Object-Oriented Framework for Large Scale Air Pollution Models

Anton Antonov

Preface

Generally in all the mathematics, the most difficult problems are the inverse problems. The general aim about air-pollution models is their usage for some strategy or policy for the sake of the society. In other words, according to some meteorological measurements and monitoring of the air pollutants (their emissions) for some period of time (month) in some region (Europe), to point out where have to be reduced the emissions of the air-pollutants according to some cost function, in order to get the level of air-pollution in some prescribed bounds. This is an optimizational problem. Hence, it is an inverse problem. The forward problem is according to the meteorological measurements and given emissions to calculate the level of the air-pollution (the concentrations of the pollutants).

The inverse problem is too big, and it is not possible to be solved now (or at least it is not). More precisely, its questions are tried to be answered by solving a number of forward problems, taken and enumerated by some strategy according to the experts' knowledge in the field. Even the solution of the forward problems is tedious and slow, since their region is very big (Europe), and about 30 - 60 pollutants are considered (i.e. they are too ambitious).

This ambitious task imposes different tradeoffs that should be addressed by a flexible programming implementation of a air-pollution model. The thesis illustrates how such a system can be build using the paradigm of the Object-Oriented Programming (OOP). It also provides the necessary background this approach to be understood by both a student in computer science, who might be quite familiar with OOP, but might lack knowledge in finite elements or chemical kinetics, and by a senior scientist who is familiar enough with the numerics, chemistry and physics, but is not familiar with the OOP.

Acknowledgments

Carrying out a problem-solving research brings together different kinds of intellectual resources. I greatly benefited from the interaction with all of my supervisors.

I would like to thank Per Grove Thomsen for all theoretically inclined conversations we had, for his practical advices, and for his advice to use PETSc.

I would like to thank Claus Bendtsen for the discussions of the object-oriented design of the Danish Eulerian Model, for his help on using C++, and that he kept the research on a strong object-oriented line. I would like to thank him especially, for his advice to use the design patterns and to attend ECOOP'99.

I would like to thank Zahari Zlatev for being patient to explain the different parts of the Danish Eulerian Model, and the way it is used. My PhD studies would be impossible without him – not so many people has developed large scale air pollution models. His help in the "model debugging" in the last programming stages is inestimable.

All my supervisors revised different parts of the thesis manuscript, and made helpful remarks. Claus Bendtsen, especially, revised the framework documentation during its creation, and Zahari Zlatev made very important remarks on the thesis contents – thanks again.

Alexander Karaivanov is another person without whom my PhD studies would be extremely difficult. I would like to thank him for helping me on every issue of system administration, debugging, language subtleness, or hard programming I brought to him. Also, I would like to thank him, and Minka Marinova, for being curious and patient listeners of my tries to explain the different mathematical topics I coped with during my PhD studies.

I would like to thank Steinn Gudmundsson in the way I promised, for the conversations we had, for his help how to use the computer resources at UNI-C, and his advises when I was taking my PhD courses.

I am grateful to Dr. James Sørlie and Mikael Zebbelin Poulsen for being peer interlocutors of our object-oriented conversations, and "we, the real programmers" talks.

Finally, I want to thank the person I was two years ago: if I face now similar amount of challenge of my knowledge, adequacy, entrepreneurship, and ego I will bypass it.

The work was partly supported by the Danish Research Agency, the National Environmental Research Institution, and the Danish Computer Center for Research and Education.

A.A.A.
17 April 2001 Lyngby

Abbreviations

ADS Advection-Diffusion Submodel

APP Air Pollution Problem

CC Computational Context

CS Chemistry Submodel

CPC-DEM Current Production Code of the Danish Eulerian Model

DEM the Danish Eulerian Model

DHL Data Handlers Layer

DP Design Patterns

ERD Entity Relationship Diagram

FE Finite Element

FEM Finite Element Method

GFEM Galerkin Finite Element Method

GFEML Galerkin Finite Element Method Layer

LRTAP Long Range Transport of Air Pollutants

LSAPM Large Scale Air Pollution Model

MG Mesh Grenerator

MGL Mesh Grenerator Layer

MPI Message Passing Interface

NF Normal Form

3NF Third Normal Form

OO Object-Oriented

OODEM the Object-Oriented Danish Eulerian Model

OOP Object-Oriented Programming

PETSc The Portable Extendable Toolkit for Scientific Computations

RDB Relational Data Bases

RSRA Recursive Static-Regridding Algorithm

SIMD Single Instruction Multiple Data

SLRA Static Local Refinements Algorithm

UML Unified Modeling Language

Contents

List of Figures

List of Tables

Chapter 1

Introduction

When we're ready to sing
We step up to the microphones
And it comes out something like this.

– Offspring, "*Intro*",
Conspiracy Of One, 2000

1.1 The air pollution problem

An exceedance of critical air pollution levels may be defined as a situation in which substances that result from human activities are sufficiently high above their normal levels to produce measurable effect on humans, animals, vegetation, or materials [41]. The effects of the air pollution are undesirable.

It is necessary to reduce the concentrations and/or depositions of damaging pollutants to acceptable (or critical) levels, or moreover to keep them after that under these levels. This can be achieved by reducing certain human-made emissions. The reduction is normally an expensive action. Therefore, reduction strategies should be planned carefully.

1.2 What are the air pollution models for?

It is well-known, that the air pollution levels in a given region depend not only on the emission sources located in it, but also on emission sources located outside the region under consideration, and even on sources that are far away from the studied region. This is due to the transboundary transport of air pollutants. The atmosphere is the major medium where pollutants can be transported over long distances. Large-scale air pollution models, in which all relevant physical and chemical processes are adequately described, have to be used to study the transboundary transport. Moreover, these models have to be defined over a *large* space domain. The Danish Eulerian Model (DEM) is an example of a large-scale air pollution model. DEM has been used in different studies concerning air pollution levels in Denmark and Europe [10, 25, 26, 46, 58, 59, 60, 61]. It is defined on a space domain the size of which is $(4800\ km \times 4800\ km)$. This domain contains the whole of Europe together with parts of Africa, Asia, the Arctic regions and the Atlantic Ocean.

The air pollution models are used to answer the following questions [41]:

- What is the contribution of a certain source to the concentration at a certain site?
- What is the most cost-effective strategy for reducing pollutant concentrations in order to satisfy an air quality standard?
- What will be the effect on the air quality in given region if a specific air pollutant emission flux is increased or reduced?

- Where should be placed a new pollutant source (industrial factory, highway, etc.) to minimize the environmental impact?
- What will be the air quality after a certain period?

1.3 Object-oriented programming fits the philosophy of the Danish Eulerian Model

Roughly speaking, there are two types of large scale air pollution models:

- comprehensive models, in which one attempts to describe in detail the physical phenomena that constitute the long-range transport of air pollutants; and
- simple models, in that can be run over large time intervals.

Examples of the two types of models are listed in [58, pages 266-267]. Since more and more additional features are incorporated to the simple models, the gap between the two groups is becoming thiner and thiner.

DEM, designed originally as a simple model, follows this process of growing. The philosophy behind it is clear from the following quotation from the Zlatev's book [58, page 268]:

> The Danish Eulerian Model was in the very beginning a simple model. ... Gradually, many other new modules have been added to the model (as, for example, a chemical scheme with non-linear chemical reactions,...) The development of the Danish Eulerian Model continues. The main idea is to continue to keep the upgraded model well-structured, so that it should be possible both to add and to remove in an easy way some modules when such an action is appropriate. In this way the model will probably (sometime in the future) be upgraded to a comprehensive model. However, this is not the most important aim. As pointed out above, it is also both desirable and important to have the possibility of running the model on long time-intervals and, thus, to be able to study seasonal variations of the concentrations.
>
> By upgrading the model in small steps (i.e. by adding and/or by removing one simple module per step), it should be possible, in principle at least, to determine the most economical set of modules that can be used for a particular task.

The paradigm of the object-oriented programming fits to this philosophy: in this paradigm we can program on an abstract level and provide the desired behavior of the model via different concretizations of the abstractions, on which the model implementation design is based on. There should be no domino effect between the modules, when the functionality of the model is updated or tuned, and in the thesis we will see how we can achieve this.

We should take in account that the simulations based on a large scale air pollution model involve heavy computations because of the large modeling domain and the number of the considered chemical components. These computations result from the rather abstract and subtle notions on which their mathematical algorithms are based on. A program that performs the simulation is, as a mater of fact, a model of these mathematical algorithms. With the object-oriented paradigm the entities, the invariants, and the relations within the subject being modeled, can be reflected in the programming code. If we want to provide an environment, where specific investigations are made, it is natural then to prepare some preliminary code that reflects the principles common for any activity in that area. That preliminary code is called framework. A framework should be very easily tuned, completed, extended to a concrete program that meets the user needs. Clearly, the framework approach for building software covers the philosophy outlined in the quotation above.

1.4 The use of local refinements

As it was said, the domain of DEM contains the whole of Europe together with parts of Africa, Asia, the Arctic regions and the Atlantic Ocean. It is clear that it is difficult to use fine grids during the

discretization of the model, because (i) this leads to very large computational tasks and (ii) the input data (meteorological data and emission data) is normally only available on coarse grids.

Sometimes it is desirable to use finer grids. There are two basic situations in which this is the case:

- **Numerical problems connected with sharp gradients of certain concentrations in some parts of the space domain.** Such problems may occur (i) under specific meteorological conditions, (ii) in the neighborhood of very large emission sources, or (iii) if a combination of the former two cases takes place. Normally, it is most appropriate to apply **dynamic** refinement in the regions when this is needed and when the conditions require such a refinement. Since this kind of refinement has to be applied when the numerical methods are unable to resolve some sharp gradients, some numerical estimates can be used to decide when to refine and how to refine the grid. More details about this kind of refinement, in the case where it is applied to large-scale air pollution models can be found in [47].

- **Need to utilize input data calculated on a finer grid in some part of the space domain.** In some sub-domains of the space domain of the model (some large cities and their neighborhoods, some countries, or parts of countries, etc.) input data on high resolution grids is available. **Static** local refinement can be used in such a case in attempt to exploit better the availability of more accurate data.

In this thesis we will be interested only in the second case, (but some information of how dynamic refinement can be implemented is discussed in Chapter 8). This means that we will assume that we have more detailed information about the input data in a given sub-domain of the model domain, and we will try to exploit this information by refining locally the grid around the given sub-domain. This can be done by using one of the following two approaches:

- **One way nesting.** If this approach is used, then the model is first run over the whole space domain, and the concentrations on the boundary of the sub-domain are stored. After that a second run is performed on the sub-domain of interest by using the stored concentrations as boundary conditions and by utilizing the more accurate input data. This process can be repeated several times (every time using a more refined grid on a smaller sub-domain), see, for example, [1]. In principle, it is not necessary to carry one additional run (or several additional runs when multiple nesting is to be used). One can do the runs together (i.e. after performing a time-step on a larger sub-domain by using a courser grid, to perform the corresponding calculations on a smaller sub-domain by using a finer grid). The advantage of this approach is its conceptual simplicity. This means that it is easy to implement it. However, there could appear some problems connected with the interpolation procedures which have to be used in order to get boundary conditions on the finer grids by using the data calculated on coarser grids. Furthermore, it is difficult to see the effect of using more accurate data in the nested sub-domains on the concentration levels outside the nested sub-domains.

- **Static refinement of the grid in the desired sub-domain.** Variable mesh sizes are used in this approach refining the grid in the desired area. In this way the disadvantages of the method based on one way nesting are in general removed, although some difficulties may arise on the inner boundaries or in the areas of transition from a coarser grid to a finer grid (these problems could be reduced by increasing the area of transition from the coarsest grid to the finest grid). This approach is more complicated than the one way nesting and, thus, it is more difficult to implement it efficiently in a code.

One can also try to refine the grid over the whole domain. In this way, equidistant grids are used and, thus, the implementation of this approach in an existing model is straight-forward. Moreover, one should expect the numerical methods to be more accurate, because the grid-size is reduced over the whole space domain. However, the computational tasks become very large when this approach is used. We will demonstrate this by an example. There is a version of the Danish Eulerian Model defined on a (480×480) equidistant grid. This means that the size of a grid-square is ($10\ km \times 10\ km$) in the refined version. If we compare this version of DEM with the basic version of DEM, which is discretized on a (96×96)

grid, (the size of a grid-square is (50 $km \times 50\ km$) in the later version), then it can easily be seen that the storage requirements and the computational work will be increased by a factor of 25 (assuming here that the two models can be run with the same time-step). In fact, some modules of the model, in which explicit methods are used have to be run with a time-step which is 6 times smaller. For these modules of the model, the computer time is increased by a factor of 150. This example illustrates why an efficient version, in which static local refinement is used, is highly desirable.

1.5 The thesis outline

The thesis describes a typical application of the object-oriented paradigm for software construction to a model based on different scientific subcultures. The basic body of the thesis (without the last two chapters, 8 and 9) can be divided into two parts:

1. A part in which the modeling principles, the mathematical methods, and the test/experiment results are described; that part consists of chapters 2, 3, 4.

2. A part in which the Object-Oriented Danish Eulerian Model (OODEM) framework is described, and the programming paradigm and methodology involved in its design are rationalized; that part consists of the chapters 5, 6, 7.

The derivation of the air pollution mathematical model, the interpretation of the static local refinements, and a comparison of the static local refinements with the static-regridding method are given in Chapter 2.

Since the main contributions are the implementation and the use of local refinements, the Galerkin Finite Element Method (GFEM) suitable for this modeling approach is described in Chapter 3. An analysis of the anisotropies, phase velocities, and group velocities when GFEM is used over (regular) triangular and rectangular grids is also given in this chapter. The main analytical tool is space Fourier analysis.

The simulation abilities of the developed GFEM advection-diffusion framework are demonstrated in Chapter 4, in which group velocity tests, rotational tests, and experiment results with real data over static locally uniform grids are described.

Readers not interested in the programming implementation can restrict their attention just on these chapters: 2, 3, 4.

The OODEM framework is described in Chapter 5. The stress is on the advection-diffusion sub-framework, since the design of this sub-framework influences the design of OODEM as a whole. The OODEM framework is described using patterns from three pattern languages: framework patterns language, design patterns language, and a (very small) language developed for DEM. A potential framework user of OODEM can read just this chapter, if he or she is familiar with the method of finite elements, the object-oriented programming, and the design patterns.

The main audience of Chapter 6 are readers familiar with FORTRAN, or some other modular programming language, but not familiar with the Object-Oriented Programming (OOP). An alternative name of the chapter would be "Object-oriented programming in 20 minutes".

The latest fashion in the OOP is the use of patterns and pattern languages, and in my opinion patterns bring the OOP into maturity. The tradition is to identify patterns by feelings: when we describe a pattern, we should feel that it is the "right" solution in its context. A more scientific approach can be used for this identification, and in Chapter 7 the micro-architectural patterns for software design, commonly known as design patterns, are rationalized using the relational data base theory of normal forms. In the chapter are also discussed two other topics: a fundamental pattern in OOP, and a rationalization how to approach two fundamental design patterns.

The possibilities for further development of OODEM are described in Chapter 8.

The major contributions presented in this thesis, and the work it relates to, are described in the last chapter, Chapter 9.

Chapter 2

Modeling of the Air Pollution Problem

You gotta keep 'em separated

– OFFSPRING, *"Come Out And Play ", Smash, 1994*

In this chapter the mathematical model of the Long-Range Transport of Air Pollution (LRTAP) is presented. For this the Zlatev's book [58] is closely followed. The derivation of the model is done by separation of the different processes that constitute the LRTAP, and deriving the mathematical models for each of them. (We will not present the derivation itself, we will just present the equations).

All processes are considered in the time interval $[0,T]$, $T > 0$, and in the space domain which is a parallelepiped in the Euclidean space $\mathbf{R}^3$, and is defined as

$$D = \{(x,y,z)|x \in [0,X], y \in [0,Y], z \in [0,Z]\},$$

where $X > 0$, $Y > 0$, and $Z > 0$.

We will denote by $c_s(x,y,z,t)$ the concentration of a given air pollutant, numbered with s, at the point $(x,y,z) \in D$, at the time $t \in [0,T]$; $s = 1,...,q$, q is the number of the pollutants studied in the model. When an equation is valid for all pollutants we will omit the index s, we will use $c(x,y,z,t)$.

All considered functions are assumed to be defined in $D \times [0,T]$. When it is relevant, and there is no danger of misunderstanding, then the independent variables x, y, z, and t will be omitted e.g. the abbreviation c will be used instead of $c(x,y,z,t)$.

2.1 The long-range transport of air pollution

The physical phenomenon, which is well known under the name "Long-Range Transport of Air Pollution" (LRTAP) consists of three major stages:

1. *Emission.* During this first stage different pollutants are emitted in the atmosphere from different emission sources. Many emission sources are anthropogenic, but some of the air pollutants are also emitted from natural emission sources.

2. *Transport.* The actual transport of air pollution takes in the second stage. The driving factor is the wind. The transport of air pollution in the atmosphere that is caused by the wind is normally called "advection of the air pollutants".

3. *Transformations during the transport.* Three major physical processes take place during the the transport of pollutants in the atmosphere:

   (a) *Diffusion.* The air pollutants are widely dispersed in the atmosphere

(b) *Deposition.* Some of the pollutants are deposed to various surfaces of the Earth (soil, water, vegetation). Two different kinds of deposition phenomena are usually considered: dry deposition and wet deposition. The dry deposition continues throughout the long range transport, while the wet deposition takes place only when it rains.

(c) *Chemical reactions.* Many different chemical reactions take place during the transport of pollutants in the atmosphere. As a result of the chemical reactions many secondary pollutants are created (the air pollutants that are emitted directly from the emission sources in atmosphere are often called primary pollutants). Some of the chemical reactions lead to a production of final species that are not damaging.

2.2 Modeling the advection

Let us denote the wind velocities along the three coordinate axes by $u(x,y,z,t)$, $v(x,y,z,t)$, and $w(x,y,z,t)$ respectively.

The long transport of air pollutants that is due to the wind can adequately be described by the following Partial Differential Equation (PDE)

$$\frac{\partial c}{\partial t} = -\frac{\partial (uc)}{\partial x} - \frac{\partial (vc)}{\partial y} - \frac{\partial (wc)}{\partial z} \tag{2.1}$$
$$\text{for } \forall (x,y,z) \in D \subset \mathbf{R}^3 \quad \text{and for} \quad t \in [0,T].$$

The above equation can be simplified by assuming that the conservation law is satisfied for the wind velocities in the lower parts of the atmosphere:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \tag{2.2}$$
$$\text{for } \forall (x,y,z) \in D \subset \mathbf{R}^3 \quad \text{and for} \quad t \in [0,T].$$

By substituting equation (2.2) in the three dimensional model (2.1) the following PDE is obtained:

$$\frac{\partial c}{\partial t} = -u\frac{\partial c}{\partial x} - v\frac{\partial c}{\partial y} - w\frac{\partial w}{\partial z} \tag{2.3}$$
$$\text{for } \forall (x,y,z) \in D \subset \mathbf{R}^3 \quad \text{and for} \quad t \in [0,T].$$

The mathematical model that is defined by (2.1) describes a pure advection process (of any pollutant) in the three dimensional space. Two-dimensional models can also be considered. In the later case the functions considered are $c(x,y,t)$, $u(x,y,t)$, and $v(x,y,t)$. These functions are considered now at any point $(x,y) \in D \subset \mathbf{R}^2$, and at any time $t \in [0,T]$, where D is a space domain in the two -dimensional Euclidean space $\mathbf{R}^2$ and $[0,T]$ is a time-interval with $T > 0$. It is normally assumed that the two-dimensional domain D is rectangular:

$$D = \{(x,y) | x \in [0,X], y \in [0,Y]\}.$$

A two-dimensional model can be obtained formally from the three-dimensional one, represented by (2.1). It is simply necessary to remove the last term in the right hand side of the partial differential equation (2.1). The result is given by:

$$\frac{\partial c}{\partial t} = -\frac{\partial (uc)}{\partial x} - \frac{\partial (vc)}{\partial y} \tag{2.4}$$
$$\text{for } \forall (x,y) \in D \subset \mathbf{R}^2 \quad \text{and for} \quad t \in [0,T].$$

It can be further assumed that

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0, \tag{2.5}$$
$$\text{for} \quad \forall (x,y) \in D \subset \mathbf{R}^2 \quad \text{and for } \forall t \in [0,T],$$

though this is not plausible, and the numerical scheme presented in the next chapter does not rely on it. Nevertheless, if it is used the three-dimensional model represented by (2.1) and (2.2), and if it is assumed that the concentrations are evenly distributed along the vertical direction, i.e.

$$\frac{\partial c}{\partial z} = 0, \text{ in } D \times [0, T],$$

then the use of 2.5 can be justified.

Remark 2.2.1 *The most important issue when the PDE (2.4) is solved, is the approximation of the space derivatives. If we want to apply a pseudo-spectral discretization method, it is easier to assume (2.5) and to discretize the equation*

$$\frac{\partial c}{\partial t} = -u\frac{\partial c}{\partial x} - v\frac{\partial c}{\partial y}, \tag{2.6}$$

instead of (2.4). The Fast Fourier Transform (FFT) is applied to the function c, then the frequency domain presentation of c is differentiated, which is then transformed back to the physical space, and it is substituted in (2.6) in order to obtain a system of ODE's, the solution of which is the new value of c.

When the PDE (2.4) is solved with a two-dimensional finite element method, an appropriate Hilbert space should be constructed. The scalar product of this space can be based of the right side of (2.4), so the assumption (2.5) is not needed.

2.3 Modeling the diffusion

Let us denote the diffusivity coefficients along the three coordinate axes by $K_x(x, y, z, t)$, $K_y(x, y, z, t)$, $K_z(x, y, z, t)$ respectively.

The diffusion process can be described by the following partial differential equation:

$$\frac{\partial c}{\partial t} = \frac{\partial}{\partial x}(K_x\frac{\partial c}{\partial x}) + \frac{\partial}{\partial y}(K_y\frac{\partial c}{\partial y}) + \frac{\partial}{\partial z}(K_z\frac{\partial c}{\partial z}). \tag{2.7}$$

It is often assumed that K_x and K_y are non-negative constants. In the DEM they are assumed to be

$$K_x = K_y = 30000m^2/s.$$

The treatment of the K_z is more difficult. Some simple suggestions are listed in [58]. These are

$$K_z = const.,$$

$$K_z = z^\alpha \qquad 0 \le z \le Z,$$

$$K_z = (Z - z)^\alpha \quad 0 \le z \le Z,$$

$$K_z = z(Z - z)^\alpha \quad 0 \le z \le Z,$$

$$K_z = z \qquad 0 \le z \le \frac{Z}{2},$$
$$K_z = Z - z \quad \frac{Z}{2} \le z \le Z.$$

Normally, advanced mechanisms are to be applied. It should also be emphasized here that non-equidistant grids are used in the vertical direction: fine resolution grids close to the surface, and sparse resolution grids close to the top boundary.

2.4 Modeling the deposition

Both dry and wet deposition processes take place during the long range transport of air pollutants. The dry deposition takes place throughout the transport, while the wet deposition takes place only when some kind of precipitation occurs in the area under consideration.

The deposition depends on the particular air pollutants that are transported – this is not the case with the advection and the diffusion. However, under certain assumptions the deposition of a certain pollutant does not depend on the deposition of the other pollutants involved in the model. If such assumptions are made then the deposition depends only on the concentration, not on its spatial derivatives.

Let us denote the dry deposition coefficient of the pollutant s with $k_{1s}(x,y,z,t)$ and its wet deposition coefficient with $k_{2s}(x,y,z,t)$, $s = 1,...,q$. Then the deposition process for the pollutant s is described by the following linear ordinary differential equation

$$\frac{\partial c_s}{\partial t} = -(k_{1s} + k_{2s})c_s. \tag{2.8}$$

2.5 Modeling the chemistry

The chemical transformations constitute one of the most important processes – the chemistry – under the pollutants transport. These transformations should be treated very carefully. It is much more complicated to derive a model for them, than for the rest of the physical processes in LRTAP. To derive the mathematical model for the advection, diffusion and deposition, we start with some acceptable assumptions, and then "just" write down the equations (PDE's or ODE's). Chemists, on the other hand, start with the observed end products of the air pollutants chemistry, and knowing what pollutants have been emitted in the atmosphere, they use some sort of reverse engineering, called "Retrosynthesis" – which can be defined as thinking backwards from relatively complex molecules to simpler ones – to obtain a set of reactions (and scenarios of them) that model the transformation of the emitted pollutants to the end products. The set of reactions and the scenarios of their sequences are called *mechanism.*

Most of the chemical mechanisms are too big for efficient implementation, so they are lumped or approximated by a smaller number of reactions. One of the most used and verified mechanisms is the Carbon-Bond Mechanism (CBM) IV [22], which is a condensed version of a larger chemical mechanism of photochemistry called CBM-EX. CBM IV has 33 species and 82 chemical reactions. In it the organic compounds are grouped (lumped) according to their carbon bond type: single, double, or carbonyl. See the CBM IV reactions in [58, Ch.2 pp.42-44].

The stoichiometry equations of the mechanism are transformed to a system of ODE's.[1] As Verwer et al. explain in [50], all chemical kinetics mathematical models can be written in the so called production-loss form

$$\frac{\partial c_s}{\partial t} = P_s(c_1,...,c_q) - L_s(c_1,...,c_q)c_s,$$

where $s = 1,...,q$, q is the number of the pollutants studied in the model. The polynomial $P_s(c_1,...,c_q)$ is the production term and $L_s(c_1,...,c_q)c_s$ is the loss term. The P_s and L_s are non-negative.

The coefficients in these ODE's depend upon time, since some of the reactions are photochemical, i.e. they depend from the light (therefore, from the position of the Sun), and the non-photochemical reactions depend upon the temperature. For more extensive explanations see [58, Ch. 2] and [41].

2.6 Introduction of the emissions

The introduction of the emission sources in the model is straightforward. Let us consider some primary pollutant whose concentration is $c_s(x,y,z,t)$. Let the positions of the sources of this pollutant be at the points $\{(x_k,y_k,z_k)|\ k = 1,...,K\}$. These are active (emit the pollutant) at the time intervals

[1] Clearly, this can be done automatically and eventually supplied with some code generation of different kind of algorithms. The program (the framework?) KPP - Kinetic PreProcessor does that. Also, see [4] for a simple program how to derive from a FORTRAN code of the right hand side of an ODE's system, the FORTRAN code for the Jacobian.

$\{[t_{k,l}^{start}, t_{k,l}^{end}] | \; l = 1, ..., L_k, \; k = 1, ..., K\}$. Under these assumptions, we can define a function $E_s(x, y, z, t)$ in the domain $D \times [0, T]$ for the concentration of the emission of the pollutant s.

2.7 The mathematical model and its splitting

By combining the mathematical expressions for the considered above five physical processes, we can obtain the LRTAP mathematical model (a system of PDE's)

$$\begin{aligned} \frac{\partial c_s}{\partial t} &= -\frac{\partial (u c_s)}{\partial x} - \frac{\partial (v c_s)}{\partial y} - \frac{\partial (w c_s)}{\partial z} \\ &+ \frac{\partial}{\partial x}(K_x \frac{\partial c_s}{\partial x}) + \frac{\partial}{\partial y}(K_y \frac{\partial c_s}{\partial y}) + \frac{\partial}{\partial z}(K_z \frac{\partial c_s}{\partial z}) \\ &+ E_s + Q_s(c_1, c_2, ..., c_q) - (k_{1s} + k_{2s}) c_s, \\ &s = 1, 2, ..., q. \end{aligned} \tag{2.9}$$

Let us summarize the meaning of the different quantities that are involved in the model

- the concentrations are denoted by c_s;
- u, v, and w are wind velocities;
- K_x, K_y, and K_z are diffusion coefficients;
- the emission sources in the space domain are described by the functions E_s;
- κ_{1s} and κ_{2s} are deposition coefficients;
- the chemical reactions used in the model are described by the non-linear functions $Q_s(c_1, c_2, \ldots, c_q)$.

The number of equations q is equal to the number of species that are included in the model.

It is difficult to treat the system of PDE's (2.9) directly. This is the reason for using different kinds of splitting. A simple splitting procedure, based on ideas discussed in Marchuk[33] and McRae et al. [35], can be defined, for $s = 1, 2, \ldots, q$, by the following sub-models:

$$\frac{\partial c_s^{(1)}}{\partial t} = -\frac{\partial (u c_s^{(1)})}{\partial x} - \frac{\partial (v c_s^{(1)})}{\partial y} \tag{2.10}$$

$$\frac{\partial c_s^{(2)}}{\partial t} = \frac{\partial}{\partial x}\left(K_x \frac{\partial c_s^{(2)}}{\partial x}\right) + \frac{\partial}{\partial y}\left(K_y \frac{\partial c_s^{(2)}}{\partial y}\right) \tag{2.11}$$

$$\frac{dc_s^{(3)}}{dt} = E_s + Q_s(c_1^{(3)}, c_2^{(3)}, \ldots, c_q^{(3)}) \tag{2.12}$$

$$\frac{dc_s^{(4)}}{dt} = -(\kappa_{1s} + \kappa_{2s}) c_s^{(4)} \tag{2.13}$$

$$\frac{\partial c_s^{(5)}}{\partial t} = -\frac{\partial (w c_s^{(5)})}{\partial z} + \frac{\partial}{\partial z}\left(K_z \frac{\partial c_s^{(5)}}{\partial z}\right) \tag{2.14}$$

The horizontal advection, the horizontal diffusion, the chemistry, the deposition and the vertical exchange are described with the systems (2.10)-(2.14). This is not the only way to split the model defined by (2.9), but the particular splitting procedure (2.10)-(2.14), which we will call *DEM splitting procedure*, has three advantages:

1. The physical processes involved in the big model can be studied separately.

2. It is easier to find optimal (or, at least, good) methods for the simpler systems (2.10)-(2.14) than for the big system (2.9).

3. If the model is to be considered as a two-dimensional model (which often happens in practice), then one should just skip system (2.14).

The numerical method presented in Chapter 2 treats the horizontal advection (2.10) and diffusion (2.11) together.

The splitting error can be analyzed employing the notion of L-commutativity and the Lie operator formalism as it is shown in [18, 30]. As it is shown by Dimov et al. in [18] the error of the DEM splitting procedure is of order $O(\tau)$, where τ is the simulation time step.

2.8 Eulerian and Lagrangian approaches

There are two main approaches for the numerical treatment of (2.9).

2.8.1 Eulerian approach

The modeling region in the so called Eulerian approach is covered with a grid on which the space derivatives in (2.9) (respectively (2.10)-(2.14)) are approximated with some finite differences scheme. Then the obtained for each concentration and each grid point ODE's with unknown time dependant function are solved.

2.8.2 Lagrangian approach

Consider the point B in the space domain. A trajectory η which arrives at B is calculated, and this trajectory can be followed backward for a certain time-period to reach some starting point A. It is also possible to reverse the direction in which the trajectory is followed: the starting point A is first fixed and the trajectory is followed forward for a certain period until the final point B is reached.

In some particular case is more convenient to consider forward trajectories, in some cases backward trajectories. Let us assume that

1. the concentrations of all air pollutants that are involved in the considered air pollution model are known at the starting pont A;

2. there is no diffusion in the modeled air pollution process;

3. a two-dimensional model is to be developed (hence the trajectory η is two-dimensional curve).

Then the concentrations of the air pollutants involved in the model at the final point B can be calculated by following the transport of the air parcel along the trajectory η from point A to point B and by taking into account

1. all contributions from the emission sources along the trajectory to the appropriate concentrations;

2. all changes of the concentrations under the transport that are due the dry deposition and wet deposition;

3. all transformations of the concentrations under the transport that are caused by the chemical reactions involved in the model.

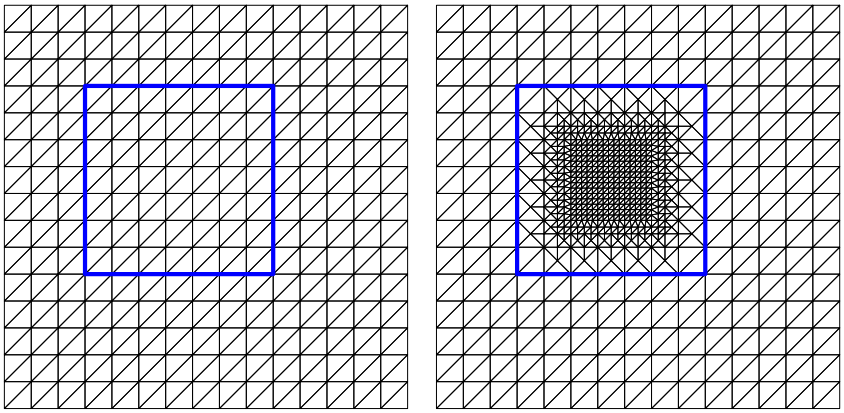


Figure 2.1: Constructing a grid with local refinement. The refinement cell consists of nested refinements with ratio $1:2$, $1:2$, $4:5$. So the triangles sides in the finest grid are 5 smaller the those of the coarsest grid.

2.8.3 The choice of model among the two groups

The choice of a model among the two groups for the air pollution problem is discussed at large by Zlatev in [58, pp. 54-67]. The Lagrangian model has the major drawback that if it is used on fine spatial grids, then the results will be normally unrealistic: most of the trajectories will "miss" many sources. As Zlatev explains, the accuracy of the results derived with an Eulerian model will be improved by refining the grid, whereas the Lagrangian models do not have this useful property. Also, the Lagrangian models ignore one of the most important physical processes in LRTAP, the diffusion process.

An implementation of an Eulerian model – the Danish Eulerian Model – is described in the thesis.

2.9 Modeling with local refinements

We gave in the introduction a motivation why static local refinements simulations should be considered instead of one-way nesting simulations. The object-oriented framework described in Chapter 5 fully supports the use of Galerkin finite element methods over static locally refined grids (mesh generation included). We will call this algorithm Static Local Refinements Algorithm (SLRA). Bellow a more detailed comparison is given between SLRA and an algorithm superior to the one-way nesting, the Recursive Static-Regridding Algorithm (RSRA; see [40]).

SLRA can be defined as follows:

The procedure for generating a grid is illustrated on Figure 2.1. We first generate the coarse grid. Then we extract a rectangular cell of it that contains the region of interest, finally we refine the selected cell to the desired resolution. The refined cell is plugged in in the coarse grid and then we generate the grid description suitable for our simulation code.[2] Clearly this algorithm can be applied for several regions of refinement as long as the refined regions do not overlap. (RSRA has the same requirement.) Then a Galerkin finite element method is applied with basis functions generated on the obtained locally refined grid.

2.9.1 Comparison with static-regridding

The RSRA is designed for time evolution problems involving PDE's that have solutions with sharp moving transitions, such as steep wave fronts and emerging or disappearing layers. RSRA is used because it is inefficient to solve these problems numerically on a grid held fixed throughout the entire computation [40].

[2] In Section 2.9.2 we describe how the different data fields are interpreted over regular and refined grids.

We will compare how RSRA and SLRA solve the following Air Pollution Problem (APP): How to utilize more detailed emission and meteorological data over a region of the computation's domain. (As it was explained above, it is inefficient to solve this problem numerically on a regular totally refined grid.

Both SLRA and RSRA use grids concentrated just of the areas of interest. The grid's construction is based on the principle of local uniform grid refinement (LUMR) – we make locally uniform, possibly nested, refinements on an uniform grid.

When the solution advances from the time level t_0 to $t_0 + \Delta t$, RSRA have the following steps (see again [40]):

1. Integrate on the coarse grid using one coarse time step Δt.

2. Integration is followed by regridding. Using the new coarse c-values decide where the fine grid will be for $t_0 \leq t \leq t_0 + \Delta t$. When we apply RSRA to APP, the regridding will always be over the same subdomain. (This differs from the description in [40], where the domain of refinement is chosen according to the sharp gradients of the coarse solution.)

3. Regridding is followed by interpolation. Return to time level t_0 and determine the initial values for the fine grid. For $t_0 \leq t \leq t_0 + \Delta t$ we need to specify boundary values at the grid interfaces where fine grid cells abut on coarse cells. Using old and new coarse c-values, at these grid interfaces are imposed Dirichlet boundary conditions via interpolation.

4. Next the fine grid is integrated over the interval $t_0 \leq t \leq t_0 + \Delta t$. The so found refined c-values are injected in the coarse grid points.

Multiple levels in RSRA are handled in a recursive fashion. It is clear from step 3 that when this algorithm is applied to APP, the simulations over the refined domain influence the solution through the inner non-physical boundaries. We can say that RSRA supposes that after the injection of the refined simulation results into the coarser ones, we will have nearly the same solution if we used global refinement.

In global air pollution modeling, a variant of RSRA is usually applied, this is the one way nesting, in which the region of the nested *rectangular* refinements is the same for each step and the injection in step 4 is not performed. The advantage of one way nesting is that the same algorithm can be used for the coarse and the finer grids. This means that if we have an algorithm for simulations on the coarser regular grid, it can be reused in an algorithm for refined simulations. This is an algorithm easier to develop than one that seeks global solutions on the locally refined grid. The algorithms for regular two dimensional grids can use some splitting procedure to become simpler and faster (see, for example, [21]). The major disadvantage is that the results of the refined simulations do not influence the coarser grid results. If some puff is coming from outside to the refined region it will be transfered to it by the boundary conditions. But if a puff is located inside the refined region, it influences only the refined solution. Even if an injection is done, it could smooth the puff or even would not be able to see it if the refinement is too 'deep' – the largest puff values could lie on the finer grid. The situation is similar when transferring the puff through the boundary: some approximation should be done from the finer to the coarser boundary.

The method we developed, SLSA, is a standard application of the two dimensional finite elements method. We first generate the locally uniform refined grid and then we find the – global – solution of APP over this grid. Some examples of the grids we use are shown in Figure 2.1, and Figures 4.25 and 4.26. The advantages of SLSA is that it takes the approximations we indicated above in a natural fashion in both ways. These inherent approximations cause some errors so the solutions are not so accurate as the solutions from totally uniform refinement of the whole grid, but are much cheaper and clearly more accurate than one-way refinement ones.

2.9.2 Grid interpretation

In this section we will show how the data over the locally refined grids is interpreted.

Let us consider the regular grids first. The modeling domain is partitioned into squares as it is shown on Figure 2.2. We will call these squares *interpretation squares* and we will call the grid formed by them *interpretation grid*. The emission data, the wind velocities, and the concentrations are related with the

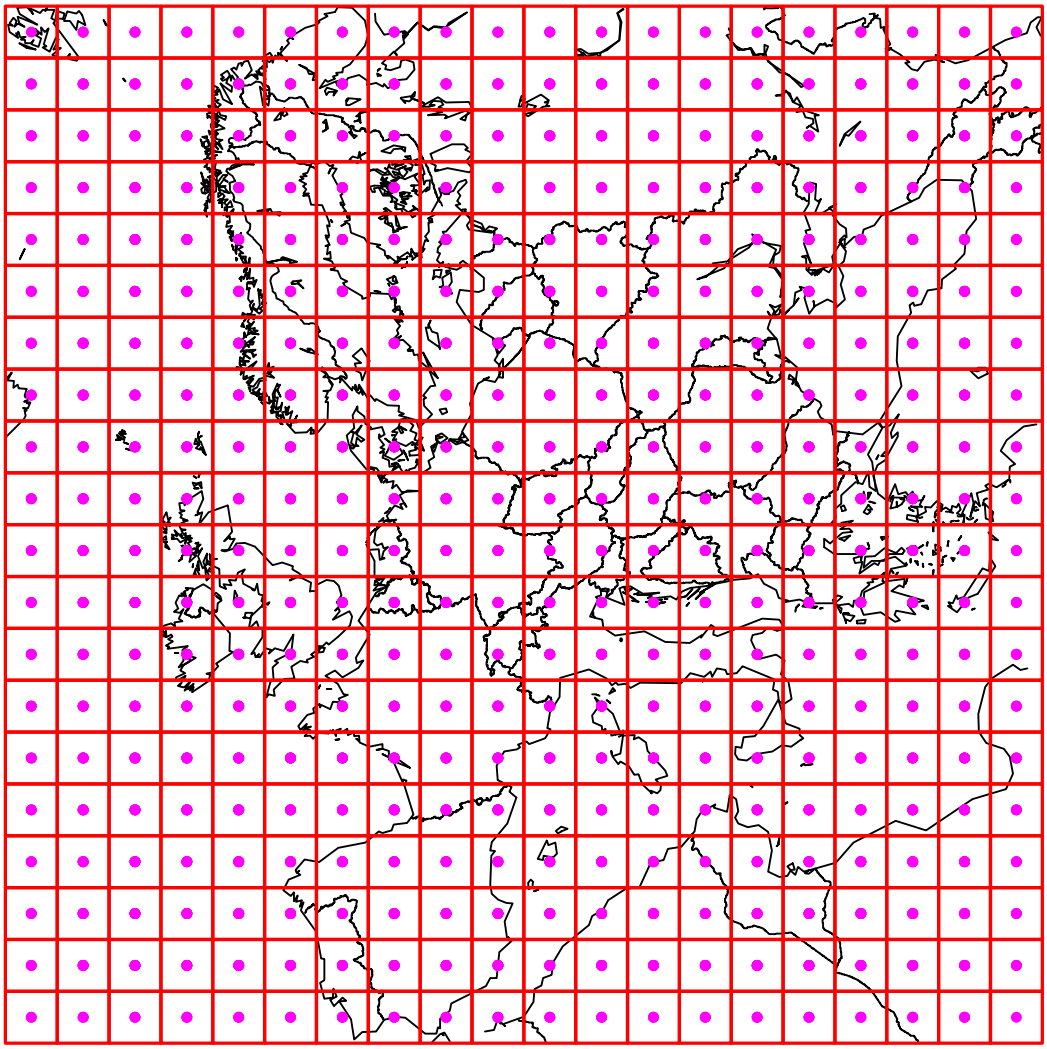


Figure 2.2: Europe partitioned into squares. The emission data, the wind velocities, and the concentrations are related with the middles of the squares.

middles of the squares, we will call them *nodes*. The computational grids are constructed in such a way that their vertexes coincide with the squares middles, the nodes. We can say that the computational grids are drawn among the nodes; see Figure 2.3.

When some part of the mesh is refined, the nodes of the refined part are interpreted in a similar way. Let us consider local refinement with ratio 1:5. This means that the interpretation squares over which the refinement is made are partitioned into 5×5 smaller interpretation squares. The interpretation is shown on Figure 2.4. Of course the nodes from the smoother (the part interfacing the initial coarse grid with the refined part) should be interpreted differently. One should be aware that not all refinements allow such an interpretation: on Figure 2.5 is shown an example of refinement which is difficult to interpret.

To explain the interpretation we have used triangular grids, but obviously the same interpretations can be applied if square grids are used.

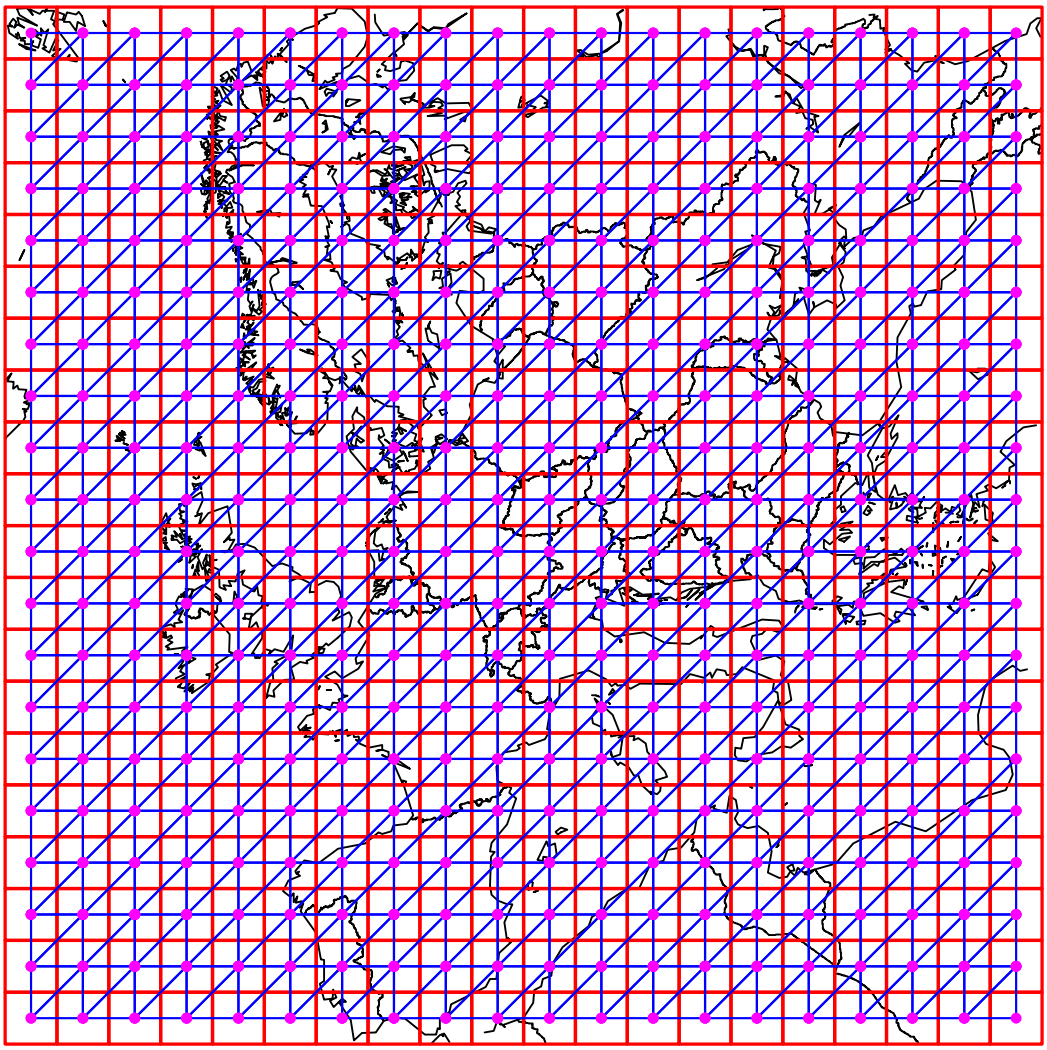


Figure 2.3: Europe partitioned into squares with triangular mesh drawn among the nodes.

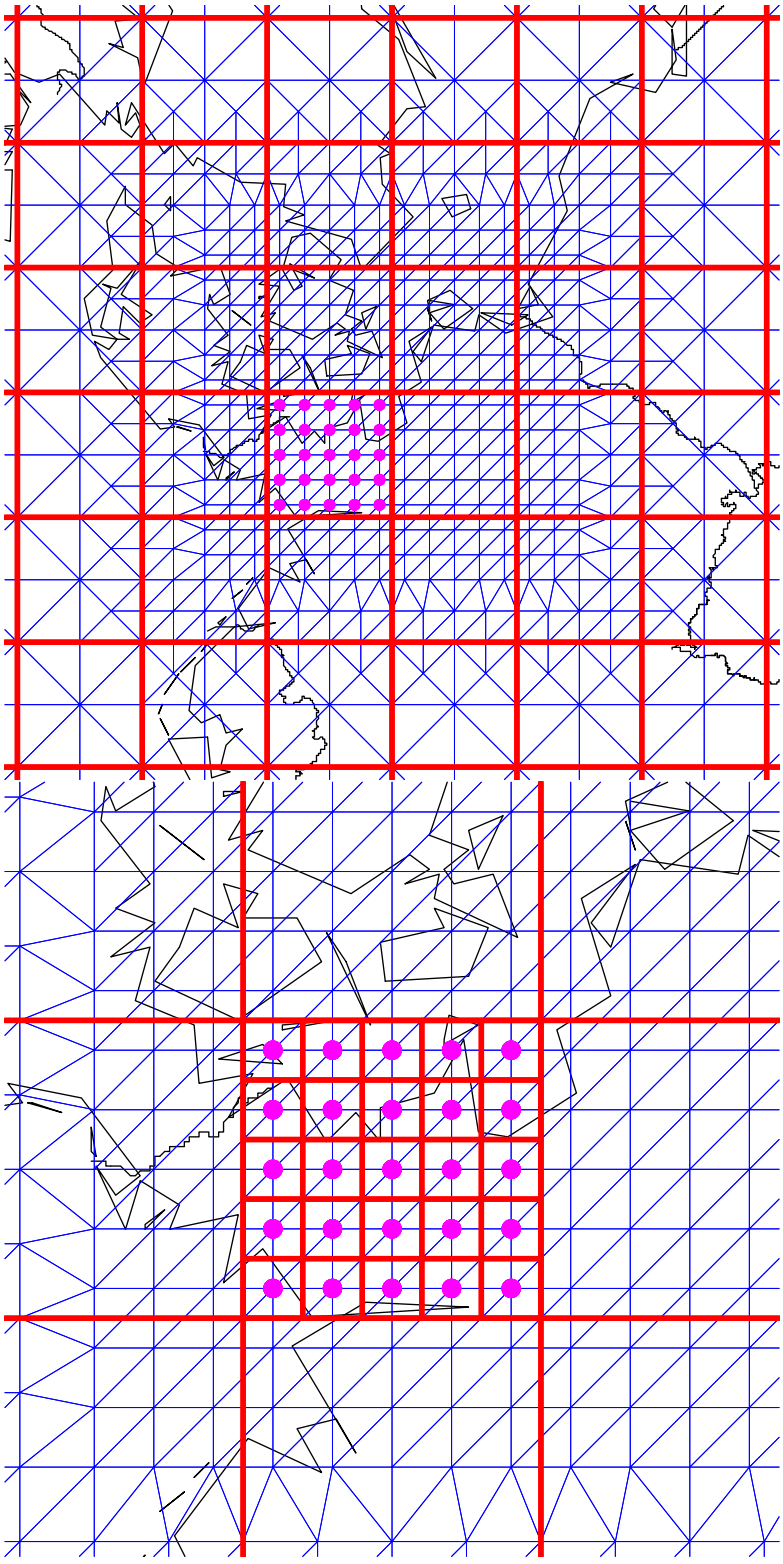


Figure 2.4: Interpretation of a local grid refinement. The red lines are from the squares partition in Figure 2.2. On the top picture are pointed out the vertexes of the refined grid that should be interpreted as middles of the refinement interpretation squares.The refinement interpretation squares are shown on the bottom figure, where the "violet node" square is zoomed.

