Adaptive spectral elements for diffuse interface multi-fluid flow

Citation for published version (APA):
Barosan, I. (2003). *Adaptive spectral elements for diffuse interface multi-fluid flow*. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Mechanical Engineering]. Technische Universiteit Eindhoven. https://doi.org/10.6100/IR566180

DOI:
10.6100/IR566180

Document status and date:
Published: 01/01/2003

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 04. Oct. 2023

Adaptive Spectral Elements
for
Diffuse Interface Multi-Fluid Flow

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Barosan, Ion

Adaptive Spectral Elements for Diffuse Interface Multi-Fluid Flow /
by Barosan I. – Eindhoven : Technische Universiteit Eindhoven, 2003.
Proefschrift. – ISBN 90-386-2585-5
NUR 919
Subject headings: mortar spectral elements; adaptive mesh refinement / spectral elements; distributed numerical simulation

This thesis was prepared with the LaTeX 2_ε Documentation System.
Printed by Universiteitsdrukkerij TU Eindhoven, Eindhoven, The Netherlands.

This research was financially supported by the Dutch Polymer Insitute (DPI), Proj. No. 161.

Adaptive Spectral Elements
for
Diffuse Interface Multi-Fluid Flow

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Eindhoven,
op gezag van de Rector Magnificus, prof.dr. R.A. van Santen,
voor een commissie aangewezen door het College voor Promoties
in het openbaar te verdedigen op
woensdag 24 september 2003 om 16.00 uur

door

Ion Barosan

geboren te Galateni, Roemenië

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr.ir. H.E.H. Meijer
en
prof.dr. P.A.J. Hilbers

Copromotor:
dr.ir. P.D. Anderson

TO: *Ilinca, Simona, Marin and Ioan,*
Diana, Alina, Iulia and Stefan.

Contents

List of Symbols

a	scalar
a_n, a_n^k, a_{nm}^k	Legendre coeficients, for element k, also called spectrum
$\mathbf{a}$	vector
$\mathbf{A}$	tensor
$\mathbf{A}^T$	transpose of $\mathbf{A}$
$\mathbf{A}_{11}^k$	boundary matrix (element k)
$\mathbf{A}_{12}^k$	coupling boundary-interior matrix (element k)
$\mathbf{A}_{11}^k$	coupling interior-boundary matrix (element k)
$\mathbf{A}_{11}^k$	interior matrix (element k)
$\mathbf{B}_{ij}^k$	discrete elemental mass matrix
c_n	solution coefficients
$\mathcal{C}^0$	space of continuous functions
$\mathcal{C}^1$	space of continuous functions in first derivative
D_{ij}^k	discrete elemental derivative operator matrix
$f, f_h, \mathbf{f}_h$	inhomogeneous term
$\mathcal{F}(x)$	function in x
h	discretization parameter
$\mathcal{H}_0^1(\Omega)$	space of functions which are square integrable and whose derivatives are square integrable over Ω

$\mathbf{I}$	interval, subdomain of $\mathbb{R}$
J^k	elemental jacobian
K	number of elements
K^M	number of master elements
K^S	number of slave elements
K_{SM}	number of functions to represent the projection space
L^k	length of subdomain k
M, N, N^S, N^M	order of polynomials
L_N	N^{th} order Legendre polynomial
$\overline{M}$	number of mortars
$\overline{N}_i$	order of polynomials on the mortars
$\mathcal{O}$	of order of
$\mathbb{P}_N$	space of polynomials of degree $\leq N$
Pqm	mortar to edge projection operator
$\mathbf{P}^S$	projection space for slave edge
$\mathbf{P}^M$	projection space for master edge
$\mathbf{Q}, \mathbf{Q}^T$	transformation operator associated with mortar to edge projection
r	regression factor for best fit
s_0	mortar offset
S	skeleton of mortar system
S_t, S_r	sums defined for least squares best fit
t	time

$u, u_h, \mathbf{u}_h$	solution function
$\tilde{u}$	extrapolated approximation to u
ω	test function
$\mathbf{W}_h$	mortar space
$\overline{x}, \overline{y}$	local mortar coordinates
$X, \mathbf{X}$	solution space for u
$X_h, \mathbf{X}_h$	discretization space
$X_h^\perp$	orthogonal complement to X_h

