_{i\alpha}u_\alpha = \sum_\alpha v_{i\alpha}u_\alpha$

Considering LB models for fluid simulations, one commonly chooses the square of the sound speed as $c_s^2 = v_l^2 C_4/C_2$ where $v_l = \Delta\mathbf{r}/\Delta t$ (other LB applications such as radio wave propagation [11] make another choice). This specific speed of sound allows us to simplify equation (3.9). It yields

$$\begin{aligned} f_i^{eq} &= \rho \frac{C_4}{{C_2}^2}\left[1 + \frac{v_{i\alpha}u_\alpha}{c_s^2} + \frac{1}{2}\left(\frac{v_{i\alpha}u_\alpha}{c_s^2}\right)^2 - \frac{\mathbf{u}^2}{2c_s^2}\right] \\ f_0^{eq} &= \rho\left(1 - \frac{C_0 C_4}{{C_2}^2}\right)\left(1 - \frac{\mathbf{u}^2}{2c_s^2}\right) \end{aligned} \tag{3.10}$$

One can show that the pressure is directly related to the density by [9, 12]

$$p(\mathbf{r}, t) = c_s^2 \rho(\mathbf{r}, t) \tag{3.11}$$

and the lattice viscosity by

$$\nu = \frac{\Delta\mathbf{r}^2}{\Delta t}\frac{C_4}{C_2}\left(\tau - \frac{1}{2}\right). \tag{3.12}$$

Equation (3.12) indicates that the relaxation time τ has to be greater or equal to 0.5. Otherwise the viscosity would be negative. The viscosity is low when τ is close to 0.5. But unfortunately, we will see in section 3.5 that when τ is approximately lower or equal to 0.6 numerical instabilities appear. Turbulence models are then employed and allow us to reach low viscosity. See section 3.5 for details. Note however that a dedicated initialization allows us to simulate a periodic fluid considering a relaxation time very close to 0.5 [12].

From a computational point of view, it is more convenient to simulate a $\bar{f}_i^{in,out} = m_i f_i^{in,out}$ population instead of an $f_i^{in,out}$ population. It implies that the density and the velocity are computed more quickly and then more efficiently by avoiding a multiplication. Therefore, the density becomes $\rho = \sum_i \bar{f}_i^{in,out}$, the momentum $\rho\mathbf{u} = \sum_i \bar{f}_i^{in,out}\mathbf{v}_i$ and equation (3.10)

$$\begin{aligned} \bar{f}_i^{eq} &= \rho t_i\left[1 + \frac{v_{i\alpha}u_\alpha}{c_s^2} + \frac{1}{2}\left(\frac{v_{i\alpha}u_\alpha}{c_s^2}\right)^2 - \frac{\mathbf{u}^2}{2c_s^2}\right] \\ \bar{f}_0^{eq} &= \rho t_0\left(1 - \frac{\mathbf{u}^2}{2c_s^2}\right). \end{aligned} \tag{3.13}$$

where

$$t_i = m_i\frac{C_4}{{C_2}^2} \quad \text{and} \quad t_0 = m_0\left(1 - \frac{C_0 C_4}{{C_2}^2}\right) \tag{3.14}$$

In what follows, the quantities $\bar{f}_i^{in,out}$ will be simply denoted by $f_i^{in,out}$.

To conclude this presentation, let us mention the recent proposition of Lallemand et al. [41]. They suggest to work on a momentum space rather than in a discrete velocity space. The numerical stability seems to be improved [42].

3.2.1 Determination of lattice weights

Let us go back to the determination of the lattice weights. They are chosen so as to ensure the lattice isotropy (i.e. any physical quantity is independent of a specific lattice orientation). The lattice isotropy is related to the isotropy of its tensors [43]. A tensor of rank r is defined as

$$T_{\alpha_1\alpha_2\cdots\alpha_r} = \sum_{i=1}^{z} m_i v_{i\alpha_1} v_{i\alpha_2} \cdots v_{i\alpha_r}. \tag{3.15}$$

In the LB hydrodynamics context, a sufficient condition for reasonable precision is to ensure the isotropy of lattice tensors up to the fourth order [9].

Jeffreys et al. [44] proved a theorem arguing that a tensor $T_{\alpha\beta}$ of rank 2 is isotropic if it is proportional to the Kronecker function $\delta_{\alpha\beta}$ and that a tensor $T_{\alpha\beta\gamma\delta}$ of rank 4 is isotropic if it is of the form $a\delta_{\alpha\beta}\delta_{\gamma\delta} + b\delta_{\alpha\gamma}\delta_{\beta\delta} + c\delta_{\alpha\delta}\delta_{\beta\gamma}$ where a,b and c are arbitrary constants. Recall that the Kronecker function $\delta_{\alpha\beta}$ is equal to 1 if $\alpha = \beta$ and 0 otherwise. The lattice tensors with odd rank vanish because of the symmetry of the lattice.

The question is then to find m_i so that the following relations, which are the first five lattice tensors, hold

$$\begin{aligned}
\sum_{i=1}^{z} m_i &= C_0 \\
\sum_{i=1}^{z} m_i v_{i\alpha} &= 0 \\
\sum_{i=1}^{z} m_i v_{i\alpha} v_{i\beta} &= v_l^2 C_2 \delta_{\alpha\beta} \\
\sum_{i=1}^{z} m_i v_{i\alpha} v_{i\beta} v_{i\gamma} &= 0 \\
\sum_{i=1}^{z} m_i v_{i\alpha} v_{i\beta} v_{i\gamma} v_{i\delta} &= v_l^4 C_4 (\delta_{\alpha\beta}\delta_{\gamma\delta} + \delta_{\alpha\gamma}\delta_{\beta\delta} + \delta_{\alpha\delta}\delta_{\beta\gamma})
\end{aligned} \tag{3.16}$$

where C_0, C_2 and C_4 are lattice dependent and $v_l = \Delta\mathbf{r}/\Delta t$. Some algebra allows us to express equations (3.16) in a more convenient way for C_2 as

$$\sum_{i=1}^{z} m_i \mathbf{v}_i^2 = dC_2 v_l^2 \tag{3.17}$$

and for C_4 as

$$\sum_{i=1}^{z} m_i \mathbf{v}_i^4 = (d^2 + 2d) C_4 v_l^4. \tag{3.18}$$

Models	Slow velocities			Fast velocities			C_0	C_2	C_4	c_s^2	t_0
	$\|\mathbf{v}_i\|$	m_i	t_i	$\|\mathbf{v}_i\|$	m_i	t_i					
D2Q7	v_l	1	1/12				6	3	3/4	1/4	1/2
D2Q9	v_l	4	1/9	$\sqrt{2}v_l$	1	1/36	20	12	4	1/3	4/9
D3Q15	v_l	1	1/9	$\sqrt{3}v_l$	1/8	1/72	7	3	1	1/3	2/9
D3Q19	v_l	2	1/18	$\sqrt{2}v_l$	1	1/36	24	12	4	1/3	1/3

Table 3.1: Constants of the most common $DdQ(z+1)$ lattices where d is the spatial dimension and z is the number of link, $\mathbf{v}_i$ is the velocity on link i, m_i are the weights associated with each link and t_i are the factors of the equilibrium function associated with each link.

One can compute the tensor of a D2Q5 lattice supposing that $m_i = 1 \quad \forall i$ [43]. Thus, the second order tensor is equal to $2\delta_{\alpha\beta}$ and the fourth to $2\delta_{\alpha\beta\gamma\delta}$ implying that the D2Q5 lattice is anisotropic at 4^{th} order. Fortunately, a similar computation indicates that an hexagonal lattice (D2Q7) is isotropic with $m_i = 1 \quad \forall i$. Other isotropic lattices (see figure 3.1) can be found by projecting the FCHC (Face Centered Hyper Cubic) 4-dimensional lattice onto a subspace [45]. The FCHC lattice is indeed isotropic and defined as

$$\begin{aligned} &(\pm1, \pm1, 0, 0), \quad (\pm1, 0, \pm1, 0), \quad (\pm1, 0, 0, \pm1), \\ &(0, \pm1, \pm1, 0), \quad (0, \pm1, 0, \pm1), \quad (0, 0, \pm1 \pm 1). \end{aligned} \tag{3.19}$$

The weights as well as various constants of the most common lattices are drawn in figure 3.1 and recapitulated in table 3.1. Note that m_0 is not involved in the tensor equations and can be set to 1 without loss of generality [37].

Note however that the D3Q15 lattice presents a checkerboard invariant [46]. Indeed, the fluid momentum may form unphysical regular patterns. Hence, the D3Q19 lattice is usually preferred when a 3D simulation is needed.

The D2Q7 and D2Q9 lattices seem to be similar from the point of view of results. The first one needs less memory to store its states while the second one allows the use of a Cartesian grid which is a simpler data structure.

3.2.2 A basic code

Enough material has been introduced to present a basic code for the simulation of a fluid with periodic boundary conditions. It is composed of a set of procedures written in C++ in a form quite close to traditional pseudo-codes. They are subject to modification according to new notions which will be presented in the following. We focus our attention on the LB D2Q9 model. Changing to other models, requires to modify some constants and lines of code. Software engineering aspects are not presently our aim. They are however largely discussed in chapter 6.

The main procedure and the definition of global variables are presented in code 3.1. An LB D2Q9 model can be initialized by the code 3.2. Local density and velocity can be computed with code 3.3. Collision and propagation procedures are the subject of code 3.4 and code 3.5 respectively.

Code 3.1 Main function and the definition of the global variables.

```
int nx=17;                          // 17 sites in the x direction.
int nz=11;                          // 11 sites in the z direction.
int nbIter=5000;                    // The process will run for 5000 iterations.
int nbDim=2                         // A two-dimensional model
int nbNeighbor=9;                   // with 9 neighbors is considered.
double tau=1.0;                     // The relaxation time is set to 1.0
float cs2=1.0/3.0;                  // The speed of sound c_s^2
float v[9][2],t[9];
double fin[17][11][9],fout[17][11][9];

int main(int argc, char ** argv)
{
  init();
   // Executes nbIter iterations
  int iter;
  for (iter=0;iter<nbIter;iter++) {
    collision();
    propagation();
  }
  return 0;
}
```

3.3 Boundary conditions

For the sake of clarity, we considered in the previous section only infinite domains. But as real experiments impose to consider finite domains, one has to treat specially the dynamics on boundaries by defining boundary conditions. Let us distinguish between the conditions which have to be applied at the interface of a fluid and a solid (boundary conditions) and the conditions aiming at settling the flow. The latter is developed in section 3.4.

Many boundary conditions have been proposed in the past [47]. The most common and simple one is the bounce-back boundary condition. It is explained in the following. This condition is only first order in terms of numerical accuracy [48, 49]. Hence, to improve it other boundary treatments have been proposed.

Code 3.2 Initialization function.

```
void init()
{
   // Sets the fluid particle velocities
  v[0][0]=+0.0; v[0][1]=+0.0;   v[1][0]=+1.0; v[1][1]=+0.0;
  v[2][0]=+0.0; v[2][1]=+1.0;   v[3][0]=-1.0; v[3][1]=+0.0;
  v[4][0]=+0.0; v[4][1]=-1.0;   v[5][0]=+1.0; v[5][1]=+1.0;
  v[6][0]=-1.0; v[6][1]=+1.0;   v[7][0]=-1.0; v[7][1]=-1.0;
  v[8][0]=+1.0; v[8][1]=-1.0;
   // Sets the equilibrium distribution function constants
  t[0]=4.0/9.0;
  t[1]=1.0/9.0;  t[2]=1.0/9.0;  t[3]=1.0/9.0;  t[4]=1.0/9.0;
  t[5]=1.0/36.0; t[6]=1.0/36.0; t[7]=1.0/36.0; t[8]=1.0/36.0;
   // Initializes the fields with a null velocity and a density equal to 1.0
  int i,j,k;
  for (i=0;i<nx;i++)
    for(j=0;j<nz;j++)
      for (k=0;k<nbNeighbor;k++)
     fin[i][j][k]=t[k];
}
```

Code 3.3 Functions computing the local density and velocity.

```
double getDensity(const int i, const int j)
{
  int k;
  double result=0.0;
  for (k=0;k<nbNeighbor;k++)
    result+=fin[i][j][k];
  return result;
}
void getVelocity(const int i, const int j, const double density, double *u)
{
  int k,d;
  for (d=0;d<nbDim;d++) {
    u[d]=0.0;
    for (k=0;k<nbNeighbor;k++)
      u[d]+=fin[i][j][k]*v[k][d];
    u[d]=u[d]/density;
  }
}
```

Code 3.4 Collision function.

```
void collision()
{
  int i,j,k,d;
  double density,u[2],p,feq;
  for (i=0;i<nx;i++)
    for (j=0;j<nz;j++) {
      density=getDensity(i,j);
      getVelocity(i,j,density,u);
      for (k=0;k<nbNeighbor;k++) {
        p=0.0;
        for (d=0;d<nbDim;d++) p+=v[k][d]*u[d];
        p=p/cs2;
        feq=t[k]*density*(1.0+p+0.5*p*p-0.5*(u[0]*u[0]+u[1]*u[1])/cs2);
        fout[i][j][k]+=1/tau*(feq-fin[i][j][k]);
      }
    }
}
```

Code 3.5 Propagation function.

```
void propagation()
{
  int i,j,k;
  int ii,jj;
  // Propagates the fields.
  for (i=0;i<nx;i++)
    for(j=0;j<nz;j++)
      for (k=0;k<9;k++) {
        ii=(i+v[k][0]+nx)%nx;
        jj=(j+v[k][1]+ny)%ny;
        fin[ii][jj][k]=fout[i][j][k];
      }
}
```

Before going through some of the existing boundary conditions, let us briefly review major contributions. Skordos [50] proposed to include velocity gradients in the equilibrium distribution function at the wall nodes. He et al. [51] extended the bounce-back condition for the non-equilibrium portion of the distribution. Inamouro et al. [52] suggested to cancel a slip velocity at the wall by using a counter slip velocity. This scheme is detailed below. Chen et al. [53] proposed to use a simple extrapolation scheme. Maier et al. [54] modified the bounce-back condition to nullify the net momentum tangent to the wall and to preserve momentum normal to the wall. All these conditions deal with flat walls. Their action is to define the unknown fields which would come from the solid. However, note that all these bounce-back condition improvements are difficult to implement for general geometries essentially because one has to know the wall orientation. With these boundary conditions, corner nodes - if there is any - require a special treatment.

Recently, some advances have been made in the treatment of curved and off-lattice boundaries [55, 56, 57]. Indeed, curved boundaries are often approximated by a series of stairs. Depending on the simulation, this approximation leads to a consequential reduction of numerical accuracy. This boundary condition category is not detailed here. The interested reader is invited to consult [55, 56, 57].

The non-slip boundary condition [52] seems to be a good candidate for a *perfect* boundary condition. This is true at least when it is used for simulating a symmetrical and large enough system. A closer look to the condition discloses that mass is not necessarily conserved. Inamouro et al. [52] did not observe this loss because of the symmetries displayed by the system. To alleviate this problem, we propose a new boundary condition: the mass conserving boundary condition. It is detailed below. Before that, we review the standard bounce-back boundary conditions.

3.3.1 Bounce-back conditions

Fullway bounce-back

Let us start with the simplest boundary condition, the so-called fullway bounce-back condition (the meaning of fullway will become clear in what follows). A fluid particle colliding a boundary site simply reverses the direction of its velocity. Hence we locally have

$$f_{\bar{k}}^{out} = f_k^{in} \tag{3.20}$$

where f^{in} is a particle field entering the boundary site, f^{out} is a particle field leaving the boundary site, k and $\bar{k}$ denote directions opposite to each other. Note that, with the bounce-back condition, the collision process (i.e. the relaxation) does not occur at the boundary.

On a boundary site, one can easily observe that mass and momentum are conserved. Also, on average, the velocity on a boundary site is zero as any

particle entering the sites with a given velocity leaves with the opposite velocity. The balance is then null. It is important to clearly indicate how, and actually when, the velocity at the boundary is measured. In principle, without any other indication, we measure the velocity at the boundary after the application of the boundary condition (and the collision rule if there is any). In other words, we measure the velocity at the boundary before the propagation process. We attract the attention of the reader once again by indicating that it is actually not rare to find a measurement indicating a null velocity at a boundary. This is usually done by considering both f^{in} and f^{out} to compute the velocity. In that artificial view of the system, the velocity is null at the boundary.

The bounce-back rule is particularly interesting because of its simple implementation and its generality. Indeed, any wall orientation as well as any border shape are treated in the same way.

Halfway bounce-back

Unfortunately in this case, the simplest is not the best. One can demonstrate that the use of the fullway bounce-back usually generates a slip velocity and may lead to poor first order numerical accuracy in space [49]. However, second order numerical accuracy in space can be obtained [49] by using the original idea of Cornubert et al. [58]. They suggested to consider the boundary halfway between the first row of sites. This is the so-called halfway bounce-back boundary condition.

Notice that the second order numerical accuracy, resulting from the use of the halfway bounce-back boundary condition, is observed and analytically proved for some simple flows such as the Poiseuille flow. It is not clear how evolves this accuracy with more complicated flows (e.g. cavity flow). Our preliminary results indicate that only a first order numerical accuracy is obtained, see figure 3.9.

Table 3.2 presents with details the difference between the halfway and the regular (fullway) bounce-back boundary conditions.

There are two manners to implement the halfway bounce-back condition. First, the idea is to apply the bounce-back condition and, in the same time step, to stream the boundary particles. In this implementation, the physical boundary (i.e. the wall) is located at the middle of the first row implying that an unused site is modeled. This implementation is intuitive but unfortunately non-local as it needs an additional streaming for these sites. It then makes it difficult to parallelize the code.

The second way to implement this consists in applying the bounce-back condition during the propagation. Indeed, on lattice boundary sites, one reverses the velocity of a particle which wants to enter a solid area. The propagation is applied for the other particles. This implementation is local. Note also that boundary sites are not modeled anymore as they are not used.

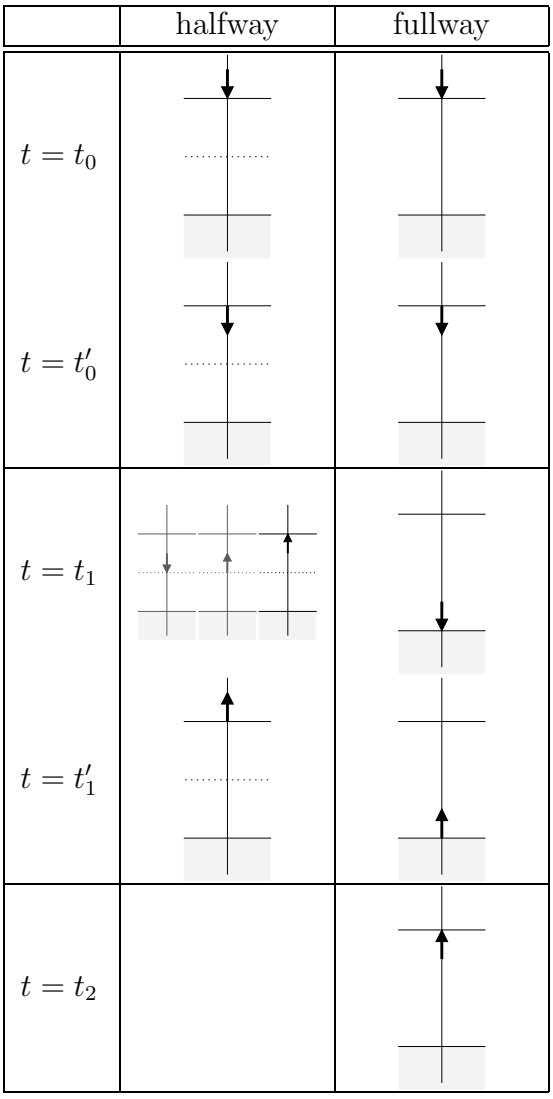


Table 3.2: Bounce-back boundary conditions on a simple D2Q5 lattice. t_t denotes a time step after a propagation process and t'_t the time after the application of the boundary condition. On the left, this is the so-called halfway boundary condition [58, 49]. In only one time step, a fluid particle goes to the boundary site, reverses its velocity and comes back. Hence, the boundary is located in the middle of the first row of fluid sites as a fluid particle needs only one time step to go forth and back. On the right, a fluid particle entering a boundary site simply reverses the direction of its velocity. This is the fullway bounce-back condition which needs two time steps to go forth and back.

Modified bounce-back

The so-called modified bounce-back is a variant of the traditional bounce-back. It consists to set the unknown incoming fields (i.e. those coming from the solid) with their mirror particle field. Hence, one has

$$f_{\bar{k}}^{in} = f_k^{in} \tag{3.21}$$

where $f_{\bar{k}}^{in}$ are unknown particle fields entering the boundary site, k and $\bar{k}$ denote directions opposite to each other. The collision operator is applied on boundaries as every fields are known. Note that this process does not necessarily conserve mass.

Application of the boundary conditions

The convergence of these three conditions are highlighted by considering the simulation of a 2D LBGK Poiseuille flow [13, 1]. The D2Q9 lattice has $N_x = 17 \times N_z$ sites. It is horizontally periodic. The flow is generated by a constant body force $G_x^* = 0.1$ along the x direction (see section 3.4 for details). The viscosity is fixed to $\nu = 0.001$ and the relaxation time τ is equal to 1.0. Assume that $\Delta\mathbf{r}/\Delta t = C$ where C is a constant value. Let say that $C = 1$ where unities are intentionally omitted for simplicity reasons. Then, we find from equation (3.12) that $\Delta\mathbf{r} = (3\nu)/(\tau - 1/2) = 0.006$. A derivation of the Navier-Stokes equation (equation (2.6)) for this specific case allows us to express the velocity in the center of the channel as $U_c = \frac{L^2\rho}{8\nu}$ where L is the height of the channel and ρ is the density fixed here to 1.0. The fullway bounce-back boundary condition imposes that $L = (N_z - 1)\Delta\mathbf{r}$ and the halfway that $L = (N_z - 2)\Delta\mathbf{r}$ or $N_z\Delta\mathbf{r}$ depending on the implementation (described above).

We define the relative error ϵ as $(|U_c - u_{N_{z/2}}|)/U_c$ where $u_{N_z/2}$ is the simulated velocity in the center of the channel. Figure 3.3 presents the relative error of the Poiseuille flow simulations for various boundary conditions.

We observe first order numerical accuracy in space for the fullway bounce-back condition as well as for the mass conserving boundary condition (details are presented below). The modified bounce-back and the halfway bounce-back present second order numerical accuracy in space. We can also observe that for a same number of lattice sites the error on the halfway bounce-back condition is smaller according to its specific implementation. Finally, note that the halfway bounce-back condition generates a velocity slightly higher than U_c while the fullway bounce-back produces a velocity slightly lower than U_c. Due to the relative error, this effect does not appear on figure 3.3.

The code 3.6 presents an implementation of the fullway bounce-back boundary condition. This code and the following are written in C++ which is, in the present form, quite close to pseudo code. We show also two implementations of the halfway bounce-back boundary condition. First, the code 3.7 presents

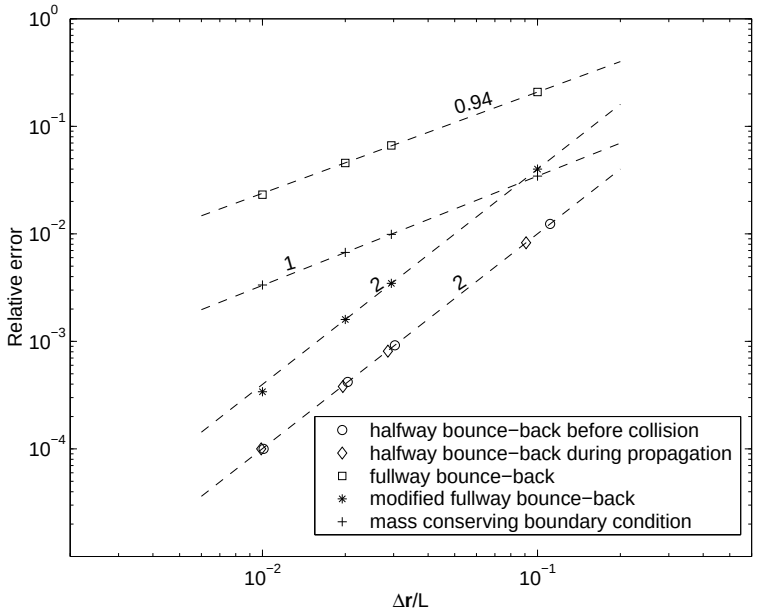


Figure 3.3: Relative errors $\epsilon = |U_c - u_{N_z/2}|/U_c$ of an LBGK Poiseuille flow simulation on a D2Q9 lattice. U_c and $u_{N_z/2}$ are the theoretical and simulated velocities in the center of the channel respectively. The lattice has $N_x = 17 \times N_z$ sites and is horizontally periodic, $\nu = 0.001$, $\rho = 1.0$ and $\tau = 1.0$. The flow is generated by a body force $G_x^* = 0.1$ along the x direction. The dashed lines are the best fits. Their slopes are reported in a log-log plot. From these slopes, on deduce that $\epsilon \sim (\Delta \mathbf{r}/L)^\alpha$, thus giving the order of accuracy.

the intuitive but non-local implementation. Second, through the modification of code 3.5, the local implementation is presented in code 3.8

Code 3.6 Fullway bounce-back boundary condition function.

```
int oppositeOf[9]={0,3,4,1,2,7,8,5,6};     // the opposite of each direction
                                                                              
void applyFullWayBounceBackOnSite(const int i, const int j)
{
  for (int k=0;k<nbNeighbor;k++) fout[i][j][k]=fin[i][j][oppositeOf[k]];
}
```

3.3.2 Non-slip boundary condition

To circumvent the slip velocity generated by bounce-back boundary conditions (fullway and halfway), Inamouro et al. [52] proposed a non-slip boundary condi-

Code 3.7 Non-local variant of the halfway bounce-back boundary condition function.

```
// Gets the indices of neighbor (iN,jN) of site (i,j) in direction k.
void getNeighborIndicesOf(const int i, const int j, const int k,
                          int &iN, int &jN);
// Returns true if the field in direction k of site (i,j) points on a solid area.
bool PointOnSolid(const int i, const int j, const int k);

void applyHalfWayBounceBackonSite(const int i, const int j)
{
  int iN,jN;
  for (int k=1;k<nbNeighbor;k++) {
    if (PointOnSolid(i,j,k)) {
      getNeighborIndicesOf(i,j,oppositeOf[k],iN,jN);
      fin[iN][jN][oppositeOf[k]]=fin[i][j][k];
    }
    else {
      getNeighborIndicesOf(i,j,k,iN,jN);
      fin[iN][jN][k]=fin[i][j][k];
    }
  }
}
```

tion. The slip velocity is defined as the velocity measured at the wall. Hence, many boundary conditions can be considered as non-slip boundary conditions (e.g. modified bounce-back) but with different numerical accuracy. However, in what follows, we will call non-slip boundary condition the Inamouro's condition.

The idea is to assume that the unknown fields are an equilibrium distribution function with a counter slip velocity. The counter slip velocity is determined so that the fluid velocity at the wall is equal to the wall velocity. The collision process can occur at the boundary as all fields are known.

In order to illustrate the Inamouro method, we consider a particular geometry of the wall. Figure 3.4 presents the two configurations which will be treated in this section. They consist in simple flat boundaries in 2D and 3D.

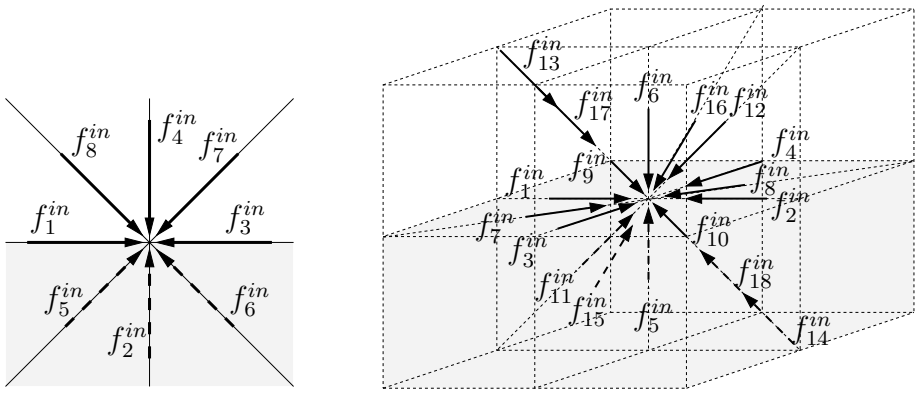
We first present the two-dimensional case (D2Q9) as it is given in [52] and next, we derive the non-slip boundary condition for the D3Q19 lattice.

From equation (3.10), the unknown fields of the example of figure 3.4(a) lead

Code 3.8 Local variant of the halfway bounce-back boundary condition function which is a modification of code 3.5.

```
// Indicates whether f[ii][jj] is a fluid site or not.
const bool isAFluidSite(const int ii, const int jj);

void propagationBis()
{
  int i,j,k;
  int ii,jj;
   // Propagates the fields.
  for (i=0;i<nx;i++)
    for(j=0;j<nz;j++)
      for (k=0;k<9;k++) {
        ii=(i+v[k][0]+nx)%nx;
        jj=(j+v[k][1]+ny)%ny;
         // Propagates only if (ii,jj) is a fluid site.
        if (isAFluidSite(ii,jj) == true)
          fin[ii][jj][k]=fout[i][j][k];
        else
           // This field will then be propagated during the next time step.
          fout[i][j][oppositeOf[k]]=fin[i][j][k];
      }
}
```



(a) (b)

Figure 3.4: Flat boundary sites after a propagation step of (a) a D2Q9 lattice and (b) a D3Q19 lattice. The unknown fields are drawn with thick dashed arrows. The gray areas represent the solid part.

to

$$f_2^{in} = \frac{1}{9}\rho'\left(1+3u_{wz}+\frac{9}{2}{u_{wz}}^2-\frac{3}{2}\left[(u_{wx}+u'_x)^2+{u_{wz}}^2\right]\right) \quad (3.22)$$

$$f_5^{in} = \frac{1}{36}\rho'\big(1+3(u_{wx}+u'_x+u_{wz})+\frac{9}{2}(u_{wx}+u'_x+u_{wz})^2 - \frac{3}{2}\left[(u_{wx}+u'_x)^2+{u_{wz}}^2\right]\big) \quad (3.23)$$

$$f_6^{in} = \frac{1}{36}\rho'\big(1+3(-u_{wx}-u'_x+u_{wz})+\frac{9}{2}(-u_{wx}-u'_x+u_{wz})^2 - \frac{3}{2}\left[(u_{wx}+u'_x)^2+{u_{wz}}^2\right]\big) \quad (3.24)$$

where $f_i^{in} = f_i^{eq}(\rho', \mathbf{u}_w + \mathbf{u}')$ denotes a field entering a site at velocity $\mathbf{v}_i$ before the collision process, u_{wx} and u_{wz} are the x and z components of the wall velocity $\mathbf{u}_w$, and ρ' and $\mathbf{u}'$ are unknown parameters, $\mathbf{u}' = (u'_x, 0)$ being the counter slip velocity. The counter slip velocity component normal to the wall is supposed to be zero because of an assumption of diffuse reflection [52]. The two unknown parameters are determined by the condition that the fluid velocity at the wall is equal to the wall velocity, i.e. $\sum f_i^{in}\mathbf{v}_i = \rho_w\mathbf{u}_w$. Moreover, the density at the wall, ρ_w, is an unknown quantity and is computed by equation (3.7). The solutions are

$$\rho_w = \frac{1}{1-u_{wz}}\left[f_0^{in}+f_1^{in}+f_3^{in}+2(f_4^{in}+f_7^{in}+f_8^{in})\right] \quad (3.25)$$

$$\rho' = 6\frac{\rho_w u_{wz}+(f_4^{in}+f_7^{in}+f_8^{in})}{1+3u_{wz}+3{u_{wz}}^2} \quad (3.26)$$

$$u'_x = \frac{1}{1+3u_{wz}}\left[6\frac{\rho_w u_{wx}-(f_1^{in}-f_3^{in}+f_8^{in}-f_7^{in})}{\rho'}-u_{wx}-3u_{wx}u_{wz}\right] \quad (3.27)$$

We now derive the non-slip boundary condition for the D3Q19 lattice considering a null velocity at a wall which is perpendicular to the z-direction, i.e. $\mathbf{u}_w = 0$. The same scheme can be used to express the non-slip boundary condition for a moving wall. The unknown fields are represented in figure 3.4(b). The density at the wall is expressed as

$$\rho_w = \sum m_i f_i^{in} = \sum_{fluid} m_i f_i^{in} + \sum_{solid} m_i f_i^{in} + \sum_{wall} m_i f_i^{in} \quad (3.28)$$

and the velocity at the wall $\mathbf{u}_w$ as

$$\rho_w\mathbf{u}_w = 0 = \sum m_i f_i^{in}\mathbf{v}_i = \sum_{fluid} m_i f_i^{in}\mathbf{v}_i + \sum_{solid} m_i f_i^{in}\mathbf{v}_i + \sum_{wall} m_i f_i^{in}\mathbf{v}_i \quad (3.29)$$

where *solid*, *fluid* and *wall* denote the direction belonging to the solid, the fluid and the wall respectively. The unknown fields (solid) are assumed to be the

equilibrium distribution function. From equation (3.10), one can then express

$$f^{in}_{i\in solid} = \rho' \frac{C_4}{{C_2}^2}\left[1 + \frac{v_{i\alpha}u'_\alpha}{c_s^2} + \frac{1}{2}\left(\frac{v_{i\alpha}u'_\alpha}{c_s^2}\right)^2 - \frac{\mathbf{u}'^2}{2c_s^2}\right] \tag{3.30}$$

where $\mathbf{u}'$ and ρ' have to be determined. For a D3Q19 model, we have that

$$\begin{aligned}
\sum_{solid} m_i v_{i\alpha} &= \begin{cases} 6v_l & \text{if } \alpha = z, \\ 0 & \text{otherwise.} \end{cases} \\
\sum_{solid} m_i v_{i\alpha} v_{i\beta} &= \begin{cases} 0 & \text{if } \alpha \neq \beta, \\ 2v_l^2 & \text{if } \alpha = \beta \neq z, \\ 6v_l^2 & \text{if } \alpha = \beta = z. \end{cases} \\
\sum_{solid} m_i v_{i\alpha} v_{i\beta} v_{i\gamma} &= \begin{cases} 6v_l^3 & \text{if } \alpha = \beta = \gamma = z, \\ 2v_l^3 & \text{if } \alpha = \beta \neq z \text{ and } \gamma = z, \\ 0 & \text{otherwise.} \end{cases}
\end{aligned}$$

We suppose again that the normal counter slip velocity component is null at the wall [52]. Hence, the counter slip velocity expresses as $\mathbf{u}' = (u'_x, u'_y, 0)$.

Multiplying equation (3.30) by m_i and $v_{i\gamma}$ and summing over $i \in solid$ gives

$$\begin{aligned}
\sum_{solid} m_i f_i^{in} v_{i\gamma} &= \rho' \frac{C_4}{{C_2}^2}\left[\sum_{solid} m_i v_{i\gamma} + \frac{1}{c_s^2}\sum_{solid} m_i v_{i\alpha} v_{i\gamma} u'_\alpha \right. \\
&+ \left. \frac{1}{2c_s^4}\sum_{solid} m_i v_{i\alpha} v_{i\beta} v_{i\gamma} u'_\alpha u'_\beta - \frac{1}{2c_s^2}\sum_{solid} m_i v_{i\gamma} \mathbf{u}'^2\right] \\
&= \rho' \frac{C_4}{{C_2}^2}\left[6v_l\delta_{\gamma z} + \frac{1}{c_s^2} 2v_l^2 u'_\gamma \delta_{\alpha\gamma\neq z} \right. \\
&+ \left. \frac{1}{2c_s^4} 2v_l^3 \mathbf{u}'^2 \delta_{\alpha\beta\neq z,\gamma z} - \frac{1}{2c_s^2} 6v_l \mathbf{u}'^2 \delta_{\gamma z}\right]
\end{aligned}$$

where $\delta_{\alpha\beta}$ is the Kronecker function equal to 1 if $\alpha = \beta$. Hence after some algebra we have

$$\sum_{solid} m_i f_i^{in} v_{i\gamma} = \begin{cases} \frac{v_l \rho'}{6} & \text{if } \gamma = z, \\ \frac{u'_\gamma \rho'}{6} & \text{if } \gamma \neq z. \end{cases} \tag{3.31}$$

The z component of the equation obtained after injection of equation (3.31) in equation (3.29) allows us to express ρ' as

$$\rho' = 6 \sum_{fluid} m_i f_i^{in} \tag{3.32}$$

because the z-component of the velocity vectors $\mathbf{v}_i$ belonging to the wall is null and the one of the $fluid$ is equal to $-v_l$. The x and y components of the equation

obtained after combining equation (3.32) and equation (3.29) allow to express the counter slip velocity as

$$u'_\gamma = -\frac{6}{\rho'} \sum_{fluid+wall} m_i f_i^{in} v_{i\gamma} \qquad \gamma = x, y. \tag{3.33}$$

Hence, the unknown fields of a flat D2Q9 and D3Q19 wall can be determined. The collision process can occur at the boundary as all fields are known.

The non-slip boundary condition is illustrated by a 2D LBGK Poiseuille flow simulation [13, 1]. The D2Q9 lattice has $N_x = 17 \times N_z = 11$ sites. It is horizontally periodic. The flow is generated by a constant body force $G_x^* = 0.1$ along the x direction (see section 3.4 for details). The viscosity is fixed to $\nu = 0.001$ and the relaxation time τ is, in this case, a variable. Assume that $\Delta\mathbf{r}/\Delta t = C$ where C is a constant value. Let say that $C = 1$ where unities are intentionally omitted for simplicity reasons. We then find from equation (3.12) that $\Delta\mathbf{r} = (3\nu)/(\tau - 1/2)$. A derivation of the Navier-Stokes equation (equation (2.6)) for this specific case allows us to express the velocity in the center of the channel as $U_c = \frac{L^2\rho}{8\nu}$ where L is the height of the channel and ρ is the density fixed here to 1.0. The bounce-back boundary condition and the non-slip boundary condition impose that $L = (N_z - 1)\Delta\mathbf{r}$. Note that u_{wx} is set to $-\Delta\mathbf{r}\, G_x^*$ as the forcing also occurs at the boundary.

Results are presented in figure 3.5. Several relaxation times have been taken into account. The velocity profile obtained with a non-slip boundary condition does not vary with relaxation time ($0.55 \leq \tau \leq 30.0$). This is not the case for the bounce-back condition for which we observe the relation between the slip velocity and the relaxation time. A wide discussion about this relation can be found in [49].

The previous Poiseuille flow was proposed by Inamouro et al. as an example in [52]. In their paper they also observed the effect of their boundary condition on a Couette flow. Both examples present some symmetries. The results of their simulations using the non-slip boundary condition lead to a very good numerical accuracy and to the exact solution for the Poiseuille flow. But considering a less symmetrical flow (e.g. cavity flow), we observe that mass is not conserved for small lattices. Note that this depends on the initial condition.

To better understand this non-conservation of mass, let us again consider the 2D case illustrated in figure 3.4(a). Mass balance due to the interaction at the wall can be expressed as

$$(f_2^{out} + f_5^{out} + f_6^{out}) - (f_4^{in} + f_7^{in} + f_8^{in}) = \rho_{out} - \rho_{in} = \Delta_\rho. \tag{3.34}$$

Hence, mass is conserved if $\Delta_\rho = 0$. Let us rewrite equation (3.34) in function of basic quantities: ρ, $\mathbf{u}$ and their spatial derivatives. Recall that equation (3.6) allows us to express

$$f_i^{out} = (1 - 1/\tau) f_i^{in} + 1/\tau f_i^{eq}. \tag{3.35}$$

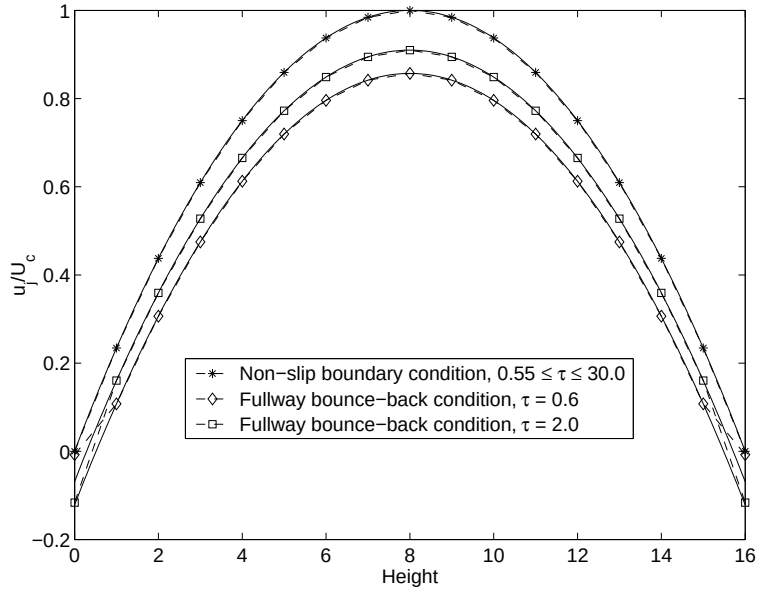


Figure 3.5: Velocity profiles of a D2Q9 LBGK Poiseuille flow simulation. U_c is the theoretical velocities in the center of the channel. u_j is the velocity at height j. The lattice has $(N_x = 17) \times (N_z = 11)$ sites and is horizontally periodic, $\nu = 0.001$, $\rho = 1.0$. The flow is generated by a body force $G_x^* = 0.1$ along the x direction. The stars report the representative velocity profile obtained with a non-slip condition for $0.55 \leq \tau \leq 30.0$. Other symbols are velocity profiles obtained with a fullway bounce-back condition. The solid lines are the best fit for which both wall sites have not been considered. We measure that parabola differ from another not only by an offset. Note also that the velocity is measured after the application of the boundary condition.

Following equation (3.13), if the wall velocity is null then the equilibrium distribution function is simply expressed as

$$f_i^{eq} = \rho_w t_i \tag{3.36}$$

where ρ_w is the density at the wall. Moreover, the dominant contribution in the Chapman-Enskog expansion [9] allows us to decompose any field f_i^{in} as a sum of its equilibrium f^{eq} and non-equilibrium part f^{neq}. Thus one has

$$f_i^{in} = f_i^{eq} + f_i^{neq} \tag{3.37}$$

with the non-equilibrium part defined as [37]

$$\begin{aligned} f_i^{neq} &= \frac{\Delta t\ \tau}{C_2 v_l^2}(v_{i\gamma} v_{i\delta} \partial_\gamma \rho u_\delta - c_s^2 \mathrm{div} \rho \mathbf{u}) \\ &= \frac{\Delta t\ \tau}{C_2 v_l^2}(v_{i\gamma} v_{i\delta} \partial_\gamma \rho u_\delta - c_s^2 (\partial_x \rho u_x + \partial_y \rho u_y)). \end{aligned} \tag{3.38}$$

Thus, using equations (3.35) and (3.36), the quantity $\rho_{out} = f_2^{out} + f_5^{out} + f_6^{out}$ (equation (3.34)) can be expressed as

$$\begin{aligned}\rho_{out} &= f_2^{out} + f_5^{out} + f_6^{out} \\ &= (1 - \frac{1}{\tau})(f_2^{in} + f_5^{in} + f_6^{in}) + \frac{1}{\tau}(f_2^{eq} + f_5^{eq} + f_6^{eq}) \\ &= (1 - \frac{1}{\tau})(f_2^{in} + f_5^{in} + f_6^{in}) + \frac{\rho_w}{6\tau}. \end{aligned} \tag{3.39}$$

Using the decomposition into an equilibrium and a non-equilibrium part equation (3.37), one can write equation (3.39) as

$$\begin{aligned}\rho_{out} &= (1 - \frac{1}{\tau})\left(\frac{\rho_w}{6} + \frac{\Delta t\ \tau}{C_2 v_l^2}(v_l^2 \partial_y \rho u_y + (2v_l^2 - 3c_s^2)\mathrm{div}\rho\mathbf{u})\right) + \frac{\rho_w}{6\tau} \\ &= \frac{\rho_w}{6} + \frac{\Delta t\ \tau(1 - 1/\tau)}{C_2 v_l^2}(v_l^2 \partial_y \rho u_y + (2v_l^2 - 3c_s^2)\mathrm{div}\rho\mathbf{u}). \end{aligned} \tag{3.40}$$

Similarly, one can express the quantity $\rho_{in} = f_4^{in} + f_7^{in} + f_8^{in}$ as

$$\begin{aligned}\rho_{in} &= f_4^{in} + f_7^{in} + f_8^{in} \\ &= \frac{\rho_w}{6} + \frac{\Delta t\ \tau}{C_2 v_l^2}(v_l^2 \partial_y \rho u_y + (2v_l^2 - 3c_s^2)\mathrm{div}\rho\mathbf{u}). \end{aligned} \tag{3.41}$$

Finally, equation (3.34) can be rewritten as

$$\Delta_\rho = \rho_{out} - \rho_{in} = -\frac{\Delta t}{C_2 v_l^2}(v_l^2 \partial_y \rho u_y + (2v_l^2 - 3c_s^2)\mathrm{div}\rho\mathbf{u}) \tag{3.42}$$

Remember that equation (3.42) has been obtained by supposing that the wall velocity is zero. A quite tedious algebraic expression should result when considering a moving wall. To avoid this laborious part, we use numerical examples and exhibit that locally the mass is not conserved when one applies the non-slip boundary condition on such a wall site.

From equation (3.42) we observe that mass is conserved under the non-slip boundary condition only if the y-component of the momentum gradient is null and if the LB fluid can be considered as incompressible implying that $\mathrm{div}\rho\mathbf{u} = 0$. From our experiments, it appears that velocity gradients at the wall are zero once the flow is established. But velocity gradients appear in the settling part especially when the geometry is asymmetrical.

We now understand better why, considering a wall at rest, the simulation of a Poiseuille flow with non-slip boundary conditions conserves mass. Indeed, because of symmetry, there is no velocity gradient for u_y in the y direction at any time (supposing a uniform initialization) and the fluid can be considered as incompressible.

Let us now concentrate on the cavity flow [13] which is a flow in a square cavity settled by a constant acceleration on its top. It is modeled by an $N \times N$ D2Q9 lattice where the walls are treated with a non-slip boundary condition except the top wall where $U_{top} = 0.1$ is imposed through the equilibrium distribution function (see section 3.4.3 for details). The two bottom corners are treated with a bounce-back boundary condition as, in the present form, the non-slip boundary condition does not allow us to deal with such a configuration. The Reynolds number is equal to 100.

We first consider the simulation of a cavity flow initialized by setting all the D2Q9 fields to 1/9 (an initial set to $t_i\rho$ is considered below). The time evolution of the density of the bottom wall site located halfway of the vertical walls is reported in figure 3.6(a) for various lattice sizes. For the same size, we show in figure 3.6(b) the global density which is the sum of all site densities divided by N^2. We observe that the system tends to reach its local equilibrium (drawn as a dashed line in figure 3.6(a)). Everything goes right until the wall effects are perceptible at the examined site. Depending on the lattice size these effects are too big to be absorbed by the system and numerical instabilities appear. This is the case when on consider a 17×17 lattice. The wall effects are also shown in figure 3.6(b).

Second, we initially set the fields to their local equilibrium. Again, density of the bottom wall site located halfway of the vertical walls as well as the global density divided by N^2 is reported for various lattice size in figure 3.6(c) and (d) respectively. We observe first that the system seems to deal better with the wall effects but after a little more than 80 iterations it also numerically blows up for a 17×17 lattice.

We conclude that this non-slip boundary condition is very sensitive to initial conditions. So an appropriate setting is crucial in order to avoid numerical instabilities. On the other hand, it is not obvious to predict the effects of these density variations on bigger Reynolds number flows for which density variations should have more influence.

Finally, note that we tried to initialize a 17×17 D2Q9 cavity lattice considering non-slip boundary conditions with the steady results of the same simulation but considering bounce-back boundary conditions. It is interesting to observe that the system does not blow up with this special initialization.