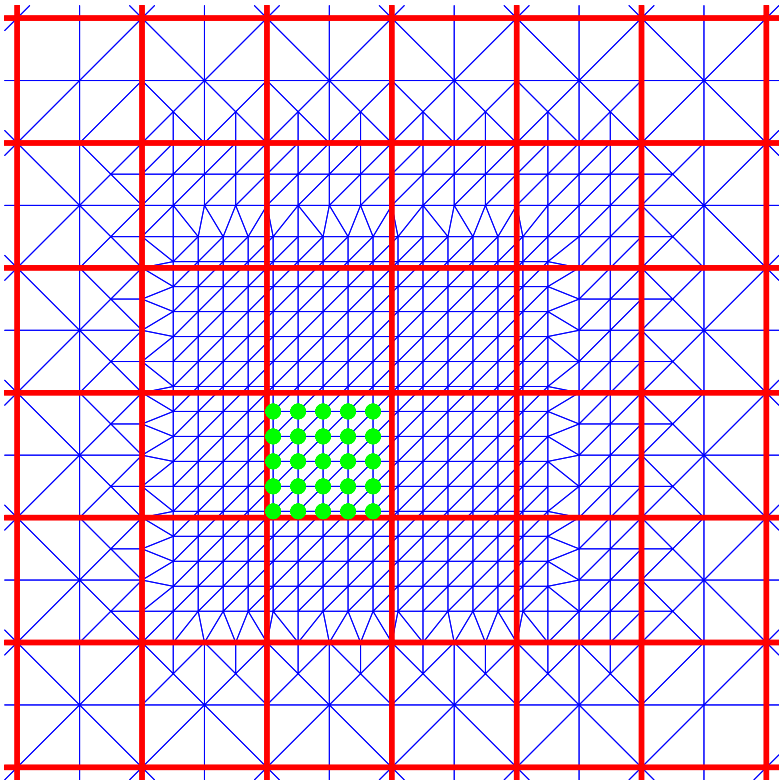


Figure 2.5: Difficult to interpret refinement.

Chapter 3

Galerkin Finite Element Method for the Air Pollution Advection-Diffusion Equation

And all the girlie say I'm pretty fly, for a white guy

– OFFSPRING, *Pretty Fly (For A White Guy), Americana, 1998*

3.1 The advection-diffusion equation

The horizontal advection-diffusion submodel of (2.9) has the form

$$\frac{\partial c}{\partial t} = -\frac{\partial (uc)}{\partial x} - \frac{\partial (vc)}{\partial y} + \frac{\partial}{\partial x}(K_x \frac{\partial c}{\partial x}) + \frac{\partial}{\partial y}(K_y \frac{\partial c}{\partial y}). \tag{3.1}$$

We will assume that the diffusion coefficients in Equation (3.1) are positive constants, sufficiently small, so that the advection dominates over the diffusion. Then the equation (3.1) becomes:

$$\frac{\partial c}{\partial t} = -\frac{\partial (uc)}{\partial x} - \frac{\partial (vc)}{\partial y} + K_x \frac{\partial^2 c}{\partial x^2} + K_y \frac{\partial^2 c}{\partial y^2}. \tag{3.2}$$

We consider the problem of finding the solution of the equation (3.2) on the rectangular domain $\Omega = [0, X] \times [0, Y]$ in the time interval $[t_0, t_1]$ with given initial conditions $c(x, y, 0) = c_0(x, y)$, $(x, y) \in \Omega$. To specify the two types of boundary conditions let us denote with $\vec{n}$ the vector outwardly normal to $\partial\Omega$, and $\vec{w} := \begin{bmatrix} u(x,y) \\ v(x,y) \end{bmatrix}$. We have an "outflow" boundary when $\vec{n} \cdot \vec{w} > 0$, and an "inflow" boundary when $\vec{n} \cdot \vec{w} < 0$. At the outflow boundary we should not specify any boundary conditions (the PDE itself prevails there). At the "inflow" part of the boundary we impose the condition $\partial c / \partial \vec{n} = 0$.

For deeper discussion of the problem and the boundary conditions, we refer to Gresho and Sani [23, Ch. 2].

3.2 The numerical scheme

3.2.1 Finite element formulation

The *weak formulation* of the problem above, based on the books of Brenner and Ridgeway-Scott [12] and Gresho and Sani [23], is as follows:

1. Define

$$a(\varphi,\psi) := \int\int_\Omega \left(\frac{\partial(u\,\varphi)}{\partial x}\,\psi + \frac{\partial(v\,\varphi)}{\partial y}\,\psi + K_x \frac{\partial\varphi}{\partial x}\frac{\partial\psi}{\partial x} + K_y \frac{\partial\varphi}{\partial y}\frac{\partial\psi}{\partial y} \right) dx\,dy$$

and

$$\langle\varphi,\psi\rangle := \int\int_\Omega \varphi\,\psi\,dxdy.$$

2. Define the normed space $V = \{\psi \in L^2(\Omega) : a(\psi,\psi) < \infty\}$ of functions with domain Ω.

3. We assume that the solution of (3.2) has the form $c(x,y,t) = g(t)\varphi(x,y)$ and it is characterized by

$$c \in V \times [t_0,t_1], \quad \left\langle \frac{\partial c}{\partial t}, \psi \right\rangle = a(c,\psi), \quad \forall\psi \in V. \tag{3.3}$$

This is the weak formulation of (3.2).

In order to derive a numerical method, we should discretize (3.3). Let $S \subset V$ be a finite dimensional subspace of V. Let us consider (3.3) with V replaced by S, namely

$$c_S \in S \times [t_0,t_1], \quad \left\langle \frac{\partial c_S}{\partial t}, \psi \right\rangle = a(c_S,\psi), \quad \forall\psi \in S. \tag{3.4}$$

If we choose a basis of S, we can express (3.4) as a system of ODE's with linear right sides. This is done in the following way:

1. Choose some functions $N_i(x,y)$, $i \in I$ (I is an index set, $|I| = dim(S)$) that form a basis in S.

2. The function $c(x,y,t)$ is approximated in the form

$$c(x,y,t) \simeq c_S(x,y,t) = \sum_{i\in I} g_i(t) N_i(x,y).$$

That form is substituted in the weak formulation (3.4)

$$\begin{aligned}
&\sum_{i\in I} \frac{\partial g_i(t)}{\partial t} \int\int_\Omega N_i(x,y)N_j(x,y)dxdy = \\
&\sum_{i\in I} g_i(t) \int\int_\Omega \left(\frac{\partial\left(u(x,y,t)N_i(x,y)\right)}{\partial x} N_j(x,y) + \frac{\partial\left(v(x,y,t)N_i(x,y)\right)}{\partial y} N_j(x,y) \right. \\
&\left. + K_x \frac{\partial N_i(x,y)}{\partial x}\frac{\partial N_j(x,y)}{\partial x} + K_y \frac{\partial N_i(x,y)}{\partial y}\frac{\partial N_j(x,y)}{\partial y} \right) dxdy, \\
&j \in I
\end{aligned} \tag{3.5}$$

3. Define the numbers

$$p_{ji} = \langle N_i, N_j\rangle = \int\int_\Omega N_i(x,y)N_j(x,y)dxdy \tag{3.6}$$

and

$$\begin{aligned}
a_{ji} &= a(N_i,N_j) = \\
&= \int\int_\Omega \left(\frac{\partial\left(u(x,y,t)N_i(x,y)\right)}{\partial x} N_j(x,y) + \frac{\partial\left(v(x,y,t)N_i(x,y)\right)}{\partial y} N_j(x,y) \right. \\
&= \left. + K_x \frac{\partial N_i(x,y)}{\partial x}\frac{\partial N_j(x,y)}{\partial x} + K_y \frac{\partial N_i(x,y)}{\partial y}\frac{\partial N_j(x,y)}{\partial y} \right) dxdy
\end{aligned} \tag{3.7}$$

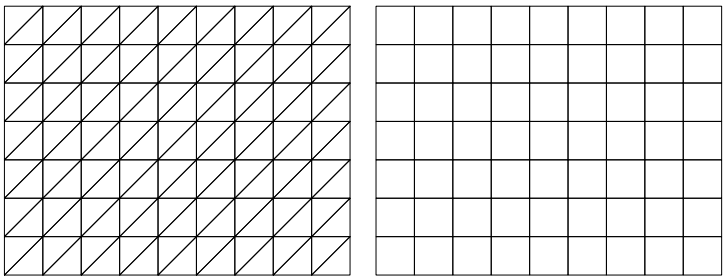


Figure 3.1: Regular triangular and regular square grids.

4. Now the equation (3.5) can be written in the following way

$$\sum_{i\in I} \frac{\partial g_i(t)}{\partial t} p_{ji} = \sum_{i\in I} g_i(t) a_{ji}, \quad j \in I,$$

which is equivalent to

$$P\frac{\partial \overrightarrow{g}(t)}{\partial t} = A\overrightarrow{g}(t), \tag{3.8}$$

where $P = \{p_{ji}\}$, $A = \{a_{ji}\}$, $i, j = 1, ..., dim(S)$, $\overrightarrow{g}(t) \in R^{dim(S)}$.

When we defined the equation (3.3) we assumed that $c(x, y, t)$ has the form $g(t)\varphi(x, y)$. In the discretization above we used the functions $N_i(x, y)$ both to express (approximate) $\varphi(x, y)$ and as the test functions $\psi(x, y)$ in (3.3). This is the so called Galerkin Finite Element Method (GFEM). This type of methods is discussed in detail in [23] and [33]. The methods in which the basis of $\varphi(x, y)$, and the test functions of $\psi(x, y)$ are different sets, are called Petrov-Galerkin methods; see again [33].

Remark 3.2.1 *In order to see that in 4. it is $a_{ji} = a(N_i, N_j)$, not $a_{ij} = a(N_i, N_j)$, let us consider a Petrov-Galerkin method with m basis functions and n tests functions. Then from (3.4) and (3.5), in view of $A\overrightarrow{g}(t)$, the operator A has to have the shape $n \times m$, since $\overrightarrow{g}(t)$ has the shape $m \times 1$. So, clearly, the row j of A represents the scalar product of the test function j with each of the basis functions. Hence $a_{ji} = a(N_i, N_j)$.*

3.2.2 Grid notations

If we want to take full advantage of the GFEM, we should, of course, use non-regular grids. Nevertheless, regular grids are more convenient to establish some properties of GFEM. That is why, we will give here the notation for both regular and irregular grids.

The approximated value of $c(x_i, y_i, t)$ in (3.2) on a general (possibly unstructured) grid with vertices indexed by some set I will be denoted with $g_i(t)$. Let $g(t) = \{g_i(t)\}_{i=1}^{|I|}$, and g^k be an abbreviation to $g(t_k)$, where $\{t_k\}_{k=0}^{N}$ is a discretization of the simulation time interval.

The regular grids consist of points $(x_m, y_n) = (mh, nh)$, where $m = 1, 2..., M$, $n = 1, 2, ..., N$ and $h = x_m - x_{m-1} = y_n - y_{n-1} = constant$. With $g_{m,n}(t)$ will be denoted the approximated value of $c(x_m, y_n, t)$ in (3.2). Let $g(t) = \{g_{m,n}(t)\}_{m=1,n=1}^{M,N}$, and g^k be an abbreviation to $g(t_k)$, where $\{t_k\}_{k=0}^{N}$ is a discretization of the simulation time interval. The regular grids used in this thesis are shown in Figure 3.1.

It will be clear from the context which of the grids (regular or irregular) is used.

3.2.3 Basis functions

We will construct two finite dimensional subspaces of V, which can substitute the space S in (3.3).

Let us consider general grid, the nodes of which are indexed by the set I. The triangular grids, we will consider, have the property that no vertex of any triangle lies on the interior of an edge of another

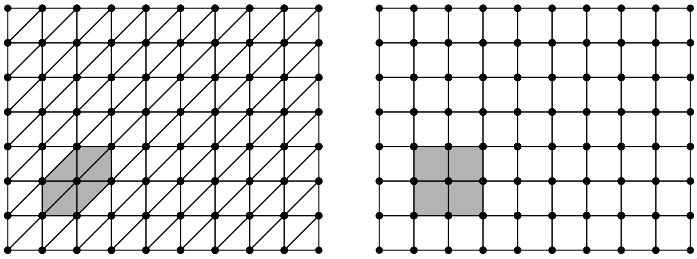


Figure 3.2: Triangulation and rectangulation of a rectangular domain.

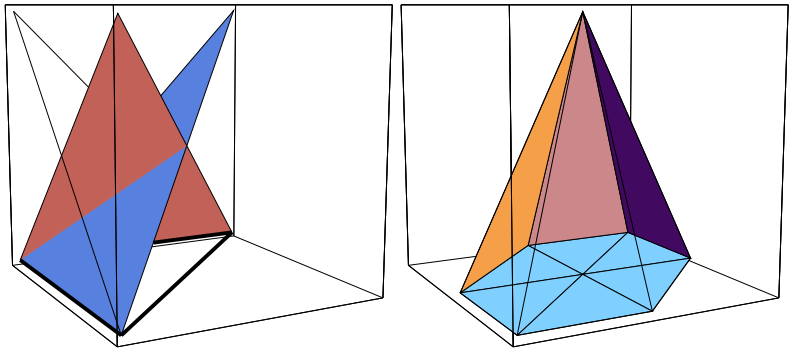


Figure 3.3: On the left: shape functions. On the right: a basis function

triangle. The rectangular grids, we will consider, have the property that the sides of all rectangles are parallel to the axes.

On Figure 3.2 are shown triangulation and rectangulation of the domain Ω. We will call the gray areas *patches* for their middle nodes. So *the patch for the node* j, consists of the elements (triangles or rectangles) to which the node with index j belongs, and the nodes laying on them. We will refer to the nodes that belong to the patch of j as neighbors of j, or mates of j.

Let us fix a node j_0 and consider the function N_{j_0} with the following properties:

- $N_{j_0}(j_0) = 1$,
- $N_{j_0}(i) = 0,\ \forall i \in I \wedge i \neq j_0$,
- N_{j_o} is 0 on every triangle (rectangle) that does not belong to the patch of j_0.

We will call N_{j_0} a *basis function* associated with j_0, or simply a *basis function,* when the node j_0 is not important. On the right on Figure 3.3 is shown a basis function for one of the inner nodes in a triangular grid. Further, *the patch for the basis function* N_j, and *the patch for the node* j, will mean the same. Clearly, the set of basis functions $\{N_i\}_{i=1}^{|I|}$ define a finite dimensional linear vector space. The basis functions are independent since their domains do not coincide. They form a basis for the piecewise linear functions on Ω induced by the grid.

The basis functions in a triangular grid define on each triangle linear functions associated with its vertexes, called *shape functions.* They have the form

$$a_1 + a_2 x + a_3 y,$$

and are plotted on the left on Figure 3.3.

For rectangular grids, the shape functions have the form

$$a_1 + a_2 x + a_3 y + a_4 xy.$$

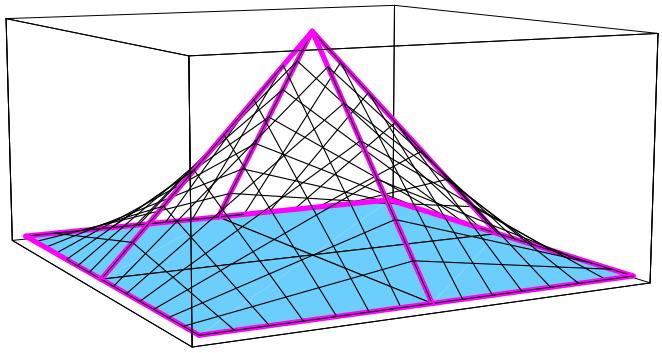


Figure 3.4: Basis function for square grid

A basis function in a rectangular grid is shown on Figure 3.4.

More about the geometry of different shape functions can be read in [51, Ch. 11]. A more theoretical construction of finite element spaces can be found in [12, Ch. 3].

3.2.4 Semi-discrete equations

The Finite Element Method (FEM), as it is characterized by Strang and Fix in [42], is a "package deal": it neither requires nor permits the user to specify independent treatment of local parts of the problem. Nevertheless, it is useful to interpret the finite element equations locally for a given patch as finite difference equations: in this way we will obtain some insight how FEM works. Other benefits of such interpretation are: (i) FEM can be compared more easily with the finite differences methods, (ii) the tools for analysis the finite difference methods can be applied, (iii) we can use it to check the correctness of the FEM programming implementation. In this section we will give just the finite difference interpretation; analysis of FEM with finite difference techniques can be found, for example, in [29], and in Section 3.5.

As it was shown in Section 3.2.1, the application of the Galerkin principle leads to $c(x,y,t) \simeq \sum_i g_i(t)N_i(x,y)$. The summation has to be carried out over all elements in the grid; (see, for example, Figure 3.2). However, the basis functions differ from zero only in a few elements containing the node i. Hence, if we assume that the wind field is constant within the each element (though it might differ from element to element), from the equation (3.5) we can write the j-th row of the matrix equation (3.8):

$$\begin{aligned} \sum_{i\in I_j} \frac{\partial g_i(t)}{\partial t} \quad & \textstyle\sum_{k\in E_j} \int\int_{\triangle_k} N_i(x,y)N_j(x,y)dxdy = \\ \sum_{i\in I_j} g_i(t) \sum_{k\in E_j} \quad & \Big(u_{\triangle_k} \textstyle\int\int_{\triangle_k} \frac{\partial N_i(x,y)}{\partial x} N_j(x,y)dxdy \\ & + v_{\triangle_k} \textstyle\int\int_{\triangle_k} \frac{\partial N_i(x,y)}{\partial y} N_j(x,y)dxdy \\ & + K_x \textstyle\int\int_{\triangle_k} \frac{\partial N_i(x,y)}{\partial x} \frac{N_j(x,y)}{\partial x} dxdy \\ & + K_y \textstyle\int\int_{\triangle_k} \frac{\partial N_i(x,y)}{\partial y} \frac{\partial N_j(x,y)}{\partial y} dxdy\Big), \end{aligned} \tag{3.9}$$

where E_j is an index set for the elements (triangles in (3.9)) that constitute the patch of N_j, and I_j is an index set for the neighbors of N_j. The values of the integrals in formula (3.9) for the triangular patch on Figure 3.5, are shown in the Tables 3.1, 3.2, 3.3.

If we assume that u and v are constant on the entire area of the patch, then the following formula holds for the triangular element patch shown on the left in Figure 3.5:

$$3\dot{g}_{m,n}(t) + \frac{1}{2}\left(\dot{g}_{m+1,n+1}(t) + \dot{g}_{m+1,n}(t) + \dot{g}_{m,n+1}(t)\right.$$

		Element No					
Node No	Spatial Coordinates	1	2	3	4	5	6
1	$m-1, n-1$	$\frac{1}{24}$	$\frac{1}{24}$	0	0	0	0
2	$m-1, n$	0	$\frac{1}{24}$	$\frac{1}{24}$	0	0	0
3	$m, n-1$	$\frac{1}{24}$	0	0	$\frac{1}{24}$	0	0
4	m, n	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{12}$
5	$m, n+1$	0	0	$\frac{1}{24}$	0	0	$\frac{1}{24}$
6	$m+1, n$	0	0	0	$\frac{1}{24}$	$\frac{1}{24}$	0
7	$m+1, n+1$	0	0	0	0	$\frac{1}{24}$	$\frac{1}{24}$

Table 3.1: Values for the integral $\int\int_{\triangle_k} N_i(x,y)N_j(x,y)dxdy$ for the j-th row of the matrix equation (3.8), where $i = 1, ..., 7$ and $k = 1, ..., 6$ in the patch shown on Figure 3.5.

		Element No					
Node No	Spatial Coordinates	1	2	3	4	5	6
1	$m-1, n-1$	$-\frac{1}{6}$	0	0	0	0	0
2	$m-1, n$	0	$-\frac{1}{6}$	$-\frac{1}{6}$	0	0	0
3	$m, n-1$	$\frac{1}{6}$	0	0	0	0	0
4	m, n	0	$\frac{1}{6}$	$\frac{1}{6}$	$-\frac{1}{6}$	$-\frac{1}{6}$	0
5	$m, n+1$	0	0	0	0	0	$-\frac{1}{6}$
6	$m+1, n$	0	0	0	$\frac{1}{6}$	$\frac{1}{6}$	0
7	$m+1, n+1$	0	0	0	0	0	$\frac{1}{6}$

Table 3.2: Values for the integral $\int\int_{\triangle_k} \frac{\partial N_i(x,y)}{\partial x} N_j(x,y)dxdy$ for the j-th row of the matrix equation (3.8), where $i = 1, ..., 7$ and $k = 1, ..., 6$ in the patch shown on Figure 3.5.

		Element No					
Node No	Spatial Coordinates	1	2	3	4	5	6
1	$m-1, n-1$	0	$-\frac{1}{6}$	0	0	0	0
2	$m-1, n$	0	$\frac{1}{6}$	0	0	0	0
3	$m, n-1$	$-\frac{1}{6}$	0	0	$-\frac{1}{6}$	0	0
4	m, n	$\frac{1}{6}$	0	$-\frac{1}{6}$	$\frac{1}{6}$	0	$-\frac{1}{6}$
5	$m, n+1$	0	0	$\frac{1}{6}$	0	0	$\frac{1}{6}$
6	$m+1, n$	0	0	0	0	$-\frac{1}{6}$	0
7	$m+1, n+1$	0	0	0	0	$\frac{1}{6}$	0

Table 3.3: Values for the integral $\int\int_{\triangle_k} \frac{\partial N_i(x,y)}{\partial y} N_j(x,y)dxdy$ for the j-th row of the matrix equation (3.8), where $i = 1, ..., 7$ and $k = 1, ..., 6$ in the patch shown on Figure 3.5.

		Element No					
Node No	Spatial Coordinates	1	2	3	4	5	6
1	$m-1, n-1$	0	0	0	0	0	0
2	$m-1, n$	0	$-\frac{1}{2}$	$-\frac{1}{2}$	0	0	0
3	$m, n-1$	0	0	0	0	0	0
4	m, n	0	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	0
5	$m, n+1$	0	0	0	0	0	0
6	$m+1, n$	0	0	0	$-\frac{1}{2}$	$-\frac{1}{2}$	0
7	$m+1, n+1$	0	0	0	0	0	0

Table 3.4: Values for the integral $\int\int_{\triangle_k} \frac{\partial N_i(x,y)}{\partial x} \frac{\partial N_j(x,y)}{\partial x} dxdy$ for the j-th row of the matrix equation (3.8), where $i = 1, ..., 7$ and $k = 1, ..., 6$ in the patch shown on Figure 3.5.

		Element No					
Node No	Spatial Coordinates	1	2	3	4	5	6
1	$m-1, n-1$	0	0	0	0	0	0
2	$m-1, n$	0	0	0	0	0	0
3	$m, n-1$	$-\frac{1}{2}$	0	0	$-\frac{1}{2}$	0	0
4	m, n	$\frac{1}{2}$	0	$\frac{1}{2}$	$\frac{1}{2}$	0	$\frac{1}{2}$
5	$m, n+1$	0	0	$-\frac{1}{2}$	0	0	$-\frac{1}{2}$
6	$m+1, n$	0	0	0	0	0	0
7	$m+1, n+1$	0	0	0	0	0	0

Table 3.5: Values for the integral $\int\int_{\triangle_k} \frac{\partial N_i(x,y)}{\partial y} \frac{\partial N_j(x,y)}{\partial y} dxdy$ for the j-th row of the matrix equation (3.8), where $i = 1, ..., 7$ and $k = 1, ..., 6$ in the patch shown on Figure 3.5.

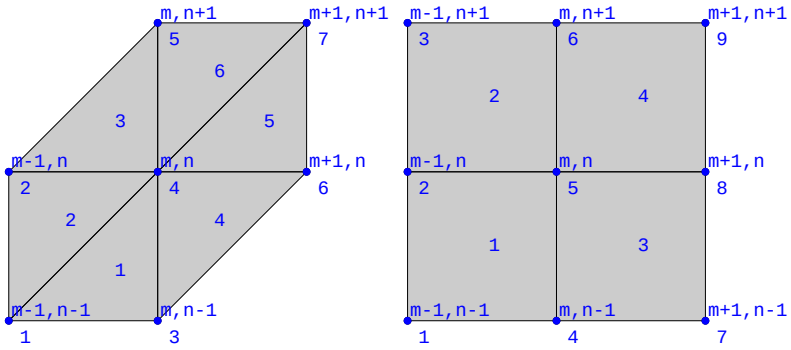


Figure 3.5: Finite element patches. In the middle of each element is printed its number. The nodes are shown with both their number and spatial indexes relatively to the node in the center, node 4 with indexes (m, n). The space step between the nodes on the same horizontal or vertical level is h.

$$
\begin{aligned}
&+\dot{g}_{m,n-1}(t) + \dot{g}_{m-1,n}(t) + \dot{g}_{m-1,n-1}(t)) = \\
u\frac{1}{h}\left(g_{m+1,n+1}(t) - g_{m,n+1}(t) + 2\left(g_{m+1,n}(t) - g_{m-1,n}(t)\right) + g_{m,n-1}(t) - g_{m-1,n-1}(t)\right) \\
v\frac{1}{h}\left(g_{m+1,n+1}(t) - g_{m+1,n}(t) + 2\left(g_{m,n+1}(t) - g_{m,n-1}(t)\right) + g_{m-1,n}(t) - g_{m-1,n-1}(t)\right) \\
K_x\frac{6}{h^2}\left(-g_{m-1,n}(t) + 2g_{m,n}(t) - g_{m+1,n}(t)\right) \\
K_y\frac{6}{h^2}\left(-g_{m,n-1}(t) + 2g_{m,n}(t) - g_{m,n-1}(t)\right).
\end{aligned}
\tag{3.10}
$$

If square elements are to be used, then the following formula can be derived:

$$
\begin{aligned}
&\frac{1}{3}\left(\dot{g}_{m+1,n+1}(t) + 4\dot{g}_{m+1,n}(t) + \dot{g}_{m+1,n-1}(t) + 4\dot{g}_{m,n+1}(t) + 16\dot{g}_{m,n}(t)\right. \\
&\left.+4\dot{g}_{m,n-1}(t) + \dot{g}_{m-1,n+1}(t) + 4\dot{g}_{m-1,n}(t) + \dot{g}_{m-1,n-1}(t)\right) = \\
u\frac{1}{h}\left(g_{m+1,n+1}(t) - g_{m-1,n+1}(t) + 4\left(g_{m+1,n}(t) - g_{m-1,n}(t)\right) + g_{m+1,n-1}(t) - g_{m-1,n-1}(t)\right) \\
v\frac{1}{h}\left(g_{m+1,n+1}(t) - g_{m+1,n-1}(t) + 4\left(g_{m,n+1}(t) - g_{m,n-1}(t)\right) + g_{m-1,n+1}(t) - g_{m-1,n-1}(t)\right) \\
K_x\frac{1}{h^2}\left(-g_{m-1,n-1}(t) - 4g_{m-1,n}(t) - g_{m-1,n+1}(t) + g_{m+1,n-1}(t) + 4g_{m+1,n}(t) + g_{m+1,n+1}(t)\right) \\
K_y\frac{1}{h^2}\left(-g_{m-1,n-1}(t) + g_{m-1,n+1}(t) + 4g_{m,n-1}(t) + 4g_{m,n+1}(t) - g_{m+1,n-1}(t) + g_{m+1,n+1}(t)\right),
\end{aligned}
\tag{3.11}
$$

(the square patch is shown on the right on Figure 3.5.)

The semi-discrete equations above are for the patches on regular mesh. The *Mathematica* appendix "Mass and Stiffness Matrices for Arbitrary Patch" demonstrates a package with which can be calculated the semi-discrete equations for general patches. Semi-discrete formulae similar to (3.11) can be found in Gresho and Sani [23].

3.2.5 Time stepping

The application of the $\theta-$method (see [28]) to handle numerically (3.8) gives

$$
g^{k+1} = g^k + \Delta t(\theta P^{-1}Ag^{k+1} + (1-\theta)P^{-1}Ag^k). \tag{3.12}
$$

We will apply the $\theta-$method with $\theta = 1/2$, which results in the well-known Crank-Nicholson method; (see again [28]). Note that (3.12) is called trapezoidal rule in the theory of numerical solution of ODE's.

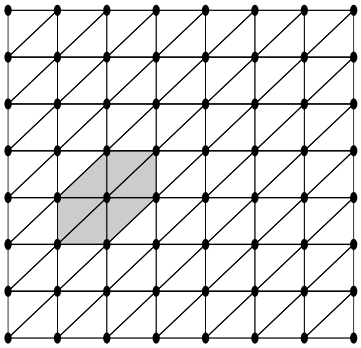


Figure 3.6: Triangulated domain. (The figure duplicates the left part of Figure 3.2 for convenience.)

3.3 The calculation of the operators

Here we will describe the algorithm used to compute the matrices P and A in Equation (3.8). The description refers to triangular elements, but it is valid for rectangular elements too.

These matrices present the scalar product $\langle \varphi, \psi \rangle$ and the bilinear operator $a(\varphi, \psi)$ in the space S, according to the basis $N_i(x,y)$, $i \in I$ defined by the mesh shown on Figure 3.6.[1] The algorithm to calculate P and A is row-wise.

The numbers $a(N_i, N_j)$ and $\langle N_i, N_j \rangle$ might be different from zero, only if the patches of the basis functions N_i and N_j overlap. The overlap of these patches always comprises one or two triangles (rectangles) if $i \neq j$. Also, the basis functions are constituted of functions which a non-zero on just one triangle. So on each triangle we will calculate the triangle's contributions to the numbers $a(N_i, N_j)$ and $\langle N_i, N_j \rangle$ and then we will sum them (the contributions) for each basis function N_i. Of course the summation should be just on the triangles constituting the patch of N_i. The contributions of the other triangles are zero. So we can rewrite the formula (3.7) for $a(N_i, N_j)$ as

$$\begin{aligned} a_{ji} = \quad & a(N_i, N_j) = \\ & \sum_{k \in E_j} \int\int_{\triangle_k} \left(\frac{\partial \left(u(x,y,t) N_i(x,y) \right)}{\partial x} N_j(x,y) + \frac{\partial \left(v(x,y,t) N_i(x,y) \right)}{\partial y} N_j(x,y) \right. \\ & \left. + K_x \frac{\partial N_i(x,y)}{\partial x} \frac{\partial N_j(x,y)}{\partial x} + K_y \frac{\partial N_i(x,y)}{\partial y} \frac{\partial N_J(x,y)}{\partial y} \right) dxdy, \end{aligned}$$

where E_j is an index set for the triangles that constitute the patch of N_j. The matrix entry a_{ji} is non-zero only if i belongs to the index set I_j of the neighbors of j. If we assume that at the time step we calculate $a(N_i, N_j)$ the wind components are constant within the triangles, we can rewrite the last formula as

$$a(N_i, N_j) = \begin{cases} \text{if } i \in I_j, & \sum_{k \in E_j} \left(u_{\triangle_k} \int\int_{\triangle_k} \frac{\partial N_i(x,y)}{\partial x} N_j(x,y) dxdy \right. \\ & \quad + v_{\triangle_k} \int\int_{\triangle_k} \frac{\partial N_i(x,y)}{\partial y} N_j(x,y) dxdy \\ & \quad + K_x \int\int_{\triangle_k} \frac{\partial N_i(x,y)}{\partial x} \frac{\partial N_j(x,y)}{\partial x} dxdy \\ & \quad \left. + K_y \int\int_{\triangle_k} \frac{\partial N_i(x,y)}{\partial y} \frac{\partial N_j(x,y)}{\partial y} dxdy \right), \\ \text{if } i \notin I_j, & 0 \end{cases} \tag{3.13}$$

The corresponding formula for the scalar product $\langle N_i, N_j \rangle$ is

$$\langle N_i, N_j \rangle = \begin{cases} \text{if } i \in I_j, & \sum_{k \in E_j} \int\int_{\triangle_k} N_i(x,y) N_j(x,y) dxdy, \\ \text{if } i \notin I_j, & 0. \end{cases} \tag{3.14}$$

[1] Note that $\langle \varphi, \psi \rangle$ is the scalar product of the Hilbert space V.

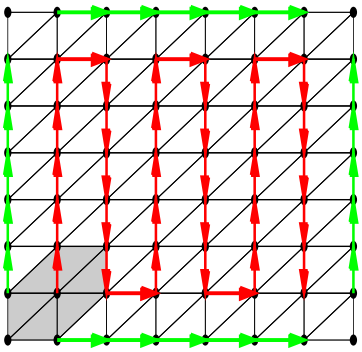


Figure 3.7: Calculation paths

The matrix P is constant, but the matrix A should be calculated at every new time step, and it is important to calculate this matrix fast. From the formula (3.13) we can see that when the patch for the node l_1 can be derived by translation from the patch of the node l_0, the integrals that correspond to $\int\int_{\triangle_k} \frac{\partial N_i(x,y)}{\partial x} N_j(x,y)dxdy$ and $\int\int_{\triangle_k} \frac{\partial N_i(x,y)}{\partial y} N_j(x,y)dxdy$ in formula (3.13), will be the same.[2] The evaluations of (3.13) for the node l_0 and l_1 will differ because of the different wind-field values on the triangles in their patches.

Let us consider the regular grid shown on Figure 3.6. We will say that two basis functions belong to the same orbit if their patches can be derived by translations from each other. For a regular triangular grid there are nine orbits of basis functions:

1. Inner basis functions. Their patches are composed of 6 triangles.
2. Left boundary basis functions.
3. Right boundary basis functions.
4. Top boundary basis functions.
5. Bottom boundary basis functions.
6. Corner functions (four different types).

We compute the integrals in (3.13) just once, for one basis function in each orbit, and reuse these values for the rest of the functions in the class. We can imagine that we start with a *representative basis function* and then we translate it over the domain triangulation in order to calculate all the numbers $a(N_i, N_j)$. Since a basis function is in one-to-one correspondence with the patch on which it is non-zero, we can speak for *representative patch* and *representative node*. Some possible paths are shown in Figure 3.7. The red one is for the inner basis functions, the green ones are for the boundary basis functions. In order to make derivation of the numbers $a(N_i, N_j)$ easier, we should easily derive the index sets E_j and I_j in (3.13) from the starting sets E_{j_0} and I_{j_0}, of the representative node j_0 of the orbit the node j belongs to.

The same strategy can be used on the mesh with refined part as the one shown on Figure 3.8. On that mesh, like in the mesh above, the paths should be over the nodes with equivalent patches.

3.4 The boundary conditions

We have not imposed any boundary conditions in the GFEM formulation. In the DEM's 2D FE advection code we use the same approach as in the DEM's 1D FE advection code: before each advection step we save the concentrations on the boundaries, and after the advection step we copy them back on those boundary points where the wind is blowing inside the computational domain.

[2] We should replace j with l_0 and l_1 respectively.

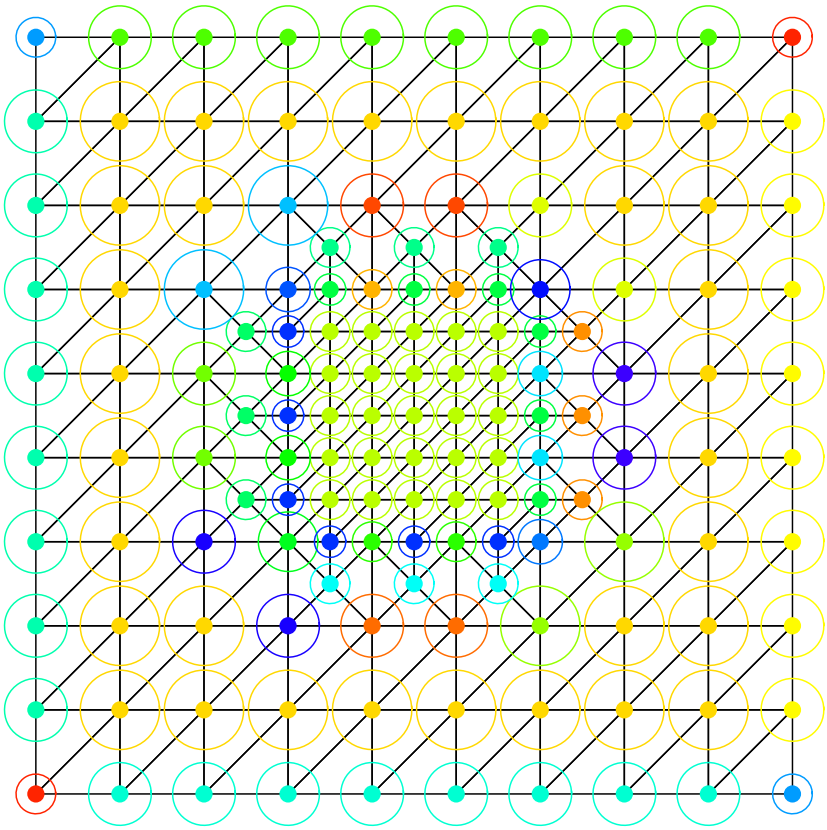


Figure 3.8: Orbits in locally refined mesh. The circles with the same color belong to equivalent patches.

3.5 GFEM stability, phase and group velocities

When we treat the advection-diffusion equation numerically, the method we use introduce errors which distort the true solution of the physical process. In this section we will analyze what the errors arising when the GFEM is used to solve the advection equation

$$\frac{\partial c}{\partial t} = -\frac{\partial (uc)}{\partial x} - \frac{\partial (vc)}{\partial y}. \tag{3.15}$$

In (3.15) c is the unknown function, while some velocity field in the fluid under consideration is defined by the functions u and v. The conclusions we will reach are valid for the advection-diffusion equation, which is easier for computer treatment (see [23, Ch.2]). The Fourier analysis technique which we use is described in detail by Vichnevetsky and Bowles in [52].

Assume that $u = W\cos\phi$ and $v = W\sin\phi$, where W and ϕ are constants. Then (3.15) can be rewritten (for this special choice of the wind velocities) as

$$\frac{\partial c}{\partial t} = -W\cos\phi\frac{\partial c}{\partial x} - W\sin\phi\frac{\partial c}{\partial y} \tag{3.16}$$

We will consider this equation in an infinite domain. To apply the Fourier analysis to this equation, we will change the regular grids defined in Section 3.2.2 to infinite regular grids with grid points $(x_m, y_n) = (mh, nh)$, where $m = ..., -1, 0, 1, ...$, $n = ..., -1, 0, 1, 2, ...$ and $h = x_m - x_{m-1} = y_n - y_{n-1} =$ *constant*. The meaning of the symbols $g_{m,n}(t)$ and g^k does not change, but $g(t)$ is defined now as $g(t) = \{g_{m,n}(t)\}_{m=-\infty,n=-\infty}^{\infty,\infty}$. Further, we will suppose that I is the set $\{(m,n) : -\infty < m < \infty, -\infty < n < \infty\}$ with some linear order[3] for its elements.

For the equation (3.16) we will define the important concepts of phase velocity and group velocity. In the next section we will study the behavior of these quantities for the GFEM introduced in Section 3.2.1. The definitions given below are based on the use of continuous and discrete Fourier transforms of the unknown function c, and on the grid-functions obtained from c on the regular mesh chosen on the space domain. The discrepancies between the phase and group velocities obtained by using the continuous Fourier transforms and the phase and group velocities obtained by using discrete Fourier transforms will give us some information about the ability of GFEM to produce good approximations.

3.5.1 Semi-discrete equations

To investigate the properties of GFEM for (3.16), we will drop the diffusion parts of the the equations (3.10) and (3.11) and obtain the corresponding equations:

- for triangles

$$\begin{aligned} &3\dot{g}_{m,n}(t) + \frac{1}{2}\left(\dot{g}_{m+1,n+1}(t) + \dot{g}_{m+1,n}(t) + \dot{g}_{m,n+1}(t)\right. \\ &\left.+\dot{g}_{m,n-1}(t) + \dot{g}_{m-1,n}(t) + \dot{g}_{m-1,n-1}(t)\right) = \\ &u\frac{1}{h}\left(g_{m+1,n+1}(t) - g_{m,n+1}(t) + 2\left(g_{m+1,n}(t) - g_{m-1,n}(t)\right) + g_{m,n-1}(t) - g_{m-1,n-1}(t)\right) \\ &v\frac{1}{h}\left(g_{m+1,n+1}(t) - g_{m+1,n}(t) + 2\left(g_{m,n+1}(t) - g_{m,n-1}(t)\right) + g_{m-1,n}(t) - g_{m-1,n-1}(t)\right) \end{aligned} \tag{3.17}$$

- for squares:

$$\begin{aligned} &\frac{1}{3}\left(\dot{g}_{m+1,n+1}(t) + 4\dot{g}_{m+1,n}(t) + \dot{g}_{m+1,n-1}(t) + 4\dot{g}_{m,n+1}(t) + 16\dot{g}_{m,n}(t)\right. \\ &\left.+4\dot{g}_{m,n-1}(t) + \dot{g}_{m-1,n+1}(t) + 4\dot{g}_{m-1,n}(t) + \dot{g}_{m-1,n-1}(t)\right) = \\ &u\frac{1}{h}\left(g_{m+1,n+1}(t) - g_{m-1,n+1}(t) + 4\left(g_{m+1,n}(t) - g_{m-1,n}(t)\right) + g_{m+1,n-1}(t) - g_{m-1,n-1}(t)\right) \\ &v\frac{1}{h}\left(g_{m+1,n+1}(t) - g_{m+1,n-1}(t) + 4\left(g_{m,n+1}(t) - g_{m,n-1}(t)\right) + g_{m-1,n+1}(t) - g_{m-1,n-1}(t)\right) \end{aligned} \tag{3.18}$$

[3] A relation in a set that defines for any pair of elements which is the smaller one is called a linear order for that set.

The corresponding patches are shown on Figure 3.5.

Let us write the equations (3.17) and (3.18) for every mesh-point of the regular infinite meshes defined above. These equations can be rewritten in matrix form by using two banded matrices P and A:

$$P\frac{dg(t)}{dt} = Ag(t). \tag{3.19}$$

The rows of the constant matrix P are formed from the coefficients of $\dot{g}_{m,n}(t)$ in the left hand side of (3.17) or (3.18). The rows of the matrix A are formed from the coefficients of $g_{m,n}(t)$ in the right hand side of (3.17) or (3.18). The matrix A is in general time dependant, but since we consider here constant winds it is constant. Both matrices P and A in (3.19) are infinite.

3.5.2 Sinusoidal solutions

The main advantage of using the Fourier method when solving (3.16) can be emulated with the heuristic concept of "sinusoidal trial solution". For equation (3.16) we need two dimensional trial solutions. The main ideas are explained in detail in the book by Vichnevetsky and Bowles, [52].

The sinusoidal function $c_\omega(x,y,t) = \eta_\omega(t)e^{i\,\omega(\cos\phi\,x+\sin\phi\,y)}$, $\phi = \arctan(v/u)$ is a solution of (3.16) provided

$$\eta_\omega(t) = \eta_\omega(0)e^{-i\,W\,\omega\,t}.$$

The vector $s = [...,e^{i\,\omega(\cos\phi\,x_m+\sin\phi\,y_n)},...]^T$, $(m,n) \in I$ is an eigenvector of P and A;(see [52, page 7].) The eigenvalues corresponding to the eigenvector s can be calculated with $Ps = \mu s$ and $As = \lambda s$. Now the sinusoidal function $g_{\omega,m,n}(t) = \nu_\omega(t)e^{i\,\omega(\cos\phi\,x_m+\sin\phi\,y_n)}$, $\phi = \arctan(v/u)$ is a solution of (3.19) provided

$$\frac{d\nu_\omega}{dt} = \widehat{A}(\omega,\phi)\nu_\omega\,, \tag{3.20}$$

where $\widehat{A}(\omega,\phi)$ is defined by

$$\widehat{A}(\omega,\phi) = \frac{\lambda}{\mu}.$$

We find after solving (3.20) that

$$\begin{aligned} g_{\omega,m,n}(t) &= \nu_\omega(0)e^{\widehat{A}(\omega,\phi)}e^{i\,\omega(\cos\phi\,x_m+\sin\phi\,y_n)} \\ &= \nu_\omega(0)e^{Re\widehat{A}(\omega,\phi)}e^{i\,\omega(\cos\phi\,x_m+\sin\phi\,y_n)+Im\widehat{A}(\omega,\phi)}. \end{aligned} \tag{3.21}$$

We can rewrite $c_\omega(x,y,t)$ as

$$c_\omega(x,y,t) = \eta_\omega(t)e^{i\,\omega(\cos\phi\,x+\sin\phi\,y)} = \eta_\omega(t)e^{i\,\omega\,\vec{r}.\vec{1}},$$

where $\vec{r} = \begin{bmatrix} x \\ y \end{bmatrix}$ and $\vec{1} = \begin{bmatrix} \cos\phi \\ \sin\phi \end{bmatrix}$. The vector $\vec{1}$ is called *the directional vector* of the wave $c_\omega(x,y,t)$. The wavelength of $c_\omega(x,y,t)$, measured in $\vec{1}$ direction, is $\lambda = 2\pi/\omega$.

3.5.3 Definitions of amplitude and phase velocity

Whereas the amplitude of the exact sinusoidal solutions remains constant in time, $|\eta_\omega(t)| = |\eta_\omega(0)|$, the amplitude of the numerical sinusoidal solutions does not necessary remain constant. We have

$$|\nu_\omega(t)| = |\nu_\omega(0)|\,e^{Re\widehat{A}(\omega,\phi)}.$$

The semi-discretization (3.19) is called *conservative* if $Re\widehat{A}(\omega,\phi) = 0$ for all ω. When $Re\widehat{A}(\omega,\phi) \leq 0$ for all ω, and the strict inequality holds for at least one ω, then the semi-discrete equation is called *dissipative*. When $Re\widehat{A}(\omega,\phi) > 0$ for some ω, then the method is unstable .

For conservative methods, the only discrepancy between numerical sinusoidal solutions and their exact counterparts is in the velocity at which they propagate. This discrepancy is called *phase velocity error*. If we define the function

$$W^*(\omega,\phi) = -\frac{Im\widehat{A}(\omega,\phi)}{\omega},$$

then we may rewrite (3.21) as

$$g_{\omega,m,n}(t) = \nu_\omega(0)e^{i\,\omega(\cos\phi\,x_m+\sin\phi\,y_n)-W^*(\omega\,\phi)}.$$

This shows that W^* is the velocity of propagation of the numerical sinusoidal solutions. W^* is called *phase velocity*. We see that, in contrast of the exact sinusoidal solutions, W^* depends on ω and ϕ.

3.5.4 Definitions of amplitude and phase velocity for full discretizations

When the Crank-Nicolson time integration method is used we obtain the discrete equation

$$\nu_\omega^{n+1} = \nu_\omega^n + \frac{1}{2}\Delta t(\widehat{A}(\omega,\phi)\nu_\omega^{n+1} + \widehat{A}(\omega,\phi)\nu_\omega^n).$$

It makes sense to look for solutions of this equation in the form

$$\frac{\nu_\omega^{n+1}}{\nu_\omega^n} = z(\omega,\phi) = constant.$$

$z(\omega,\phi)$ is called an *amplification factor* for the discrete algorithm (3.12). The true ratio for sinusoidal solutions of the continuous equation (3.16)

$$\frac{c_\omega(x,y,t+\Delta t)}{c_\omega(x,y,t)} = e^{i\,\Delta t\,\omega(\cos\phi\,x+\sin\phi\,y)},$$

is replaced with the numerical approximation

$$\frac{g_{\omega,m,n}(t+\Delta t)}{g_{\omega,m,n}(t)} = z(\omega,\phi).$$

Comparing these two expressions and separating the amplitude $|z(\omega,\phi)|$ and the phase $\angle z(\omega,\phi)$, lead us to the following result

$$z(\omega,\phi) = |z(\omega,\phi)|\,e^{i\angle z(\omega,\phi)} = |z(\omega,\phi)|\,e^{-i\omega W^*(\omega,\phi)\Delta t}.$$

The last formula defines the phase velocity of the full discretization as

$$W^*(\omega,\phi) = \frac{-\angle z(\omega,\phi)}{\omega\Delta t}.$$

3.5.5 Definition of group velocity

From the phase velocity we can calculate *the group velocity*, which is defined as

$$V(\omega,\phi) = \frac{d(\omega\,W^*(\omega,\phi))}{d\omega}.$$

As Lighthill [32] explains

> "Perhaps the most fundamental property of the group velocity may be mentioned at once ... The energy of sinusoidal waves is propagated *not* at the wave speed W, but at the group velocity V."[4]

The theory of group velocity is important when explaining the propagation of spurious, short-wavelength oscillations which appear near discontinuities and sharp gradients in discrete approximations of hyperbolic equations. More about group velocity in numerical methods can be read in [52], [54] and [23].

[4]The symbols c and U from the original text were changed to W and V for compliancy with the rest of the notation in the thesis.