Greek Symbols

γ, γ_i	mortar
Γ	elemental edge
Γ_i^k	elemental edge i of element k
$\Gamma_{i,j}$	interface (boundary) between elements i and j
δ_{ij}	Kronecker edge
Δt	time step size
ϵ	error norm, tolerance
Λ	interval]-1;1[
ξ, η	zero of N^{th} order polynomial, collocation point $\xi, \eta \in \Lambda$
Π_h	projection onto X_h
ρ	Gauss Lobatto weights
σ_e, σ	decay rate, exponential

$\sum'$	direct stiffness summation
v_i^1, v_i^2	vertex
Φ	mortar function
Φ, Ψ	arbitrary functions
$\boldsymbol{\Phi}$	polynomial of degree $N-2$
$\Psi_i^{N-1}, \Psi_i^{N-2}$	polynomials of degree $N-1$, $N-2$
ψ	mortar test function
$\boldsymbol{\Omega}$	domain of solution (excluding boundary), subdomain of $\mathbb{R}$
$\overline{\boldsymbol{\Omega}}$	closure of Ω (including boundaries)

Other Symbols

∇	gradient operator
∇^2	Laplace operator
$(\cdot,\cdot)$	inner product
$\|\cdot\|_{\mathcal{L}^2}$	$\mathcal{L}^2$ norm
$\|\cdot\|_{\mathcal{H}^1}$	$\mathcal{H}^1$ norm
$\|\cdot\|_{\infty,GL}$	∞, GL norm
$(\cdot,\cdot)_{GL}$	Gauss Lobatto quadrature
$\Longrightarrow$	maps to
$\partial\Omega$	boundary of Ω

Superscripts

k	refers to element k
T	transpose of matrix

Other Notations

bold	discrete vector
PatchSegment	C++ class
SEPRAN	Fortran library
$ProjectQT$	C++ class member function
$edge$	C++ class data member
$\tilde{}$	extrapolated values
$\tilde{}$	restriction to

Chapter 1

Introduction

1.1 Motivation of this thesis

The diffuse-interface approach has been used to study a wide range of phenomena involving topological changes: nucleation and growth, spinodal decomposition, droplet breakup and coalescence (Anderson *et al.*, 1998). These models have successfully been applied to situations in which the physical phenomena of interest have a length scale commensurable with the thickness of the interfacial region and fluid flows involving large interface deformations and/or topological changes, such as droplet breakup and coalescence.

Most of the studies on topological changes focus on small-scale systems, in which it is assumed that the numerical interface thickness is close to the real interface thickness. In general, for large systems, for which the droplet size is much larger than the physical value of the interface thickness, the real interfacial thickness can not be captured numerically. The scaling in such systems needs special attention, because if the real interfacial thickness is to be replaced by a numerically acceptable thickness, we have to make sure that we are still describing the same system with the same interfacial tension and diffusion (Verschueren, 1999).

Commonly, diffuse-interface models introduce a small length scale (the interface width), which places stringent conditions on numerical solution methods. Based on the Cahn-Hilliard expression of the free energy (Cahn and Hilliard, 1958) the critical size L_c is calculated to be:

$$L_c = \frac{\xi}{\mathcal{O}(C)} \ , \tag{1.1}$$

with C the Cahn number and ξ the interface thickness. Small interfacial thicknesses would require the use of a smaller Cahn number in the simulations and consequently, extremely small mesh sizes and, hence, require excessive computational times. For Cahn numbers, typically used in the simulations ($C = 0.02$), and the typical interface thickness (order of magnitude 10 nm), the computational domain used has a length of the order of 500 nm. If we want to extend to larger systems, the real interface thickness can not be captured numerically in general.

To circumvent the above mentioned problems for large systems, two possibilities can be used:

1. scaling of the system (Verschueren, 1999; Lowengrub *et al.*, 1998)
2. adaptive mesh refinement (AMR).

In this thesis an adaptive mesh refinement (AMR) technique based on the mortar spectral element method will be implemented to capture the interface thickness adaptively. The refinement algorithm will track the movement of the diffuse-interface, refining the mesh only around the moving interface, and coarsening the mesh in the rest of the computational domain.