3.3.3 Mass conserving boundary condition

A boundary condition preserving the total amount of mass in a given system is obviously an important issue. Moreover, applying the collision operator on the wall leads to a more continuous velocity profile. We define a new boundary condition called mass conserving boundary condition. Basically, the idea is to impose a null velocity at the wall as well as mass conservation. This leads to three equations with usually more than three unknowns, when a 2D system is

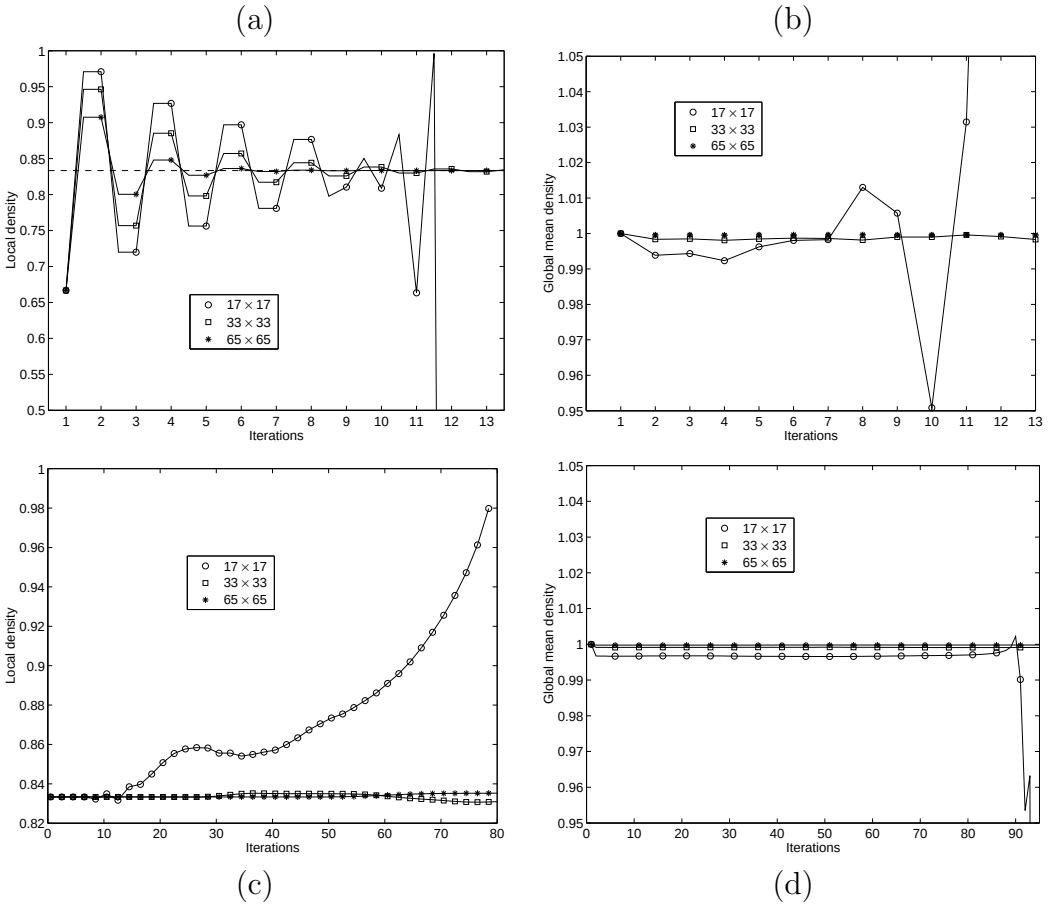


Figure 3.6: Local and global densities of cavity flows in function of simulation time for various D2Q9 $N \times N$ lattice size. The local density is the bottom wall site density located halfway of the vertical walls while the global density is the sum of local densities over all sites divided by N^2. The top velocity is set to 0.1 through the equilibrium distribution function. The Reynolds number is equal to 100. On (a) and (b) we initially set all fields to 1/9 while on (c) and (d) fields are set to their equilibrium value. We observe numerical instabilities for small lattice size.

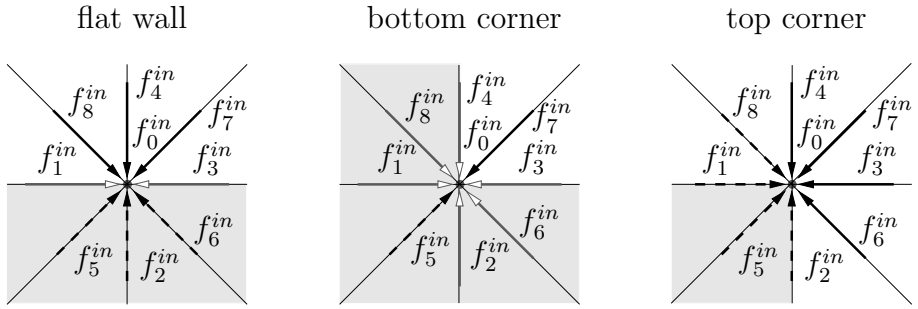


Figure 3.7: Three wall site configurations for which the mass conserving boundary condition is developed. We use plain arrows, dashed arrows and gray arrows to draw the fluid, solid and wall fields respectively. Gray areas represent the solid.

considered.

The fields on a wall site can be decomposed into three categories: the fluid fields (incoming from the fluid), the solid fields (outcoming from the wall) and some free fields (wall fields) which do not influence the system at all. The free fields can be temporarily used to close the system. They are set to zero just after determination of the wall fields (i.e. the unknowns).

Three specific configurations, which are used in the simulations below, are now presented. They are illustrated in figure 3.7.

The configuration presented in figure 3.7(c) will be used to simulate the top corners of a cavity flow which are considered as fluid sites. A major condition in the system is to conserve mass. Thus, fields which want to go out the domain are mirrored in order to be conserved. This is the reason why fields f_4^{in} and f_7^{in} are considered as fluid fields (plain arrows in figure 3.7(c)).

Let us start with the flat wall configuration illustrated in figure 3.7(a). Mass conservation is expressed by

$$f_2^{out} + f_5^{out} + f_6^{out} = f_4^{in} + f_7^{in} + f_8^{in} \tag{3.43}$$

and the velocity at the wall by

$$\sum_{i=1}^{8} f_i^{in} \mathbf{v}_i = 0. \tag{3.44}$$

We want to compute f_0^{in}, f_1^{in}, f_3^{in}, f_2^{in}, f_5^{in} and f_6^{in} so that, after a regular collision step f_2^{out}, f_5^{out} and f_6^{out} obey equation (3.43). An additional constraint to find the unknown f_i^{in} is the no-slip condition (equation (3.44)).

Hence, as the velocity is imposed to be null at the wall, the equilibrium distribution function is expressed as $f_i^{eq} = t_i \rho_w$ where ρ_w is the density at the

wall ($\rho_w = \sum f_i^{in}$). Considering that f_i^{out} can be written as $(1-1/\tau)f_i^{in}+(1/\tau)f_i^{eq}$, one can rewrite equation (3.43) as

$$\begin{aligned}
f_2^{out} + f_5^{out} + f_6^{out} &= f_4^{in} + f_7^{in} + f_8^{in} \\
(1 - \frac{1}{\tau})\left(f_2^{in} + f_5^{in} + f_6^{in}\right) + \frac{1}{\tau}\left(f_2^{eq} + f_5^{eq} + f_6^{eq}\right) &= f_4^{in} + f_7^{in} + f_8^{in} \\
(1 - \frac{1}{\tau})\left(f_2^{in} + f_5^{in} + f_6^{in}\right) + \frac{1}{\tau}\frac{\rho_w}{6} &= f_4^{in} + f_7^{in} + f_8^{in} \\
(1 - \frac{1}{\tau})\left(f_2^{in} + f_5^{in} + f_6^{in}\right) + \frac{1}{6\tau}\left(f_0^{in} + \cdots + f_8^{in}\right) &= f_4^{in} + f_7^{in} + f_8^{in} \\
(1 - \frac{5}{6\tau})\left(f_2^{in} + f_5^{in} + f_6^{in}\right) &= (1 - \frac{1}{6\tau})\left(f_4^{in} + f_7^{in} + f_8^{in}\right) \\
&- \frac{1}{6\tau}\left(f_0^{in} + f_1^{in} + f_3^{in}\right) \qquad (3.45)
\end{aligned}$$

The two components of equation (3.44) yield

$$f_1^{in} + f_5^{in} - f_6^{in} - f_3^{in} - f_7^{in} + f_8^{in} = 0 \qquad (3.46)$$

$$f_2^{in} + f_5^{in} + f_6^{in} - f_4^{in} - f_7^{in} - f_8^{in} = 0. \qquad (3.47)$$

Thus we have three equations and six unknowns. There is some degrees of freedom that we call the free fields. We choose their value so as to have the more symmetrical solution.

Defining $\rho_{in} = f_4^{in} + f_7^{in} + f_8^{in}$ and using equation (3.47), one can reformulate equation (3.45) and find a relation between the free fields

$$f_3^{in} = 4\rho_{in} - f_1^{in} - f_0^{in}. \qquad (3.48)$$

We close the system by imposing the following relations

$$f_0^{in} = 0\,, \qquad f_1^{in} = f_3^{in}\,, \qquad f_5^{in} = f_7^{in}. \qquad (3.49)$$

Hence one can find with relations 3.49 used in equations (3.46), (3.47) and (3.48) that

$$\boxed{\begin{array}{c} f_0^{in} = 0, \\ f_1^{in} = f_3^{in} = 2\rho_{in}, \qquad f_5^{in} = f_7^{in} \\ f_6^{in} = f_8^{in}, \qquad f_2^{in} = f_4^{in} \end{array}} \qquad (3.50)$$

We observe that condition 3.50 also preserves the lattice symmetry and that this condition is a kind of modified bounce-back in which we specify adequately the unused fields in order to conserve mass.

Following the same ideas, the unknown fields of figure 3.7(b) can be determined. One finds

$$\begin{aligned} f_0^{in} &= 0, & f_5^{in} &= f_7^{in} \\ f_2^{in} &= f_4^{in}, & f_6^{in} &= f_8^{in} \\ f_1^{in} &= 8f_7^{in} & f_3^{in} &= 8f_7^{in} \end{aligned} \tag{3.51}$$

In the same way, the unknown fields of figure 3.7(c) can be expressed as

$$\begin{aligned} f_0^{in} &= \tfrac{8}{5}(f_3^{in} + f_4^{in} + f_6^{in} + f_7^{in}), \\ f_5^{in} &= f_7^{in}, & f_2^{in} &= f_4^{in} \\ f_6^{in} &= f_8^{in}, & f_1^{in} &= f_3^{in} \end{aligned} \tag{3.52}$$

To validate the previous results, we focus our attention on the simulation of the cavity flow [13] by considering two types of boundary conditions: the fullway bounce-back and the mass conserving boundary conditions. The Inamouro non-slip boundary condition is not considered because it does not conserve mass (numerical instabilities appear for small systems) and it is non applicable on corners. The cavity is modeled by a D2Q9 lattice composed of $N \times N$ sites. The flow is settled on its upper line by setting at each iteration a velocity equal to $\mathbf{U}_{top} = (0.1, 0)$ through the equilibrium distribution function, see section 3.4.3. Note that the velocity is also imposed at both top corners. We consider a Reynolds number equal to 100. The viscosity is then expressed as $U_{top}L/Re$ where L is equal to $(N-1)\Delta\mathbf{r}$. Thus following equation (3.12) for the viscosity, the relaxation time is expressed as $1/2 + 0.3(N-1)$.

Figure 3.8 presents a zoom of the x velocity profiles located on a line in the middle of the cavity close to the bottom wall, i.e. $x = L/2$ and $z \in [0, 0.06 \cdot L]$. The bounce-back and the mass conserving boundary condition have been used. We observe that the velocity profiles is smooth as one gets closer to the wall when the mass conserving boundary condition is considered. It is essentially due to the application of the collision process also on the boundary. We note as well that velocities increase with the lattice resolution when the mass conserving boundary condition is used while this is the opposite in the bounce-back case.

We report in figure 3.9 a relative error in order to appraise the convergence to the theoretical velocity profile. The unknown theoretical profile is again approximated by the simulated velocity profile on a 513×513 lattice. The relative error is defined as the mean square of the difference between simulated and theoretical profiles. This error is of the second order for approximately all three boundary

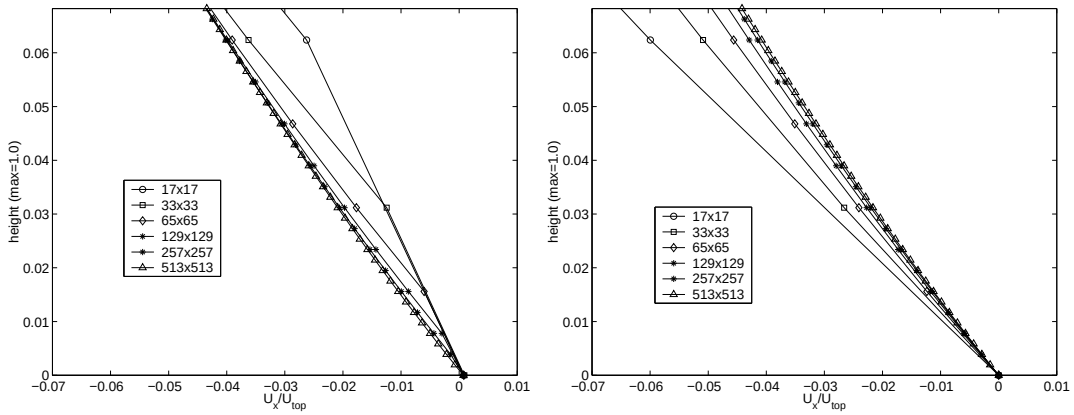


Figure 3.8: Zoom of the x velocity profiles located in the middle of the cavity close to the bottom wall. Profiles are simulated on various lattice size $N \times N$ depicted by different symbols. On the left and on the right the simulation results are reported under a bounce-back boundary condition and under the mass conserving boundary condition respectively. The Reynolds number is 100.

conditions we considered.

Hence the numerical accuracy is, in this case, of second order in space for both bounce-back boundary conditions. This indicates that the convergence of fullway and halfway bounce-back conditions are not always of first and second order respectively.

Finally, figure 3.10 shows the center positions of the primary vortex generated by a cavity flow at a Reynolds number of 100. The positions of the center have been obtained by using a spline interpolation. Various boundary conditions and lattice size have been considered. We approximate the unknown theoretical center position as the result obtained by the simulation on a lattice of size 513×513. They are slightly different depending on the boundary condition used. However, these centers are in agreement with previous studies (references can be found in [59]) whose results are reported on figure 3.10. Although they used the same model, Hou et al. [59] found slightly different results. This is probably due to the way they measure the vortex center. They presumably do not interpolate.

3.3.4 Free boundary

The last boundary condition presented in this section is the free boundary condition. It consists of a boundary between a fluid and an other (actually not simulated) where their frictions are not taken into account. Hence, the tangential motion of the fluid flow on the boundary is free and no momentum is to be exchanged with the boundary along the tangential component. This boundary is usually rectilinear and of constant shape. Let us note that we call free surfaces the boundaries for which fluids influence each other. Such boundaries are

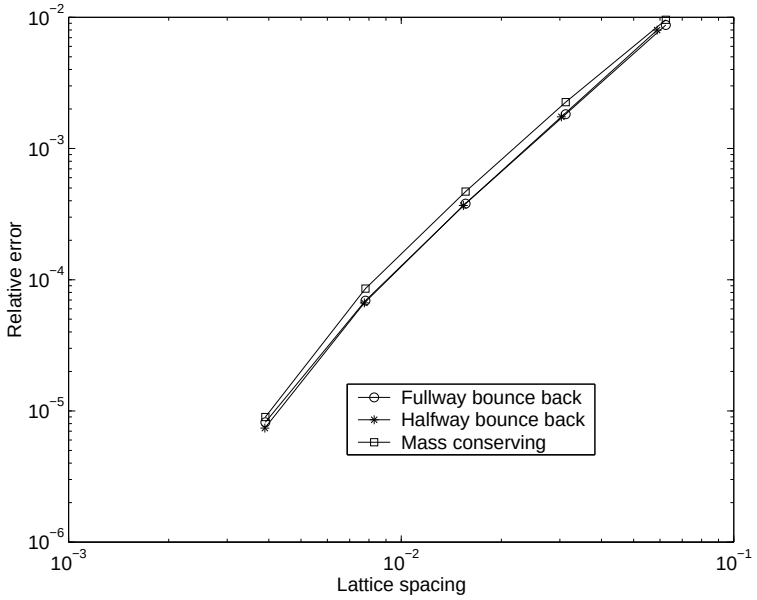


Figure 3.9: Convergence of simulated x velocity profiles in the middle (i.e. $x = L/2$) of a $L \times L$ cavity flow at a Reynolds number equal to 100. The real cavity size is equal to $L = 1 = (N-1)\Delta\mathbf{r}$ where N is the number of lattice sites and $\Delta\mathbf{r}$ is the lattice spacing. Fullway bounce-back, halfway bounce-back and mass conserving boundary conditions have been considered. The relative error is computed as the mean square of the difference between simulated and theoretical profiles. The unknown theoretical profile is approximated by the simulated velocity profile on a 513×513 lattice.

quite complicated and are not treated in this thesis. The interested reader can nevertheless find some pointers in [12].

Depending on the simulation and on the expected results, we use two ways to deal with such boundaries. First, we use the **bounce-forward condition**. On free boundary sites, it consists in reflecting particles in a specular way. One can easily show that this condition implies no tangential momentum transfer [12]. To illustrate this condition, figure 3.11 sketches the bounce-forward of a particle entering a horizontal boundary site in direction 5.

Second, we can set the local velocity of free boundary sites according to their neighborhood in order to impose some null velocity gradients normal to the boundary.

3.3.5 Discussion

The bounce-back boundary condition is definitely the most simple condition to use especially because of its common and easy formulation. However we saw

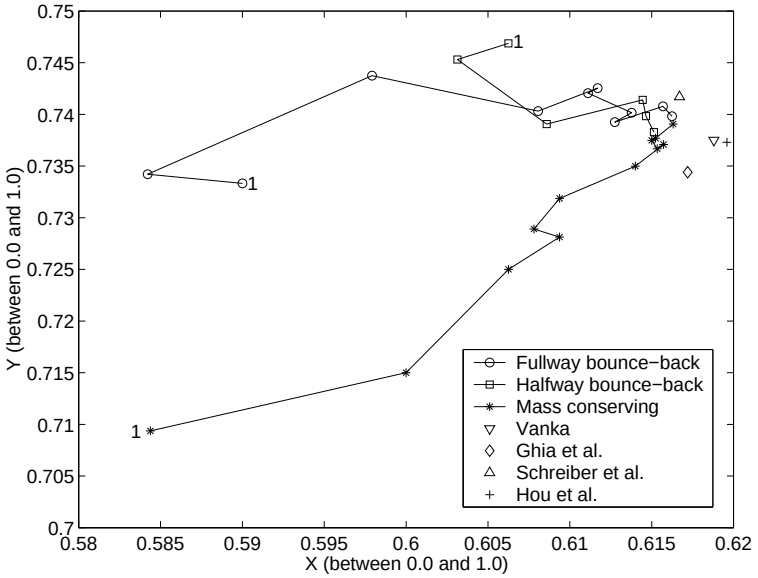


Figure 3.10: The center positions of the primary vortex in a cavity flow are reported for various lattice size and boundary conditions. The Reynolds number is equal to 100. The center positions are interpolated and are usually not located on a lattice point. Circles, squares and stars are the results of our simulations considering fullway bounce-back, halfway bounce-back and mass conserving boundary conditions respectively. The centers positions are on a way starting from the coarsest grid (indicated by a label 1) until the finest one. Results obtained by others have also been depicted (all references can be found in [59]). All measurements agree within 0.5%.

Figure 3.11: Bounce-forward of a particle entering a horizontal boundary site in direction 5.

that its simplicity appreciably may affect the numerical accuracy. Moreover, a slip velocity as well as a discontinuity in velocity profiles appear when it is used. Thus, depending on the flow and on the question one is interested to answer, an other condition can be used.

The non-slip boundary condition proposed by Inamouro et al. [52] works quite well for the Poiseuille flow. It is developed for flat boundaries. We observed and explained that this condition does not conserve mass when there are velocity gradients normal to the boundary, or compressibility effects.

To remedy these problems, we proposed a new boundary condition called mass conserving boundary condition. The main idea is to ensure mass conservation and a null velocity on boundary sites. The continuity of velocity profiles is also improved by giving the opportunity to apply the collision operator on boundary sites. However for the flow we tried, numerical accuracy is not better than those obtained with bounce-back conditions.

3.4 Flow settlement

In this section, we deal with the problem of flow settlement. Independently of a lattice field initialization, one has to stimulate the fluid in order to obtain a desired flow. The initialization is usually done so as to have a fluid at rest. We then talk about flow settlement. We present three ways to settle a flow. First, we consider the application of a constant body force representing typically the gravity. Second, we show how to deal with a pressure gradient. Finally, we present a way to set a velocity profile locally in order to accelerate all the fluid. Other methods as well as references can be found in [12].

3.4.1 Body force

A common way to settle a flow is to impose on each lattice site a constant body force $\mathbf{G}^* = \mathbf{G}/\rho^{real}$ where ρ^{real} is the real fluid density and $\mathbf{G}$ is the real gravitational vector. Hence equation (3.6) can be rewritten as

$$f_i^{in}(\mathbf{r} + \Delta t\ \mathbf{v}_i, t + \Delta t) = \frac{1}{\tau} f_i^{eq}(\mathbf{r}, t) + (1 - \frac{1}{\tau}) f_i^{in}(\mathbf{r}, t) + g_i \qquad (3.53)$$

where the forcing terms g_i are given by

$$\begin{aligned} g_i &= \frac{m_i \Delta t}{C_2 v_l^2} v_{i\alpha} G_\alpha^* \\ &= \frac{t_i C_2 \Delta t}{C_4 v_l^2} v_{i\alpha} G_\alpha^*. \end{aligned}$$

By summing equation (3.53) over i, one can show that the density of the system is unchanged. But considering the momentum which is obtained from

equation (3.53), which amounts to multiplying equation (3.53) by $v_{i\beta}$ and summing over i, one finds that a quantity equal to $\Delta t\ \mathbf{G}^*$ is added to each lattice site. This corresponds to the amount added by the gravitational term in the Navier-Stokes equation.

Some simple flows allow us to find a relation between a characteristic velocity of the flow and the body force necessary to its settlement. This is for example the case of the 2D Poiseuille flow at low Reynolds number (< 2000) which yields a laminar regime. Indeed, considering the Navier-Stokes equation, we observe that both terms $\partial\mathbf{U}/\partial t$ and $\mathbf{U}\cdot\nabla\mathbf{U}$ are null because of the stationarity, respectively the incompressibility of the flow. Thus, assuming that there is no pressure gradient, one can express the body force as

$$\frac{\mathbf{G}}{\rho^{real}} = \mathbf{G}^* = -\nu^{real}\nabla^2\mathbf{U} \tag{3.54}$$

where ν^{real} and ρ^{real} are the viscosity respectively the density of the real fluid and where $\mathbf{U}$ is well known for this simple flow [13]. Indeed, the transversal velocity is null and the longitudinal velocity is expressed as

$$U_x(y) = 4U_c\frac{y}{D}\left(1 - \frac{y}{D}\right) \tag{3.55}$$

where U_c is the maximum velocity in the center of the channel and D is the channel diameter. Differentiating equation (3.55) twice allows us to write

$$\partial_x^2 U_x = -\frac{8U_c}{D^2} \quad \text{and} \quad \partial_y^2 U_y = 0. \tag{3.56}$$

Replacing equation (3.56) into equation (3.54), one finds

$$G_x^* = \frac{8\nu^{real}U_c}{D^2} \quad \text{and} \quad G_y^* = 0. \tag{3.57}$$

Unfortunately, most flows are not as simple as a Poiseuille flow. Hence, body forces have to be set empirically to produce a desired flow. It is usually a small value typically equals to 10^{-5}. Note also that for large systems, the complete flow settlement may require an important amount of iterations. This amount increases with the Reynolds number. A simulation showing this point is presented at the end of this section (see figure 3.18). We give in section 4.6 a way to speed up this process by successively refining the lattice.

The body force application function consists of a few lines of code. They are presented for 2D lattices in code 3.9.

Note finally that the application of a constant body force seems to spoil turbulent fluctuations. Therefore, an other flow settlement condition should be used when one deals with (highly) turbulent flows.

Code 3.9 Body force application function for 2D lattices.

```
void applyBodyForceOnSite(const int i, const int j, const float H[2])
 // where H = ΔtC2/C4G*
{
  int k;
  for (k=0;k<nbNeighbor;k++)
    fout[i][j][k]+=t[i]*(v[k][0]*H[0]+v[k][1]*H[1]);
}
```

3.4.2 Pressure gradient

In laboratory experiments, flows are often driven by pressure gradients. Recall that pressure is directly related to density by equation (3.11). Hence, the use of this condition leads to a compressibility error $O(M^3)$. Reider et al. [60] showed that convergence of the incompressible equations is obtained only if the compressibility errors are smaller than the discretization error. Roughly speaking, one can argue that a small density gradient can be tolerated by the system as far as the lattice spacing is small enough.

One of the easiest ways to impose a pressure (density) gradient is to maintain a density difference between the inlet and the outlet of the simulation by adding a quantity ∇p^* at the inlet and subtracting the same quantity at the outlet.

For the sake of clarity, we now focus our attention on the Poiseuille flow on a channel of length L and diameter D. The channel is longitudinally periodic. The inlet is defined to be the first column and the outlet the last column. Note that a little adaptation is necessary when considering another flow but the main idea remains.

The lattice periodicity allows us to modify the inlet only. Indeed, values subtracted to the inlet on the backward directions will be propagated to the outlet. It then produces the same effect than subtracting these values at the outlet directly. Hence, one can rewrite equation (3.6) for the inlet sites only as

$$f_i^{in}(\mathbf{r}_{inlet} + \Delta t\ \mathbf{v}_i, t + \Delta t) = \frac{1}{\tau} f_i^{eq}(\mathbf{r}_{inlet}, t) + (1 - \frac{1}{\tau}) f_i^{in}(\mathbf{r}_{inlet}, t) + p_i \qquad (3.58)$$

where the pressure components p_i are given by

$$p_i = \frac{t_i}{c_s^2} \mathbf{v}_i \cdot \nabla p^*$$

where $\nabla p^* = L \nabla p / \rho^{real}$ is the desired pressure gradient between the inlet and the outlet (L is the distance between these regions).

Again, the simplicity of the Poiseuille flow allows us to obtain an expression between the velocity in the center of the channel and the pressure gradient. Recall that both terms $\partial \mathbf{U}/\partial t$ and $\mathbf{U} \cdot \nabla \mathbf{U}$ are null because of the stationarity

respectively the incompressibility of the flow. Thus assuming that there is no body force, the Navier-Stokes equation becomes

$$\frac{1}{\rho^{real}}\nabla p = \frac{\nabla p^*}{L} = \nu^{real}\nabla^2\mathbf{U}. \tag{3.59}$$

Replacing equation (3.56) into equation (3.59), one obtains

$$\partial_x p^* = -\frac{8\nu^{real}U_c L}{D^2} \quad \text{and} \quad \partial_y p^* = 0. \tag{3.60}$$

Maier et al. [54], Zou et al. [61] as well as Inamouro et al. [52] propose other ways to impose a pressure gradient. The interested reader is invited to consult these references.

3.4.3 Velocity profile

In many situations, it is convenient and common to assign a given velocity profile $\mathbf{u}^{in}$ as well as a density ρ^{in} at the inlet. Depending on the method, one can impose a zero gradient condition or simply nothing at the outlet if the domain is periodic.

A specified inlet velocity profile is easily implemented by constantly settling the inlet fields with the equilibrium population obtained from equation (3.13). Thus one has for all inlet sites the following condition

$$f_i^{out}(\mathbf{r}_{inlet}, t) = \frac{1}{\tau}f_i^{eq}(\rho^{in}, \mathbf{u}^{in}) + (1 - \frac{1}{\tau})f_i^{in}(\mathbf{r}_{inlet}, t), \quad \forall t$$

where ρ^{in} and $\mathbf{u}^{in}$ may depend on $\mathbf{r}_{inlet}$.

Now, we present three ways to treat the outlet. First, the simplest, the outlet is not treated. It means that nothing special is done on this region. Depending on the flow and on the geometry, results with good accuracy are obtained. Second, a zero gradient condition is imposed. Supposing that the outlet is the last column, a zero gradient condition is set by simply copying the fields of the before last column into the last one (i.e. the outlet). This second condition works well essentially if the outlet is located sufficiently far from the inlet. Otherwise, some numerical instabilities may appear. Third, the more complicated solution consists in reflecting back a portion of fluid particles which reach the outlet. This portion is adjusted so as to ensure mass conservation. The interested reader can find a more detailed explanation in [12].

3.4.4 Discussion around the Poiseuille flow

The flow settlement conditions presented in this section are now employed for the simulation of a Poiseuille flow defined on a LBGK D2Q9 $N_x \times N_z = 20 \times 11$ x-periodic lattice. The relaxation time $\tau = 1.0$ and the maximum (central) velocity

$U_c = 0.1$. We consider a fullway bounce-back condition which implies that the channel diameter $D = N_z - 1 = 10$.

Results are presented from figures 3.12 to 3.17. Let us comment these figures.

Figure 3.12 presents simulation results obtained using a body force in order to settle the flow. We observe that the density profile is constant over the system except obviously on the boundary sites ($z = 0$ and $z = N_z - 1$) where wall fields do not exist. The velocity profile is identical in every transversal cuts. Its expected maximum value is not reached because of the bounce-back boundary condition used which implies a slip velocity (see section 3.3 for details).

Contrarily to the body force condition, the other flow settlement conditions are applied on a specific region. It usually implies discontinuity in profiles around the inlet.

Figure 3.13 shows results of a pressure gradient employment. We observe that density is obviously not constant as the pressure is directly related to it. Density as well as velocity make a brutal jump close to the first column. Things are going better far from this part. We observe that the velocity varies longitudinally. This expected behavior ($\rho\mathbf{u}$ is supposed to be constant) is related to the longitudinal density variation imposed by the flow settlement condition. On the other hand, we note that the momentum ρu_x varies longitudinally. This variation is in fact not in contradiction with the continuity equation as the latter is recovered neglecting some error terms $O(\Delta\mathbf{r}^2)$ [9, 12]. Indeed, we observed a decrease of the longitudinal momentum gradient when decreasing the lattice spacing. Again, notice that the expected maximum velocity is not reached due to velocity gradient and because of boundary condition imprecisions.

Figures 3.14 and 3.15 presents the utilization of the most employed flow settlement boundary condition. It consists in imposing a constant (spatially and temporally) velocity profile on the first column and, for the results of figure 3.15 only, in applying a null longitudinal velocity gradient on the last column. We again observe a brutal variation in the profiles. Although the cuts seem to be the ones expected, we note the complicated variation of the profile shapes induced by forcing the velocity to be a constant velocity profile. The system experiences troubles to accommodate itself with this too simple constant velocity profile.

In figures 3.16 and 3.17, the imposed velocity profile is set to be the expected one, meaning a parabolic velocity profile. However, a null longitudinal velocity gradient at the outlet is considered for the results of figure 3.17. We again have a brutal variation in the profiles but varying essentially by a gradient. These results seem to be similar to those obtained with a pressure gradient. However the number of iterations needed to settle the flow is much lower with this condition. See figure 3.18 below for details.

The impact of the null longitudinal velocity gradient imposed at the end of the channel is not easy to observe. However, its use is crucial when the geometry presents more irregularities.

Considering a channel flow, a null horizontal velocity gradient is set on the

last channel column by proportionally copying the fields from the before last column. Indeed, a simple copy of the fields of sites located at $x = N_x - 2$ into those located at $x = N_x - 1$ would increase the density. Thus, to conserve mass, one uses the relation

$$f_i^{in}(\mathbf{r}_{N_x-1}) = f_i^{in}(\mathbf{r}_{N_x-2}) \cdot \frac{\rho_{N_x-1}}{\rho_{N_x-2}}, \quad \forall i \tag{3.61}$$

where $\mathbf{r}_{N_x-1}$ and $\mathbf{r}_{N_x-2}$ are sites located at $x = N_x-1$ and $x = N_x-2$ respectively, ρ_{N_x-1} and ρ_{N_x-2} are densities of sites $\mathbf{r}_{N_x-1}$ and $\mathbf{r}_{N_x-2}$ repsectively.

Also, it is interesting to note that the same simulation as the one presented in figure 3.16, but initialized with its stationary state, gives identical results.

The number of iterations needed to obtain a stationary Poiseuille flow is measured by observing the maximum velocity of the middle transversal cut ($N_x = 10$). This number of iterations is obtained when the difference between two successive maximum velocities is smaller than 10^{-10}. Results are reported in figure 3.18 for various Reynolds number obtained by modifying the relaxation time. We observe that the body force and the pressure gradient conditions take more time to establish the desired flow while imposing velocity profiles is much faster. This difference increases with the Reynolds number.

In conclusion, we will use the body force when the system and the Reynolds number are small. We will otherwise impose an adapted velocity profile at the inlet. A null longitudinal velocity gradient will be set if necessary.

A more theoretical and formal study is necessary to better understand the impact of flow settlement conditions. This is one of the numerous tasks listed in section 7.3.

3.5 Turbulence models

An important class of applications in hydrodynamics consists of high Reynolds number flows. A simple way to reach high Reynolds numbers is to reduce the viscosity by making the relaxation time τ close to 0.5. Unfortunately, numerical instabilities may develop in this case, due essentially to velocity gradients. To alleviate this problem, one can have recourse to turbulence models. Note that an adapted fluid initialization may, in some cases, cancel the numerical instabilities [12].

In the present document, essentially because of its simplicity, we will focus our attention on the Smagorinsky turbulence model and give the main ideas of the $k - \epsilon$ model family. These models have been presented in a CFD context in section 2.2.4 respectively section 2.2.5. They are now adapted to LBGK numerical schemes.

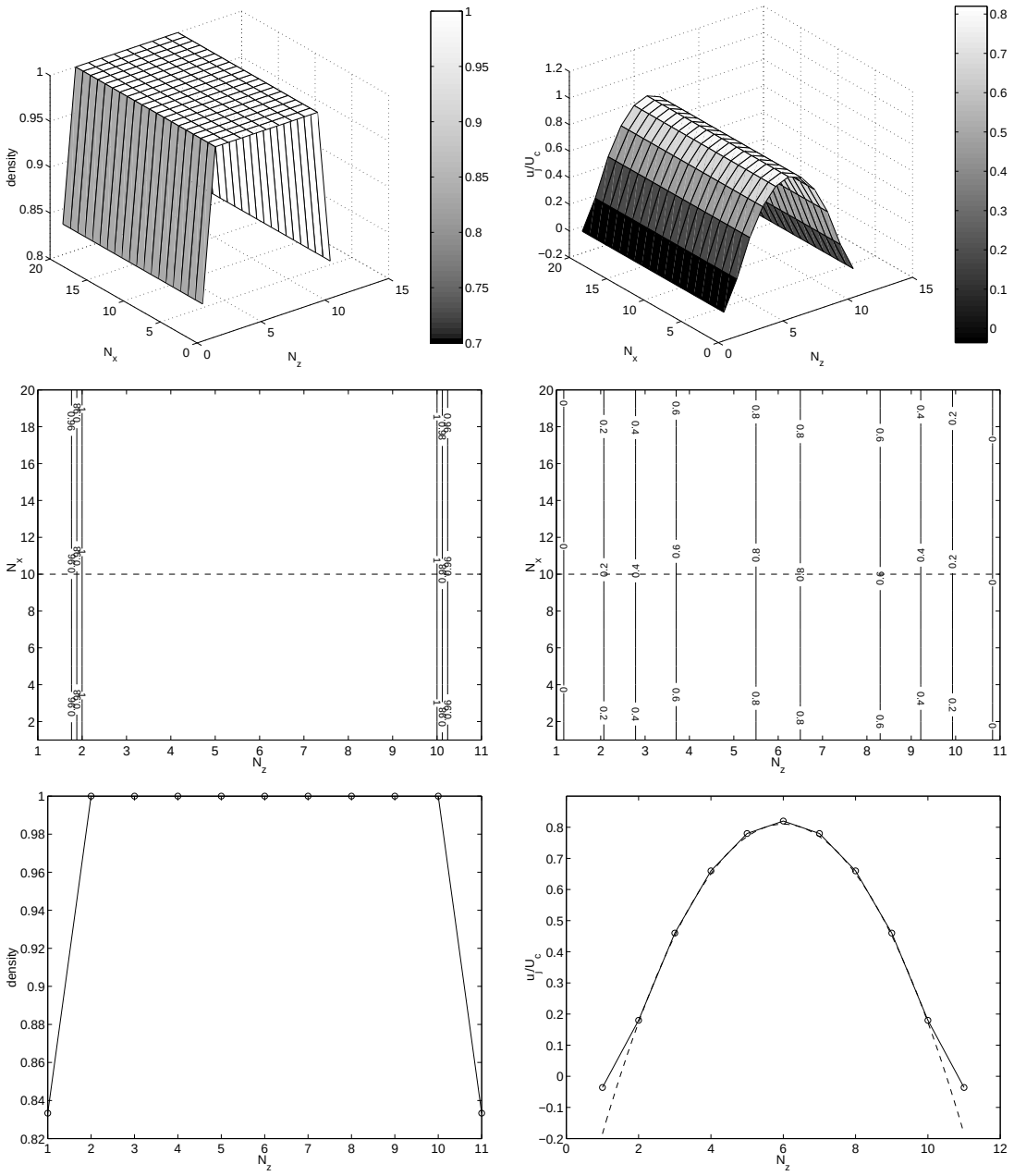


Figure 3.12: Velocity and density profiles of a Poiseuille flow on a D2Q9 $N_x \times N_z = 20 \times 11$ lattice. A body force flow settlement condition is employed. The relaxation time $\tau = 1.0$ and the maximum velocity $U_c = 0.1$. A fullway bounce-back boundary condition is used. Density and velocity results are presented on the left, respectively right column. The first row presents 3D views while its contour lines are drawn on the second row. The last row shows vertical cuts of the profiles at $N_x = 10$ indicated by a dashed line on the graphs of the second raw. A fit of the bulk points is drawn by a dashed parabola on the velocity profile cut.

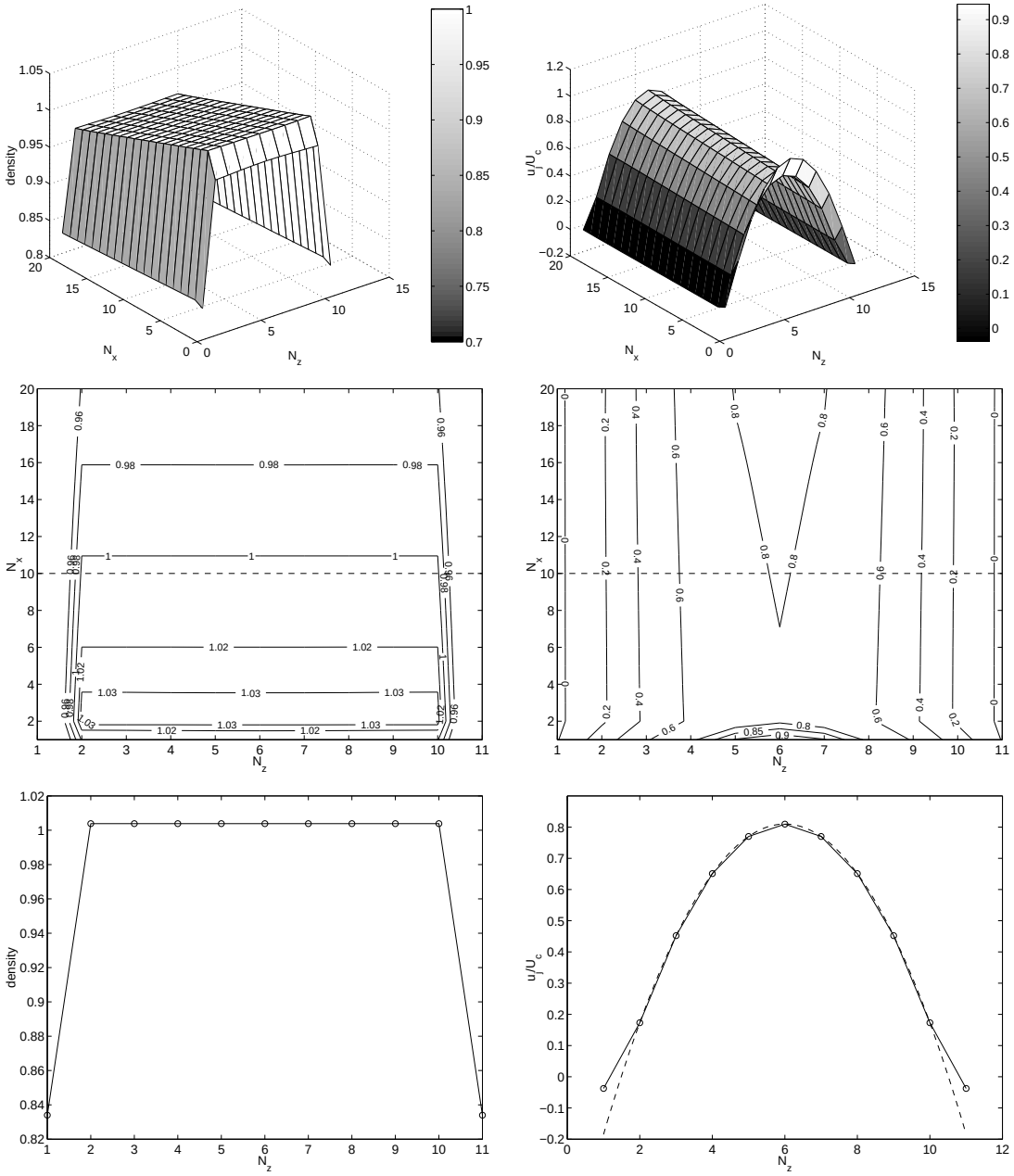


Figure 3.13: A pressure gradient is imposed so as to settle the flow of the same experiment as the one presented in figure 3.12.

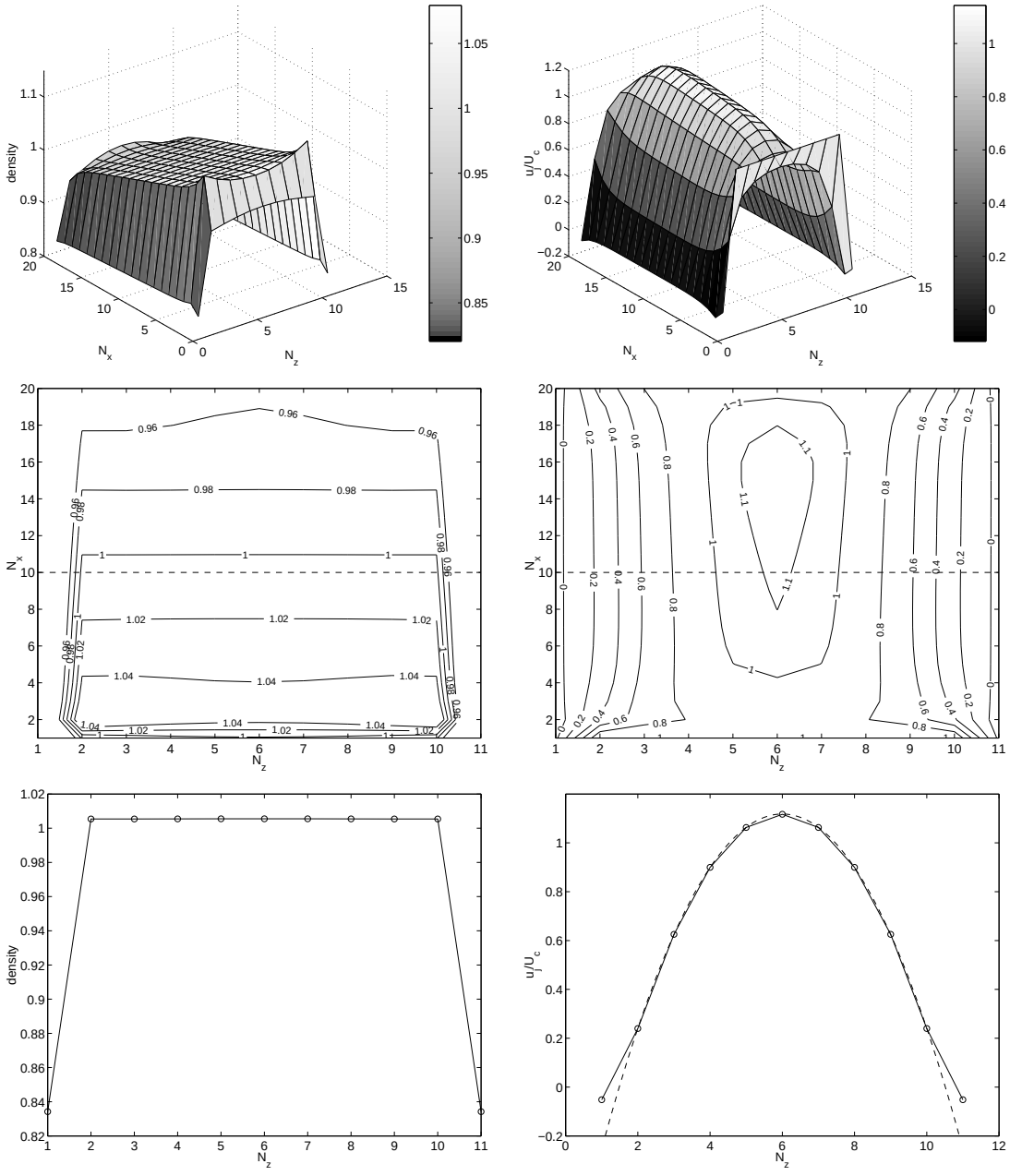


Figure 3.14: A uniform velocity profile equal to U_c is imposed on the first channel column so as to settle the flow of the same experiment as the one presented in figure 3.12.