3.5.6 Details about the application of *Mathematica*

The theoretical material in the previous section can be used to compute symbolically expressions for the amplification factor $z(\omega,\phi)$. We did all the computations with *Mathematica* [56]. Therefore, we will start with a short description of some details about the particular application of *Mathematica* in our case.

First we should obtain the expression for $\widehat{A}(\omega\,\phi)$. It is very convenient to implement the difference operators involved in the full discretizations. Their associativity comes automatically, but their linearity should be programmed. With them we obtain full discretizations for the symbol `c[m,n,j]` which stands for $c(mh,nh,j\Delta t)$. The expressions of these full discretizations involve `c[m`$\pm k_1$`,n`$\pm k_2$`,j`$\pm k_3$`]`. We can use pattern replacement to replace `c[m`$\pm k_1$`,n`$\pm k_2$`,j`$\pm k_3$`]` with $\nu((j\pm k_3)\Delta t)$`*Exp[`$i\omega((m\pm k_1)h\cos\phi+(n\pm k_2)h\sin\phi)$`]`. We derived (3.17) and (3.18) with our *Mathematica* code for checking of our GFEM implementation; see the *Mathematica* appendix "Mass and Stiffness Matrices for Arbitrary Patch".

With $\widehat{A}(\omega\,\phi)$ we can compute $z(\omega,\phi)$. The expressions for $z(\omega,\phi)$ could be quite long and their simplification can take some time.

After $z(\omega,\phi)$'s calculation, if we assume that the units for the grid step, the wind velocity, and the time step are respectively *meter*, *meter/second*, and *second*, $z(\omega,\phi)$ can be put in dimensionless form in the following way:

1. First we replace every h with `M[`h`,Meter]`, W with `M[`W`,Meter/Second]`, Δt with `M[`Δt`,Second]`;
2. Then we replace every `Meter` with $\frac{1}{h}$, and every `Second` with $\frac{c}{h}$;
3. Last we replace every expression `M[something1,something2]` with `something1*something2`.

Mathematica does not evaluate the functions `Arg` and `Abs` for non-numerical expressions, so we had to program how `Arg[`$z(\omega,\phi)$`]` and `Abs[`$z(\omega,\phi)$`]` are calculated.

3.5.7 Studying some properties of GFEM

Theorem 3.5.1 *The expression of $z(\omega,\phi)$ related to the GFEM in Section 3.2.1 is*

1. *for regular triangular mesh:*

$$z(\omega,\phi)=\frac{z1(\omega,\phi)-\frac{W\Delta t}{h}\,(z2(\omega,\phi)\,\cos(\phi)+z3(\omega,\phi)\,\sin(\phi))}{z1(\omega,\phi)+\frac{W\Delta t}{h}\,(z2(\omega,\phi)\,\cos(\phi)+z3(\omega,\phi)\,\sin(\phi))},$$

where

$$\begin{aligned}
z1(\omega,\phi) &= 1+e^{i\,\omega\,\cos(\phi)}+e^{i\,\omega\,\sin(\phi)}+6\,e^{i\,\omega\,(\cos(\phi)+\sin(\phi))}+e^{2\,i\,\omega\,(\cos(\phi)+\sin(\phi))}\\
&\quad +e^{i\,\omega\,(2\,\cos(\phi)+\sin(\phi))}+e^{i\,\omega\,(\cos(\phi)+2\,\sin(\phi))},\\
z2(\omega,\phi) &= -1+e^{i\,\omega\,\cos(\phi)}-2\,e^{i\,\omega\,\sin(\phi)}+e^{2\,i\,\omega\,(\cos(\phi)+\sin(\phi))}\\
&\quad +2\,e^{i\,\omega\,(2\,\cos(\phi)+\sin(\phi))}-e^{i\,\omega\,(\cos(\phi)+2\,\sin(\phi))},\\
z3(\omega,\phi) &= \left(-1+e^{i\,\omega\,\sin(\phi)}\right)\left(1+2\,e^{i\,\omega\,\cos(\phi)}+e^{i\,\omega\,(\cos(\phi)+\sin(\phi))}\left(2+e^{i\,\omega\,\cos(\phi)}\right)\right);
\end{aligned}$$

2. *for square mesh:*

$$z(\omega,\phi)=\frac{1-\frac{3\,i}{2}\,\frac{W\Delta t}{h}\left(\frac{\cos(\phi)\,\sin(\omega\,\cos(\phi))}{2+\cos(\omega\,\cos(\phi))}+\frac{\sin(\phi)\,\sin(\omega\,\sin(\phi))}{2+\cos(\omega\,\sin(\phi))}\right)}{1+\frac{3\,i}{2}\,\frac{W\Delta t}{h}\left(\frac{\cos(\phi)\,\sin(\omega\,\cos(\phi))}{2+\cos(\omega\,\cos(\phi))}+\frac{\sin(\phi)\,\sin(\omega\,\sin(\phi))}{2+\cos(\omega\,\sin(\phi))}\right)}. \tag{3.22}$$

Moreover, for square mesh the discretization is conservative, i.e. $z(\omega,\phi)=1$.

Proof: The first part of the proof has been done by using *Mathematica* to derive the symbolical expressions for $z(\omega,\phi)$ for the method in subsection 3.2. It can easily be proved from (3.22) that for square elements $|z(\omega,\phi)|=1$ – the denominator of $z(\omega,\phi)$ in (3.22) is the complex conjugate of the numerator.

Figures 3.9 and 3.11 show the polar plots of the phase and group velocities calculated for a regular triangular mesh and for a square mesh respectively. The corresponding mesh patches are shown on Figure 3.5. The velocities are calculated for the values 0.2, 1, 2, of the Courant number, $W\,\Delta t/h$. Figures 3.10 and 3.12 show by using logarithmic-linear plots how $W^*(\omega)$ and $V(\omega)$ change with $\lambda = 2\pi/\omega$, for the angles where the distortions are the greatest – for regular triangular mesh they are respectively $\pi/4$ and $\pi/2$; for square mesh $\pi/2$ and $\pi/2$.

From these plots we can draw the following conclusions and observations related to GFEM schemes.

- We can see the famous (infamous) anisotropy property of the triangle meshes: short-length waves propagate faster in the direction perpendicular to the triangles hypotenuses.
- The group velocity for wavelength $2h$ is negative and it is 3 times greater in magnitude than the wind speed.
- The negative group velocity of the $2h$-wave is very anisotropical: on Figure 3.9 (Figure 3.11) it is along the axes, which from the triangular (square) element patch on Figure 3.5 means that in general it is along the lines from node 3 to 5, and from node 2 to 6 (from node 4 to 6, and from node 2 to 8).
- Waves with wavelength $3h$ have group velocity 0. This can be seen on Figures 3.10 and 3.12.

An important property of the discretization (3.12) is that it is conservative, that is, $|z(\omega,\phi)| = 1$. There were no success in the tries to prove (with the automatic symbolic transformations made by *Mathematica*) that $|z(\omega,\phi)| = 1$ holds for triangular elements. Nevertheless, it can be seen on Figure 3.13 that $|z(\omega,\phi)|$ is at least very close to 1. There are ten different circles on that figure, but since all of them practically coincide we can see just the last one.

3.5.8 Concluding remarks

The material presented in this section came from the efforts to judge the quality of the developed GFEM advection submodel, and the failure to find descriptions how the used Fourier technique is applied to the schemes we were concerned – GFEM on triangles and squares. Most of the literature provides discussions just for the semi-discrete equations, or full-discretizations for 1D problems/schemes. If we want to compare finite element discretizations with splitting methods, we should use the full discretization. Such a comparison for 2D schemes is presented in [5].

It is controversial whether we can use phase and group velocity results to judge how good are the GFEM schemes. Nevertheless, this approach gives insight of the behavior of the numerical methods of this type.

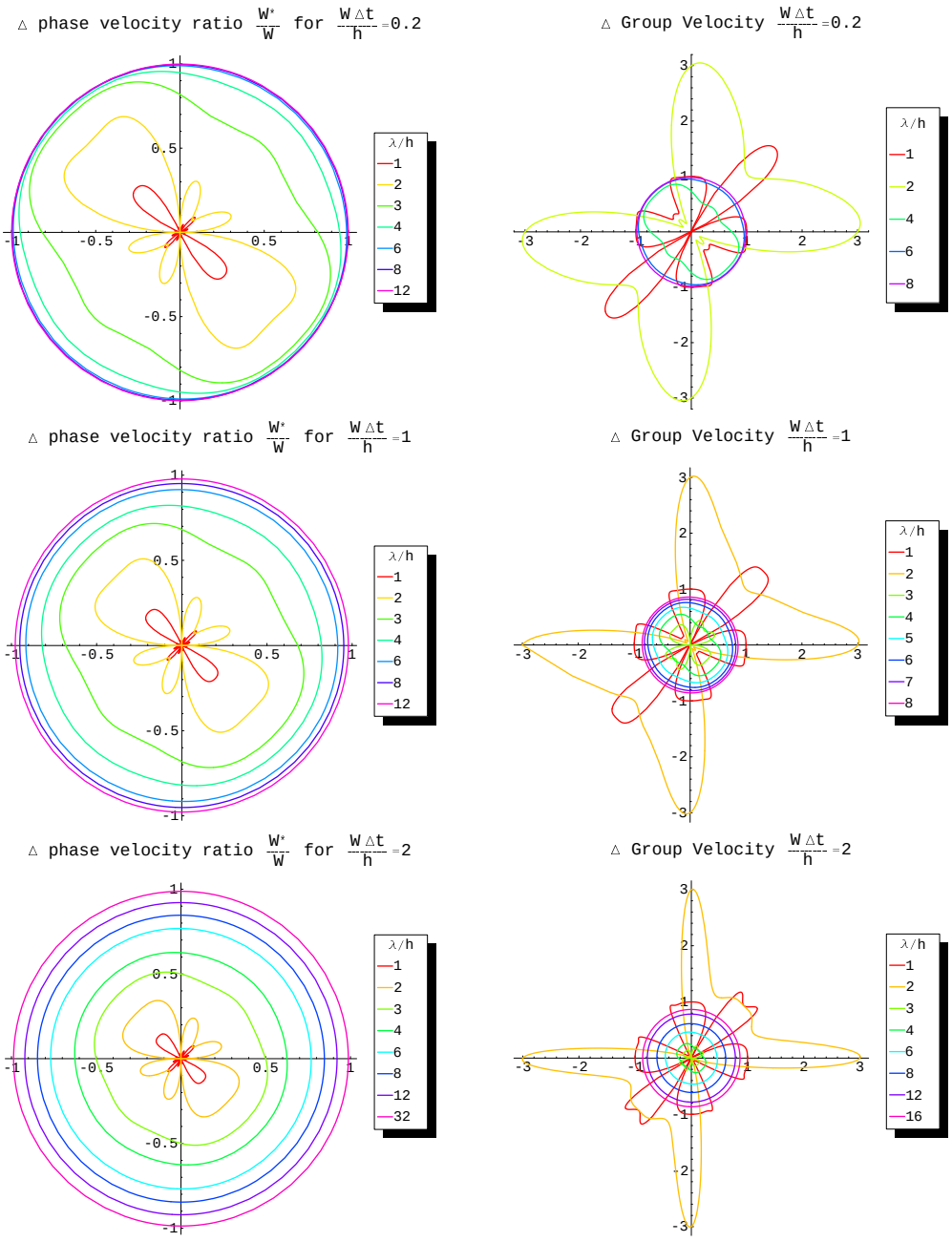


Figure 3.9: Phase and group velocities for GFEM on triangular mesh for the values 0.2, 1, 2, of the Courant number, $\frac{W\Delta t}{h}$.

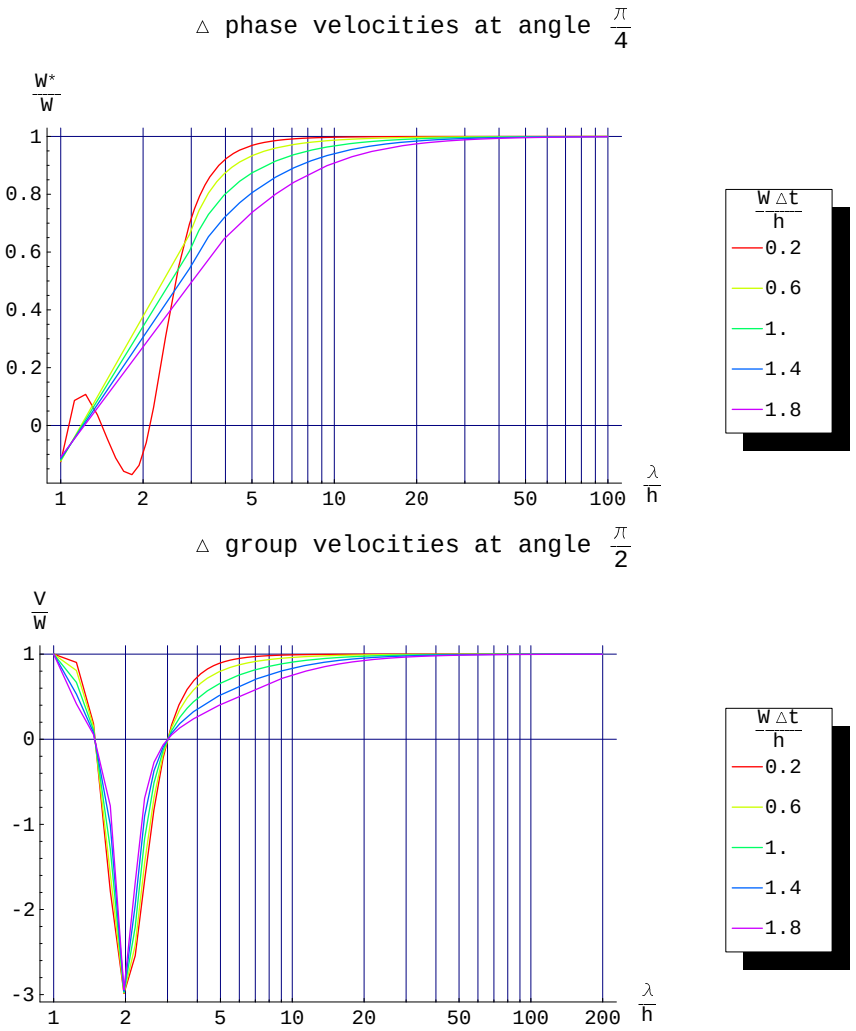


Figure 3.10: Logarithmic-linear plots of the phase velocities at angle $\frac{\pi}{4}$ and group velocities at angle $\frac{\pi}{2}$ for the Courant numbers 0.2, 0.6, 1.0, 1.4, 1.8.

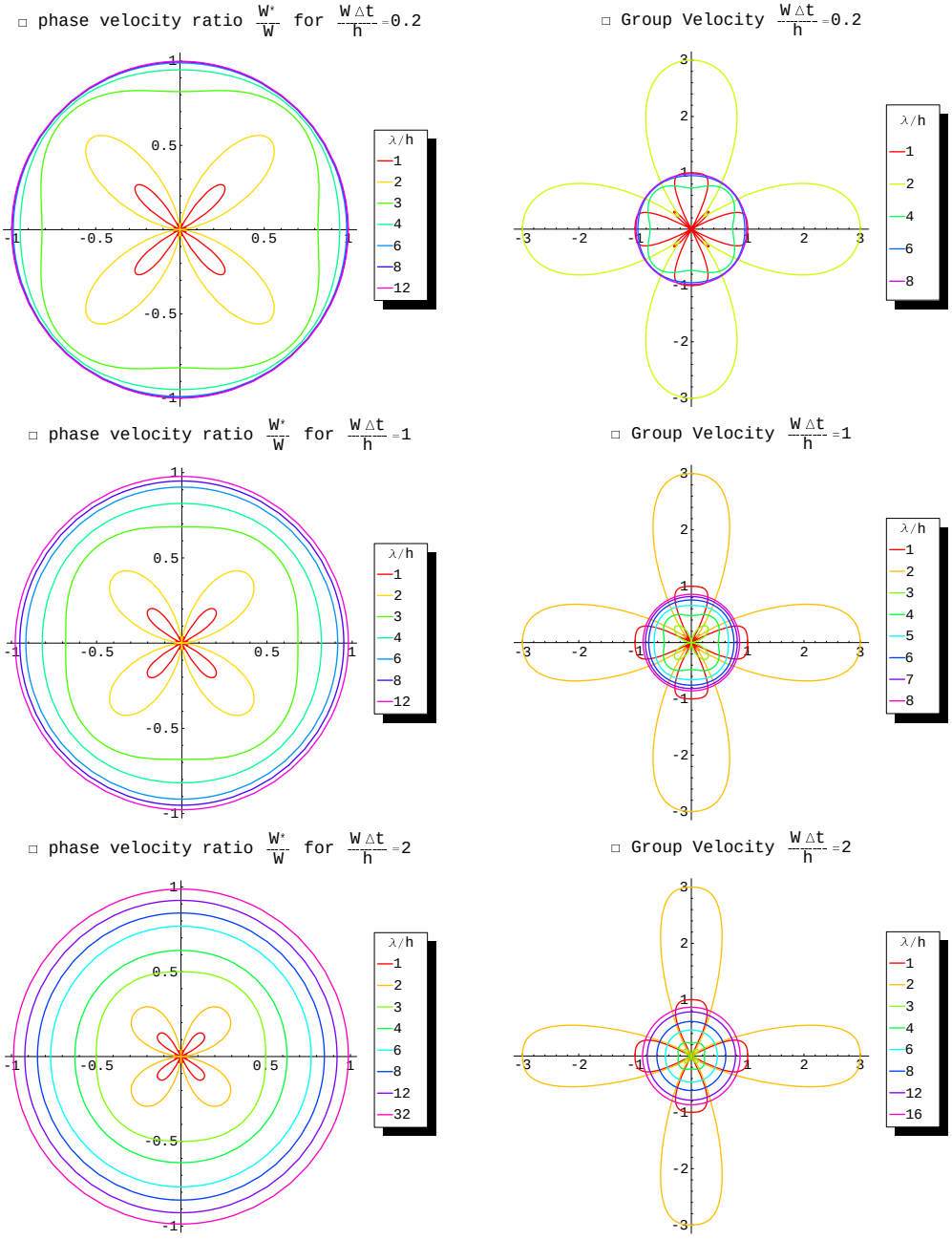


Figure 3.11: Phase and group velocities s for GFEM on rectangular mesh for the values 0.2, 1, 2, of the Courant number, $\frac{W\Delta t}{h}$.

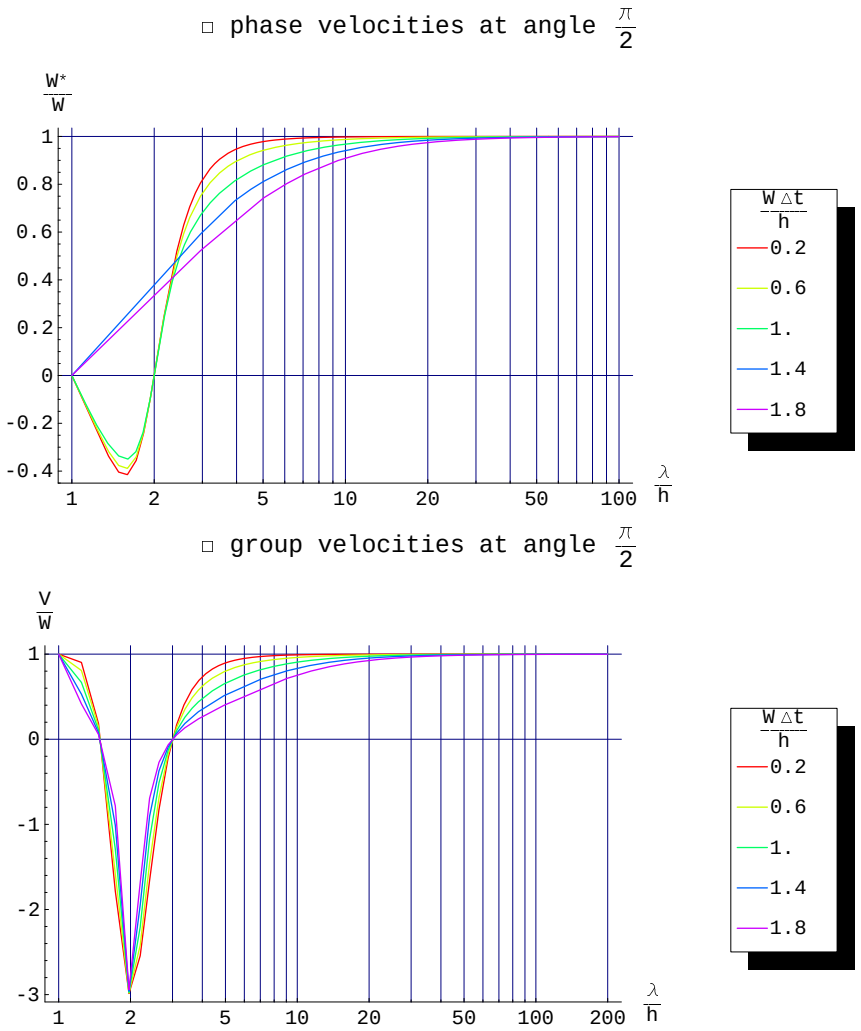


Figure 3.12: Logarithmic-linear plots of the phase and group velocities at angle $\frac{\pi}{2}$ for the Courant numbers 0.2, 0.6, 1.0, 1.4, 1.8.

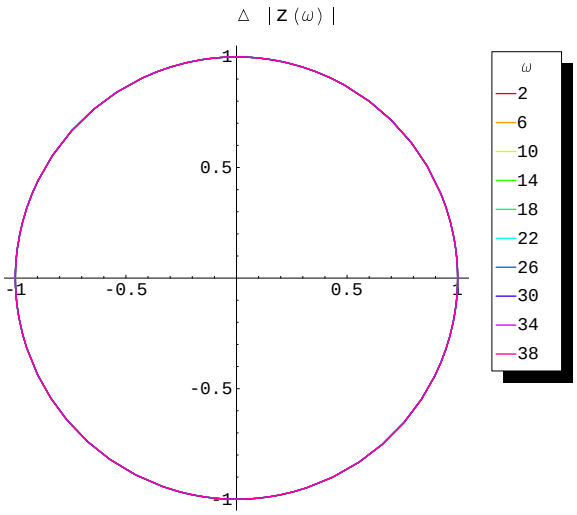


Figure 3.13: $|z(\omega, \phi)|$ for ten different ωs.

Chapter 4

Tests and Experiments

Hey man you know I'm really okay
The gun in my hand will tell you the same

– OFFSPRING, *"Bad Habit"*,
Smash, 1994

4.1 Group velocity tests

The test results presented in this section are intended to demonstrate the negative group velocity, calculated in Section 3.5. They should also bring more insight how the numerical errors introduced by GFEM look like. We will restrict ourself to experiments on just triangular regular meshes. The mesh is 96×96 with space step $\Delta x = 50km$. The constant wind from left to right in x-direction has speed $30m/s = 108km/h$.

Let us emphasize that these pure advection tests are intended to investigate pathological cases, in which significant part of the initial conditions consist of high frequency Fourier modes. They were carried out, because it is interesting to see the two-dimensional counterparts of the one dimensional group velocity tests in [23]. In the (rare) practical cases, when one meets initial conditions with significant high frequency Fourier modes, he or she can use some artificial diffusion to improve the qualities of the solution.

4.1.1 Cut cone test

Our first test is for the transport in x-direction, from left to right, of a cut, very sharp cone. The 3D view of the initial condition is the top picture on Figure 4.3. On Figures 4.1 and 4.2 is shown a profile of the initial condition and its discrete Fourier transform. The profile of the propagation is shown on Figure 4.4.

The definition of the discrete Fourier transform used to draw Figure 4.2 is

$$v_s = \frac{1}{\sqrt{n}} \sum_{r=1}^{n} u_r e^{2\pi i (r-1)(s-1)/n}.$$

We can see, that the higher the index s is, the higher is the wave number (the frequency of sin and cos) in v_s. The mesh can resolve the waves with wavelength $\geq 2\Delta x$, which correspond to the lower(left) half of the spectrum: the rest of them are aliased; (see [23, 48] for aliasing errors discussions) . So, when we look at Figures 4.3 and 4.4, we should have in mind the lower part of the spectrum, $s < n/2$.

4.1.2 $2\Delta x$-wave test

Our second test is for the transport in x-direction, left to right, of the $2\Delta x$ wave, the one that propagates to the left (with negative speed) on Figure 4.8. The 3D view of the initial condition is the top picture

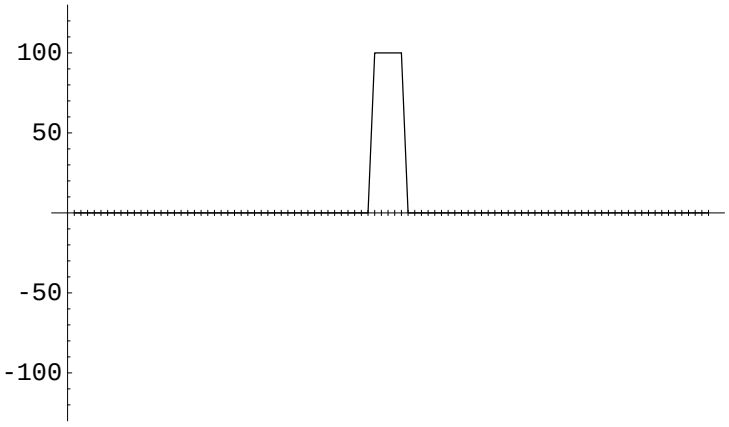


Figure 4.1: Cut cone initial condition. The initial condition u_0 was formed on 96×96 grid; see the top picture on Figure 4.3. Here is shown the middle slice of it, $u_0[1:96, 48]$.

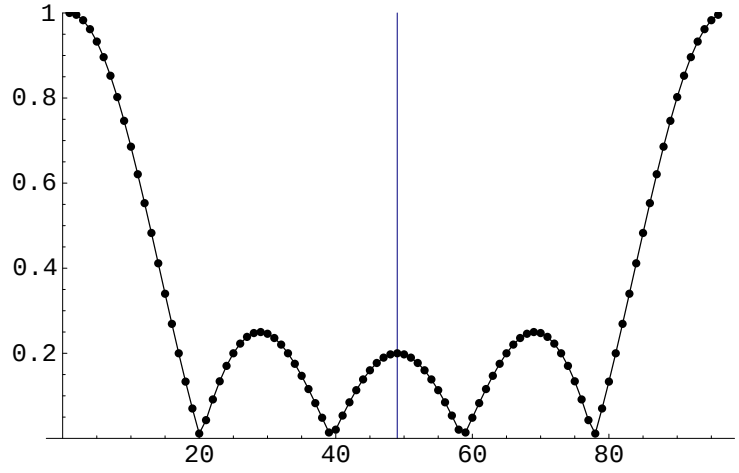


Figure 4.2: The normalized discrete Fourier transform of the middle slice of the cut cone initial condition. The mesh 'see' just the waves that correspond to the lower(left) half of the discrete spectrum.

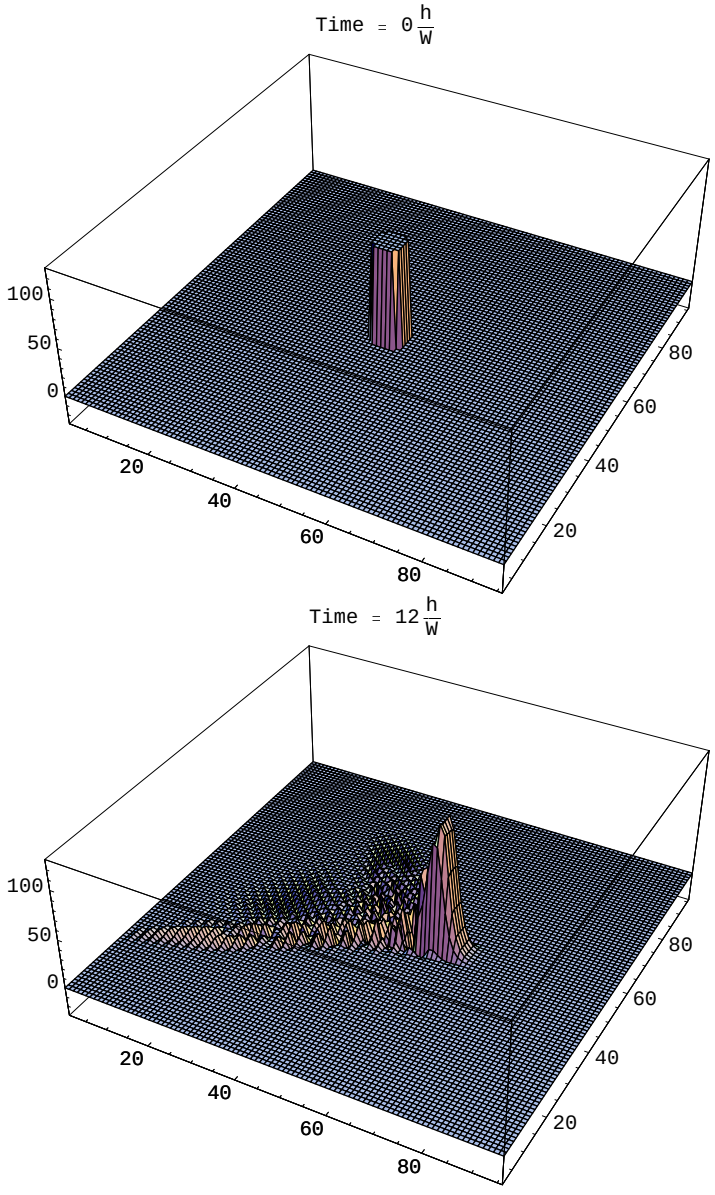


Figure 4.3: Propagation of the cut cone. The wind is in x-direction, from left to right.

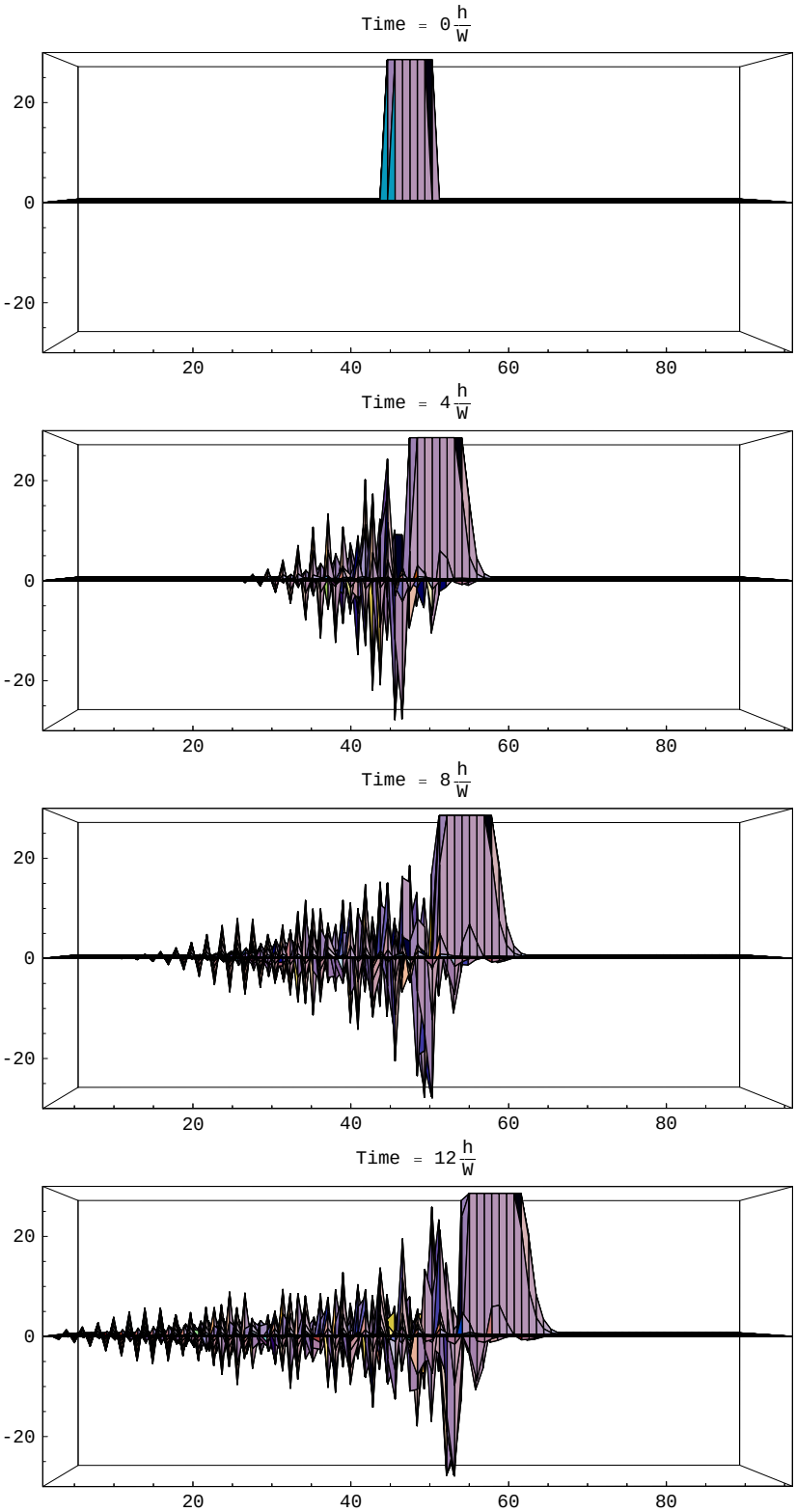


Figure 4.4: Propagation of the cut cone, x-profile. The wind is in x-direction, left to right. Note the wave propagating backwards with 3 times greater speed. The height of the cut cone is 100, on plot range on this figure is from -30 to 30.

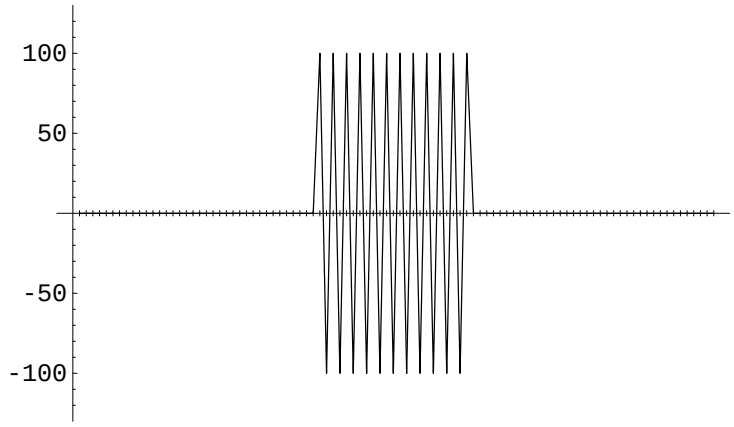


Figure 4.5: $2\Delta x$-wave initial condition. The initial condition u_0 was formed on 96×96 grid; see the top picture on Figure 4.7. Here is shown the middle slice of it, $u_0[1:96, 48]$.

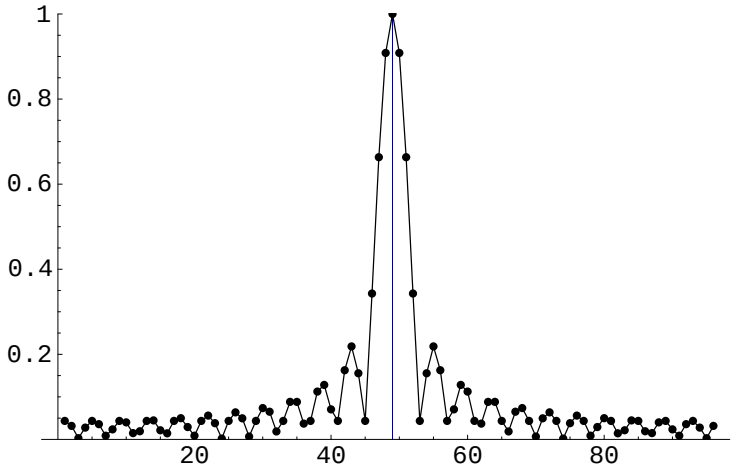


Figure 4.6: The normalized discrete Fourier transform of the middle slice of the $2\Delta x$-wave initial condition. The mesh 'see' just the waves that correspond to the lower(left) half of the discrete spectrum.

on Figure 4.4. On Figures 4.5 and 4.6 is shown a profile of the initial condition and its discrete Fourier transform. The profile of the propagation is shown on Figure 4.8.

We can see from Figure 4.6 the dominance of the $2\Delta x$ eigenvector; see Section 3.5.2, the wave directional vector is $\vec{1} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$. Hence the solution is backward translation at the negative group velocity found in Section 3.5.

4.2 Inner boundaries

Another important question is how the numerical scheme propagates waves through the inner boundaries of a local refinement. A two dimensional analog of the theoretical analysis presented by Vichnevetsky in [53] is quite interesting to carry on, though it involves *huge* symbolic computations. Nevertheless, we can answer this question comparing results of computations over regular and over refined grids.

We made four experiments each of which is a translation of a cone in $x-$direction with wind speed $10m/s$. The meshes that correspond to the experiments are:

1. regular grid 48×48 and grid step size $h = 50000m$;

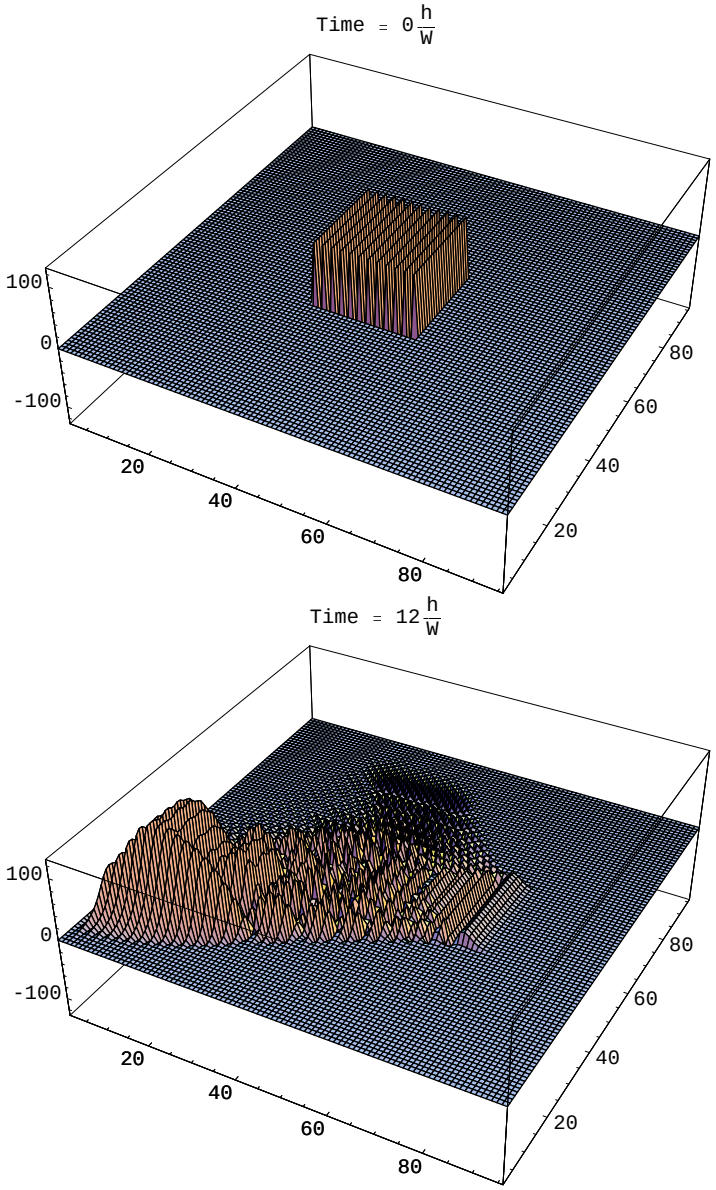


Figure 4.7: Propagation of the $2\Delta x$-wave. The wind is in x-direction, left to right.

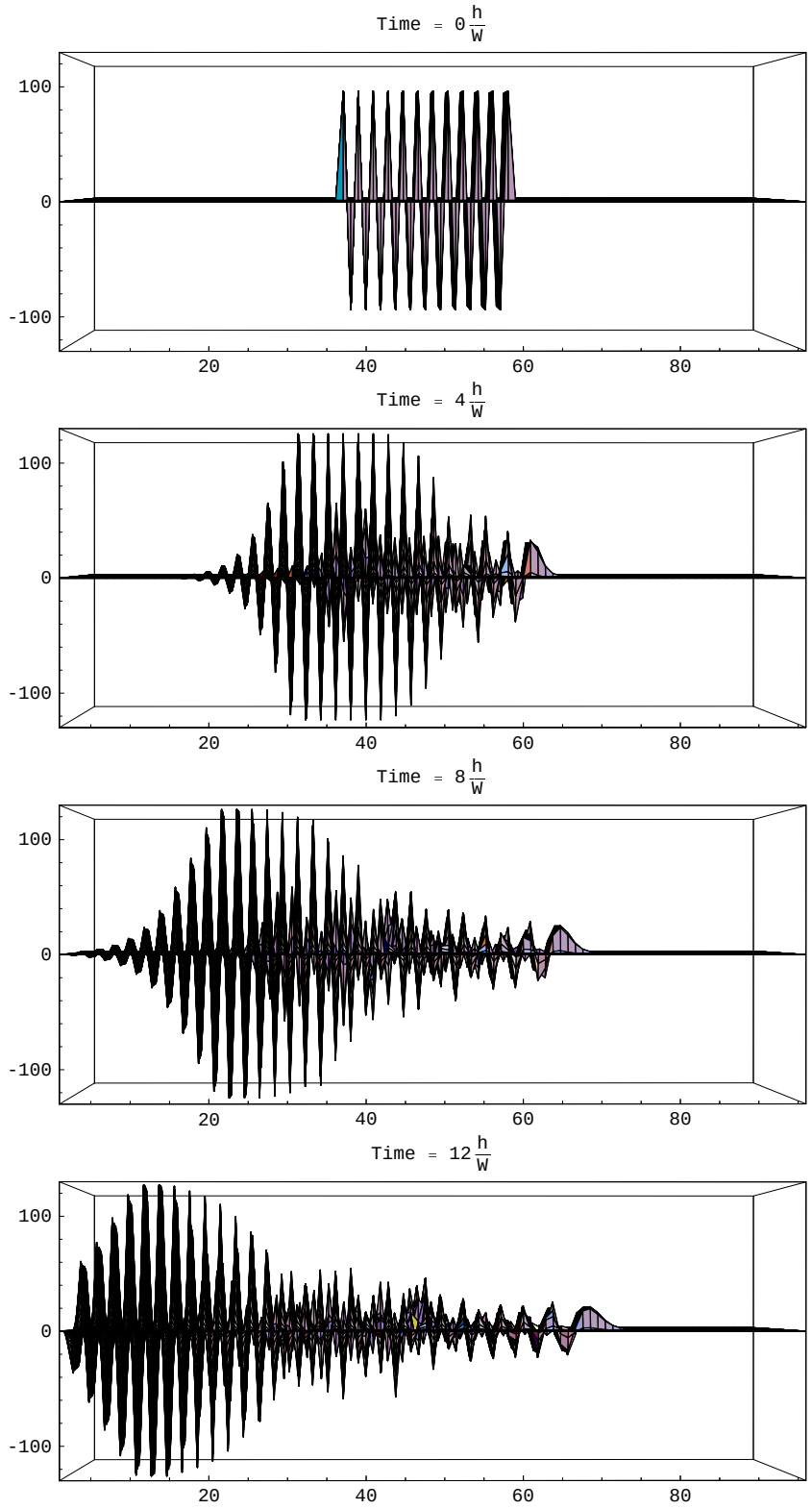


Figure 4.8: Propagation of the $2\Delta x$-wave, x-profile. The wind is in x-direction, left to right. We can see that the wave is moving backwards with 3 times greater velocity.

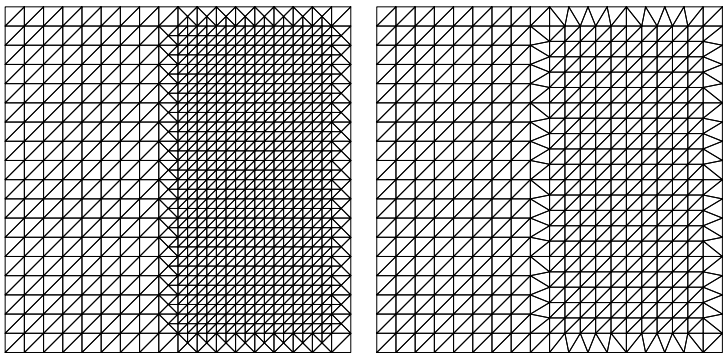


Figure 4.9: Reflection meshes. Left: for experiment 2.; right for experiment 3.

2. grid with $1:2$ refinement on the right half of the 48×48 regular grid above, item 1;

3. grid with $4:5$ refinement on the right half of the 48×48 regular grid above, item 1;

4. regular grid 96×96 and grid step size $h = 25000m$.

All experiments start with the cone's pick placed at the left half of the mesh on the point $x = 5.75 \times 10^8$, $y = 1.175 \times 10^8$. Since the translation of the cone in 1. coincides with translations on the left halfs of the meshes in 2. and 3. (see Figure 4.9), we compare the results of experiments 2. and 3. with this of 1. We can see from Figure 4.10 that the cone picks for 1. and 2. differ with less that 1%, and the picks for 1. and 3. differ less than 2.5%. We made experiment 4. just for information.

We can make experiments with transportation of the cone back and forth through the inner boundary. The results are summarized on Tables 4.1, 4.2, 4.3, 4.4.

Our conclusion is that the inner boundaries dump the cone's pick to acceptable level, and we can carry on the experiments with meshes generated in this fashion.

4.3 Rotational tests

In this section we will consider the test of the rotation of cone concentration distribution (Zlatev [58], Crowley [16], Molenkampf [37]) for the GFEM method in Chapter 3 for different meshes.

4.3.1 On triangular mesh

The error in computations with locally refined grid is determined by the error on the coarsest part of it. We choose to compare the rotational test results over the following grids:

1. regular triangular grid 96×96 with grid step size $h = 50km$;

2. grid with local $1:2$ refinement, the coarsest grid in which the refinement is placed is like in item 1.

In both tests the cone peak with value 100 is placed at the point $(23h, 47h)$, the cone base radius is $12h$, and the cone base is fully contained in the square that confines the most refined region of the locally refined mesh (see Figure 4.13).

4.3.1.1 Results for regular mesh

The initial concentration distribution and the concentration distributions after 1, 10, 100 rotations with pure advection (without diffusion) are shown on Figure 4.11. Each rotation is made in 1600 steps. The corresponding contour plots are shown on Figure 4.12.

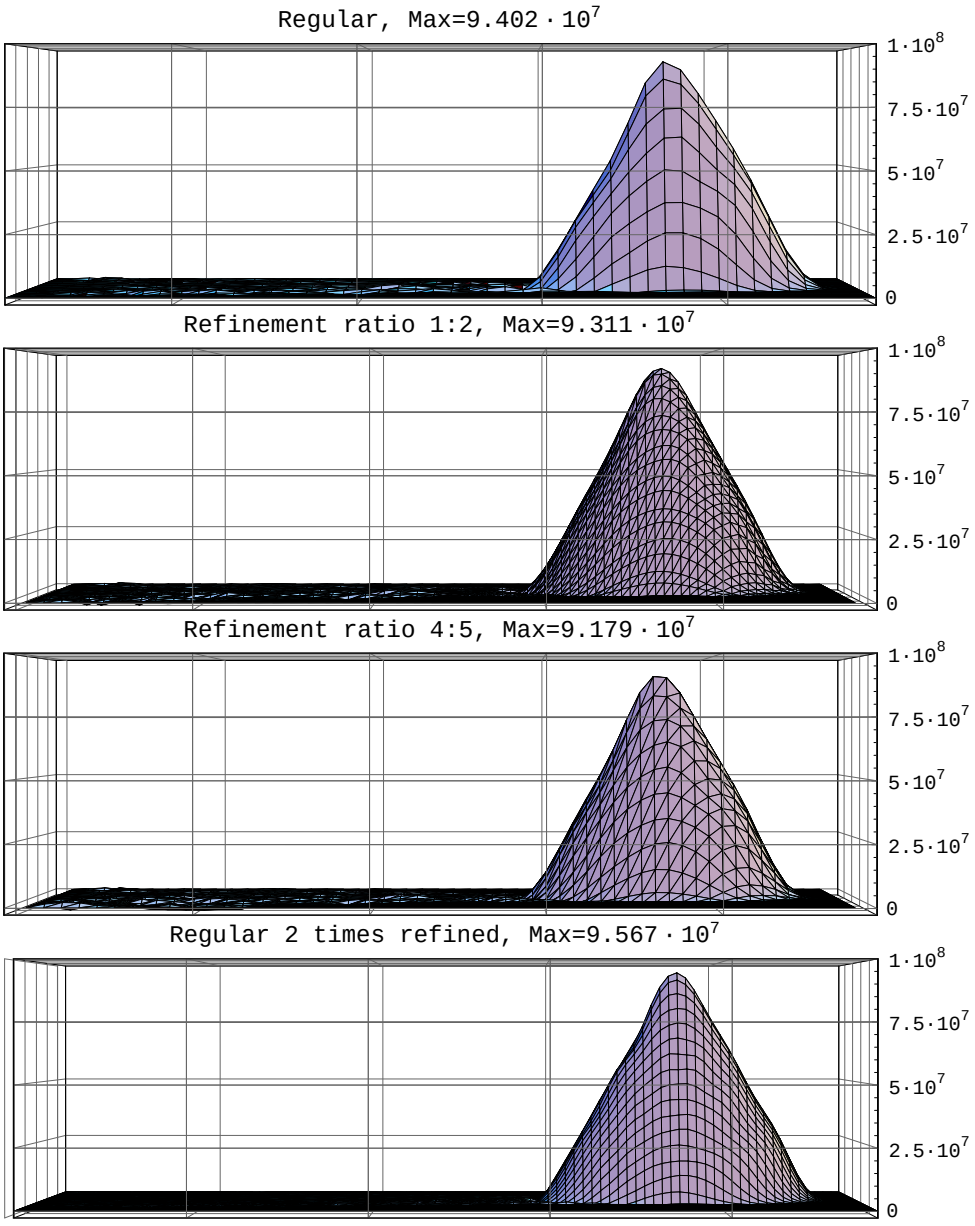


Figure 4.10: Results of the reflection test. The non-regular meshes a refined on their right half; see 4.9.