Spectral element methods (SEM) are high-order (*p*-type) weighted residual techniques for the numerical solution of the partial differential equation

$$\mathcal{L}(u) = f \text{ in } \mathbf{\Omega} \tag{1.2}$$

here $\mathcal{L}$ is a continuous positive-definite differential operator and $f \in \mathcal{C}^0(\overline{\mathbf{\Omega}})$, that combine the generality of *h*-type finite element techniques (Ciarlet, 1978; Schwab, 1998) with the rapid convergence rate of spectral methods (Gottlieb and Orszag, 1977; Canuto *et al.*, 1988). Handling complex geometries $\mathbf{\Omega}$ by the spectral element method mainly relies on a domain decomposition, first introduced by Patera (1984), most often without overlapping domains. The computational domain is broken up into K elements $\mathbf{\Omega}^k \in \mathbf{\Omega}$ of the spectral type, on each of which the variables are approximated by N^{th} order tensor product polynomial expansions. Variational projection operators and Gauss numerical quadrature are used to generate a discrete set of equations $\mathbf{A}\ \mathbf{u} = \mathbf{f}$. Coupled to fast order-independent iterative solvers (Canuto *et al.*, 1988; Barrett, 1994) these discretizations yield numerical algorithms which have proven to be computationally efficient on both serial and parallel processors (Henderson and Karniadakis, 1991). It is proven (Maday *et al.*, 87) that convergence of the spectral element approximation to the exact solution is exponential, where convergence is achieved by increasing the degree N of the polynomial approximation, while keeping the number of elements K as well as their identity fixed.

The limitation of the spectral element method is its lack of flexibility and generality with respect to complex geometry mesh generation and locally refined resolution capabilities. These limitations severely hinder any development in the areas of mesh generation, adaptive mesh refinement and the treatment of *moving boundaries*. The basic spectral element method, relies on a domain decomposition that consist of identical conforming elements, so that the decomposition of the domain must also satisfy some properties of conformity that are standard in spectral elements. To avoid these limitations, in this thesis we will develop an adaptive mesh refinement method based on the mortar element method, first introduced by Bernardi *et al.* (1994).

The rigidity of the conforming formulation, is eliminated by allowing non-conforming matching between sub-domains, functionally (the polynomial degree per element can vary) as well as geometrically (multiple elements can share a single edge of an adjacent element). Figure 1.1 illustrates the difference between the two types of refinement, left functionally called *p*-refinement, and right geometrically, called *h*-refinement. In the spectral element methods literature, instead of *p*-refinement the term *N*-refinement is used. In this thesis, we freely use both terms in a interchangeable manner.

The non-conforming formulation of a spectral element discretization is the most crucial development needed for the extension to adaptive methods. Without the flexibility afforded by non-conformity, adaptive methods would be very cumbersome and inefficient. The spectral element method can combine the advantages of finite element methods and spectral methods, provided that a non-conforming formulation is developed in a way consistent with the

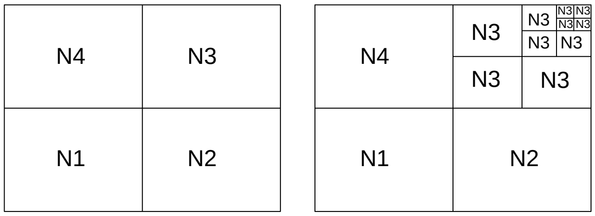


Fig. 1.1: Functionally (*left*) and geometrically (*right*) non-conforming mesh. N_i is the degree of the polynomial approximation.

convergence properties of the existing conforming formulation. The mortar element method, considered here, represents a domain decomposition approach (Chan and Mathew, 1994) in which there is a clean decoupling of a local residual evaluation per element, where the spectral element structure is preserved, by transmission of continuity and boundary conditions per element.

To estimate what order of approximation and what size of elements are required, for a specific mesh, is a difficult task. In most other studies it is based on trial and error. To avoid this, we need an adaptive mesh refinement method combined with an automatic mesh generation method. To achieve this goal, it is necessary to develop some error estimators to serve as criteria for refinement decisions (Mavriplis, 1990). The role of the error estimators is, of course, to provide an estimate of the actual error on a per element basis as well as globally. This estimate is used to detect where to refine the mesh locally.