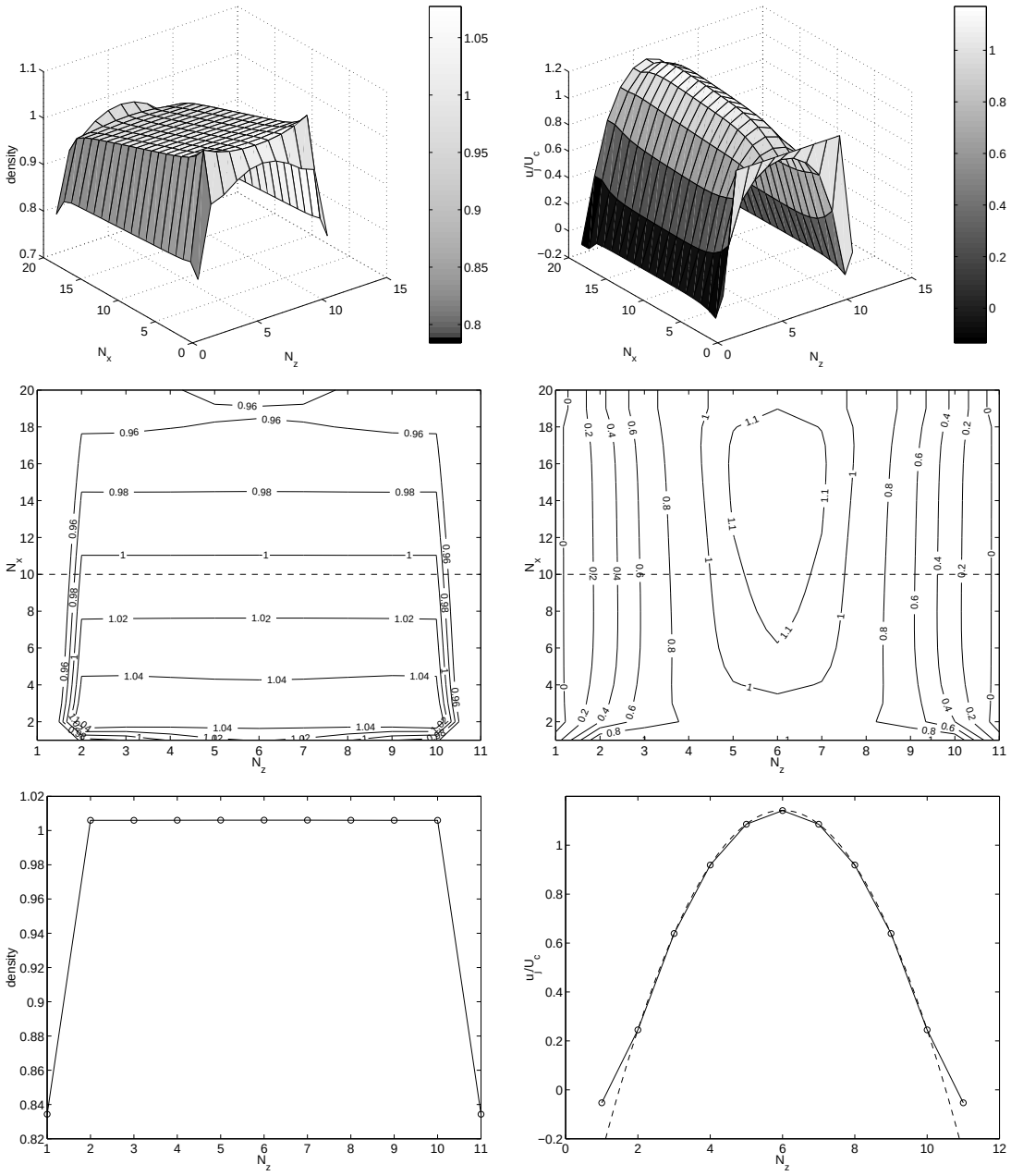


Figure 3.15: A uniform velocity profile equal to U_c is imposed on the first channel column so as to settle the flow of the same experiment as the one presented in figure 3.12. A null horizontal velocity gradient is set on the last channel column by proportionally copying the fields from the before last column, see equation (3.61).

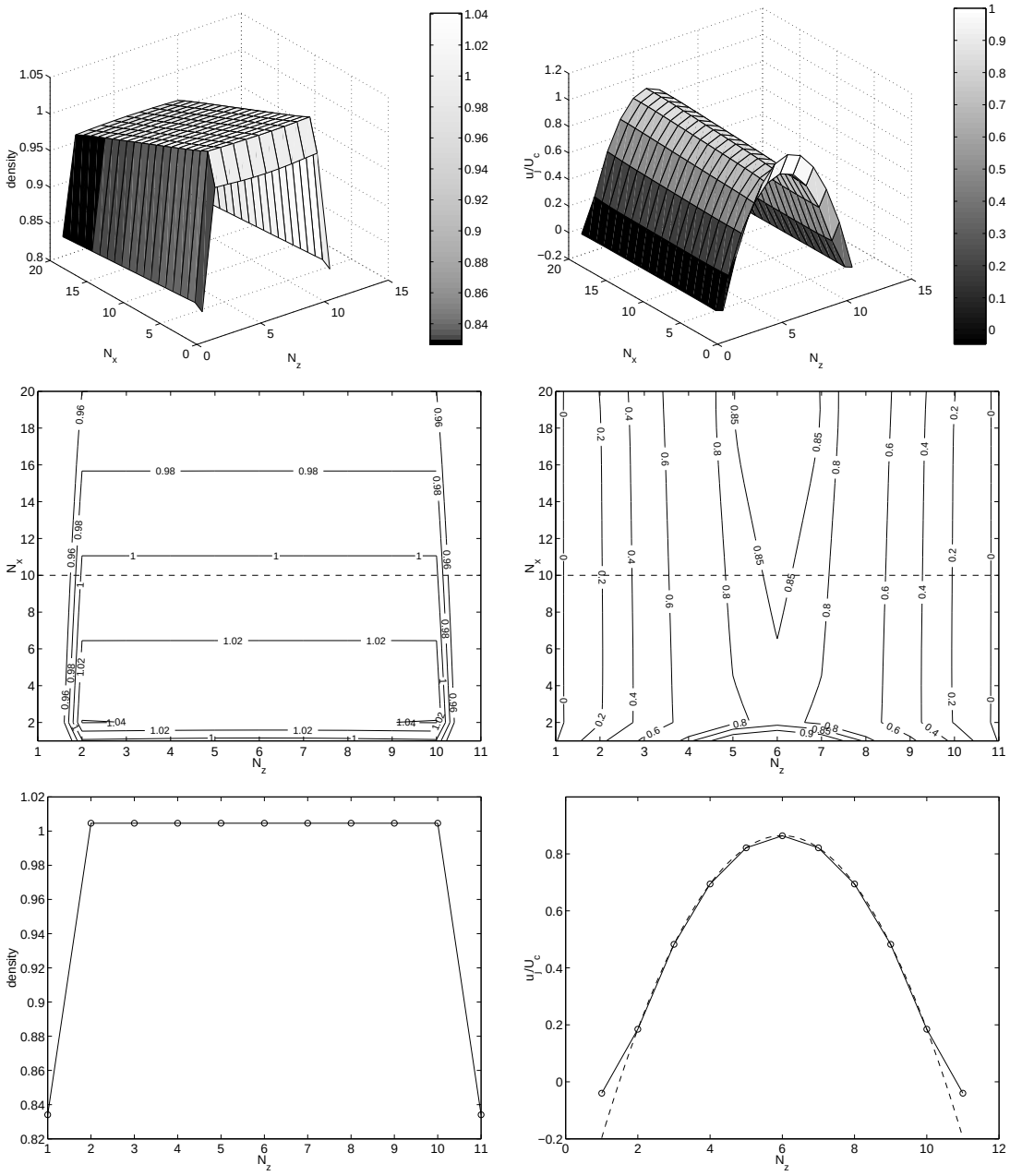


Figure 3.16: A parabolic velocity profile with a maximum equal to U_c is imposed on the first channel column so as to settle the flow of the same experiment as the one presented in figure 3.12.

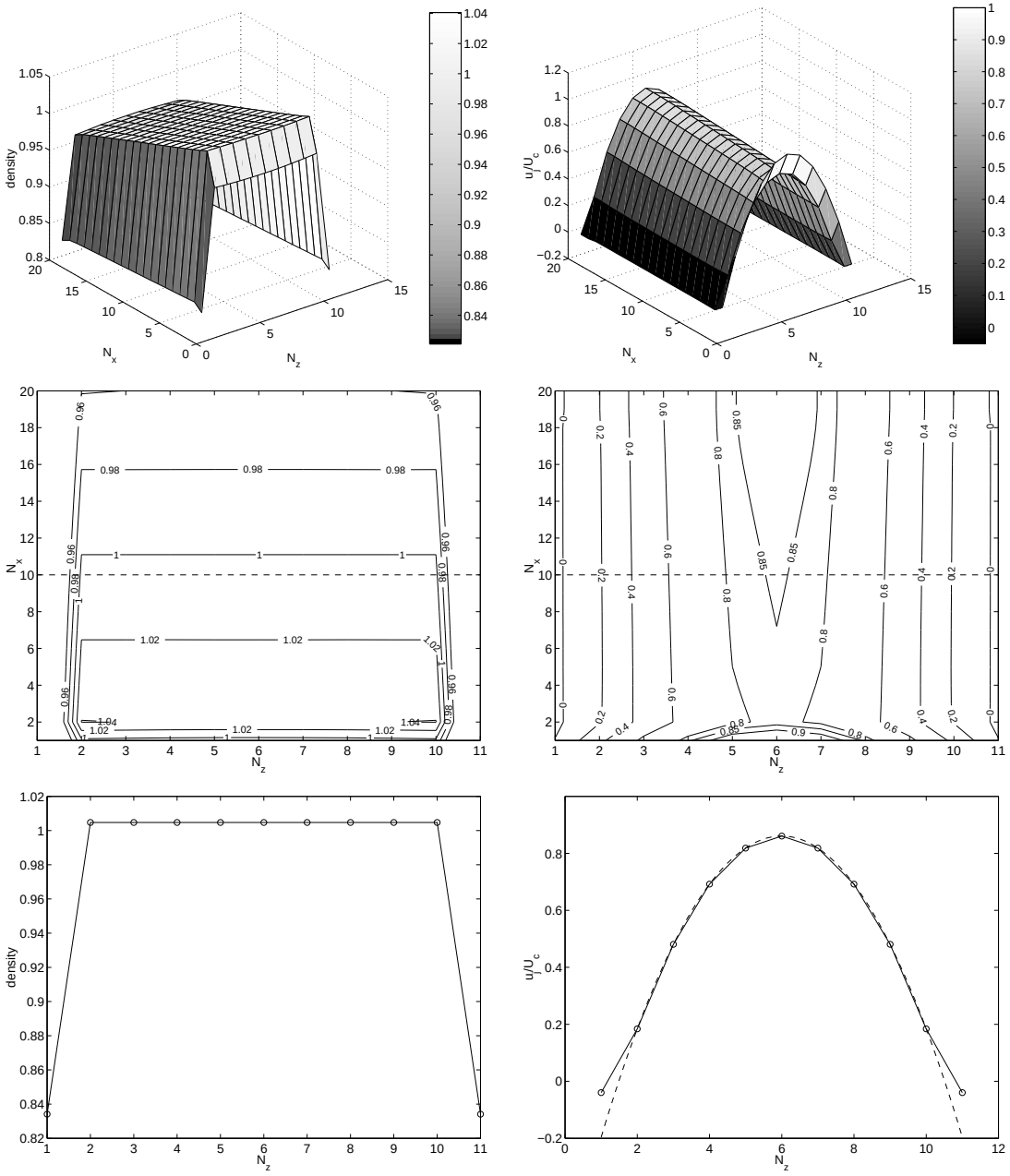


Figure 3.17: A parabolic velocity profile with a maximum equal to U_c is imposed on the first channel column so as to settle the flow of the same experiment as the one presented in figure 3.12. A null horizontal velocity gradient is set on the last channel column by proportionally copying the fields from the before last column. A simple copy of the fields would increase the density.

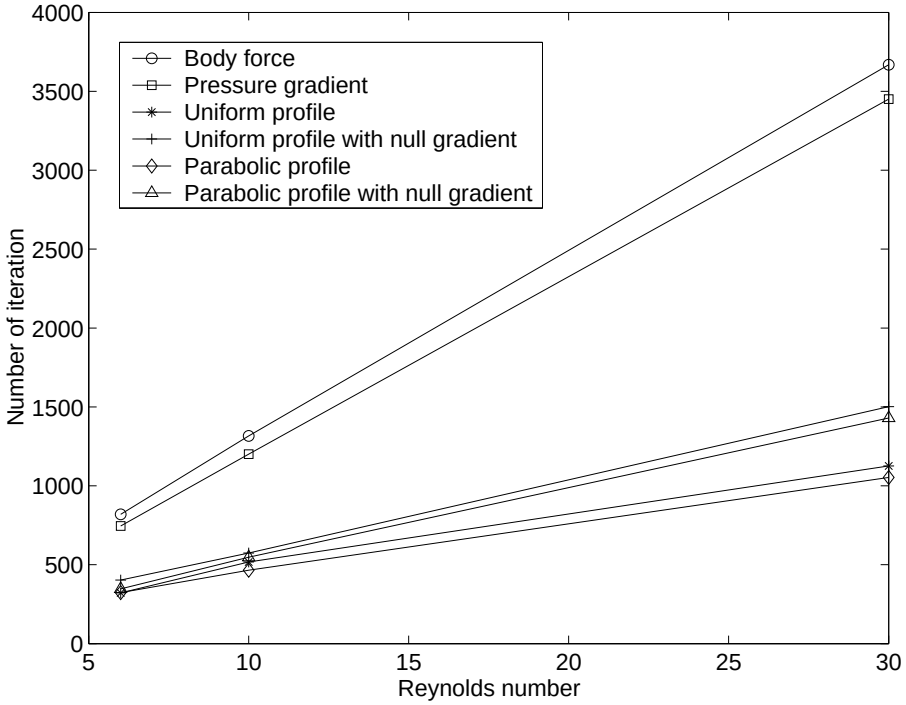


Figure 3.18: Necessary number of iterations to obtain the stationary Poiseuille flow described in figure 3.12 versus the Reynolds number. Various flow settlement conditions have been considered.

3.5.1 The Smagorinsky subgrid model

One of the most famous large eddy simulation (LES) model is the one due to Smagorinsky [25]. This approach was first proposed for LBGK models by Hou et al. [62]. One assumes that a turbulent viscosity (ν_t) results from the unresolved scales, that is the scales below the lattice spacing $\Delta\mathbf{r}$. These scales are thus filtered. The main idea is to increase locally the relaxation time τ by defining a new space and time dependent relaxation time $\tau_{tot} = \tau + \tau_t$ where τ_t is the turbulent contribution. We assume a fixed value for τ.

Then the total viscosity can be decomposed as

$$\nu_{tot} = \nu + \nu_t$$

where ν is the viscosity corresponding to τ and is given by equation (3.12). The new contribution ν_t is the so-called turbulent viscosity (see section 2.2.2 for details) resulting from the filtered scales. In the Smagorinsky model [25], it is expressed as

$$\nu_t = (C_{smago}\Delta)^2|S| \tag{3.62}$$

where Δ is the filter size, whose magnitude usually corresponds to $\Delta\mathbf{r}$ and $|S| = \sqrt{2S_{\alpha\beta}S_{\alpha\beta}}$ is the magnitude of the strain-rate tensor $S_{\alpha\beta} = 1/2(\partial_\beta u_\alpha + \partial_\alpha u_\beta)$.

Thus the turbulent viscosity increases with $|S|$, so that the total viscosity is larger in regions close to obstacles. It is illustrated in figure 3.19.

In the LB scheme, the quantity $S_{\alpha\beta}$ can be computed locally, without taking extra derivatives, by only considering the non-equilibrium momentum tensor as [37]

$$S_{\alpha\beta} = -\frac{1}{2\rho\tau_{tot}\Delta t}\frac{C_2}{C_4}\Pi^{(1)}_{\alpha\beta} = -\frac{1}{2\rho\tau_{tot}\Delta t}\frac{C_2}{C_4}\sum_i v_{i\alpha}v_{i\beta}(f_i - f_i^{(0)}). \tag{3.63}$$

Thus, from equations (3.62) and (3.63), the turbulent viscosity can be expressed as

$$\nu_t = \frac{1}{\sqrt{2}\rho\tau_{tot}\Delta t}\frac{C_2}{C_4}(C_{smago}\Delta)^2\sqrt{Q} \tag{3.64}$$

where $Q = \Pi^{(1)}_{\alpha\beta}\Pi^{(1)}_{\alpha\beta}$. From equation 3.12, one has

$$\tau_{tot} = \frac{\Delta t}{\Delta \mathbf{r}^2}\frac{C_2}{C_4}(\nu + \nu_t) + \frac{1}{2}. \tag{3.65}$$

Considering also that

$$\tau = \frac{\Delta t}{\Delta \mathbf{r}^2}\frac{C_2}{C_4}\nu + \frac{1}{2}$$

and replacing equation (3.64) into equation (3.65), one can solve the resulting second order equation and then express the local relaxation time as

$$\tau_{tot} = \frac{1}{2}\left(\sqrt{\tau^2 + \frac{1}{\Delta \mathbf{r}^2}(C_{smago}\Delta)^2\left(\frac{C_2}{C_4}\right)^2\frac{\sqrt{8Q}}{\rho}} + \tau\right). \tag{3.66}$$

The filter size Δ is usually set to one lattice spacing. The quantity C_{smago} is typically smaller than 0.5. It tunes the effect of the subgrid model and should be adjusted empirically depending on the desired flow pattern. Note that the quantity C_{smago} is possibly space dependent although we do not consider such a dependence in this thesis.

To illustrate how much and where total viscosities are modified, we consider a LBGK D3Q19 flow around a cylinder of diameter $D = 20$ on a toric $N_x \times N_y \times N_z = 140 \times 5 \times 60$ lattice. Bounce-back conditions have been considered around the cylinder. The constant $C_{smago} = 0.32$ and the filter size $\Delta = \Delta\mathbf{r}$. The flow is settled by imposing a uniform velocity profile $U_x = 0.1$ on the plane $x = 0$. The fluid flows from left to right with Reynolds number $Re = 10^5$. Using equations (2.7) and (3.12), one finds that

$$\tau = \frac{3DU_x}{Re} + \frac{1}{2} = 0.50006 \quad \text{and} \quad \nu = \frac{1}{3}(\tau - \frac{1}{2}) = 2 \cdot 10^{-5}.$$

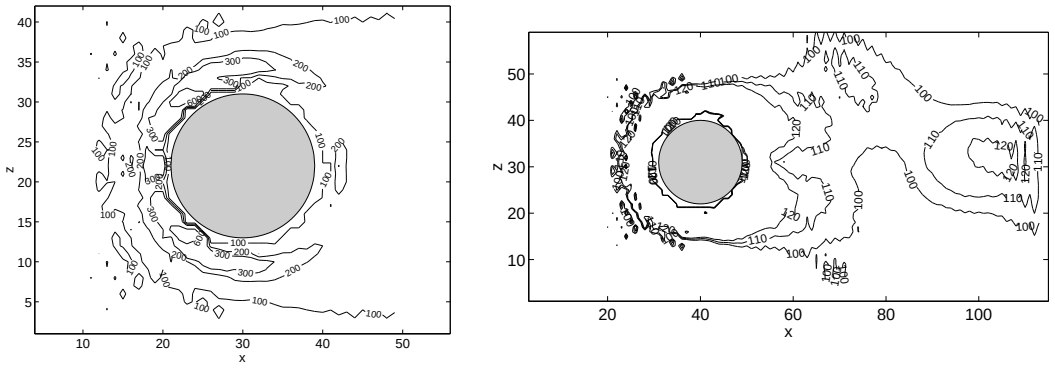


Figure 3.19: Ratio ν_t/ν of a LBGK D3Q19 flow around a cylinder of diameter $D = 20$ on a toric $N_x \times N_y \times N_z = 140 \times 5 \times 60$ lattice. Left and right panels show two different views at different scales corresponding of the same flow. Bounce-back conditions have been considered around the cylinder. The constant $C_{smago} = 0.32$ and the filter size $\Delta = \Delta\mathbf{r}$. The flow is settled by imposing a uniform velocity profile $U_x = 0.1$ on the plane $x = 0$. The fluid flows from left to right with Reynolds number $Re = 10^5$.

Figure 3.19 presents the ratio ν_t/ν where turbulent viscosities ν_t have been averaged over 40000 iterations starting after 10000 iterations. The measurements have been performed on the plane $y = 1$. We observe that the bigger viscosity increase is located upstream around the cylinder. The increase is also visible downstream the cylinder, making a kind of streamlines. We also note the asymmetrical pattern presented on the right of figure 3.19. This is due to the short averaging period of time considered.

Recall that a flow energy spectrum is defined as the Fourier transform of the square of the flow velocity (see section 2.1.3 for details). Such a spectrum behaves as power of law of exponent $-5/3$ in the inertial range. Figure 3.20 presents energy spectrums of LBGK D3Q19 shear flows on a $20 \times 20 \times 20$ box where C_{smago} is set to 0.224, 0.1 and 0.0894. Considering $\Delta = \Delta\mathbf{r}$, the term $(C_{smago}\Delta)^2$ is then equal to 0.05, 0.01 and 0.008 respectively. The flow is settled by imposing a constant velocity $U_x = 0.1$ on the plane $z = 19$ and $U_x = -0.1$ on the plane $z = 9$. Periodic boundary conditions are considered. Velocity and energy are measured at the point $(x, y, z) = (5, 5, 5)$ starting after 5000 iterations and ending at 25000 iterations. The Reynolds number is set to 10^5 implying $\tau = 3 \cdot 0.1 \cdot 20/10^5 + 0.5 = 0.50006$.

We observe that the exponent $-5/3$ is better recovered in the inertial range for a value of C_{smago} around 0.1. Moreover, we note that a too large constant leads to a poor behavior which does not show all scales. A too small constant seems to yield a wrong spectrum from the point of view of its exponent and the length of its inertial range. This experiment shows that, for a given experiment,

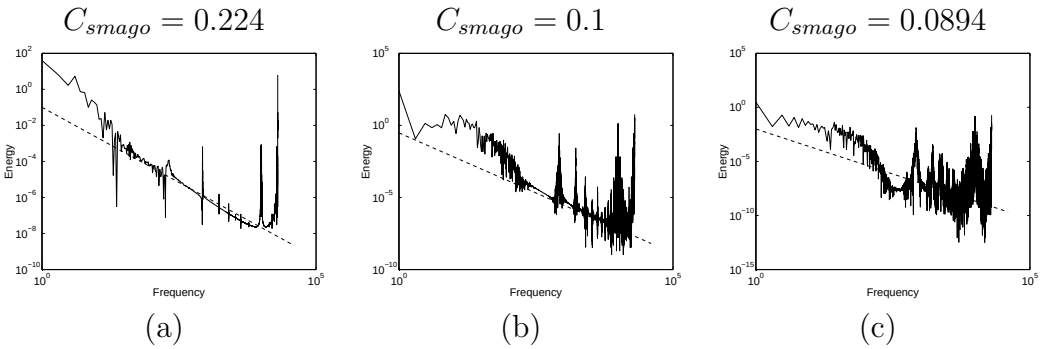


Figure 3.20: Energy spectrums of LBGK D3Q19 shear flows on a $20 \times 20 \times 20$ box where C_{smago} is set to (a) 0.224, (b) 0.1 and (c) 0.0894 and $\Delta = \Delta\mathbf{r}$. The flow is settled by imposing a constant velocity $U_x = 0.1$ on the plane $z = 19$ and $U_x = -0.1$ on the plane $z = 9$. Periodic boundary conditions are considered. The energy is measured at the point $x = 5$, $y = 5$ and $z = 5$ starting after 5000 iterations and ending at 25000 iterations. The Reynolds number is set to 10^5. We observe that the exponent $-5/3$ is better recovered in the inertial range for a $C_{smago} = 0.1$, i.e. the subfigure (b).

there is an adapted value of C_{smago}. In this case, we would choose a value between $C_{smago} = 0.1$ and $C_{smago} = 0.2$.

The problem of adjusting correctly C_{smago} remains open. In order to model the boundary layer near a wall, one may expect that the value of C_{smago} is zero at the boundary of an obstacle and then increases to reach its bulk value as one moves away from the wall. No obvious theory describes how this variation should be. Moreover, we performed some simulations considering a linear as well as an exponential decrease close to boundaries. They did not show a real change on the main features of the flow. Therefore, we assume that the simplified procedure of having a non-zero constant is sufficient here.

Setting C_{smago} large enough, say 0.5, allows us to deal with stable numerical scheme even with relaxation times very close to 0.5. It implies that a null viscosity and consequently an infinite Reynolds number would be taken into account. From a physical point of view, it does not make any sense. In their paper [62], Hou et al. claimed they simulated a cavity flow at a Reynolds number equal to 10^6. Their numerical scheme was probably stable but it is not clear whether they simulated a *real* fluid. However, the question aiming at setting a viscosity limit which can be simulated by LBGK schemes is a hard and open problem. Let us present an idea giving a clue on the way to find a value of this limit.

We saw in section 2.1.3 that a turbulent flow develops eddies. The Kolmogorov theory indicates that a highly turbulent flow produces smaller eddies than a lesser one. These eddies have obviously an influence on the velocity $\mathbf{u}(\mathbf{r}, t)$. They can be highlighted by analyzing the Fourier transform of the velocity on a given

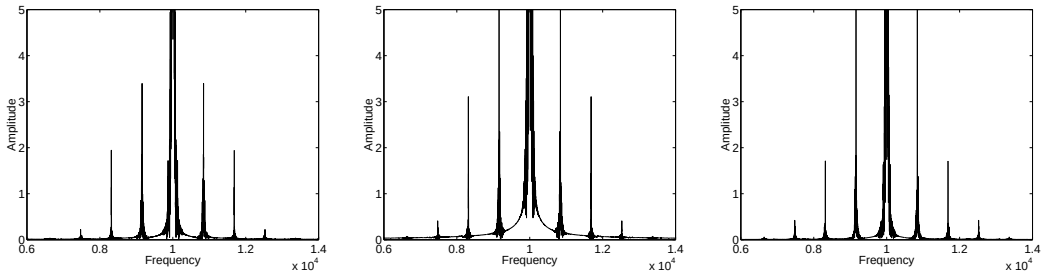


Figure 3.21: Fourier transforms of shear flow velocities measured on the point $5 \times 5 \times 5$ at various Reynolds number: (a) 10^4, (b) 10^5 and (c) 10^6. The geometry is the one presented in figure 3.20 but with a $C_{smago} = 0.1$ for the three cases.

position. Indeed, the more turbulent the flow, the bigger the number of activated frequencies of the Fourier transform. We consider the same shear flow as the one presented in figure 3.20 but here with a unique $C_{smago} = 0.1$. Figure 3.21 presents Fourier transforms of velocities measured on the point $(5, 5, 5)$ at various Reynolds number: 10^4, 10^5 and 10^6. We observe that transform (b) presents more activity than transform (a). This is the expected result as more eddies develop in case (b). On the other hand, transform (c) shows less activity than transform (b). It indicates that the flow (c) do not represent a *real* flow essentially because the imposed viscosity is too small. This simple example hints at the existence of a viscosity limit which probably depends on the flow. Hence, imposing small viscosities require a special attention even though the numerical scheme is stable.

Therefore, one has to compute locally the relaxation time τ_{tot} before applying the collision operator. It is easily implemented and presented in code 3.10.

To conclude, there are still some open questions which have been highlighted in this section. More work is necessary to be able to define a viscosity limit. Also, the treatment of the constant C_{smago} close to the wall is not yet clearly understood and has to be clarified. However, this numerical model gives a simple way to deal with high Reynolds number flows even if there are some unclear points. Some examples of use are presented in section 3.7.

3.5.2 The $k - \epsilon$ model family

The $k - \epsilon$ models, not used in the present work, are however the most used turbulence models in industrial applications. They have been widely analyzed and many variants exist in a CFD context [1]. Recall from section 2.2.5, that k is the kinetic turbulent energy created by the eddies and ϵ the dissipation energy occurring when eddies disappear.

Let us just quote the main contributions in the LB context. Succi et al. [63]

Code 3.10 Function computing the local relaxation time τ_{tot} for a D2Q9 lattice.

```
double computeTauTot(const double feq[9], const double density)
{
  int k,alpha,beta;
  double fneq[9],q,pi;
   // Computes the non-equilibrium distribution part
  for (k=1;k<nbNeighbor;k++) fneq[k]=fin[k]-feq[k];
   // Computes the tensor Q = Π^(1)_αβ Π^(1)_αβ
  q=0.0;
  for (alpha=0;alpha<2;alpha++)
    for (beta=0;beta<2;beta++) {
      pi=0.0
      for (k=1;k<nbNeighbor;k++)
        pi+=v[k][alpha]*v[k][beta]*fneq[k];
      q+=pi*pi;
    }
   // Returns the result where CD2= (C_smago Δ)^2
  return 0.5*(sqrt(tau*tau+CD2*c2*c2/c4/c4*sqrt(8.0*q)*/density)+tau);
}
```

propose to consider extra energy fields used to compute k and ϵ. Other approaches using LBGK in conjunction with finite difference schemes for the solution of $k-\epsilon$ equations have been proposed by Texeira [64]. He successfully simulated flows over a backward facing step at a Reynolds number equal to 5000 [64].

3.5.3 Discussion

The Smagorinsky subgrid model is definitely simpler than any of the $k-\epsilon$ models. The latter need more computational time than its competitor. Although they probably produce results of better quality, especially in the treatment of the wall, we do not need such heavy models for the application we shall present in chapter 5 (virtual rivers). In this work, we restrict ourselves to the simple Smagorinsky subgrid model.

3.6 From LBGK mesoscopic scale to reality

This section is devoted to a description of the rules giving the way to transform LBGK quantities (mesoscopic scale) into real values (macroscopic scale). Recall that the macroscopic scale is the scale at which the global phenomenon is observed by human eyes and the mesoscopic scale refers to objects larger than an atom,

Denomination	LBGK context	Real context
Fluid field	f_i	$f_i^{real} = \Delta m\ f_i$
Propagation velocity	$\mathbf{v}_i$	$\mathbf{v}_i^{real} = \Delta\mathbf{r}/\Delta t\ \mathbf{v}_i$
Speed of sound	c_s	$c_s^{real} = \Delta\mathbf{r}/\Delta t\ c_s$
Density	$\rho = \sum_i f_i$	$\rho^{real} = \Delta m\ \rho$
Velocity	$\mathbf{u} = 1/\rho \sum_i f_i \mathbf{v}_i$	$\mathbf{u}^{real} = \Delta\mathbf{r}/\Delta t\ \mathbf{u}$
Relaxation time	τ	$\tau^{real} = \Delta t\ \tau$
Viscosity	$\nu = C_4/C_2(\tau - 0.5)$	$\nu^{real} = \Delta\mathbf{r}^2/\Delta t\ \nu$
Pressure	$p = c_s^2\rho$	$p^{real} = \Delta\mathbf{r}^2/\Delta t^2 \Delta m\ p$

Table 3.3: Conversion rules between LBGK quantities and their corresponding physical values.

but smaller than anything that can be manipulated with human hands. See section 1.2 for a more detailed discussion on scales.

The three scale factors $\Delta\mathbf{r}$, Δt and Δm are used to scale LBGK quantities into real quantities. The scale factor $\Delta\mathbf{r}$ is the length of a lattice spacing expressed in *meter*, Δt is the time elapsed during one iteration expressed in *second* and Δm is the mass of an elementary particle expressed in *kilogram*. Supposing that ρ_{fluid} is the real density of the fluid and that initially the LBGK density $\rho = 1$ on each site, one can compute $\Delta m = \rho_{fluid}\ \Delta\mathbf{r}^3$. Table 3.3 summarizes the conversion rules between LBGK quantities and their corresponding physical values.

3.7 Some LBGK simulations

To illustrate the concepts presented in this chapter, we describe two simulations in this section. First, we show how an LBGK model can simulate the flow over a backward facing step at moderate Reynolds number. Second, we exhibit an experiment scanning a large range of Reynolds numbers. This corresponds to the determination of the drag force around a cylinder.

3.7.1 Flow over a backward facing step

The flow over a backward facing step is illustrated in figure 3.22. It is interesting because it presents some complex flow patterns already at moderate Reynolds number ($R_e = 5000$). They principally consist of a vortex starting close to the step, moving downstream, disappearing and so on. Thus, the simplicity of the geometry and the richness of the flow behavior has interested more than one scientists.

Jovic et al. [65] conducted experimental measurements for different Reynolds number flows over a backward-facing step. They reported the mean velocities and

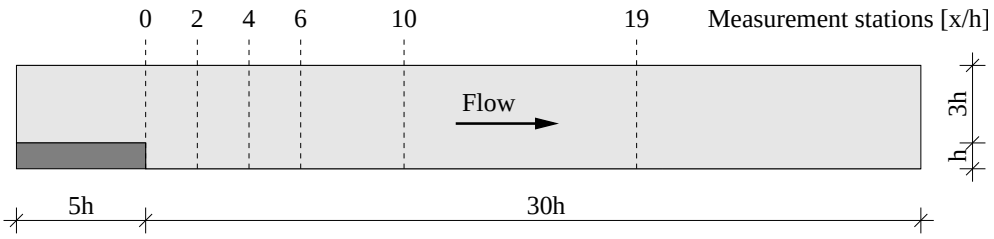


Figure 3.22: Geometry of the backward facing step.

their variations. These measurements have been numerically simulated in [24, 66] with good accuracy, respectively using a direct numerical method and solving a set of transport equations. Texeira [64] used a LBGK model with a $k-\epsilon$ turbulence model to produce acceptable results. Note also that Nakkasyan [67], using a LBGK with a Smagorinsky subgrid model, published promising results on the present experiment.

The flow is simulated on a LBGK D2Q9 lattice with the Smagorinsky subgrid model considering $C_{smago} = 0.1$. The Reynolds number $Re = 5000$. The lattice has $N_x \times N_y = (5h + 30h) \times 4h$ with $h = 20$. The flow is settled by applying a uniform velocity profile $U_{entry} = 0.1$. The relaxation time τ can then be computed as

$$\tau = \frac{C_2}{C_4}\frac{hU_{entry}}{Re} + \frac{1}{2} = 3 \cdot \frac{20 \cdot 0.1}{5000} + \frac{1}{2} = 0.5012$$

A null velocity gradient is imposed at the outlet. Fullway bounce-back boundary conditions are considered and the free surface is simulated by setting a bounce-forward boundary condition on the last layer of the lattice.

Due to the non-stationarity of the flow, we measure the mean velocity profiles of the flow, i.e. $\langle u_x \rangle$ and $\langle u_y \rangle$, their variations, i.e. $\langle u'_x u'_x \rangle$, $\langle u'_y u'_y \rangle$, and the variation correlations $\langle u'_x u'_y \rangle$. Measurements are executed between iterations 50 000 and 250 000. Notice that these variations are the components of the Reynolds stress tensor.

Figure 3.23 presents the streamlines obtained with the mean flow. We measure a reattachment length of the larger vortex equal to $5.7h$. It is close to $6h$ which was measured in laboratory by Jovic et al. [65].

The velocity profiles of our simulation are presented in figure 3.24 and compared with laboratory velocity profiles [65]. The mean flow is in good agreement with the laboratory measurement. However, the velocity variations and their correlations still need to be improved. A first lead could be to enhance boundary conditions. A second one could be to append an adapted wall treatment in the turbulence model.

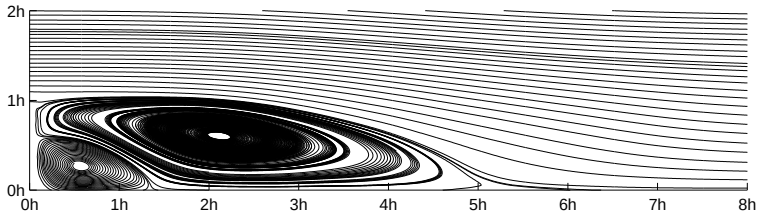


Figure 3.23: Mean flow streamlines simulated on a LBGK D2Q9 lattice with a Smagorinsky subgrid model considering $C_{smago} = 0.1$. The Reynolds number $Re = 5000$.

3.7.2 Drag force on a cylinder

An important quantity in hydrodynamics is the force exerted on an obstacle by a fluid flowing around it. This force is called the drag force and is strongly related to the obstacle shape. For a same cross section, the drag force is bigger on a cylinder than on a wing. Forces around obstacles of various shapes have been analyzed and presented in [14]. In this section, we will only consider cylinders.

In order to normalize and then to compare experiments, one introduces the drag force coefficient which is dimensionless. It is expressed as

$$\mathbf{C}_D = \frac{\mathbf{F}_D}{\frac{1}{2}\rho \mathbf{U}^2 D}$$

where $\mathbf{F}_D$ is the drag force per unit length, ρ the fluid density, $\mathbf{U}$ the fluid velocity and D the cylinder diameter.

The drag force coefficient measured experimentally [13] on a cylinder has been presented in figure 2.4. At low Reynolds number ($< 10^{-2}$), we have $C_D \propto Re^{-1}$. Then, over a wide range of Reynolds number (10^{-2} to $3 \cdot 10^5$) the drag force coefficient varies little. There is a dramatic departure from this behavior at $Re \approx 3 \cdot 10^5$. The drag force coefficient drops by a factor of over 3.

We consider a LBGK D3Q19 flow around a cylinder of diameter $D = 20$. The lattice has $N_x \times N_y \times N_z = 7D \times 3D \times 5$ and is periodic in every directions. The flow is settled by imposing a constant velocity profile ($u_x(\mathbf{r}_{inlet}, t) = U_{entry} = 0.1, \quad \forall t$) and the fullway bounce-back boundary condition is considered around the cylinder. The Smagorinsky subgrid model with $C_{smago} = 0.1$ is activated only when $\tau < 0.55$.

One knows that the fluid acceleration is the variation in time of the velocity and that the drag force is equal to the acceleration multiplied by the mass. Each field colliding with the cylinder in the direction i represents a momentum equal to $\mu_i = f_i(\mathbf{r}_b)\mathbf{v}_i$. Considering the bounce-back boundary condition, a momentum equal to $-\mu_i$ is created. It consequently yields a variation of $2\mu_i$. Thus, one can express the drag force as

$$\mathbf{F}_D = \sum_{\mathbf{r}_b} \sum_i 2 f_i(\mathbf{r}_b)\mathbf{v}_i$$

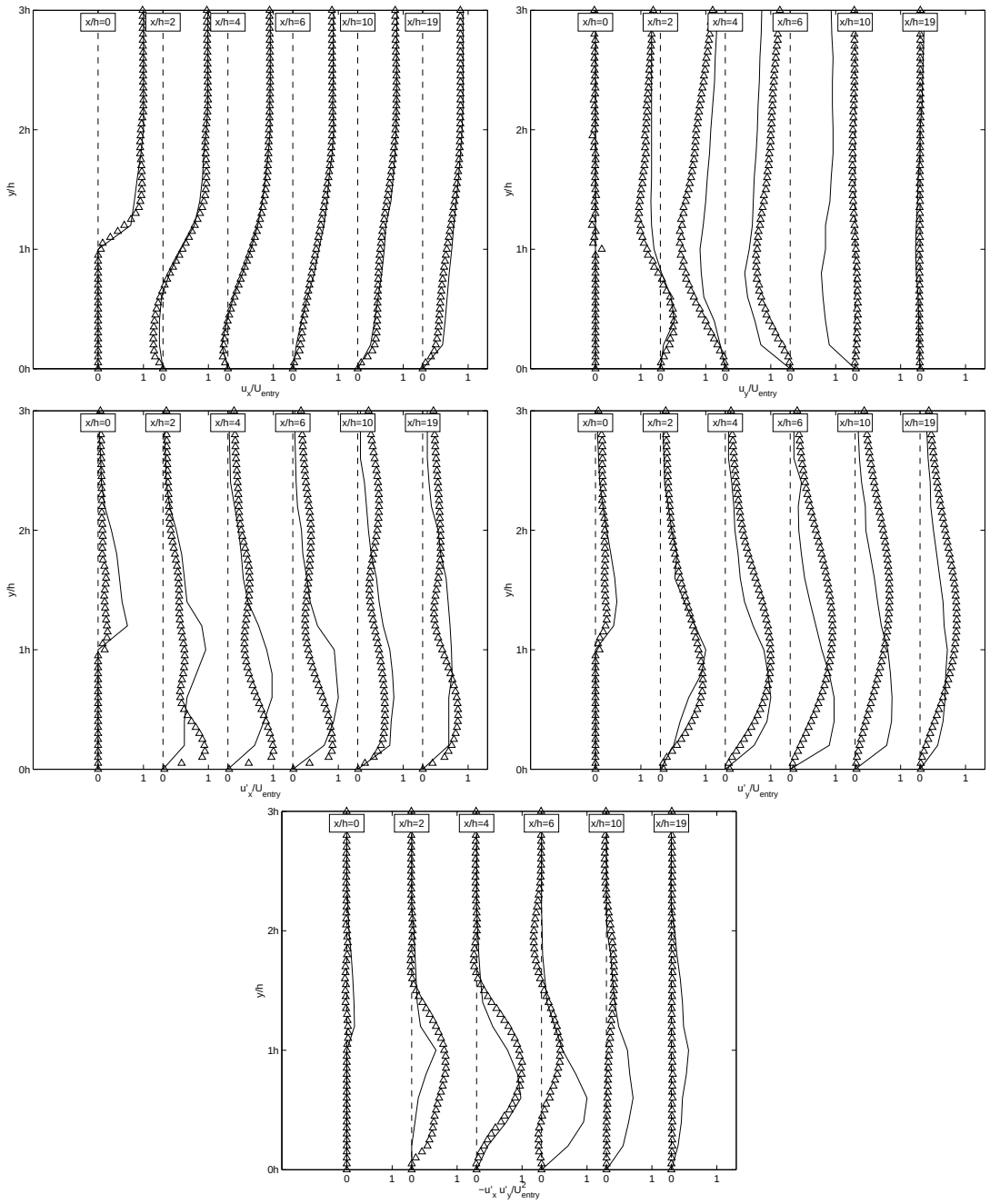


Figure 3.24: Velocity profiles of the flow simulated on a LBGK D2Q9 lattice with a Smagorinsky subgrid model considering $C_{smago} = 0.1$. The Reynolds number $Re = 5000$. Mean velocities $\langle u_x \rangle$ (denoted as u_x) and $\langle u_y \rangle$ (denoted as u_y) are reported as well as their variations $\langle u'_x u'_x \rangle^{0.5}$ (denoted as u'_x), $\langle u'_y u'_y \rangle^{0.5}$ (denoted as u'_y) and $\langle u'_x u'_y \rangle$ (denoted as $u'_x u'_y$). Solid lines are the results from Jovic et al. [65] while triangles represent ours.

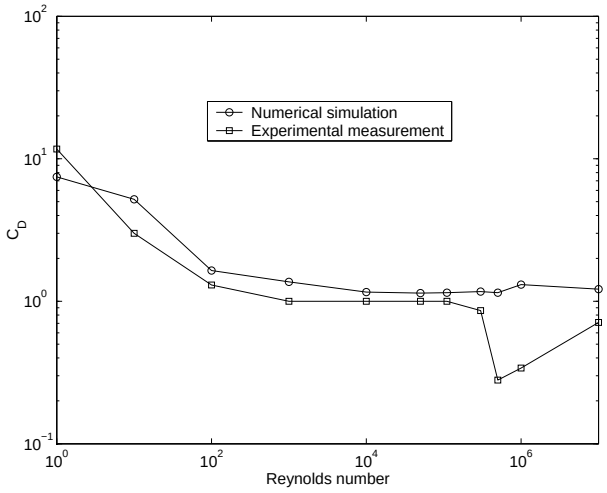


Figure 3.25: Component x of the drag force coefficient exerted on a cylinder of diameter $D = 20$ due to a LBGK D3Q19 flow on a $N_x \times N_y \times N_z = 7D \times 3D \times 5$ lattice. It is periodic in every direction. The flow is settled by imposing a constant velocity profile and the bounce-back boundary condition is considered around the cylinder. The Smagorinsky subgrid model with $C_{smago} = 0.1$ is activated only when $\tau < 0.55$. The experimental measurement is taken from [13].

where $\mathbf{r}_b$ are the lattice sites making up the cylinder boundary. Figure 3.25 presents drag force coefficients averaged from 2000 to 3000 iterations. The behavior is the one expected for low and moderate Reynolds numbers. But our LBGK model does not seem to be able to simulate a flow at very high Reynolds number (i.e $R_e > 10^5$) as the expected drop is not captured.

3.8 Summary

The present chapter has been devoted to the presentation of the lattice Boltzmann (LB) models. The reader interested in a comparison between LB and finite difference methods for simple geometries may have a look at [68, 69]. Moreover, Bernsdorf et al. [70] illustrate the high geometrical flexibility of LB methods by considering its application to the numerical simulation of a porous sedimentary layer. We essentially learn from these comparisons that LB methods are competitive with CFD methods. However, both methods (LB and CFD) are not universal tool. Indeed, some applications are easily simulated with LB models while this is not the case with CFD models, and inversely! Chapter 5 is a good example of simulations well captured by LB methods.

To conclude, let us summarize the main points of this chapter. After recalling some fluid dynamics essentials, we gave a simple and a pragmatic explanation of LBGK models. Basic codes were also provided to help accelerate the learning

process for beginners. The famous bounce-back halfway and fullway boundary conditions are carefully described. After developing the 3D expression of the Inamouro non-slip boundary condition, we highlighted theoretically and practically that the Inamouro non-slip boundary condition does not ensure mass conservation. A new boundary condition called mass conservation boundary condition, was proposed. It ensures the conservation of the total mass. Then, the various ways to settle a flow were discussed. The Smagorinsky turbulence model was described in the LBGK context. Through experiments, we validated the use of this turbulence model but stressed the fact that high Reynolds number can not be achieved although no numerical instabilities happen. Finally, two simulations were presented. They deal with concepts presented in this chapter and shows that LB models reproduce quite well some simulation with a reasonable CPU effort (ranging from a few seconds to a few hours).

Chapter 4

Mesh refinement

We present in this chapter the way to refine a lattice. This process is usually called mesh refinement. We start with the motivations and the general ideas of the mesh refinement. Next, we review the main contributions and describe our approach which is new and promising. We validate our new refinement technique on two examples. Finally, we present a way to accelerate the flow settlement by using a mutli-grid approach.

4.1 Motivations

We mentioned previously that neglecting some error terms (up to second order in space and time), fluid governing equations can be recovered by using the LBGK collision operator (see equation (3.6)). These error terms are related to the lattice spacing $\Delta\mathbf{r}$ and the time step Δt. Hence, both variables have to be taken as small as possible in order to minimize the error.

On the other hand, the computational time increases polynomially with the dimension on $\Delta\mathbf{r}$ and Δt. Also, the necessary memory space is related to $\Delta\mathbf{r}$. Thus, to save time and memory space, one has to set $\Delta\mathbf{r}$ and Δt as large as possible.

Usually, there is a compromise between computational time and precision of the results. An other way, rather common in CFD, is to adapt locally $\Delta\mathbf{r}$ and Δt. This is technically realized by defining different lattices covering different parts of the domain. Places where more precision is needed, are covered by a finer lattice. Elsewhere, a coarser lattice is defined which requires less computation time.

4.2 General ideas

Refining a lattice consists essentially in dividing the lattice spacing $\Delta\mathbf{r}$ by a refinement factor n_{ref}. Note that the latter is usually set to 2 for practical reasons essentially due to the parallelization of the model. Indeed, a refinement

factor greater than 2 would imply an increase of the distance between some sites of the fine lattice, which need an interpolation, and the sites of the coarse lattice. The lattice decomposition necessary to a parallel execution would be also more difficult.

In order to keep the molecular velocities identical, a common choice is also to divide the time step Δt by n_{ref}. We consider here only one refinement level leading to a fine and a coarse lattice. However, a multi-refinement can be easily achieved and does not affect the following derivation.

To connect lattices, one has to ensure that their physical quantities are identical. By changing $\Delta \mathbf{r}$ and Δt by the same refinement factor, we observe in table 3.3 that the viscosity is the only physical quantity affected by this modification. The relaxation time is used to ensure that the viscosity coincides on both lattices. From equation (3.12), one shows that

$$\tau_f = \frac{\Delta \mathbf{r}_c}{\Delta \mathbf{r}_f}\left(\tau_c - \frac{1}{2}\right) + \frac{1}{2} \tag{4.1}$$

where subscripts f and c denote quantities of fine and coarse lattices respectively. Figure 4.1 represents a typical lattice refinement.

By rescaling the relaxation time, physical quantities are identical on both lattices. However, their connection requires also a rescaling of the fields f_i. This point is treated differently depending on the author. We present and then discuss these different techniques.

4.3 Existing models

4.3.1 The Filippova model

The well-known grid refinement model is the one due to Filippova et al. [55]. Their main results are reported here. In their scheme, they proposed the following relations between the fields of the fine and the coarse lattice:

$$\begin{aligned} f_i^{out,c} &= f_i^{eq,f} + (f_i^{out,f} - f_i^{eq,f})\frac{n_{ref}(\tau_c - 1)}{\tau_f - 1} \\ f_i^{out,f} &= \tilde{f}_i^{eq,c} + (\tilde{f}_i^{out,c} - \tilde{f}_i^{eq,c})\frac{\tau_f - 1}{n_{ref}(\tau_c - 1)} \end{aligned} \tag{4.2}$$

where $\tilde{f}_i$ denotes the spatially and temporally interpolated value of the coarse grid fields and c and f indicate quantities belonging to the coarse or the fine lattice respectively. Note that the whole computational domain is covered with the coarse lattice. Finer lattices are defined locally in certain regions, e.g. around obstacles. Algorithm 4.1 presents how Filippova et al. deal with both lattices.