Steps	Direction	Position	Cone Peak $\times 10^7$
160	$1 \rightarrow 2$	_ _ _/_	9.3
320	$1 \leftarrow 2$	_/_ _ _	9.83
480	$1 \rightarrow 2$	_ _ _/_	9.3
640	$1 \leftarrow 2$	_/_ _ _	9.8
800	$1 \rightarrow 2$	_ _ _/_	9.3
960	$1 \leftarrow 2$	_/_ _ _	9.78
1120	$1 \rightarrow 2$	_ _ _/_	9.3
1280	$1 \leftarrow 2$	_/_ _ _	9.76
1440	$1 \rightarrow 2$	_ _ _/_	9.29
1600	$1 \leftarrow 2$	_/_ _ _	9.75
1760	$1 \rightarrow 2$	_ _ _/_	9.29
1920	$1 \leftarrow 2$	_/_ _ _	9.74
2080	$1 \rightarrow 2$	_ _ _/_	9.29
2240	$1 \leftarrow 2$	_/_ _ _	9.73
2400	$1 \rightarrow 2$	_ _ _/_	9.29
2560	$1 \leftarrow 2$	_/_ _ _	9.72
2720	$1 \rightarrow 2$	_ _ _/_	9.29
2880	$1 \leftarrow 2$	_/_ _ _	9.72
3040	$1 \rightarrow 2$	_ _ _/_	9.28
3200	$1 \leftarrow 2$	_/_ _ _	9.71

Table 4.1: Cone peak in the refined and coarse parts of the grid respectively; the grid is with $1 : 2$ refinement

Steps	Direction	Position	Cone Peak $\times 10^7$
80	$1 \rightarrow 2$	_ _/_ _	9.22
240	$1 \leftarrow 2$	_ _/_ _	9.39
400	$1 \rightarrow 2$	_ _/_ _	9.24
560	$1 \leftarrow 2$	_ _/_ _	9.39
720	$1 \rightarrow 2$	_ _/_ _	9.24
880	$1 \leftarrow 2$	_ _/_ _	9.39
1040	$1 \rightarrow 2$	_ _/_ _	9.23
1200	$1 \leftarrow 2$	_ _/_ _	9.39
1360	$1 \rightarrow 2$	_ _/_ _	9.23
1520	$1 \leftarrow 2$	_ _/_ _	9.39
1680	$1 \rightarrow 2$	_ _/_ _	9.22
1840	$1 \leftarrow 2$	_ _/_ _	9.39
2000	$1 \rightarrow 2$	_ _/_ _	9.22
2160	$1 \leftarrow 2$	_ _/_ _	9.39
2320	$1 \rightarrow 2$	_ _/_ _	9.21
2480	$1 \leftarrow 2$	_ _/_ _	9.38
2640	$1 \rightarrow 2$	_ _/_ _	9.21
2800	$1 \leftarrow 2$	_ _/_ _	9.38
2960	$1 \rightarrow 2$	_ _/_ _	9.2
3120	$1 \leftarrow 2$	_ _/_ _	9.38

Table 4.2: Cone peak when it passes the inner boundary of the grid with $1 : 2$ refinement.

Steps	Direction	Position	Cone Peak $\times 10^7$
160	$4 \rightarrow 5$	___/_	9.25
320	$4 \leftarrow 5$	_/___	9.82
480	$4 \rightarrow 5$	___/_	9.25
640	$4 \leftarrow 5$	_/___	9.79
800	$4 \rightarrow 5$	___/_	9.25
960	$4 \leftarrow 5$	_/___	9.76
1120	$4 \rightarrow 5$	___/_	9.25
1280	$4 \leftarrow 5$	_/___	9.75
1440	$4 \rightarrow 5$	___/_	9.24
1600	$4 \leftarrow 5$	_/___	9.74
1760	$4 \rightarrow 5$	___/_	9.24
1920	$4 \leftarrow 5$	_/___	9.73
2080	$4 \rightarrow 5$	___/_	9.24
2240	$4 \leftarrow 5$	_/___	9.72
2400	$4 \rightarrow 5$	___/_	9.24
2560	$4 \leftarrow 5$	_/___	9.71
2720	$4 \rightarrow 5$	___/_	9.24
2880	$4 \leftarrow 5$	_/___	9.7
3040	$4 \rightarrow 5$	___/_	9.24
3200	$4 \leftarrow 5$	_/___	9.69

Table 4.3: Cone peak in the refined and coarse parts of the grid respectively; the grid is with $4 : 5$ refinement.

Steps	Direction	Position	Cone Peak $\times 10^7$
80	$4 \rightarrow 5$	__/__	9.24
240	$4 \leftarrow 5$	__/__	9.38
400	$4 \rightarrow 5$	__/__	9.22
560	$4 \leftarrow 5$	__/__	9.38
720	$4 \rightarrow 5$	__/__	9.22
880	$4 \leftarrow 5$	__/__	9.38
1040	$4 \rightarrow 5$	__/__	9.22
1200	$4 \leftarrow 5$	__/__	9.38
1360	$4 \rightarrow 5$	__/__	9.22
1520	$4 \leftarrow 5$	__/__	9.38
1680	$4 \rightarrow 5$	__/__	9.22
1840	$4 \leftarrow 5$	__/__	9.38
2000	$4 \rightarrow 5$	__/__	9.21
2160	$4 \leftarrow 5$	__/__	9.37
2320	$4 \rightarrow 5$	__/__	9.21
2480	$4 \leftarrow 5$	__/__	9.37
2640	$4 \rightarrow 5$	__/__	9.21
2800	$4 \leftarrow 5$	__/__	9.37
2960	$4 \rightarrow 5$	__/__	9.21
3120	$4 \leftarrow 5$	__/__	9.37

Table 4.4: Cone peak when it passes the inner boundary of the grid with $4 : 5$ refinement.

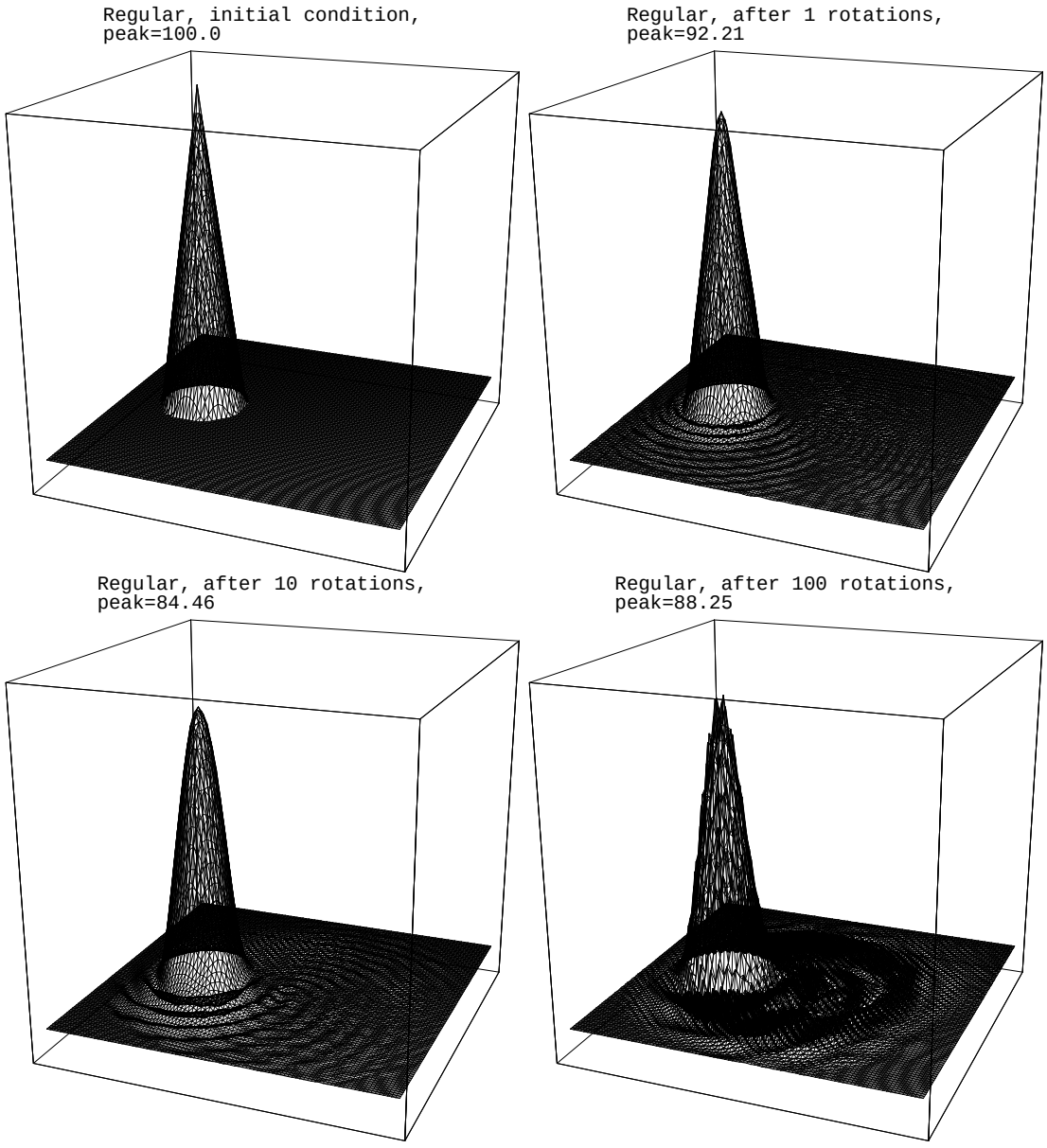


Figure 4.11: Rotational test on regular triangular mesh. Concentration distribution initially and after 1, 10, 100 rotations. One rotation is made with 1600 steps.

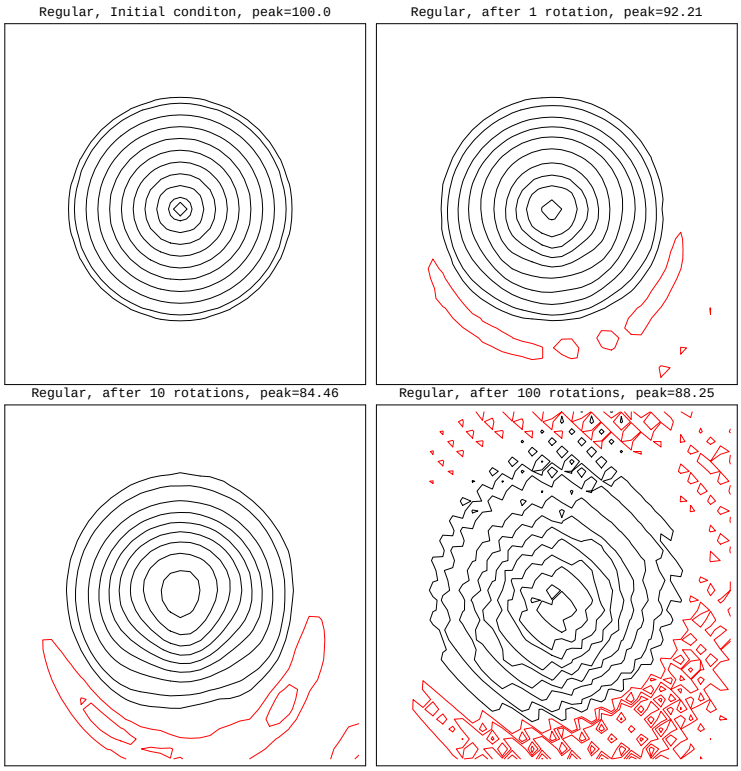


Figure 4.12: Rotational test on regular triangular mesh. Contour plots of the concentration distribution initially and after 1, 10, 100 rotations. The red contours are at $-5, -3, -1$ the black contours are at $5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 95$.

4.3.1.2 Results for locally refined mesh

On Figure 4.13 are shown the initial concentration distribution, and the concentration distributions after 1, 10, 100 rotations with pure advection (without diffusion). Each rotation is made within 1600 steps. On Figure 4.14 are shown the contour plots of the distributions.

4.3.1.3 Comments

We can explain the persistent track of wiggles in the 100 rotations picture with the zero group velocity of the waves with length $3h$; see Figure 3.9. After each rotation there are left over, non-moving waves, that after some, $50 - 100$, rotations heap over each other.

We did other rotational tests on refined grids: with several times smaller number of steps for one rotation, with sharper cone, with deeper refinement. The cone rotation test is quite heavy: the cone has discontinuous derivatives at its top and its base. After some number of rotations we start to have wiggling solutions, though they can be stabilized if after each advection step some filtering technique is used. Just for our rotational tests we used the filter described in [35]. (The experiments with real data shown bellow are done without filtering, though the advection is combined with diffusion which has similar to the filtering effect.) In the specified above experiment with $1 : 2$ local refinement (pure advection, no filtering), such instabilities grow quite fast after 80 rotations. For the regular grid the parasite waves are quite tangible after 100 rotations. In both cases the cone looks quite well – no wiggles, peak > 85 – for the first 20 rotations. (In the literature usually no more than $2 - 5$ rotations are discussed.) Comparing the regular and refined grid rotational test results we can see that the later compare quite well with the former.

Analysis of the spurious reflection induced by the inner boundaries in the one-dimensional case is presented in [54]. A concise description of the stability theory, developed by Gustafsson, Kreiss, and Sundstrom for finite difference models for hyperbolic equations, is given in [48].

4.3.2 On square mesh

The rotational test with a regular square grid failed because of unbounded oscillations at the boundaries. We can try a variant of this test in which the grid is as it is shown on Figure 4.15.There are no oscillations with this grid, see Figures 4.16, 4.17. In order to convince ourself that the simulations with this grid behave normally, we can make a test in which the cone is translated in a diagonal direction through the grid and out of it. The results of this test can be seen on Figure 4.18.

4.3.3 In three dimensions

We can make three dimensional advection simulations splitting the equation:

$$\frac{\partial c}{\partial t} = -\frac{\partial (uc)}{\partial x} - \frac{\partial (vc)}{\partial y} - \frac{\partial (wc)}{\partial z}$$

into

$$\frac{\partial c}{\partial t} = -\frac{\partial (uc)}{\partial x} - \frac{\partial (vc)}{\partial y} \tag{4.1}$$

$$\frac{\partial c}{\partial t} = -\frac{\partial (wc)}{\partial z}. \tag{4.2}$$

We can use a 2D advection simulator for (4.1), and 1D one for (4.2). On Figure 4.19 is shown a simulation of the equation

$$\begin{aligned} \frac{\partial c}{\partial t} = -(z - \frac{Z}{2})\frac{\partial c}{\partial x} - 0\frac{\partial c}{\partial y} - (\frac{X}{2} - x)\frac{\partial c}{\partial z} \\ +K(\frac{\partial^2 c}{\partial x^2} + \frac{\partial^2 c}{\partial y^2} + \frac{\partial^2 c}{\partial z^2}), \\ K = 30000 m^2/s, \end{aligned}$$

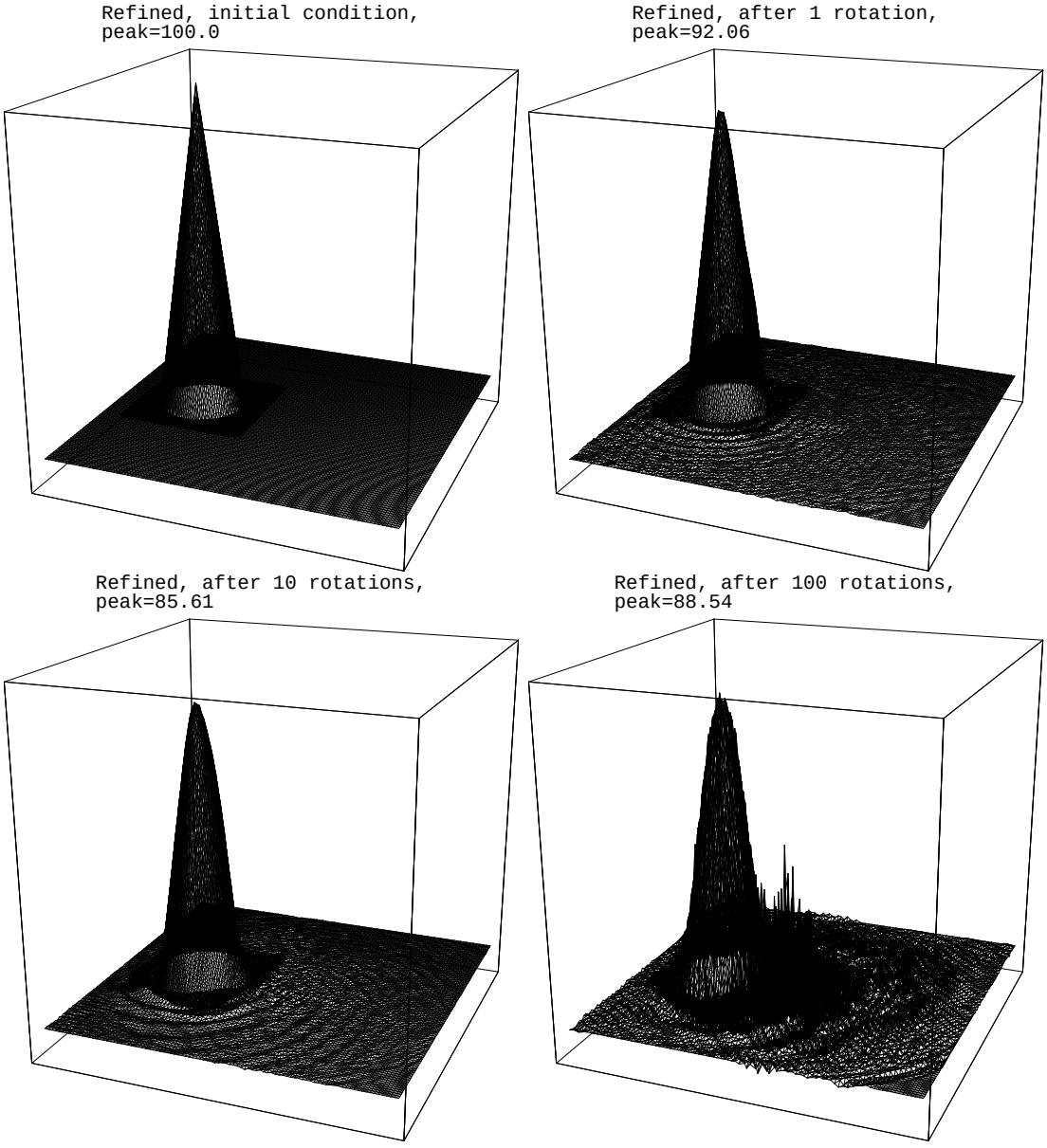


Figure 4.13: Rotational test on refined triangular mesh. Concentration distribution initially and after 1, 10, 100 rotations.One rotation is made with 1600 steps.

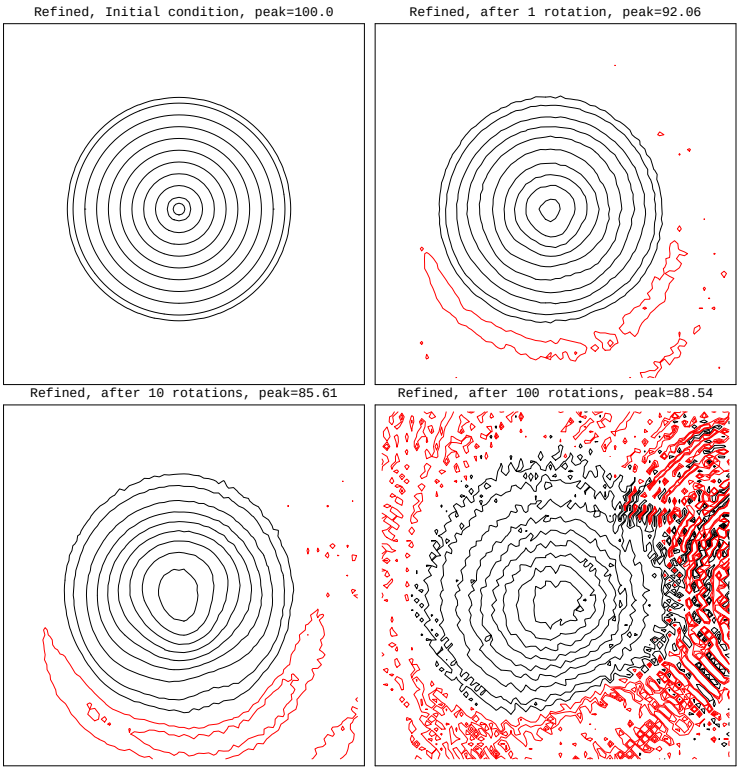


Figure 4.14: Rotational test on regular triangular mesh. Contour plots of the concentration distribution initially and after 1, 10, 100 rotations. The red contours are at $-5, -3, -1$ the black contours are at $5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 95$.

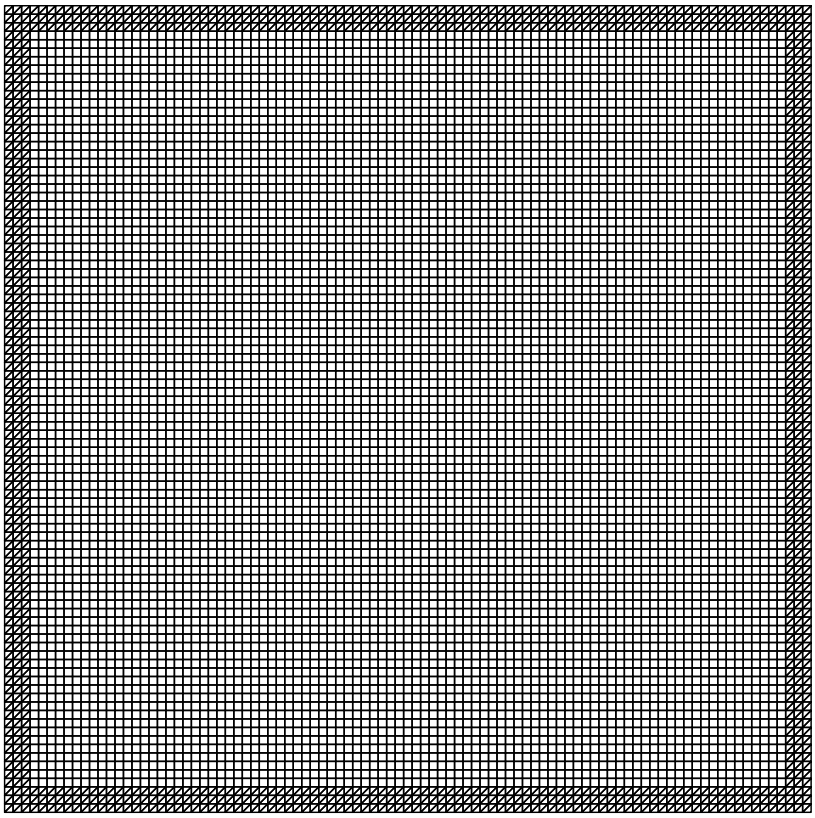


Figure 4.15: Square element grid padded with triangles.

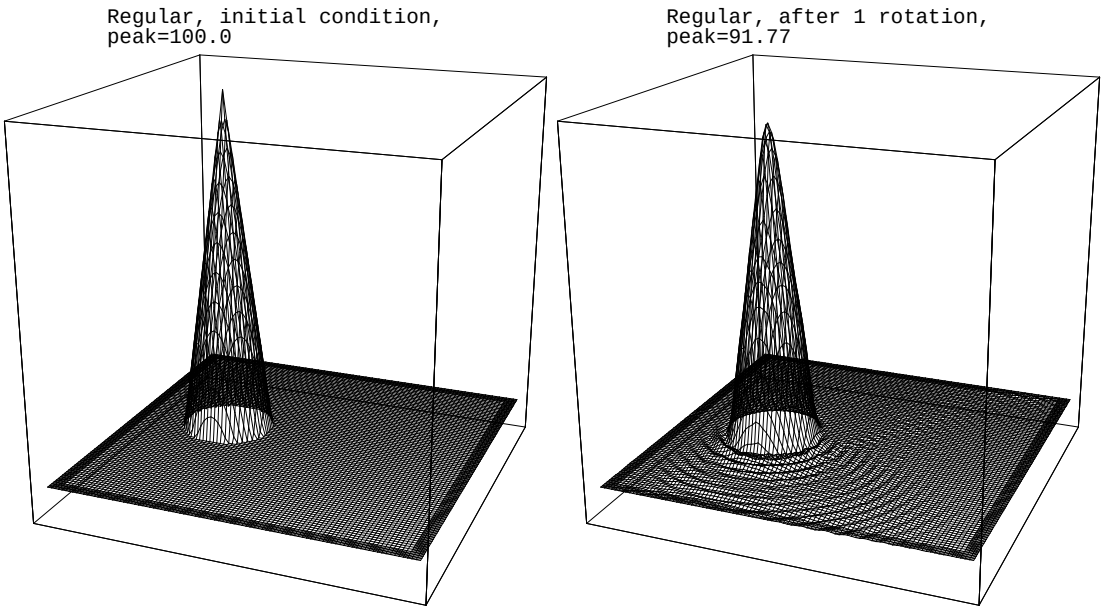


Figure 4.16: Rotational test on square element grid padded with triangles (Figure 4.15). Concentration distribution initially (on the left) and after 1 rotation made with 1600 steps (on the right).

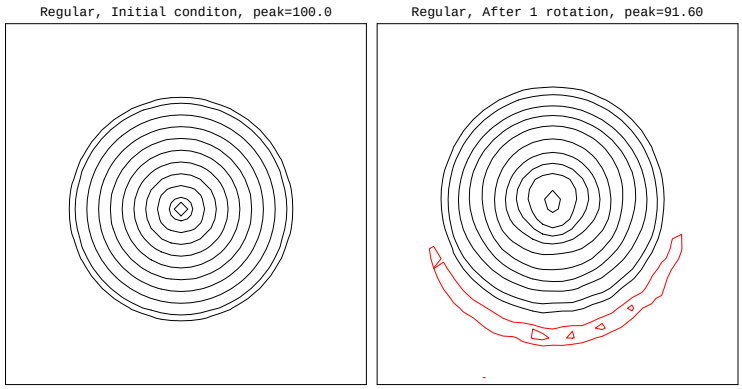


Figure 4.17: Rotational test on square element grid padded with triangles (Figure 4.15). Contour plots of the concentration distribution initially (on the left) and after 1 rotation (on the right). The red contours are at $-2, -1$; the black contours are at $5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 95$.

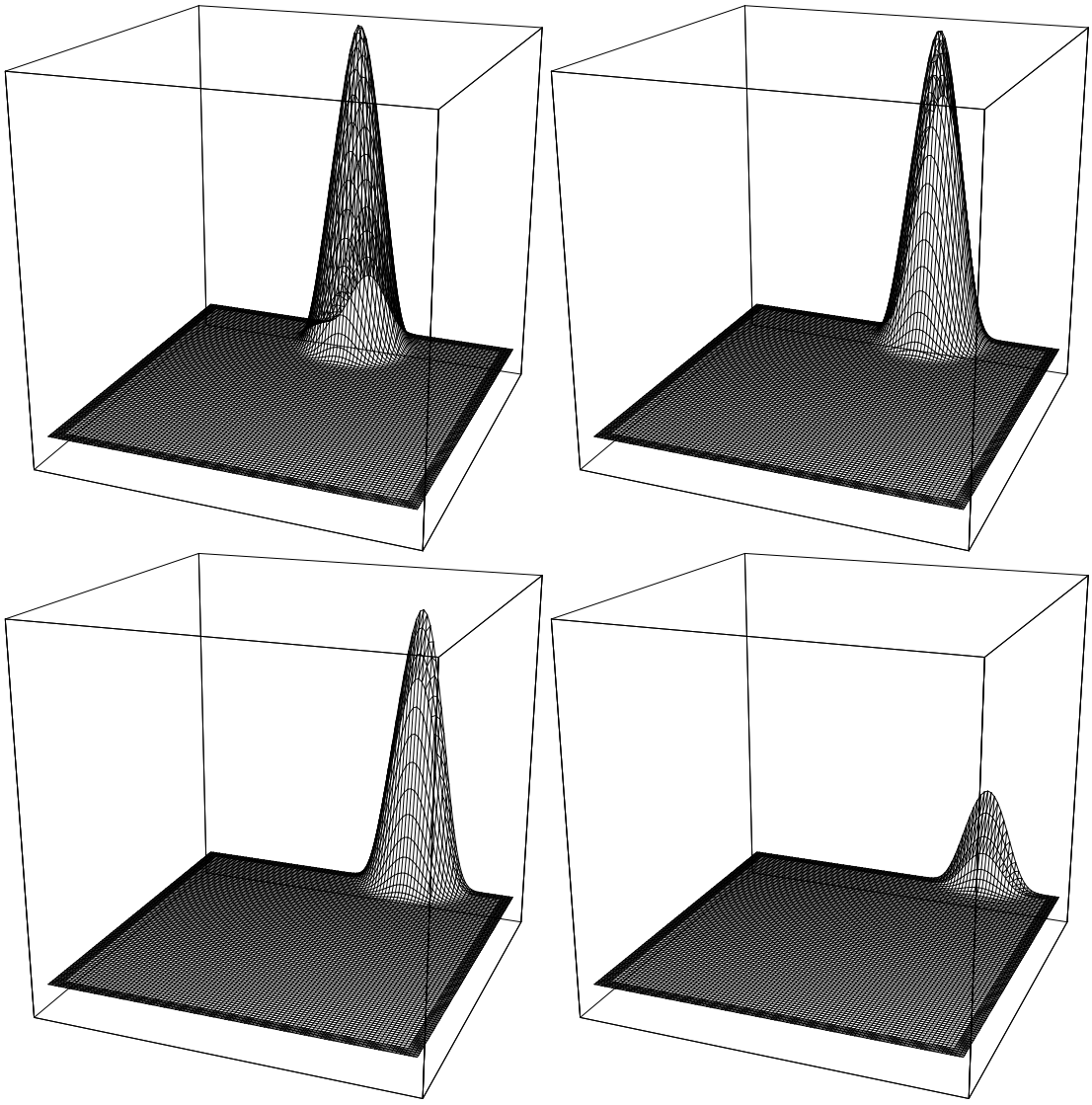


Figure 4.18: Translation out of the square element grid padded with triangles.

in the parallelepiped

$$D = \{(x, y, z) | \, 0 \leq x \leq X, \, 0 \leq y \leq Y, \, 0 \leq z \leq Z\}.$$

This test lacks physical meaning in vertical direction: it was made just to demonstrate the functionality of the GFEM sub-framework of OODEM (see Chapter 5). A more appropriate test, in which in the same parallelepiped, D, is solved the equation

$$\frac{\partial c}{\partial t} = (y - \frac{Y}{2})\frac{\partial c}{\partial x} - (x - \frac{X}{2})\frac{\partial c}{\partial y} - (x - \frac{X}{2})\frac{\partial c}{\partial z},$$

is presented in [14].

4.3.4 For non-conforming elements

The idea behind the so called non-conforming finite elements is to relax the condition of continuity. See [12]. On Figure 4.21 it is shown a grid and the "non-conforming" nodes that correspond to the basis functions in this method. The reader might refer to the *Mathematica* appendix "Patch Calculations for Non-conforming Finite Elements". The results of the rotational test are shown on Figure 4.22. They look bad, but demonstrate the abilities of the GFEM sub-framework of OODEM; (see Chapter 5).

4.4 Tests with high resolution emission data

In this section the simulation results produced with the Current Production Code of DEM (CPC-DEM) are compared with those of OODEM, with emissions from for the European Commission Environment Directorate-General project AUTO-OIL II [19].

Both models use the same implementation of the chemistry sub-model (QSSA). The space discretization of the advection sub-model of CPC-DEM is implemented, via splitting along the axes, with one dimensional finite elements; see [21]. The system of ordinary differential equations is solved with predictor-corrector method with several different correctors; see [57].

Fine resolution emissions are available at the National Environmental Research Institute for NO_x, VOC, and CO for year 1995 in five cities in Europe: London, Cologne, Milan, Lyon, and Athens. The emissions are given as rectangular arrays of $2km \times 2km$ squares. By pasting the fine resolution emissions to the coarser emissions ($50km \times 50km$) provided by EMEP [44], we create a new emissions data set. We consider this new data set as a new emission scenario – we will call it CITIES scenario. The old scenario is based on the emissions provided by EMEP – we will call it the EMEP scenario. Our goal is to identify qualitatively what are the differences of the long range air pollution transport produced by each of these two scenarios.

We are particularly interested in the distribution of the ozone excess. We compare the averaged for each month daily ozone maximums derived from the two scenarios.

With CPC-DEM the simulations are performed with a totally refined, 480×480 grid, each square of which is $10km \times 10km$. The fine resolution emissions are approximated (via aggregation) for this grid. Both the advection and the chemistry time steps are $150s$. To maintain the stability, the advection step cannot be increased. The simulation results of the CITIES scenario are divided by the simulation results of the EMEP scenario. The distribution of resulted ratios over Europe for July and August are shown on Figures 4.23 and 4.24.

With OODEM we used the grid shown on Figures 4.25 and 4.26. The grid is constructed from a 96×96 regular grid with resolution $50km \times 50km$. The finest parts of it are with resolution $10km \times 10km$. The advection time step is $900s$ (six times greater than the advection time step in CPC-DEM.) The chemistry time step is $150s$. The results obtained with OODEM are shown on Figures 4.27 and 4.28.

To see the qualitative resemblance of the results we can color them independently of each other – see Figure 4.29 and Figure 4.30. We can see that, although we have more detailed pattern in the results of CPC-DEM, they are qualitatively in good agreement with the results from OODEM. We should note that, although our scheme uses local fine resolution data with locally refined grid, the solutions are influenced

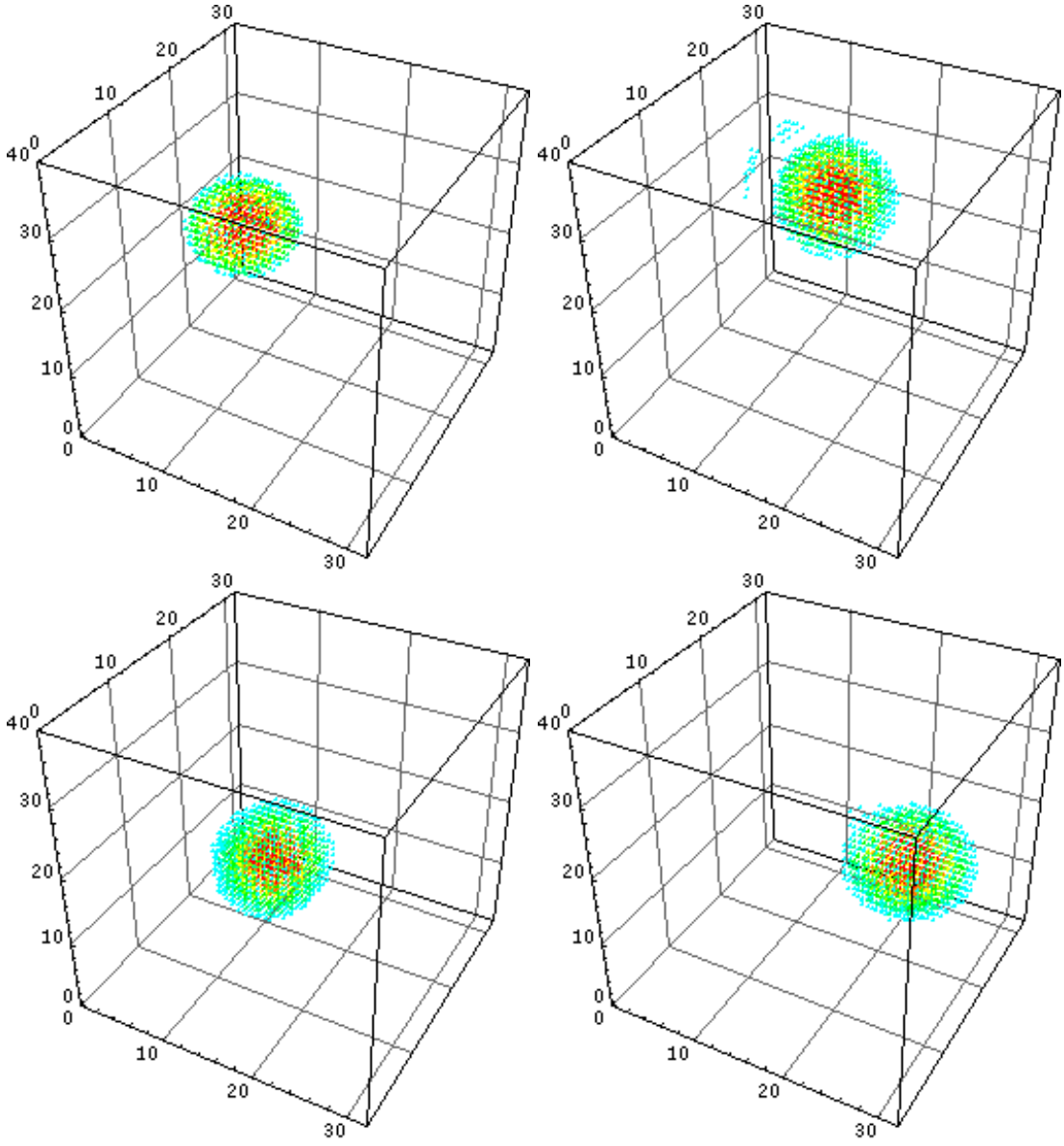


Figure 4.19: 3D advection simulation as 1D×2D . The legend for the colors is shown on Figure 4.20.

The pictures are placed in the following fashion: at 1 x Delta t ⟶ at 3 x Delta t ↓ at 9 x Delta t ⟵ at 6 x Delta t .

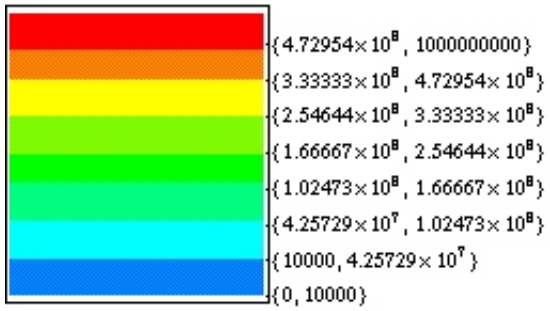


Figure 4.20: Color legend for the 3D rotation test results on Figure 4.19.

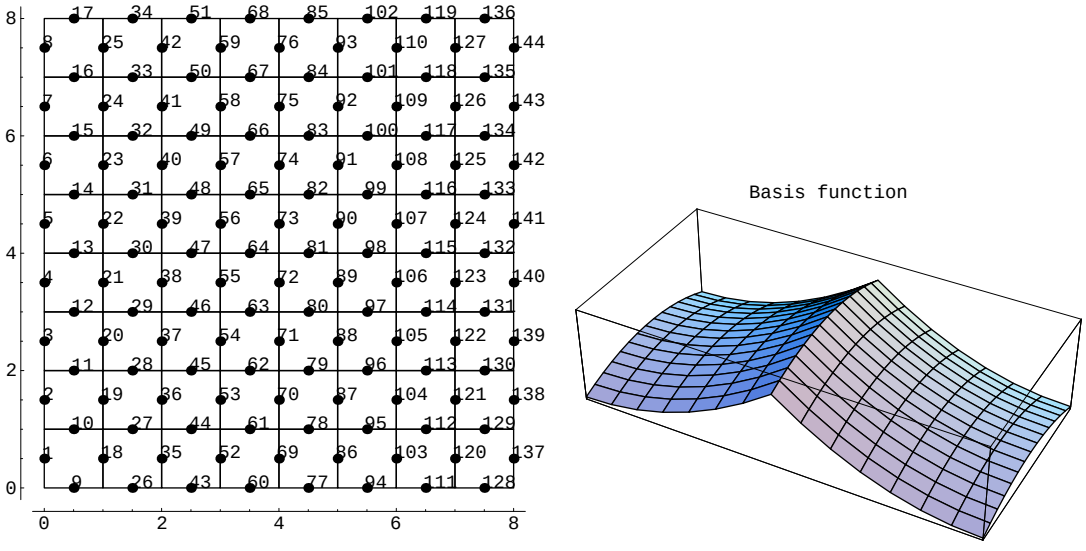


Figure 4.21: Non-conforming finite elements grid and a basis function.

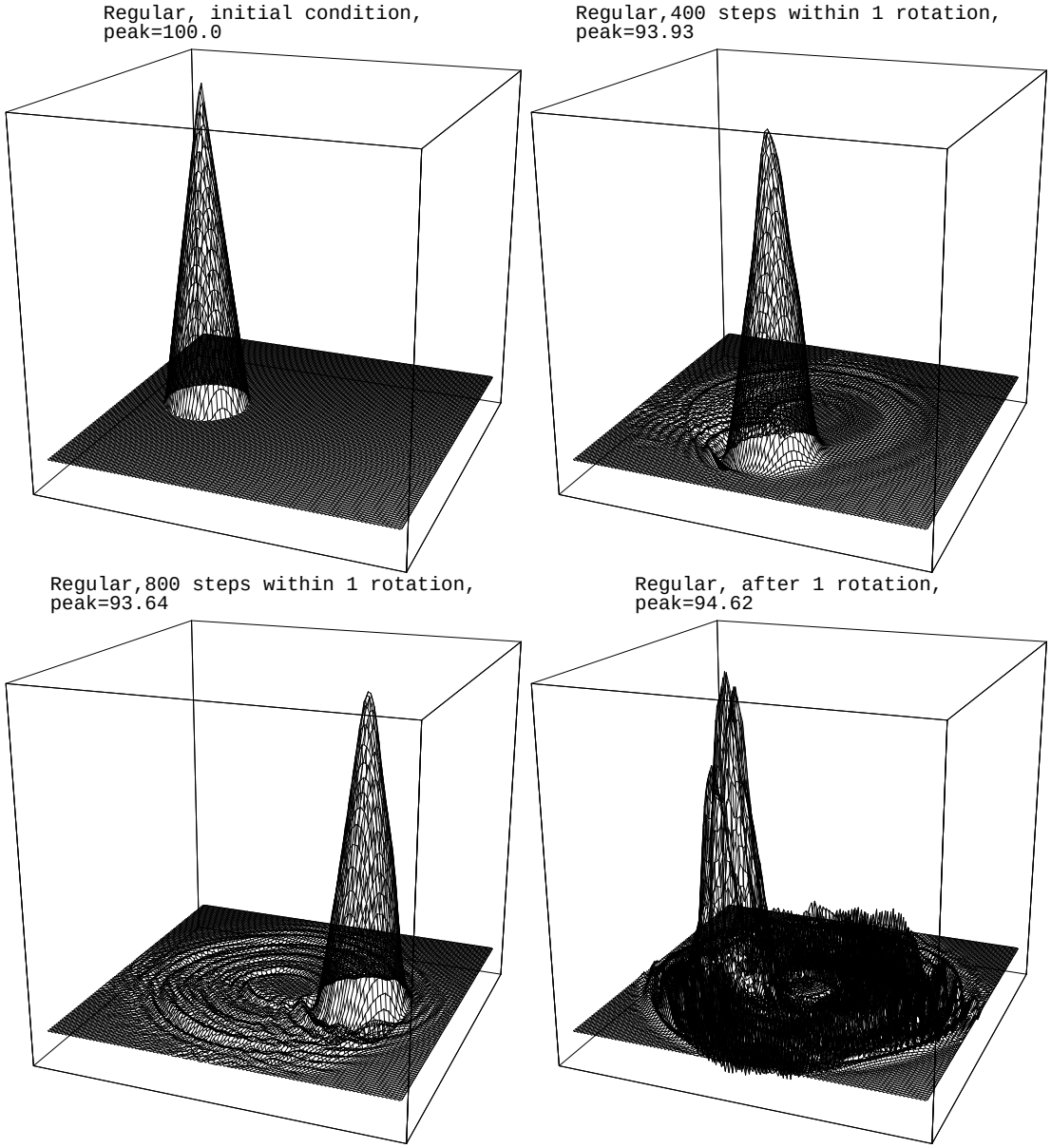


Figure 4.22: Rotational test results with non-conforming elements.

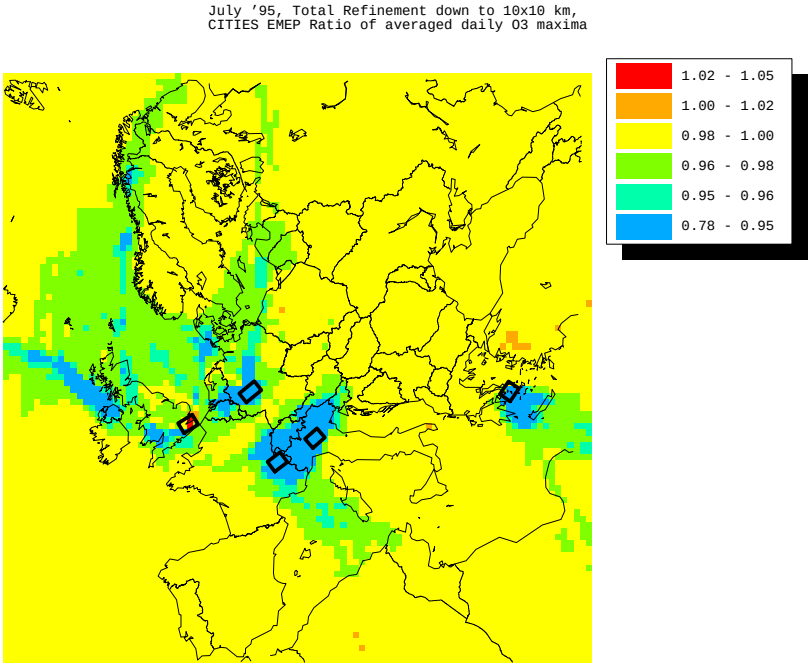


Figure 4.23: Ratios (cities averaged daily ozone maxima for July 1995 divided by EMEP averaged daily maxima for July 1995) derived with totally refined grid 480×480 with resolution $10km \times 10km$. The results are projected on a regular 96×96 grid.

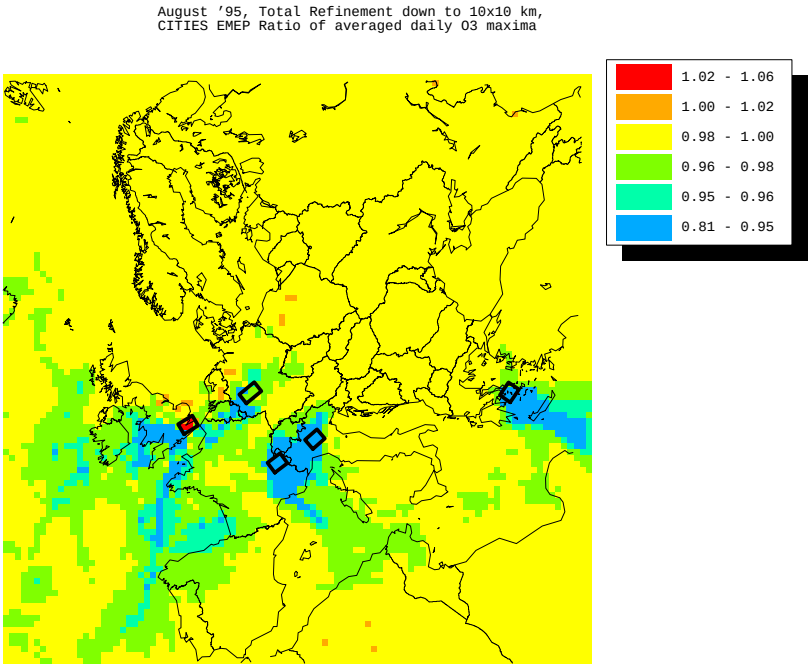


Figure 4.24: Ratios (cities averaged daily ozone maxima for August 1995 divided by EMEP averaged daily maxima for August 1995) derived with totally refined grid 480×480 with resolution $10km \times 10km$. The results are projected on a regular 96×96 grid.