Adaptive SEMs have gained importance because they provide robustness, reliability, and time and space efficiency. In such a method, the computational domain is first discretized to create a mesh. During the solution process, portions of the discrete domain are spatially refined or coarsened (*h-refinement*), the method order is varied (*p/N - refinement*). Each method concentrates the computational effort in areas where the solution resolution would otherwise be inadequate (Clark *et al.*, 1994). Computationally demanding problems make parallel computation essential. However, parallelism introduces complications such as the need to balance processor loading, to coordinate inter-processor communications, and to manage the distribution of the data. In general, the standard methodology for optimizing parallel SEM programs relies on a static partitioning of the mesh across the cooperating processors. In the adaptive case, a good initial partition is not sufficient to assure high performances. Due to the adaptivity, the load balance necessitates a dynamic partitioning and redistribution of data. The chosen data structure must support this dynamic mesh migration (Williams, 1992). Parallelism is generally explicit, achieved through the use of a message passing library such as the Message Passing Interface - MPI (MPI, 1994), and requires a partitioning algorithm to distribute data among processing nodes. The adaptive SEM computations can be distributed in a natural way by a domain decomposition of the underlying mesh, but being adaptive, these meshes will change and the system must account for this. Adaptive algorithms that utilize unstructured meshes (Adjerid *et al.*, 1992; Armstrong, 1991; Shephard, 1988a,b; Williams, 1992) make the task of balancing processor computational load more difficult than with uniform structures. The dynamically adapted mesh has an equivalent graph representation $G = (V, E)$, where mesh elements serve as the graph vertices V and connections between mesh elements are the graph edges (E). Graph partitioning algorithms produce some partitions V_k with the goals of placing equals numbers of vertices in each subset of V_k, while minimizing the number of edges ”cut” by partitions. The graphs partitioning is related to the mesh partitioning problem by assigning each partition V_k to a processor unit. Vertices represent units of work to be balanced among the processors, while the edges represent the communication needed. For more information about the mesh partitioning, the interested reader is referred to (Berger and Saltzman, 1993; Farhat and Lesoinne, 1993; Shephard *et al.*, 1995; Barnard and Simon, 1994; Hendrickson and Leland, 1993; Sohn *et al.*, 1996) and (Gervasion, 1998; Patra and Oden, 1995).

1.2 Objective of this thesis

The main research objective of this thesis is to apply adaptive mesh refinement techniques to reduce the length-scale problems in the diffuse-interface techniques and to track the movement of the boundary interface for different values of the Cahn number C. The emphasis in this thesis is on the software implementation of the non-conforming discretization and error estimators as a development towards adaptive mesh refinement techniques. The flexibility and effectiveness of the implementation will be illustrated by several test problems and applying the diffuse-interface model for multi phase flow problems. The initial development of the software will not incorporate the mesh partitioning process needed for parallel processing.

We implement two key features in the development of adaptive meshes: the non-conforming mortar element method and a single mesh *a posteriori* error estimates. The distributed mesh structure in our system is based on the Voxel Data Base (VDB) introduced by Williams (1992), which provides the operators to create and manipulate distributed mesh data. The VDB structure will be extended to support also the manipulation of the computational data associated with the mesh (see subsection 4.4). Using the developed techniques, an application based on the diffuse interface model, will be investigated :

- diffuse interface modelling of the morphology and rheology of immiscible polymer blends using mortar elements.

1.3 Thesis outline

The outline of the thesis is as follows. In Chapter 2, we present the basics of the spectral element method, concentrating on the formulation and solution techniques for the Poisson equation. The chapter ends with a few one-dimensional applications that use adaptive mesh refinement techniques. The mortar element method based on the non-conforming discretization is introduced in Chapter 3 for the solution of a two-dimensional Poisson equation. Also, in Chapter 3, the single mesh *posteriori error* estimators for spectral element techniques are introduced. The chapter ends with several applications that use the mortar element method and unstructured approaches on high-order mesh elements. Chapter 4 provides details of the software architecture of the adaptive mesh refinement implementation. The basic operations for the spectral element method, the mesh and data structure are presented. Two wrapping techniques for LAPACK and BLAS are compared. Chapter 5 provides illustrations of the non-conforming formulation for the diffuse interface modelling of the morphology and rheology of immiscible polymer blends. Concluding remarks and possible directions for future research are presented in Chapter 6. Appendix A presents a detailed description of error estimators, for both, one-dimensional and two-dimensional cases. Finally, Appendix B provides details of the wrapping techniques used in the software implementation of the adaptive mesh refinement process.