They showed [71] that for steady state flows, the numerical scheme can be accelerated by use of only one subcycle in step 2.

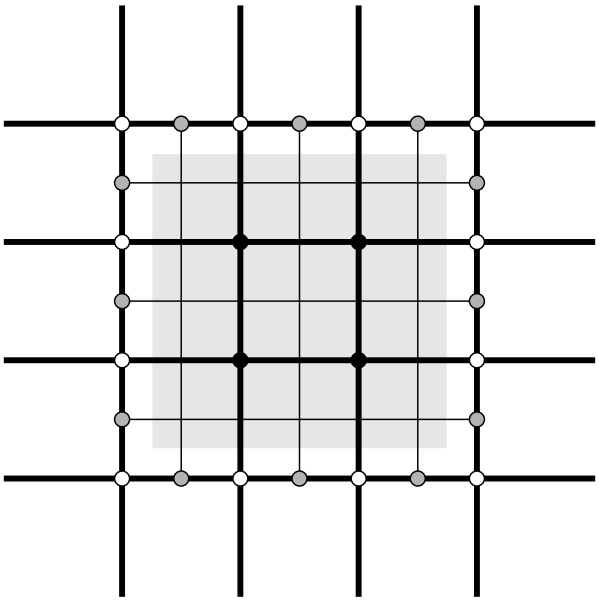


Figure 4.1: Local refinement of a lattice covering the entire domain. The refinement factor is $n_{ref} = 2$ and the fine lattice has 7×7 sites. White circles indicate common lattice sites which are on the boundary of the fine lattice. Gray circles are lattice sites on the boundary of the fine lattice which belong to it only. Black circles represent the internal lattice sites common to both lattices. The gray area shows the fine lattice sites which are not directly set by the coarse fields.

4.3.2 The Lin model

Lin et al. [72] proposed a simpler algorithm without considering a rescaling process. They argued that the fields f_i are interchangeable after the streaming step in spite of their different characteristic scales and accuracy. Considering $n_{ref} = 2$, algorithm 4.2 shows their technique.

They successfully simulated a cavity flow considering a coarse lattice covering the entire domain and finer lattices simulating more precisely the corners. They carefully analyzed the impact of the refinement on the computational time.

Two major remarks on the existing models presented above have to be made. First, equations(4.2) of the Filippova model present a singularity when $\tau = 1$ which reduces the generality of the model. Second, the Lin model is slightly inaccurate as they consider the fields as interchangeable. This is not the case although the difference between coarse and fine fields is small.

1. Do two propagation-collision steps on the coarse lattice. Two steps are necessary for the time interpolation.
2. Do n_{ref} times
   (a) With second-order interpolation in space and time, compute $\tilde{f}_i^{out,c}$ where it is necessary on the boundary of the fine lattice (gray circles in figure 4.1).
   (b) On the same boundary (gray and white circles in figure 4.1), compute $\tilde{f}_i^{eq,c}$ ($\tilde{f}_i^{out,c}$ allow to compute ρ and $\mathbf{u}$).
   (c) According to equation (4.2), compute $f_i^{out,f}$ on the boundary of the fine lattice (gray and white circles in figure 4.1).
   (d) Do one propagation-collision step on the fine lattice (gray area in figure 4.1). Note that the collision process do not occur at the boundary.
3. On nodes common to both lattices except the ones at the boundary (black circles in figure 4.1), compute $f_i^{eq,f}$ ($f_i^{out,f}$ allow to compute ρ and $\mathbf{u}$) and then $f_i^{out,c}$ according to equation (4.2).
4. Go back to 1 until the simulation is finished.

Algorithm 4.1: Evolution algorithm of the Filippova model.

1. Do two collision-propagation steps on the coarse lattice. Two steps are necessary for the time interpolation.
2. Do 2 times
   (a) With an interpolation in space and time, compute $f_i^{in,f}$ from $f_i^{in,c}$ where fields are missing on the boundary of the fine lattice (gray circles in figure 4.1).
   (b) Compute $f_i^{in,f}$ from $f_i^{in,c}$ on common sites on the boundary of the fine lattice (white circles in figure 4.1).
   (c) Do one collision-propagation step on the fine lattice.
3. Go back to 1 until the simulation is finished.

Algorithm 4.2: Evolution algorithm of the Lin model.

4.4 A new approach

Recall that a field can be decomposed into its equilibrium and its non-equilibrium part as

$$f_i^{in} = f_i^{eq} + f_i^{neq}.$$

Let us examine them. First, equation (3.13) shows that the equilibrium part is a function of ρ and $\mathbf{u}$ expressed as

$$f_i^{eq} = \rho t_i \left[1 + \frac{v_{i\alpha}u_\alpha}{c_s^2} + \frac{1}{2}\left(\frac{v_{i\alpha}u_\alpha}{c_s^2}\right)^2 - \frac{\mathbf{u}^2}{2c_s^2}\right] = f(\rho, \mathbf{u}).$$

It is neither dependent on $\Delta\mathbf{r}$ nor Δt. Second, equation (3.38) indicated that the non-equilibrium part yields

$$f_i^{neq} = \frac{\Delta t\ \tau}{C_2 v_l^2}(v_{i\gamma}v_{i\delta}\partial_\gamma \rho u_\delta - c_s^2 \mathrm{div}\rho\mathbf{u}) = \Delta t\ \tau\ C(\rho, \mathbf{u}) \tag{4.3}$$

where $C(\rho, \mathbf{u})$ is a function of ρ and $\mathbf{u}$ only. The quantities depending on ρ and $\mathbf{u}$ are identical on lattices of different scales as ρ and $\mathbf{u}$ are lattice-blind. Consequently, only the non-equilibrium part has to be rescaled when one wishes to connect different lattices.

It follows that one can express a relation between fields of different lattices as

$$f_i^{eq} = f_i^{eq,f} = f_i^{eq,c} = f_i^{eq}. \tag{4.4}$$

Considering equation (4.3) and the fact that $C(\rho, \mathbf{u})$ is identical on both lattices, the following relation between non-equilibrium distributions can be written

$$f_i^{neq,f} = \frac{\Delta t_f\ \tau_f}{\Delta t_c\ \tau_c} f_i^{neq,c} = \frac{\tau_f}{n_{ref}\tau_c} f_i^{neq,c}. \tag{4.5}$$

Combining equations (4.4) and (4.5), one can easily express how to transform coarse to fine fields

$$\begin{aligned} f_i^{in,c} &= f_i^{eq} + f_i^{neq,c} \\ &= f_i^{eq} + f_i^{neq,f}\frac{n_{ref}\tau_c}{\tau_f} \\ &= f_i^{eq} + (f_i^{in,f} - f_i^{eq})\frac{n_{ref}\tau_c}{\tau_f} \end{aligned} \tag{4.6}$$

Similarly one can express the fine fields transformation. Hence, fields are transformed by the following relations

$$\boxed{\begin{aligned} f_i^{in,c} &= f_i^{eq} + (f_i^{in,f} - f_i^{eq})\tfrac{n_{ref}\tau_c}{\tau_f} \\ f_i^{in,f} &= \tilde{f}_i^{eq} + (\tilde{f}_i^{in,c} - \tilde{f}_i^{eq})\tfrac{\tau_f}{n_{ref}\tau_c} \end{aligned}} \tag{4.7}$$

where $\tilde{f}_i$ denotes the spatially and temporally interpolated value of the coarse grid fields.

Let us give some details about interpolation in space and time. We start with the space interpolation. Suppose that the field $f_i^{in,f}(\mathbf{r})$ is missing and that $\mathbf{r}$ is not a coarse lattice site. Thus, one needs to compute $\tilde{f}_i^{in,c}$ to use the transformation equation (4.7). Suppose that there are k_n neighbors involved in the interpolation located at position $\mathbf{r}_j$ with $j \in \{1, 2, \cdots, k_n\}$. Note that k_n is usually equal to 2 but could be equal to 3 in the case of oblique or curved boundaries. The unknown fields amount to

$$\tilde{f}_i^{in,c}(\mathbf{r}) = \frac{1}{D_r} \sum_j^{k_n} d_j f_i^{in,c}(\mathbf{r}_j)$$

where d_j is the distance between $\mathbf{r}$ and $\mathbf{r}_j$ and $D_r = \sum_j d_j$.

Let us have a look at the interpolation in time. It is necessary to maintain a correspondence between both lattices. Indeed, some iterations occur at times on the fine lattice which do not exist on the coarse one, i.e. not multiples of Δt_c. Hence, the interpolated fields have to take this into account. Suppose that the fields of the coarse lattice are known at time t_0 and $t_{n_{ref}}$. The interpolation in time can be computed as

$$\tilde{f}_i^{*,c}(t_k) = f_i^{*,c}(t_0) + \frac{k}{n_{ref}}(f_i^{*,c}(t_{n_{ref}}) - f_i^{*,c}(t_0)) \ , \quad 0 \leq k \leq n_{ref}$$

where the subscript $*$ denotes either in or eq.

Considering a coarse lattice covering the entire domain and some local fine lattices, algorithm 4.3 illustrates how our new approach works.

Fine lattices use information from the coarse lattice only at its boundaries. This may suggest that the coarse lattice should not cover the entire domain but only the parts where finer lattices are not defined. This is actually a bad suggestion and there are many reasons to set a coarse lattice covering all the domain. Principally, it would be technically difficult to determine coarse fields at boundaries if the lattice does not cover the whole domain. Moreover, the computer code dealing with an entire lattice instead of splitting it in many coarse lattices, is made easier. Performances would probably be affected especially due to cache memory effects. See section 6.6 for a description of these effects.

This approach has the advantage of being more general than the one of Filippova et al. [55]. Indeed, the singularity arising when $\tau = 1$ is not present anymore. Moreover, the collision operator is applied also on the boundary of finer lattices which is not the case in the Filippova model. This approach is obviously more accurate than the simple one proposed by Lin et al. [72] as they do not consider the non-equilibrium part of the distribution.

The next section is devoted to the validation of our approach and to the numerical comparison of the three techniques presented here.

1. Do two collision-propagation steps on the coarse lattice. Two steps are necessary for the time interpolation.
2. Do n_{ref} times
   (a) By interpolating in space and time, compute $\tilde{f}_i^{in,c}$ where fields are missing on the boundary of the fine lattice (gray circles in figure 4.1).
   (b) On the same boundary (gray and white circles in figure 4.1), compute $\tilde{f}_i^{eq,c}$ ($\tilde{f}_i^{in,c}$ allow to compute ρ and $\mathbf{u}$).
   (c) According to equation (4.7), compute $f_i^{in,f}$ on the boundary of the fine lattice (gray and white circles in figure 4.1).
   (d) Do one collision-propagation step on the fine lattice (gray area and the boundary in figure 4.1).
3. On common nodes to both lattices except the ones at the boundary (black circles in figure 4.1), compute $f_i^{eq,f}$ ($f_i^{in,f}$ allow to compute ρ and $\mathbf{u}$) and then $f_i^{in,c}$ according to equation (4.7).
4. Go back to 1 until the simulation is finished.

Algorithm 4.3: Evolution algorithm of our new approach.

4.5 Validation

4.5.1 Field decomposition

We start this validation part by highlighting the field decomposition into an equilibrium and a non-equilibrium part. For that, we consider a D2Q9 LBGK Poiseuille flow on a channel of length L and diameter $D = 1.0$. It is discretized on a lattice $N_x \times N_z = 11 \times 33$ which is longitudinally periodic. The flow is settled by imposing a constant body force and the Inamouro non-slip boundary condition is used. With the present boundary conditions a refinement amounts only to a z direction refinement. Hence a 11×65 fine lattice can be defined which corresponds to a refinement factor $n_{ref} = 2$. We set the viscosity $\nu = 0.005$ and the maximum velocity $U_c = 0.1$. Assume that $\Delta\mathbf{r}/\Delta t = C$ where C is a constant value. Let say that $C = 1$ where unities are intentionally omitted for simplicity reasons. Thus, one can easily compute the relaxation times as

$$\tau_c = \frac{C_2}{C_4 \Delta \mathbf{r}_c} \cdot \nu + 0.5 = \frac{12}{4 \cdot 1/32} \cdot 0.005 + 0.5 = 0.98$$
$$\tau_f = \frac{C_2}{C_4 \Delta \mathbf{r}_f} \cdot \nu + 0.5 = \frac{12}{4 \cdot 1/64} \cdot 0.005 + 0.5 = 1.46$$

where τ_c and τ_f are the relaxation times of the coarse and the fine lattice respectively. Forcing terms are computed as

$$\begin{aligned} g_{ix}^c = t_i v_{ix} \frac{\Delta \mathbf{r}_c C_2}{C_4} \cdot \frac{8\nu U_c}{D^2} = t_i v_{ix} \frac{12}{32 \cdot 4} \cdot \frac{8 \cdot 0.005 \cdot 0.1}{1^2} &= 3.75 \cdot 10^{-4} \cdot t_i v_{ix} \\ g_{ix}^f = t_i v_{ix} \frac{\Delta \mathbf{r}_f C_2}{C_4} \cdot \frac{8\nu U_c}{D^2} = t_i v_{ix} \frac{12}{64 \cdot 4} \cdot \frac{8 \cdot 0.005 \cdot 0.1}{1^2} &= 1.875 \cdot 10^{-4} \cdot t_i v_{ix} \end{aligned}$$

where g_{ix}^* are the forcing terms where the symbol $*$ denotes the coarse or the fine lattice. Note that $g_{iz}^* = 0$ for both lattices.

Arbitrarily, the 9 fields f_i are measured at the point ($x = 0$, $y = D/4$) common to both lattices. As expected, their equilibrium parts are identical up to the discretization errors (i.e. $\left| f^{eq,f} - f^{eq,c} \right| < 10^{-8}$). Thus, we focus our attention on the non-equilibrium part of the fields which is computed as $f_i^{neq} = f_i^* - f_i^{eq}$ where the symbol $*$ indicates if this part is computed considering the incoming or the outcoming fields, i.e. using Filippova's model or ours. We then talk about incoming or outcoming non-equilibrium part. Note that when nothing is specified the incoming non-equilibrium part is considered.

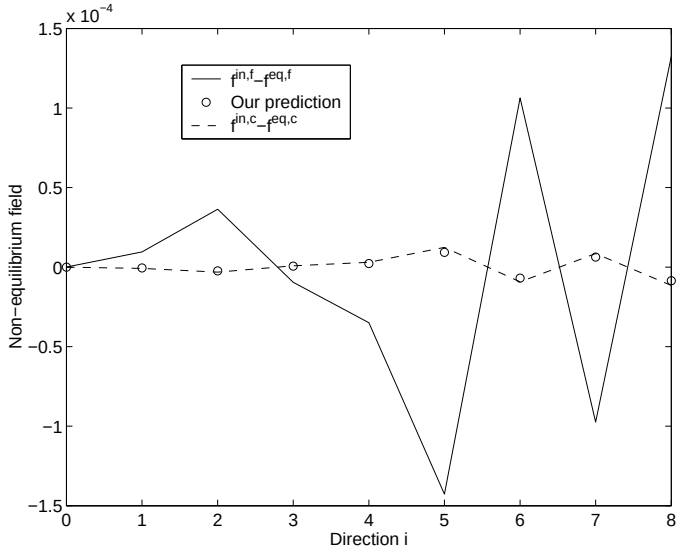
We simulate the stationary flow on the coarse and on the fine lattice. The two sets of transformation equations (4.7) and (4.2) are used to switch from coarse to fine fields. The results simulated on the fine lattice are used to check the accuracy of each method. The results are presented in figure 4.2. Notice that both transformation equations, Filippova's and ours, produce accurate results. Finally, we observe that the non-equilibrium part is rather small. It represents a small percentage of the total, i.e. $f^{neq}/f \ll 1$.

From this first validation, we conclude that the non-equilibrium part is non-zero and dependent on $\Delta \mathbf{r}$ and Δt. Hence, one has to rescale them to use the values of one lattice to set the values of an other. Moreover, our new approach and the one of Filippova seem to give similar results. We also argue that the Lin model presented above misses some aspects of the LBGK models as it does not consider a rescaling process. Then, depending on the simulation (if gradients are large), it can be prejudicial to use this model.

4.5.2 Local refinement of a Poiseuille flow

We continue our validation by considering a local lattice refinement. Consider again a LBGK D2Q9 Poiseuille flow on a coarse $N_x \times N_z = 10 \times 17$ lattice which is longitudinally periodic. The coarse lattice is locally refined by considering a patch 10×15 which refines the first 7 sites of the coarse lattice along z direction. Note that a non-periodic flow would also imply a longitudinal refinement. The flow is settled by applying a constant body force and the Inamouro non-slip boundary condition is used. The diameter of the channel $D = 1.0$, the viscosity in the center of the channel $U_c = 0.1$. In order to highlight the singularity around

Our transformation



Filippova's transformation

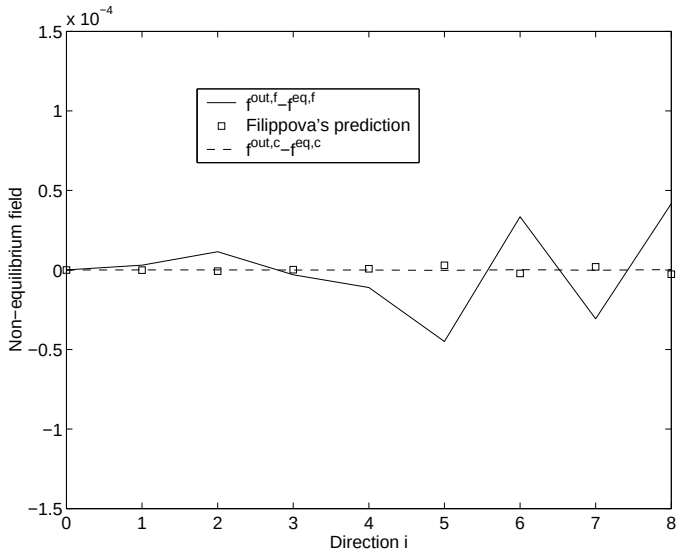


Figure 4.2: Prediction of coarse from fine non-equilibrium fields using (top) our transformation and (bottom) Filippova's transformation equations, see equations(4.7) and (4.2) respectively. We consider a Poiseuille flow on a D2Q9 LBGK in a channel of length L and diameter D. The measurements are done at the point ($x = 0$, $y = D/4$). The space is discretized on a 16×33 coarse lattice and a 16×65 fine lattice. Both lattices are longitudinally periodic. The flow is settled by imposing a constant body force and the Inamouro non-slip boundary condition is used. We set the viscosity $\nu = 0.005$ and the maximum velocity $U_c = 0.1$. We observe accurate transformation for both cases.

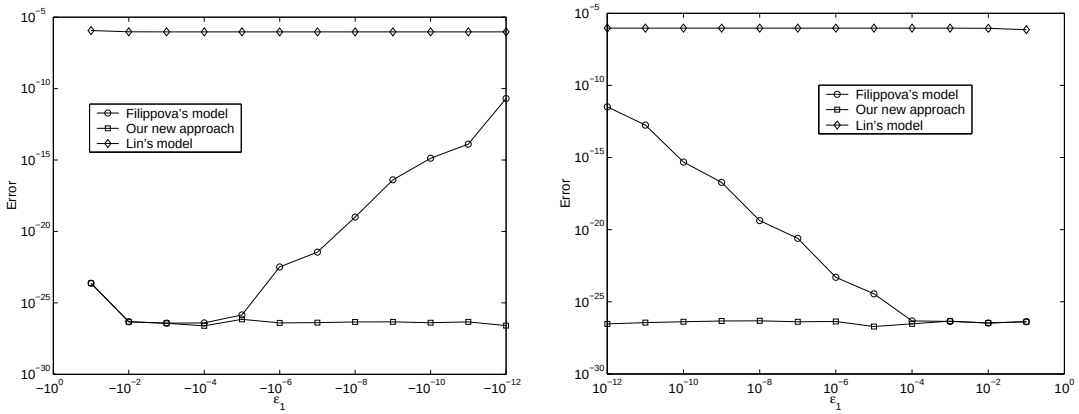


Figure 4.3: Error defined by equation (4.8) on the fine lattice considering $\tau_c = 1 \pm \epsilon_1$ for the three presented algorithms. The error is presented for (a) negative and (b) positive values of ϵ_1.

$\tau_c = 1$ in the Filippova model, we choose $\tau_c = 1.0 \pm \epsilon_1$. The relaxation time on the fine lattice is computed from equation (4.1).

Using algorithms 4.1, 4.2 and 4.3 we simulate the flows on both lattices. Missing fields of the fine lattice on the top layer (i.e. $z = 14 \cdot \Delta\mathbf{r}_f$) are then determined by the coarse ones.

With the considered Inamouro's boundary conditions, simulations results on any (complete) lattice differ from the theoretical ones only by the numerical precision of the computer[1]. Hence, the results of the coarse lattice can be considered as exact. This is not the case for the results of the fine lattice as it is not complete. So in order to check the accuracy of the three algorithms we measure an error on the fine lattice as

$$\text{error} = \sqrt{\frac{\sum_z (u_x(z) - U_x(z))^2}{15}} \tag{4.8}$$

where $u_x(z)$ is the simulated velocity and $U_x(z) = 4\frac{U_c(z\Delta\mathbf{r}_f)}{D}(1 - \frac{z\Delta\mathbf{r}_f}{D})$ is the theoretical velocity. Figure 4.3 presents this error for $\tau_c = 1 \pm \epsilon_1$. We observe that the Filippova model and our approach have the same error which is close to the computational numerical error. However for $1 - 1 \cdot 10^{-3} \leq \tau_c \leq 1 + 1 \cdot 10^{-3}$ the Filippova model blows up and is consequently non-usable. On the other hand, we note that the Lin model, which does not rescale the fields, has a larger error.

We attract the attention of the reader to the fact that the use of body forces to settle a flow requires a special treatment as they are not identical on different lattices. One has to take this into account when employing an algorithm using

[1]Considering a Poiseuille flow, recall that Inamouro's condition produces exact results.

incoming fields. Indeed, usually a body force is added during the collision process (see code 3.9). Computed fields from equation (4.7) would then not see the body force at the right scale. There is two solutions to this problem. First, it simply consists in adding the body force just after the propagation process making the incoming fields. Consequently, the body force would be added at the right scale. Second, one could take into account a body force rescaling in the transformation equation (4.7).

4.6 Flow settlement acceleration by using a multigrid approach

4.6.1 Description

We now describe a useful utilization of the multigrid techniques: the acceleration of the flow settlement. Indeed, a non-negligible part of the computation time is devoted to settle the flow. Thus, one is interested in reducing this time by accelerating the flow settlement process.

Note that the larger the lattice, the longer the time to settle the flow. So considering a lattice L_0 with a lattice spacing equal to $\Delta\mathbf{r}_0 = \Delta\mathbf{r}$, we propose to settle the flow on a lattice L_1 twice as coarse as L_0, i.e. with a lattice spacing $\Delta\mathbf{r}_1 = \Delta\mathbf{r}/2$, and use the flow in L_1 as initial condition for L_0. To settle the flow on L_1, one can again consider a coarser lattice L_2. And so on. Hence, we have a hierarchical process iterated l_{ref} times which accelerates the flow settlement process.

Let us present the way to connect two successive lattices. We learned from experiments presented above that only the non-equilibrium part has to be rescaled (equilibrium parts being, neglecting discretization error, identical). The non-equilibrium part represents a few percent of the total. Moreover the difference between non-equilibrium fields of different lattices is rather small. Considering that the connection between lattices L_i and $\mathsf{L}_{\mathsf{i}+1}$ is made once, we argue that it is unnecessary to rescale the fields. Indeed, the non-rescaling error is insignificant compared to the error made by interpolating unknown fields of sites present in lattice $\mathsf{L}_{\mathsf{i}+1}$ only. Hence, fine fields common to both lattices are set with coarse fields without rescaling, the others are spatially interpolated.

A stopping criterion is needed to stop automatically the process on lattice L_i in order to start the process on lattice $\mathsf{L}_{\mathsf{i}-1}$. There are various ways to interrupt the process. The simplest consists in stopping the flow if the difference between two given successive values in time is smaller than ϵ_{ref}. Depending on the flow, the quantity ϵ_{ref} has to be very small to stop the process at the right time. On the other hand, if a specific characteristic of the flow is known before the simulation, one can use it to improve the stopping criterion.

The way to accelerate the flow settlement process is summarized in algorithm 4.4.

1. Allocate memory space for lattice $\mathsf{L}_{\mathsf{l}_{\mathsf{ref}}}$.
2. Initialize $\mathsf{L}_{\mathsf{l}_{\mathsf{ref}}}$. For example, with the equilibrium distribution function.
3. Do l_{ref} times
   (a) Do one collision-propagation step on the current lattice.
   (b) If the stopping criterion is reached
      i. If the current lattice is L_0 go to 4.
      ii. Allocate memory space for the next lattice.
      iii. Set the fields of the current lattice to the next one.
      iv. Spatially interpolate the missing fields on the next lattice.
      v. Deallocate memory space of the current lattice.
   (c) Else go back to (a).
4. Continue the simulation with an established flow.

Algorithm 4.4: Algorithm for the acceleration of the flow settlement process.

4.6.2 Application

We apply this hierarchical process to settle a LBGK D2Q9 Poiseuille flow on a longitudinally periodic 10×257 lattice. Due to the longitudinal periodicity of the lattice, a refinement level consists in considering a lattice twice as coarse only in the z direction. The flow is settled by imposing a constant body force and the Inamouro non-slip boundary condition is used. We set the viscosity $\nu = 0.005$ and the maximum velocity $U_c = 0.1$. Table 4.1 presents the results.

Let us give some explanations on how to read table 4.1. Horizontally, it is decomposed on three rows one per stopping conditions which are detailed below. A special row, indicating the computational time necessary to satisfy a given stopping condition for all lattices of a given refinement level, is inserted. This level indicates the number of extra lattices used to produce the flow on the finer lattice. Vertically, we report the number of iterations to reach a stopping condition considering a given refinement level. The first column is the number of lattice sites corresponding to the refinement level considered. Finally, the last column reports the gain G_r to initialize a given lattice as a fluid at rest or with the fields of a coarser lattice, see below for details.

Three stopping criteria have bee taken into account ((a), (b) and (c)). First, the longitudinal simulated velocity u_x is measured on the middle of the channel at

	N_z	Refinement level 0	1	2	3	4	5	6	7	G_r [%]
(a)	257	134018	84418	83589	83750	84054	84008	83291	83833	37
	129	-	68757	44372	44464	44334	44504	44408	44543	35
	65	-	-	35149	20926	20928	20975	20908	20895	40
	33	-	-	-	18069	13516	13501	13506	13424	25
	17	-	-	-	-	9260	7524	7527	7491	19
	9	-	-	-	-	-	4688	4084	4090	13
	5	-	-	-	-	-	-	2315	2137	8
	3	-	-	-	-	-	-	-	1025	-
		402.9	340.1	318.7	322.0	324.9	324.5	321.2	322.4	
		Elapsed time [s]								
(b)	257	83839	34258	34272	34272	34272	34272	34272	-	59
	129	-	41917	17799	17799	17799	17799	17799	-	58
	65	-	-	20954	6673	6673	6673	6673	-	68
	33	-	-	-	10468	5874	5874	5874	-	44
	17	-	-	-	-	5216	3497	3497	-	33
	9	-	-	-	-	-	2573	1969	-	23
	5	-	-	-	-	-	-	1217	-	-
		263.6	149.5	134.9	132.6	126.7	130.1	131.3	-	
		Elapsed time [s]								
(c)	257	79580	29994	30017	30017	30017	30017	30017	30017	62
	129	-	40678	16559	16559	16559	16559	16559	16559	59
	65	-	-	20775	6493	6493	6493	6493	6493	69
	33	-	-	-	10596	6002	6002	6002	6002	43
	17	-	-	-	-	5387	3668	3668	3668	32
	9	-	-	-	-	-	2714	2109	2109	22
	5	-	-	-	-	-	-	1324	1148	13
	3	-	-	-	-	-	-	-	577	-
		253.5	143.2	127.3	120.5	122.5	119.0	123.3	123.9	
		Elapsed time [s]								

Table 4.1: Evolution of the necessary number of iterations to simulate a D2Q9 LBGK Poiseuille flow on a 10×257 lattice. We consider three stopping conditions: (a) $|u_x^t(\lfloor N_z/2\rfloor)-u_x^{t-1}(\lfloor N_z/2\rfloor)|<10^{-16}$, (b) $|u_x^t(\lfloor N_z/2\rfloor-1)-U_x(D/2-\Delta r_z)|<10^{-8}$ and (c) $\sqrt{\sum_{i_z}(u_x^t(i_z)-U_x(i_z\Delta r_z))^2/N_z}<10^{-9}$ where u_x and U_x are the simulated and theoretical longitudinal velocities respectively. The refinement level indicates the number of coarse lattices considered. Computation times on a Pentium III/1.5 GHz under Linux are reported below each simulation results. The quantity G_r expresses the gain for starting the simulation with the fields of the coarse lattice rather than with a fluid at rest.

time t and compared with the one at time $t-1$. If their difference is smaller than $\epsilon_{ref} = 10^{-16}$ then the process stops. Note that a bigger value of ϵ_{ref}, say 10^{-10}, would make an earlier stop because the increase of velocity is small especially when one considers a body force. Hence, one has

$$\left| u_x^t \left(\left\lfloor \frac{N_z}{2} \right\rfloor \right) - u_x^{t-1} \left(\left\lfloor \frac{N_z}{2} \right\rfloor \right) \right| < \epsilon_{ref} = 10^{-16}$$

Second, we use the well-known parabola velocity profile of the Poiseuille flow $U_x(y) = 4U_c y/D(1 - y/D)$ to improve the stopping criterion. Precisely, we compare the longitudinal simulated velocity at height $\lfloor N_z/2 \rfloor - 1$ with its theoretical corresponding value U_x. The exact center of the channel has not been considered because this lattice site exists on all lattices and is consequently exact when the criterion is reached. The next lattice would also have the exact value and then immediately stop the process. The second stopping criterion yields

$$\left| u_x^t \left(\left\lfloor \frac{N_z}{2} \right\rfloor - 1 \right) - U_x \left(\frac{D}{2} - \Delta r_z \right) \right| < \epsilon_{ref} = 10^{-8}$$

Finally, we compare the entire longitudinal velocity profile with the expected profile to decide whether the process has to be stopped. Thus, one can write the condition

$$\sqrt{\frac{\sum_{i_z} (u_x^t(i_z) - U_x(i_z \Delta r_z))^2}{N_z}} < \epsilon_{ref} = 10^{-9} \tag{4.9}$$

Note that the different values of ϵ_{ref} have been chosen empirically because they produce good result. Other computers or other simulations would lead to other choices.

Looking at the results on table 4.1, we note that the first condition needs more iterations to stop and is rather noisy. Indeed, for a given N_z, the number of iterations is not the same depending on the initialization. This is not the case with the other stopping criteria. Moreover, we notice a larger number of iterations when one considers the second stopping criterion.

To quantify the benefit of initializing the system by the fields of a coarser lattice rather than starting from a fluid at rest, we introduce the refinement gain G_r. It is expressed in percent as

$$G_r = \frac{N_{rest} - N_{coarse}}{N_{rest}} \cdot 100$$

where N_{rest} and N_{coarse} are the numbers of iterations necessary to establish the flow initialized as a fluid at rest or by a coarser lattice respectively. Depending on the stopping criterion, we note in table 4.1 for large enough lattice, that the quantities G_r are approximately identical. Their values decrease with the size of the lattice.

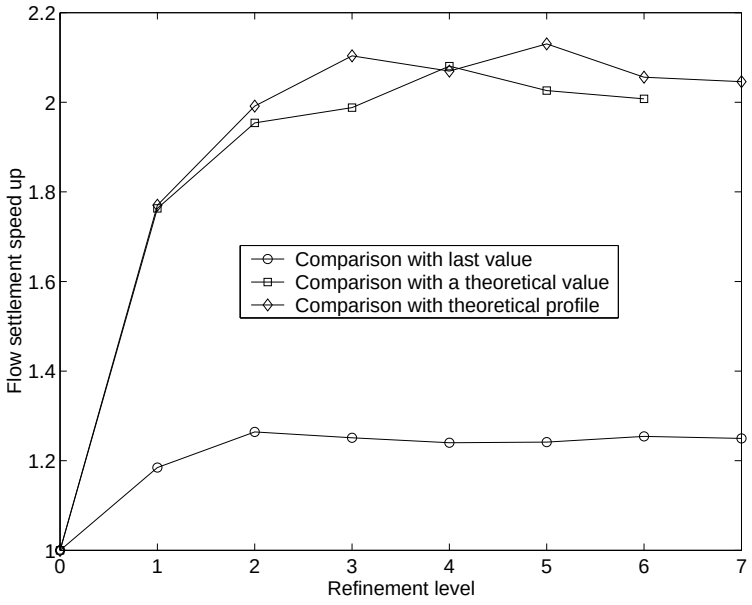


Figure 4.4: Speedups of the flow settlement process of a LBGK D2Q9 Poiseuille flow on a 10×257 lattice. The flow is settled by imposing a constant body force and the Inamouro non-slip boundary condition is used. We set the viscosity $\nu = 0.005$ and the maximum velocity $U_c = 0.1$. Speedups obtained with three different stopping criteria are reported.

Due to the longitudinal periodicity of the lattice, a refinement level consists in considering a lattice twice as coarse only in the z direction. This is the reason why we observe a factor 2 (and not 4) between the diagonal elements of table 4.1 for every stopping criteria.

With computational times reported on table 4.1, one can determine the benefit of considering such a hierarchical algorithm by computing a speedup. The speedup of the flow settlement process is defined as

$$S_i = \frac{t_0}{t_i}$$

where t_i is the computation time necessary to the refinement at level i. Then considering a refinement at level i, the quantity S_i indicates how many times the flow settlement process goes faster. The speedups of the present example are computed for the different stopping criteria and are reported in figure 4.4. We observe that an appropriate choice of stopping criterion leads to a speedup approximately equal to 2.2.

These speedups indicate that the biggest contribution to the gain in performance is obtained after only one refinement. However, the code realizing such an algorithm is straightforwardly changed into a hierarchical one. So we argue that

it is better to consider a large, if possible the maximum, number of refinement levels.

The end of this section is devoted to explain how the speedup is only around 2 and not bigger.

Note that the fields tend *exponentially* in time towards their steady state. Indeed, after few iterations they have almost reached their steady values, but a large number of iterations is still needed to satisfy any stopping criterion. The acceleration of the flow settlement process can only reduce this first part essentially because the error introduced by the interpolation is large at this scale. Then, any lattice must have itself performed a large number of iterations to reach its steady state. This explains why the above speedups are only equal to 2. The following example illustrates the explanation.

Figure 4.5 reports the evolution of the fields by reporting the values of the velocity in the middle of the channel when 0 and 7 refinement levels are considered. The stopping criterion is given by equation (4.9). The settings are the same as in figure 4.4. We notice the exponential form of the curves and the cusps indicating that a new finer lattice is considered and initialized by the previous one.

Table 4.1 indicates that the simulation considering 0 refinement level needs $8 \cdot 10^4$ iterations to satisfy the stopping criterion. This implies a computational time equal to $8 \cdot 10^4 \ \delta t_0$ seconds where δt_0 is the computational time to simulate one iteration. To compare the time evolution of different refinement levels, one needs to express the number of iterations in the same unity $\delta t_0 = 2^i \cdot \delta t_i$. Thus, the number of iterations to satisfy the stopping criterion when 7 refinement levels are considered is equal to

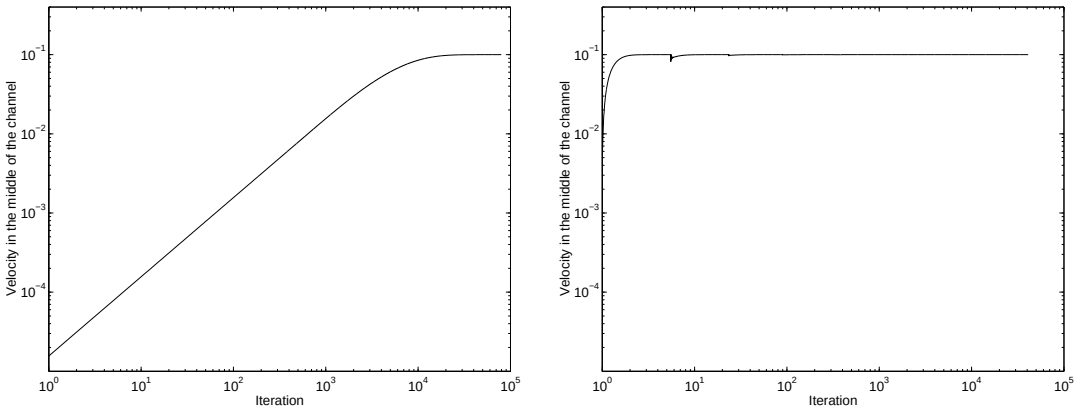
$$\frac{577}{2^7} + \frac{1148}{2^6} + \frac{2109}{2^5} + \frac{3668}{2^4} + \frac{6002}{2^3} + \frac{6493}{2^2} + \frac{16559}{2^1} + 30017 = 4.1 \cdot 10^4 \ \delta t_0$$

Figure 4.5(a) indicates that, after approximately $3 \cdot 10^4$ iterations, the middle velocity has almost reached its steady value. However, $5 \cdot 10^4$ iterations are needed to perform the simulation. Figure 4.5(b) shows that the first $3 \cdot 10^4$ needed in (a) are avoided by the acceleration process. However, one still needs to perform the second part to satisfy the stopping criterion.

4.7 Summary

In this chapter, we presented the basic ideas governing mesh refinement techniques. After a review of the existing models we showed our new approach.

Considering examples, we learned that the fields are not identical on different lattices. Their non-equilibrium part has to be rescaled. We also quantified an error on a simple flow when the fields f_i are rescaled or not. The difference is appreciable and can be important depending on the aim of the simulation.



(a) (b)

Figure 4.5: Evolution of the velocity in the middle of a channel. The settings are the same as in figure 4.4. (a) 0 and (b) 7 refinement levels have been considered. Iterations are measured in the common unity Δt.

The Filippova model singularity around $\tau = 1$ has been highlighted. Other values of τ produce the same small error when one uses our approach or the one proposed by Filippova.

Finally, we use the mesh refinement ideas to accelerate the flow settlement process. We obtained a speedup of 2.2.

Chapter 5

A virtual river model

5.1 Introduction

5.1.1 Motivations

Particle transport by a fluid is a physical phenomenon frequently occurring in nature. For example, water transports sediments in rivers, wind moves snow particles on mountains or sand particles in the desert. In this thesis, we focus our attention on erosion-transport phenomena occurring in marine environments. The simplest phenomena are quite well understood essentially because they are reproducible in laboratory or numerically (see subsection 5.1.2 for details). Let us consider some examples.

Dam managers have to periodically flush the reservoir which is formed by the river upstream of the dam. This part is sometimes almost at rest when water gates are almost closed. Hence, the resulting weak current implies that sediments transported by the river are more susceptible to deposit. After a few years, the reservoir has to be flushed for transporting the deposit beyond the dam. Otherwise the river bed would change implying some undesirable floods. A real example as well as details on the way to flush a reservoir are presented in appendix B. Other examples are the erosion phenomena around marine structures due to currents or waves. Some special protections have to be designed in order to prevent any damage. Figure 5.1 shows a nice example of scours around a submarine structure. These examples illustrate some erosion phenomena for which basic mechanisms of their formation are understood. However, more complicated ones such as the formation and the evolution of meanders in rivers or the erosion occurring in coastal environment (e.g. erosion at a river mouse) still need a complete explanation.

A model in laboratory is usually too expensive and too complicated to be undertaken. Hence, a numerical model is necessary for a better understanding of these phenomena. Numerical models are also important to predict the effects of erosion processes in order to reduce the possible risks related to their formations

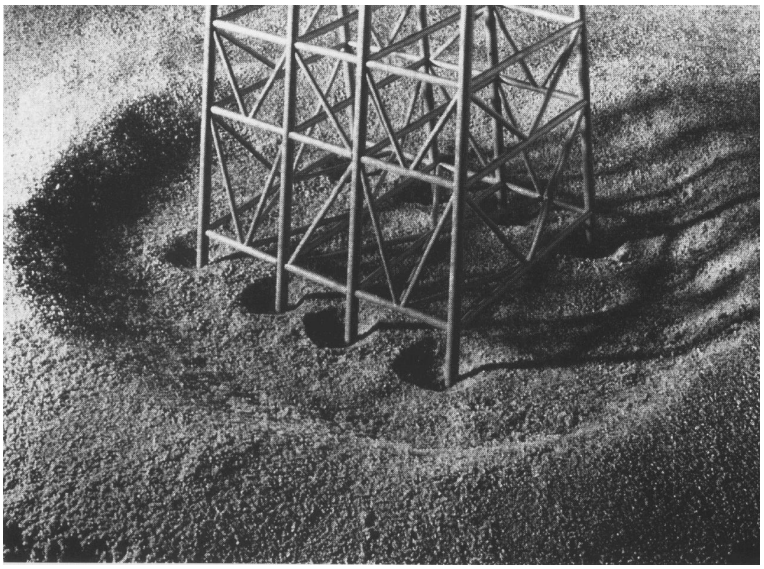


Figure 5.1: Representation of global and local scour development around a jacket structure. This picture has been scanned in [6].

or to adapt the design of surrounding infrastructures.

In this chapter, we define the salient features of our numerical model, the so-called virtual river model. It is composed of two basic ingredients: a lattice Boltzmann model for the water and a cellular automata for the sediments.

Other components typically found in rivers, such as fishes or vegetation, have not been taken into account as they are not directly related to erosion processes. However, the interested readers can consult some of the papers presented in [73] where their consideration is crucial.

5.1.2 State of the art

Models of sediment transport in rivers can be classified into two categories: laboratory and numerical experiments. The first one has the major advantages of being accurate and of reproducing exactly the expected behavior. On the other hand, laboratory experiments deal generally with too simple flows and geometries. Their construction is expensive and needs a lot of room. Also, the measurement processes require sophisticated and expensive tools. Some examples are described in [74, 75, 5, 6].

Numerical experiments are then an alternative. Usually, they are based on an iterative process which consists in computing the velocity repartition by a finite difference scheme and then determining the change of the bed. Some numerical models are presented in [76, 77, 78, 6]. They are more manageable and less expensive than the laboratory experiments but they hardly deal with complicated

geometries. It is also difficult to interact with elementary components of the flow (e.g. sediments) as they are usually approximated by a mean value.

Our virtual river model has the main quality of the usual numerical models, but it also allows us to deal with complicated geometries and to directly interact with sediments by defining special rules. This model has been applied and studied by my predecessor A.Masselot in his thesis [79] and references therein. He used this model to simulate the transport of snow by wind. In the context of a virtual river, snow is replaced by sediment and wind by water. The main difference between sediment and snow particles is the high cohesion of the latter. Indeed, sediment particles can not pile up indefinitely and have to topple. A new toppling rule, detailed below, is then added to the model. The applications proposed below are promising.

5.1.3 Overview of the model

The fluid-sediment system is described in terms of a mesoscopic dynamics: fictitious fluid and sediment particles move on a regular lattice, synchronously at discrete time steps. Recall that an interaction is defined (equation (3.6)) between fluid particles that meet simultaneously at the same lattice site. Fluid particles obey collision rules which reproduce, in the macroscopic limit, the Navier-Stokes equation.

The granular material moves under the combined effect of the local fluid velocity field and gravity. Particles reaching the ground solidify, pile up and topple if necessary. Hence, they change the shape of the boundary of the fluid. At the top of the deposition layer, erosion takes place and, if the fluid flows fast enough, it can pick up solid particles and transport them further away. We will see, trough examples, that this simple set of rules is sufficient to catch the main ingredients of particle motion (saltation, creeping, suspension) under the action of a fluid.

5.1.4 Organization of the chapter

The chapter is organized as follows. To begin, we describe the set of rules governing the sediment particles. Then, we present two applications of our virtual river model. First, as a validation, we simulate the formation of scours under submarine pipelines which is a known process. Second, we use this model to understand how meanders in rivers form and evolve. In this second application, the model is, for the first time, used as a research tool. Indeed, the process is usually incorrectly explained and a simulator can be useful to verify our new explanation. Finally, a discussion closes the chapter.

5.2 The sediment model

The second important ingredient of our model are the sediments. Here, particles are represented by an integer $n(\mathbf{r}, t) \geq 0$ indicating how many pseudo-particles are present at site $\mathbf{r}$ and time t. Sediments move on the same lattice as the fluid particles and interact with them. Since $n(\mathbf{r}, t)$ can take any positive value, we term our model a multi-particle CA.

It is important to remember that in our mesoscopic approach, we do not try to represent a specific granular material. Rather, we want to capture the generic features of the erosion-deposition process. The existence of a universal behavior in systems with many interacting particles is common in many areas of science and there are numerous examples where the macroscopic behavior depends very little on the microscopic details of the system. For this reason, we may expect (and this will be confirmed by our results) that, at first approximation, our dynamics of fictitious particle produces the same deposition patterns as real systems, even if all parameters are not of the correct order of magnitude.

5.2.1 Transport rule

In this subsection, we describe the rule of motion for the sediment particles. After each time step, the particles jump to a nearest neighboring site, under the action of the local fluid flow and gravity force. Gravity is taken into account by imposing a falling speed $\mathbf{u}_{fall}$ to the particles. The suspensions are passive particles and their presence does not modify the flow field, except when they form a solid deposit (i.e. a new boundary). However, it would be quite easy to modify the fluid properties so as to make the relaxation time τ vary according to the local density of transported particles to account for the fact that the fluid viscosity depends on the concentration of the suspensions.

If the local fluid velocity at site $\mathbf{r}$ is $\mathbf{u}(\mathbf{r}, t)$, the particles located at that site will move to site $\mathbf{r} + \tau_s(\mathbf{u} + \mathbf{u}_{fall})$, where τ_s is the time unit associated with the motion of the granular particles. Unfortunately, this new location is usually not a lattice site. The solution to this problem is then to consider a stochastic motion: each of the $n(\mathbf{r}, t)$ particles jumps to a neighboring site $\mathbf{r}+\Delta t\mathbf{v}_i$ with a probability p_i proportional to the projection of $\tau_s(\mathbf{u} + \mathbf{u}_{fall})$ on the lattice direction $\Delta t\mathbf{v}_i$. The quantity τ_s is adjusted so as to maximize the probability of motion, while ensuring that the jumps are always smaller than a lattice constant. An adequate τ_s setting accelerates the particle motion as it increases the probability to move.

An example of the transport rule is presented in figure 5.2 for the hexagonal D2Q7 model (the other topologies are widely treated in [79]). In this case (see figure 5.2), three probabilities p_0, p_{i_1} and p_{i_2} must be computed using the two relations

$$\frac{\sin\alpha}{\tau_s|\mathbf{u} + \mathbf{u}_{fall}|} = \frac{\sin\beta}{p_{i_2}} = \frac{\sin\gamma}{p_{i_1}}, \quad p_0 + p_{i_1} + p_{i_2} = 1 \tag{5.1}$$

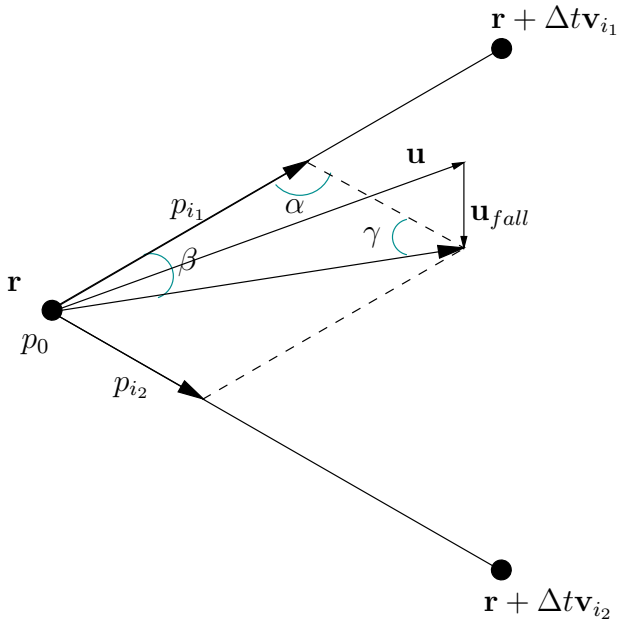


Figure 5.2: Illustration of the motion rule in case of the D2Q7 model. Three lattice sites are shown, $\mathbf{r}$, $\mathbf{r}+\Delta t\mathbf{v}_{i_1}$ and $\mathbf{r}+\Delta t\mathbf{v}_{i_2}$ which correspond to the possible destinations of the particle subject to a fluid velocity $\mathbf{u}$ and a fall velocity $\mathbf{u}_{fall}$. The corresponding probabilities p_{i_1} and p_{i_2} are obtained by the projection shown in the figure.

where α equals 120 degrees.