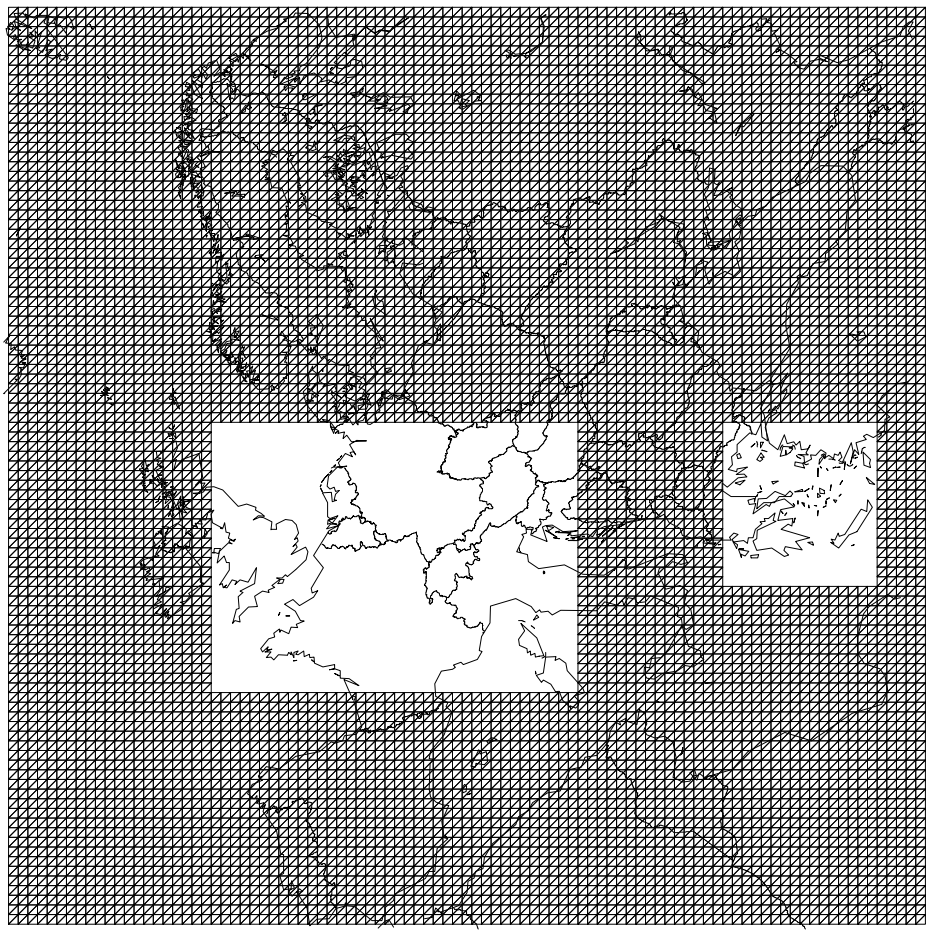


Figure 4.25: Coarse grid with two regions where local refinement is to be prepared. The grid originates from 96×96 grid. In the empty places are inserted the cells of finer grid shown in Figure 4.26.

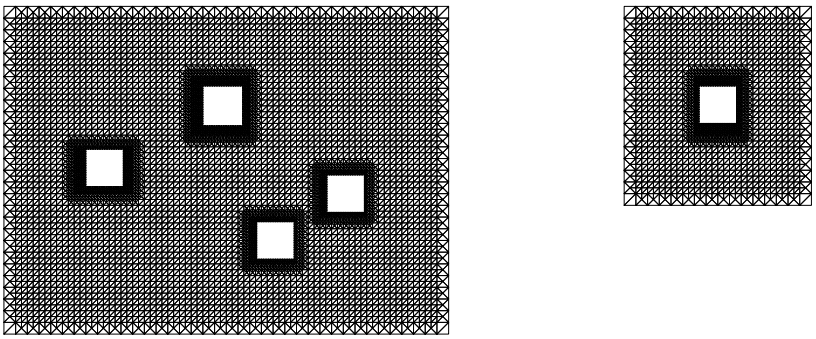


Figure 4.26: Regions in which local refinement has been applied. The places where the refined data is pasted are left empty. The big rectangle area on the left and the big square on the right are 2 times finer that the courser grid. The 5 cells inside them are with nested refinement ratios: $1:2$, $4:5$. The white squares are with resolution $10km \times 10km$.

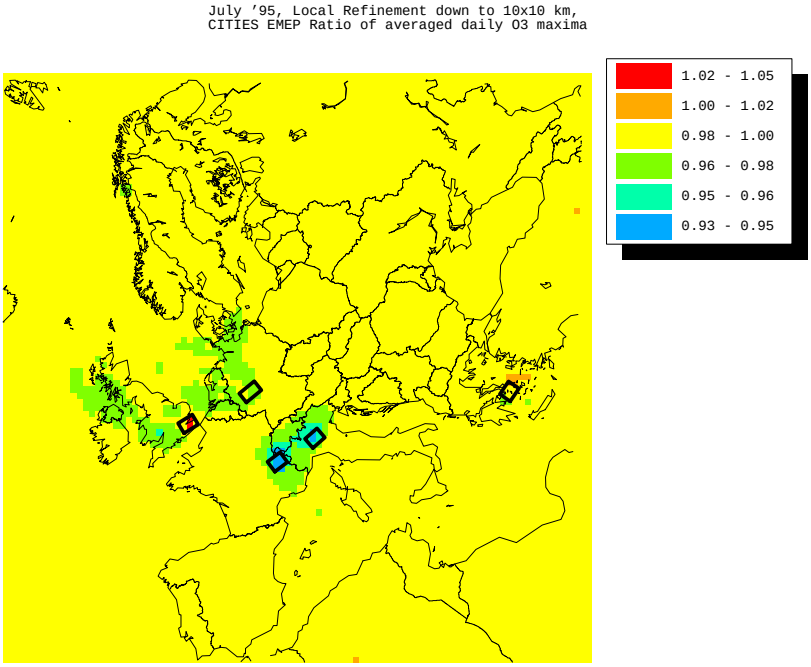


Figure 4.27: Ratios (cities averaged daily ozone maxima for July 1995 divided by EMEP averaged daily maxima for July 1995) derived with locally refined grid shown on Figures 4.25 and 4.26. The results are projected on a regular 96×96 grid.

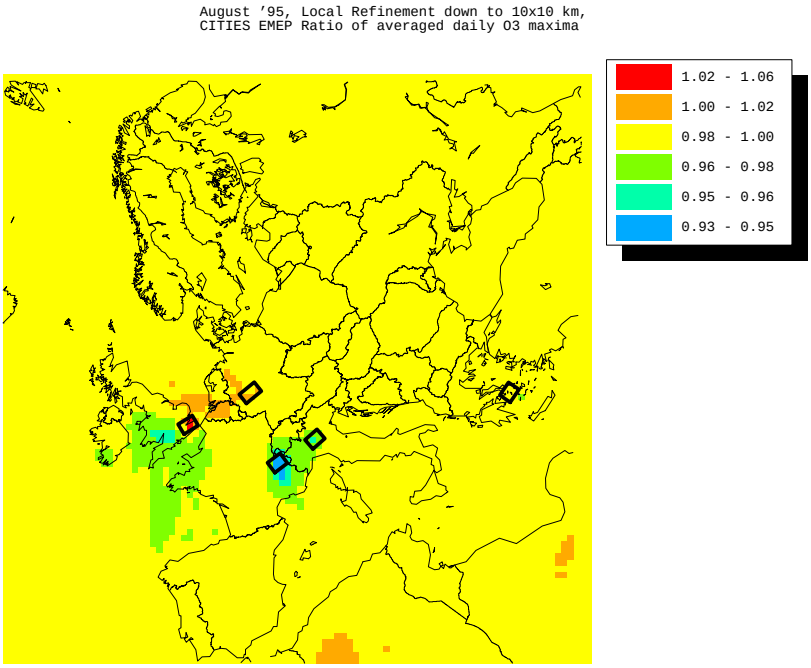


Figure 4.28: Ratios (cities averaged daily ozone maxima for August 1995 divided by EMEP averaged daily maxima for August 1995) derived with locally refined grid shown on Figures 4.25 and 4.26. The results are projected on a regular 96×96 grid.

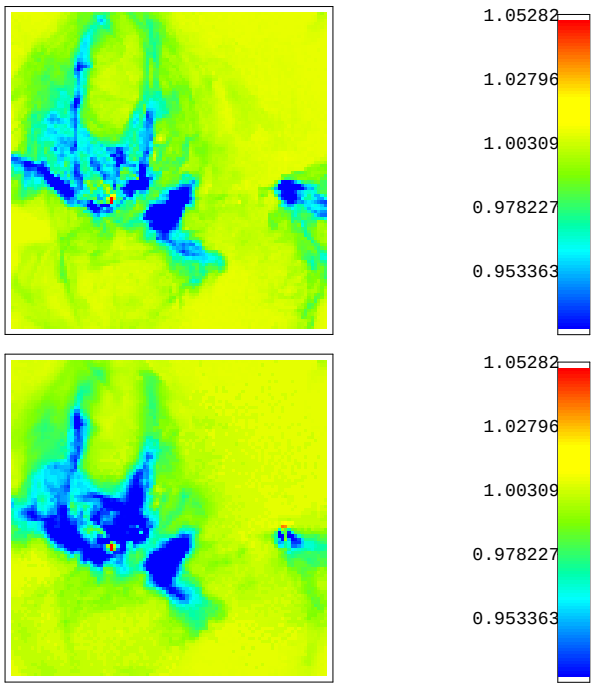


Figure 4.29: Results for the ratio CITIES/EMEP for July '95, projected on a regular 96×96 grid, colored within the ratio variation of the OODEM results. Above: ratio from CPC-DEM; bellow: ratio from OODEM.

globally: we can see the impact of the refined emissions on the whole model domain. This cannot be achieved with the one-way nesting method. It is $12 - 15$ times cheaper, in terms of computing time, to obtain the results with OODEM, than with CPC-DEM. Moreover, the storage requirements are also reduced when local refinements are used.

We made experiments with deeper nesting, down to resolution $2km \times 2km$, restricting ourself just to the London area. Here are shown just the results for July'95. The grid resembles the grid shown on Figure 2.1 on page 27. The nested refinement on that figure was repeated two times, i.e. in the regular inner part of the framed cell, another cell of the same type was nested . This gives refinement ratio $1 : 25$, needed to produce the resolution $2km \times 2km$ from $50km \times 50km$. The results for the CITIES/EMEP ratio from the two refinements for the whole model area, and for the London area respectively, are shown on Figures 4.31 and 4.32. The results are only for July '95.The number of days in which the ozone excess is above $90ppb$ for the London area, and the area around London, are shown on Figure 4.33.

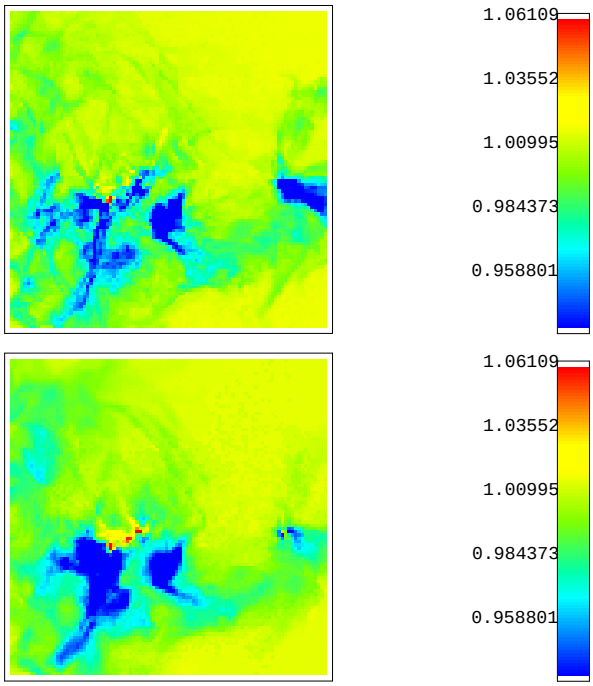


Figure 4.30: Results for the ratio CITIES/EMEP for August '95, projected on a regular 96×96 grid, colored within the ratio variation of the OODEM results. Above: ratio from CPC-DEM; bellow: ratio from OODEM.

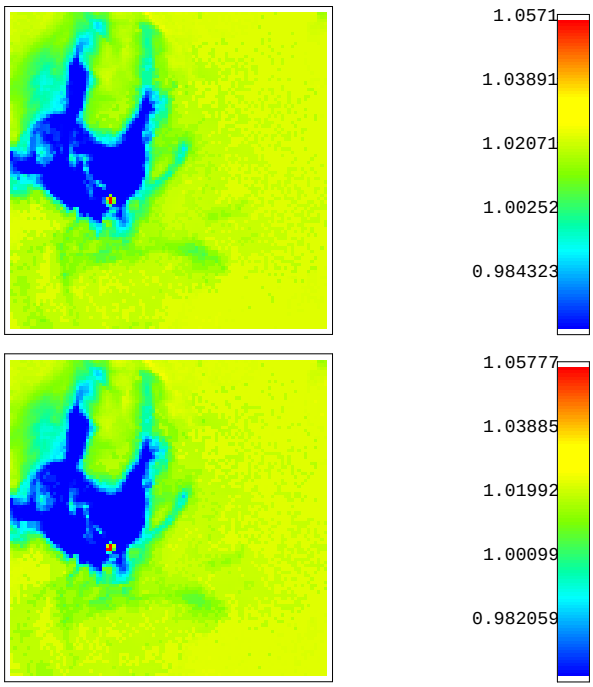


Figure 4.31: Independently colored results for July'95 projected on a regular 96×96 grid. Above: from OODEM with resolution down to $10km \times 10km$; bellow: from OODEM with resolution down to $2km \times 2km$.

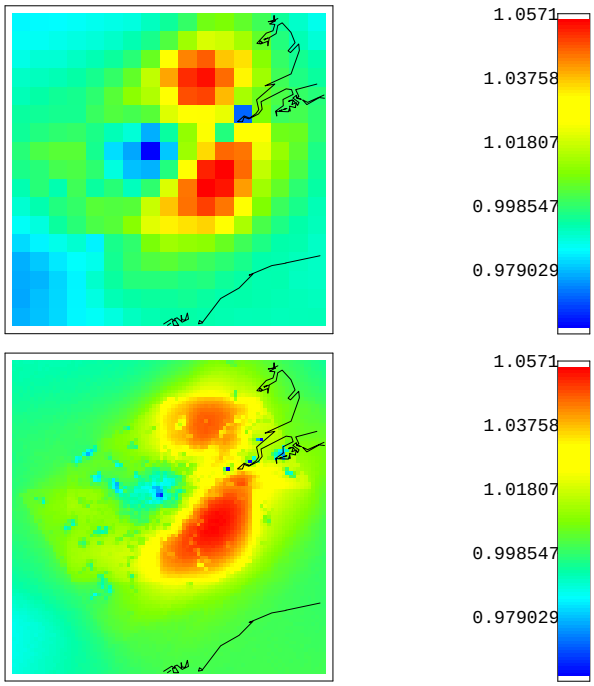


Figure 4.32: Independently colored results for July'95 refined grids on the London area. Above: from OODEM with resolution down to $10km \times 10km$; bellow: from OODEM with resolution down to $2km \times 2km$.

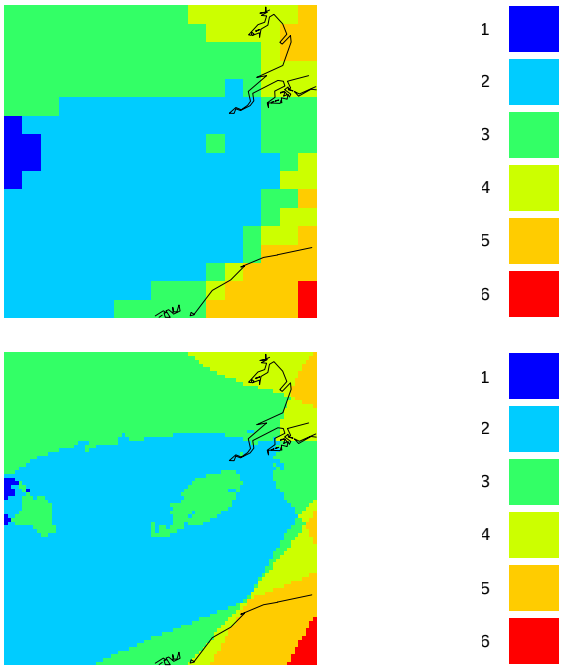


Figure 4.33: The number of days with ozone excess above $90ppb$ for the London area with CITIES scenario July '95. Above: from OODEM with resolution down to $10km \times 10km$; bellow: from OODEM with resolution down to $2km \times 2km$. The corresponding monthly averaged minimal and maximal values (of the ozone excess on London) are:

averaged	10x10	2x2
Min O_3, ppb	47.40	34.21
Max O_3, ppb	65.17	65.43

Chapter 5

Object-Oriented Framework for DEM

Original prankster
...Original yeah

– OFFSPRING, *"Original Prankster",*
Conspiracy Of One, 2000

5.1 Introduction

Simulations based on a Large Scale Air Pollution Model (LSAPM) involve heavy computations because of the large modeling domain and the number of the considered chemical components. These computations result from the rather abstract and subtle notions on which their mathematical algorithms are based on. A program that perform the simulation is, as a mater of fact, a model of these mathematical algorithms. Object-Oriented(OO) languages provide paradigm, in which the entities, the invariants, and the relations within the subject being modeled, can be reflected in the programming code. If we want to provide an environment, where area specific investigations are made, it is natural then to prepare some preliminary code that reflects the principles common for any activity in that area. That preliminary code is called framework. A framework should be very easily tuned, completed, extended to a concrete program that meets the user needs.

On the other hand, if we want to have an implementation that meets the objectives of efficiency, extensibility and understandability, we should make tradeoffs. For example, we should trade extensibility for efficiency, or understandability for both extensibility and efficiency. It is good to use *patterns* for the design problems that arise when these tradeoffs are modeled with an OO language.

The notion of pattern relies on the notion of *force.* The demands, the desires for qualities like efficiency, extensibility and understandability are examples of forces. The entropy trend induced by some unbalanced combination of them is also a force. The following broad definition of a pattern is from "The Timeless Way of Building" by C.Alexander. The word "spatial" from the original text is replaced with the word "software":

> Each pattern is a [named] three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain *software* configuration which allows these forces to resolve themselves.

The patterns commonly referred as *design patterns,* are for micro-architecture implementations in object-oriented development. There are also patterns for macro-architecture issues like those met in a framework construction. A good introduction and reference to software patterns is [6].

The individual patterns are not isolated. Each pattern depend on the smaller patterns it contains, and on the larger patterns in which it is contained. The patterns' combinations and mutual definitions, form a *pattern language.* The system of forces which a pattern allows to resolve, resolve themself across, within, upon the smaller patterns the pattern contains and the larger ones in which it is contained. Resolving

becomes resolution. The pattern language serves to discuss, to explain and to share the mental images of this resolving.

To make the Object Oriented framework for the Danish Eulerian Model (OODEM) I used the framework pattern language described in [11], completed with the design patterns language described in [20]. In the following sections it is explained how the resulting language was applied to the identified problems and design issues. The presentation is based on the approaches suggested in [13, 27].

The explanations are accompanied with Unified Modeling Language (UML) class and sequence diagrams. The UML class diagrams express the static structure of a system in terms of classes and relationships. The UML sequence diagrams illustrate interactions between objects using a temporal structure that represents the order of communication [38]. The reader might refer to Appendix A: "Object-Oriented Terms Glossary", and Appendix B: "UML Quick Reference".

In this chapter I will use both the 'I' form and the 'we' ('me and the reader') form. When is built a software system like OODEM, many design decisions are made according to the designer's taste, temperament, and knowledge of the area. Although I have tried to present them as inevitable consequences of the addressed problems and the adopted assumptions, I cannot claim that they are, as we do when we describe and discuss mathematical knowledge and research. Hence, for these decisions I will use the personal 'I' form. For the common, established approaches, decisions, etc., I will use the 'we' form.

Remark 5.1.1 *This chapter is the conceptual documentation of OODEM. OODEM's class documentation, and documented examples, are web available at* http://www.imm.dtu.dk/~uniaaa/OODEM/ *. The "Mesh Generator Reference Manual" for the OODEM's mesh generator can be also found on this site.*

5.2 Frameworks

We will adopt the following definition of a framework:

> "A framework is mainly a product for developers used as an integrated development environment that facilitates, supports, guides, confines and helps the developers in building application in a well defined domain. " [11]

The framework user will be referred as both developer or framework user to distinguish him/her from the framework constructor. To discuss and judge how good is the framework we need to enlist and name the following primal qualities (see again [11]):

- **Effectiveness:** How effective is the framework? We can measure the framework effectiveness by the speed-up factor determined comparing the development cost with the framework with the development cost without the framework;
- **Extensibility:** How rigid is the framework structure? Any architecture imposes constrains.[1] When a the effectiveness of a framework is evaluated we should look at the specific areas covered by the framework. Often effectives and extensibility conflict each other.
- **Coverage:** How wide is the range of the framework applicability? Or, how wide is the range where the framework give the expected speed up factor? A framework can be extensible but have a narrow range of coverage.
- **Simplicity and Understandability:** How fast the developer learns to use the framework? The framework concepts can be too abstract or its architecture can be too difficult to understand;
- **Application Qualities – Efficiency, Portability, Parallelization:** These questions are more technical. We will use design patterns to facilitate them.

[1] And vice versa.

5.3 Building of the OODEM framework

5.3.1 The addressed problem

The development task is the implementation of an object-oriented framework for DEM. The framework should be amenable for simulations with local refinements, 3D simulations, and inclusion of new chemical schemes. The simulations should be run on parallel computers. It should be easy to change the numerical methods for the treatment of both the advection-diffusion and the chemistry processes.

The way this task was approached is to adopt the splitting procedure (2.10)-(2.14) and to build first a framework for the Advection-Diffusion Submodel (ADS). It was assumed that the ADS framework should be flexible on what parallel execution model is used. The Chemistry Submodel (CS) is easier to treat in parallel, so its conceptually sequential framework should inherit/obey the data structures and design features imposed by the ADS framework.

It is important to identify the areas of flexibility in the framework – we will call them *hot areas*. It is also important to identify the areas with stable requirements, where no flexibility is desired – we will call them *cold areas*. The identification of the cold areas is important since they determine the framework structure, and hence its speed up factor.

I identified the hot and cold areas that follow.

5.3.1.1 Hot areas

- Parallel solvers for the Advection-Diffusion Submodel
- Solvers for the Chemistry Submodel
- Data handlers

5.3.1.2 Cold areas

Here are enlisted the directions on which the framework is not supposed to provide flexibility. With three diamonds ($^{\diamond\diamond\diamond}$) are superscribed the characteristics which are immovable, irremovable, and totally frozen. Two diamonds ($^{\diamond\diamond}$) are for characteristics that can be changed with big effort – I will need at least two-three months to change one of them. With one diamond ($^{\diamond}$) are marked the characteristics that can be easily changed – I will need no more than one-two weeks to change one of them.

- Model splitting (2.10)-(2.14) $^{\diamond\diamond\diamond}$
- The ADS is treated with Galerkin$^{\diamond\diamond\diamond}$ Finite Element$^{\diamond\diamond}$ methods with piecewise (bi-)linear$^{\diamond}$ basis functions with static$^{\diamond\diamond}$ grids
- The grids are locally uniform$^{\diamond}$. The uniformly refined regions are rectangular$^{\diamond}$ and have sides parallel $^{\diamond}$ to the axes
- 3D simulations are carried as a number of horizontal layers placed vertically over each other $^{\diamond\frac{\diamond}{2}}$

5.3.2 The patterns used

I considered first the "well defined domain" of the ADS treated with finite elements. One natural approach[2] to start the building of a framework is the following:

1. First we identify the abstract concepts that should be reflected in the framework. The knowledge of these concepts is not trivial and thus the developer is not assumed to have them. To identify a concept means to identify the notions it is based on, and their interactions. In the ADS framework, these concepts consist of the Galerkin Finite Element Methods(GFEM).

[2]"Natural" because I discovered it myself, and then I found it in [11].

2. Since the concepts are "high" abstractions, we might feel the need to provide a package that implements the context where these concepts can have shape within. The context is not the concepts themself. It is implementations of concepts from a lower level of abstraction, and their interactions (the operations between them). For example, the systems of linear equations are conceptually lower than the GFEM, and a linear solver package is required for GFEM implementations.

That approach leads to the pattern **Products and Building Blocks (Conceptual layering)** presented in [11]: the framework elements (classes in most cases) are divided into two layers, products and building blocks. The developer assembles building blocks to construct product level concepts.

As it is indicated in [11], applying the pattern **Conceptual layering** has the following consequences (+ is for positive feature, - is for drawback):

\+ **Simplicity**

\+ **Framework integration** – Frameworks with common building blocks can be integrated more easily.

\+ **Extensibility**

\+ **Framework constructor guidance** – The framework constructor itself becomes more aware of the developer needs e.g. "getting results with minimum knowledge".

\- **Not so effective** to reuse the user-defined products as building blocks.

\- **Framework coverage is not very good** – The framework specializes in product concepts construction. It is not very easy to define or customize a building block.

The last drawback can be overcame if we provide a framework for the building blocks. If we do that we will have **Multi-Level Framework**, the application of which has the following consequences:

\+ **Effectiveness** – The overall framework become more effective because each framework can specialize in its own field

+/- **Simplicity** – Each framework is simpler, but the their combination might be more complicated

\+ **Framework integration** – The building blocks framework may be used to support other frameworks

\- **Framework construction costs are not paid back**

5.3.3 The solution

A **Multi-Level Framework** with three sub-frameworks. The highest layer is the DEM Layer (DEML) that creates objects from the lower layer frameworks for ADS and CS, and uses splitting to carry on the simulations. The ADS and CS frameworks are described bellow. The general OODEM framework, derived after the design and implementation of these two, is described in Section 5.11.

5.4 Framework design of the advection-diffusion submodel

In this section we will approach the ADS framework building in the way how an ignorant about the advection numerics and physics programmer does. Here I have in mind a real programmer, i.e. willing and creative. His goal is to write a program that meets the requirements: hence he inquires the minimum knowledge sufficient to write it and no more.[3]

So let us begin with the task specification, which unfolds with two more sub-task specifications in Section 5.4.2 and Section 5.4.4. These task specifications give the shape of the proposed framework layering in Section 5.4.5 on page 93.

[3]And he does not mind to be superficial about the things the program is about.

5.4.1 Problem specification: Advection-Diffusion Module

5.4.1.1 Intend

To simulate an advection-diffusion process for a given time interval with a given time step.

5.4.1.2 Explanation of the context

We seek a discrete approximation of the solution of the equation

$$\frac{\partial c}{\partial t} = -\frac{\partial (u\, c)}{\partial x} - \frac{\partial (v\, c)}{\partial y} + K_x \frac{\partial^2 c}{\partial x^2} + K_y \frac{\partial^2 c}{\partial y^2}. \tag{5.1}$$

in the time interval $[t_0, t_1]$, over the 2D region Ω. The time interval is divided to time steps, each of length Δt. For each time step should be produced a two dimensional array with the values of the function $c(x, y, t)$ at given points from the region the simulation is carried on. We will refer to these points as *nodes*. Usually the nodes are the vertexes of a mesh that covers the region Ω for the numerical discretization of (5.1). If we have an index set I assigned to the nodes, we will refer to a particular node by its index. The planar point associated with the node i , $i \in I$, will be denoted as Q_i ,

When building the numerical method, it is more convenient first to consider the equation

$$\frac{\partial c}{\partial t} = -\frac{\partial (u\, c)}{\partial x} - \frac{\partial (v\, c)}{\partial y} \tag{5.2}$$

instead of (5.1). Since the problem we consider is advection dominated, if the implementation behaves well with (5.2), it will behave also well with (5.1). The inclusion of the terms

$$K_x \frac{\partial^2 c}{\partial x^2} + K_y \frac{\partial^2 c}{\partial y^2} \tag{5.3}$$

to an implementation for (5.2) is simple, since K_x and K_y are constants, and $u = u(x, y, t)$ and $v = v(x, y, t)$ depend both upon space and time; (i.e.(5.2) is complex enough to represent (5.1)).

5.4.1.3 Objectives

Bellow is a list of the objectives the module should meet.

- The module should be fast.
- The module should run on parallel computers.
- The module should be flexible on:
  - the discretization mesh. It is especially desirable, the to be able to deal with meshes refined on some particular region;
  - how the time stepping is done;
  - the libraries used;
  - the data feeding.
- The module should be extendible to 3D advection-diffusion simulation module made either with 1D×2D advection splitting (e.g.10×96×96), or with genuine 3D numerical approximation.

5.4.1.4 Approach

We should choose an environment where a concrete advection module will be produced.[4] Then C++ is a natural choice for a programming language. We should try to build a framework that meets the objectives. Building a framework requires an expert who can anticipate the code evolution according to the numerical methods and the OO-techniques involved in it. Since I was not such an expert, I chose a representative numerical method for a backbone implementation, which would be completed to a more flexible system. Being representative, the method would require the programming of the most of the features required by the other methods included in the framework.

The representative numerical method is 2D Galerkin Finite Element Method (GFEM) with piecewise linear basis functions on a static grid (see Chapter 3). This means that when solving (5.2), at each time step we reduce the solution of the equation (5.2) to the solution of the equation

$$Pg' = Ag, \tag{5.4}$$

where A is a large, sparse, banded, different on each time step matrix, and P is a large, sparse, and banded constant matrix. Since the grid is static, the matrix A have the same structure on the different time steps. The vector g comprises the values of the function $c(x, y, t)$ on the nodes linearly ordered in some way. Hence the advection module should be able to calculate the time dependent A matrix. And the constant matrix P might be calculated elsewhere.

Some natural questions arise. They are listed bellow in groups.

- **Faster calculation of** A. Can we minimize the calculations for obtaining A? That is, can we deduce some subset of A from another? Or can we decompose A as a sum and/or a product of matrices, so we will need to calculate just some of them? Can we approximate A, and is it faster to approximate it? How is the matrix A constructed? What is its interpretation?
  These questions are answered in the next sections – Section 5.4.2, Row Reuse, and Section 5.4.4, Operator Approximation.

- **Mesh generation**. How is the mesh generated? With what programming system? How is the mesh described? Should we develop a mesh generator or use something from the World Wide Web?

- **Linear Algebra libraries**. What libraries will be used? Should we build our own interface for these libraries?

5.4.2 Problem specification: Row Reuse

5.4.2.1 Intend

To make the calculation of A in the (5.4) faster. Row Reuse calculates a row of A using the data and the calculations from another row of A.

5.4.2.2 Explanation

The matrix A can be interpreted "row-wise". If the number of nodes is N, to each node is assigned a number between 1 and N. A row i in the matrix A consists of the scalar products between the basis function defined by the node i and each of the other basis functions. So the element a_{ij} of A is the scalar product between the basis function of node i and the basis function of node j. One of the FEM's properties is that these scalar products are zero except for the neighbors of the node. That is why the matrix A is sparse.

When the time changes, the basis functions are the same, but the way their scalar product is taken differs. On each time step the scalar product $a(N_i, N_j)$ of two basis functions N_i and N_j is defined as

[4] Without any doubts we will use the object-oriented paradigm. If the reader is in doubt, he/she should read the entire content of the books [43] and [34] and re-iterate several times.

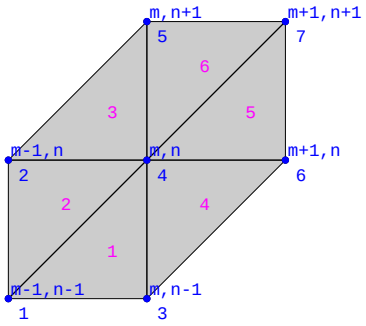


Figure 5.1: Triangular patch. Here (m,n) plays the role of j in the formula (5.5). $E_{(m,n)}$ goes from 1 to 6 over the red numbers. $I_{(m,n)}$ goes from 1 to 7 over the blue numbers.

$$a(N_i,N_j)=\begin{cases}\text{IF } i\in I_j, & \sum_{k\in E_i}\left(u_{\triangle_k}\int\int_{\triangle_k}\frac{\partial N_i(x,y)}{\partial x}N_j(x,y)dxdy\right.\\ & \left.+v_{\triangle_k}\int\int_{\triangle_k}\frac{\partial N_i(x,y)}{\partial y}N_j(x,y)dxdy\right),\\ \text{IF } i\notin I_j, & 0.\end{cases}\tag{5.5}$$

The values of $u_{\triangle_k}$ and $v_{\triangle_k}$ depend on time. The index set E_i indexes the triangles that consist the domain of N_i, I_i is the index set of the neighbors of N_i. (See Figure 5.1 for some visual aid.) A more extensive explanation can be found in Section 3.4.

It is obvious that we can make something like "row reuse" when we calculate the matrix A. More precisely, if we have calculated the integrals in (5.5) for a patch we should use just the right u and v to calculate $a(N_i,N_j)$ with (5.5) for some another equal (geometrically equivalent) patch.

5.4.2.3 Objectives

Faster than the usual computation of a row. Flexible on the algorithms that traverse the rows of A.

5.4.2.4 Approach

Dividing the rows of A into orbits: a set of nodes is called an *orbit* if for every two nodes i and j from that set there is such a translation T that for each $k\in E_i$ exists exactly one $l\in E_j$ for which $T(\triangle_k)\equiv\triangle_l$. If Q_i and Q_j are the planar points corresponding to the nodes i and j respectively, the translation T is defined by the vector Q_j-Q_i. Hence the number of the neighbors of j and i is the same, $|I_i|=|I_j|$. Obviously each orbit can be determined by any of its elements. Some orbits are shown on Figure 5.2.

5.4.3 Problem specification: Diffusion Inclusion

5.4.3.1 Intend

To change a framework for the equation (5.2) on page 89 into a framework for equation (5.1).

5.4.3.2 Explanation

As it was mentioned in Section 5.4.1.2 the flows we consider are advection dominated. The framework that handle the calculation of the integrals in equation (5.2) on page 89 and uses the Row Reuse pattern (Section 5.4.2) to calculate the operators P and A in 5.4.1.4 should be extended to handle the full advection-diffusion equation (5.1). With Diffusion Inclusion the formula (5.5) becomes

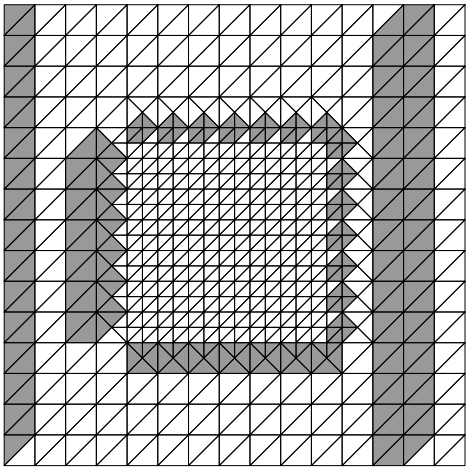


Figure 5.2: Orbits

$$a(N_i, N_j) = \begin{cases} \text{if } i \in I_j, & \sum_{k \in E_i} \Big(u_{\triangle_k} \int\int_{\triangle_k} \frac{\partial N_i(x,y)}{\partial x} N_j(x,y) dxdy \\ & \quad + v_{\triangle_k} \int\int_{\triangle_k} \frac{\partial N_i(x,y)}{\partial y} N_j(x,y) dxdy \\ & \quad + K_x \int\int_{\triangle_k} \frac{\partial N_i(x,y)}{\partial x} \frac{\partial N_j(x,y)}{\partial x} dxdy \\ & \quad + K_y \int\int_{\triangle_k} \frac{\partial N_i(x,y)}{\partial y} \frac{\partial N_j(x,y)}{\partial y} dxdy \Big), \\ \text{if } i \notin I_j, & 0. \end{cases} \tag{5.6}$$

5.4.3.3 Objectives

Harmless to the rest of code. Quick and small changes.

5.4.3.4 Approach

We have a framework that computes orbits of the integrals in equation (5.2) on page 89. It should be easy to duplicate the program lines in it concerning these integrals, and in this way change the framework functionality for handling the model by the equation (5.1).

5.4.4 Problem specification: Operator approximation

5.4.4.1 Intend

To use matrix operations for calculation of the operator A in (5.4).

5.4.4.2 Explanation

Calculations using matrix operations are faster since there are standard architecture optimized libraries for them, e.g., BLAS, PETSc.

The operator A depends upon time, because the wind field components u and v in (5.2) depend upon time. Though the data for them is provided in time intervals larger than the simulation step Δt. Since we are using chapeau functions the formulas for the calculation of the elements of A are linear, and we have

$$A\left(u\left(t_0 + \theta(t_1 - t_0)\right), v\left(t_0 + \theta(t_1 - t_0)\right)\right) = \tag{5.7}$$

$$A\left(u(t_0), v(t_0)\right) + \theta\left(A\left(u(t_1), v(t_1)\right) - A\left(u(t_0), v(t_0)\right)\right), \tag{5.8}$$

where $0 \leq \theta \leq 1$, and t_0 and t_1 are the times at which two consecutive wind field datasets are provided. So instead of calculating (5.7) we can calculate (5.8), which involves just matrix operations.

5.4.4.3 Objectives

Faster than the usual computation of the operator A.

5.4.4.4 Approach

Since we have constant time step Δt and because of the linearity of A, we can at time t_0 calculate the matrix $A\left(u\left(\theta(t_1 - t_0)\right), v\left(\theta(t_1 - t_0)\right)\right)$ with $\theta = \frac{\Delta t}{t_1 - t_0}$, and then for $t = t_0 + k\Delta t$ calculate $A\left(u(t), v(t)\right)$ as

$$A\left(u(t_0), v(t_0)\right) + k\,\theta A\left(u(t_1) - u(t_0), v(t_1) - v(t_0)\right). \tag{5.9}$$

The class responsible for the calculation of A for each iteration, should also be responsible for the calculation of

$$\theta A\left(u(t_1) - u(t_0), v(t_1) - v(t_0)\right).$$

When the diffusion is included in (5.2), i.e. when the equation (5.1) is solved (See Section 5.4.3 on page 91), there should be a method that excludes the diffusion when $A\left(u\left(\theta(t_1 - t_0)\right), v\left(\theta(t_1 - t_0)\right)\right)$ is calculated, since the framework now computes by default the operator

$$A\left(u(t_0), v(t_0)\right) + D + k\,\theta A\left(u(t_1) - u(t_0), v(t_1) - v(t_0)\right)$$

instead of (5.9).

5.4.5 The applied framework pattern

At first I was thinking in terms of the **Conceptual Layering** pattern with the participants:

1. ***Conceptual Layer*** Galerkin Finite Element Methods Layer (GFEML): calculating the operators for spatial discretization and time stepping;
2. ***Building blocks*** Files with mesh description, and their handling;
3. ***Building blocks*** The PETSc library for parallel solving systems of linear equations using MPI, [9], [8];[5]

When I applied this pattern with the participants above I was actually the applying the **Multi-Level Framework** pattern. The content of the sections above suggest the following conceptual layers:

1. ***Conceptual Layer*** Galerkin Finite Element Methods Layer: calculating the operators for spatial discretization, time stepping; see Section 5.5 on page 94;
2. ***Conceptual Layer*** Mesh Generation Layer(MGL): mesh generation, integrals on the mesh; see 5.7 on page 103;
3. ***Building blocks*** Linear Solvers Layer(LSL): library for solving systems of linear equations;
4. ***Conceptual Layer*** Data Handling Layer: the only data the advection module needs is the wind field data; the data should be approximated on the refined meshes; see 5.6 on page 97.

From Section 5.4.2 it is clear that we should facilitate the Row Reuse approach. Since we use static mesh, the orbits can be determined outside GFEML. Of course the generation of the mesh itself, should be also outside of the GFEML. I chose to make the mesh generation and all additional visualization routines with the computer algebra system *Mathematica*; see [56].

[5] PETSc is tuned to wide range of parallel architectures, distributed and shared.

Finite element methods reduce the problem of solving a differential equation to a system of linear equations. So we need building blocks for solving parallel sparse linear systems. That level is provided by the library PETSc.

One of the drawbacks of **Multi-Level Framework** is that the construction costs are not payed back. It is not clear for me whether all the *Mathematica* code I wrote will be (re-)used. Nevertheless, I needed these programs to understand how the whole system will be organized. Large part of the *Mathematica* code is to check the results from the refinement computations: this kind of visualization is not handled by the existing UNIRAS FORTRAN 77 code of DEM. On the other hand, having a powerful computer algebra tool like *Mathematica* included in the framework, speed-ups the embedding, into the production code, of the mathematical concepts in which the framework unrolls. Also, it is easier to analyze the production code properties that are rooted with some preliminary prototyping in the computer algebra tool. In general, this combination should make easier the resolution of the mathematical ideas to the practical computations. We should also have in mind, that the *Mathematica* mesh generation code can be used as a prototype for Python[6] or C++ implementation of it.

PETSc is a powerful object oriented library for scientific computations. The motto of the team developing it

> "Software is like a garden; it needs constant loving care to thrive."[9]

explains my trust to it as a well designed, rich library, and a valuable tool for programming and performance monitoring/tuning.

5.5 Object-oriented class design of the GFEM layer

The framework layers were identified in the sections above. Now we ready to design the structure of the classes that embody and implement the concepts of GFEML (see Section 5.4.5).

5.5.1 The applied design patterns

I have applied the following design patterns:

1. **Template Method** for the hierarchy of GFEM classes;
2. **Abstract Factory** to enforce consistency of the GFEM class building;
   Abstract Factory is applied upon **Template Method**;
3. **Strategy** for computation of the operators A and P;
4. **Strategy** for orbit handling;
5. **Strategy** that provide the wind field values to a GFEM object.

5.5.2 Forming the finite element method class

A Galerkin finite element numerical algorithm, that finds an approximate solution of the evolution equation (written in operator form)

$$P(x,y)\frac{\partial c}{\partial t} + A(x,y,t)c = 0, \tag{5.10}$$

over the time interval $[t_0, t_1]$, and has the following steps:

1. Calculate a matrix approximation $\overline{P}$ of the operator $P(x,y)$.
2. Calculate a matrix approximation $\overline{A}$ of the operator $A(x,y,t)$.

[6] A powerful pure object-oriented script language. See www.python.org .

3. Impose the boundary conditions on $\overline{P}$ and $\overline{A}$.
4. Solve the derived ODE's with, say, the Crank-Nicholson scheme for the time step Δt. I.e. find $c(x, y, t + \Delta t)$.
5. If the end of the time interval t_1 is not reached go to 2.

We will call this algorithm *Iteration*. *Iteration* is invariant when different spatial discretizations are used. If the time stepping in item 4 will be the same for all space discretizations confined by ADS framework, then it is obvious that the **Template Method** Design Pattern (DP) should be applied. (See the Applicability section of **Template Method** in [20].) Even if the time stepping differ we can use either **Strategy** DP, or overload the signature of *Iteration*. The class that contains *Iteration* and play the role of ***AbstractClass*** in **Template Method** will be called `FEM2D`. This class should be sub-classed when is desired a GFEM with some new properties. Because finite element methods for the spatial discretization of (5.10) have different approximate operators $\overline{P}$ and $\overline{A}$, it is natural to provide a separate hierarchy for the operators. The operators use the Row Reuse-inspired class hierarchies. Since it is preferable to localize the class configuration responsibility in just one class, I made `FEM2D` to play also ***AbstarctFactory*** in **Abstract Factory**. I choose `FEM2D` to provide its own configuration from a framework understandability point of view.

The application of these two patterns is shown on Figure 5.3.

5.5.3 Applying the Row Reuse approach

The Row Reuse approach (see Section 5.4.2) suggests to have a class, different objects of which traverse the different orbits and calculate the formula (5.5) for each node within their orbit. This class is called `OrbitNode`.

`OrbitNode` should have method that fills the row `row_number` of the matrix `operatorA`. The method's signature is `CalculateAKIntegrals(Matrix operatorA, int row_number)`. This method needs the values of the wind field on the elements the node `row_number` is a member of (the `row_number`'s patch). So it is natural to have another class, let us call it `Wind`, that represents the wind field and has methods like `WindXOnElement(int element)` and `WindYOnElement(int element)` for the values of $u_{\triangle_k}$ and $v_{\triangle_k}$ in (5.5). Since we have different types of wind field presentations, it is natural to apply the **Strategy** DP for the wind field classes. The class `Wind` plays ***Strategy***. An orbit can be traversed or stored in different ways. So, the **Strategy** DP can be applied again. The class `Orbit` plays ***Strategy***. The corresponding design is shown of Figure 5.4.

When the mesh is regular we can compute the orbit members quickly at run time. That is why int the current implementation (just) for the regular grid computations the classes `OrbitNode` and `Orbit` are joined into one class named `Node`, and is added another class `Mesh`.

5.5.4 UML sequence diagram

The sequence diagram on Figure 5.5 shows a simplified interaction of the ADS objects that perform the GFEM algorithm. Below are given comments of the different time levels of this diagram.

1. Creation. As it was said `FEM2D` acts like **Abstract Factory**: it triggers the creation of the desired descendants of `Operator2D`, `Orbit`, `Wind`, `Node`. In the illustrated sequence is used the descendant `OrbitNode` of `Node` in order to show its interaction with the orbit class `Orbit`.
2. The `Operator2D` object asks the PETSc `Mat` object for the range of the matrix owned by the local processor assuming that the matrix is laid out with the first $n1$ rows on the first processor, the next $n2$ rows on the second, etc.
3. The `FEM2D` class starts the simulation loop. In each loop step the matrix presented by the `Operator2D` object is recalculated. Because A's ownership range is known, the `OrbitNode` object is asked to calculate just those rows of the matrix A which are within this range.
   Although the calculation of the matrix P is not included in the sequence diagram, it is done in the same way outside of the simulation loop (since P is constant).

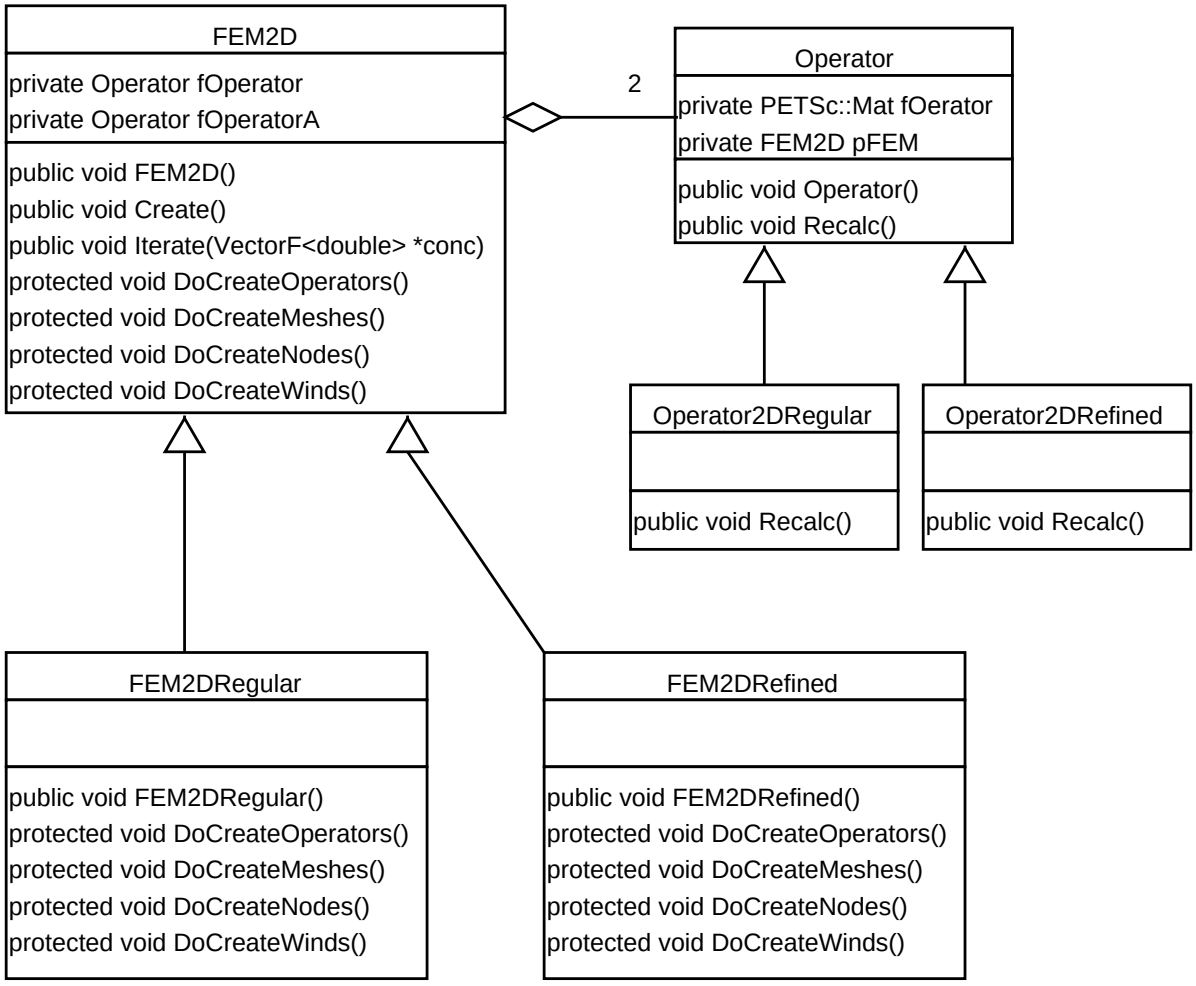


Figure 5.3: **Template Method** and **Abstract Factory** applied to the class `FEM2D`.