Chapter 2

Basics of the Spectral Element Methods with Adaptive Mesh Refinement

In this chapter the basics of the spectral method are introduced and discussed. First, the one-dimensional technique is described, which will allow the introduction of the two-dimensional spectral technique, next the adaptive mesh refinement technique for a one-dimensional model equation is outlined, and several results are presented.

2.1 Introduction

An adaptive formulation of the spectral element method is aimed at increasing the flexibility and range of capabilities of high-order spectral methods in general. While spectral methods provide highly accurate solutions to partial differential equations governing complex physical phenomena, their use has been limited to idealized research problems due to their lack of geometric flexibility (Canuto *et al.*, 1988). In this chapter, we investigate an adaptive spectral element method for one-dimensional problems which automatically allocates resolution where it is most needed in an optimal fashion.

Previous work in non-conforming discretization (Maday *et al.*, 1989) and error estimators (Mavriplis, 1990) for the spectral element method constituted a first step towards an adaptive formulation, and will be presented in the next chapter. Spectral element methods are weighted residual techniques for the approximation of partial differential equations that combine the rapid convergence rate of spectral methods with the generality of finite element techniques. By subdividing a complex domain into elements, an accurate solution of many problems can be derived with substantially fewer degrees of freedom than would be required with a lower-order discretization.

Spectral and *h-p* finite element methods are most commonly based on Chebyshev and Legendre polynomials. These polynomials are the eigenfunctions of an appropriately defined singular Sturm-Liouville problem and form an expansion basis for representing square-integrable functions $u(x) \in \mathcal{L}^2$. Maday and Patera (Maday and Patera, 1988) have shown that this approach provides a weak $\mathcal{C}^1$ continuity (*continuity in function and its first derivative*) across an element interface. In the variational approach, continuity across element interfaces is naturally imposed. Exponential convergence of numerical solutions in practical situations depends on a number of factors. One of them is the non-uniformity of the mesh, that can degrade convergence of the solution. Such a feature must be isolated or resolved before fast convergence *(exponential convergence)* is realized.

The accuracy of the approximation can be improved by either increasing the number of sub-domain elements, called *h-refinement*, or by increasing the polynomial order of a fixed number of elements, called *N-refinement*(*p-refinement*), or by moving element boundaries. As the main goal of the thesis is to investigate adaptive methods for moving interface problems, we consider only the *h-refinement* method combined with moving elements boundaries, keeping the polynomial order fixed.

2.1.1 One-dimensional spectral element methods

In order to describe the main basic aspects of spectral element methods, we provide a step by step formulation of the one-dimensional spectral element solver for a Poisson problem. In higher dimensions, most of the basic operations are the same except the geometry, that has to be represented with more complicated elements using tensor products of the one-dimensional approximation. The details of more general elliptic equations can be found in Maday and Patera (1988).

Consider the solution of the Poisson equation on a domain Ω of $\mathbb{R}$:

Problem 1: Find *u(x)* $\in \mathcal{C}^2(\Omega) \cap \mathcal{C}^1(\overline{\Omega})$ such that:

$$-\frac{\partial^2 u}{\partial x^2} \;=\; f \text{ on } \Omega \;=\; \{x | x_l < x < x_r\}, \;\text{ and }\; u = 0 \;\text{ on }\; \partial\Omega, \tag{2.1}$$

where $\partial\Omega \;=\; \{x_l, x_r\}$ is the boundary of Ω and $f \in \mathcal{C}^0(\Omega)$ is a prescribed function. Without loss of generality, homogeneous Dirichlet boundary conditions are imposed.

We now define the space of acceptable solutions to equation (2.1). First, we recall that the Lebesque space $\mathcal{L}^2(\Omega)$ is defined as:

$$\mathcal{L}^2(\Omega) \;=\; \{v \text{ measurable over } \Omega \text{ and } \int_\Omega v^2\, dx < \infty\}, \tag{2.2}$$

and that this space is equipped with a scalar product

$$(u, v) \quad = \quad \int_\Omega u(x)v(x)\, dx,\; \forall u, v \in \mathcal{L}^2(\Omega), \tag{2.3}$$

and associated norm $\|v\| = (v, v)$. Also, we define the Sobolev space $\mathcal{H}^1(\Omega)$ consisting of all functions that are in $\mathcal{L}^2(\Omega)$ and whose first derivatives are also in $\mathcal{L}^2(\Omega)$. The reader is refered to e.g. Adams (1985) for the definition of standard spaces, norms and inner products.