This stochastic cellular automata rule produces a particle motion with the correct average trajectory and a variance which can be interpreted as a local diffusive behavior [79]. Indeed, it can be shown [79] that the diffusion coefficient can be expressed as $D = D^*((\Delta\mathbf{r})^2/\Delta t)$, where $\Delta\mathbf{r}$ and Δt are the space and time spacings, respectively. The quantity D^* is a dimensionless coefficient related to the lattice topology which, in our case, is $D^* \approx 1/2$. Therefore, when reducing the lattice spacing with the usual constraint $\Delta\mathbf{r} = \Delta t$, the full diffusion constant decreases as $\mathrm{O}(\Delta\mathbf{r})$ and vanishes for fine enough grids. However, we noticed from various numerical simulations that the deposition patterns we obtain are quite robust to a change of the grid size. The noise produced by our transport rule seems necessary to initiate erosion (e.g. ripple formations) but the amplitude of this noise is much less relevant.

The transport rule can be decomposed into two phases. The first one consists in directing the sediments according to $\tau_s(\mathbf{u} + \mathbf{u}_{fall})$ while the second in propagating the sediments which is similar to the one presented for the fluid (see code 3.5). This propagation process is executed at the end of the rule application. Considering a D2Q7, the implementation of the first phase is presented in

code 5.1 where line 8 can be realized by using equation (5.1). Lines 17 to 23 can be numerically improved. Indeed, instead of drawing a random number for each sediment, one can use a binomial distributed variable representing the number of successful drawings within $n > 20$ trials each of them with a probability of success p. The procedure is detailed in [79, 80].

Code 5.1 One of the two phases of the transport rule: the sediment orientation. A D2Q7 lattice is considered.

```
void directSedimentsOfASite(const float U[2])                                  1
 // where U = τ_s(u + u_fall)                                                  2
{                                                                              3
  int p[3],i[3],j,k;                                                           4
  int numberOfSediment[nbNeighbor];                                            5
  float prob;                                                                  6
   // Computes from U motion probabilities and directions of motion            7
  p=computeMotionProbability(U);                                               8
   // Determines the direction i_1 and i_2                                     9
  i=determineDirection(U);                                                     10
   // Stores the number of sediments in each direction                         11
  for (k=0;k<nbNeighbor;k++) {                                                 12
    numberOfSediment[k]=getSedimentParticle(k);                                13
    setSedimentParticle(k,0);                                                  14
  }                                                                            15
   // Directs stochastically the sediments                                     16
  for (k=0;k<nbNeighbor;k++)                                                   17
    for (j=0;j<numberOfSediment[k];j++) {                                      18
      prob=getRandomNumber();                                                  19
      if (prob <= p0) addASedimentParticle(0);                                 20
      else if (prob <= p0+p1) addASedimentParticle(i[1]);                      21
      else addASedimentParticle(i[2]);                                         22
    }                                                                          23
}                                                                              24
```

Notice that code 5.1 proposes to store not only the number of sediments but also the directions from where they came. The main reason is to send the sediments back, if the local number of sediments is greater than the threshold N_{thres} which implies a solidification of the site. Note that this procedure may cause some sediments to indefinitely travel between two solidified sites. However, this special configuration occurs with a very low (i.e. negligible) probability. Moreover, some inertial effects can also been added when the direction is known.

5.2.2 Deposition rule

The next aspect of the particle dynamics is the deposition rule. Under the combined effect of the fluid and gravity, particles can land on a solid site (e.g. the bottom of the system or the top of the deposition layer). Motion is no longer possible and particles start piling up. In our model, up to N_{thres} particles can accumulate on a given site. The quantity N_{thres} gives a way to specify the spatial scale of the granular particles with respect to the fluid system. When this limit is reached, the site solidifies and new incoming particles pile up on the site directly above. The solid sites formed in this way represent obstacles on which an appropriate boundary condition is applied. Thus, this solidification process implies a dynamically changing boundary for the fluid. A pseudo-code of this rule is presented in code 5.2.

Code 5.2 Implementation of the deposition rule.

```
void depositSedimentsAtSite()                                          1
{                                                                      2
  for (k=0;k<nbNeighbor;k++) {                                         3
    p=getSedimentParticle(k);                                          4
    if (isThereNeighborInDirection[k]) {                               5
      if (getSolidSediment() < Nthres) addSolidSediment(p);            6
      else addSedimentParticle(oppositeOf[k],p);                       7
    }                                                                  8
  }                                                                    9
}                                                                      10
```

Note that, on the other hand, the fluid is not affected by the presence of the rest particles piling up on top of a solid site. Also, these rest particles are no longer subject to the suspension transport rule. Only the erosion mechanism discussed below can move them away.

5.2.3 Toppling rule

As sediment particles do not have infinite cohesion, it is realistic to consider the following toppling rule: when a lattice site contains an excess of δN deposited particles with respect to its left or right neighbors, toppling occurs. During this process, all unstable sites send a given portion of their grains in excess to the less occupied neighbors, see figure 5.3. With this rule, the stable configuration may not be reached in one iteration and, for this reason, the model allows the toppling and transport processes to take place at different time scales. This would imply that the toppling process would be applied a few times while the transport process would be applied once. Technically, this is achieved by applying n_t (e.g.

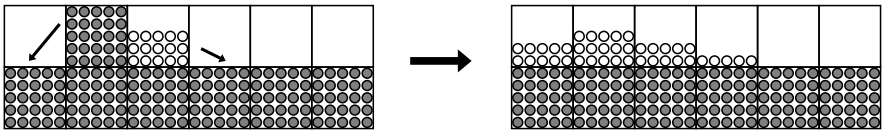


Figure 5.3: Illustration of the toppling rule with $\delta N = 5$ and $N_{thres} = 20$. Gray particles indicate a solid site.

$n_t = 10$) toppling rules only, after each n_w (e.g. $n_w = 20$) applications of the whole rules.

The quantities δN and N_{thres} give a simple way to adjust the angle of repose of the pile. In the stable state, the model tolerates a maximum difference of δN particles between two adjacent sites. Two solidified sites which differ vertically by one lattice spacing, are at least horizontally separated by k sites where $k = \lceil N_{thres}/\delta N \rceil$. Hence, the angle of repose $\alpha_{rep} \leq 45^o$ satisfies $\tan \alpha_{rep} = 1/k$ as shown in figure 5.4. Angles of repose larger than 45^o require to know more than the closest neighborhood of a site. Indeed, a large angle of repose may imply to compare sites differing vertically by a few lattice sites.

Figure 5.5 illustrates the effect of changing α_{rep} on two toppling simulations. At the initial stage ($t = 0$) one considers two piles prepared with their angle of repose (in this example, 25^o and 35^o, respectively). Then a flow from left to right is turned on and the piles have to adjust by taking into account the erosion (whose mechanism is explained below) produced by the flow. One observes that the downstream slopes keep the same angle while the upstream slopes become less steep.

Considering a 2D lattice, a pseudo-code for the toppling rule is presented in code 5.3.

In code 5.3, remark that the toppling process is a local rule. Indeed, the excess of sediments are not directly deposited on neighbor piles. They are actually directed in the direction of the piles. Hence, the application of this rule does not require communications. Moreover, erosion is favored when this local toppling process occurs. This behavior is intuitive and has been observed by the author when he played with sand for hours at the top of one of the numerous Namibian

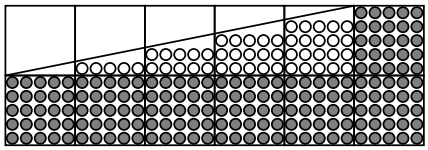


Figure 5.4: The angle of repose can be adjusted from the relation $\tan \alpha = (\delta N/N_{thres})$. Here, $\delta N = 5$ and $N_{thres} = 20$.

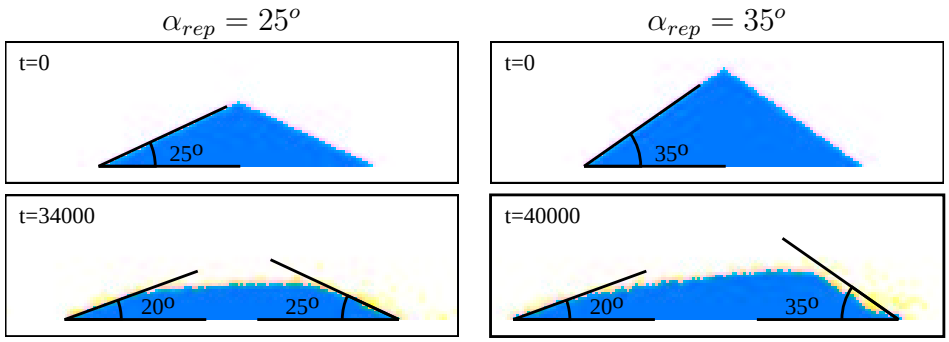


Figure 5.5: Evolution of stable piles subject to a D2Q7 flow streaming from left to right. This is a moderate turbulent flow ($Re \approx 10^4$) with $U_{max} = 0.1$ as the maximum velocity. The number of solidified sediments per site $N_{thres} = 20$. The erosion probability $p_{erosion} = 0.005$ (see below for details). The falling velocity $\mathbf{u}_{fall} = (0, -U_{max}/10)$. Two different angles of repose α_{rep} are considered. The variable t indicates the number of iterations. At $t = 0$, the flow is applied and a stationary regime is reached after a few thousand of iterations. Note that the effect of the flow is to modify the leeward slopes and also to move the piles to the right.

Code 5.3 Implementation of the toppling rule. A 2D lattice is considered.

```
void topplingAtSite()                                                    1
{                                                                        2
  int deltaRight,deltaLeft,delta;                                        3
  int deltaMax=tan(reposeAngle)*Ntresh; // reposeAngle=α_rep             4
   // Computes the height difference with right and left neighbors       5
  deltaLeft=getSolidSediment()-getSolidSedimentLeftNeighbor();           6
  deltaRight=getSolidSediment()-getSolidSedimentRightNeighbor();         7
   // Selects the maximum height. Randomly selected if equal.            8
  delta=max(deltaLeft,deltaRight);                                       9
   // Executes the toppling                                             10
  if (delta > deltaMax) {                                                11
    dif=(int) ((delta-deltaMax)*0.5);                                    12
    subSolidSediment(dif);                                               13
    addSedimentParticlesIntoMaxDirection(dif);                           14
  }                                                                      15
}                                                                        16
```

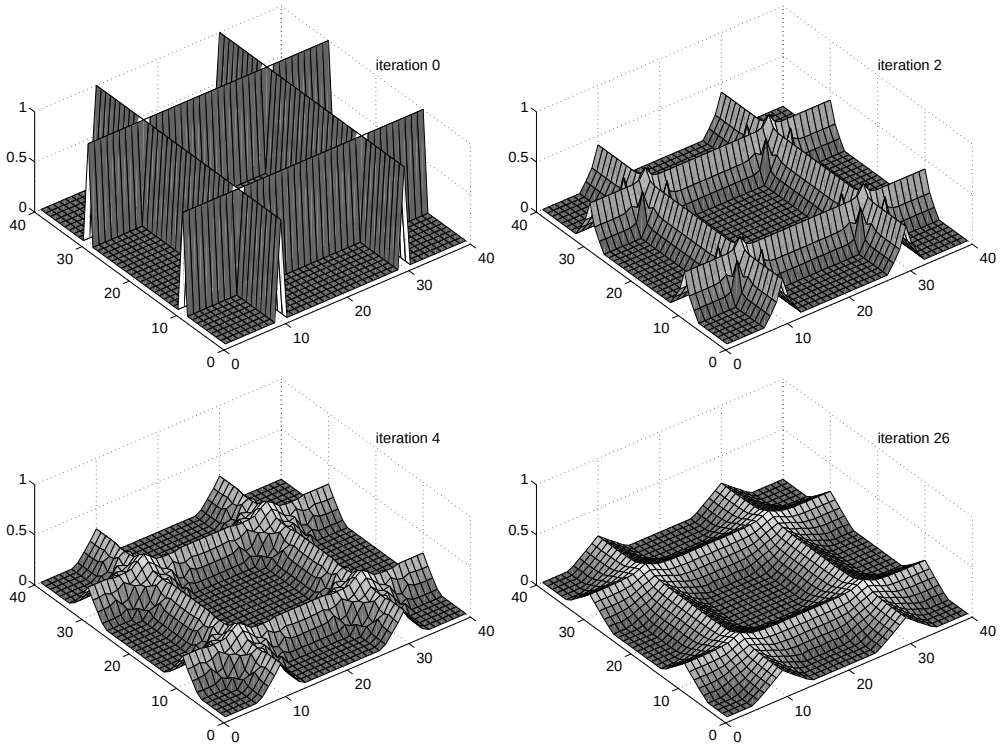


Figure 5.6: Application of the 3D toppling rule. The toric domain has 40×40 sites where each one has 8 horizontal neighbors (NW, N, NE, W, E, SW, S and SE).

sand dunes.

Let us conclude this subsection by explaining a way to apply the rule in 3D. It consists in determining the difference of solidified sediments (δ_i^{solid}) between each site and its horizontal neighbors. Let k_N be the number of neighbors where $\delta_i^{solid} > \delta N$ and δ_{max} is the maximum of the k_N δ_i^{solid} values. The toppling rule in 3D consists in sending $(\delta_{max} - \delta N)/k_N \cdot f_{top}$ solid particles on the k_N sites where f_{top} is fixed to 0.5. This choice of f_{top} is intuitive and leads to good results. However, we observe that setting $f_{top} < 0.5$ implies a slowing down of the process reaching a final state which is identical to the one obtained with $f_{top} = 0.5$. On the other hand, we notice that choosing $f_{top} > 0.5$ leads to a different and more spread final state.

This interesting subject has been treated by many papers, books and thesis. Parameters and techniques have been theoretically and practically analyzed. The interested reader can find pointers in [81].

An application of this rule is presented in figure 5.6. We observe that on a simple toric domain of 40×40 sites, a steady state is reached after 26 iterations considering an initial unstable configuration.

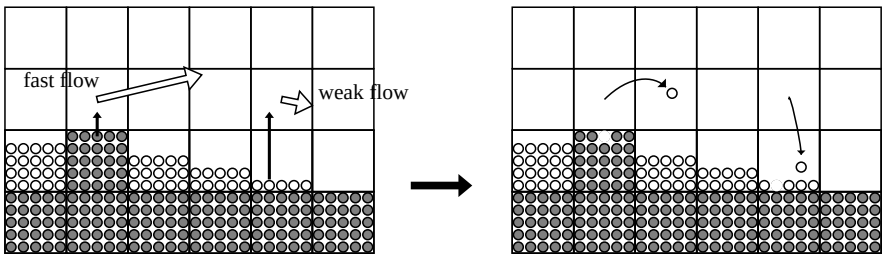


Figure 5.7: Erosion rule. Eroded particles are pushed on the upper lattice site, where they are again subject to the transport rule.

5.2.4 Erosion rule

Finally, we describe the rule implementing the erosion process. The mechanism we propose is quite simple and corresponds to making again the deposited grains available for transport: with probability $p_{erosion}$ each particle belonging to the first N_{thres} particles of the deposition layer (either a solid site or the rest particles that have accumulated directly above) is moved one lattice spacing up, in a site where the transport rule applies.

If the local fluid velocity on that site is big enough, the particle will be picked up and moved further away. Otherwise, if the flow is slow, the resulting motion will be to land again on the same site where the particle started off. An illustration is given in figure 5.7.

Code 5.4 presents a pseudo-code of this erosion rule. Again, the stochastic process which consists in drawing a random number for each particle can be improved by using a binomial distributed variable. See the above discussion for details.

This rule captures the important effect that a strong flow will result in an important erosion process. It also implements naturally the idea that erosion starts only if the local speed is larger than some threshold. One could also make the probability parameter $p_{erosion}$ depend on the amount of particles already in suspension, as it is often suggested in phenomenological models. However, in our simulations this does not turn out to be necessary and $p_{erosion}$ is a constant that is modified only from one simulation to the other, when representing different qualities of sediment.

Note that, in this model, there is no need to describe other transport mechanisms than the one defined above: creeping, saltation and suspension all naturally emerge from our rule at the macroscopic scale [79].

Code 5.4 Implementation of the erosion rule.

```
void erodeASite()
{
  // Gets the number of erodible sediments. They can be located on two
  // sites as the first N_thres sediments are subject to be eroded.
  int n=getNbErodableSediments();
  // Computes stochastically the number of sediments to erode.
  int hit=0;
  for (int i=0;i<n;i++)
    if (getRandomNumber() < perosion) hit++;
  // Operates the local erosion
  subHitSolidSediment();
  addHitSedimentInNormalDirection();
}
```

5.2.5 Note on the model parameters

Several parameters can be tuned to modify the behavior of our model. For the fluid, one can typically vary some parameters on the flow settlement process or change the viscosity (by changing τ).

The interaction between the fluid and the suspensions is determined by the falling speed $\mathbf{u}_{fall}$. This parameter is linked to the grain diameter as a large grain will fall rapidly and inversely. Similarly, the erosion probability $p_{erosion}$ is a quantity containing the effect of several parameters, such as the lift force, grain mobility and inter-particle cohesion.

The properties of the sediments are described by the angle of repose α_{rep} and the quantity N_{thres} which, as explained previously, is the amount of grain that are needed to fill a region whose height and width is one lattice spacing (in 2D).

Thus, as opposed to the conventional description, the grain diameter is not a parameter of our model. However, the quantities $\mathbf{u}_{fall}$, $p_{erosion}$ and N_{thres} are clearly linked to it.

5.3 Applications

This section presents two applications of our virtual river model. First we validate our model by simulating the formation of scours under submarine pipelines. Second, we use the model to understand how meanders in rivers form and evolve. In this second application, the model is, for the first time, used as a research tool. Indeed, the process is usually incorrectly explained and a simulator can be useful to verify our new explanation.

5.3.1 Scours under submarine pipelines

Motivation

Marine structures are vulnerable to the erosion of the sediments around their base due to the scouring action of currents (e.g. waves or steady). The scour thus formed can jeopardize the integrity of the structure and must be accounted for at the design stage.

The large number of structures, flows and models have been the subject of many papers and books. Such a structure has been chosen as an example in figure 5.1. Two books reviewing the subject have been written, one by Whitehouse [6] and the other by Hoffmans et al. [82].

In this subsection, we focus our attention on scours under submarine pipelines in a steady current. Such pipelines are commonly used to transport oil, gas or even water on the sea bed. At installation time, they rest on an erodible bed. The current around the pipe causes the formation of an excavation below the pipe, called a scour. The scour can typically reach a depth approximately equal to the pipe diameter. At first sight, these scours can damage the pipe due to the vibration induced by the non-uniformity of the bed. However considering that an important part of the installation cost is devoted to the pipe protection from environmental changes, current or even anchoring, these scours can be seen as an opportunity to self-bury the pipe.

Given that the deeper the scour the better protection, one could be interested to enhance the depth of the scour. For that a spoiler can be attached onto the pipe in order to modify its cross section and thus stimulate the fluid around the pipe. Laboratory experiments [83] have investigated the influence of the orientation and size of the spoiler. Essentially, these show that a vertical spoiler measuring the half of the pipe diameter produces the best enhancement approximatively equal to 50% of the scour depth.

Before seeing that our model can successfully address the problem of the scour formation under submarine pipeline in a steady current, let us have a look at the phenomenology of the process.

Phenomenology

This subsection reviews some phenomenological features of the scour formation process under a pipe [82, 6]. Figure 5.8 presents the situation of interest. The scour which forms under the pipe occurs in two main phases, namely the onset and the erosion. The onset of the scour is directly related to the two vortices in front of and behind the pipe. Each one digs up a hole. The scour formation process breaks out when these holes meet together under the pipe. The onset process is illustrated in figure 5.8.

When the water has started to flow underneath the pipe, the scour formation stage takes place. First, the gap under the pipe is small while the downstream hill

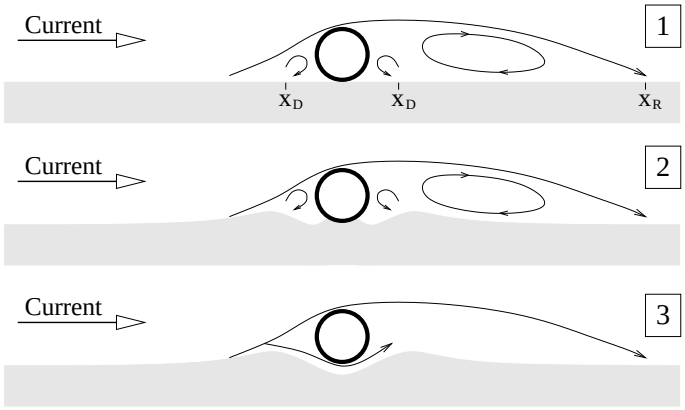


Figure 5.8: Three main stages of the scour onset process under a unidirectional current. (1) Due to the current, three main vortices appear in the pipe neighborhood. (2) The two small vortices up and downstream the pipe start to dig a hole. (3) After a while, the holes meet each other below the pipe and the scour formation process breaks out. x_D and x_R denote re-attachment points.

is relatively high. At this point, the pressure upstream the pipe is rather high and consequently the fluid is accelerated. Many sediment particles are ejected and the scour development is fast. Then, as the scour depth increases, the velocity under the pipe decreases. It follows that the scour development slows down and progressively reaches an equilibrium.

Various laboratory experiments (see for example [75]) show that the depth of the scour depends essentially on the pipe diameter D, the flow velocity $\mathbf{U}$, the kinematic viscosity of the fluid, and the sediment properties. These studies also highlight that the ratio between the scour depth and the pipe diameter should be comprised between 0.2 and 1.0.

Also, it turns out that the scour formation process is not too much influenced, in an appropriate range, by turbulence. Indeed for Reynolds numbers between 10^4 and 2×10^5, Kjeldsen et al. [75], as well as Bijker et al. [84] propose formulae indicating that the scour depth is strongly controlled by the pipe diameter. Following the same idea, Sumer and Fredsøe [85] propose an empirical formula $S/D = 0.6 \pm 0.1$ indicating the dominant role of the pipe diameter.

Existing models

The scour formation process has been studied in flumes by various authors [74, 75, 5]. Numerical models have been proposed in [76, 77, 78]. They are based on an iterative process which consists in computing the velocity repartition by a finite difference scheme and then determining the change of the bed. They essentially reproduce the experimental results quite well.

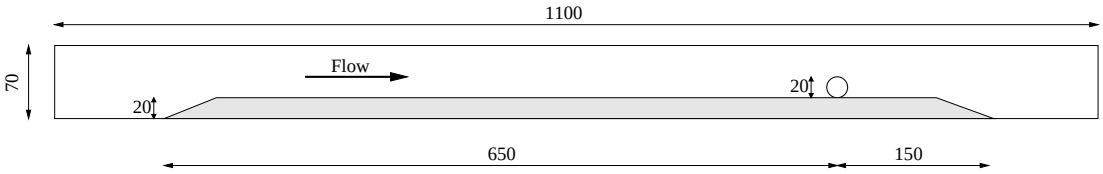


Figure 5.9: Geometry of the numerical experiment. White and gray areas are fluid and sediment areas respectively. Distances are expressed in lattice sites.

Results

We present here the predictions of our model for the scour formation process in a steady current. We consider a virtual river made up of a layer of fluid on top of an initially flat bed of sediments with uniform properties. A pipeline is laid on the top of the bed, as illustrated in figure 5.9.

Due to the geometry of the problem, a two-dimensional simulation is considered on a D2Q7 lattice. The x-axis corresponds to the direction perpendicular to the pipe and the z-axis to the vertical direction. The system size is $L_x = 1100$ and $L_z = 70$, in lattice units. The bottom line ($z = 0$) is an impermeable wall on which sand grains deposit and fluid particles bounce back if they ever reach this region. For the other boundaries of the system, the conditions change whether one considers a fluid or a sand particle.

For the fluid, the system is periodic along the x-axis. All fluid particles that reach the $x = L_x$ line are re-injected on the left side and vice versa. On the line $x = 0$, the fluid is accelerated uniformly so as to produce a water current flowing from left to right, with entry speed $U_{entry} = 0.1$ (in lattice units). The viscosity is tuned so as to obtain a Reynolds number $Re \approx 10^4$, which is a typical value in scouring experiments [75, 76]. On the upper line ($z = L_z$) a zero vertical velocity is imposed. Note that these boundary conditions are implemented so that the total amount of fluid in the simulation remains constant

A typical flow obtained around a pipe is presented in figure 5.10 where the stationary average streamlines and average horizontal velocity profile are plotted. This simulation compares well with the experiment by Jensen [86]. In addition, this flow is generic (with respect to the size of the main eddies and re-attachment points) of the situations considered in laboratory experiments when studying the scour formation process [75, 76].

For the sediments, the parameters we have chosen for the simulation are summarized in table 5.1. The falling speed is given in lattice units and the other parameters are pure numbers. The values of these parameters are chosen empirically, so as to reproduce the experiments presented by Mao [87]. However, other values can be considered and other erosion patterns could emerge.

As shown in figure 5.9, the setting of our numerical simulation corresponds

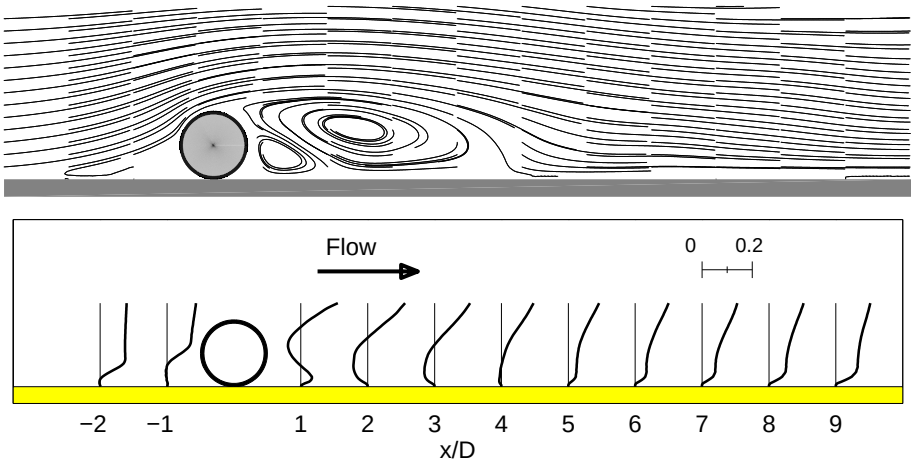


Figure 5.10: Average flow pattern around a pipe in a turbulent regime, as obtained with the LB method on a D2Q7 lattice. The Reynolds number $Re = 10^4$. The Smagorinsky subgrid model is used with a constant $C_{smago} = 0.4$. The flow is settled by imposing a velocity $U_{entry} = 0.1$ at the inlet, and the cylinder diameter $D = 20$. The upper panel shows the streamlines and the lower one the horizontal velocity profile at several locations.

precisely to the experiment done by Mao [87]. The simulation starts with an erodible bed and all grains that reach the right (and possibly the top) of the system are lost.

u_{fall}	N_{thres}	p_{erosion}	α_{rep}
0.006	10	0.01	$20^o - 40^o$

Table 5.1: Values of the model parameters for the sediments.

Since the toppling and transport processes are not taking place at the same time scale, we empirically choose to make 20 steps of pure toppling every 20 iterations of the dynamics.

The evolution of the bed profile predicted by our model is shown in figure 5.11, at four typical stages of the scour formation. The profile at $t = 200\ 000$ corresponds to a steady state. It compares well with the experimental profiles found by Mao [87] (white and black circles in the figure). Also, we observe the asymmetry of the scour which, as expected, is steeper on the left part than on the right part. This gives a good validation of the stationary properties of our model.

Note that the precise value of the repose angle α_{rep} is not critical as long as it is larger than the slopes of the eroded bed (i.e. 14^o and 19^o). It is admitted that $\alpha_{rep} = 20^o$ is too small to describe the sediments properties in Mao's experiment,

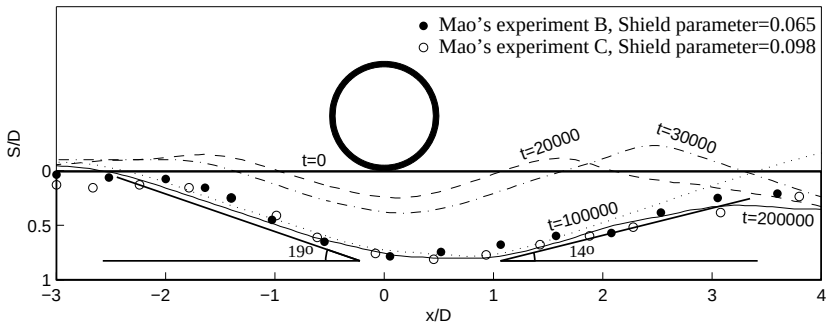


Figure 5.11: Characteristic stages of the scour evolution process where S denotes the scour depth, D the pipe diameter and x the horizontal distance from the center of the pipe. Erosion profiles are presented at several time steps. They are in good agreement with the experimental findings shown as the black and white circles.

but we have observed that a larger value of α_{rep} gives the same erosion pattern too. This is not surprising since the erosion slopes result from the effect of the current and not from the repose angle.

Our model also gives a dynamical description of the scour formation (figure 5.11 shows the erosion profile at several time steps). In particular, we can observe the protuberance which forms past the pipe and travels downstream, as in real experiments.

In order to have a quantitative validation of the time development, we compare the scour depth as a function of time obtained from our simulation, with the phenomenological relation proposed by Sumer [85]

$$S_t = S(1 - e^{-t/T}) \tag{5.2}$$

describing how the final profile is reached where S_t denotes the scour depth at time t, S the final equilibrium value and T is the characteristic time scale of the scour process representing the time period during which a substantial scour develops.

In our simulation we have recorded the scour depth for several time steps. Results are shown in figure 5.12, as white circles. An exponential fit (solid line) shows a very good agreement with the behavior described by (5.2), thus indicating that the dynamic properties are correctly captured by our approach. Here we obtain a characteristic time $T \approx 29700$ iterations. This quantity depends on the values chosen for the parameters such as $\mathbf{u}_{fall}$, N_{thres}, $p_{erosion}$. If one modifies them in such a way as to model a larger grain diameter d, one also measures an increase of T. This behavior is in agreement with phenomenological observations [6].

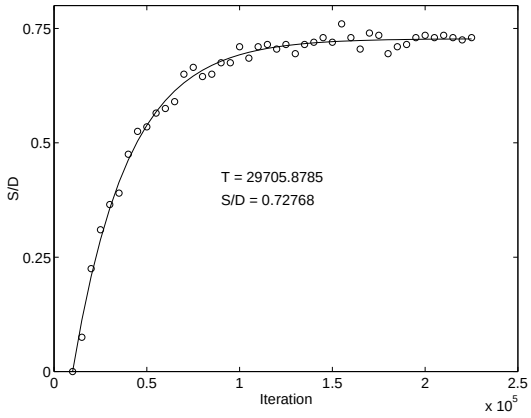


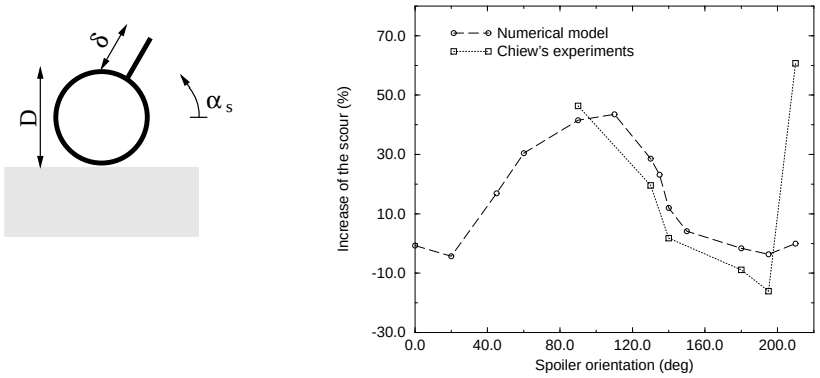
Figure 5.12: Time evolution of the scour depth. Simulated depth are displayed as circles, while the solid line is the best fit from equation (5.2). The offset at $S_t = 0$ indicates that the origin of time does not correspond to the beginning of the erosion process.

Finally, let us consider the case of a pipe with a spoiler. As mentioned previously, a submarine pipeline is more protected against possible damages when it is buried in the sea bed. The hole resulting from the scour process can be used to self-bury the pipe. Following this idea, spoilers of various angles can be attached to the pipe in order to speed up and increase the scour formation process [83]. As recommended in [83], the length of the spoiler δ is set to $D/2$ where D is the pipe diameter. Figure 5.13 presents the simulation results where various orientations of the spoiler α_s have been considered. We observe a good agreement with the experimental measurements [83]. We also predict the increase of the scour for $\alpha_s < 90^o$. However, we do not catch the special case which consists in placing the spoiler against the current and lightly touching the ground. This is probably due to a discretization effect of our cellular automata.

5.3.2 Meandering rivers

In this subsection, we focus our attention on the process of meandering[1] rivers. Before starting with the phenomenology of the process, let us mention the famous paper by Leopold et al. [88], which gives many references and explanations on meandering rivers, and also the recent thesis of Lancaster [89] giving an up to date review.

[1]The word meander comes from an Asian river called *Maiandrose* which has probably disappeared.



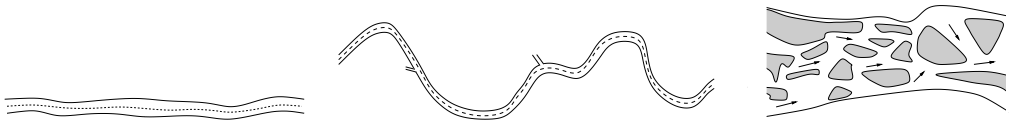
a) b)

Figure 5.13: a) Submarine pipeline with a spoiler. The spoiler angle is denoted by α_s, the pipe diameter is D and the spoiler length is δ. b) Increase of the scour depth versus the spoiler angle α_s. Results from our numerical model are drawn as circles and experiments of Chiew [83] as squares.

Phenomenology

Flowing rivers may be classified into 3 categories illustrated in figure 5.14: straight, meandering and braided. These categories are essentially related to the flow velocity and to the characteristics of the banks. Note that the velocity is linked to the slope of the bed which varies, and so do the characteristics of the banks, along the path of the river. It implies that a river, from beginning to end, may follow any of the three previously introduced patterns along its path. However, it is unusual to observe straight rivers for a distance exceeding 10 channel widths [90].

To determine the impact of the flow velocity, Schumm et al. [91] studied in laboratory the transition between straight, to meandering and finally, to braided rivers by modifying the slope of the flume (i.e. the slope of the bed). They emphasized the flume gradient as the controlling variable. But it would seem reasonable to consider other parameters such as bank characteristics.



(a) (b) (c)

Figure 5.14: Different types of flowing rivers: (a) straight, (b) meandering and (c) braided river.

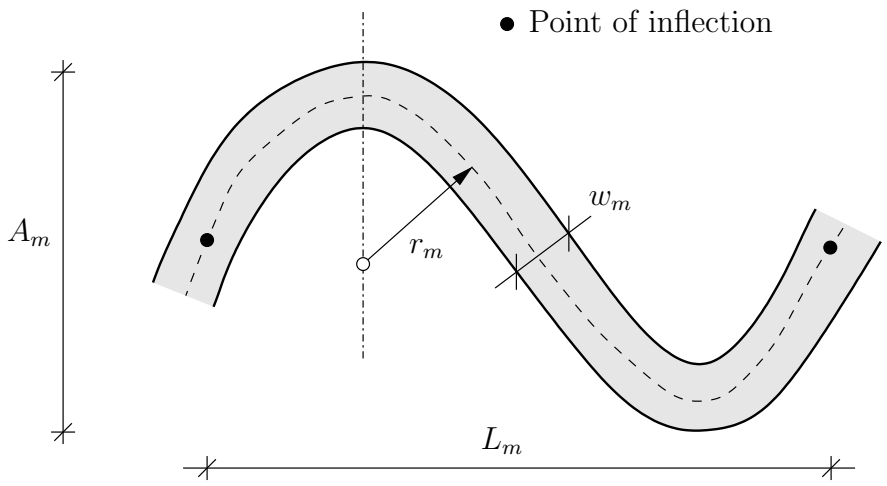


Figure 5.15: Geometric characteristics of a bend.

Notice that meandering rivers are also important because abandoned channels (i.e. oxbow lakes, see below for details) of the main river will silt up. Over thousands of years these silted channels are compressed and form shale beds between which oil is often found [92].

The geometric characteristics of a bend are sketched in figure 5.15. It is amazing to observe that most river meanders have nearly the same ratio: $L_m/w_m \approx 12$, $L_m/r_m \approx 4$ and $A_m/w_m \approx 5$ where L_m and A_m are respectively the length and the amplitude of a meander and r_m is the bend radius. To illustrate these ratios, geometric characteristics of real rivers such as the huge Mississippi (~ 2.2 kilometers large) and of laboratory flumes (~ 1 meter large) are reported and fitted by lines on the graphs of figure 5.16. These data were the subject of an appendix in [88] and of similar graphs. We observe the expected ratios and a low dispersion around the lines. Leopold et al. [88] measured meanders created in glaciers which are sediment-free channels. The ratio L_m/w_m, reported by a square in figure 5.16, seems to follow the same river behavior. They argued that shapes of curves in rivers are determined primarily by the dynamics of the flow rather than by relation to debris load.

At first sight, one may consider the meandering process as a 2D phenomenon. The shape of the river bed and the velocity profiles of the flow actually clearly indicate that this is a 3D process. Indeed, although the bed profile is approximately symmetric around the inflection point, one observes that the depth of the river is higher on the outer than on the inner part of a bend. Moreover, Leliavsky [93] showed that the depth behaves like the inverse of the bend radius.

The velocity profile is also more complicated than a 2D profile for several reasons. First, contrarily to common ideas, one measures at the surface and at the bed level a higher longitudinal velocity at the inner part rather than at the

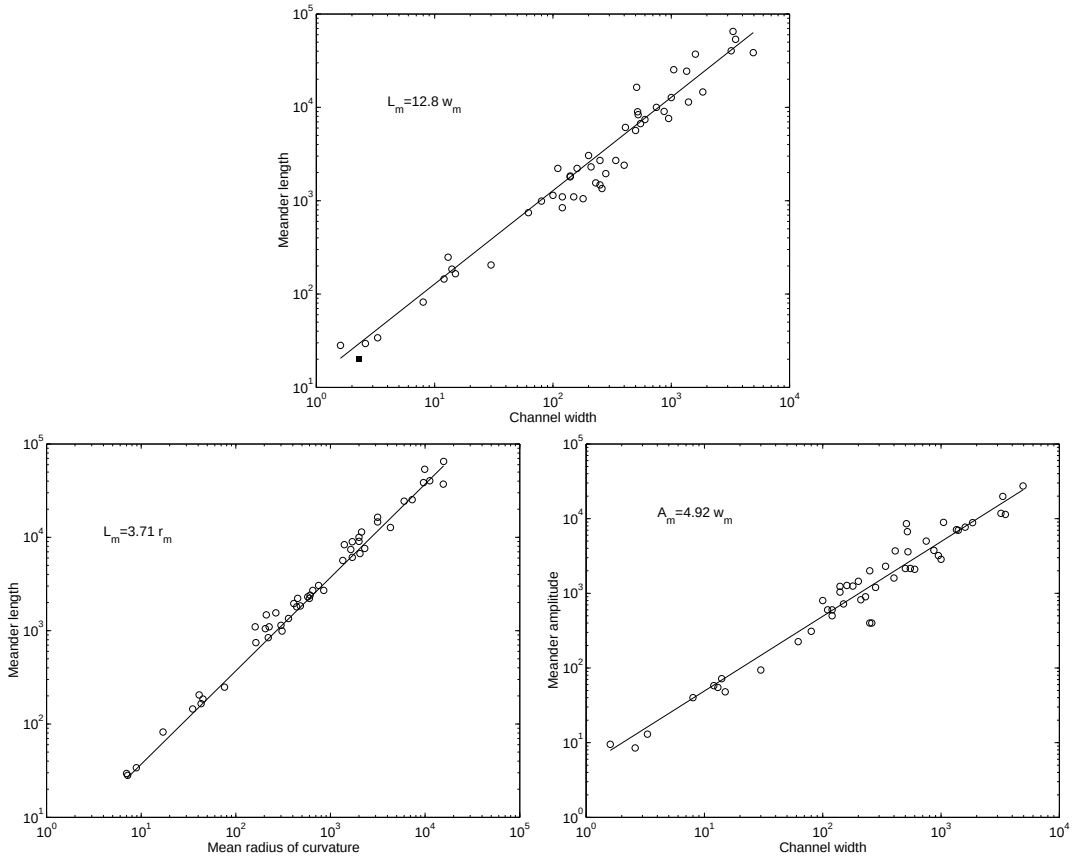


Figure 5.16: Geometric characteristics of real rivers such as the huge Mississippi (~ 2.2 kilometers large) and of laboratory flumes (~ 1 meter large) are reported and fitted by lines. The ratio are reported on each graph. The filled square drawn on the top graph is the ratio L_m/w_m of meanders created in a glacier.

outer part. Second, a crucial secondary flow which is the velocity projection on a transverse plan, making up a kind of helix, is initiated by the sinuous 3D shape of the bed [94]. The river bed and the flow velocity are sketched in figure 5.17.

Erosion processes occurring in rivers imply that river beds dynamically change. Basically, sediments are eroded in the outer part and deposited in the inner part of a bend making up the so-called pointbars. Consequently, the bend progresses laterally as sediments are eroded in the outer part and deposited in the inner part. Moreover, Matthes [95] observed that channels also move downstream and not only laterally.

The evolution of a bend is not infinite. Indeed, its progression is stopped when both extremities of the loop finally get connected. The flow choosing the shortest way, makes a shortcut by not using the bend anymore. This is the so-called cutoff process. The fluid is consequently at rest in the loop. The bend then forms a

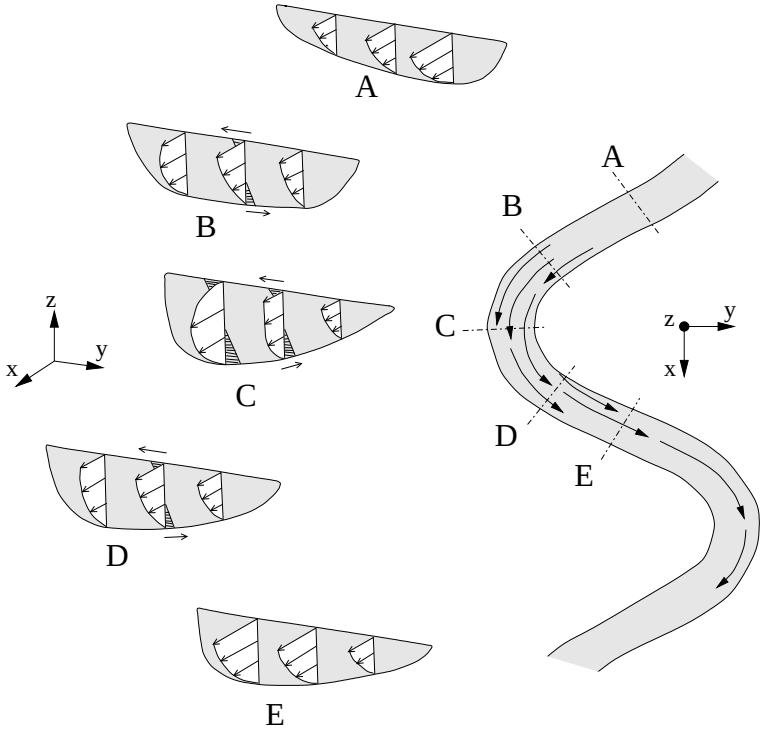


Figure 5.17: Sketch of the river bed and the flow velocity. The sketch is reproduced from [88]. The left hand side holds some cross sections indicated on the right hand side by mixed lines.

lake which is called oxbow lake[2]. An oxbow lake formation is illustrated in figure 5.18.

To finish this subsection devoted to the phenomenology of the meandering rivers, let us have a look at some real flowing rivers presented and commented in figure 5.19.

State of the art

The approaches that attempt to explain both the geometry and dynamics of meandering rivers, are very diverse. The aim of this subsection is not to give an exhaustive review but rather to give some pointers to the main characteristic publications.

Chang et al. [96] proposed a model based on the minimum energy dissipation rate hypothesis. Thakur et al. [97] used models based on the most probable path assumption. Ikeda et al. [98] as well as many others after them [99, 100] made use

[2]This word comes from the U-shaped frame forming a collar about an ox's neck and supporting the yoke.

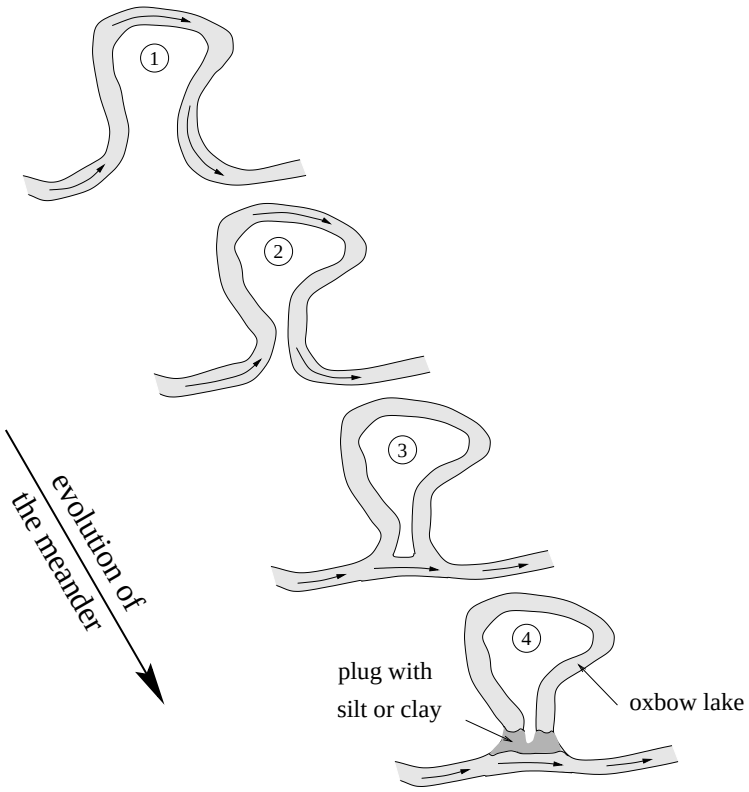


Figure 5.18: Evolution of a bend until the creation of an oxbow lake.

of erosion models obtained by studying the instabilities of the flow in channels using a perturbation method. Ferguson et al. [101] proposed a statistical model which generates meandering patterns that look realistic at large scales. Liverpool et al. [102] proposed a statistical model motivated by the physical nonlinear dynamics of the river channel migration and by describing heterogeneity of the terrain by noise. Dodds et al. [103] studied the scaling laws of river networks. Sellin [104] identified in laboratory the flow mechanisms in channels of complex geometry. Murray et al. [105] successfully described braided rivers by using cellular automata. Howard et al. [106, 107] proposed some sufficient conditions to numerically simulate river meandering. Recently (2001), Lancaster et al. [108] developed a simple model of river meandering.

Despite all these efforts, we are not aware of any numerical simulation of the river meandering process with a model where the full hydrodynamic flow, as well as the erosion-deposition phenomena, are all taken into account. In addition, no final theory seems to exist in order to explain the mechanisms responsible for the meandering.