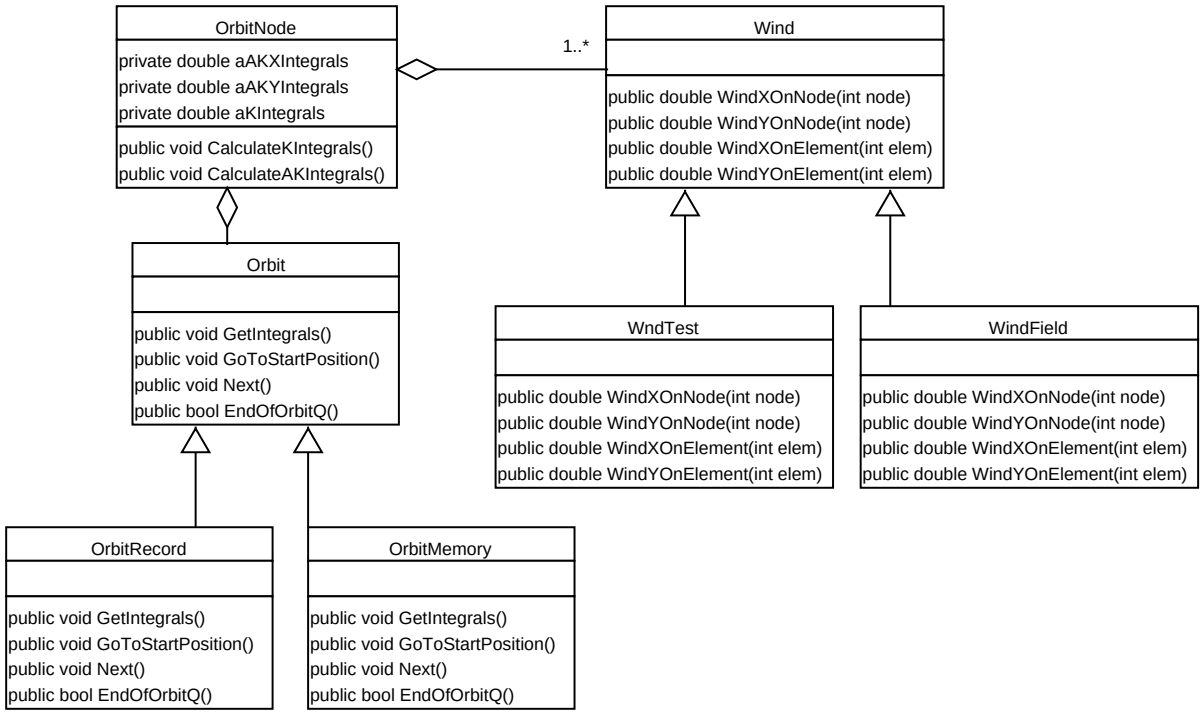


Figure 5.4: Class design for the Row Reuse pattern

4. The recalculated operators are set in the PETSc object SLES (System of Linear Equations Solver). Then the solution is obtained, together with the number of iterations.

5.6 Object-oriented design of the Data Handlers layer

5.6.1 Problem specification: Data handling

5.6.1.1 Intend

To provide the values of a changing in space and time scalar field on a locally refined mesh.

5.6.1.2 Explanation of the context

We have data files that represent on some regular rectangular grid the scalar fields used in DEM. Different files can represent data on different grids. The file grids can be finer or sparser than the basic, regular, simulation grid. Over the basic grid we have one or several refinements; see Figure 5.6. For a refinement there can be, or cannot be, a file with refined data – the field data should be approximated for the new nodes the refinements introduce.

There are two different types of approximations *emission like* type and *wind like* type. The values of the wind field should be also given on the elements of the grid, not just on the grid's nodes. Since the simulations are made in parallel, each processor should have its part of the data (this is a SIMD model). The data files can have different formats (ASCII, HDF, binary, etc.)

5.6.1.3 Objectives

Able to use FORTRAN 77 arrays. Amenable for usage in FORTRAN 77 code. Amenable for easy, natural enhancement for handling unstructured grids. Able to use different file formats. The field approximations and parallel distribution must not be slow.

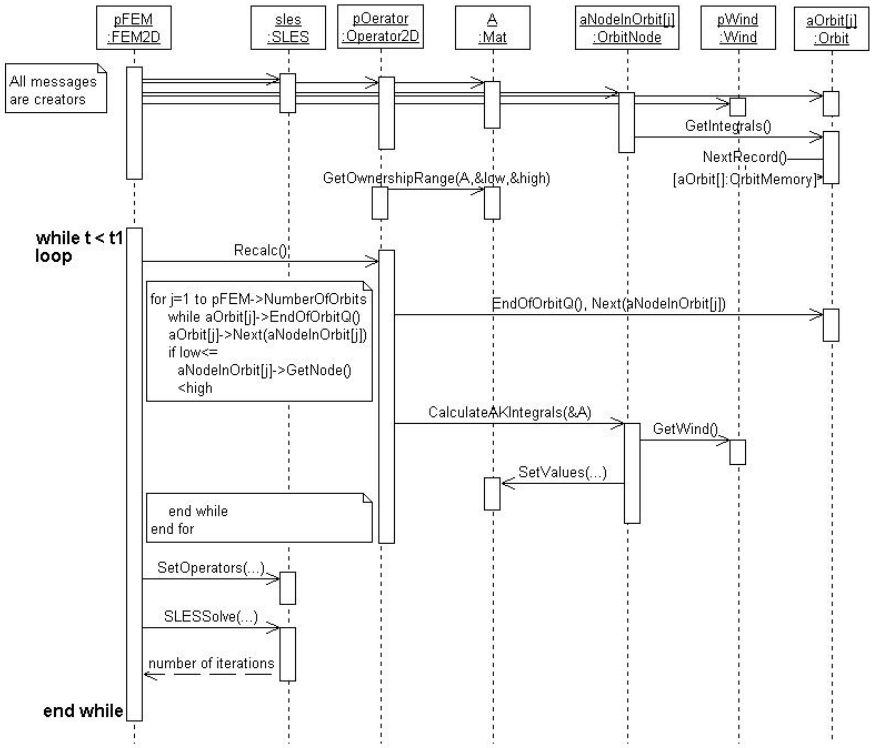


Figure 5.5: Sequence diagram for the GFEM layer

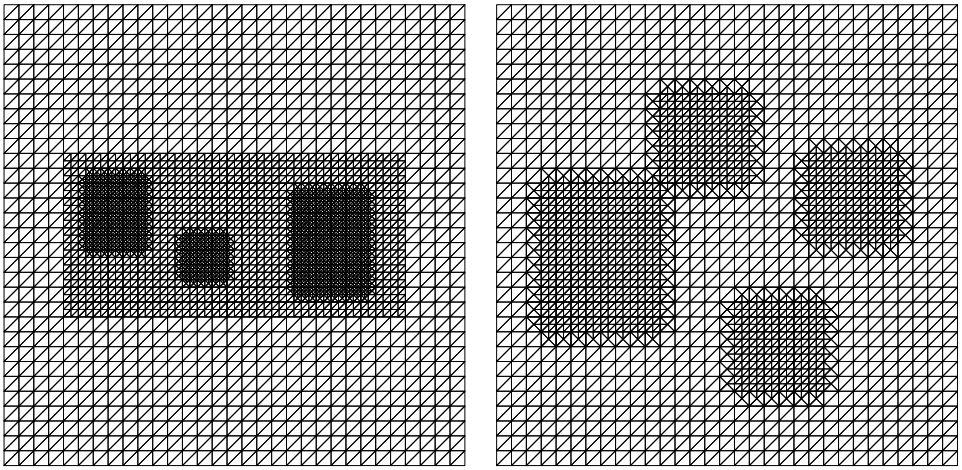


Figure 5.6: Example meshes

5.6.1.4 Approach

As it is decided in Section 5.4.1.4 for the computer presentation of the concentration field we use an one-dimensional array (the vector g): we have introduced a linear order of the nodes of grid. The same linear order should be imposed on the other fields in DEM. It is natural to think that we have an entity (a black box, a class) called *Field*, which have the method `Item(integer n)` that gives the value of the *Field* on the node with the number `n`. Another natural method is `GetFieldIn(double dataBuffer[], double time)`, which gives the the values of the entire approximated in space and time *Field* in the array `dataBuffer` at the time `time`. Of course, we can load the entity *Field* with the responsibilities of the overall data feeding process, which is natural to divide in the following stages:

1. Reading of the data files
2. Data approximation on space and time (on the coarse regular grids)
3. Nested refinements handling
4. Element approximation for the wind field
5. Parallel distribution

The parallel distribution can be done as preprocessing, i.e., the data is splitted into n parts for n-process simulation. But this preprocessing will pass all the stages above.

5.6.2 The design patterns used

I have used the following design patterns

1. **Essence** for the notion of *Field*. *Field* is the class that encapsulates all stages;
2. **Strategy** for Reading of the data files;
3. **Strategy** for Data approximation on space and time;
4. **Chain of Responsibility**/(**Composite**) for Nested refinements handling;
5. **Decorator** for Element approximation of the wind field;
6. **Decorator** for Parallel distribution;
7. **Builder** to construct the objects of *Field*.

5.6.3 Fields

A field changes in time and space. The class *Field* serves as an abstract class that provides an interface for data reading and writing, building of *Field* objects and parallel division algorithms. The class serves the transition of FORTRAN code to C++. It has data reading and writing interface and parallel division algorithms interface. Its **builder interface** is for adding data readers and writers, and data handlers for grids with local refinements.

The class evolved (mutated) over the stages ***Strategy/State*** in **Strategy/State** DP, ***AbstractFactory*** in **AbstractFactory**, `Product` in **Builder**. The data reading and writing methods can be seen as **Template Method** primitive operations.

5.6.4 Readers and Writers

We should have different algorithms to read the data stored in different formats. The same is true for the data writing. Though all this algorithms should have the same interface: `GetDataIn(...)` for the readers, `PutDataFrom(...)`. This is a clear situation where the **Strategy** DP should be applied. We can say that the `GetDataIn` (`PutDataFrom`) interface of *Field* is propagated to the Readers (Writers) hierarchies.

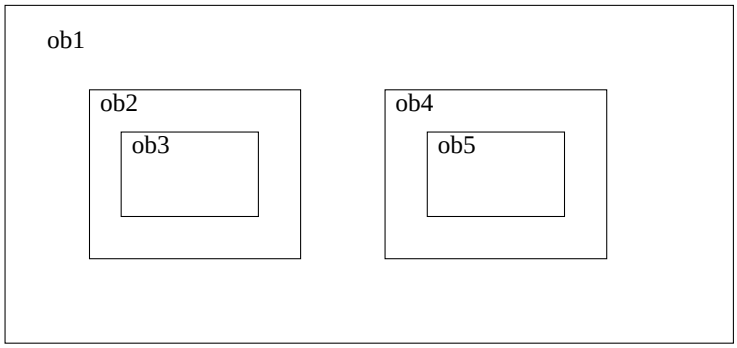


Figure 5.7: Nested mesh objects

5.6.5 Nested data handlers

For each data file we make, possibly with some approximation, a data buffer. For each type of approximation over a mesh region we have a *NestedDataHandler* class. Under **data handling** we understand the mapping from the mesh to the data in the files. A *NestedDataHandler* class gives the value that corresponds to a node in particular region of the mesh calculated from the data buffer. The node value can depend on several entries in the data buffer.

From the mesh generator we have the rules that describe from which entries a node's value is determined. The *NestedDataHandler*s hierarchy gives a class for each different type of rules.

5.6.6 Chain of responsibility vs. Composite

5.6.6.1 Problem specification

We have a square region (of Europe and the surrounding seas) covered with a mesh like the ones on Figure 5.6. The nodes of the mesh are ordered in some way. The *Field* over the region is formed for the data in several different files. (One for the coarse mesh, and several for the refined data.) We want to find the *Field* value of a given node or a finite element of the mesh.

5.6.6.2 Why Chain of Responsibility

If we have objects for every separate region of the mesh, we can see the process of retrieving data in the following way: We ask $ob1$ for the value on node i. If $ob1$ cannot give the value it forwards the request to $ob2$. If $ob2$ cannot give the value it forwards the request to $ob3$. If $ob3$ cannot give the value it forwards the request to $ob4$. If $ob4$ cannot give the value it forwards the request to $ob5$. If $ob5$ cannot give the values it gives 0 or issues an error. See also Figure 5.13.

5.6.6.3 Why Composite

We can have a dictionary that says in which file we can find the value for a given node number. The dictionary is hold by *Composite_Interface* and the leafs are different *NestedDataHandlers*. The advantages of **Composite** against **Chain of Responsibility** are that it retrieves faster the nodes' values and uses simple algorithm. Also with **Composite** can be done approximations of the node value based on several data files (when the nodes belong to the smoothing region).

The **Chain of Responsibility** design in the data handling layer can be easily transformed to **Composite**. Actually we should just add a new class playing **Composite**, and extend the interface of ***NestedDataHandler***.

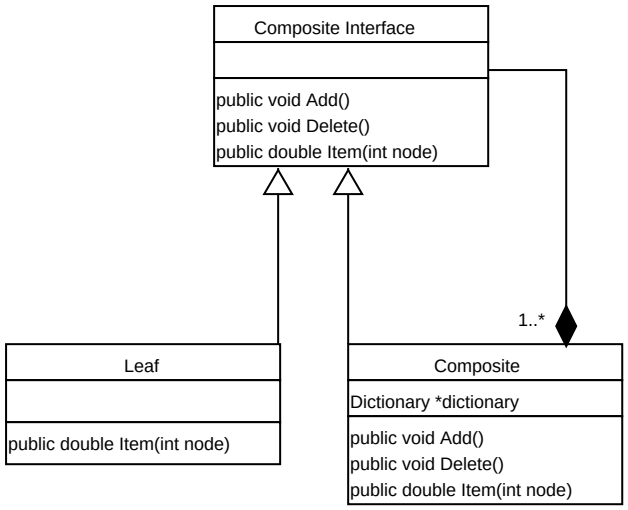


Figure 5.8: The **Composite** design pattern

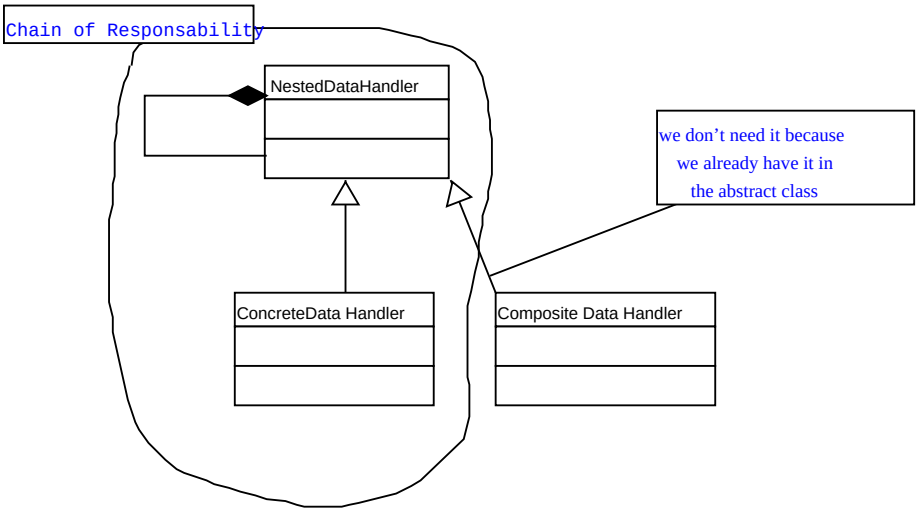


Figure 5.9: Moving from **Chain of Responsibility** to **Composite**

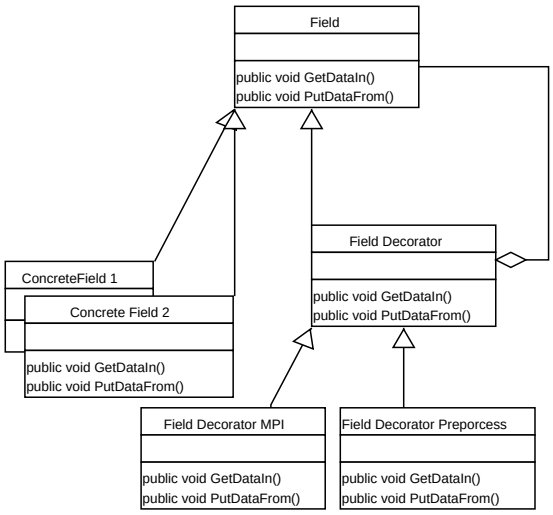


Figure 5.10: **Decorator** for parallel distribution

5.6.7 Adding responsibilities to the *Field* class

5.6.7.1 Decorator for parallel data handling

We can see the parallel distribution of the handled data as additional responsibility to the primal handling. Let us consider two types of parallel data handling. The first is when the data is handled just on the master process and then the master process sends to the other processes the data they need. The master process approximates the data, partitions it, and distributes it among all processes. Since often several(many) simulations are done over the same data, but with different scenarios, each process can save the part of the data it uses to a file for the next simulations. The data feeding directly from these files is the second type of parallel data handling.

Obviously the **Decorator** pattern can be applied to *Field* and its hairs, seeing *Field* as ***Component*** and its hairs as **ConcreteComponents**. The **ConcreteDecorator**s will be the two types of parallel data handling described above.

5.6.7.2 Decorator for element approximation

We can consider the element approximation as additional responsibility to the data handling that provides the node's value over a *Field*. The *Element decorator* playing `ConcreteDecorator` in the pattern, will posses a `TNestedElemHandler` object to approximate over the elements a *Field* which is already approximated over the nodes.

5.6.8 UML sequence diagrams

The creation of the objects of the DHL is facilitated with the **Builder** DP. The interface of the *Field* class was extended with the **Product** interface.

A possible creation scenario is shown on Figure 5.12. We can see how the *Field* object `FieldNestedRules` is configured with some desired combination of *NestedDataHandler* objects via the **Product** interface methods `AddXXX()`.

The stages the of the DHL data handling process with these objects are shown on Figure 5.13:

1. The main program inquires the values of a data field with the *Field* method `GetDataIn`. The request is handled differently on the different processes.
   On the master process (note the `[rank==0]` condition) the `GetDataIn` message triggers the approximation of the whole data field with the *NestedDataHandler* objects:

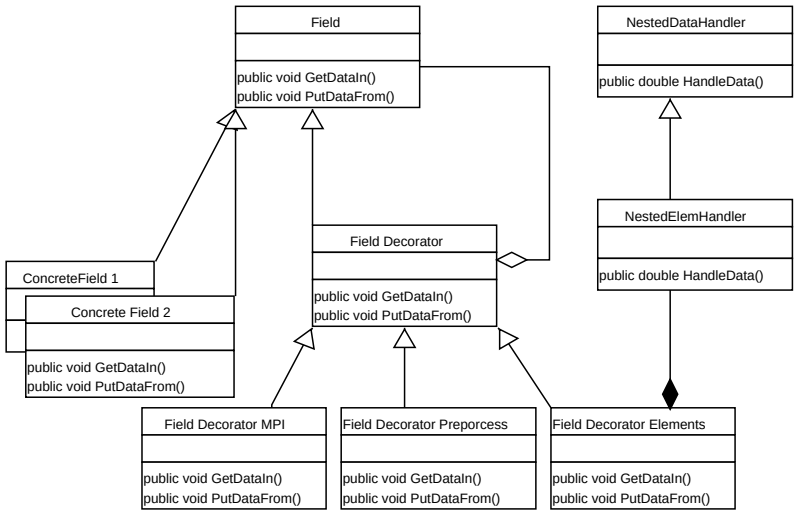


Figure 5.11: **Decorator** for parallel distribution and element approximation

(a) For each node i in the field the request `HandleData(`i`)` is sent to the *NestedDataHandler* object responsible for the mother mesh, `pNest`;

(b) If the node i cannot be handled by the `pNest` object, it forwards the request to the object responsible for the smoothing region, `pSmooth`;

(c) If the node i cannot be handled by the `pSmooth` object, it forwards the request to the object responsible for the refined region, `pBird`;

(d) If the node i cannot be handled by the `pBird` object, it issues error message;

2. After the approximation is ready on the master process, the parts of the partitioned approximated field are sent to the child processes.
   On the child processes the `GetDataIn` message causes the `FieldDecoratorMPI` object to wait to receive the portion of the field for the local processor.

3. The local parts of a field can be written on files and then reused in a consequent run with the same data.

5.7 Mesh Generator layer

5.7.1 The addressed problem

We should generate locally uniform grids with nested refinements. The grids are static. The Mesh Generator Layer (MGL) should be flexible on where is the particular region on which the grid is concentrated. Typically, a grid, once generated, will be used for a long term. It is assumed interactive generation of the grids.

There should be routines that support the division of the grids into orbits: the Row Reuse orbits[7], which are sets of geometrically equivalent patches, can be identified in the MGL: the GFEML will read their description from files with suitable format. To identify an orbit means to find the nodes it consists of and to calculate the integrals in formula (5.6) for a representative patch.

The MGL should generate files with approximation rules that describe the *emission like* type and *wind like* type approximations over the grids with refinement; (see also Section 5.6.1.2). These files should provide description for fast retrieving of the approximation rules.

[7]See Section 5.4.2.

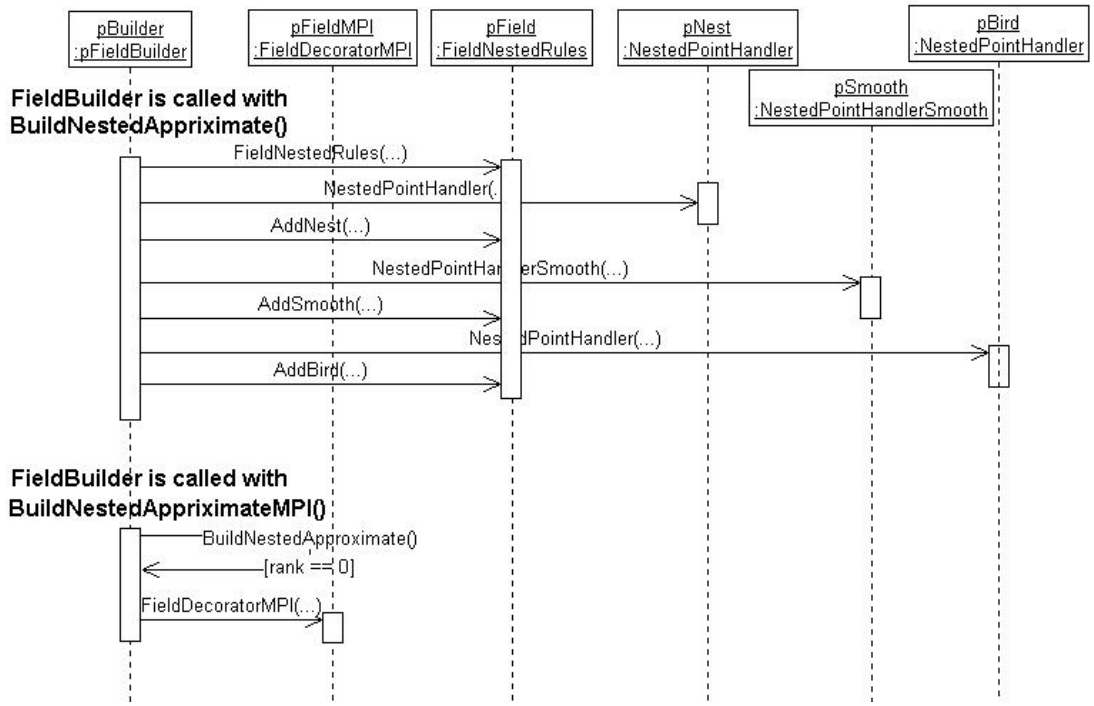


Figure 5.12: Sequence diagram of object creation in the Data Handlers Layer

5.7.2 The general patterns used

Two general patterns were used:

1. **"Divide and conquer in black boxes"**, and
2. **"When you need a screwdriver, use the general purpose hammer"**.

The "general purpose hammer" is *Mathematica*, and the functions I wrote in it are based not on the object-oriented approach, but on the "quick and dirty" approach, and the good, old fashioned, functional decomposition approach.

5.7.3 Algorithms + data = mesh generator

Since the grids are static, i.e. the speed of the generation is not crucial, and the grids are supposed to be composed interactively with the user (a human being), I decided the data structure of the grids to be a list of two-dimensional elements (triangles or squares), every element being given with the coordinates of its vertexes. Although redundant (a node is repeated several times: 6 for triangles and 4 for squares, when classical conformal methods are applied), this structure is very convenient *to device and to write* the routines for refinement, interface refinement(smoothing), integrals calculation, orbit determination, grid manipulation, grid visualization, and file description generation for the GFEML. The grid manipulation, especially, becomes quite elegant, since we can do intersections, unions, and complements of different grids or grid parts. This sort of "set manipulation tailoring" appears to be quite suitable to compose locally uniform grids.

The process of grid generation goes through the following nine steps (see the *Mathematica* appendixes "Mesh Generation for the Object-Oriented Danish Eulerian Model. User's Guide", "Mesh Generation for the Object-Oriented Danish Eulerian Model. Programmer's Guide"):

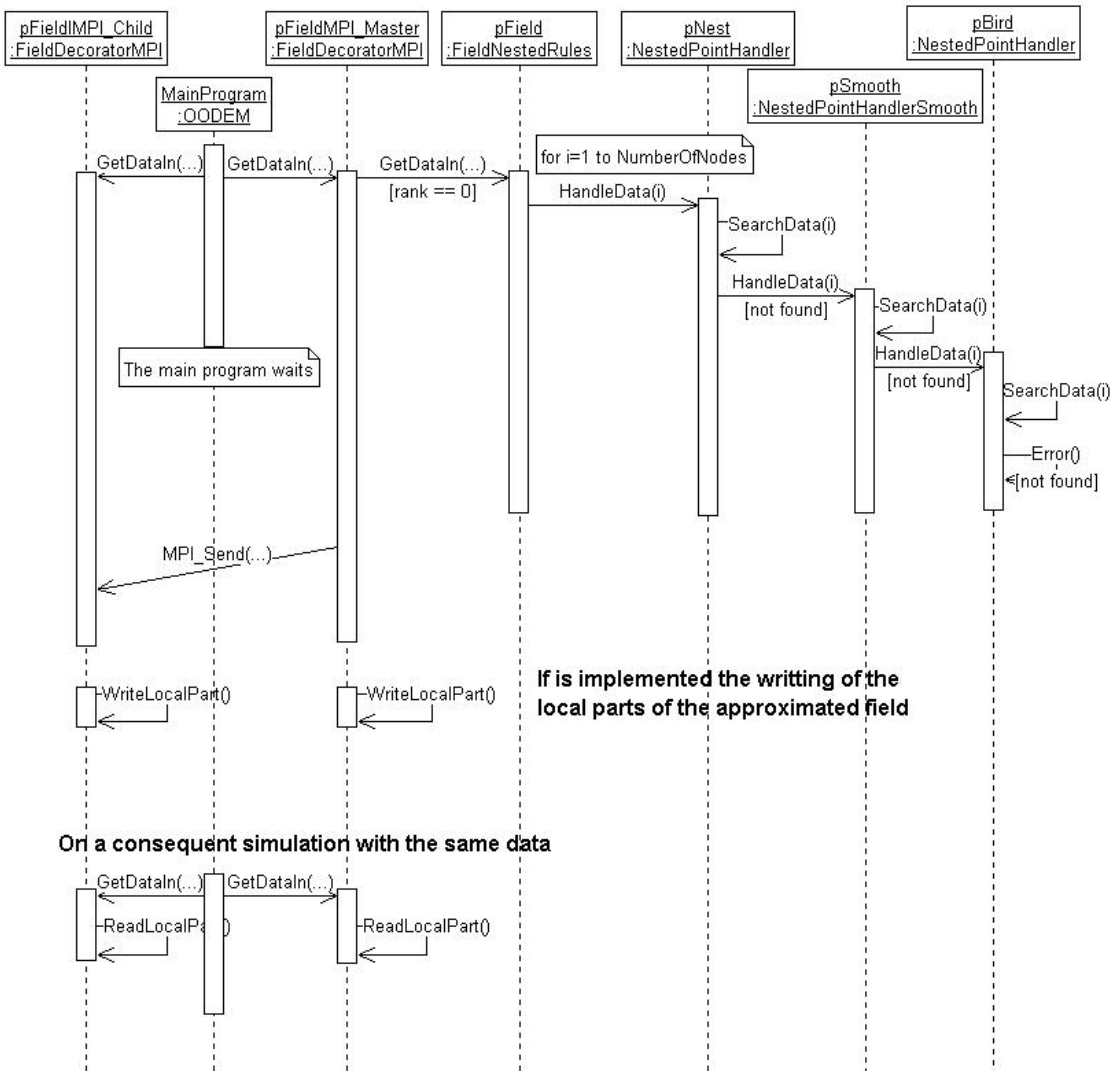


Figure 5.13: Sequence diagram of data handling by the Data Handlers Layer

1. Initialization of the grid parameters and generating a regular grid;
2. Cutting off a cell from the regular grid, where the refinement should be ;
3. Telescopic refinement of the cell to the desired level;
4. Plugging in of the refined cell;
5. Ordering of the nodes and the elements of the grid
6. Preparing a binary tree for the retrieving of the queries: (i) which are the neighbors of a node (ii) to which elements a node belongs to;
7. Finding the orbits;
8. Writing the orbits (this includes the calculation of the integrals);
9. Preparing and writing the rules and dictionaries for the Data Handling Layer.

A grid with several disjoint refinements passes the same stages. It is possible to tailor grids with overlapping regions, but this needs familiarity with *Mathematica*.

5.8 Usage of PETSc

The procedures of the library for scientific computing PETSc [9], [8], are called in various places in the C++ code of the ADS framework. Since in my opinion PETSc is a very well designed object-oriented library, I took the approach to make their names, functionality and relations standard i.e. if we want to use an other library or our own linear solvers code, they should mimic the PETSc library.

That imitation comes easily, since the GFEM concept classes can be just templated with a class that interfaces the chosen library. Essentially this is the **Strategy** DP with interface class methods that correspond, and have the names, of the PETSc routines. We can apply the pattern **Adapter** to provide the classes corresponding to the classes of PETSc `Mat`(parallel matrix), `Vec` (parallel vector), `SLES` (parallel linear systems solver), etc.[8]

5.9 Framework design of the chemistry sub-model

We cannot give concise and at the same time precise description of the CS like the one of the ADS. We refer to [58, Section 2.6] for a complete description of the CS from mathematical (mathematician's) point of view. The description here is just enough for addressing the program design of the CS framework. (The reader might want to refer to Section 2.5.)

5.9.1 The addressed problem

The CS is based on the so called box-model. The modeling region is partitioned, by the grid, into boxes; see 2.9.2. It is assumed that at the time discretization levels the pollutants within a box are well mixed: the chemical reactions, on which the chemistry mathematical model is based on, are written with this assumption. We can say that our modeling region, Europe, is covered with stirred tank reactors;[9] see Figure 5.14.

From the chemical reactions, chosen according to some reaction mechanism, is derived a system of ODE's. Some of reactions are photochemical i.e. they begin with a photon absorption by an atom or a molecule. Therefore they will depend from the intensity of the sun light, which on a given time level depends upon the geographical coordinates of the box region, and the height and density of the clouds in it. The cloudiness is read from the files with meteorological data, but the photochemical coefficients of the reactions should be calculated for each time level.

[8] I suppose in some cases we can just use `typedef`.

[9] "Stirred tank reactor" is a chemical engineering modeling device for immediate and perfect mixing.

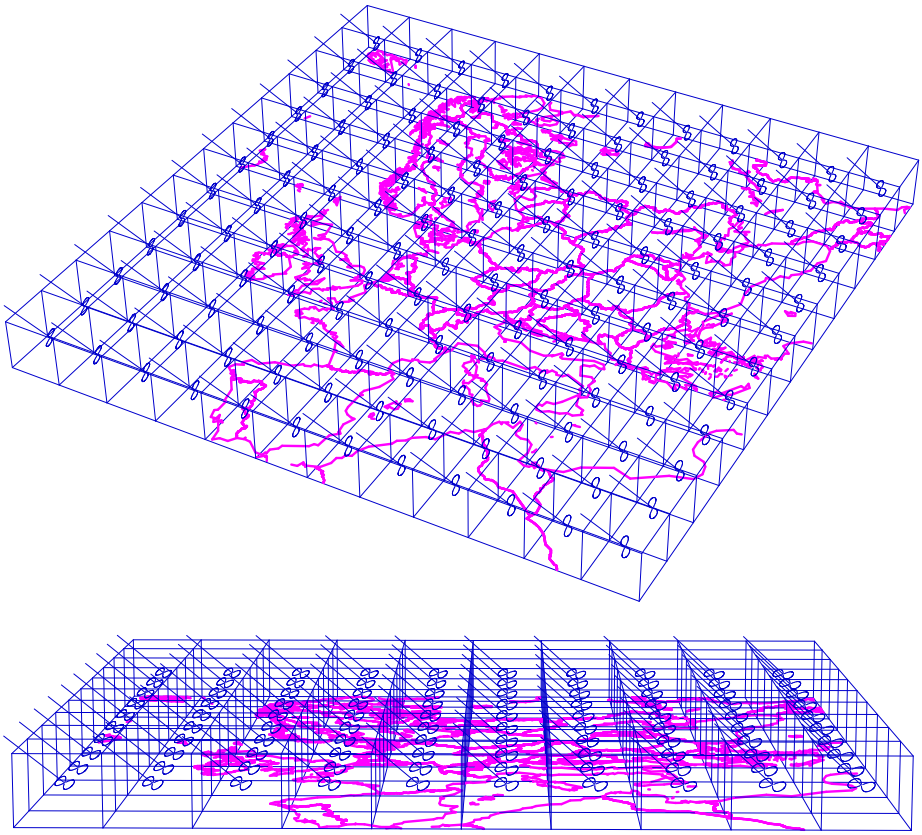


Figure 5.14: Europe covered with stirred tank reactors.

The symbol  is used in chemical engineering to designate a stirred tank reactor.

In the CS are also included dry and wet deposition simulations.

The simulation of the chemical reactions is computationally the heaviest task in DEM. So it is important to be fast. The CS program design should be independent from the adopted reaction mechanism; (currently in DEM is used CBM IV).

5.9.2 The patterns used

The work on the CS is collective: different kind of scientists work on it. I have the understanding that a group of chemists develop a reaction mechanism, a physicist works out the formulae for the photochemical coefficients, another physicist derives the deposition model, and finally a numerical analyst finds a suitable numerical method for the stiff system of ODE's derived from the chemical reactions. (Even if one person is going through all of the stages, he/she acts as the mentioned scientists.)

From numerical point of view the hardest question is how to treat numerically the stiff ODE system. It is suitable to tackle this problem separated from the others mentioned above: they are somewhat easier: if we have a solver for this system we should (just) attach to it routines for the rest of the computations: the photochemical coefficients, and the deposition. It is natural to apply the **Decorator** DP to coat the ODE system solver with the other routines, with which it should be included in DEM.

The pattern application is shown on Figure 5.15. Note the class called `ChuncksDecorator`, which provides better cash utilization.

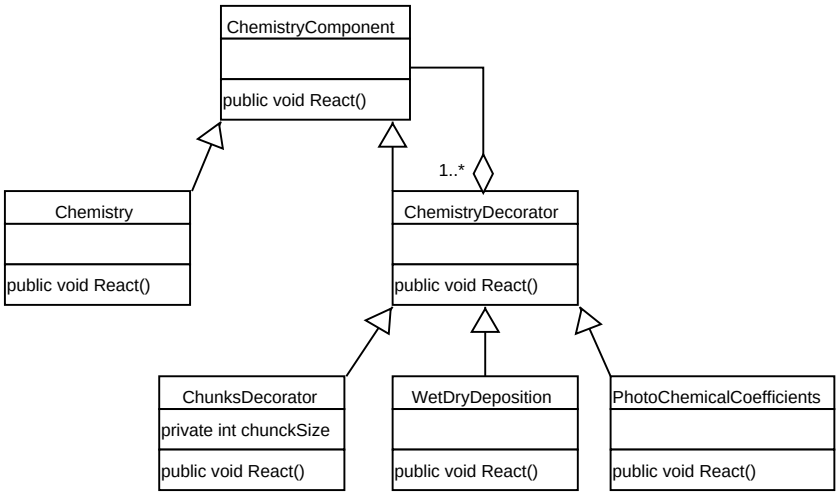


Figure 5.15: **Decorator** applied to the chemistry sub-model

See Chapter 8 for discussion of possible implementational extensions for the chemistry submodel.

5.10 Parallelism handling

It was decided to develop a mesh generator for locally uniform grids, and to employ PETSc ([8],[9]) for the parallel solution of the linear systems.

Since PETSc is based on the Message Passing Interface (MPI), and because the MPI model runs on all parallel architectures, MPI is reflected in the GFEM layer. The idea behind the way the parallelism is facilitated is similar to the ideas used in OpenMP, HPF and PETSc: the user achieves parallelism via domain decomposition, designing sequential code that is made parallel with minimal changes (comments in OpenMP and HPF, and name suffixes in PETSc). The **Decorator** DP for the data feeding was applied to the GFEM conceptual layer, and the **Strategy** DP for the parallel/sequential GFEM computations. Using these two patterns the sequential and the parallel code look in the same way: in order to add new parallel behavior, new classes are added; the existing code is not changed. Also, the use of the **Strategy** DP makes the GFEM classes independent of the linear solvers package.

The mesh generator provides description of the designed by the user grid. The grid nodes are ordered linearly according to their spatial coordinates. They are divided into orbits: each orbit contains nodes with equal patches. The description is read by the GFEM layer and all grid nodes are distributed in equal portions among the parallel processes. GFEM classes know the lowest and the highest number of the nodes any of the processes is responsible for. In this way a process can work with its own (possibly empty) subset of the nodes of each orbit.

For the machine presentation of the GFEM operators are used sparse matrices distributed over the processors; their handling is provided by PETSc.

5.11 The overall OODEM framework design

In the sections above I tried to show how the OODEM framework emerged. The top DEM layer is still in transition period; just the top layer of the CS framework is developed. Nevertheless, the general architecture of OODEM is clear; on Figure 5.16 are shown its layers.

The DEM layer is derived from the two main programs of DEM:

1. the first is for the setting up of parameters, the data files opening, and arrays initialization;
2. the second follows the splitting procedure in which

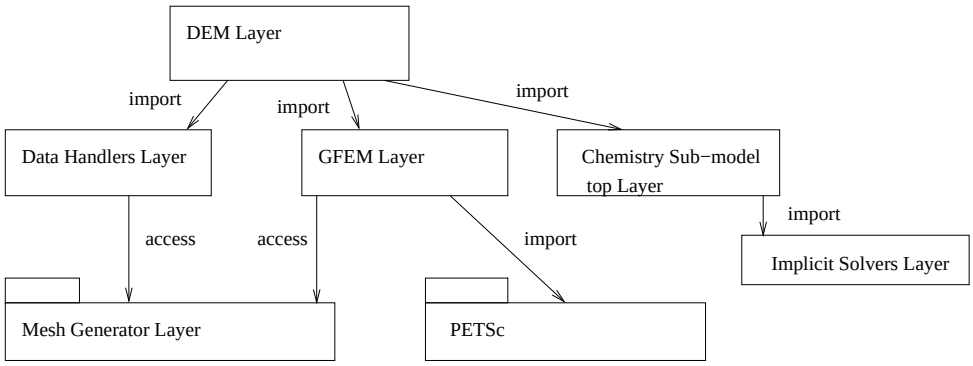


Figure 5.16: The structure OODEM framework

(a) before the simulation of each time step are called routines for reading of the emissions and the meteorological fields

(b) after each simulated time step is called a results storage routine.

If to all of the sub-layer classes used in the DEM layer is applied some creational DP like **Abstract Factory** or **Builder,** we can provide rather stable code for these two routines. See Section 8.3 in Chapter 8 for extended description of the DEM layer and discussion for its OO design.

Chapter 6

Object-Oriented Programming

It's cool to hate

– Offspring, *"Cool To Hate", Ixnay On The Hombre, 1997*

6.1 Why object-oriented programming and how to invent it

Mastering a language comes when we can feel the rules it is based on, and have the ability to reinvent them. Most (all) of the programmers can invent constructions like `for` and `if` and principles like "context encapsulation" and "divide and conquer". Here we will show the motivation behind the OO techniques and how to invent the basic mechanisms of OO programming: inheritance, polymorphism and dynamic binding. In a way you cannot discover them independently: if you discover one, the others comes immediately.

6.1.1 Stable programming code

When we program we do not want to change the programming code much. We want the code we write to be stable. For example, we can summate the first 10 numbers with the code

```
s:=0;
s:=s+1;
s:=s+2;
s:=s+3;
s:=s+4;
s:=s+5;
s:=s+6;
s:=s+7;
s:=s+8;
s:=s+9;
s:=s+10;
```

But what will happen if we want to summate the first 100 or the first 326 numbers instead? We should introduce more code. Of course, it is much better to have a construction like `for` and to write

```
s:=0;
for i:=1 to 10
  s:=s+i;
end
```

This is not so vulnerable to changes. We should just change `10` to whatever we want.

Now we should note that this code describes possibly hundreds or thousands of assignments and summations for someone who performs its algorithm. We have captured some invariant for the operations we want to perform, and used the invariant to write shorter and usable algorithmic description. This description – this code – is more stable than the description we started with.

In general, programming is based on invariants. Finding invariants and using them for shorter and more stable algorithmic descriptions. We will see in the next sections that OO paradigm offers mechanisms, additional to the those of the modular programming, and gives us better ways to use the invariants we discover.

Remark 6.1.1 *The paradigm of "Design by contract" by Bertran Mayer (one of the OOP creators) increases program reusability by making explicit in the code the invariants used to invent the algorithms.*

6.1.2 Stable programs with modular programming techniques

Modular programming usually advises to divide and conquer in black boxes, or more precisely, to divide and conquer in as much as it is suitable(possible) black boxes with the hope that this will produce program designs able to response to the changes of their subject. This is true to some extend, because in a structure which is unfolded and fine grained to the right levels, we can easier add new modules and new relations between them. Books like "Algorithms + data structures = programs" (Niklaus Wirth [55]) explore this to great extend. They also emphasize how important it is to start with the right data structures. In this book and other books for modular programming, writing programs is viewed, more or less, as solving mathematical problems: after careful consideration we come with some data/algorithm design. This sounds and in my opinion *is* a great approach, though it is not very applicable to start building programs with limited (though growing) knowledge about their subject, and hence with limited anticipation of their complete code.

6.1.3 Stable programs with polymorphism

The principle of modularity is extended in the OO paradigm with the concept of encapsulation. To the concept of encapsulation are added (the more important) concepts of the polymorphism, the inheritance, and the dynamic binding. So, let us try to invent them all.

We will use a typical example in which the need of them will become clear.

Let us consider the following code

```
Proc(double array x, int n)
  for i:=1 to n
    AddOne(x_i)
  end
end
```

where,

```
AddOne(double x)
  change x to x+1;
end
```

Apparently `Proc` is intended to be used for arbitrary long collections(arrays) of type `double`. What will happen if we want `Proc` to be applied to collections that contain different types of data? For example, if x_5 is a string, x_5=`'5'`, and x_7 is a vector, x_7=`{1,13,4}`? The operation `AddOne` is perfectly defined on x_5 and x_7: `AddOne('5')='6'`, `AddOne({1,13,4})={2,14,5}`.

One way to provide this wider functionality is to introduce some kind of type checking and use `case` statement (or nested `if-elses`), i.e. something like

```
Proc(array x, int n)
  for i:=1 to n
```

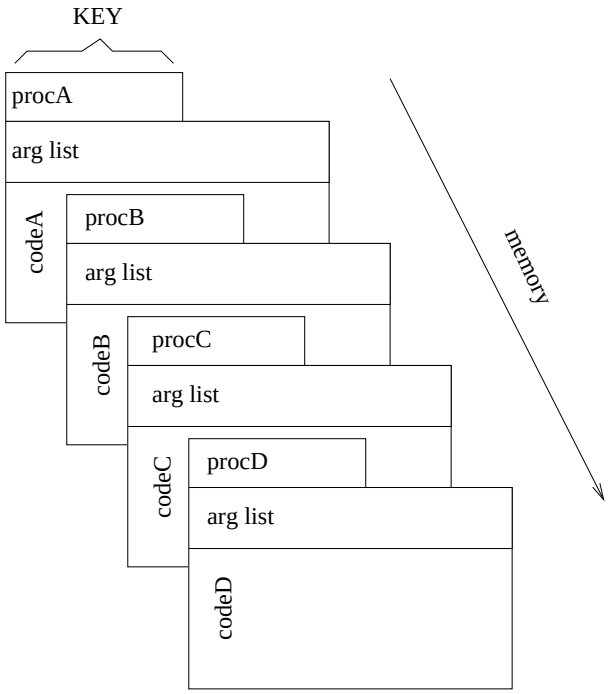


Figure 6.1: A model how the procedures are placed in the memory. Their names, `procA`, `procB`, `procC`, `procD`, are interpreted us keys with which the program execution system locates the code behind them.

```
      case x_i is double
        AddOne(x_i)
      case x_i is string
        AddOneForStrings(x_i)
      case x_i is vector
        AddOneForVector(x_i)
    end
  end
```

This code is not very stable: it should be changed each time we introduce a new type that can be put in the collection `x`. Also, it just doesn't look so clear as its first variant.

The solution above was in the paradigm of the modular (non object-oriented) languages. Let us look at one way out of this situation. To do that we will need a closer (but not much) look at how a classical, non object-oriented language like FORTRAN works.

In FORTRAN 77 after the compilation of a program, the code of a procedure is identified (can be found in the memory) by its name, and vice versa: the name of a procedure determines its code. There is one-to-one correspondence between the procedure name and its code. This restricts us to decompose the functionality of our programs to operations with precisely defined argument list. On Figure 6.1 is shown a model of how the procedures are placed in the memory with their names interpreted as a key with which the program execution system locates the procedures code.

It is clear that the code of a procedure can be defined with both its name and the arguments i.e. we can have "two field" keys as it is shown on Figure 6.2. The code of a routine can be located by its name and the type of an argument it uses. This makes the correspondence between the procedure code and its names one-to-many, and is facilitated in FORTRAN 90/95 by the mechanism of function overloading.

Instead of the function centric paradigm of FORTRAN 77, we can have an argument centric one, in which the argument types are the "primal" key. In correspondence to the ability to define a new function in FORTRAN, we can provide the ability to define a new type. It is especially interesting to define inclusion relations between the argument types considered as primal keys: if for a given type there is no

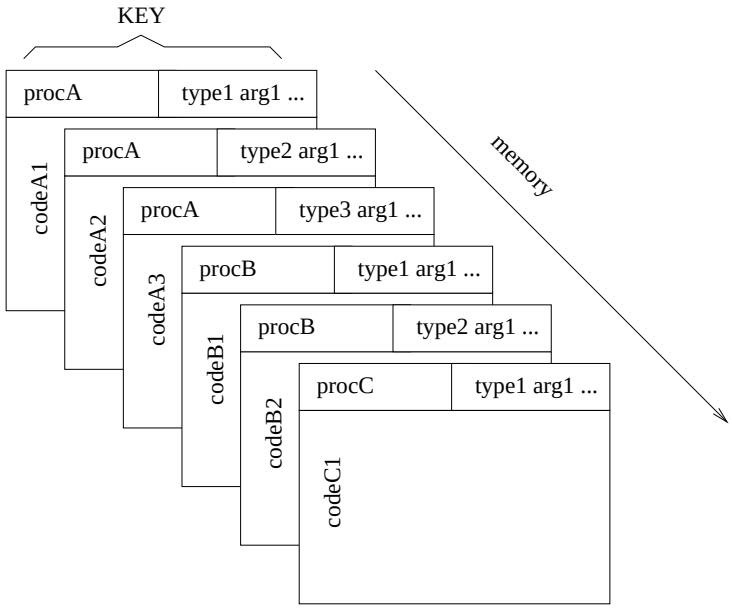


Figure 6.2: Memory model when the procedure name *together with* the argument list is interpreted as a key for the code behind them.

provided code for a given procedure name, it is taken the code provided for this procedure name and a type including the given type. This leads us to the object-oriented programming – we have chosen a type-oriented, and hence argument-oriented, point of view. The type inclusion idea leads us to the object-oriented notion of inheritance.

Let us now rephrase in more mathematical terms the idea of the transition from functional to argument oriented view of the programming code.

When we write a couple of symbols in the form $f(x)$, we just *emphasize* that we mean that f is applied to x. But we can write $(f\,x)$ with the same meaning, or even $(f)x$ or $x(f)$. It is just a matter of notation but there is no reason not to thing that x is applied to f. In mathematics (functional analysis) usually is used the notation $x^{**}(f)$ to denote a value *equal* to $f(x)$, and x^{**} is considered as a function, which domain is a set of functions. For example, if f is defined to be $f(y) := y + 1$, and g is defined as $g(y) = y^2$, then $5^{**}(f) = 6$, and $5^{**}(g) = 25$. Another example comes from Figure 6.2 above. If we consider the functions

```
procA:{type1,type2,type3}→{codeA1, codeA2, codeA3},
procB:{type1,type2}→{codeB1, codeB2},
procC:{type1}→{codeC1}
```

then the code in their co-domains is covered by the co-domains of the functions

```
type1**:{procA,procB,procC}→{codeA1,codeB1,codeC1},
type2**:{procA,procB}→{codeA2,codeB2},
type3**:{procA}→{codeA3}
```

This functions induce another ordering in the memory which is shown on Figure 6.3. From this ordering it is just natural to do the grouping shown on Figure 6.4, and in this way introduce the notion of *class*. The class encapsulates the behavior of each type (i.e. its procedures), and possibly some appropriate data or data structures.