The solution u of equation (2.1) belongs to $\mathcal{H}^1_0(\Omega)$, the space of $\mathcal{H}^1(\Omega)$ containing all functions $\mathcal{H}^1(\Omega)$ that vanish at the boundary $\partial\Omega$. Now, problem 1 is also well-posed in $X \;=\; \mathcal{H}^1_0(\Omega)$, in the sense that the following formulation of the problem admits a unique solution:

Problem 2: Find a *u(x)* $\in X$ such that:

$$\int_\Omega \frac{\partial u}{\partial x}\frac{\partial v}{\partial x}\, dx \;=\; \int_\Omega f v\, dx,\; \forall v \in X, \tag{2.4}$$

or in a more concisely abstract form:

Find a *u(x)* $\in X$ such that:

$$a(u, v) \; = \; (f, v), \; \forall v \in X, \tag{2.5}$$

where the continuous bilinear form a is defined as:

$$a(u, v) \; = \; \int_{\Omega} \frac{\partial u}{\partial x}(x) \, \frac{\partial v}{\partial x}(x) \, dx, \; \forall u, \; v \; \in \; X, \tag{2.6}$$

and the scalar product as:

$$(f, v) \; = \; \int_{\Omega} f(x) v(x) \, dx, \; \forall f \; \in \mathcal{L}^2(\Omega), \; \forall v \; \in \; \mathcal{H}_0^1(\Omega). \tag{2.7}$$

The discretization involves decomposing the domain Ω into sub-domains Ω^k such that

$$\overline{\Omega} \; = \; \bigcup_{k=1}^{K} \overline{\Omega}^k, \; \forall k, l, \; k \neq l : \;\; \Omega^k \cap \Omega^l \; = \; 0, \tag{2.8}$$

where each sub-domain Ω^k is of length L^k. Here $\overline{\Omega}$ signifies the closure of the domain Ω.

Equation (2.4) is still an infinite-dimensional problem, because the space X contains an infinite number of functions. For a Galerkin numerical approximation of problem 2.1, the variational form (2.4) is tested with respect to a family of discrete finite dimensional spaces X_h, where $h = (N, K)$ denotes a discretization parameter, as follows:

Find $u_h \; \in \; X_h$ such that:

$$\sum_{k=1}^{K} a(u_h, v_h)_{\Omega^k} \; = \; \sum_{k=1}^{K} (f, v_h)_{\Omega^k}, \; \forall v_h \; \in \; X_h. \tag{2.9}$$

In the conforming spectral element method, X_h is taken to be a subspace of $\mathcal{H}_0^1(\Omega)$, consisting of all piecewise high-order polynomials of degree less than or equal to N defined on Ω^k:

$$X_h \; = \; X \cap \mathbb{P}_{N,K}(\Omega). \tag{2.10}$$

The space $\mathbb{P}_{N,K}(\Omega)$ is a space defined for each discretization parameter h over the domain Ω such that:

$$\mathbb{P}_{N,K}(\Omega) \; = \; \{\Phi \in \mathcal{L}^2(\Omega), \Phi|_{\Omega^k} \in \mathbb{P}_N(\Omega^k)\}, \tag{2.11}$$

where $\mathbb{P}_N(\Omega^k)$ is the space of all polynomials of degree less that or equal to N on each sub-domain Ω^k. The space $\mathbb{P}_{N,K}(\Omega)$ ensures that the solution is integrable over Ω, whereas $\mathcal{H}_0^1$ ensures continuity over Ω.

Equation (2.9) must be numerically integrated with sufficient accuracy such that the quadrature errors are of the same order as the approximation error. The convergence and convergence order of u_h towards u is determined essentially by stability and approximation theory. The ellipticity and continuity of the bilinear form ensures the existence and uniqueness of the solution (Maday and Patera, 1988). Approximation theory consists of considering the infimum of $\|u - v_h\|_{1,\Omega}$ over all $v_h \in X_h$, where $\| \cdot \|_{1,\Omega}$ refers to the $\mathcal{H}^1$ norm over Ω. The quadrature applied to integrate equation (2.9) is the Gauss-Lobatto Legendre (GLL) quadrature since it includes boundary points of the interval $\Lambda = [-1, 1]$ as collocation points.