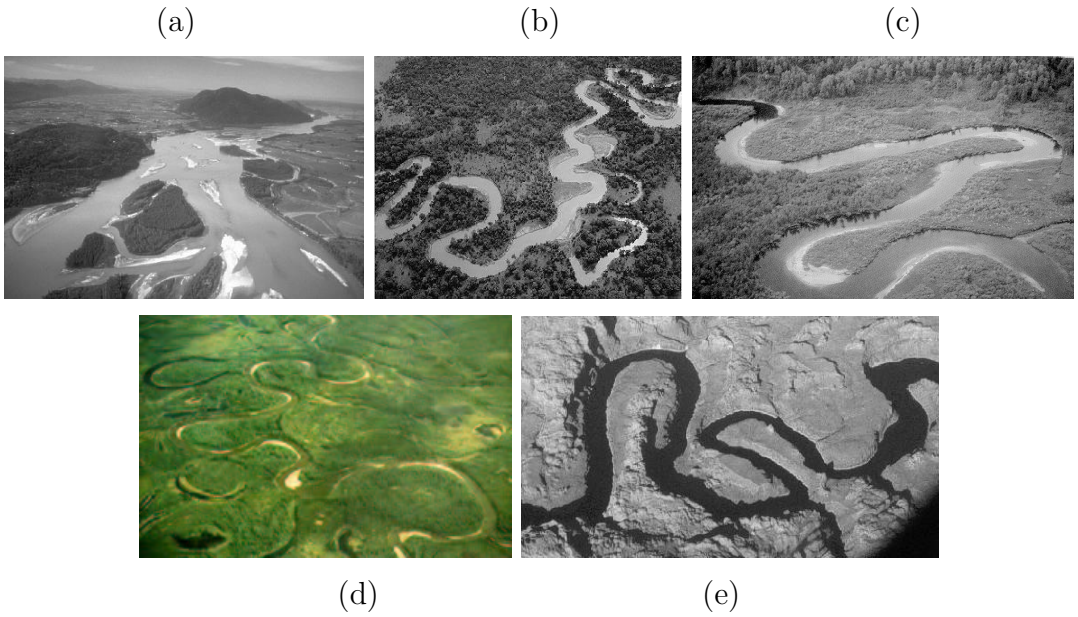


Figure 5.19: Illustration of real flowing rivers. (a) A braided river. (b) Cutoffs and their oxbow lakes. (c) Deposits on the inner part of bends making up the so-called pointbars. (d) and (e) Meandering rivers in two different kinds of vegetation.

Meander dynamics at mesoscopic scale

At a mesoscopic scale, we focus our attention on the processes responsible for the start of the meandering process and its progression. Some ingredients explained below can probably be found in the literature. But we did not see until now any explanation putting them all together at this specific scale.

It is observed, and confirmed by our model, that the secondary flow in straight channel is made of two rolls, as sketched in figure 5.20.

As long as this secondary flow contains two rolls with opposite rotation, the erosion is symmetrical on both sides of the river. But both rolls can merge into one larger roll, which is a more stable hydrodynamical pattern, when the channel is not perfectly straight. In this case, the erosion-deposition process acts

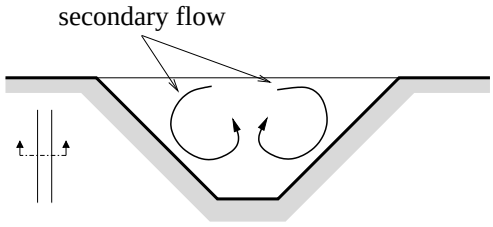


Figure 5.20: Sketch of the secondary flow in a straight channel.

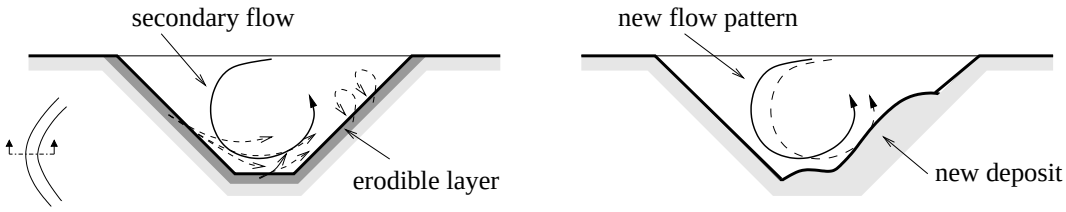


Figure 5.21: Asymmetrical secondary flow in a curved channel. The erosion (dashed curved arrows) on the outer part is favored due to the combined effect of the secondary flow (solid curved arrows) and gravity. New patterns result from the process.

differently on each side by eroding on the outer part and by depositing on the inner part of a bend. We argue that this asymmetrical erosion-deposition process is due to the combined effect of the secondary flow and gravity. This process is illustrated in figure 5.21.

Hence, figure 5.21 illustrates how this asymmetry results in the displacement of the river bed and, progressively, causes the meandering to occur.

Preliminary results

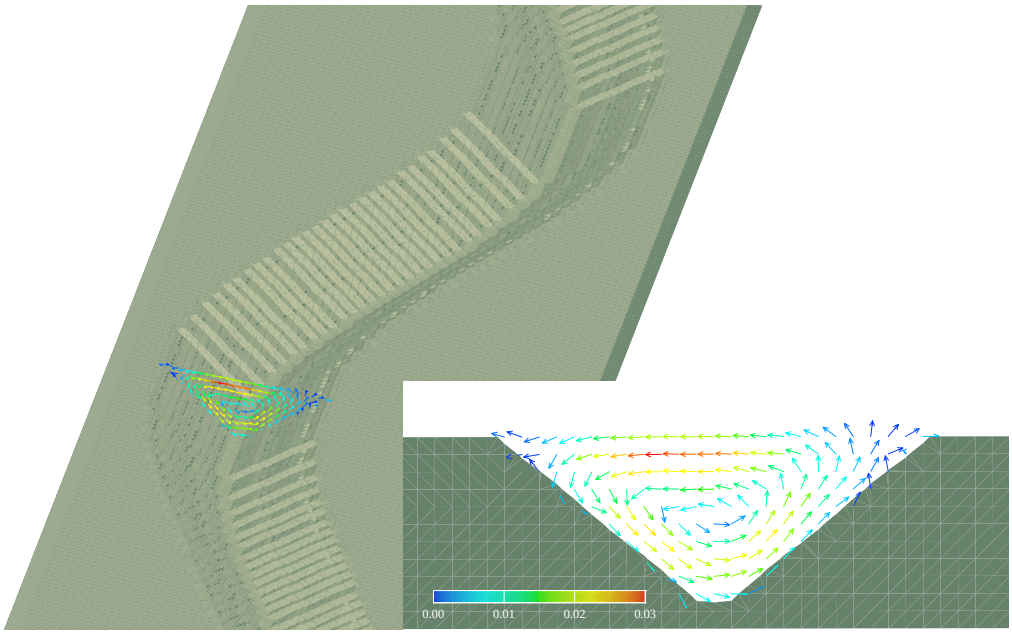
This subsection presents some preliminary results obtained with our numerical model on a x-periodic D3Q19 lattice of size $N_x \times N_y \times N_z = 250 \times 70 \times 12$. The flow is settled by imposing a constant velocity $U_{entry} = 0.1$ at the inlet. The fluid bounces back on the deposit and a bounce forward is defined at the top of the lattice ($z = N_z - 1$) in order to simulate a free surface. To produce a flow sufficiently turbulent, the relaxation time τ is set to 0.50001. The Smagorinsky subgrid model is used with a constant $C_{smago} = 0.1$.

A pile of sediments at rest starting at $x = 30$ and ending at $x = 220$ is initially set. Except for an initial and virtual sinusoidal river of depth 10, sediments are set everywhere. The virtual river forms a V-shape with a maximum width equal to 25 sites. Sediments reaching the outlet or the top of the domain are removed. The erosion probability $p_{erosion} = 0.01$, the quantity $N_{thres} = 50$, the falling velocity $\mathbf{u}_{fall} = (0, 0, -U_{entry} \cdot 0.05)$ and the angle of repose $\alpha_{rep} = 40^o$.

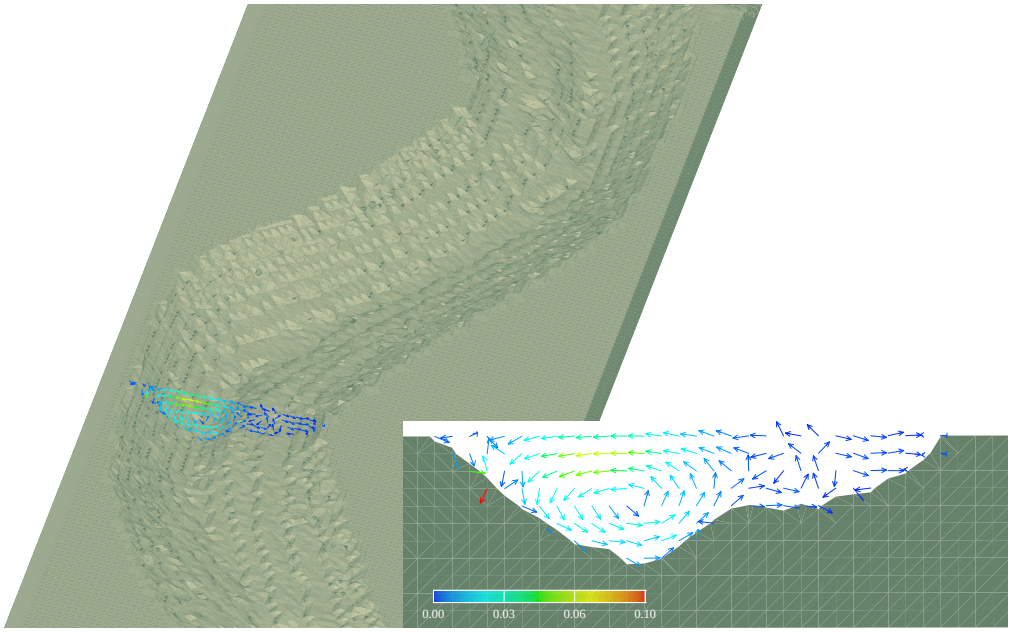
Hence starting from a sinusoidal channel, the simulation results are presented in figure 5.22. They took 40 minutes on our 32 pentium cluster.

In conformity with the previous theory, we observe the expected progression of the bend which moves, in this case, to the right. The secondary flow reported in figure 5.22 is also in agreement with the theory.

However, more work is necessary to simulate the whole phenomena and to produce some cutoffs and oxbow lakes. Ongoing work showing that meanders move laterally but also downstream are in progress and will be reported as soon as possible.



$t = 0$



$t = 10\ 000$

Figure 5.22: Preliminary simulation results of a meander evolution. The simulation starts at $t = 0$, by considering a sinusoidal meander. The arrows represent the secondary flow. After 10 000 iterations, the meander has progressed and has formed a new secondary flow which is shifted.

5.4 Summary

In this chapter, we presented our virtual river model. It is composed of a LB fluid and of a sediment cellular automata. The rules of the latter are explained in detail.

The scour formation under submarine pipelines is used to validate our model. We successfully reproduce the results found in literature. Next, we employed our model to deal with the complicated meandering process. We gave clear explanations of the process occurring at a mesoscopic scale. They were confirmed by numerical simulations.

However, more work is necessary to obtain a quantitative validation of the above theory. Some further work could be to generate fully developed meanders, and check the phenomenological laws concerning the wave length and amplitude of the pattern as a function of the river parameters (see figure 5.16).

Chapter 6

Implementation of lattice Boltzmann models

6.1 Introduction

In the lattice Boltzmann (LB) community, performance issues are given more attention than software engineering concepts such as reusability or clarity. Indeed, adding too many software layers in the computer implementation can strongly affect the performances and is consequently not adapted. On the other hand, in order to produce a general environment for lattice based models, some reusability is highly desirable. An application can be developed in this environment by defining some components and by reusing some others such as lattice topology or parallel execution methods.

To illustrate these two trends, see for example [11, 109] and [110] which are two implementations of a lattice Boltzmann model simulating radio wave propagation. Each one has its own quality. The first one is efficiently coded while the second one is much more reusable and elegant, but rather slow.

Due to the success of the LB approach and the large range of possible applications, there is a growing demand for an efficient and reusable tool, saving developing time without degrading too much CPU performances. An interesting attempt is made in [111] for cellular automata. A working environment and a specific language that extends C are provided. This system is however too general to ensure optimal performance for lattice based models.

In this thesis, we propose an object oriented approach to implement a parallel environment for applications using a lattice such as LB models. This environment is called PELABS (**P**arallel **E**nvironment for **LA**ttice **B**ased **S**imulations). Among others, some lattice topologies and some parallel execution methods are proposed in PELABS. These components are consequently shared by every applications and implemented once. To deal with a parallel execution, it uses a graph partitioning technique to decompose the domain across several processors. This

technique allows us to treat efficiently any irregular domain. We also show that, when the domain is sparse enough, our approach is faster than the direct implementation which extends the computation to a rectangular domain and masks the sites that should not be present.

The chapter is organized as follows. First, we propose a short review of some computer science concepts used in the chapter. Then we explain the main mechanisms to parallelize LB models. Next, we present our new parallel environment PELABS and we analyze its parallel performances. Finally, we focus our attention on cache memory by optimizing its use. A discussion closes the chapter.

6.2 Some reviews of computer programming

6.2.1 Programming language evolution

Computer science has been for a long time motivated by the needs of the physicists. Some languages have been created to satisfy their numerical requirements (essentially the Fortran family). On the other hand, some pure computer scientists have directed their researches towards the important question: can we improve the software quality? Their various answers have been the starting point of many languages (SmallTalk, Eiffel, Ada, C++). These languages have in common the concept of object-oriented (OO) concepts which is introduced in subsection 6.2.2. Figure 6.1 illustrates the evolution of programming languages.

We observe in figure 6.1 that the OO concepts introduced in 1969, are now extensively used. However, they are abundantly criticized by performance unconditionals. Indeed, they argue that the use of object-oriented languages dramatically slows down any application. We will see below that the computational time of our OO implementation divided by the computational time of a typical procedural implementation is equal to 2.

After many years, both branches finally seem to merge. Indeed, the Fortran community has taken the turn by proposing an OO version: Fortran 2000. This language is, at writing time, only a specification. However, it seems promising as it deals with performances and with code quality.

Recall that in this work, our aim is to produce an efficient and reusable environment. Hence, we are looking for a language which is OO and efficient. Its large distribution would also be an advantage. Our choice was naturally set on C++ [113]. However, in C++, managing pointers and memory is cumbersome.

6.2.2 Object oriented programming

This subsection is devoted to explaining the salient features of object-oriented (OO) programming. The reader interested in details may consult [114, 115].

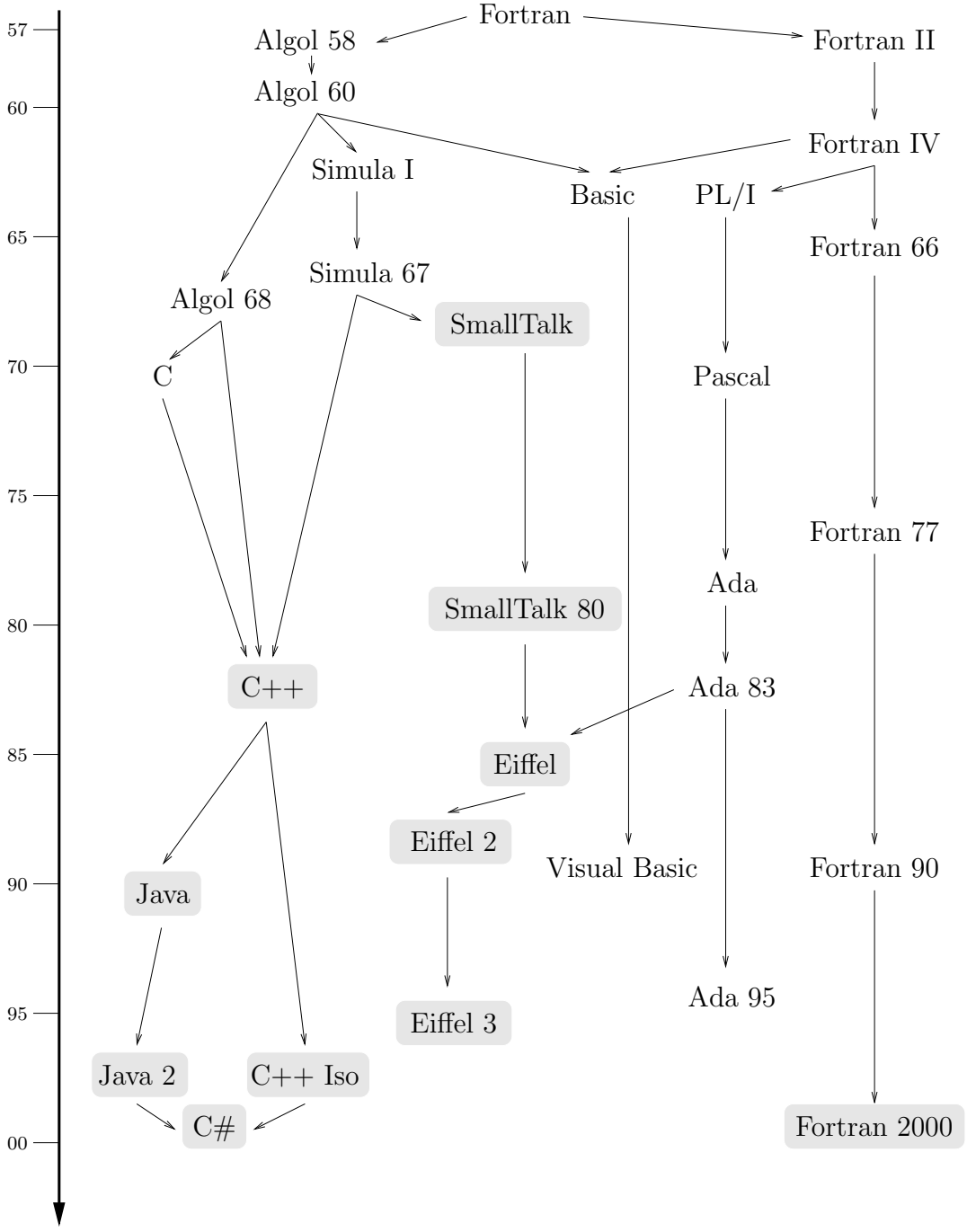


Figure 6.1: A simplified diagram of the evolution of programming languages. Languages surrounded by a gray box are object-oriented. The vertical arrow represents the elapsed years. A very complete evolution can be found in [112].

OO programming is a type of programming in which coders define not only the data type of a data structure, but also the types of operations (methods) that can be applied to the data structure. In this way, the data structure becomes a class that includes both data (attributes) and methods. In addition, programmers can create relationships between classes. For example, classes can inherit characteristics from other classes. An instance of a given class is called an object.

One of the principal advantages of OO programming techniques over procedural programming techniques is that they enable programmers to create modules that do not need to be changed when a modification is added. A programmer can simply create a new object that inherits many of its features from existing objects. This makes OO programs easier to modify and to maintain.

To perform OO programming, one needs an OO programming language. Java, Smalltalk and C++ are three of the most popular languages. As explained in the previous subsection, we will, in the context of this thesis, use C++ essentially for its good numerical performances.

Before having a precise definition, let us have a look at an example illustrating some important OO concepts. In this example, we are interested to define a part time worker which works the morning as a teacher and the afternoon as a vine grower. This worker is thus a teacher and a vine grower with their own specific activities and salaries. However, both of them share some characteristics such as their name, age or even number of children. This entity can be called a person. An OO architecture is presented in figure 6.2 where a `PartTimeWorker` class inherits from `Teacher` and `VineGrower` classes. The two latter inherit from the `Person` class. Methods such as `getNbStudent()` or `getNbVineyard()` has been inherited in `PartTimeWorker`. Notice that `getSalary()` has been overloaded with a new appropriate definition.

The simple example of figure 6.2 proposes an illustration of multiple inheritance which is not allowed in some languages (Java for example). It also shows that a class modification is automatically passed on to its sub-classes.

More formally, Meyer who is the creator of Eiffel and worked in OO concepts for long time, proposed [114] seven conditions[1] necessary for a language to be OO. We add some comments(*in italic*) to their original formulations:

1. *The application is decomposed into modules according to their data structure.* The data is the main component imposing the structure of the code.

2. *The objects have to be described as implementation of abstract data types.* Many languages, dealing only with modules, offer enough functionalities to manage the first two points. Ada, Modula-2 or Fortran 90 are good examples.

3. *Unused objects have to be deallocated by the system with no intervention of the programmer.* This condition is rarely satisfied with common OO lan-

[1] originally called *7 steps to the objects happiness.*

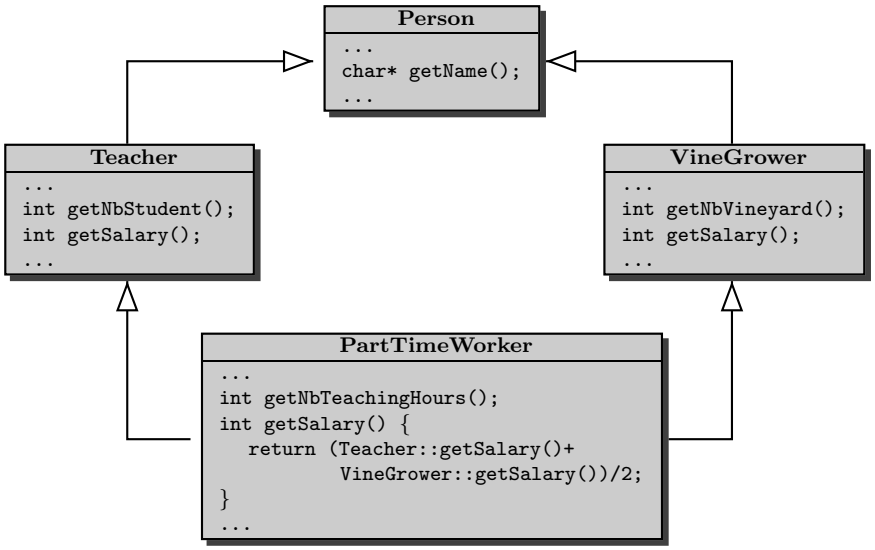


Figure 6.2: Class decomposition example where the main goal is to give a description of a part time worker which is a teacher and a vine grower. The arrows mean *inherits from*. To save space, only one representative method has been reported.

guages (Java or Eiffel manage it any way). We think that this is not a necessary condition as the programmer can do this properly and systematically.

4. *Each type is a module and each module is a type.* This is the definition of classes which are composed of a set of attributes managed with some methods.

5. *A class can be defined as an extension or a restriction of an other class.* This defines the basic principle of inheritance. The methods inherited should be overloaded if necessary.

6. *A program entity should be able to refer to objects of various classes (polymorphism) and operations should be able to have different versions for different classes (overloading).* The notion of polymorphism means that a class `B` inheriting from `A` is of type `B` but also `A`. It implies that a variable of type `A` can be set with a variable of class `B`. The opposite is not true. Notice that the methods defined only in `B` are obviously not accessible.

7. *A class should be able to inherit from many others.* The multiple inheritance is allowed.

A last ingredient, which in Meyer's point of view is not crucial, is the genericity. This allows to define classes composed of unknown types. For example,

suppose that one defines the template class `List<CellType>` which features a list composed of elements of type `CellType`. Hence, a list of `PartTimeWorker` can be realized by instantiating a list of type `List<PartTimeWorker>`.

In summary, OO programming offers a new and powerful model for writing computer software. Objects are *black boxes* with which one interacts through their methods. This approach speeds up the development of new programs, and, if properly used, improves the maintenance, reusability, and modifiability of software. Notice however that OO programming requires a major shift in the programmer's way of thinking.

6.2.3 Parallel programming

Programs are usually sequential meaning that their execution is carried out by only one processor. The computational time required by their execution varies from a few seconds to a few weeks. Especially in the second case, speeding up the computational time is of obvious interest to every one. Essentially, two approaches may be taken into account to improve the performances. Notice that a combination of them should be considered.

The first one consists in improving the mono processor technology such as its frequency or the efficiency of internal processor components. Compilers can also be improved to produce executable codes which are more and more optimized. The code is then unchanged and, roughly, a factor between 2 and 4 is obtained each year.

Second, one can use parallel computers. They imply the modification of the sequential code by parallelizing it. The aim of this subsection is to give some aspects of parallel computing which helps to understand the ideas of the chapter. However, we refer the parallel programming novice to introductory textbooks such as [116].

There mainly exists two classes of parallel programming paradigms: the shared memory and the distributed memory paradigm. The first one consists in sharing the data through a common memory by using compilation directives. Processors generally access the memory via a bus which becomes a bottleneck if the number of processors is too large. This paradigm is popular because it allows us to deal with a parallel machine without major changes to the sequential code. On the other hand, the computer scientist has a little control on the parallelization implying that generally the performance enhancement could be higher.

The second paradigm (distributed memory) consists in distributing the data to the processors so as to equitably share the work load. Processors needing an information located in an other processor have to communicate through messages. The messages travel through the network which interconnects the processors. Notice that one is obviously interested to keep the amount of communications

as low as possible. Indeed, it represents some extra work which takes a non-negligible amount of time.

This paradigm implies more work but allows us to use a large number of processors efficiently. Note that there are some promising attempts to mix both paradigms, see among others papers in [117].

To finish this short introduction, let us talk about parallel machines. They can also be classified into two categories. First, there are those completely built (mainframes) by constructors (such as IBM, Sun or also Hitachi). They are expensive (a few million euros) but their components are of high quality. These machines come with many useful softwares allowing to easily deal with them. Second, in the past ten years, some home made parallel computers appeared. They are composed of domestic elements (Pentium, FastEthernet) and have offer approximately the same power as the ones built by constructors. Their main advantage is their price, often less than 100 000 euros. But their construction definitely needs specialized skills and some home made softwares have to be designed.

In this thesis, we will focus our attention on the distributed memory paradigm applied to a cluster of standard Pentiums. Its interconnection network is composed of a switch and some FastEthernet links.

To conclude this subsection, let us give some values related to the network considering that the total communication time is composed of a start-up time, the so-called latency, and a communication time which is, for a given bandwidth, proportional to the message size. The network of our Pentium cluster has a latency of $150\mu s$ and a bandwidth of $12MB/s$. To compare, the network of an IBM SP2 has a latency of $10\mu s$ and a bandwidth of $100MB/s$. If the communications represent a large part of the execution time, a supercomputer such as the SP2 will be more efficient.

6.3 Lattice Boltzmann parallelization

Considering a distributed memory paradigm, the parallelization of LB models consists in sharing the work load by dividing the lattice into smaller pieces. These pieces are equitably distributed to the processors in order to balance the load.

We saw in chapter 3 that the dynamics of LB models only requires direct neighbor knowledge. Consequently, one defines regular site as a site with all its neighbors on the same processor and a boundary site as a site with at least one of its neighbors on an other processor. It implies that, for regular site, the dynamics is not altered by the domain decomposition. This is not the case for the sites forming the boundary. Complete knowledge of the neighborhood for these sites implies some communications.
A basic parallel algorithm consists in

1. sending missing informations needed by boundary sites;

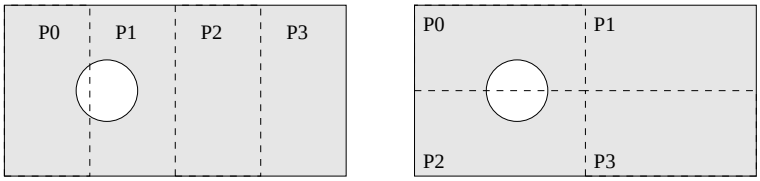


Figure 6.3: Lattice decomposition into 4 bands (left) and into 4 boxes (right). Sites in the circular white area are not used.

2. receiving the information;

3. applying the dynamics on the whole sub-domain.

This simple algorithm implies blocking communications as the processors wait during the communication time. An enhancement consists in overlapping the communications with the computations. The algorithm becomes

1. send missing information;

2. apply the dynamics on sites not on the boundary;

3. receive the information;

4. apply the dynamics on the boundary.

As mentioned previously, a domain has to be decomposed into subdomains. Some typical domain decompositions are presented in figure 6.3. Choosing a decomposition or another depends essentially on the ratio of computational over communication time. Indeed, a decomposition into too thin bands would imply too many communications with respect to the computations. On the other hand, for the same communication volume, a decomposition into boxes rather than bands adds some latency time because there are 4 neighbors and consequently 4 communications are needed instead of 2. Hence, depending the application and its geometry, one has to adjust the decomposition method. Notice that a decomposition into bands or boxes are directly related to the implementation by a multi-dimensional array. We will see, in the following section, that our environment adjusts automatically the decomposition method by using a graph partitioning technique usable in any case.

Hence, an overhead to the initial amount of work is added. It is composed of extra operations such as the communications or the task schedule. Consequently, the speedup should not be equal to the number of processors where the speedup is defined as

$$S_p = \frac{T_s}{T_p} \tag{6.1}$$

where T_s is the computational time of the best sequential code and T_p is the computational time of the parallel execution producing the same results.

6.4 PELABS

6.4.1 Description

In general a 2D lattice based application with any neighborhood topology is represented as a 2D array (called a matrix in what follows) in the computer. Examples were presented in various codes of chapter 3. The main advantage of this data structure is the intrinsic ease to access any site and its neighbors. Therefore, fast memory access is possible without indirection. On the other hand, a matrix requires to keep the domain intact even if it presents some *holes* such as obstacles in the simulation of porous media flows, or buildings in a wave propagation simulation in a city. Hence, unused areas are represented in the matrix. Moreover, modifications, for example changing the lattice topology, often imply many changes in the code and can be responsible for many bugs. Below, we shall call this approach the classical method.

We argue that one can implement these models more adequately by considering some software engineering aspects. Indeed, to reduce development time, one is interested to write codes which are reusable. A modification somewhere in the code should not imply too many modifications elsewhere. The parallelization should be transparent to the user. However, all these aspects should alter the performances the least possible. But, one has to keep in mind that the previous aspects will generate a performance degradation. Let us try to keep it as low as possible.

Hence, we propose to represent the lattice (and its operators) using an object oriented approach. We consider a vector of cells instead of a matrix. Every cell knows its neighborhood via a list of indexes relative to this vector. This vector can be seen as a flattened matrix to which topological information has been added. Figure 6.4 sketches the transformation from a matrix to a vector of cells. Notice that lattices of any topology and of any dimension are flattened in the same way. The proposed environment is called PELABS (**P**arallel **E**nvironment for **LA**ttice **B**ased **S**imulations)

With this representation the unused areas (holes) are no longer included in the computation and domains of any shape can be treated equally well. Also, it is easy to remove or add an arbitrary number of cells, move a block of cells of any shape to another location or to another processor. Finally a lattice traversal is straightforwardly executed.

6.4.2 Architecture

This technical subsection presents the object-oriented architecture of PELABS by using the UML standard [118]. PELABS is decomposed into five libraries:

- The `Timer` library: It features some timer functionalities by proposing a timer which measures the number of seconds elapsed (`RT_Timer`) and a

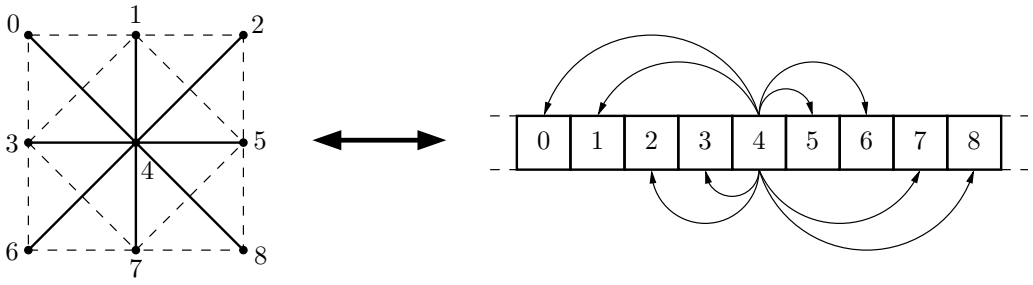


Figure 6.4: Illustration of the transformation of a D2Q9 lattice implemented as a matrix (left) to a vector of cells (right). Only the neighbors of the site labeled 4 are represented.

timer which counts the number of CPU cycles elapsed (`CPU_Timer`) during an execution. These classes inherit from the super-class (`Timer`) which groups the common features and attributes. Figure 6.5 presents the library.

- The `Misc` library: It groups some miscellaneous services. The type `Bool`, which is not available in every compilers, is defined. Two random number generators are provided (`Random` and `RandomS`). The second one is static in the sense that the random seed is unique and thus common to each instance. Vectors and matrices (`Vector` and `Matrix`) are described independently of the element type. Classical operations are provided for them. Character strings (`String`) and generic unidirectional list (`List`) are defined. By inheriting from `Packable`, transmissible versions of the two latter are proposed. See the `ComBox` description for details on packable objects. Figure 6.6 presents the library.

- The `ComBox` library: It provides a communication toolbox. The `ComBox` class encapsulates the functionalities of traditional communication libraries such as MPI [119] or PVM [120]. Hence, it enables us to start and stop a parallel session, and to communicate messages of basic types. The `ComBuf` class describes communication buffers. Packing and unpacking methods of various types are provided. The size of the communication buffer is simply

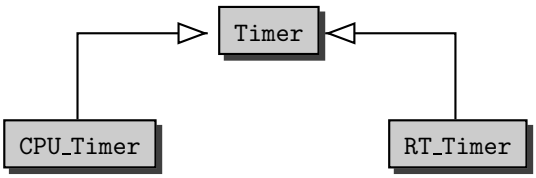


Figure 6.5: The `Timer` library. The arrows mean *inherit from*.

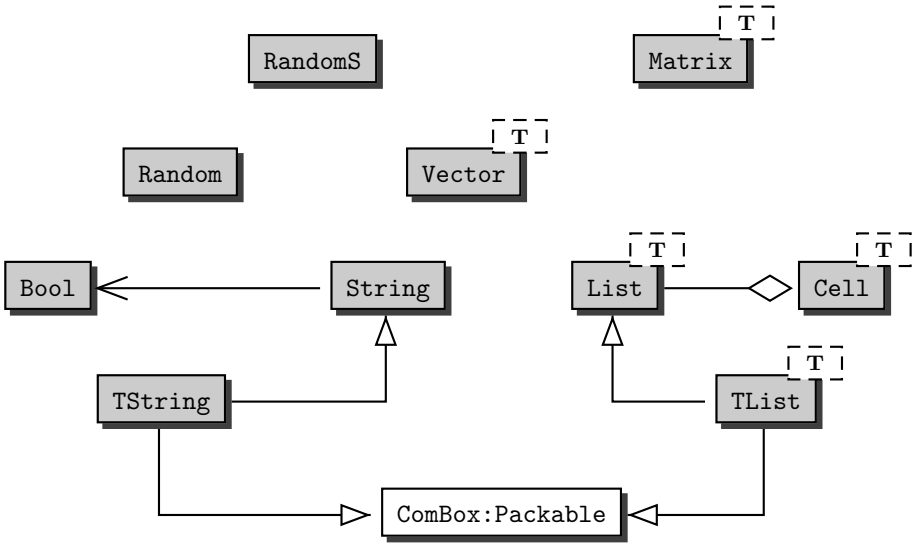


Figure 6.6: The `Misc` library. Gray boxes are library classes. White boxes are classes of an other library. The closed arrows mean *inherit from*, the open ones stand for *use* and the diamond ones mean *composed of*. The boxed upper symbol `T` denotes a template class.

but efficiently managed by allocating more memory than necessary when space is needed. This can be seen as an anticipation. Hence, the number of reallocation is low and the space allocated is not too big. The transmissible notion has been introduced by defining the virtual class `Transmissible` which imposes to define some communication methods in all classes inheriting from `Transmissible`. Moreover, we define the virtual class `Packable` which imposes to define pack and unpack methods in all classes inheriting from it. Hence, the `TString` class, defined in the `Misc` library, is `Packable`. It implies that pack and unpack methods have been implemented and, by inheritance, any packable object can be sent and received. Figure 6.7 presents the library.

- The `GI` library: It offers a useful graphical interface. It allows us to open windows in which basic shapes can be drawn. Files in `ppm` format can be created. Figure 6.8 presents the library.

- The `Lattice` library: It describes some of the most used $DdQ(q+1)$ lattices where d denotes the number of dimensions and q the number of links. However, note that the `Lattice` architecture enables us to append lattices which are not provided with the library. Any lattice inherits from `VLattice` which is a virtual class imposing to its heirs to define some simple memory allocation methods. The class `VLattice` groups the common features to all lattices such as methods managing the parallel execution, graphical methods and some methods dealing with memory. Moreover, any lattice is

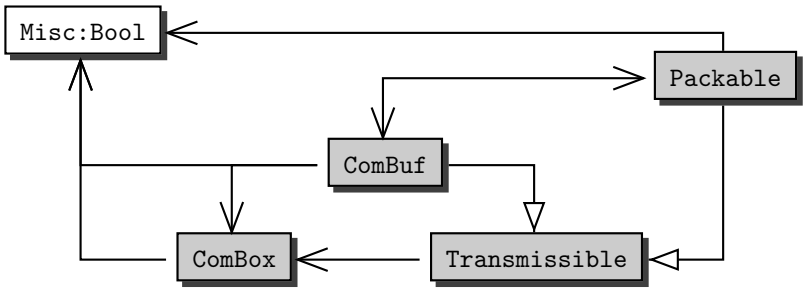


Figure 6.7: The `ComBox` library. Gray boxes are library classes. White boxes are classes of an other library. The closed arrows mean *inherit from* while the open ones mean *use*.

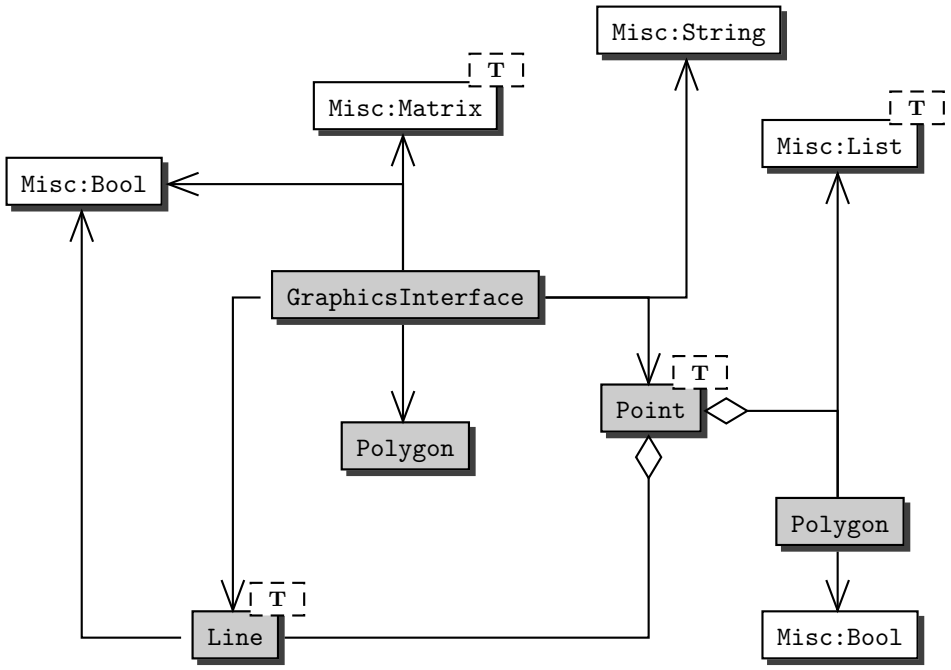


Figure 6.8: The `GI` library. Gray boxes are library classes. White boxes are classes of an other library. The open arrows stand for *use* and the diamond ones mean *composed of*. The boxed upper symbol `T` denotes a template class.

composed of cells. The basic class is `VCell` which is a packable class. `VCell` is a virtual class meaning that some methods have to be defined by each heir. These methods are essentially pack and unpack methods. Communications necessary to the parallelization are identified with a communication table (`ComTable`) indicating which indexes have to be sent and which ones have to be received. To describe the geometry of a lattice, one has to define a lattice shape (`LatticeShape`) which is an abstract object. The shape can be in 2D (`LatticeShape2D`) or in 3D (`LatticeShape3D`). These objects are defined as a box from which elements (e.g. squares, disks, cylinders) can be removed. These elements are defined by a 2D or a 3D mask or by simple shapes consistent with the dimension. Figure 6.9 presents the library.

With these building blocks, we can construct a model (`AModel`), i.e. any lattice based application. A model uses a specific lattice by inheriting from it. It consequently also inherits from `VLattice` which among others provides methods for managing a parallel execution. A model needs some cells (`ACell`) which are implemented by inheriting from `VCell` and by defining appropriate attributes. This is depicted in figure 6.10.

The piece of program shown in code 6.1 illustrates the way to define a real-valued lattice based application on a D2Q9 lattice. A typical example of the propagation step, as well as the collision method corresponding to equation (3.6) are respectively presented in code 6.2 and code 6.3.

6.4.3 Parallelization

Domain decomposition

We want to provide a general domain decomposition applicable to every lattice (i.e. to a `VLattice`). We choose to divide the space into k subsets. To ensure load balancing in a simple way, we can set $k = \alpha p$, where p is the number of processors and α is a positive integer constant. The subsets are then randomly distributed to the processors in a round robin way such that every processor will have α subsets, corresponding to all regions of the domain. This technique however is not scalable and, if p is large, one has to choose $\alpha = 1$ to ensure locality of the communications. In what follows, we will suppose that the load is equitably balanced. Thus we will only consider the case $k = p$.

In most current lattice based implementations, the subsets are rectangular areas (e.g. bands or squares). From a compactness point of view (i.e. communication to computation ratio), this is not the best choice especially when the domain is sparse.

The general decomposition problem can be formulated as follows: find k subsets such that the number of cells with a neighbor in a different subset is minimized. This is nothing else than the so-called graph partitioning problem that we shall solve for the given lattice topology.

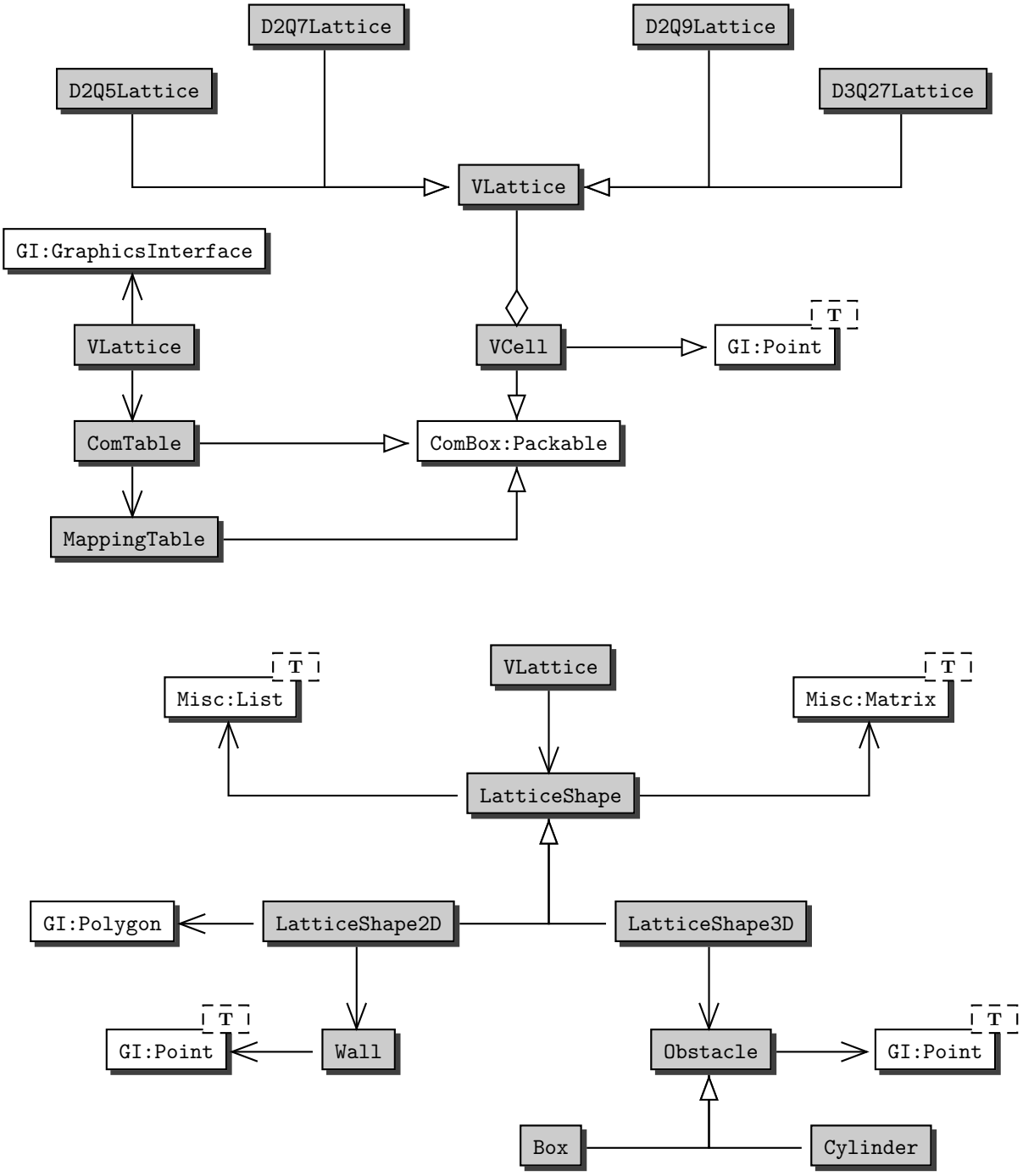


Figure 6.9: The `Lattice` library. Gray boxes are library classes. White boxes are classes of an other library. The closed arrows mean *inherit from*, the open ones stand for *use* and the diamond ones mean *composed of*. The boxed upper symbol **T** denotes a template class.

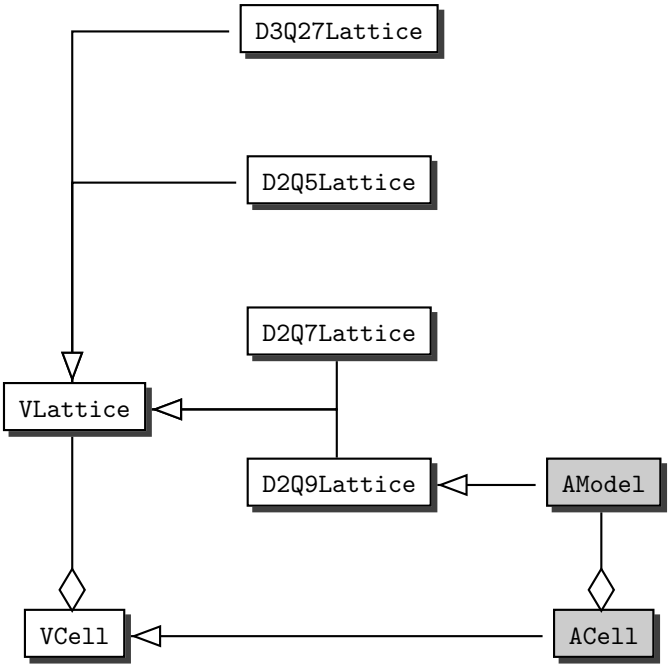


Figure 6.10: A typical model (`AModel`) defined on a D2Q9 lattice. The model is composed of cells defined by the class `ACell`. White boxes are PELABS classes. The closed arrows mean *inherit from* and the diamond ones stand for *composed of*.

Graph partitioning problem

The graph partitioning problem (GPP) amounts to looking for k subsets of a graph where the number of cut edges is minimized. A cut edge is defined as an edge which connects two nodes belonging to two different subsets.

More formally, let $G = (V, E)$ be a graph where V is a set of n nodes and E is a set of e edges. Let $M = (m_{ij})$ with $i, j = 1, \cdots, n$ be a connectivity matrix describing E. Let k be a positive integer. A k-way partition of G is a set of disjoint subsets $s_1, \cdots, s_k$ of nodes such that $\bigcup_{i=1}^{k} s_i = G$.