Let us write the initial code for the procedure `Proc` having in mind the notational equivalence $f(x) \equiv (f)x \equiv x(f)$: instead of writing `AddOne(`$x_i$`)`,we will write x_i`.AddOne()`,

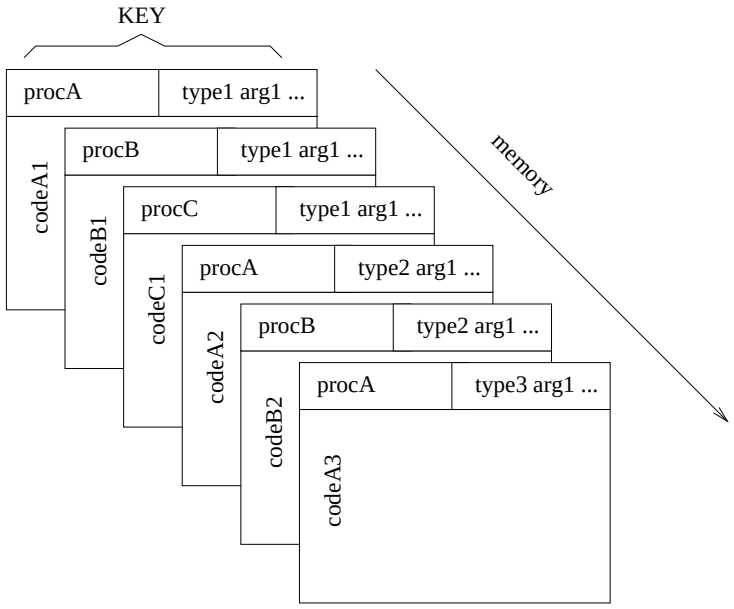


Figure 6.3: Type-wise reordering of memory layout on Figure 6.2. This one leads to the grouping shown on Figure 6.4.

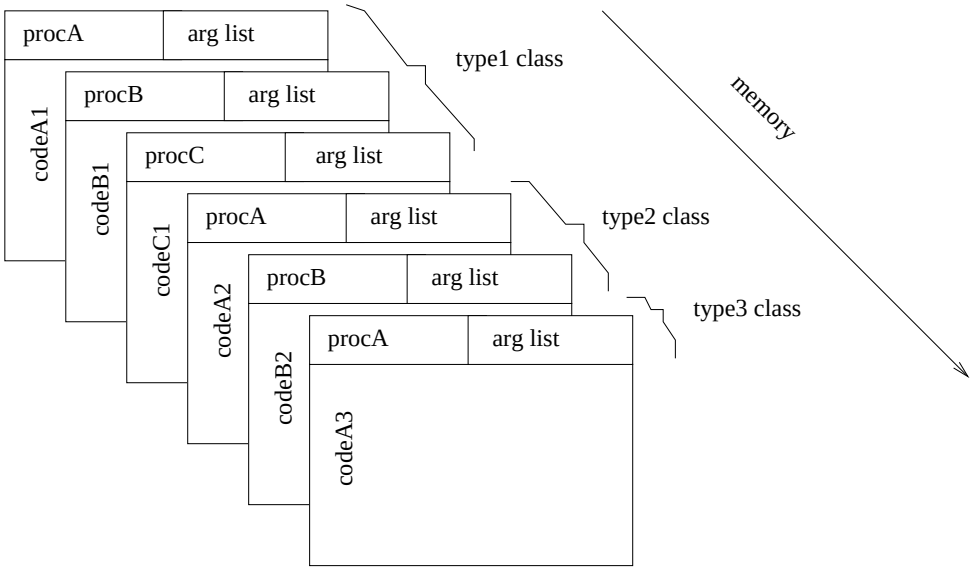


Figure 6.4: Class definition coming easily from the memory layout on Figure 6.3.

```
Proc(array x, int n)
  for i:=1 to n
    x_i.AddOne() ≡x_i(AddOne)
  end
end
```

Here is assumed that each x_i has its own definition of how *to be applied* on `AddOne`. This code would be invariant for whatever types we have in the collection `x`, as long as we have mechanisms that define and allow the implementation of this kind of function application.

We wrote the code above thinking that the array `x` is heterogeneous, but to specify the code to a compiler, `x` need to be defined of some type before the code compilation. We can resolve this sort of opposite forces defining a type that is common for all possible elements of `x`, and allowing/forcing the types of the `x`'s elements to be interpreted as this common one. We can define a common type, named `incrementable` which has the ability of being incrementable - be applied on `AddOne`. The types like `double`, `string` and `vector`, can take the places of `incrementable`'s elements, providing their own version of `AddOne`.

The ability for specifying the interpretation of some type as another one (more general, more common) is called *inheritance*. Getting the exact behavior at run time of `AddOne`, for any element of `x`, is called *dynamic binding*. The ability to substitute a type with a heir of it is called *polymorphism*. Dynamic binding is the mechanism that provides polymorphism at run time. Clearly, inheritance and polymorphism increase the stability of a program builded with them, since their provide multiple variants how the algorithms of the program to be completed to its executional form.

Remark 6.1.2 *In most non-OO languages we can define heterogeneous collections using pointers.*

Remark 6.1.3 *The first variant of `Proc` is in one-to-one correspondence with the code of `AddOne`. In order to provide stability for the code that calls Proc in its second variant, we introduced more one-to-one correspondences between `Proc` and variants of `AddOne` for different types. Polymorphism gives us the ability to specify one-to-many correspondences between `Proc` and the possible codes for `AddOne`.*

Remark 6.1.4 *Polymorphism in C++ is implemented via* virtual functions *[43]. When a function in a type is defined virtual, it is considered the possible application of the dynamic binding mechanism to its signature. Clearly, this decreases the code stability: the polymorphism should be anticipated in order to be provided. The reason however is due to efficiency.*

Remark 6.1.5 *The example above can be done via the mechanism of overloading provided by most languages. Overloading provides polymorphism, but it is too shallow: it provides just one level polymorphism. Though this is enough for the **Strategy** pattern (see [20], or the next chapter in the thesis). The overloading resolution mechanism in C++ [43] acts like the **Chain of Responsibility** pattern (see again [20]).*

Chapter 7

Design Patterns

... And when in that position
I'm the luckiest man alive

– OFFSPRING, *"Me & My Old Lady"*,
Ixnay On The Hombre, 1997

7.1 Introduction

In this chapter we will discuss design patterns and their usefulness. There are two ways to go into patterns: historically, and from the notion of normal forms, which is a more scientific one. We will take the later.

Normal forms are everywhere in mathematics. 'Normal' or 'canonical' means simpler in some sense. Going to normal form is usually like applying a pattern in the sense introduced by Alexander in [2]. We speak about normal or canonical form of curve equations in analytical geometry. We make LU or QR factorizations of matrices. The expressions in functional languages are reduced to their normal form. There are normal forms for both continuous and discrete dynamical systems. When we do data base design employing relational algebra, we try to make design in a normal form.

One with a background in the field of data bases, will naturally search how to apply the principles from that field to OOP.[1] These principles allow us to be ignorant in a very smart sense. A data base design build with them, i.e. a data base design in a normal form, is very stable: when our knowledge of the modeled problematic domain increases, we will never pre-structure this design, we will just add new elements to it. Stable programming code is highly desirable especially in the rapidly developed fields of scientific computation. It is potentially easier in the OO paradigm to make reusable and extendable code, though OO paradigm reaches its maturity with the development of the Design Patterns (DP) introduced first in [20]. We can say that a design pattern prescribe normal form for the software structure applicable in a given context.

In this chapter we will introduce the basic **Reflexivity principle** and we will show how we can invent some of the fundamental design patterns with it. Throughout in the text we will try to see the design patterns as normal forms and interpret them from the point of view of the theory of Relational Data Bases (RDB, [17]).

Remark 7.1.1 *In the previous chapter we actually used the* ***Reflexivity principle*** *which led us to the invention of the object oriented programming. We can say that the* ***Reflexivity principle*** *is one of the fundamental concepts in programming and mathematics. Bellow is described its appearance in programming.*

Remark 7.1.2 *A good, World Wide Web available introduction and reference to software patterns is [6].*

A reader not familiar with OOP might refer to Appendix A, where are given some of the basic OO terms.

[1] Especially, if he or she is doing an object-oriented design of a large scale air pollution model.

7.2 Reflexivity principle

We will consider the **Reflexivity principle** as a pattern in the template of AG Communication Systems.[2] The discussion throughout the section motivates the order and the names of the topics of this template.

Let us have the following context:

7.2.0.1 Context

We have an algorithm based on some invariant of the data it operates on.

That context might arrive when we reuse code written in non-OO languages. Let us, say, have code like that

```
for i:=1 to n          AddOne(double x)
 AddOne(x_i)             return x+1
end                    end
```

What happens when the invariant `AddOne()` is based on, brakes for some new, unforeseen types of x_i? If, say, x_5 is a string, or x_6 is a vector. We can state that we have the following problem:

7.2.0.2 Problem

How to make the algorithm invariant, although the invariants (the properties) of the data it operates on, are changed.

Whatever the solution is, it should take into account the following desires we are going to pursue with any solution, i.e. the following forces:

7.2.0.3 Forces

1. We want the algorithm to be stable (non-changeable).
2. We want the algorithm to be applied to some broader or more specialized types of data.

The solution is to transform the code in such a way, that the invariant which is subject to breaking, is partitioned among the data.

7.2.0.4 Solution

If the data are not classes, define classes for them. All the operations, the algorithm do over the data, should be moved in (added to) the data's class as members.

The application of the solution to the code given above produces

```
for i:=1 to n
 x_i(AddOne)
end
```

In the resulting context, after applying the Solution above, the data can be sub-typed. If the algorithm, the code we want to be stable, and the data, the code which is subject to change, are placed within the same class, that leads to the **Template Method** design pattern (see [20, page 325]).

If the code we want to be stable, and the code which is subject to change, are independent enough to be in different classes, that leads to the **Strategy** design pattern (see [20, page 315]).

Applying **Reflexivity principle** increases the coupling between the classes. For example, if the initial algorithm is a class member and uses the context of its class, the algorithm's operations moved to the data classes will need to know the algorithm's class.

[2] The same format is used by Jim Coplien to describe C++ idioms in [15].

◊**idiom** ... 3: a style or form of artistic expression that is characteristic of an individual, a period or movement, or a medium or instrument <the modern jazz idiom>;◊

7.2.0.5 Rationale

This is a general approach in the object-oriented programming, and it is: the objects do the algorithm. A mathematical interpretation of the solution can be written as

$$f(x_i) = x_i^{**}(f),\ x_i \in I.$$

If the algorithm computes $f(x_i)$ over $i \in I$, where I is some index set, the algorithm can compute instead $x_i^{**}(f)$[3], provided f is a member of the type x.

We may claim that the **Reflexivity principle** is a pattern because it fulfills all topics (the answer is positive to all of them) of the pattern checklist bellow by Doug Lea [31]. The questions marked with $\surd$ are answered positively; the questions marked with $\Diamond$ are either not addressed or are not addressed directly in the pattern description.

A Pattern...

- Describes a single kind of problem. $\surd$
- Describes the context in which the problem occurs. $\surd$
- Describes the solution as a constructable software entity. $\surd$
- Describes design steps or rules for constructing the solution. $\surd$
- Describes the forces leading to the solution. $\surd$
- Describes evidence that the solution optimally resolves forces. $\Diamond$
  We don't but it should be clear.
- Describes details that are allowed to vary, and those that are not. $\Diamond$
  See the comments after the Solution section.
- Describes at least one actual instance of use. $\Diamond$
  We haven't, but clearly there are many.
- Describes evidence of generality across different instances. $\Diamond$
- Describes or refers to variants and subpatterns. $\surd$
- Describes or refers to other patterns that it relies upon. $\surd$
- Describes or refers to other patterns that rely upon this pattern. $\surd$
  This pattern is fundamental.
- Relates to other patterns with similar contexts, problems, or solutions. $\Diamond$
  This pattern is fundamental. In mathematics, we can consider, for example, Lebesque integration, wavelets, and finite elements as applications of the **Reflexivity principle**. See the Rationale section above.

7.3 Inventing the two basic design patterns: Template Method and Strategy

We will reach the two basic design patterns **Template Method** and **Strategy** in the following way.[4]

Let us have some code that does a computation. This computation will be coded differently by different people, but although its elements are approached differently they will correspond to each other in their role in the computation. We will refer to these computational elements and their connections as

[3] Written in OO notation as $x_i.f$.

[4] It is, actually, a scenario of contemplations and encountered problems.

Computational Context (CC) or just context. Our goal is to make OO design for this context that makes the code stable when reused.

To stabilize the context we should identify its unchangeable and its changeable parts. We call changeable those computational parts that become different when are changed the data structures used in the computation, or when are changed the assumptions how the computation should be done. The unchangeable part(s) should call the changeable parts. These references change the state of the computation, so we will call them operations. (The calls are operations on some data used in the context.)

If all operations are changed (more or less) together it is suitable to have the following structure: a class for the unchangeable, invariant part of CC, and descendants of this class (heirs) that define their own operations that complete the extracted unchangeable code to a desirable variant for the CC.

If for different cases the behavior of different subsets of the operations are changed, and especially when the behavior of some of the operations should be changed "on flight", instead of the structure above we should have a class, or module, where the stabilizing context should be placed, and different class hierarchies for the subsets of operations that change together. For example, if we have three operations with independent behavior we should have three different class hierarchies for each of them. If the behavior of two of the operations must change simultaneously, there is no sense to have such fine grained structure (it is also prone to inconsistencies of operation combinations), we can have just two hierarchies.

Have we stabilized our context? Well we have extracted its stable, invariant part, and we provided a structure where are confined the changes for a new desired behavior of the CC.

The structure described first leads to the design pattern called **Template Method**. The second, to **Strategy**.

◊How do we cope with the changeable!? We separate its changeable part from its unchangeable part. This is an application of the divide and conquer principle. Since the changeable and its relations to the unchangeable are so diverse we have so many design patterns and books about OO programming.◊

7.4 Normal forms for design patterns

The will to apply the theory of normal forms to the patterns for micro-software design known as design patterns, can be reasoned in several (overlapping) ways.

First, patterns by definition are reoccuring configurations or rules, which have the property that they resolve the forces in their contexts (see the introduction of Chapter 5). The identification, the discovering of a pattern starts with its verbalization, and if it feels right, further evidences of its invariantness are tried to be found.The pattern definition, as it can be seen from the pattern checklist above, ensures the rights of the pattern existence with various topics, like evidences of its use and its connection to other patterns. In general, finding a pattern relys on somebody's experience and ability to discuss it within a specific domain like architecture, software building, or music composition.We will feel comfortable with a software system if it is flexible and extendable, and as it was concluded in the previous section, we provide these qualities to a software system by separation of the unchangeable from the changeable, and addressing the connections between them. Ultimately, we will be addressing the connections and the combinations of pieces of code, and this can be put into some mathematical context. So, it is natural to introduce Relational Algebra and the theory of Normal Forms for the combinations of these pieces of code.

Second, when we develop or add new features to a software system we want:

- to preserve its consistency – otherwise it will not meet the requirements,
- to be domino effect free – the addition of a new feature should not invoke change of modules out of its code space.

In databases we are interested in the same qualities: consistency and minimum number of updates. So, it is natural to look for analogous approaches to achieve them. The primal approach in databases is to minimize the functional dependencies, which is also used by the theory of normal forms that guides the building of data bases with designs that are not prestructured when new data entities are included.

To interpret a program from a data base point of view, we should specify what is considered to be an atomic datum in it, and particularly, when two datums differ. It is natural the code of a procedure to be considered as an atomic data. We have two view points of distinguishing the codes of two pieces of code:

- we will say that two pieces of code are equivalent if they only differ by a specified small number of signatures in them,
- we will say that two pieces of code are equivalent if there are no differences between them.

We will need the first viewpoint when we consider the redundancy in the codes written within the modular programming paradigm: when this codes are redesigned within OOP we will use the second viewpoint.

The rigidness in the programs come from one-to-one relationships between the codes. (What else!) This can be seen from the following example. When a function calls another one, its code is completed with the callee's code.

```
func1(...)          func2(...)
 operators;          operators;
 call func2(...);   end
 operators;
end
```

It is clear that this introduces one-to-one connection between the codes of the caller and the callee. From another hand, when a function calls a method of an object, the function code can be potentially completed with different codes via polymorphism.We will see how design patterns achieve flexibility using this.

The interpretations we will make are from code reuse / code redundancy point of view. Clearly, this is a simplification, but it is one of the most important aspects in programming, and gives the ability to apply the normalization theory. Normalization theory is applied to large databases, so similarly to make the interpretation in question we will consider large programs in a large time span, where different combinations of different pieces of code are needed.

The relational database interpretations will be presented with both relational tables and Entity Relationship Diagrams (ERD). ERD are not so expressive as the relational tables, but the judgment whether an ERD is in a normal form is easier.

If database design is in a normal form, this does not mean that it is not vulnerable to changes: the normalization theory deals with the data on the semantic level, where different kind of misconceptions can occur. Nevertheless, third and higher normal forms are good feature for an RDB design to have. Design patterns, from another hand, are considered to have misconception robustness: even if they are applied with some misconception of the addressed problematic area, the pattern language should provide a pattern, or patterns which resolve the right forces, and to which the applied one should mutate; (see for example Section 5.6.6).

The interpretation viewpoint should become clear from the next subsection about the design pattern **Template Method**. The interpretations of the other patterns are described assuming that the next subsection is read.

7.4.1 On Template Method

The design pattern **Template Method** is explained in full detail in [20]. Here we will make a rationalization of it, that should explain how, why, when it is a great approach.

Template Method's structure is shown on Figure 7.1.

Clearly in this DP the TM interface is separated to two parts:

- `Alg()`;
- `Op1()`, `Op2()`.

The clients of the TM's hierarchy (TM's hierarchy is a concrete application of **Template Method**) refer just to TM:

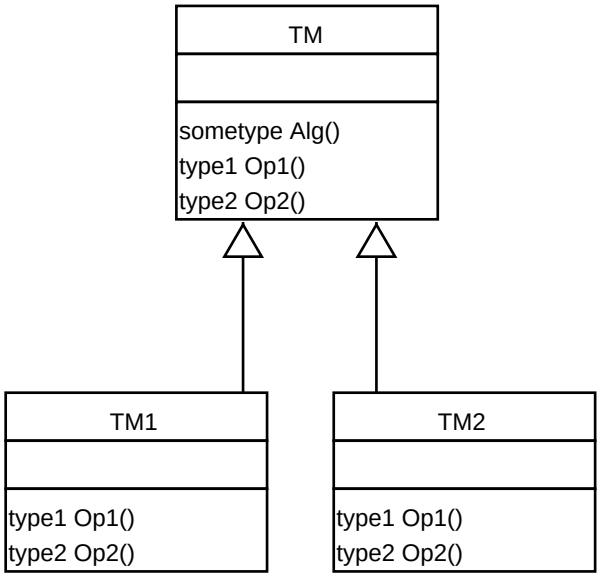


Figure 7.1: The design pattern **Template Method**.

Personal ID	Name	Town	County
19700417-3245	Bodil Jensen	Roskilde	Roskilde
19650823-3434	Marie Bertz	Lyngby	Lyngby-Taarbek
19801212-1875	Rasmus Petersen	Noerebro	Koebenhavn N
19450323-5676	Peter Plougmann	Lyngby	Lyngby-Taarbek
...	...	...	...

Table 7.1: Person-Town-County data base

```
TM objTM;
...
objTM.Alg();
```

The re-users of the TM's hierarchy provide their own code for `Op1()` and `Op2()`. They reuse `Alg()`'s code, so they should not know the details of it, although they know what it is doing.

As we can see, this is a clear application of the principle divide and conquer. The invariant code is separated from the changeable code. In this way we can confine the changes with stable design in the cases when the pattern is applicable.

To understand why and how the pattern works we can try to use the notion of functional dependence from relational data bases. First let us consider an example of a problematic database and how it is cured.

Our example data base is very simple, with just one table. The table contains the names of the Danish citizens, their personal IDs, the town and the county they are living; see Table 7.1.

Now, if for some reason the counties are changed – it can be their boundaries, or their number, or simply a different set of them is made – then we should change each row in the table where the (Town, County) couple has become inconsistent. If the total number of citizens living in a town moved to another county is 10000 we should do 10000 updates. This is too much since the changed information is much less: probably the total number of all counties in Denmark is 50-60.

The problem with this data base is that it has a transitive relationship in the table Person-Town-County. The person's name (or more precisely ID) defines the town where he or she lives, and the town defines the county it is assigned to. Hence the person's ID defines the county. This is a transitive relationship over the relation "defines"; see Figure 7.2.

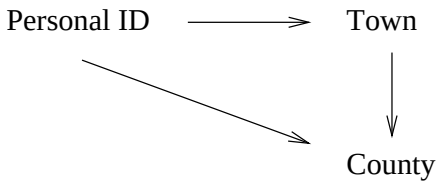


Figure 7.2: Transitive relationship between Person ID, Town and County.

Personal ID	Name	Town
19700417-3245	Bodil Jensen	Roskilde
19650823-3434	Marie Bertz	Lyngby
19801212-1875	Rasmus Petersen	Noerebro
19450323-5676	Peter Plougmann	Lyngby
...	...	...

Table 7.2: The Person-Town table

The way to cure this situation is to divide the table into two: Person-Town and Town-County. This is shown on Figures 7.2 and 7.3. Now, when the counties are reshaped, we have to do just the minimum necessary updates in order to have a consistent database.

The benefits from the **Template Method** DP are similar from the algorithm reuse point of view. To see this, we should consider how a concrete application of the pattern is used in a large program, and compare this program with one that doesn't use **Template Method** to have the same functionality. We will describe them both.

Let us now imagine that we have some large program, developed within the modular programming paradigm. The program has large number of modules (computational contexts) that need data transformation provided by some function `Alg`. The function `Alg` calls two other functions: `Op1` and `Op2`. Let us assume that at some point some of the modules need the `Alg`'s data transformation on slightly different data. To do this `Alg` needs different codes for the functions `Op1` and `Op2`, but otherwise `Alg()`'s code does not change. Some modules still need the transformation for the first data format. How should we provide this new functionality for the program?

Of course the answer to this question depends upon the programs and the data structures involved. Though the "universal" solution is to write another function `AlgNew` that copies the code of `Alg` and calls the functions `Op1New` and `Op2New` instead of `Op1` and `Op2`. `Op1New` and `Op2New` handle the new data format. If we want another new data format for `Op1` and `Op2` we should make another copy of `Alg`.

The reason for this redundant copies of `Alg` is that we have many-to-one relationship between the code of the modules and the `Alg`'s signature, and we have one-to-one relationship between `Alg`'s signatures and the signatures of `Op1` and `Op2`. This is the same like having the table shown on Table 7.4. Because in statically typed languages there is one-to-one correspondence between the function's signature and its code we have the Table 7.5.

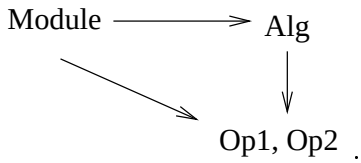
Table 7.5 is like the bad database table Person-Town-County (Table 7.1), since we have in it the transitive relationship: Module →Alg→ (Op1, Op2),

Town	County
Roskilde	Roskilde
Lyngby	Lyngby-Taarbek
Noerebro	Koebenhavn N
Lyngby	Lyngby-Taarbaek
...	...

Table 7.3: Town-County table

Caller's code	Callee's signature in the Caller	First signature in the Callee	Second signature in the Callee
code of module1	Alg signature	Op1 signature	Op2 signature
code of module2	Alg signature	Op1 signature	Op2 signature
code of module3	AlgNew signature	Op1New signature	Op2New signature
code of module4	AlgNew signature	Op1New signature	Op2New signature
...	...	...	...

Table 7.4: Signatures table for the modular programming program.



Now let us consider the variant of the program in which its modules call the `Alg` function placed within a class hierarchy given by the **Template Method**. Because of the inheritance, the relationship between `Alg`'s code and the codes of `Op1` and `Op2` is one-to-many. This is like dividing the code table, Table 7.5, to two tables: one for the relation of the modules with the `Alg`'s code via `Alg`'s signature, and the other for the relation of `Alg`'s code with the codes for `Op1` and `Op2`, via `Op1` and `Op2` signatures; see Tables 7.6 and 7.7. In Table `7.7` `Alg`'s code is on each row, that is why in GoF's [20] is said that in **Template Method** the code of `Alg` is factored out. Note, that on Table 7.7 the operations used in `Alg` have the same signatures, it is the code behind these signatures that differs.

The modules still should specify what combination of `Alg` with `Op1` and `Op2` codes they want to use, but the code of `Alg` is not duplicated. **Template Method** provides flexibility for the combinations of `Alg` with `Op1` and `Op2`. If we want flexibility on what combinations the modules use, we should use creational design patterns.

In a program written with non-OO languages, the signature determines the code behind it, though it is not like that in OO-languages: different codes placed in different classes can correspond to the same signature. Nevertheless, within a class there is just one code that corresponds to a given signature. So it is relevant to rewrite Tables 7.6 and 7.7 using class IDs instead of signatures as it is shown on Tables 7.8 and 7.9. We can also extend the notion of signature to include the ID of the class to which is attached the code of the signature, and call it **extended signature**. The class ID can be the class name since it identifies the class uniquely within a program.

The ERD diagram corresponding to the RDB interpretation given by Table 7.8 and Table 7.9, is shown on Figure 7.3.

Remark 7.4.1 *In non OO languages like C and Pascal we can use pointers to functions as parameters to tell `Alg` what `Op1` and `Op2` it should call. Though this is, in a way, implementing the dynamic binding.*

7.4.2 On Strategy

The **Strategy** DP is described in details in [20]. The structure of this pattern is shown on Figure 7.4.

Here we will go directly to the RDB interpretation of this pattern from the code reuse point of view. The purpose of the pattern is to provide one-to-many relationship between the context containing the signature of an operation and the code behind this signature – clear manifestation of polymorphic behavior; see Table 7.10. If the non-polymorphic approach is used, we will have the table Table 7.11: the code of the callers is not so stable now, since the code should be changed when we want the callers to use different concrete strategy.

Like in the previous section, Table 7.10 can be rewritten with class IDs as it shown on Table . In the following sections we will use just "class ID" tables.

The ERD diagram corresponding to the RDB interpretation given by Table 7.10 and Table 7.12, is shown on Figure 7.5.

Caller's code	Callee's signature in the Caller	Callee code	First signature in the Callee	Callee1 code	Second signature in the Callee	Callee2 code
code of module1	Alg signature	Alg code	Op1 signature	Op1 code	Op2 signature	Op2 code
code of module2	Alg signature	Alg code	Op1 signature	Op1 code	Op2 signature	Op2 code
code of module3	AlgNew signature	Alg code	Op1New signature	Op1New code	Op2New signature	Op2New code
code of module4	AlgNew signature	Alg code	Op1New signature	Op1New code	Op2New signature	Op2New code
...	...	...	...	...	...	...

Table 7.5: Code table for the modular programming program.

Caller's code	Callee's signature in the Caller
code of module1	Alg signature
code of module2	Alg signature
code of module3	Alg signature
code of module4	Alg signature
...	...

Table 7.6: Module→Alg table

Caller code	First signature in the Caller	Callee1 code	Second signature in the Caller	Callee2 code
Alg code	Op1 signature	Op1 code	Op2 signature	Op2 code
Alg code	Op1 signature	Op1 code	Op2 signature	Op2 code
Alg code	Op1 signature	Op1New code	Op2 signature	Op2New code
Alg code	Op1 signature	Op1New code	Op2 signature	Op2New code
...	...	...	...	...

Table 7.7: Alg→(Op1,Op2) table. Rows 2 and 4 should be deleted since they repeat 1 and 3 respectively.

Caller's code	Class ID `Alg` is called from
code of module1	TM
code of module2	TM
code of module3	TM
code of module4	TM
...	...

Table 7.8: Module→Alg with class IDs

Caller (Alg) class ID	`Op1` class ID	`Op1` code	`Op2` class ID	`Op2` code
TM	TM1	Op1 code	TM1	Op2 code
TM	TMNew	Op1New code	TMNew	Op2New code
...	...	...	...	...

Table 7.9: Alg→(Op1,Op2) table with class IDs. Here we have deleted the rows 2 and 3; see the legend of Table 7.7.

Caller's code	Callee's signature in the Caller	Callee's code
code of module1	Strategy (S)	ConcreteStrategy1 (S1) code
code of module2	Strategy (S)	ConcreteStrategy2 (S2) code
code of module3	Strategy (S)	ConcreteStrategy1 (S1) code
code of module4	Strategy (S)	ConcreteStrategy3 (S3) code
...	...	...

Table 7.10: Polymorphic Module→Strategy table

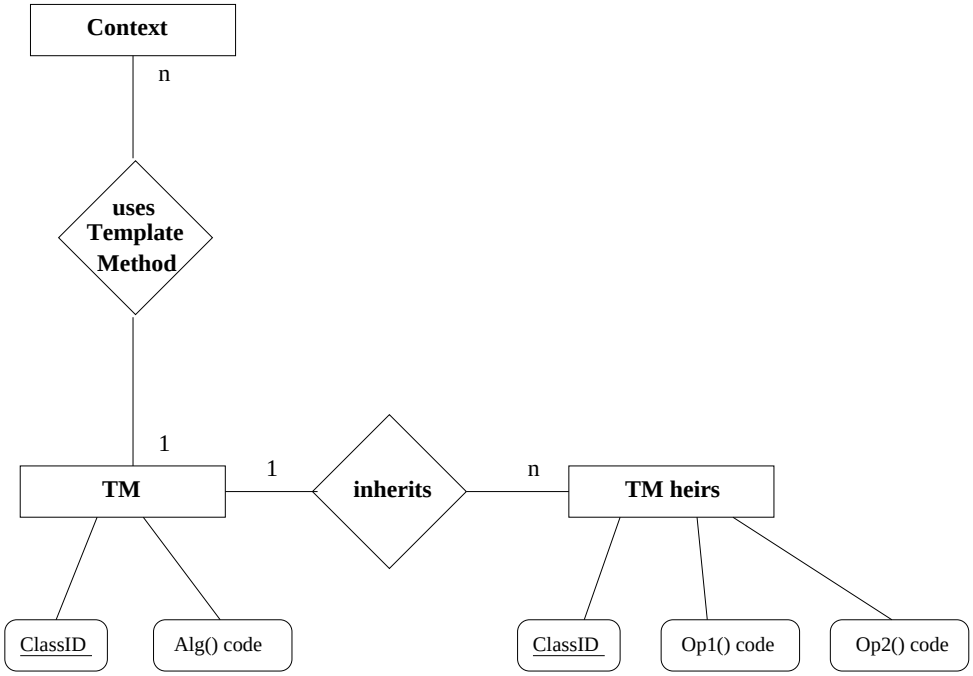


Figure 7.3: ERD interpretation of **Template Method**.

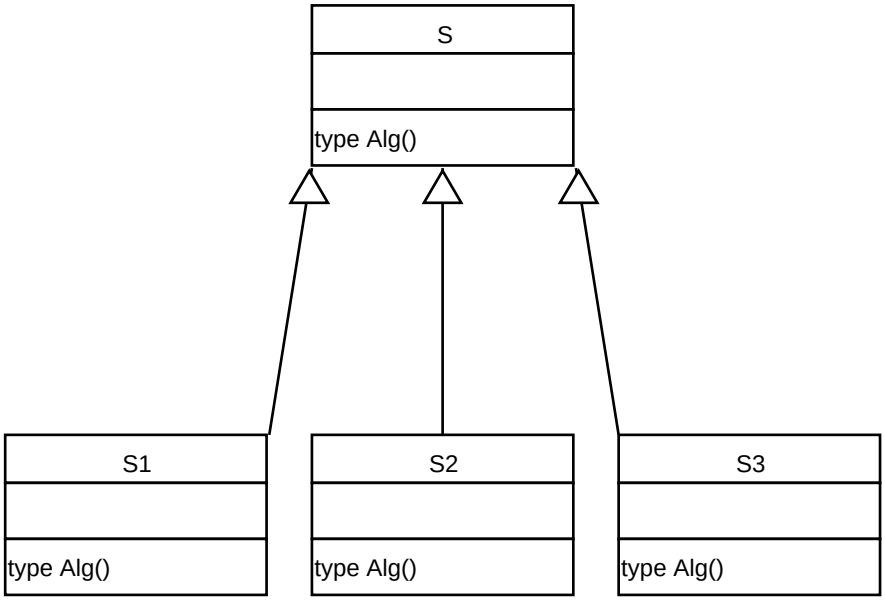


Figure 7.4: The design pattern **Strategy**.

Caller's code	Callee's signature in the Caller	Callee's code
code of module1	ConcreteStrategy1 (S1)	ConcreteStrategy1 (S1) code
code of module2	ConcreteStrategy2 (S2)	ConcreteStrategy2 (S2) code
code of module1	ConcreteStrategy1 (S1)	ConcreteStrategy1 (S1) code
code of module3	ConcreteStrategy2 (S2)	ConcreteStrategy2 (S2) code
...	...	...

Table 7.11: Non-polymorphic Module→Strategy table

Caller's code	Callee's class ID in the Caller	Class ID of the callee's code
code of module1	Strategy (S)	ConcreteStrategy1 (S1)
code of module2	Strategy (S)	ConcreteStrategy2 (S2)
code of module3	Strategy (S)	ConcreteStrategy1 (S1)
code of module4	Strategy (S)	ConcreteStrategy2 (S2)
...	...	...

Table 7.12: Class ID Module→Strategy table

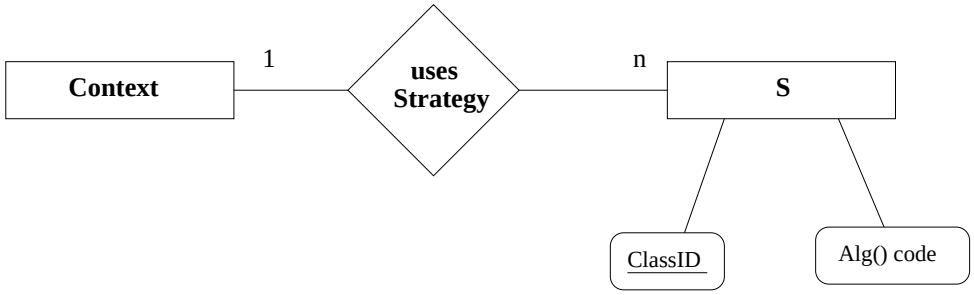


Figure 7.5: ERD interpretation of **Strategy.**

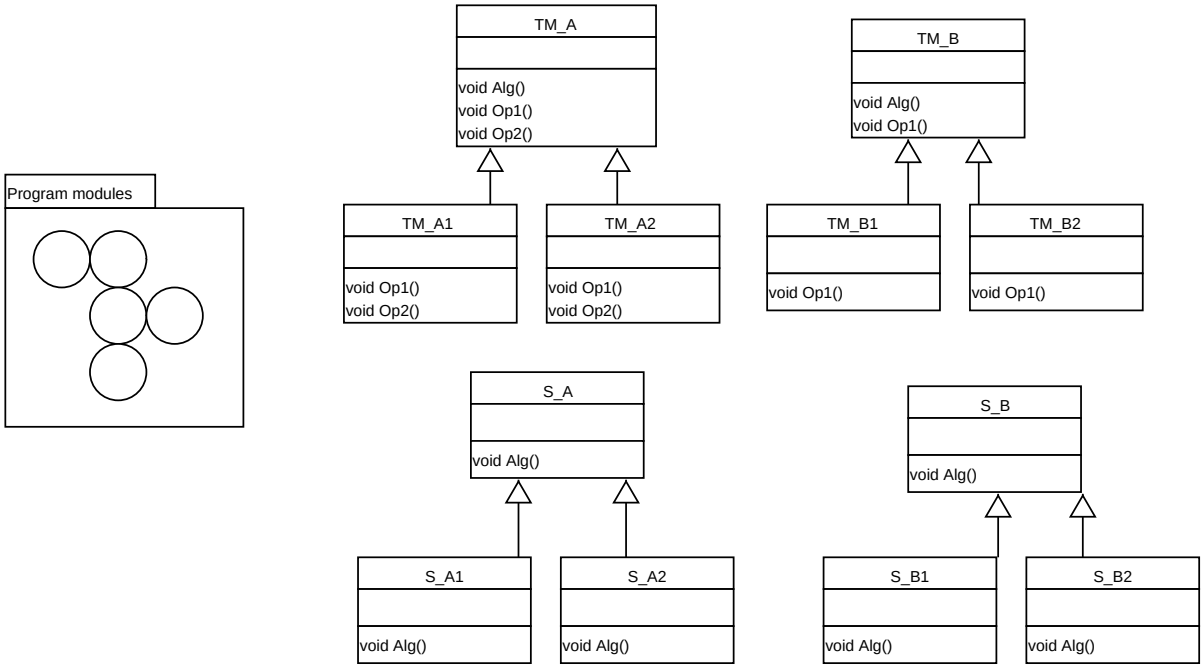


Figure 7.6: Class hierarchies from **Template Method** and **Strategy**

7.4.3 On Abstract Factory

Following our line of rationalization let us suppose that we have created a number of hierarchies based on the design patterns **Template Method** and **Strategy**. We want to configure the contexts using them with particular (meaningful, consistent) combinations of them; see Figure 7.6. More important, in order to benefit from applying these patterns, we want only their abstract classes to be presented in the program modules: having explicit references to concrete classes of the DP hierarchies will compromise the non-transitivity achieved by applying them in the rest of the code (the modules).

The solution given by **Abstract Factory** DP is to introduce another level of indirection, and it is much in the spirit of the **Reflexivity principle**: **Abstract Factory** is a class hierarchy that is a dual view of the **Template Method** and **Strategy** hierarchies. The structure of the **Abstract Factory** hierarchy is given from the consistent (desirable) combinations of concrete **Template Method** and **Strategy** classes; see Figure 7.7.

In order to make the RDB interpretation of **Abstract Factory**, let us examine more closely the way it makes the context code stable. The context uses objects from **Strategy** and/or **Template Method** hierarchies. If the context is not responsible for the creation of these objects, the only references to them are via the client interfaces of the respective DP, i.e. the code of the context contains just names of the objects and the signatures of the client operations of **Strategy** or **Template Method**, (`Alg()` in both cases). The context code might have the type declaration of the objects. If the declarations are made with the interface classes of the DP's, as they should be, the context code is clean of any reference to a specific class: it uses just the abstract interfaces. [5]

Now if the context is responsible for the objects creation, the only specific code added to it are the class creators. This introduces transitive relationship between the context code and the class hierarchies. This can be seen from the example

```
void Context(...) {
  Strategy obj;
  ...
```

[5] And yet the context provides the desired flexibility, because is polymorphic based.

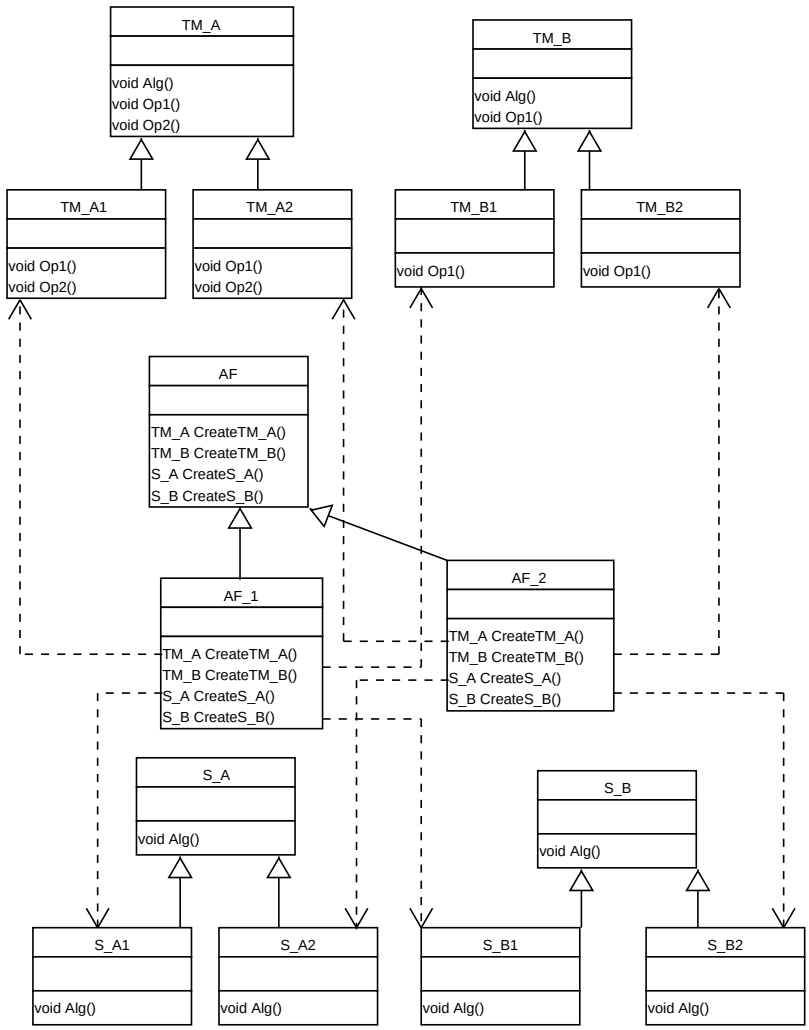


Figure 7.7: **Abstract Factory**

```
    obj = new ConcreteStrategyA(...);
    ...
    obj.Alg(...);
    ...
}
```

Clearly the line

```
obj = new ConcreteStrategyA(...);
```

induces one-to-one relationship between the context and the concrete strategy `ConcreteStrategyA`, and therefore we cannot take advantage of the polymorphic behavior provided by **Strategy**.

When we discussed the RDB interpretation of the design patterns **Strategy** and **Template Method**, we considered just context code that uses the signatures of some class methods – the context can use objects that are global in some scope (or class), or the context can be a procedure parametrized with the class types. This suited our RDB interpretation, since in data bases the data records of the entities are constructed of the data base system not by the entities themself. Though in the case we consider now, the entity Context creates its own records and we should widen the scope of code we model with the RDB paradigm. In a way this means that the RDB model will become more detailed.

So the context uses several **Strategies** and **Template Methods**, and we want to configure it with different combinations of them, without changing the creational part of the context code a lot and often (as we would do if we leave the one-to-one correspondence indicated above). It is natural "to factor out" this creational part in some relevant class structure. In this factorization is kept the information with what combination of algorithms the context should be completed with – the RDB interpretation with an ERD diagram is shown on Figure 7.8. This particular factorization gives us the possibility to create just the feasible or meaningful combinations.

From the Figure 7.8 we cannot see that the Context knows about S_A and T_A, though in its code we have lines like

```
Strategy obj;
obj.Alg(...)
```

We can see these code lines as parts of the mechanism that constructs the relations of the datums we consider (programming codes).

Clearly the RDB interpretation of the **Abstract Factory** is in 3NF.

7.4.4 On Decorator

Strategy provides the Context with different algorithms, and **Template Method** provides the Context with one algorithm with changing parts. The **Decorator** DP provides the context with one algorithm which is wrapped with other algorithms, i.e. over the core algorithm are added responsibilities.

The **Decorator** is explained in full detail in [20]. Its structure is shown on Figure 7.9.

The RDB interpretation of the pattern is again from the point of view what code the Context is completed with at run time. The ERD diagram is shown on Figure 7.10. The diagram says that a Component given to the Context has a relation from the table WrapOver (Table 7.15). That leads to consecutive cascade additions of code. The WrapOver relation has the constrain that all chains of WrapOver ID's should end up with a record for which ClassID = ConcreteComponent, and which does not "wrap" anything. (For example, follow in Table 7.15 the chain that starts from the record that has WrapOverID=wrov3.)

Note that the WrapOver part of the ERD diagram contains just ClassID's, no code. It is also like that implementationly: **Decorator** achieves the described cascade code wrapping via the dynamic binding provided by the OO language implementation; (just) the code is provided by the user. Clearly, the ERD diagram is in 3NF.

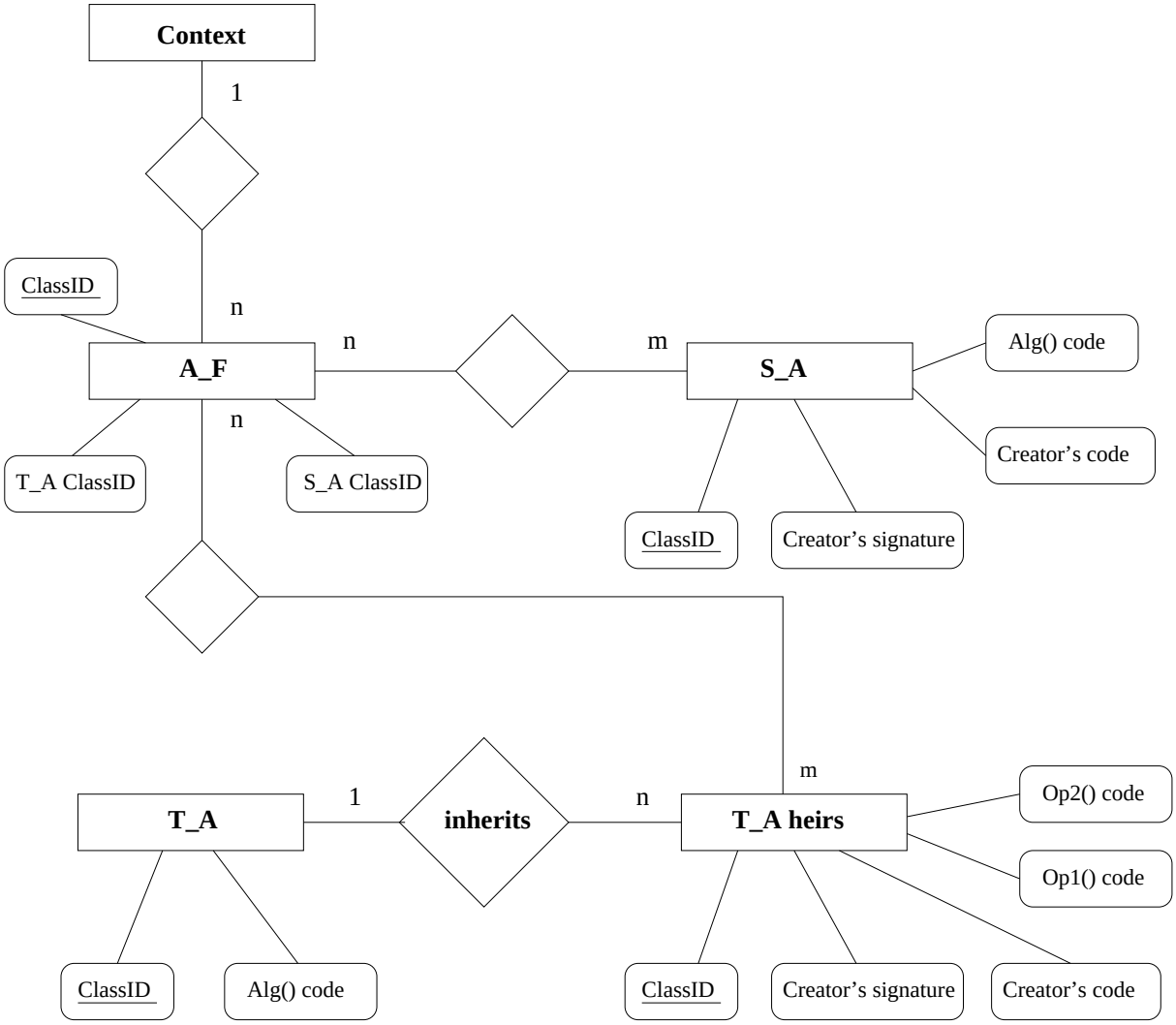


Figure 7.8: Context Abstract Factory Strategy Template Method

Component	
ClassID	Code for Operation()
Component	core code
ConcreteDecoratorA	add. func. code
ConcreteDecoratorB	add. func. code
...	...

Table 7.13: Component relation

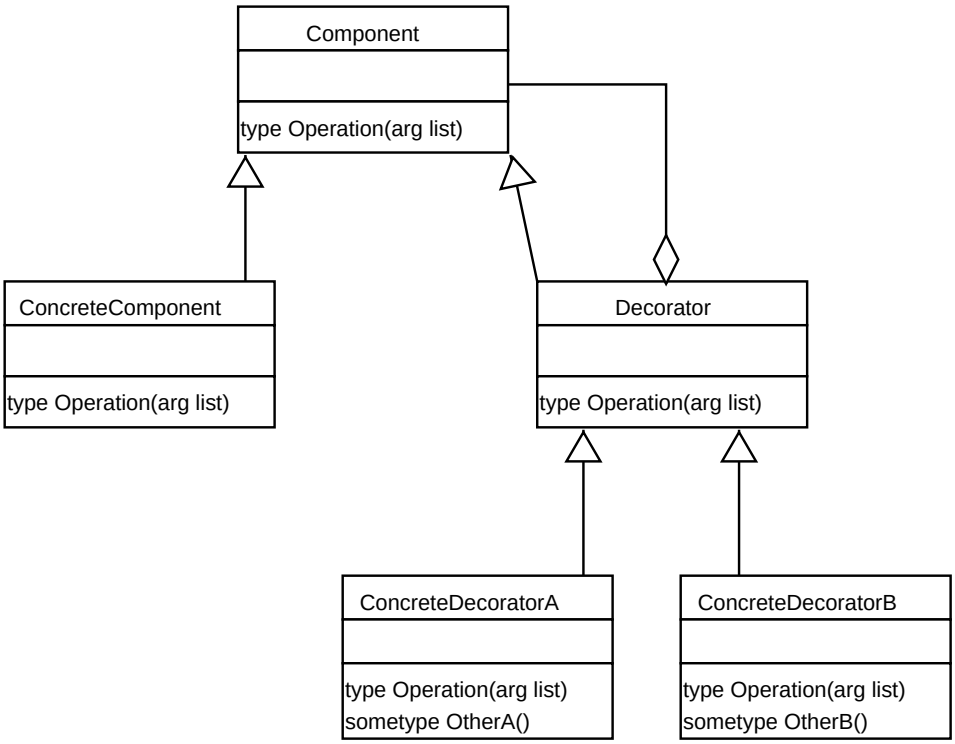


Figure 7.9: The structure of the design pattern **Decorator**

OtherMethods		
ClassID	Signature	Code
ConcreteComponent	another1(int, int,...)	code
ConcreteDecoratorA	OtherA(double, ...)	code
ConcreteDecoratorB	OtherB(list, ...)	code
ConcreteDecoratorB	OtherB1(int, ...)	code
...	...	...

Table 7.14: OtherMethods relation

WrapOver		
WrapOverID	ClassID	Next WrapsOverID
wrov0	NULL	NULL
wrov1	ConcreteComponent	NULL
wrov2	ConcreteDecoratorA	wrov1
wrov3	ConcreteDecoratorB	wrov2
wrov4	ConcreteDecoratorA	wrov5
wrov5	ConcreteComponent	NULL
...	...	...

Table 7.15: WrapOver relation

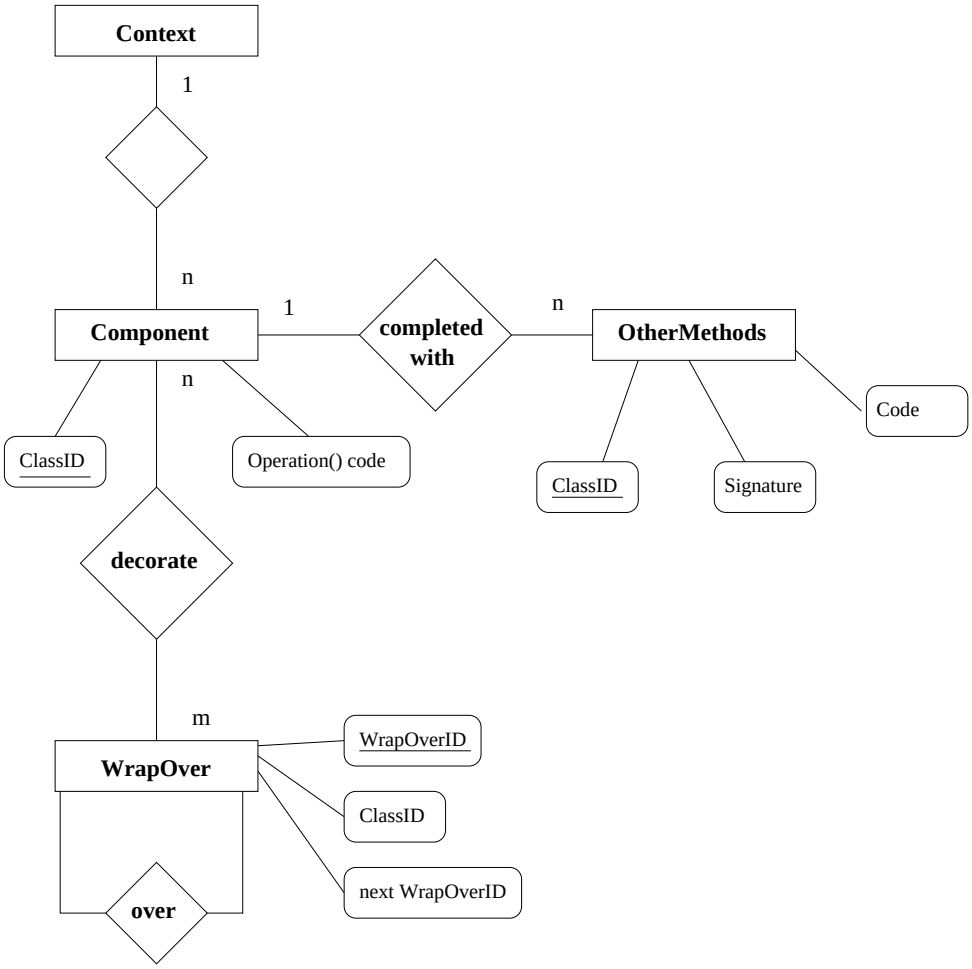


Figure 7.10: ERD for the design pattern **Decorator**

Chapter 8

Future Extensions

All I need to be
Cause I went away
But what I really left
Left behind was me

– OFFSPRING, *"Million Miles Away"*,
Conspiracy Of One, 2000

There are a lot of possible improvements and extensions of the work carried on. Some of them are discussed bellow. In this chapter I use again both the "we" and the "I" forms: see the remark at the end of the introduction to Chapter 5.

8.1 Higher order finite elements

The advection-diffusion framework is amenable for higher order GFEM, though their application to the air pollution problem has to be justified. As an example we will consider the application of a quadratic basis functions GFEM with triangular grid. As it can be seen from Figure 8.1, this method introduces additional nodes (the red nodes on that figure). If the initial grid is $n \times n$, the number of the additional nodes is $3(n-1)^2+2(n-1)$. So, the total number of nodes is approximately $4n \times n$, and the computational work will be approximately 4 times more. Nevertheless, if the wind is assumed to be constant within a triangle, as in OODEM, it is not clear what is the benefit of this additional work.

Clearly, the question whether is beneficial to apply quadratic GFEM to the air pollution problem can be answered with experiments. Since the wind field used now in DEM is on a 32×32 grid (each square is $150km \times 150km$), and is step-approximated on the 96×96 grid, there is a possibility the quadratic GFEM to give better results.

Unfortunately, we cannot make a theoretical analysis how the quadratic GFEM distorts the sinusoidal solutions of the advection equation (3.16) as we did in Chapter 3, since we have four different types of patches, and hence we cannot apply the space Fourier analysis. (Though, we can of course derive the semi-discrete equations, and try to apply to them, for example, time Fourier analysis.)

Finally, let us note that the additional mesh generator algorithms needed to produce the description of a quadratic grid are quite similar to those used to describe the non-conformal finite element grid.

8.2 Dynamic and adaptive mesh refinement

The ability of dynamic mesh refinement is an attractive feature to add to OODEM. This feature can be added in two ways: (i) by moving (part of) the developed *Mathematica* mesh generator to C++, (ii) by providing an interface to an external for OODEM mesh generator.

Let us specify further the first approach, (i). Let us first suppose that we want to use just one refined cell (see Figure 2.1 for visual aid). The refined cell is not statically placed but is moved within the grid

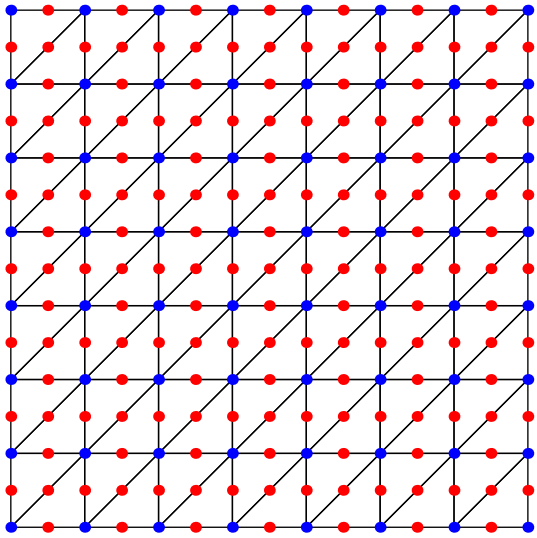


Figure 8.1: Grid and nodes for a quadratic GFEM on triangles. The red nodes are the "new" nodes, needed to define the quadratic basis functions. Just the blue nodes are needed when linear shape functions are used.

region during the simulation. The size of the cell and the depth of its refinement does not change. As it is explained in the *Mathematica* appendix "Mesh Generation for the Object-Oriented Danish Eulerian Model, Programmer's Guide", after the cell is refined to the desired level, it should be plugged in into the rest of the grid. Then the nodes and elements of the grid are sorted, the orbits are found, and the approximation rules are produced. So, we can make a description of the cell, that according to which cell elements will be adjacent in the sorted grid, and if the initial coarse grid is sorted itself, when we plug in the cell, with this description we will put its elements in the right places. The cell has the same number the orbits, and the geometry of their representative patches does not change: just the nodes the orbits consist of. One of the advantages of this semi-sorting is that the grid generation will be fast.

This approach can be used also when we have several cells, but in this case it might be better to have full grid sorter in C++: this will give better functionality with less programming effort. The grid generation though will be slower than the one-cell approach.

The use of an external for OODEM mesh generator can be done in two principal ways: (i) its grid descriptions are adapted (transformed) to comply with those used in OODEM, (ii) the classes in the GFEM layer are sub-classed (extended) to utilize the grids of the generator. Which of these two ways is more relevant depends of the chosen grid generator. We should point out, that the GFEM layer is greatly unfolded in the direction of locally uniform grids usage. So, if the we want to use other type of grids, it is better to take the second approach.

Finally, let us note that OODEM is amenable (ready for use) for simulations based on the static-regridding algorithm [40], if rectangular areas of regridding are used. (Hence, OODEM is amenable for one-way nesting too.)

8.3 Completing the design of OODEM

As it was said in the end of Chapter 5, OODEM is still in a transition period. To discuss its completion to a fully object-oriented system, let us first describe the preprogrammed C/C++ programs in the DEM layer, the dependency graph of which is shown on Figure 8.2.

The structure of the DEM layer is in one-to-one correspondence with the structure of the FORTRAN 77 production code of DEM.

The main program, prog35.C

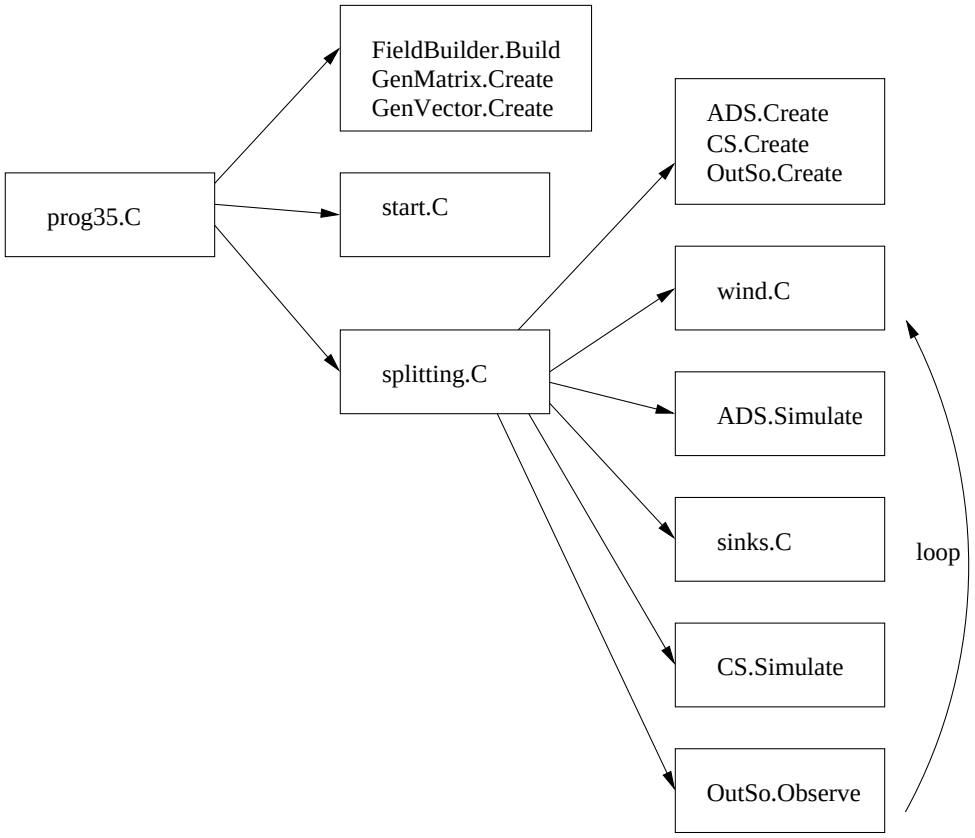


Figure 8.2: Dependency graph of the DEM layer

- creates, via FieldBuilder, objects that read and approximate the emission and meteorology data fields;
- initializes a lot of data structures directly calling the creators of the classes GenVector and GenMatrix;
- assigns values to several arrays;
- calls the function (subroutine) start.C, which does initial readings of the data files;
- calls the function (subroutine) splitting.C.

The function start.C

- reads the initial data fields and transforms them;
  - the actual assignment to the field arrays is made via an auxiliary array (named rhelp or something of the sort);
  - some of the fields read are splitted into two model fields, e.g. the NO_x data field is splitted into the model fields NO and NO_2;
- re-maps the values of the emission matrix, for convenience of the chemistry routines.

The function splitting.C

- creates the objects for advection and chemistry simulation from the GFEM layer and the Chemistry layer, and the observer object of type OutSo;
- reads the wind field with the function wind.C;
- does the advection simulation;
- reads the emission and meteorological fields with the function sinks.C;
- does the chemistry simulation;
- aggregates some chosen fields with the object of type OutSo, which uses its own simulation clock.

The function winds.C, using a kind of simulation clock, either

- reads two consecutive wind field snapshots from the data file,
- or approximates the wind field (or the operator A in (5.4)) .

The function sinks.C

- uses some sort of simulation clock to decide when to do the steps bellow;
- reads the data fields and transforms them; only meteorological data is read;
- modifies the emissions using the new meteorological data;

The simulation clock of these routines is implemented by several global variables (and they are changed in some messy way). So it is natural to make a class called SimulationClock and to embed it with the **Singleton** design pattern in OODEM. It is save each processor to have its own SimulationClock object.

The main conclusion from the description above is that we should enhance the functionality of the class *Field* (see Section 5.6.3): if every modeling field is not just an array, but a *Field* object, we can introduce inter-*Field* manipulation operations that will greatly reduce the code in the DEM layer. (Most of the functionality of this code will be covered by the *Field* hierarchy.) The new inter-*Field* operations are:

- field modifiers – functions that, e.g. scale and shift the values of a field;
- field assignments – given two fields A and B, the values of A are assigned through some modifier to B.

The current OODEM implementation has modifiers (C/C++ functions), references of which can be passed to the *Field* class, but this feature is not used. Instead of this function reference approach, that gives poor performance, it is better to use the "pointer-to-function as template parameter" approach described in [49].

Unfortunately, the interface of *Field* is already quite heavy, so we can either

- introduce a lighter, more abstract than *Field* class, and apply to it the already applied to *Field* **Decorator** DP,
- or, we can apply the **Adapter** DP to *Field* and the array classes GenVector and GenMatrix,
- or, simply go ahead with a heavier *Field*.

I am much in favor of a solution that combines the first two approaches, applying the **Adapter**'s object structure. We can apply this variant of **Adapter** by (i) making a super-class of *Field*, let us call it *SuperField*, and (ii) make a sub-class of *SuperField*, let us call it *ModelField,* that stores field data in a GenVector or GenMatrix object, has the inter-*Field* operations, and eventually delegate the operation of reading (and approximating) data to a *Field* object. The **Decorator** now decorates *SuperField* instead of *Field*. Clearly, if this solution is applied the class names can be changed to names that reflect closer their intent and functionality.

After the deposition fields are read in sinks.C, they are modified with the solution of the ODE's (2.8) in Section 2.4. This modification can be covered by the *Field* **Decorator** with the concrete decorators. Then a lot of the parameter initializations in prog35.C can be distributed among the identified decorators.

Other conclusions from the DEM layer code is that it is a good idea to provide dictionaries for the modeled pollutants. Every dictionary gives a number associated with a pollutant name, and vice versa. This will greatly simplify and make less error prone the field re-mapping manipulations. Clearly, introducing this indirection in a naive way will slow down the performance of the computationally heaviest DEM part – the chemistry. We can use either macroses or templates for making the dictionaries. We can also provide some script program for code manipulation that supports the dictionary functionality.

It is a good idea to use the library A++/P++ [39] for the arrays in DEM. The library is well characterized by the following quotation:

> A++ is a serial C++ array class, P++ is the parallel version of the exact same array class interface. A++/P++ can be characterized as a parallel FORTRAN 90 in C++; fundamentally, A++/P++ is simple.[39]

The OODEM classes GenVector and GenMatrix has the same intend. Blitz++ is another alternative of them that can be considered.

The reduction of the DEM layer code with the approaches sketched above, makes it much easier to identify the classes of its design. The classes can resemble the functional decomposition shown on Figure 8.2. We can have a Splitting class based on splitting.C that uses a Splitting builder class based on prog35.C. Splitting calls in its creator a routine corresponding to start.C. We should note that start.C and sinks.C should be greatly dissolved by the *Field* class, and they should mutate to some sort of **Composite**s.

8.3.1 Including new design patterns in OODEM

8.3.1.1 Including Observer

An application of the **Observer** DP should replace the result storage routines (also OutSo class). This will give greater flexibility for the choice what results to be stored and how. **Observer** can use the Data Handlers Layer. Also, it can be used for run time monitoring of the simulation.

8.3.1.2 Including Memento

The design pattern **Memento** can be used to back up the state of a simulation, and then to start it later from the point already reached. This could be very useful if a lot of scenarios have to be run, in order to find the optimal one. If we have run-time observers of a simulation, we may decide to interrupt it at some point according to the observers output, and start another one.

8.3.1.3 Including Visitor

Another pattern of applicational interest is **Visitor**: its double dispatching will provide the power to enlarge OODEM with new distinct, and unrelated functionalities, without changing any of its existing classes (this means also that no code will be added to them). With **Visitor** we can easily make consistency checks when new features of OODEM are developed. For this, **Visitor** can be used together with the patterns for logging diagnostic messages in [24].

Remark 8.3.1 *Implementations of the* SimulationClock *and the* **Observer** *DP were made, though they are not currently included in OODEM.*

8.4 Developing the chemistry framework

One of the most important extension is the building of the OODEM chemistry sub-framework. I have some working experience with the object-oriented Generic ODE Systems Solver (GODESS) [45] and with the DEM chemistry box model programs (I have rewritten and redesigned the later to the OO language Eiffel [36]). So, I would apply the **Essence** DP to the concrete component `Chemistry` in the applied **Decorator** DP, and for this new, conceptual class – let us call it *Chemistry* – I would provide the following class hierarchies:

- linear solvers classes (they can be based on PETSc);
- classes for performing the Newton-Raphson integration method;
- classes organized with **Template Method** and/or **Strategy** DP's that provide simple Runge-Kutta ODE methods;
- the **Abstract Factory** DP to provide the flexibility for the building of a concrete *Chemistry* solver.

It can be seen from [3], that the numerical analysts at NERI use a framework of FORTRAN 77 programs for designing and testing new ODE solvers tuned to the air pollution problem area. The suggested class hierarchy above should enforce/duplicate these programs, and should be able to include new FORTRAN solver routines or reaction mechanism routines.

We can use GODESS to find new, alternative ODE integration methods, though it should be kept in mind that the framework GODESS is designed to device new ODE methods and to compare them: the algorithms performance is not that important as in DEM. This approach implies that the DEM developed methods should provide the same statistics as GODESS. Alternatively, they can be pre-programmed to GODESS (what I did) which has the drawback that the pre-programmed code should be kept up to date. Third alternative is to incorporate the DEM implemention to GODESS, if GODESS is open for that. (If we provide GODESS with a routine that calculates the right hand of the ODE's system $\mathbf{y}' = f(\mathbf{y})$ we can use all of the methods included in it, which are more than 50.[1]

[1] Nevertheless, two years and a half ago I found GODESS difficult to use when I tried to apply it for the air pollution chemistry with changing photochemical coefficients. It was easy to use it with constant photochemical coefficients, though completely uninteresting (for my supervisors).

8.5 Monitoring with the ALICE memory snooper

The ALICE memory snooper [7] can be used to observe and change the variables (memory) of a running application. In this way, we can change the behavior of the running program or/and use the observers of the **Observer** DP. In general, using this tool fits the inclusion of **Memento** and **Observer**.

Some experiments were made with it on a SP machine, though the used C code is not included in OODEM.

Chapter 9

Conclusion

Conspiracy of one

– OFFSPRING, *"Conspiracy Of One"*,
Conspiracy Of One, 2000

The thesis has presented the mathematical engineering issues for the implementation of the Galerkin finite element method for the advection-diffusion submodel of the problem of air pollution simulation over a large region. The framework for the advection-diffusion submodel was conceptually described with problem specifications, framework patterns, and design patterns. This framework comprises the most influential parts – the GFEM and the Data Handlers layers – of the framework for large scale air pollution modeling OODEM. Tests and experiments carried on with it and OODEM was demonstrated. The used Galerkin finite element method was analyzed. Some of the fundamental design patterns were analyzed and rationalized. Both analyses was accompanied with mathematical and computer science theoretical material. The general design of OODEM was outlined.

The entire work done during the Ph.D. studies is summarized bellow.

9.1 Major contributions

9.1.1 Development: OODEM framework

We can say that the measure of success for the Ph.D. project, is how useful, and how easy to use, is the developed OODEM framework, and also, how quickly we can have new ideas developed with it. These usefulness, easiness to use, and versatility, would reveal themself in a longer term. Nevertheless, the employment of the object-oriented paradigm, and especially the employment of the design patterns, give us confidence that the framework has these properties. The easiness to use is tightly connected with the documentation: a framework without a documentation is not a framework. [1]

Figure 9.1 presents the current state of the OODEM development and documentation, and below are listed the framework features.

9.1.1.1 OODEM features

- Employs operator splitting for the different physical processes
- Genuine two-dimensional advection simulation
- Numerical zooming with static grids over several specified by the framework user regions
- MPI based; able to run on both shared and distributed memory machines
- Amenable for extension to three-dimensional simulations (1D×2D or 3D)

[1] So, good documentation is crucial. And if the documentation is bad it is better not to have it at all.

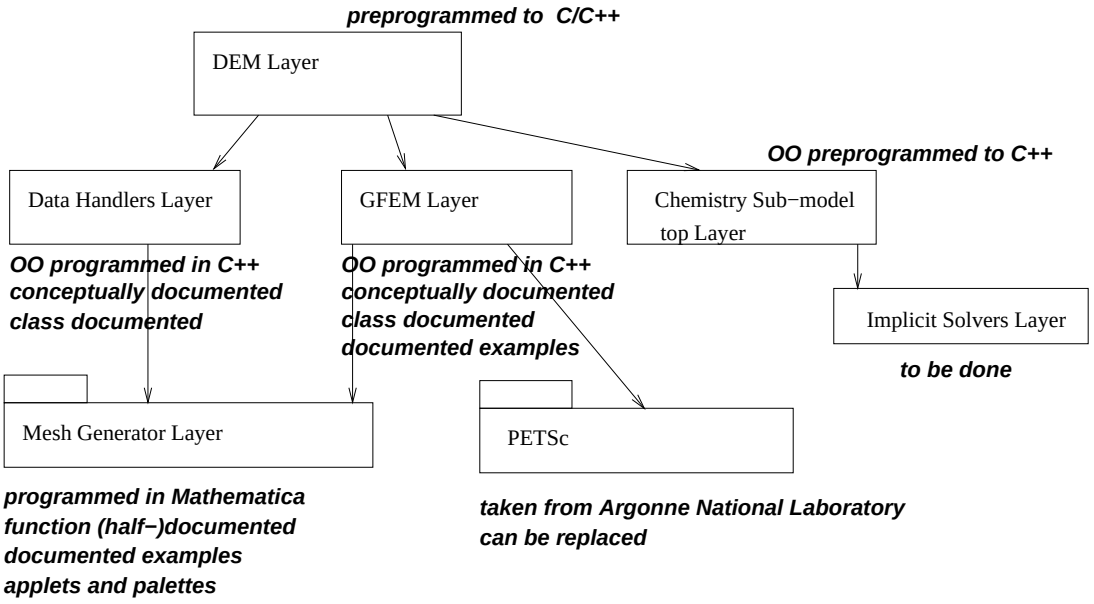


Figure 9.1: The developed OODEM framework. The meaning of "OO programmed" is "Programmed in the object-oriented paradigm using design patterns". The conceptual documentation is presented in Chapter 5.

- Amenable for non-conforming and high-order finite element methods
- Has a mesh generator for locally uniform grids with triangular and rectangular element geometry
- Framework's mesh generator is independent from the rest of the framework; hence, can be replaced with another one
- Framework's mesh generator has routines for visualizing grids, their interpretation and their place over a given geographical map
- Most of the framework layers are with object-oriented design using design patterns
- Flexible data feeding
- Conceptually documented, class documented, documented examples; the documentation uses UML; the documentation is web available
- Employs PETSc for the solution of large sparse linear systems (other library can be used)

9.1.2 Scientific: design patterns as normal forms

This is my favorite contribution. It was shown that the design patterns for building reusable software can be interpreted as prescriptions how to make software structures that are in normal form from relational database point of view. In the contexts where they are applicable, the design patterns provide software structures free of transitive relationships, which is a necessary condition for qualities like flexibility and extensibility.

The benefits of such an interpretation are three-fold.

First, although the design patterns had proven to be useful, they are hard to be believed, yet embraced, by someone who has no experience with object-oriented programming. By the interpretation, the programs built with design patterns will be in normal form, so their flexibility and extensibility would not be reasoned, discussed, or believed: their flexibility and extensibility will be proven. This was my initial intention: *to prove* that the design of OODEM has the desired qualities.

Second, and this is more important, the interpretation can be used to discover and question new design patterns. There is a great demand for domain specific design patterns and this interpretation might help to discover new ones. It should be possible in principle, that instead of learning (mastering) the problem area first and then choosing the relevant patterns, to identify first the places of transitive relations in the problem area and then address them with suitable patterns. Some of the identified in the later way patterns might be not discovered yet.

Third, it is interesting to judge whether a given set of design patterns is a complete set or not. The interpretation can be used to check a pattern language for its completeness, and to fill in the missing patterns in it.

9.1.3 Project management: paradigm shift in DEM

The most important impact of the presented work is the paradigm shift it makes in DEM, and in this way makes it more attractive for computer science students, who are led by desires to satisfy their creativity, not by a will to understand the puzzle of Nature.

In general, the object-orientation, the use of the commonly known design patterns, the number of available C++ source navigation tools, and the conceptual, class, and examples documentation, should make OODEM attractive to newcomers, who want to mess up with the modern programming techniques, and (quickly) achieve results in a non-trivial field of nature processes simulation.

To preserve the style of OODEM we can use the following rules: (i) the new code, or the new feature, should be either within the already applied patterns, or it should use design patterns for its structure; (ii) the new code should be at least class documented.

9.2 Minor contributions

9.2.1 Field visualization routines

A number of routines were developed for visualizing the input and output fields and their approximation in the simulation code. This is very useful for debugging. The output fields (the results) can be visualized with different contour, scatter, and density plots.

9.2.2 Package for analysis of a GFEM

A package was developed for analytical calculation of the integrals of a GFEM method, its semi-discrete equations, phase and group velocities, and anisotropies. The package has also routines for visualization of patches and basis functions.

9.3 Byproducts

9.3.1 Package for translating *Mathematica* expressions to High Performance Fortran

FORTRAN 90/95 is an object-based language: it has a complete set of operations for the data structure array. From another side, the FORTRAN 90/95 loop statements, and the added to them comments of High Performance Fortran (or Open MP), can be interpreted as second order functions in a functional language, and vice versa: the second order functions in a functional language have natural interpretation as specifications of data-parallel computations. Therefore, it is easy to translate a functional language code that manipulates arrays, to the data-parallel language High Performance Fortran.

9.3.2 Package for making dimensionless models

The package is an *Mathematica* implementation of an algebraic proof of the Buchingham's Pi-theorem. It can be used to convert to dimensionless form a large set of equations. The package was developed in order to find theoretically the ratio between the advection and chemistry time steps.

9.3.3 Code generation of the Jacobian and unrolled Gauss elimination

We can facilitate the Implicit Solvers Layer with different kind of code generation. Two *Mathematica* notebooks was made: the first is for the Jacobian generation from a given right hand, the second is for the generation of FORTRAN code for the unrolled solution with Gauss elimination of a system of linear equations.

9.3.4 Advection simulations with lattice gas automata

Cellular automata and their derivative, the lattice gas automata, can be used to simulate *simultaneously* – without splitting – all physical processes in the air pollution mathematical model. The literature of this field was studied. A C++ program and a *Mathematica* notebook was developed for lattice gas simulations of the creeping snow problem, which is basically advection with changing boundaries. The simulations had instabilities, and my conclusion was if one wants to apply these methods, a greater level of brightness is required from him or her, than for the level required to apply the standard PDE and ODE numerics.

9.3.5 *Mathematica* implementation of the Strassen algorithm for fast matrix multiplication

Most of the performance models of the Strassen algorithm for fast matrix multiplication (of large matrices) assume that the scalar multiplication and addition are equally expensive, since these operations are hardware implemented. This is not true in a system that provides arbitrary precision – like *Mathematica* – when multiplication and addition are software implemented. So, new models should be derived and investigated. Together with the Strassen algorithm for fast matrix multiplication, was implemented and analyzed also the Strassen algorithm for fast matrix inversion, and the algorithm for fast multiplication of complex matrices.

The work on this project was very helpful to complete my mathematical engineering skills, and was quite inspirational for what kind of patterns should be used when numerical algorithms are developed.

Appendix A

Object-Oriented Terms Glossary

Taken from the GoF's book [20].

abstract operation An operation that declares a signature but doesn't implement it. In C++, an abstract operation corresponds to a pure virtual member function. acquaintance relationship A class that refers to another class has an acquaintance with that class.

aggregate object An object that's composed of subobjects. The subobjects are called the aggregate's parts, and the aggregate is responsible for them.

aggregation relationship The relationship of an aggregate object to its parts. A class defines this relationship for its instances (e.g., aggregate objects).

class A class defines an object's interface and implementation. It specifies the object's internal representation and defines the operations the object can perform.

class diagram A diagram that depicts classes, their internal structure and operations, and the static relationships between them.

class operation An operation targeted to a class and not to an individual object. In C++, class operations are are called static member functions.

concrete class A class having no abstract operations. It can be instantiated. constructor In C++, an operation that is automatically invoked to initialize new instances.

coupling The degree to which software components depend on each other.

delegation An implementation mechanism in which an object forwards or delegates a request to another object. The delegate carries out the request on behalf of the original object.

design pattern A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context.

destructor In C++, an operation that is automatically invoked to finalize an object that is about to be deleted.

dynamic binding The run-time association of a request to an object and one of its operations. In C++, only virtual functions are dynamically bound.

encapsulation The result of hiding a representation and implementation in an object. The representation is not visible and cannot be accessed directly from outside the object. Operations are the only way to access and modify an object's representation.

framework A set of cooperating classes that makes up a reusable design for a specific class of software. A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations. A developer customizes the framework to a particular application by subclassing and composing instances of framework classes.

friend class In C++, a class that has the same access rights to the operations and data of a class as that class itself.

inheritance A relationship that defines one entity in terms of another. Class inheritance defines a new class in terms of one or more parent classes. The new class inherits its interface and implementation from its parents. The new class is called a subclass or (in C++) a derived class. Class inheritance combines interface inheritance and implementation inheritance. Interface inheritance defines a new interface in terms of one or more existing interfaces. Implementation inheritance defines a new implementation in terms of one or more existing implementations.

interaction diagram A diagram that shows the flow of requests between objects.

interface The set of all signatures defined by an object's operations. The interface describes the set of requests to which an object can respond. metaclass Classes are objects in Smalltalk. A metaclass is the class of a class object.

method See operation.

object A run-time entity that packages both data and the procedures that operate on that data.

object composition Assembling or composing objects to get more complex behavior.

object diagram A diagram that depicts a particular object structure at run-time.

object reference A value that identifies another object.

operation An object's data can be manipulated only by its operations. An object performs an operation when it receives a request. In C++, operations are called member functions. Smalltalk uses the term method. Both method and operation for the C++ member functions are used in the thesis

overriding Redefining an operation (inherited from a parent class) in a subclass.

parameterized type A type that leaves some constituent types unspecified. The unspecified types are supplied as parameters at the point of use. In C++, parameterized types are called templates.

parent class The class from which another class inherits. Synonyms are superclass (Smalltalk), base class (C++), and ancestor class.

polymorphism The ability to substitute objects of matching interface for one another at run-time.

private inheritance In C++, a class inherited solely for its implementation.

protocol Extends the concept of an interface to include the allowable sequences of requests.

receiver The target object of a request.

request An object performs an operation when it receives a corresponding request from another object. A common synonym for request is message.

signature An operation's signature defines its name, parameters, and return value.

subclass A class that inherits from another class. In C++, a subclass is called a derived class.

subsystem An independent group of classes that collaborate to fulfill a set of responsibilities.

subtype A type is a subtype of another if its interface contains the interface of the other type.

supertype The parent type from which a type inherits.

type The name of a particular interface.

Appendix B

UML Quick Reference

Figure B.1: Package, dependency, note

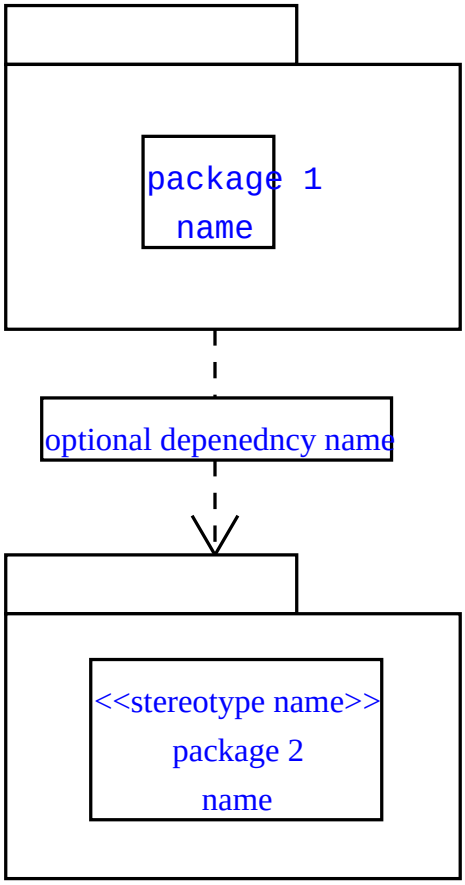


Figure B.2: Class Deffinition

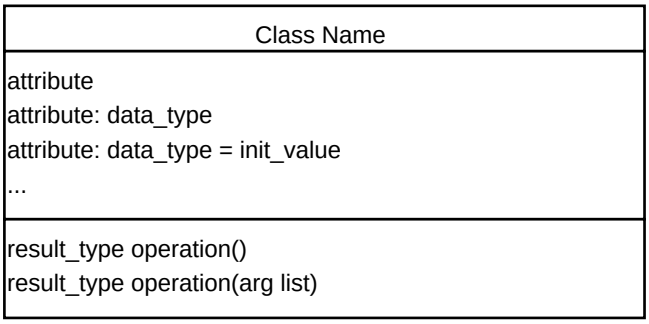


Figure B.3: Association

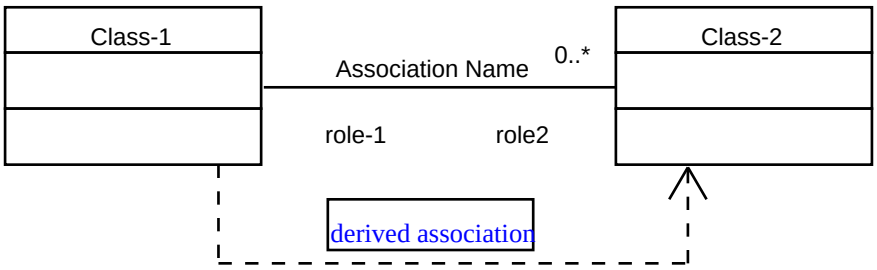


Figure B.4: Aggregation, navigability, and multiplicity

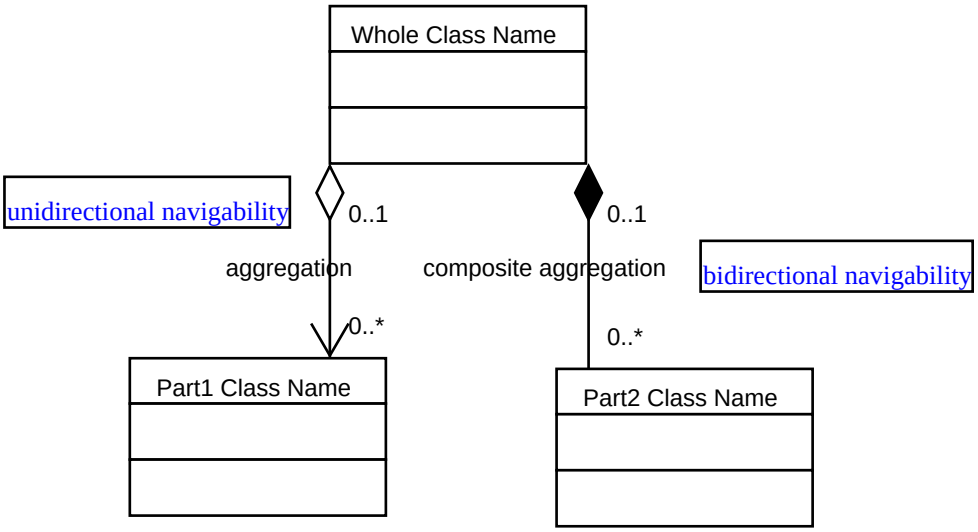


Figure B.5: Generalization/Specialization

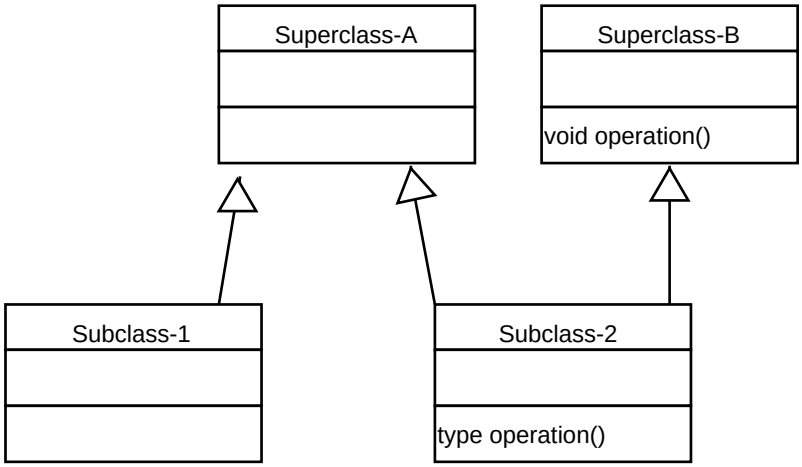
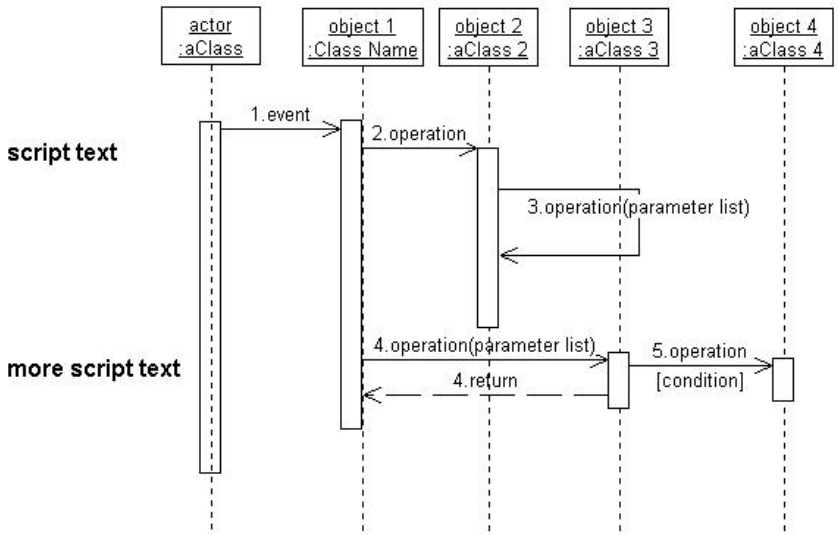


Figure B.6: Sequence diagram



Bibliography

[1] I. Ackerman, H. Hass, B. Schell, and F. S. Binkowski. Regional modelling of particulate matter with made. *Environmental Management and Health*, 10:201–208, 1999.

[2] Christorpher Alexander. *"The Timeless Way of Building"*. Oxford Univ Pr (Trade), 1979.

[3] V. Alexandrov, A. Sameh, Y. Siddique, and Z.Zlatev. Numerical integration of chemical ode problems airsing in air pollution problems. *Envinronmental Modelling and Assesment*, 2:365–377, 1997.

[4] Anton Antonov. Jacobian code generation. http://www.imm.dtu.dk/ũniaaa/OODEM/JacobianGeneration.nb, 2001.

[5] Anton Antonov, Per Grove Thomsen, and Zahari Zlatev. Phase and group velocities of 2d numerical approximations for hyperbolic equations. unpublished.

[6] Brad Appleton. Patterns and software: Essential concepts and terminology. http://www.enteract.com/b̃radapp/docs/patterns-intro.html.

[7] Ibrahima Ba and Barry Smith. The ALICE memory snooper. http://www.mcs.anl.gov/ams.

[8] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[9] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc home page. http://www.mcs.anl.gov/petsc, 2000.

[10] A. Bastrup-Birk, J. Brandt, I. Uria, and Z. Zlatev. Studying cumulative ozone exposures in europe during a 7-year period. *Journal of Geophysical Research*, 102:23917–23935, 1997.

[11] Shai Ben-Yehuda. Pattern language for framework construction. In Robert Hanmer, editor, *The 4th Pattern Languages of Programming Conference 1997*, number 97-34 in Technical Report. Washington University, Technischer Bericht, 1997. http://jerry.cs.uiuc.edu/p̃lop/plop97/Workshops.html.

[12] S. Brenner and L. Ridgway-Scott. *The Mathematical Theory of Finite Element Method*. Springer-Verlag, 1994.

[13] Ian Chai. Pedagogical framework documentation: How to document object-oriented frameworks. summary of an empirical study. In *World Engineering Congress '99*, 1999. Ian Chai has a PhD thesis under the same name; the article can be found at http://st-www.cs.uiuc.edu/c̃hai/writing/wec99.html.

[14] Jesper Christensen. Testing advection schemes in a three-dimensional air pollution model. *Mathl. Comput. Modeling*, 18(2):75–88, 1993.

[15] James. O. Coplien. C++ idioms. In Neil Harrison, Brian Foote, and Hans Rohnert, editors, *Pattern Languages of Program Design 4*, volume 4, pages 167–198. Addison-Wesley Longman, Inc., 2000.

[16] W. P. Crowley. Numerical advection experiments. *Mon. Weth. Rev.*, 96:1–11, 1968.

[17] C. J. Date. *Introducation to Database Systems*, volume 1 of *Systems Programming Series*. Addison-Wesley, 4 edition, 1986.

[18] I. Dimov, I. Faragó, and Z. Zlatev. Commutativity of the operators in splitting methods for air pollution models. Technical Report 04/99, Bulgarian Academy of Sciences, 1999.

[19] European Commission for the Environment. Auto-Oil II Programme. http://europa.eu.int/comm/environment/autooil/, 2000.

[20] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[21] K. Georgiev and Z. Zlatev. Parallel sparse matrix algortihms for air pollution models. *Parallel and Distributed Computing, to appear*.

[22] W. Gery, G. Z. Whitten, J. P. Killus, and M. C. Dodge. A photochemical kinetics mechanism for urban and regional computer modeling. *Journal of Geophysical Research*, 94:12925–12956, 1989.

[23] P. M. Gresho and R. L. Sani. *Incompressible Flow and Finite Element Method*. John Willey and Sons, 1998.

[24] Neil B. Harrison. Patterns for logging diagnostic messages. In Robert Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design*, Softwatre Patterns Series, chapter 16. Addison Wesley, 1998.

[25] R. M. Harrison, Z. Zlatev, and C. J. Ottley. A comparison of the predictions of an eulerian atmospheric transport chemistry model with experimental measurements over the north sea. *Atmospheric Environment*, 1994.

[26] O. Hov, Z. Zlatev, R. Berkowicz, A. Eliassen, and L. P. Prahm. Comparison of numerical techniques for use in air pollution models with non-linear chemical reactions. *Atmospheric Environment*, 23:967–983, 1988.

[27] Ralph Johnson. Documenting frameworks using patterns. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) '92 Proceedings*. ACM Press, 1992.

[28] J. D. Lambert. *Numerical Methods for Ordinary Differential Equations*. Wiley, Chichester, 1991.

[29] Hans Petter Langtangen. *Computatinal Partial Differential Equations*. Springer, 1999.

[30] D. Lanser and J. G. Verwer. Analysis of operator splitting for advection-diffusion-reaction problems from air pollution modeling. *Journal of computational and applied mathematics*, pages 201–216, 1999.

[31] Doug Lea. Checklist: A pattern ... Doug Lea's homepage http://g.oswego.edu/.

[32] James Lighthill. *Waves in Fluids*. Cambridge University Press, 1978.

[33] G. I. Marchuk. *Methods for Numerical Mathematics*. Springer-Verlag, 2 edition, 1982.

[34] Bertran Mayer. *Object-Oriented Software Construction*. Prentice Hall, 2 edition, 2000.

[35] G. McRae, W. R. Goodin, and J. H. Seinfield. Numerical solution of the atnospheric diffusion equation for chemically reacting flows. *Journal of Computational Physics*, 45(1):356–396, 1982.

[36] Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, 1992.

[37] C. R. Molenkampf. Accuracy of finite-difference methods applied to the advection equation. *J. Appl. Meteor.*, 7:160–167, 1968.

[38] Pierre-Alain Muller. *Instant UML.* Wrox Press, 1997.

[39] Daniel Quinlan and Nehal Desai. A++/P++ tutorial guide. http://www.llnl.gov/casc/Overture.

[40] R.A.Trompert and J.G.Verwer. A static-regridding method for two dimensional parabolic partial differential equations. *Applied Numerical Mathematics*, 8:65–90, 1991.

[41] John H. Seinfeld and Spyros N. Pandis. *Athmospheric chemistry and physics.* John Wiley & Sons, 1998.

[42] G. Strang and G. J. Fix. *An analysis of the Finite Element Method.* Prentice-Hall, 1973.

[43] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 3 edition, 1997.

[44] Leonor Tarrason, Sergey Dutchak, Jan Schaug, and Markus Amman. EMEP home page, co-operative programme for monitoring and evaluation of the long range transmission of air pollutants in europe. http://www.emep.int, 2001.

[45] GODESS team. Godess: Generic ode solving system. http://www.godess.org, 2000.

[46] C. Ambelas Skjøth, A. Bastrup-Birk, J. Brandt, and Z. Zlatev. Studying variations of pollution levels in a given region of europe during a long time-period. *Systems Analysis Modelling Simulation*, 37:297–311, 2000.

[47] A. S. Tomlin, S. Ghorai, G. Hart, and M. Berzins. 3d adaptive unstructured meshes for air pollution modelling. *Environmental Management and Health*, 10:267–274, 1999.

[48] Lloid N. Trefethen. Group velocities in fnite difference schemes. *SIAM Review*, 24(2):113–136, 4 1984.

[49] Todd Veldhuizen. Techniques for scientific C++. http://www.extreme.indiana.edu/tveldhui/papers/techniques/

[50] J. G. Verwer, W. H. Hundsdorfer, and J. G. Blom. Numerical time integration for air pollution models. Technical Report MAS-R9825, CWI, 1998. Invited survey paper for APMS'98 (Air Pollution Modelling and Simulation, INRIA, Champs-sur-Marne, Oct. 26-29, 1998.

[51] Robert Vichevetsky. *Computer methods for Partial Differential Equations*, volume 1 of *Computational Mathematics.* Prentice-Hall, 1981.

[52] R. Vichnevetsky and J. B. Bowles. *Fourier Analysis of Numerical Approximations of Hyperbolic Equations.* SIAM, 1982.

[53] Robert Vichnevetsky. Propagation through numerical mesh refinement for hyperbolic rquations. *Mathematics and Computers in Simulation*, 23:344–353, 1981.

[54] Robert Vichnevetsky. Wave propagation analysis of difference schemes for hyperbolic equations: A review. *Numerical Methods in Fluids*, 7(5):409–452, 5 1987.

[55] Niklaus Wirth. *Algorithms + data structures = programs.* Series in Automatic Computation. Prentice Hall. ASIN: 0130224189.

[56] Stephen Wolfram. *Mathematica: A System for Doing Mathematics by Computer.* Wolfram Media, Cambridge University Press, 4 edition, 1999.

[57] Z. Zlatev. Application of predictor-corrector schemes with several correctors in solving air pollution problems. *BIT*, 24:700–715, 1984.

[58] Z. Zlatev. *Computer Treatment of Large Air Pollution Models.* Kluwer, 1995.

[59] Z. Zlatev, I. Dimov, Tz. Ostromsky, G. Geernaert, I. Tzvetanov, and A. Bastrup-Birk. Calculating losses of crops in denmark caused by high ozone levels. *Environmental Modeling and Assessment*, to appear.

[60] Z. Zlatev, J. Fenger, and L. Mortensen. Relationships between emission sources and excess ozone concentrations. *Computers and Mathematics with Applications*, 32:101–123, 1996.

[61] Z. Zlatev, G. Geernaert, and H. Skov. A study of ozone critical levels in denmark. *EUROSAP Newsletter*, 36:1–9, 1999.