For any $v \in V$, we define the index function σ as: $\sigma(v) = b$, if $v \in s_b$ where $b = 1, \cdots, k$. The cost of a partition is defined as

$$C = \sum_{\forall ij, \sigma(i) \neq \sigma(j)} m_{ij}$$

The GPP consists in minimizing C, for a given k. This problem is known to be NP-Complete [121]. Indeed, a combinatorial calculation shows that for a small graph of 40 vertices there are 10^{20} 4-way possible partitions.

Code 6.1 Main part of the declaration of a class corresponding to a physical model evolving on a D2Q9 lattice in which the particles are represented by real values. The lattice as well as its cells are respectively inherited (line 1) and used as an attribute (line 16). Consequently, these two components are completely reusable.

```
class AModel: public D2Q9Lattice                                          1
{                                                                         2
public:                                                                   3
  AModel(...);                  // Constructs the object.               4
  ~AModel();                    // Destroys the object.                 5
   // Main methods                                                        6
  void init();                  // Initializes the model.               7
  void propagation();           // Propagates particles along the links. 8
  void collision();             // Collides particles according to a    9
                                // collision operator.                  10
                                                                         11
   // Other methods applicable to this model                             12
  ...                                                                    13
private:                                                                 14
   // Main attribute                                                     15
  ACell **localCells_;  // Vector of cells with four links and           16
                        // where particles are real value.               17
   // Other attributes                                                   18
  ...                                                                    19
};                                                                       20
```

Therefore, one has to find relevant heuristics to solve this problem. After the famous Kernighan-Lin algorithm [122] in 1970 many interesting approaches have been proposed such as greedy heuristics [123, 124, 125], genetic algorithms [126, 127], simulated annealing techniques [128], spectral partitioning methods [129] or multilevel algorithms [130].

Two important selection criteria for such methods are the cost of the partition computation and the quality of the partition. The multilevel algorithms seem to satisfy well these criteria [130, 131].

There are several available libraries solving the GPP. The most famous ones are METIS [132, 133], CHACO [134], PARTY [135] and SCOTCH [136]. In our case, the METIS library which implements multilevel algorithms, turned out to be the best one. Indeed, our tests show that METIS is faster and that its partitions are well adapted to our needs.

To illustrate the use of the METIS library, we present in figure 6.11 an irregular domain made of buildings and free space. Figure 6.12 presents the computed

Code 6.2 Implementation of the propagation method of the basic equation (3.6).

```
void AModel::propagation()
{
  int c,i,n;
  // getNbNeighbor() is a static method implemented in ACell.
  int nbNeighbor=localCells_[0]->getNbNeighbor();
  // Propagates the particles from the state t to the state t + 1.
  // nbCell_ is defined and set in VLattice.
  for (c=0;c<nbCell_;c++)
    for (i=0;i<nbNeighbor;i++) {
      n=localCells_[c]->getNeighbor(i);
       // The last argument set to true indicates to set
       // particles in the temporary state.
      localCells_[n]->setParticle(i,localCells_[c]->
                                  getParticle(i),true);
    }
  // Sets the current state to be the one at time t + 1. This method,
  // defined in ACell, is static and consequently applied to the first cell only.
  localCells_[0]->switchState();
}
```

Code 6.3 Implementation of the collision method of the basic equation (3.6).

```
void AModel::collision()
{
  int c;
  float density,u[2];
  // Methods computeWaterDensity, computeWaterVelocity and update are defined
  // in ACell. The last on implement the collision operator. A complete example is
  // presented in appendix C.
  for (c=0;c<nbCell_;c++) {
    density=localCells_[c]->computeWaterDensity();
    localCells_[c]->computeWaterVelocity(density,u);
    localCells_[c]->update(density,u);
  }
}
```

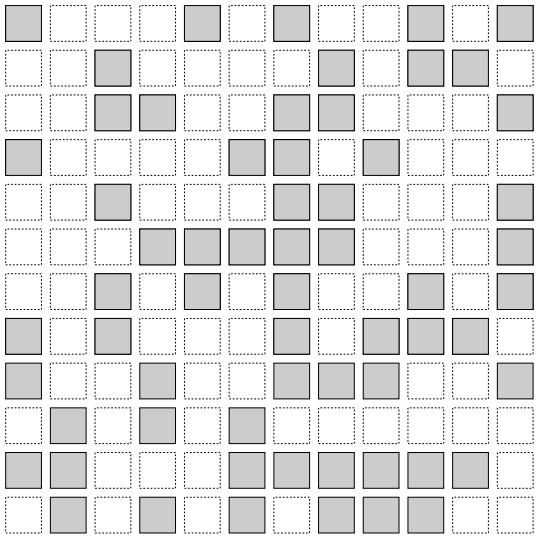


Figure 6.11: A 800 by 800 Manhattan-like city composed of buildings in gray. These have been selected with a probability of 0.4 among the regular 12 by 12 building grid. The irregularity $\delta = 0.294$ represents the ratio between the gray area and the total area.

partition which took 9 seconds on a 266 MHz pentium.

We introduce the following notation: n^2 is the number of sites, p is the number of processors and δ is a real value $\in [0; 1]$ expressing the domain irregularity, i.e. the fraction of the n^2 sites which actually does not belong to the domain. We also assume that this irregularity is roughly equally distributed over the domain.

We observe two types of boundaries in the partition of figure 6.12. The first one is the border between two different subsets while the second one is an internal boundary. As only the first one induces communications we will subsequently call it boundary. As evidenced in figure 6.13, we also observe that the subsets are often bounded by holes. This is an efficient way for the graph partitioning algorithm to reduce the effective length of the boundary, i.e. the part involved in inter-processor communications. Thus, bigger is the irregularity, smaller is the perimeter of the subset.

Subset boundary length

To estimate the communication time, we need to compute the length of the subset boundary. It is actually a difficult problem to find an exact analytical expression for this length. Indeed, its geometry is quite complex. As an example, see the thick line in figure 6.13.

However, subset perimeter length can be estimated from its area S as $a_1\sqrt{S}$. Thus for example if the area is a square $a_1 = 4$, and $a_1 = 2\sqrt{\pi}$ if the area is a disk. In our case the mean subset area is equal to $\frac{n^2}{p}(1 - \delta)$. But as shown in

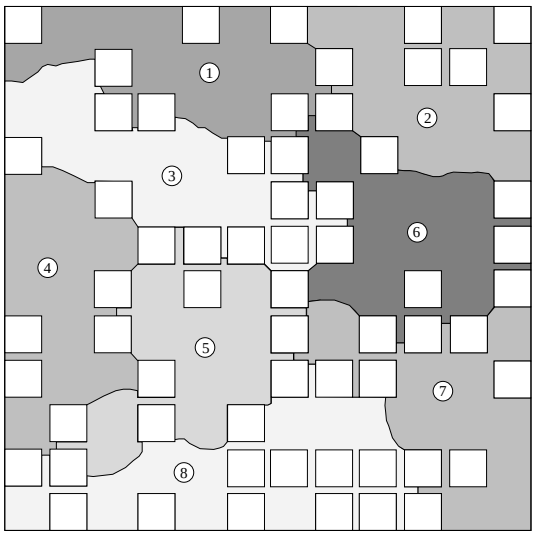


Figure 6.12: A 8-way partitioning of the domain presented in figure 6.11 using the METIS library. The domain has approximatively 450000 nodes and the decomposition took 9 seconds on a 266 MHz pentium.

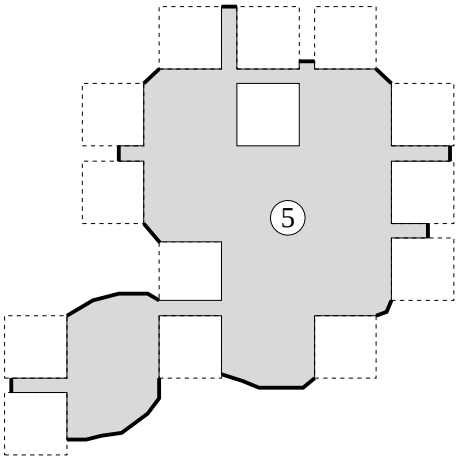


Figure 6.13: Zoom of the subset labeled by a 5 in figure 6.12. Allocated sites are drawn in gray while the white region represents non allocated areas. The continuous lines are the boundary of the subset where thicker lines are those which imply communications. Dashed lines depict borders belonging to another subset.

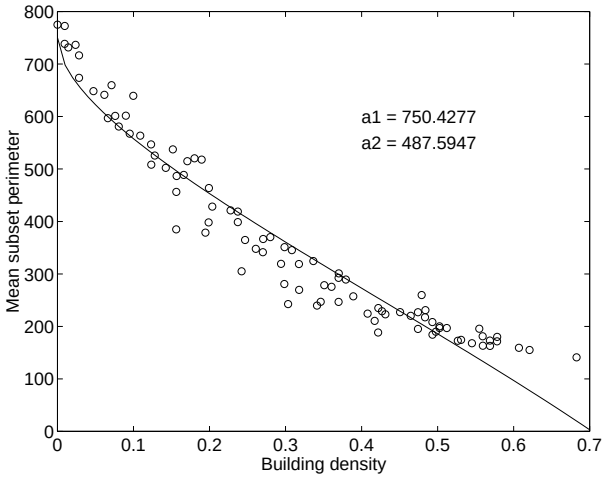


Figure 6.14: Mean subset perimeter of the 8-way partitioning of various irregular domain. These are built from a regular Manhattan-like city (see figure 6.11) where building blocks are selected with appropriate probabilities. Circles are the measured mean value while solid line is the fit computed from (6.2).

figure 6.13 the boundary is not continuous. Then a better approximation of the length of the subset boundary can be done by subtracting a quantity proportional to the hole perimeters: $a_2\sqrt{\frac{n^2}{p}\delta}$. Thus an estimate of the boundary length can be written as

$$L(\delta) = a_1\sqrt{\frac{n^2}{p}(1-\delta)} - a_2\sqrt{\frac{n^2}{p}\delta} \tag{6.2}$$

where a_1 and a_2 are constant values for a given number of processors (subsets).

To validate (6.2), we consider a 8-way partitioning of various irregular domains of different densities. These are built from a regular Manhattan-like city (see figure 6.11) where building blocks are selected with appropriate probabilities. Mean boundary lengths of subsets are computed exactly and as can be seen in figure 6.14, they are satisfactorily fitted with (6.2).

Communications

In the classical approach, where the domain D is represented as a d-dimensional array and the irregularities are accounted for by masking the corresponding sites, inter-processor communications are easily performed by exchanging regular array sections between adjacent processors.

In the PELABS case, the communication step consists in building a communication buffer for every processor belonging to the neighborhood of each subset boundary.

We use asynchronous, non-blocking communications in which communications overlap with the computation. To this end, one proceeds as follows: (i) the boundary cells are sent out; (ii) the inside part of each subdomain is updated; (iii) the neighbor cells are received from the remote processors and the subset boundaries are updated. In most lattice based applications, the communications are totally overlapped with the propagation-collision step thus the communication time is apparently non-existent.

6.5 PELABS performances

6.5.1 Outline

In order to compare PELABS with the classical method (full array, block partitioning), we consider the run time complexity of both methods. For the sake of comparison, we take a 2D domain. For the classical method, we also assume that the domain is decomposed into bands. The communications are then composed of the lines (i.e. the boundary sites) between each band. Non-blocking message passing primitives are also used to overlap communications with computations. With this method, even if the domain is sparse, every band has the same size. A surplus of time for the domain traversal and of data to communicate are expected.

6.5.2 Theoretical models of performances

PELABS

The execution time of the three phases: collision, in-processor propagation and inter-processor propagation (communication) of PELABS can be written as

- **Collision:** $\frac{n^2}{p}(1-\delta)$ sites are computed in time $T^{(1)}_{comp}$.
- **Propagation:** $\frac{n^2}{p}(1-\delta)$ sites are shifted in time $T^{(1)}_{shift}$.
- **Communication:** We suppose that each subset has $p' \leq p-1$ neighbors and that the cell distribution among the processors is uniform (note that p' is independent of p if the latter is large enough). Thus we need p' communication buffers to realize the communication. The time to build each of the p' messages is $\frac{L(\delta)}{p'}T_{pack}$. All together the p' non-blocking communications require a time $p'T_{startup}$ to be initialized and each one is executed in a time proportional to the message size, namely $\frac{L(\delta)}{p'}T_{comm}$.

Thus, the total theoretical execution time can be written as

$$\begin{aligned} T_{PELABS}(p,n,\delta) &= \frac{n^2}{p}(1-\delta)T_1 + L(\delta)T_{pack} \\ &+ \max(0, L(\delta)T_{comm} - T^{(1)}_{col}) + p'T_{startup} \end{aligned} \tag{6.3}$$

where $T_1 = T^{(1)}_{comp} + T^{(1)}_{shift}$ and $T^{(1)}_{col} = \frac{n^2}{p}(1-\delta)T^{(1)}_{comp}$. The maximum function used above takes into account the overlap between the communication and the collision time.

Classical method

The theoretical execution time for the three stages of the classical method can be expressed as

- **Collision:** $(1-\delta)\frac{n^2}{p}$ sites are computed in time $T^{(2)}_{comp} = T^{(1)}_{comp}$ and $\delta\frac{n^2}{p}$ are visited in time T_{visit}.
- **Propagation:** $(1-\delta)\frac{n^2}{p}$ sites are shifted in time $T^{(2)}_{shift}$ and $\delta\frac{n^2}{p}$ are visited in time T_{visit}.
- **Communication:** the non-blocking communications are started up in a time $2T_{startup}$ and executed in a time $2nT_{comm}$ as the length of the band boundary is equal to n and there are 2 sides.

The execution time for this classical method can be summarized as

$$\begin{aligned} T_{classic}(p,n,\delta) &= \frac{n^2}{p}(T_2 - \delta\cdot(T_2 - 2T_{visit})) \\ &+ \max(0, 2nT_{comm} - T^{(2)}_{col}) + 2T_{startup} \end{aligned} \tag{6.4}$$

where $T_2 = T^{(2)}_{comp} + T^{(2)}_{shift}$ and $T^{(2)}_{col} = \frac{n^2}{p}(1-\delta)T^{(2)}_{comp} + \delta\frac{n^2}{p}T_{visit}$. Again the maximum function describes the overlap between the communication and the collision time.

6.5.3 Measured performance

A typical application

To validate the theoretical model of performances, we choose the so-called ParFlow application [109, 9] (wave propagation in a city) which requires a square lattice and a simple collision operator: one has typically

$$f_i(t+1, \vec{r}+\vec{v}_i) = \frac{\mu}{2}\sum_{j=1}^{4} f_j - f_{i+2}$$

where μ is an attenuation coefficient depending on the city geometry and building properties (for example $\mu = 1.0$ means that there is no obstacle while $\mu = 0.0$ means that there is a building). If the wave is reflected on the building

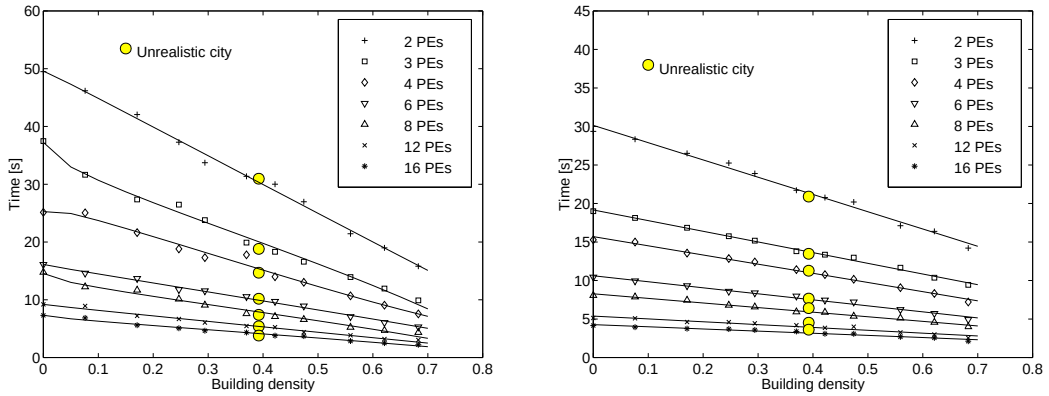


Figure 6.15: Execution time as a function of the building density, for several number of processors, on a Manhattan-like city of size $n = 800$ (see figure 6.11), during 100 iterations. The left hand side shows the PELABS times, while the execution times of the classical method are depicted on the right. Fit (solid line) of the measured execution times (isolated points) for various numbers of processors and various building densities according to equations (6.5) and (6.6). Filled disks are execution times for the unrealistic city on figure 6.16.

walls, it is unnecessary to compute the wave inside the buildings and, as before, δ corresponds to the ratio between the building area and the total area.

In order to have various irregular domains, we build from the regular Manhattan-like city shown in figure 6.11 different domains by varying the selection probability of buildings.

Data fits and discussion

Figure 6.15 shows the actual performance of the two implementations of the ParFlow simulation. For $\delta = 0$, the plain method is twice faster than the PELABS implementation. However, as the irregularity of the domain increases, the two execution times intersect and then PELABS becomes faster.

The performance model proposed in the previous section can be validated with the present time measurements. Let us rewrite (6.3) and (6.4) for given values of n and p. For n large enough, the collision time is greater than the communication time and the next to the last term of (6.3) and (6.4) vanishes. Thus

$$T_{PELABS}(\delta) = c_1(1-\delta) + c_2(\sqrt{1-\delta}) + c_3(\sqrt{\delta}) + c_4 \quad (6.5)$$

$$T_{classic}(\delta) = c_5\delta + c_6 \quad (6.6)$$

with $c_1 = \frac{n^2}{p}T_1$, $c_2 = a_1\sqrt{\frac{n^2}{p}}T_{pack}$, $c_3 = -a_2\sqrt{\frac{n^2}{p}}T_{pack}$, $c_4 = T_{startup1}$, $c_5 = -\frac{n^2}{p}(T_2 + 2T_{visit})$ and $c_6 = T_{startup2} + \frac{n^2}{p}T_2$.

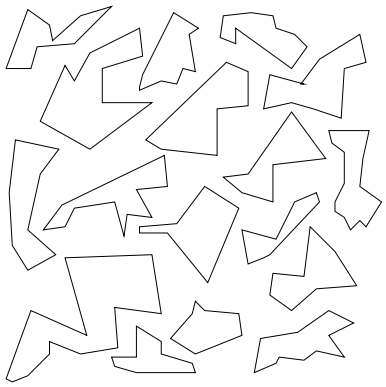


Figure 6.16: An unrealistic city of size 800 by 800. Its irregularity corresponds to $\delta = 0.392$.

Using these equations, the values of the c_i's can be extracted from the measured performance through a mean-square data fit. Figure 6.15 shows the prediction of (6.5) and (6.6) as a function of the building densities δ, for various number of processors p. We see that the theoretical curves fit the measured times pretty well.

In our performance model the domain irregularity is described by the real value δ. It is not clear that a single parameter is enough to characterize the average subset perimeter length. To validate this assumption, a simulation is also run on an unrealistic city (see figure 6.16), whose domain irregularity departs significantly from that of figure 6.11. The corresponding execution times are also shown in figure 6.15 (large gray circles). These results confirm that δ is a good metrics of the domain geometry when computing the run time performance of our implementation.

The value of $\delta = \delta_{critic}$ for which the execution time of the classical and PELABS methods are equal depends on the application, the domain size and the number of processors. For the Parflow simulation, the value of δ_{critic} is depicted in figure 6.17 for various numbers of processors.

More generally, we may assume from this study that δ_{critic} is typically larger than 1/2. From figure 6.15, we also see that, for a large enough irregularity, the execution time of the PELABS implementation varies linearly with δ. We have already noticed that the subset perimeter length $L(\delta)$ decreases rather quickly when δ increases. Thus, this linear behavior is in agreement with equation (6.3) where the contribution $L(\delta)T_{pack}$ is neglected.

Remembering that for n large enough, computation overlaps with communication, a simple expression can be found for δ_{critic} by equating equations (6.4) and (6.3), without the communication steps and with $L(\delta) \to 0$. We obtain, after

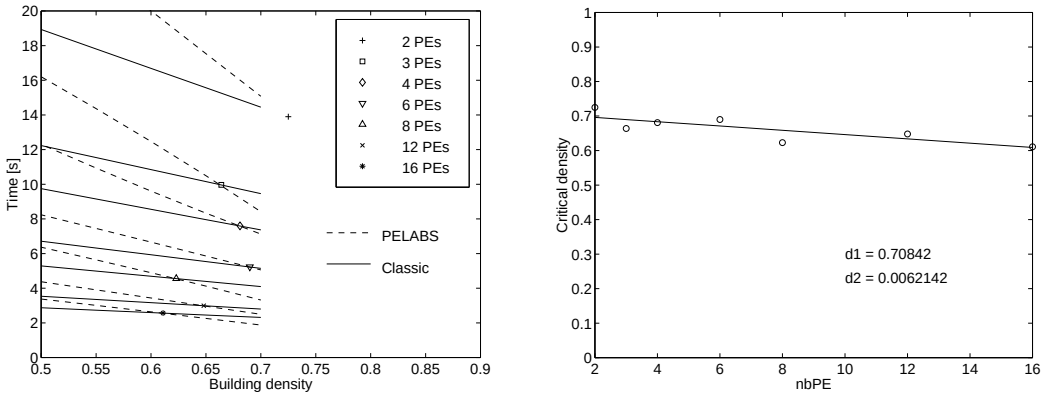


Figure 6.17: On the left, execution time versus building density for the two models. The intersection shows the value $\delta = \delta_{critic}$ for which the execution times are equal. Dashed lines are PELABS execution times while solid lines are those of the classical method. On the right, δ_{critic} is given versus the number of processors. The circle are the measured intersections and the straight line is the fit predicted by equation 6.7.

some algebra

$$\delta_{critic} = d_1 - d_2 \frac{p}{n^2} \tag{6.7}$$

This relation is confirmed by figure 6.17, where we observe that δ_{critic} decreases linearly as p increases. The other prediction of equation (6.7) is that δ_{critic} increases as the domain size n increases, which is a reasonable assumption although we have not yet checked it explicitly through a simulation.

6.6 Employing METIS towards optimal cache memory use

6.6.1 Motivation

A lattice used in a typical simulation requires several MBytes of memory. As the cache memory in this case is not big enough, cache misses occur. In order to minimize their number, we divide into regions the subdomains associated to each process. This decomposition is done to reduce the inter-memory communications. This is most efficient if the region and cache memory size are comparable.

After an introduction to cache memories, we propose a theoretical and simplified model of a cache memory which provides a better understanding of its functioning. Next, we present and discuss results obtained with a classical simulation.

6.6.2 Introduction to cache memories

A cache memory is a small and fast memory holding recently-accessed data. It is designed to speed up subsequent access to the same data.

When data is read from, or written to, main memory a copy is also saved in the cache, along with the associated main memory address. The cache monitors addresses of subsequent reads to see if the required data is already in the cache. If it is (a cache hit) then it is returned immediately and the main memory read is aborted (or not started). If the data is not present in the cache memory (a cache miss) then it is fetched from main memory and also saved in the cache.

The cache is built from faster memory chips than main memory so a cache hit takes much less time to complete than a normal memory access. The cache may be located on the same integrated circuit as the CPU, in order to further reduce the access time. In this case it is often known as the primary cache. Its size is typically equal to 32 KBytes. A secondary cache level, typically of 1 MBytes, can be placed outside the CPU chip.

The most important characteristic of a cache is its hit rate which is the fraction of all memory accesses within the cache. This in turn depends on the cache design but mostly on its size relative to the main memory. The size is limited by the cost of fast memory chips.

The hit rate also depends on the access pattern of the particular program being run (the sequence of addresses being read and written). Caches rely on two properties of the access patterns of most programs: temporal locality and spatial locality. The first one says that if something is accessed once, it is likely to be accessed again soon. The second tells that if one memory location is accessed then nearby memory locations are also likely to be accessed. In order to exploit spatial locality, caches often operate on several words at a time, a block typically equal to 256 Bytes.

When the cache is full and it is desired to cache another line of data then a cache entry is selected to be written back to main memory or flushed. The new line is then put in its place. Which entry is chosen to be flushed is determined by a replacement algorithm dependent on the architecture (usually least frequently used data).

6.6.3 Simplified theoretical model of a cache memory

In order to understand the mechanism of data exchange between a main and a cache memory, we propose a theoretical model based on a four neighbor topology. Generalizing to an other topology is straightforward.

We suppose our simplified cache memory to be divided into N_{block} blocks of size k. An access to data not present in the cache memory produces a cache miss. This implies loading a block which contains the referenced data in the main memory. If the cache memory is full, the older block is flushed, thus freeing

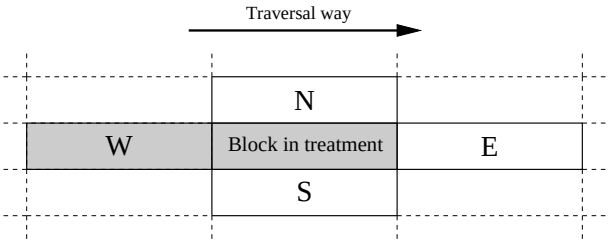
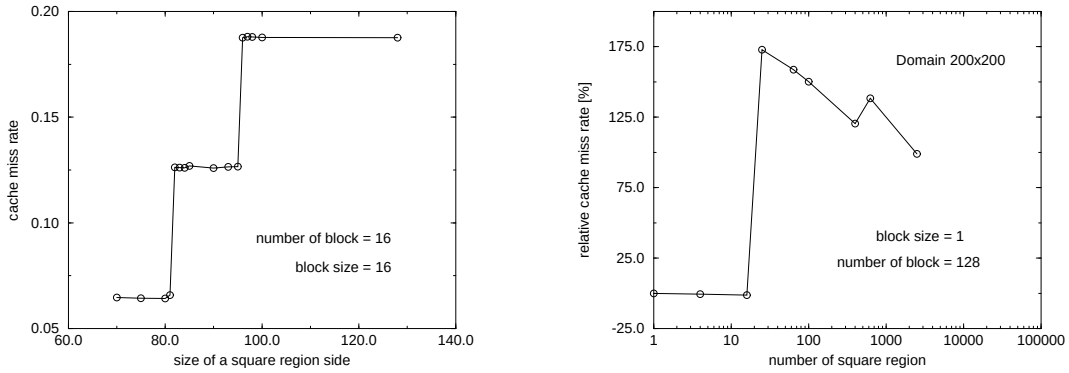


Figure 6.18: Neighbors to load during the treatment of the middle block. The gray areas represent blocks which are necessarily loaded during the treatment of the previous blocks.

some space for the new block.

We consider a rectangular region of size $n_x \times n_y$. The traversal of a rectangular area from left to right and from top to bottom allows to exhibit, in our theoretical model, three main cache memory regimes for a cache size C_{cache}. Indeed, a block in treatment implies the loading of one, two or three of its neighbor blocks depending on their presence in the cache memory. This phenomenon is illustrated in figure 6.18. Considering the traversal path, the west neighbor is always present in the cache memory. We define as optimal regime the one which requires the loading of only one (the south) neighbor during the treatment of a block. This regime takes place if the cache memory can contain at least $3n_x - k$ cells ($3n_x \leq C_{cache} + k$). The value k is subtracted from $3n_x$ because the south block is never present in the cache. The optimal regime can therefore be realized with a size slightly smaller than $3n_x$. A much smaller cache memory size implies the loading of three neighbors and corresponds to the worst regime ($3n_x - k > C_{cache} + k$). However, an intermediate regime exists when each block treated requires loading only two neighbors ($3n_x - k \leq C_{cache} + k < 3n_x$). Consequently, a natural way to minimize the number of cache misses is to *reduce* the region size. Thus, the cache memory works in the optimal regime when it can contain at least $3n_x$ cells. With a fixed size of cache memory, these regimes are highlighted in figure 6.19a for various region sizes. We observe that the length of the intermediate plateau is equal to k. As an illustration, the figure 6.20 highlights the three regimes obtained with three different cache memory size.

We apply this model on a square domain divided into regions. Their shapes are square and equal. Let us note that we will talk about domain or subdomain when the context is sequential or parallel respectively. A domain is traversed from left to right and from top to bottom as well as the inner part of the region. The results of a program simulating our model are presented in figure 6.19b for a domain of size $D = 200 \times 200$ and a cache memory of $N_{bloc} = 128$ blocks of size $k = 1$. Insofar as the size of a region is too big, three of its lines do not fit in the cache. Consequently, the regime is not optimal. Beyond the optimal



a) b)

Figure 6.19: a) The three cache memory regimes, predicted by our theoretical model, correspond to the three plateaus of the graph. The latter displays the cache miss rate vs. the size of a square region side. The length of the intermediate plateau is equal to k. b) On a domain of size $D = 200 \times 200$ and a cache memory of $N_{block} = 128$ blocks of size $k = 1$, we measure the relative cache miss rate for a partitioning into equal square regions. This rate can be expressed as $\frac{m^1 - m^n}{m^1}$, where m^n is the cache miss rate for a partitioning into n regions. Therefore, it measures an increase of performances which will allow a comparison with figure 6.22. Insofar as the size of a region is too small, the relative rate is not maximal. Beyond the optimal partitioning into 25 regions, the rate decreases towards the initial rate.

partitioning (in 25 regions in this example), the performances decrease until they correspond to those of a single region. Indeed, the memory arrangement for one or D regions (i.e. one cell per region) is the same. The decrease in performance implies an increase of the total boundary length which is unfavorable in terms of cache misses.

The total access time T_{total} can be expressed as

$$T_{total} = 4D(mT + t) \approx 4DmT \tag{6.8}$$

where T is the main memory access time, t the cache memory access time, m the cache miss rate and D the size of the domain. As $t \ll T$, t can be neglected in comparison with mT when m is sufficiently large. The approximation in equation (6.8) becomes valid since the size of the domain is much larger than the size of the cache memory. Equation (6.8) will allow us to compare our theoretical model with a real simulation in subsection 6.6.5.

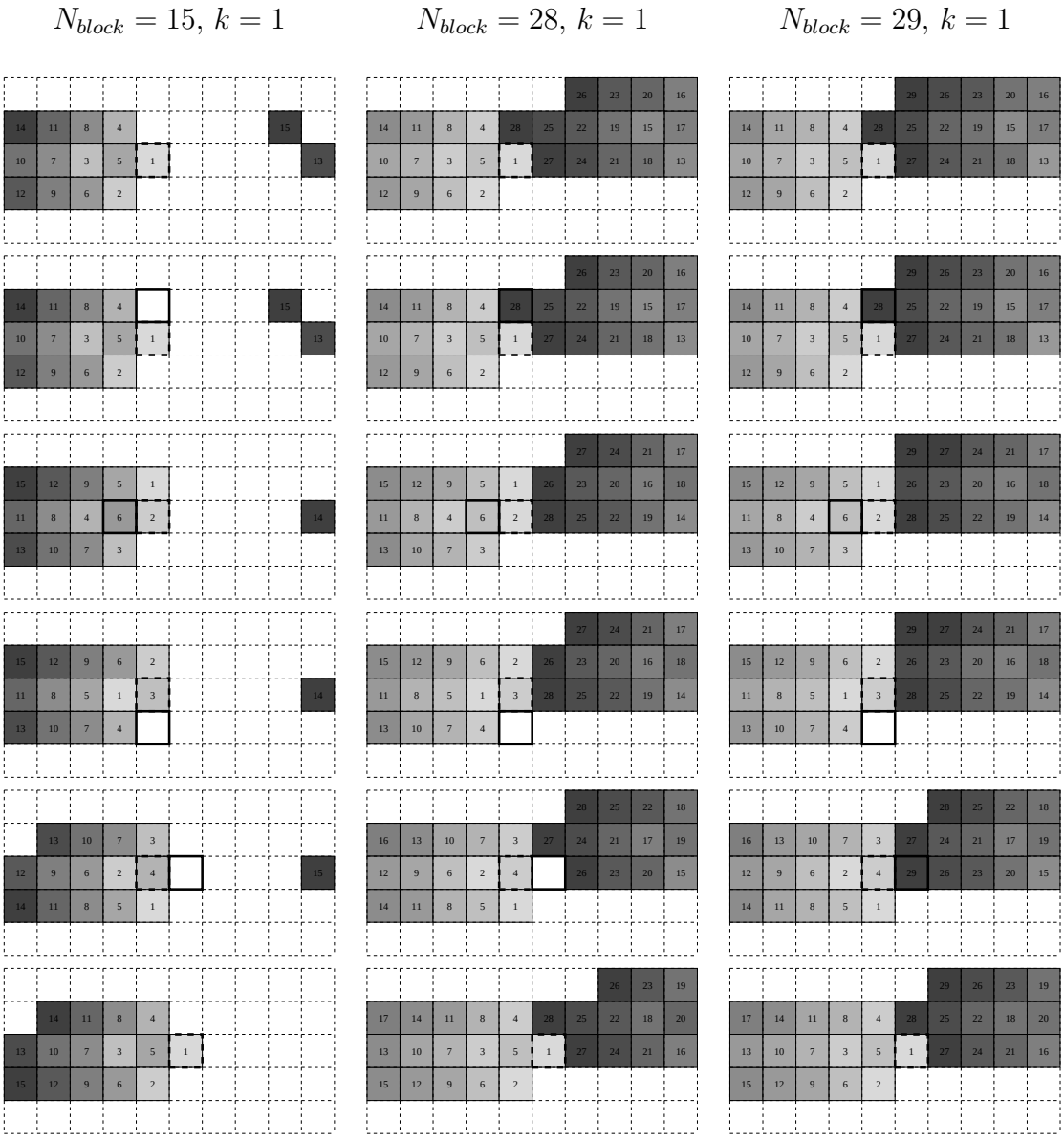


Figure 6.20: The three cache memory regimes highlighted on a domain of size 10×5 for three sizes of cache memory. A cell being updated is drawn as a dashed thick square. During this process, a neighbor to be loaded is drawn as a solid thick square. Blocks present in cache memory are filled with a gray level depending on their age. It corresponds to the time spent in the cache and is also reported. On the right, as $30 = 3n_x \leq C_{cache} + k = 30$, we observe that the optimal regime takes place. The middle scenario shows the intermediate regime where $29 = 3n_x - k \leq C_{cache} + k = 29 < 3n_x = 30$. Finally, the left scenario where $29 = 3n_x - k > C_{cache} + k = 16$ exhibits the worst regime.

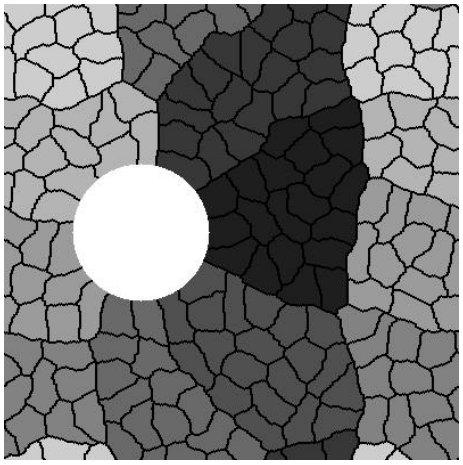


Figure 6.21: Decomposition of an irregular toric domain of size 400×400 into 8 subdomains divided into 20 regions with the use of METIS. Various gray levels indicate the subdomains. The domain is the one of a simulation of a flowing fluid around a cylinder which is not represented in memory.

6.6.4 Use of METIS

We saw in section 6.4 that PELABS proposes a domain decomposition into subdomains with the use of METIS. Remember that these subdomains are frequently non rectangular.

In our theoretical model, the numbering of the lattice was based on a decomposition of each subdomain into rectangular regions. This is not yet possible since various lattice topologies are taken into account and the subdomain geometry is unknown. Again but at another scale, we use METIS to decompose a subdomain into regions and then rearrange the cells in the main memory. An illustration of these various decomposition levels is given in figure 6.21.

The local lattice to each process is fed to METIS as a graph. The output is a vector labelling each cell with its region number. The region traversal determines the cell arrangement. Each region is scanned from left to right and from top to bottom. Other scans have been tried in particular the one which consists in numbering the inner part first and then the boundary. This technique is considered in order to enhance the reuse of the previously loaded blocks. But all of these more complicated numbering techniques do not produce better performances.

6.6.5 Results and discussion

We consider a typical application which is the simulation of a flowing fluid with lattice Boltzmann techniques (see chapter 3 for more details). For that, we define a model `RiverModel` inheriting from a hexagonal lattice `D2Q7Lattice` and uses

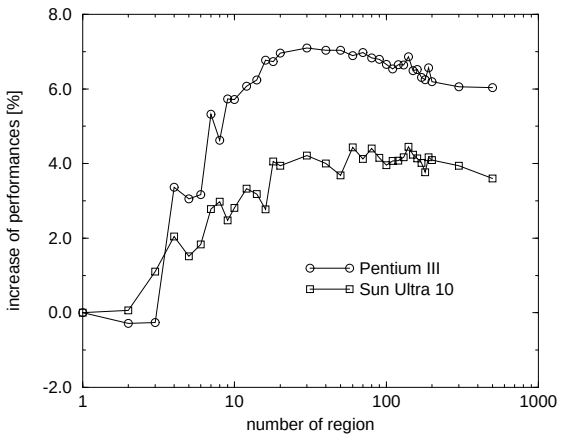


Figure 6.22: Increase of performances vs. the number of region. The wraparound domain considered has a size of 200×200.

`RiverCell` cells. The data structure of a `RiverCell` essentially encapsulates the fluid particles and the neighborhood information. Its size is equal to 128 bytes.

On a toric domain of size 200×200, the performance increase for two architectures (Pentium III and Sun Ultra 10) obtained when the number of regions varies is reported in figure 6.22. To highlight the impact of the memory arrangement, we only measure the propagation step in the flowing fluid simulation.

The Pentium and the Sun have a first cache memory level of 32 KBytes divided into two equal parts reserved for the instructions and for the data. A second level of cache memory of 1 MBytes and 2 MBytes are respectively disposed between the first level and the main memory of the Pentium and the Sun.

We observe in the results of figure 6.22 that the optimal regime is reached when the domain is decomposed into approximatively 30 regions of $\frac{200 \cdot 200}{30} \approx 1333$ cells. For square regions, one side would consist of $\sqrt{1333} \approx 37$ cells. Three lines of such a region occupy $37 \cdot 128 \cdot 3 \approx 14$ KBytes. This is the limit previously established in order to be in the optimal regime (16 KBytes). A heuristic formula giving the number of regions n_{reg} necessary to be in the optimal regime is established. This number is expressed as

$$\sqrt{\frac{D}{n_{reg}}} \cdot C_{cell} \cdot 3 = C_{cache} \Rightarrow n_{reg} = \frac{9 \cdot C_{cell} \cdot C_{dom}}{C_{cache}^2} \tag{6.9}$$

where C_{cache} is the cache memory size in bytes, C_{cell} the number of bytes per cell, n_{reg} the number of regions to be in the optimal regime, D the number of cells in the domain and $C_{dom} = D \cdot C_{cell}$ the size of the domain in bytes.

We observe in figure 6.22 that the results obtained with the Pentium are more

sensitive to a memory arrangement than the Sun. Indeed, the access time to the main memory is longer for the Pentium than for the Sun.

The graph in figure 6.19b presents a theoretical curve which reproduces well the results of figure 6.22. This agreement follows from the linear approximation (6.8). The theoretical curve is more angular than the real one because of the second cache memory level. It plays the role of an ultra-fast main memory.

If one considers an application in its own globality, run in parallel on 8 processors (Pentium III), the typical gain is only of 2%. The domain considered was 600×600. We note that the computational and communication phases are dominant in comparison with memory accesses.

It is important to notice that a memory arrangement implies to move physically the cells in the main memory and not to perform a traversal with many jumps. This traversal requires the computation of the jumps which produces a sufficiently important overhead masking the gain.

To conclude, we have observed that the performances are rather low. However, the decomposition into regions and subdomains are identical problems at two different scales. Hence, their treatment is carried out by the same unity in PELABS and consequently does not require any additional programming effort.

6.7 Summary

In this chapter, we talked about lattice Boltzmann implementation. After motivating our programming language choice, we briefly introduced the main concepts of object-oriented programming and of parallel programming. Next, we explained the usual parallelization of lattice Boltzmann models.

To satisfy some software engineering criteria and some numerical performance requirements, we proposed a parallel and reusable environment called PELABS. Its object-oriented architecture is detailed by UML diagrams and codes.

Graph partitioning techniques are used to decompose the lattice and thus to provide a general parallel framework for any application using lattices. The numerical performances of PELABS are analyzed theoretically and numerically. Roughly, we found that PELABS is at worse two times slower than a usual implementation when the domain is regular. But it can be faster if the domain irregularities are denser.

Finally, we studied the reorganization effect of the cells in the vector. By using the same tool as the one used for the decomposition, we show that a slight improvement can be obtained.

Chapter 7

Conclusion

7.1 Recapitulation and contributions

This section summarizes the contents of this thesis and emphasizes its contributions.

In chapter 2, we started by reviewing some essentials of fluid dynamics. Then, we introduced the basic ingredients of the turbulence theory and presented some traditional numerical models.

In chapter 3, a simple and self-contained presentation of the lattice Boltzmann models was done. It included basic pieces of code which allow the beginners to quickly start to deal with these models. Boundary conditions were then thoroughly treated including the famous bounce-back boundary conditions which are often considered too simple to be discussed. Through experiments, we quantified the discretization error related to these boundary conditions. With the hope of having a *perfect* boundary condition, the so-called non-slip boundary condition, was presented for flat walls in 2D. We gave all details of a 3D derivation. Its discretization error is null for some simple flows (e.g. Poiseuille flow). However, it is unusable for most flows as their boundaries are almost never flat. Moreover, observing that the non-slip boundary condition does not necessarily conserve mass, we proposed a new boundary condition which conserves mass. It is called mass conserving boundary condition. This new condition was compared to the others for a cavity flow. We observed a non-conservation of mass when using non-slip boundary conditions. The discretization error was similar for bounce-back boundary conditions and for our new condition. However, its use leads to a more continuous dynamics as it is also applicable on wall sites. This is not the case for bounce-back boundary conditions as some fields are missing. Then, the various ways to settle a flow were presented and commented. Next, the Smagorinsky model of turbulence was discussed in the lattice Boltzmann context. We highlighted that even if we use a turbulence model, a very high Reynolds number ($Re > 10^6$) can not be reached. We gave some clues about the ways to estimate

this limit.

The chapter 4 proposed a state of the art concerning mesh refinement techniques. We pointed out the disadvantages of existing models and we defined a new algorithm dealing with the shortcomings of its predecessors. We compared this algorithm with existing models on simple experiments. Then, our new mesh refinement algorithm was used to speed up the flow settlement process. We observed a speedup of approximately 2.

Then, in chapter 5, we presented our virtual river model by defining the rules acting on the sediments. We were mainly concerned with two applications. First, we considered the well-known process of scour formation under submarine pipelines due to a uniform current. Our results were in agreement with those found in the literature. Moreover, we successfully simulated the expected effect on the scour depth of adding a spoiler on top of the pipeline. Second, we focused our attention on formation and progression of meanders in rivers. We gave an explanation on how it should work. Our numerical model gave some promising preliminary results indicating that our explanation is valid.

Chapter 6 started to review some computer science aspects. Then we presented PELABS in details. PELABS is our parallel and reusable environment for lattice based simulations. To deal efficiently with any domain, we used a graph partitioning technique. The PELABS numerical performances were measured and successfully compared to a theoretical model of performance. Despite the legend surrounding object oriented programming which pretend that it is so slow, we measured, at worst, a factor 2 between computational times of a traditional implementation and PELABS. We also showed that if the domain is sparse enough, PELABS can produce better performances than a traditional implementation. Then, we studied the effect of moving cells inside the memory in order to speed up the traversal of the data structure. The aim is to make a more optimal use of the cache memory. Roughly, we observed that the line by line arrangement is close to the best arrangement which can be realized by using again a graph partitioning technique.

7.2 Discussion

Experiences collected all along this thesis allow us to comment some aspects of the lattice Boltzmann models.

Since several years, the understanding of these models has considerably progressed. From simple flows in straight channels, one is today able to simulate complex flows composed of many fluids or even fluid flowing into complex geometries. These models are often presented as competitors to classical methods (generically called CFD, see section 2.2). However, it is important to keep in mind that a long way is still to be covered so as to reach the know-how acquired in CFD. For example, flow settlement techniques (section 3.4) or turbulence mod-

els are poorly understood in the lattice Boltzmann context but they have been largely studied and are, consequently, better mastered in CFD.

The major advantage of the lattice Boltzmann models are their ability to solve tough problems where other methods struggle or fail. The evolution towards a sediment model is a perfect example. The applications presented above have shown that, on one hand, well-known processes may be well reproduced (scour under a submarine pipeline) and on the other hand that the current understanding may be improved (meanders in rivers). These models can then be effectively considered as efficient tools allowing us to explore new ways.

The ease to produce computer implementation of the lattice Boltzmann models is also an important advantage. Indeed, the code is simpler than those necessary to implement CFD techniques. Moreover, it is simple to parallelize as the lattices used are essentially regular. However, one is rapidly tempted to forget that the implementation is not only limited to a non-modular code. Many progresses are achieved every day in software engineering. Indeed, they should be used more systematically in order to produce code of better quality and thus to create an osmosis between physicists and computer scientists.

7.3 Further work

As in the majority of the PhD thesis, answers to the initial problem raise new questions. Let us explain some of them and give answering clues.

Our new boundary condition (mass conserving boundary condition) was introduced considering three different boundary orientations. The others still have to be established in order to be able to simulate flows of any geometry. These orientations do not seem to present any special difficulty. The flow around a cylinder is probably a good candidate to test the validity of our condition for any orientation. Giving a Reynolds number, reattachment lengths could be measured and compared to expected lengths obtained by real experiments and by LB simulations considering other boundary conditions (e.g. bounce-back).

We saw some flow settlement conditions. For efficiency reasons, setting the flow through the local equilibrium distribution function is the most used condition. However, as we saw with a simple example, velocity and density profiles are not perfectly set. More work is necessary to obtain a better condition which properly and, if possible, quickly sets a flow.

Too often, one hears that really high Reynolds numbers ($Re > 10^6$) can be reached if a model of turbulence is used. Even if one uses the Smagorinsky turbulence model, it seems clear that a null viscosity is not reachable. Hence, there exist a viscosity limit. In this document, we gave some clues about how to determine this limit. So it would be interesting to accurately determine this limit. Moreover, we argue that a large enough system is able to simulate all scales, thus we may compare a direct numerical simulation with a smaller one

using the Smagorinsky model of turbulence. The turbulent viscosity obtained with the small system could be compared to the physical viscosity of the large one. The Smagorinsky constant could be, by the way, determined.

We expressed a new algorithm to refine lattices which was employed and compared to existing models considering a simple example. It could be interesting to use it on a more complicated example such as a flow around a cylinder.

We proposed a theory at a mesoscopic scale which explains the meandering process. We exhibited some preliminary results indicating the validity of our theory. However, we did not simulate the whole process starting from a straight channel. Such a simulation would be the first numerical model simulating this amazing process at a mesoscopic scale.

All along this document, lattice Boltzmann models have shown their ability to efficiently deal with tough problems where other methods struggle or fail. Although these numerical models are not completely mastered, the simulations performed above allows us to foresee a promising future for them.

Appendix A

Publications

A.1 Description

During this work, several documents were published in journals or presented in conferences. In this appendix, we introduce them and give their full references as they are not present in the general *Bibliography* located at the end of the present document.

The environment PELABS was first presented on the HPCN-99 (High Parallel Computers and Network) conference [1]. The paper was later selected for a special issue hosted by the FGCS (Future Generation of Computer Systems) journal where some technical points were added [2]. Our study on cell reorganization in memory was presented at RenPar-2000 (Rencontres du Parallélisme) [3].

Our first attempts to simulate scour formation under submarine pipelines was the subject of a paper presented at the DSFD99 (Discrete Simulation of Fluid Dynamics) conference [4]. More advanced results were discussed on the HydroInformatics-2000 conference [5]. To conclude this application, we published a paper with many details to JCP (Journal of Computational Physics) [6].

Our preliminary results on meandering river simulations were presented at the DSFD-2000 conference as a poster [7] and, through the CCP-2001 (Conference on Computation Physics), published in the Computer Physics Communications journal [8].

Our new boundary condition and our new mesh refinement algorithm give rise to two papers [9, 10]. Moreover, these lattice Boltzmann improvements will be presented at the DSFD-2002 conference [11].

Besides virtual rivers, we were interested in other complex systems such as ant colonies [12] and urban traffic flow [13, 14, 15] which are implemented in PELABS.

A.2 References

[1] Alexandre Dupuis and Bastien Chopard. An efficient and reusable parallel library based on a graph partitioning technique. In Peter Sloot, Marian Bubak, Alfons Hoekstra, and Bob Hertzberger, editors, *HPCN Europe 1999*, pages 319–328, Amsterdam, April 1999. Springer-Verlag.

[2] Alexandre Dupuis and Bastien Chopard. An object oriented approach to lattice gas modeling. *Future Generation Computer Systems*, 16(5):523–532, March 2000. Best selected papers from HPCN'99.

[3] Alexandre Dupuis, Paul Albuquerque, and Bastien Chopard. Impact de l'arrangement des données en mémoire dans un environnement parallèle dédié aux automates cellulaires. In H. Guyennet et al., editor, *RenPar'12*, pages 205–210, Besançon, France, June 2000.

[4] B. Chopard, A. Masselot, and A. Dupuis. A lattice gas model for erosion and particles transport in a fluid. In Yu Chen et al., editor, *Proceedings of the LGA'99 conference, Tokyo*, volume 129, pages 167–176. Computer Physics Communications, 2000.

[5] Alexandre Dupuis and Bastien Chopard. Lattice gas simulation of sediment flow under submarine pipelines with spoilers. In *4th International Conference on HydroInformatics*, Cedar Rapids, USA, July 2000.

[6] Alexandre Dupuis and Bastien Chopard. Lattice gas modeling of scour formation under submarine pipeline. *Journal of Computational Physics*, 2002. To appear.

[7] A. Dupuis and B. Chopard. Meandering in rivers: A lattice gas approach. In *poster presented in Discrete Simulation of Fluid Dynamics conference*, Santa Fe, USA, 2000.

[8] B. Chopard and A. Dupuis. Lattice Boltzmann models: an efficient and simple approach to complex flow problems. In Computer Physics Communications, editor, *Conference on Computational Physics*, Aachen, Germany, 2001.

[9] A. Dupuis and B. Chopard. On a lattice Boltzmann mass conserving boundary condition. *IJMPC*, 2002. In preparation.

[10] A. Dupuis and B. Chopard. A general lattice Boltzmann mesh refinement algorithm. *IJMPC*, 2002. In preparation.

[11] B. Chopard and A. Dupuis. Boundary conditions and multigrid approaches revisited. In *Proceedings of DSFD2002, Shangai.*, 2002. To be presented.

[12] P. Albuquerque and A. Dupuis. A parallel ant colony algorithm for sorting and clustering. In *ACRI conference (Cellular Automata for Research and Industry)*, 2002. To be presented.

[13] Bastien Chopard, Alexandre Dupuis, and Pascal O. Luthi. A cellular automata model for urban traffic and its application to the city of geneva. In M.Schreckenberg and D.E.Wolf, editors, *Workshop on Traffic and Granular Flow '97*, pages 153–168, Duisburg, october 1997. Springer.

[14] Alexandre Dupuis and Bastien Chopard. Parallel simulation of traffic in Geneva using cellular automata. *Parallel and Distributed Computing Practices*, 1(3):79–92, September 1998.

[15] A. Dupuis and B. Chopard. Cellular Automata of traffic: a model for the city of Geneva. *Network and Spatial Economics*, 2001.

Appendix B

The Verbois reservoir flush

B.1 Description

In this appendix, we focus our attention on a flush which occurred in May 2000 in one of the reservoirs of a European river: the Rhône. This reservoir is located in Verbois, a place in the Geneva country. This operation took 6 days and consisted of three steps: draining (2 days), flushing (3 days) and filling up (1 day).

Dam managers have to periodically flush the reservoir which is formed by the river upstream of the dam. This part is sometimes almost at rest when the water gates are almost closed. Hence, the resulting weak current implies that sediments transported by the river are more subject to deposition (e.g. 400 000 m^3 of sediments are deposited each year in the Verbois reservoir). After a few years, the reservoir has to be flushed, resulting in the transport of the deposit beyond the dam. Otherwise the river bed would change, thus implying some undesirable floods.

The draining step consists in opening the water gates of the Verbois dam and to close the ones of the Seujet dam which is far away upstream the reservoir. It implies a decrease of the river level (16 meters in this case). Then, the flushing step begins by largely opening the water gates of the Seujet dam producing a flow with strong current. This flow will easily erode the reservoir deposit and will transport it beyond the dam. Finally, one closes the water gates of the Verbois dam so as to fill up the reservoir.

B.2 Illustration

This section illustrates the May 2000 flush of the Verbois reservoir. Figure B.1 presents the river as it was before the flushing process.

Figure B.2 and B.3 presents the river during the flush. We observe the low level of the river. This event is an exceptional chance to see the bed of a river *without* the river. Fortunately, the shape of the bed is in agreement with the ones

Figure B.1: The Rhône before the May 2000 flush. On the left and on the right are presented views of the river upstream and downstream the Peney bridge where the photographs were taken.

presented in subsection 5.3.2.

Figure B.2: The Rhône during the may 2000 flush. These photographs are views taken upstream the Peney bridge.

During the flushing process, the concentration of sediments in the water is high. This is illustrated in figure B.4 where a downstream view of the dam is presented. We observe the muddy water flowing through the dam.

Figure B.3: The Rhône during the May 2000 flush. These photographs are views taken downstream the Peney bridge.

Figure B.4: Muddy water flowing through the Verbois dam.

Appendix C

PELABS: a complete example

C.1 Description

In this appendix we list a complete application as an example. It consists in simulating a flow around a cylinder. The flow is modeled by a D2Q9 LB fluid on a 120×40 lattice. The cylinder is located horizontally at 50 sites from the left and at 20 sites from the bottom. Its diameter is equal to 10 lattice sites. The flow is settled by imposing a constant velocity profile at the inlet (x=0). The simple bounce-back boundary condition is considered around the cylinder. The relaxation time $\tau = 1$.

The following files are listed below:

- `flow.dat` containing the application parameters.
- `flow.cc`, which is the main program, instantiating the model and its cells.
- `RiverModel.hh` and `RiverModel.cc` describing the model.
- `RiverCell.hh` and `RiverCell.cc` implementing the cells used by the model.

C.2 `flow.dat`

```
120             % dx
40              % dy
50              % xr
20              % yr
10              % diameter
0.1             % uEntry
5000            % numberOfIteration
1.0             % tau
100             % step
```

C.3 `flow.cc`

```
#include "RiverModel.hh"
```

```
#include "String.hh"
#include "RT_Timer.hh"
#include "LatticeShape2D.hh"

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

int main(int argc, char **argv)
{
  int numberOfIteration,startMesure;
  float uEntry;
  float dx,dy,diameter,xr,yr;
  float tau;
  int step;
  // Creates the communication toolbox
  ComBox comBox;
  comBox.start(2,"flow",argc,argv,verbose,"hosts.dat",circularContinue);
  // Opens the parameter file
  ifstream inputFile;
  inputFile.open("flow.dat");
  if (!inputFile) {
    cout << "Can't open the input file flow.dat" << endl;
    return 1;
  }
  // Reads the parameters
  String endOfLine;
  inputFile >> dx >> endOfLine;
  inputFile >> dy >> endOfLine;
  inputFile >> xr >> endOfLine;
  inputFile >> yr >> endOfLine;
  inputFile >> diameter >> endOfLine;
  inputFile >> uEntry >> endOfLine;
  inputFile >> numberOfIteration >> endOfLine;
  inputFile >> tau >> endOfLine;
  inputFile >> step >> endOfLine;
  // Creates the shape
  LatticeShape2D shape(dx,dy,finiteBorder,wraparoundBorder);
  // Sets the mask
  Matrix<int> mask(dx,dy);
  mask=0;
  int radius=diameter/2;
  int ix,iy,r;
  for(ix=0;ix<dx;ix++)
    for (iy=0;iy<dy;iy++) {
      r=(ix-xr)*(ix-xr)+(iy-yr)*(iy-yr)-radius*radius;
      if (r <= 0) mask.set(ix,iy,1);
    }
  shape.setMask(mask);
  // Instantiates the lattice
  RiverModel lattice(shape,uEntry,tau);
  lattice.init();
  // Starts the time
  int iter=0;
  RT_Timer timer;
  timer.start();
  // Main loop
  while (1) {
    // Applies the dynamics
    lattice.collision();
    lattice.sendBoundaries();
    lattice.propagation();
    iter++;
    // After step iteration, gives some informations
```

```
    if (iter % step == 0) {
      timer.stop();
      if (comBox.myPE() == 0) {
        cout << "Iter no. " << iter << " nbWaterPart = "
             << lattice.getNbWaterParticle()
             << " t= " << timer.elapsed() << " [s] (" << iter << ")." << endl;
      }
      else
        lattice.getNbWaterParticle();
      timer.reset();
      timer.start();
    }
    // Breaks the loop if the number of iterations have been done
    if (iter >= numberOfIteration) break;
  }
  // Write some results
  lattice.writeResults();
  // Terminates the parallel session
  comBox.terminate();
  return 0;
}
```

C.4 RiverModel.hh

```
#ifndef RIVERMODEL_HH
#define RIVERMODEL_HH

#include "RiverCell.hh"
#include "D2Q9Lattice.hh"
#include "Polygon.hh"
#include "Random.hh"
#include "Vector.hh"
#include "Bool.hh"

class RiverModel: public D2Q9Lattice
{
public:
  RiverModel(LatticeShape2D &shape, const float uEntry,const float tau);
  ~RiverModel();
  void init();
  void propagation();
  void collision();
  virtual VCell** allocateCells(const int nbCell, bool temporary=false);
  virtual VCell** deallocateCells(const int nbCell, VCell** cells,
                                  const int firstCellToDelete=0);
  virtual VCell** getCells() const;
  WaterType getNbWaterParticle();
  void writeResults();
private:
  RiverCell **localCells_;
  Random rand_;
  float uEntry_;
  float tau_;
};
#endif
```

C.5 RiverModel.cc

```
#include "RiverModel.hh"
#include "conditions.hh"
```

```
#include <math.h>
//=======================================================
RiverModel::RiverModel(LatticeShape2D &shape, const float uEntry,
       const float tau)
{
  tau_=tau;
  uEntry_=uEntry;
  int nreg=10;
  localCells_=(RiverCell **) initLattice(shape,false,nreg);
}
//=======================================================
RiverModel::~RiverModel()
{
}
//=======================================================
void RiverModel::init()
{
  int i;
  for (i=0;i<nbCell_;i++) localCells_[i]->allocateRestOfCellAttributes();
  int k;
  for (i=0;i<nbCell_;i++) {
    *localCells_[i]=0.0;
    for (k=0;k<9;k++)
      localCells_[i]->setWaterParticle(k,1/9.0);
  }
}
//=======================================================
void RiverModel::propagation()
{
  int i,k,n;
  WaterType pW;
  // Sets the temporary state to zero
  for (i=0;i<nbCell_;i++)
    if (localCells_[i]->getCellType() != boundary)
      localCells_[i]->setTempToZero();
  for (i=0;i<nbCell_;i++) {
    if (localCells_[i]->getCellType() != boundary) {
      localCells_[i]->setWaterParticle(0,localCells_[i]->getWaterParticle(0),true);
      for (k=1;k<9;k++) {
        n=localCells_[i]->getNeighbor(k);
        pW=localCells_[i]->getWaterParticle(k);
        // There is no neighbor -> reflecting
        if (n == -1) localCells_[i]->addWaterParticle(oppositeOf(k),pW,true);
        else localCells_[n]->addWaterParticle(k,pW,true);
      }
    }
  }
  receiveBoundaries();
  int ii;
  for (ii=0;ii<nbIndexesOfBoundaryCells_;ii++) {
    i=indexesOfBoundaryCells_[ii];
    for (k=1;k<9;k++) {
      n=localCells_[i]->getNeighbor(k);
      pW=localCells_[i]->getWaterParticle(k);
      if (n != -1) localCells_[n]->addWaterParticle(k,pW,true);
    }
  }
  // Switchs
  localCells_[0]->switchState();
}

//=======================================================
void RiverModel::collision()
{
  int i,p,n,y;
```

```
  WaterType density;
  WaterType u[2];
  for (i=0;i<nbCell_;i++) {
    if (localCells_[i]->getCellType() != boundary) {
      if (localCells_[i]->getCellType() == border) {
      }
      else if (localCells_[i]->getX() == 0) {
        density=localCells_[i]->computeWaterDensity();
        u[0]=uEntry_;;
        u[1]=0.0;
        localCells_[i]->update(density,u);
      }
      else {
        density=localCells_[i]->computeWaterDensity();
        localCells_[i]->computeWaterVelocity(density,u);
        localCells_[i]->update(density,u);
      }
    }
  }
}
//=======================================================
VCell** RiverModel::allocateCells(const int nbCell, bool temporary)
{
  RiverCell **localCells;
  int i;
  localCells=new RiverCell*[nbCell];
  for (i=0;i<nbCell;i++)
    localCells[i]=new RiverCell(c,uEntry_,tau_,shape_->getDx(),shape_->getDy());
  if (temporary == false) {
    localCells_=localCells;
    nbCell_=nbCell;
  }
  return (VCell **) localCells;
}
//=======================================================
VCell** RiverModel::deallocateCells(const int nbCell, VCell **cells,
    const int firstCellToDelete)
{
  RiverCell **realCells=(RiverCell **) cells;
  int i;
  for (i=firstCellToDelete;i<nbCell;i++) delete realCells[i];
  if (firstCellToDelete == 0) {
    delete [] realCells;
    return 0;
  }
  else {
    RiverCell **tempCell=new RiverCell*[firstCellToDelete];
    for (i=0;i<firstCellToDelete;i++) tempCell[i]=realCells[i];
    delete [] realCells;
    return (VCell **) tempCell;
  }
}
//=======================================================
WaterType RiverModel::getNbWaterParticle()
{
  WaterType nbPartLocal=0.0;
  WaterType nbPart=0.0;
  int i,k;
  for (i=0;i<nbCell_;i++)
    if (localCells_[i]->getCellType() != boundary)
      for (k=0;k<9;k++)
        nbPartLocal+=localCells_[i]->getWaterParticle(k);
  if (comBox_.myPE() == 0) {
    WaterType  tmp;
    for (k=1;k<comBox_.nbPE();k++) {
```

```
#ifdef DOUBLE_PRECISION
      comBox_.recvDouble(k,&tmp,1);
#else
      comBox_.recvFloat(k,&tmp,1);
#endif
      nbPart+=tmp;
    }
    nbPart+=nbPartLocal;
  }
#ifdef DOUBLE_PRECISION
  else comBox_.sendDouble(0,&nbPartLocal,1);
#else
  else comBox_.sendFloat(0,&nbPartLocal,1);
#endif
  return nbPart;
}
//=======================================================
VCell** RiverModel::getCells() const
{
  require ("localCells != 0", (localCells_ != 0));
  return (VCell **) localCells_;
}
//=======================================================
void RiverModel::writeResults()
{
  int i,p;
  RiverCell **cells;
  int nbCell;
  WaterType density,u[2];
  ComBuf buf;
  buf.init(1500000);
  // Receives sublattices and computes the mean velocities
  if (comBox_.myPE() == 0) {
    for (p=0;p<comBox_.nbPE();p++) {
      if (p == 0) {
        cells=localCells_;
        nbCell=nbCell_;
      }
      else {
        buf.recvFrom(p);
        buf.unpack(nbCell);
        cells=(RiverCell **) allocateCells(nbCell,true);
        for (i=0;i<nbCell;i++)
          cells[i]->allocateRestOfCellAttributes();
        for (i=0;i<nbCell;i++)
        cells[i]->unpackFrom(buf);
      }
      int x,y;
      for (i=0;i<nbCell;i++) {
        if (cells[i]->getCellType() != boundary){
          density=cells[i]->computeWaterDensity();
          cells[i]->computeWaterVelocity(density,u);
          x=cells[i]->getX();
          y=cells[i]->getY();
        }
      }
      if (p != 0) deallocateCells(nbCell, (VCell **) cells);
    }
  }
  // ---------------------------------------
  // if PE != 0
  else {
    // Packs the cells
    buf.init();
    buf.pack(nbCell_);
```

```
    for (i=0;i<nbCell_;i++)
      localCells_[i]->packInto(buf);
    // Sens it to the PE 0
    buf.sendTo(0);
  }
}
```

C.6 RiverCell.hh

```
#ifndef RIVERCELL_HH
#define RIVERCELL_HH

#include "VCell.hh"
#include "Vector.hh"

#ifdef DOUBLE_PRECISION
#define WaterType double
#else
#define WaterType float
#endif

class RiverCell : public VCell
{
public:
  RiverCell(float c[9][2], const WaterType uEntry,
            const WaterType tau, const int dx, const int dy);
  ~RiverCell();
  virtual unsigned char getColor() const;
  virtual VCell& operator=(const VCell &cell);
  virtual RiverCell& operator=(const RiverCell &cell);
  virtual RiverCell& operator=(const WaterType particle);
  virtual void setNeighbor(const int direction, const int neighbor);
  virtual int getNeighbor(const int direction) const;
  WaterType getWaterParticle(const int direction, const bool getTempState=false) const;
  void setWaterParticle(const int direction, const WaterType particle,
                        const bool assignToTempState=false);
  void addWaterParticle(const int direction, const WaterType particle,
                        const bool assignToTempState=false);
  void subWaterParticle(const int direction, const WaterType particle,
                        const bool assignToTempState=false);
  WaterType computeWaterDensity(const bool considerTempState=false) const;
  void computeWaterVelocity(const WaterType density, WaterType u[3]) const;
  virtual int getNbNeighbor() const;
  void update(const WaterType density, const WaterType u[3]);
  void allocateRestOfCellAttributes();
#ifdef PARALLEL
  virtual void packInto(ComBuf &buf);
  virtual void unpackFrom(ComBuf &buf);
  virtual void packParticlesInto(ComBuf &buf);
  virtual void unpackParticlesFrom(ComBuf &buf);
  virtual int getSizeOfOnePackParticle() const;
#endif
  void switchState();
  void setTempToZero();
private:
  WaterType *water_;
  int neighbors_[9];
  static WaterType c_[9][2];
  static WaterType tau_;
  static WaterType uEntry_;
  static int dx_;
  static int dy_;
```

```
  static int switch_;
};
//=======================================================
inline void RiverCell::setNeighbor(const int direction, const int neighbor)
{
  require ("0 <= direction <= 8", ((direction >= 0) && (direction <= 8)));
  neighbors_[direction]=neighbor;
}
//=======================================================
inline int RiverCell::getNeighbor(const int direction) const
{
  require ("0 <= direction <= 8", ((direction >= 0) && (direction <= 8)));
  return neighbors_[direction];
}
// ======================================================================
inline WaterType RiverCell::getWaterParticle(const int direction, const bool getTempState) const
{
  require ("0 <= ((switch_+9)%18)+direction < 18",
           ((((switch_+9)%18)+direction >= 0) && (((switch_+9)%18)+direction < 18)));
  if (getTempState) return water_[((switch_+9)%18)+direction];
  else return water_[switch_+direction];
}
// ======================================================================
inline void RiverCell::setWaterParticle(const int direction,
const WaterType particle, const bool assignToTempState)
{
  require ("0 <= ((switch_+9)%18)+direction < 18",
           ((((switch_+9)%18)+direction >= 0) && (((switch_+9)%18)+direction < 18)));
  if (assignToTempState) water_[((switch_+9)%18)+direction]=particle;
  else water_[switch_+direction]=particle;
}
// ======================================================================
inline void RiverCell::addWaterParticle(const int direction,
const WaterType particle, const bool assignToTempState)
{
  require ("0 <= ((switch_+9)%18)+direction < 18",
           ((((switch_+9)%18)+direction >= 0) && (((switch_+9)%18)+direction < 18)));
  if (assignToTempState) water_[((switch_+9)%18)+direction]+=particle;
  else water_[switch_+direction]+=particle;
}
// ======================================================================
inline void RiverCell::subWaterParticle(const int direction,
const WaterType particle, const bool assignToTempState)
{
  require ("0 <= ((switch_+9)%18)+direction < 18",
           ((((switch_+9)%18)+direction >= 0) && (((switch_+9)%18)+direction < 18)));
  if (assignToTempState) water_[((switch_+9)%18)+direction]-=particle;
  else water_[switch_+direction]-=particle;
}
//=======================================================
inline WaterType RiverCell::computeWaterDensity(const bool considerTempState) const
{
  if (considerTempState) {
    int s=(switch_+9)%18;
    return (water_[s+0]+
    water_[s+1]+water_[s+2]+
    water_[s+3]+water_[s+4]+water_[s+5]+
    water_[s+6]+
    water_[s+7]+water_[s+8]);
  }
  else
    return (water_[switch_+0]+
    water_[switch_+1]+water_[switch_+2]+
    water_[switch_+3]+water_[switch_+4]+water_[switch_+5]+
    water_[switch_+6]+
```

```
    water_[switch_+7]+water_[switch_+8]);
}
//=======================================================
inline int RiverCell::getNbNeighbor() const
{
  return 8;
}
//=======================================================
inline void RiverCell::switchState()
{
  switch_=(switch_+9)%18;
}
#ifdef PARALLEL
//=======================================================
inline int RiverCell::getSizeOfOnePackParticle() const
{
  return 10*sizeof(water_[0]);
}
#endif
#endif
```

C.7 RiverCell.cc

```
#include "RiverCell.hh"
#include "iostream.h"
#include "D2Q9Lattice.hh"
#include <math.h>
WaterType RiverCell::c_[9][2];
WaterType RiverCell::tau_=0.5;
WaterType RiverCell::uEntry_=0.1;
int RiverCell::dx_=0;
int RiverCell::dy_=0;
int RiverCell::switch_=0;
//=======================================================
RiverCell::RiverCell(float c[9][2], const WaterType uEntry, const WaterType tau,
                     const int dx, const int dy)
{
  uEntry_=uEntry;
  tau_=tau;
  int k;
  for (k=0;k<9;k++) {
    neighbors_[k]=-1;
    c_[k][0]=c[k][0];  c_[k][1]=c[k][1];
  }
  dx_=dx;
  dy_=dy;
  water_=0;
}
//=======================================================
RiverCell::~RiverCell()
{
  if (water_ != 0) delete [] water_;
}
//=======================================================
unsigned char RiverCell::getColor() const
{
  return 0;
}
//=======================================================
VCell& RiverCell::operator=(const VCell &cell)
{
  int k;
```

```
  for (k=0;k<9;k++) {
    if (water_ != 0)
      setWaterParticle(k,((RiverCell *) &cell)->getWaterParticle(k));
    setNeighbor(k,cell.getNeighbor(k));
  }
  this->setCellType(cell.getCellType());
  this->setX(cell.getX());
  this->setY(cell.getY());
  return *this;
}
//=======================================================
RiverCell& RiverCell::operator=(const RiverCell &cell)
{
  int k;
  for (k=0;k<9;k++) {
    if (water_ != 0)
      setWaterParticle(k,cell.getWaterParticle(k));
    setNeighbor(k,cell.getNeighbor(k));
  }
  this->setCellType(cell.getCellType());
  this->setX(cell.getX());
  this->setY(cell.getY());
  return *this;
}
//=======================================================
RiverCell& RiverCell::operator=(const WaterType particle)
{
  int k;
  if (water_ != 0)
    for (k=0;k<9;k++)
      setWaterParticle(k,particle);
  return *this;
}
//=======================================================
void RiverCell::computeWaterVelocity(const WaterType density, WaterType u[2]) const
{
  u[0]=0.0;
  u[1]=0.0;
  int i;
  for (i=1;i<9;i++) {
    u[0]+=c_[i][0]*water_[switch_+i];
    u[1]+=c_[i][1]*water_[switch_+i];
  }
  u[0]=u[0]/density;
  u[1]=u[1]/density;
}
//=======================================================
void RiverCell::update(const WaterType density, const WaterType u[2])
{
  Vector<WaterType> dif(9);
  WaterType feq[9];
  int k;
  WaterType prod,norm2,tauTot;
  const WaterType t0=4.0/9.0;
  const WaterType t1=1.0/9.0;
  const WaterType t2=1.0/36.0;
  norm2=u[0]*u[0]+u[1]*u[1];
  feq[0]=t0*density*(1.0-1.5*norm2);
  for (k=1;k<9;k++) {
    prod=(c_[k][0]*u[0]+c_[k][1]*u[1])*3.0;
    if (k < 5)
      feq[k]=t1*density*(1.0+prod*(1.0+prod*0.5)-norm2*1.5);
    else
      feq[k]=t2*density*(1.0+prod*(1.0+prod*0.5)-norm2*1.5);
  }
```

```
  if (getX() == 0) tauTot=1.0;
  else tauTot=1/tau_;
  for (k=0;k<9;k++)
    water_[switch_+k]+=tauTot*(feq[k]-water_[switch_+k]);
}
//=======================================================
void RiverCell::allocateRestOfCellAttributes()
{
  water_=new WaterType[18];
  int k;
  for (k=0;k<18;k++) water_[k]=0.0;
}
//=======================================================
void RiverCell::setTempToZero()
{
  int k;
  int s=(switch_+9)%18;
  for (k=0;k<9;k++) water_[s+k]=0.0;
}
#ifdef PARALLEL
//=======================================================
void RiverCell::packInto(ComBuf &buf)
{
  packParticlesInto(buf);
  buf.pack(&neighbors_[1],8);
  char type=getCellType();
  buf.pack(type);
  buf.pack(getX());
  buf.pack(getY());
}
//=======================================================
void RiverCell::unpackFrom(ComBuf &buf)
{
  unpackParticlesFrom(buf);
  neighbors_[0]=-1;
  int size=8;
  int *pointer=&neighbors_[1];
  buf.unpack(pointer,size,false);
  char type;
  buf.unpack(type);
  setCellType((CellType) type);
  float x,y;
  buf.unpack(x);
  buf.unpack(y);
  setX(x);
  setY(y);
}
//=======================================================
void RiverCell::packParticlesInto(ComBuf &buf)
{
  buf.pack(&water_[switch_+0],9);
}
//=======================================================
void RiverCell::unpackParticlesFrom(ComBuf &buf)
{
  WaterType *pointerF=&water_[switch_];
  int n=9;
  buf.unpack(pointerF,n,false);
}
#endif
```

Bibliography

[1] Stephen B. Pope. *Turbulent flows*. Cambridge University Press, 2000.

[2] F. Abraham, D. Brodbeck, R. Rafey, and W. Rudge. Instability dynamics of fracture: a computer simulation investigation. *Physical Review Letters*, 73, 1994.

[3] K Nagel, M Rickert, and C L Barrett. Large-scale traffic simulations. In J. M. L. M. Palma and J. Dongarra, editors, *Vector and Parallel Processing – VECPAR'96*, volume 1215 of *Lecture Notes in Computer Science*, pages 380–402. Springer, 1997.

[4] R.J. Garde and K.G. Ranga Raju. *Mechanics of Sediment Transportation and Aluvial Stream Problems*. Wiley eastern limited, 1977.

[5] Ye Mao. Seabed scour under pipelines. In *Seventh International Conference on Offshore Mechanics and Arctic Engineering*, pages 33–38, Houston, Texas, February 1988.

[6] Richard Whitehouse. *Scour at marine structures*. Thomas Telford Publications, London, 1998.

[7] A.J.C. Ladd. Numerical simulation of particulate suspensions via a discretized boltzmann equation. *J. Fluid Mech*, 271:285,310, 1994.

[8] D. Mueller. Using triangulations in computer simulations of granular media. *Mathematical Modelling and Scientific Computing*, 6, 1996.

[9] Bastien Chopard and Michel Droz. *Cellular Automata Modeling of Physical Systems*. Cambridge University Press, 1998.

[10] A. Dupuis and B. Chopard. Cellular Automata of traffic: a model for the city of Geneva. *Network and Spatial Economics*, 2001.

[11] Pascal O. Luthi. *Lattice Wave Automata*. PhD thesis, University of Geneva, 1998. `http://cui.unige.ch/spc/phds/vague.pdf`.

[12] Sauro Succi. *The Lattice Boltzmann Equation, For Fluid Dynamics and Beyond*. Oxford University Press, 2001.

[13] D.J. Tritton. *Physical fluid dynamics*. Clarendon Press, 1988.

[14] Alexander J. Smits. *A Physical Introduction to Fluid Mechanics*. John Wiley & Sons, inc., 2000.

[15] Uriel Frisch. *Turbulence: the legacy of A.N. Kolmogorov*. Cambridge University Press, 1995.

[16] Mark Nelkin. Resource letter tf-1: Turbulence in fluids. *American Journal of Physics*, 68(4):310–318, 2000.

[17] M. Van Dyke. *An Album of Fluid Motion*. The Parabolic Press, Standford, CA, 1982.

[18] A.N. Kolmogorov. The local structure of turbulence in incompressible viscous fluid for very large reynolds number. *Dokl. Akad. Nauk SSSR*, 30:9–13, 1941. (reprinted in Proc. R. Soc. Lond. A **434**, 9-13 (1991)).

[19] A.N. Kolmogorov. On degeneration (decay) of isotropic turbulence in an incompressible viscous liquid. *Dokl. Akad. Nauk SSSR*, 31:538–540, 1941.

[20] A.N. Kolmogorov. Dissipation of energy in locally isotropic turbulence. *Dokl. Akad. Nauk SSSR*, 32:16–18, 1941. (reprinted in Proc. R. Soc. Lond. A **434**, 15-17 (1991)).

[21] A.N. Kolmogorov. A refinement of previous hypotheses concerning the local structure of turbulence in a viscous incompressible fluid at high reynolds number. *Journal of Fluid Mechanics*, 13:82–85, 1962.

[22] F. Anselmet, Y. Gange, and E.J. Hopfinger. High-order velocity structure functions in turbulent shear flows. *Journal of Fluid Mechanics*, 140:63–89, 1984.

[23] J. Boussinesq. Essai sur la théorie des eaux courantes. *Mém. prés. par div. savants à l'Acad. Sci*, 23:1–680, 1877.

[24] H. Le, P. Moin, and J. Kim. Direct numerical simulation of turbulent flow over a backward-facing step. *J. Fluid Mech.*, 330:349–374, 1997.

[25] Joseph Smagorinsky. General circulation experiments with the primitive equations: I. the basic equations. *Mon. Weather Rev.*, 91:99–164, 1963.

[26] Boris Galperin and Steven A. Orszag. *Large eddy simulation of complex engineering and geophysical flows*. Cambridge Univeristy Press, 1993.

[27] D.H. Rothman and S. Zaleski. Lattice-gas models of phase separation: interface, phase transition and multiphase flows. *Rev. Mod. Phys*, 66:1417–1479, 1994.

[28] J.-P. Boon, editor. *Advanced Research Workshop on Lattice Gas Automata Theory, Implementations, and Simulation*, volume 68 (3/4). J. Stat. Phys, 1992.

[29] J. Hardy, Y. Pomeau, and O. de Pazzis. Time evolution of a two-dimensional model system. I. Invariant states and time correlation functions. *J. Math. Phys.*, 14:1746, 1973.

[30] U. Frisch, B. Hasslacher, and Y. Pomeau. Lattice-gas automata for the Navier-Stokes equation. *Phys. Rev. Lett.*, 56:1505, 1986.

[31] G. Doolen, editor. *Lattice Gas Method for Partial Differential Equations*. Addison-Wesley, 1990.

[32] G. Zanetti. Hydrodynamics of lattice-gas automata. *Phys. Rev. A*, 40:1539–1548, 1989.

[33] F. Higuera, J. Jimenez, and S. Succi. Boltzmann approach to lattice gas simulations. *Europhys. Lett*, 9:663, 1989.

[34] F. Higuera, J. Jimenez, and S. Succi. Lattice gas dynamics with enhanced collision. *Europhys. Lett*, 9:345, 1989.

[35] H. Chen, S Chen, and W.H. Matthaeus. Recovery of the navier-stokes equations using a lattice gas boltzmann method. Los Alamos Nat.Lab archives http://xyz.lanl.gov/form/comp-gas/, 1994.

[36] P. Bathnagar, E.P. Gross, and M.K. Krook. A model of collision processes in gases. *Physical Review Letters*, 94(511), 1954.

[37] B. Chopard, P. Luthi, and A. Masselot. Cellular automata and lattice Boltzmann techniques: An approach to model and simulate complex systems, 1998. http://cui.unige.ch/∼chopard/CA/Book/related.html.

[38] Hudong Chen, Shiyi Chen, and W.H. Matthaeus. Recovery of navier–stokes equations using a lattice-gas boltzmann method. *Phys. Rev. A*, 45:R5339–42, 1992.

[39] Y.H. Qian, S. Succi, and S.A. Orszag. Recent advances in lattice boltzmann computing. In D. Stauffer, editor, *Annual Reviews of Computational Physics III*, pages 195–242. World Scientific, 1996.

[40] J. Sterling and S. Chen. Stability analysis of Lattice Boltzmann methods. *J. Comp. Phys.*, 1994.

[41] Pierre Lallemand and Li-Shi Luo. Theory of the lattice Boltzmann method dispersion, dissipation, isotropy, Galilean invariance, and stability. Technical Report TR-2000-17, ICASE, Nasa Research Center, 2000.

[42] Manfred Krafczyk, 2000. Private communication.

[43] Dieter Wolf-Gladrow. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models*. Springer-Verlag, 2000. Lecture Notes in Mathematics, volume 1725.

[44] H. Jeffreys. *Cartesian Tensors*. Cambridge University Press, 1965.

[45] D. d'Humières and P. Lallemand. Lattice gas models for 3d hydrodynamics. *Europhys. Lett.*, 2:291, 1986.

[46] Drona Kandhai. *Large Scale Lattice-Boltzmann Simulations*. PhD thesis, University of Amsterdam, 1999.

[47] S. Chen and G.D. Doolen. Lattice Boltzmann methods for fluid flows. *Annu. Rev. Fluid Mech.*, 30:329, 1998.

[48] Donald Ziegler. Boundary conditions for lattice Boltzmann simulations. *Journal of Statistical Physics*, 71(5/6):1171–1177, 1993.

[49] X. He and L.-S. Luo. Lattice Boltzmann model for the incompressible Navier-Stokes equation. *J. Stat. Phys.*, 88:927–944, 1997.

[50] P. Skordos. Initial and boundary conditions for the lattice Boltzmann method. *Physical Review E*, 48(6):4823–4841, 1993.

[51] H. He and Q. Zou. Analysis and boundary condition of the lattice Boltzmann BGK model with two velocity components. Technical Report LA-UR-95-2293, Los Alamos National Laboratory, 1995. Downloadable from comp-gas/9507002.

[52] Takaji Inamouro, Masato Yoshino, and Fumimaru Ogino. A non-slip boundary condition for lattice Boltzmann simulations. *Phys. Fluids*, 7(12):2928–2930, 1995.

[53] Shiyi Chen, Daniel Martinez, and Renwei Mei. On boundary conditions in lattice Boltzmann methods. *Phys. Fluids*, 8(9):2527–2536, 1996.

[54] Robert Maier, Robert Bernard, and Daryl Grunau. Boundary conditions for the lattice Boltzmann method. *Phys. Fluids*, 8(7):1788–1801, 1996.

[55] Olga Filippova and Dieter Hänel. Grid refinement for lattice-BGK models. *Journal of Computational Physics*, 147, 1998.

[56] Renwei Mei, Li-Shi Luo, and Wei Shyy. An accurate curved boundary treatment in the lattice Boltzmann method. *Journal of Computational Physics*, 155:307–330, 1999.

[57] Olga Filippova, Sauro Succi, Francesco Mazzocco, Cinzio Arrighetti, Gino Bella, and Dieter Hänel. Multiscale lattice Boltzmann schemes with turbulence modeling. *Journal of Computational Physics*, 170:812–829, 2001.

[58] R. Cornubert, D. d'Humières, and D. Levermore. A Knudsen layer theory for lattice gases. *Physica D*, 47(241), 1991.

[59] S. Hou, Q. Zou, S. Chen, G.D. Dolen, and A.C. Cogley. Simulation of cavity flow by the lattice boltzmann method. *Journal of Computational Physics*, 118:329–347, 1995.

[60] M. Reider and J. Sterling. Accuracy of discrete-velocity BGK models for the simulation of the incompressible Navier-Stokes equations. *Comput. Fluids*, 24(4):459–467, 1995.

[61] Q. Zou and X. He. On pressure and velocity boundary conditions for the lattice Boltzmann BGK model. *Phys. Fluids*, 9(6):1591–1598, 1997.

[62] S. Hou, J. Sterling, S. Chen, and G.D. Doolen. A lattice subgrid model for high Reynolds number flows. *Fields Institute Communications*, 6:151–166, 1996.

[63] S. Succi, G. Amati, and R. Benzi. Challenges in lattice boltzmann computing. *J. Stat. Phys.*, 81:5, 1995.

[64] C. Texeira. Incorporating turbulence models into the Lattice Boltzmann method. *Int. J. Mod. Phys. C*, 9(8):1159, 1998.

[65] S. Jovic and D. Driver. Reynolds number effect on the skin friction in seperated flows behind a backward-facing step. Technical Report TM 108807, NASA, 1994.

[66] K. Hanjalic and S. Jakirlic. Contribution towards the second-moment closure modelling of separating turbulent flows. *Computers and Fluids*, 27(2):137–156, 1998.

[67] Alice Nakkasyan. Channel flow simulations with lattice Boltzmann. In *IMACS conference*, Lausanne, Switzerland, 2000.

[68] D.R. Noble, J.G. Georgiadis, and R.O. Buckius. Comparison of accuracy and performance for the lattice Boltzmann and finite volume difference simulations of steady viscous flow. *Int. J. Numer. Meth. Fluids*, 23:1–18, 1996.

[69] N. Satofuka, H. Nishida, and N. Okaza. Comparison of lattice Boltzmann and Navier-Stokes simulations of homogeneous isotropic turbulence. In *XIXth. Int. Cong. of Theoretical and Applied Mechanics*, pages 25–31, Kyoto, Japan, August 1996.

[70] J. Bernsdorf, F. Durst, and M. Schäfer. Comparisson of cellular automata and finite volume techniques for simulation of incompressible flows in complex geometries. *Int. J. Numer. Meth. Fluids*, 29:251–264, 1999.

[71] Olga Filippova and Dieter Hänel. Acceleration of Lattice-BGK schemes with grid refinement. *Journal of Computational Physics*, 165:405–427, 2000.

[72] Ching-Long Lin and Young Lai. Lattice Boltzmann method on composite grids. *Physical Review E*, 62(2):2219–2225, 2000.

[73] *HydroInformatics conference*, Cedar Rapids, USA., July 2000.

[74] Yee-Meng Chiew. Mechanics of local scour around submarine pipelines. *Journal of Hydraulic Engineering*, 116(4):515–529, April 1990.

[75] S.P. Kjeldsen, O. Gjørsvik, K.G. Bringaker, and J. Jacobsen. Local scour near offshore pipelines. In *Second International Conference on Port and Ocean Engineering under Arctic Conditions*, pages 308–331, Iceland, 1973. University of Iceland.

[76] B. Brørs. Numerical modeling of flow and scour at pipelines. *Journal of Hydraulic Engineering*, 125(5):511–523, May 1999.

[77] Fangjun Li and Liang Cheng. Numerical model for local scour under offshore pipelines. *Journal of Hydraulic Engineering*, 125(4):400–406, April 1999.

[78] Nils Olsen and Morten Melaaen. Three-dimensional calculation of scour around cylinders. *Journal of Hydraulic Engineering*, 119(9):1048–1054, September 1991.

[79] Alexandre Masselot. *A new numerical approach to snow transport and deposition by wind: a parallel lattice gas model*. PhD thesis, University of Geneva, March 2000. `http://cui.unige.ch/spc/phds/lbmsnow.pdf`.

[80] B. Chopard, L. Frachebourg, and M. Droz. Multiparticle lattice gas automata for reaction-diffusion systems. *Int. J. of Mod. Phys. C*, 5:47–63, 1994.

[81] O. Marguin. *Abelian Sandpile on a Rectangular Lattice*. PhD thesis, University Claude Bernard, Lyon 1, 1997.

[82] G.J.C.M. Hoffmans and H.J. Verheij. *Scour manual*. A.A. Balkema Publishers, Rotterdam, 1997.

[83] Yee-Meng Chiew. Effect of spoilers on scour at submarine pipelines. *Journal of Hydraulic Engineering*, 118(9):1311–1317, 1992.

[84] E.W. Bijker and W. Leeuwenstein. Interaction between pipelines and the sea bed under the influence of waves and currents. In B. Denness, editor, *Sea Bed Mechanics, IUTAM-IUGG Symp*, pages 235–242, London, September 1984. Graham and Trotman.

[85] B.M. Sumer and J. Fredsøe. A review of wave/current induced scour around pipelines. In *23rd International Coastal Engineering Conference*, volume 3, pages 2839–2852, Venice, Italy, 1992.

[86] B.L. Jensen. Large-scale vortices in the wake of a cylinder placed near a wall. In *Second International Conference on Laser Anemometry - Advances ans applications*, pages 153–163, Strathclyde, UK, September 1987.

[87] Ye Mao. The interaction between a pipeline and an erodible bed. Technical report, Institute of Hydrodynamics and Hydraulic Engineering, Technical University of Denmark, 1986.

[88] L.B. Leopold and M.G. Wolman. River meanders. *Bulletin of the Geological Society of America*, 71:769–794, 1960.

[89] Stephen Lancaster. *A nonlinear River Meandering Model and its Incorporation in as Landscape Evolution Model*. PhD thesis, Oregon State University, 1998.

[90] L. Leopold and M. Wolman. River channel patterns-braided, meandering, and straight. *U.S. Geol. Survey Prof*, pages 39–85, 1957.

[91] S. Schumm, M. Mosley, and W. Weaver. *Experimental Fluvial Geomorphology*. John Wiley Publishers, 1987.

[92] D. Swanson. The importance of fluvial processes and related reservoir deposits. *J. Pet. Technol.*, pages 368–377, April 1993.

[93] S. Leliavsky. An introduction to fluvial hydraulics. Constable and Co., London, 1955.

[94] J. Thomsen. On the origin of windings of rivers in alluvial plains. *Royal Soc. London*, 25:5–6, 1879.

[95] G. Matthes. Basic aspects of stream meanders. *Am. Geophys. Union TRans.*, 22:632–636, 1941.

[96] H. Chang. On the cause of river meandering. In W. White, editor, *Proceedings of International Conference on River Regime*, 1988.

[97] T.R. Thakur. Chain model of river meanders. *Journal of Hydrology*, 12:25–47, 1970.

[98] S. Ikeda, G. Parker, and K. Sawai. Bend theory of river meanders. *J. Fluid Mech.*, 112:363–377, 1981.

[99] P. Blondeaux and G. Seminara. A unified bar-bend theory of river meanders. *J. Fluid Mech.*, pages 449–470, 1985.

[100] G. Parker and E. Andrews. On the time development of menader bends. *J. Fluid Mech.*, 162:1361–1373, 1986.

[101] R.I Ferguson. Distributed periodic model for river meanders. *Earth Surface Processes*, 1:337–347, 1976.

[102] T. Liverpool and S. Edwards. Dynamics of a meandering river. *Physical Review Letters*, 75(16):3016–3019, October 1995.

[103] P.S. Dodds and D.H. Rothman. A unified view of scaling laws for river networks. cont-mat/9808244 v2, January 1999.

[104] R.H.J. Sellin. Towards the identification of flow mechanisms in channels of complex geometry. In *XXIV Congress of IAHR, Madrid*, pages A–521, 1991.

[105] A. Brad Murray and Chris Paola. A cellular model of braided rivers. *Nature*, 371:51–57, Septembre, 1st 1994.

[106] Alan Howard and Thomas Knutson. Sufficient conditions for river meandering: A simulation approach. *Water resources research*, 20(11):1659–1667, November 1984.

[107] A.D. Howard. Simulation model of meandering. In Charles M. Elliott, editor, *River Meandering*, pages 952–963. Am. Soc. Civ. Eng. NY, 1984. Conference Rivers '83, New Orelans, LA.

[108] S.T. Lancaster and R.L. Bras. A simple model of river meandering and its comparison to natural channels. *Hydrological Processes*, 2001. in press.

[109] B. Chopard, P.O. Luthi, and Jean-Frédéric Wagen. A lattice boltzmann method for wave propagation in urban microcells. *IEE Proceedings - Microwaves, Antennas and Propagation*, 144:251–255, 1997.

[110] Frédéric Guidec, Patrice Calégari, and Pierre Kuonen. Parallel irregular software for wave propagation simulation. In *HPCN'97 High-Parallel Computing and Networking*, Lecture Notes in Computer Science, pages 84–94. Springer-Verlag, 1997.

[111] S. Di Gregorio, R. Ringo, W. Spataro, Giandomenico Spezzano, and Domenico Talia. A parallel cellular environment for high performance scientific computing. In H. Liddell at al., editor, *HPCN'96 High-Peformance Computing and Networking*, pages 514–521, Berlin, 1996. Springer-Verlag.

[112] `http://merd.net/pixel/language-study/diagram.html`.

[113] Bjarne Stroustrup. *Le language C++*. International thomson publishing, 1992.

[114] Bertrand Meyer. *Conception et programmation par objets: pour du logiciel de qualité*. Centre National des Lettres, 1991.

[115] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1993.

[116] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: design and analysis of algorithm*. Benjamin/Cummings, 1994.

[117] H. Guyennet et al., editor. *RenPar'12*, Besançon, France, June 2000.

[118] S. Alhir. *UML in a nutshell*. O'Reilly, 1998.

[119] `http://www-unix.mcs.anl.gov/mpi/index.html`.

[120] `http://www.epm.ornl.gov/pvm/pvm_home.html`.

[121] M. Garey, D. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[122] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(1):291–307, 1970.

[123] Roberto Battiti, Alan Bertossi, and R. Rizzi. Randomized greedy algorithms for the hypergraph partitioning problem. In *DIMACS Workshop on Randomization Methods in Algorithm Design*, October 1997.

[124] Roberto Battiti and Alan Bertossi. Greedy, prohibition, and reactive heuristics for graph partitioning. *IEEE Transactions on Computers*, to appear.

[125] M.Laguna, T.A. feo, and H.C. Elrod. A greedy randomized adaptative search procedure for the two-partition problem. *Operations Research*, 42:667–687, 1994.

[126] Thang Nguyen Bui and Byung Ro Moon. Genetic algorithm and graph partitioning. *IEEE Transactions on Computers*, 45(7):841–855, July 1996.

[127] Gregor von Laszewski and Heinz Mühlenbein. Partitioning a graph with a parallel genetic algorithm. In *Parallel problem solving from nature*, pages 165–169, 1991.

[128] D.S. Johnson, C.R. Aragon, L.A. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation. *Operations Research*, 37:865–892, 1989.

[129] Alex Pothen, H.D. Simon, Lien Wang, and Stephen T. Bernard. Towards a fast implementation of spectral nested dissection. In *Supercomputing '92*, pages 42–51, 1992.

[130] George Karypis and Vipin Kumar. A fast and highly quality multilevel scheme for partitioning irregular graphs. Technical Report 95-035, Departement of Computer Science, University of Minnesota, 1995.

[131] Robert Leland and Bruce Hendrickson. An empirical study of static load balancing algorithms. In *Scalable High-Performance Computing Conference (SHPCC'94)*, pages 682–685, 1994.

[132] http://www-users.cs.umn.edu/∼karypis/metis/.

[133] George Karypis and Vipin Kumar. *A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*, November 1997.

[134] http://www.cs.sandia.gov/CRF/chac.html.

[135] http://www.uni-paderborn.de/cs/robsy/party.html.

[136] http://www.labri.u-bordeaux.fr/Equipe/ALiENor/membre/pelegrin/scotch/.