The GLL quadrature is defined as follows:

$$\int_{-1}^{1} \Phi(\xi) d\xi = \sum_{i=0}^{N} \rho_i \Phi(\xi_i) + \epsilon_N, \ \forall \Phi \in \mathbb{P}_{2N-1}(-1, 1), \tag{2.12}$$

$$\xi_0 = -1, \ \ \xi_N = 1 \ \ L'_N(\xi_i) = 0 \, \forall i \in \{1, 2, \ldots, N-1\}, \tag{2.13}$$

with ρ_i the weights, ξ_i the collocation points, L_N is the N^{th} order Legendre polynomial and the error $\epsilon_N \ \sim \mathcal{O}(\Phi^{2N}(\zeta))$ for some point ζ, $-1 < \zeta < 1$; as long as the integrand is a polynomial of degree less that *2N-1* this quadrature rule is exact (Davis and Rabinowitz, 1984).

In order to be able to apply the quadrature, an affine transformation is used to map each spectral element Ω^k to the interval $\Lambda = [-1, 1]$ $(x \in \Omega^k, \xi \in \Lambda, x \Longrightarrow \xi)$. Due to this transformation the terms in equation (2.9) can be written as:

$$a(u_h, v_h)_{\Omega^k} \quad = \quad \int_{\Lambda} \frac{\partial u}{\partial \xi} \frac{\partial v}{\partial \xi} \, J^{-1} \, d\xi, \tag{2.14}$$

$$(f, v_h)_{\Omega^k} \quad = \quad \int_{\Lambda} f v_h(\xi) \; J \; d\xi, \tag{2.15}$$

where J is the Jacobian of the transformation given by:

$$J \ = \ \frac{dx}{d\xi}, \ \ (x \in \Omega^k, \ x \Longrightarrow \xi). \tag{2.16}$$

Applying the GLL quadrature to the system (2.9) yields the following fully discrete problem:

Find $u_h \ \in \ X_h$ such that:

$$\sum_{k=1}^{K} a(u_h, v_h)_{GL} \ = \ \sum_{k=1}^{K} (f, v_h)_{GL}, \ \forall v_h \ \in \ X_h. \tag{2.17}$$

The corresponding discrete inner product $(\cdot, \cdot)_{GL}$ with induced norm $\| \cdot \|_{GL}$ is given by:

$$(u, v)_{GL} \ = \ \sum_{i=0}^{N} J_i \; u(\xi_i) v(\xi_i) \rho_i, \ \forall u, v \in \mathcal{C}^0(\Lambda), \tag{2.18}$$

where $J_i = J(\xi_i)$. Furthermore, the discrete bilinear form $a(\cdot, \cdot)_{GL}$ is given by

$$a(u, v)_{GL} \ = \ \sum_{i=0}^{N} \frac{1}{J_i} \, \frac{\partial u}{\partial \xi}(\xi_i) \frac{\partial v}{\partial \xi}(\xi_i) \rho_i, \ \forall u, v \in \mathcal{C}^1(\Lambda), \tag{2.19}$$

where ξ_i are the Gauss-Lobatto points.

The Galerkin approximation gives the best approximation in the restricted space X_h, but the success of the method lies in the selection of the basis functions. To standardize the basis, we introduce a coordinate transformation to the elemental nodes as:

$$x_i^k = a^k + \frac{b^k - a^k}{2}(1+\xi_i),\ i \in \{0, \cdots, N\}, \tag{2.20}$$

where $\Omega^k = [a^k, b^k]$ represents the current element, ξ_i are roots of $(1-\xi^2)L'_N(\xi) = 0$ and L'_N denotes the derivative of L_N with respect to ξ. The elemental Lagrangian interpolants $h_i(\xi)$ are chosen as a basis so that

$$\forall w_h \in X_h \ \ w_h^k(x) = w_i^k h_i(\xi),\ \ i \in \{0, N\},\ \ x \in \Omega^k,\ \xi \in \Lambda,\ x \Longrightarrow \xi \tag{2.21}$$

where $w_i^k = w_h^k(\xi_i)$ and the notation $\Longrightarrow$ signifies a mapping.
The interpolants h_i have the following properties:

$$h_i(\xi_j) = \delta_{ij},\ \forall i, j \in \{0, N\}^2, h_i \in \mathbb{P}_N(\Lambda). \tag{2.22}$$

where δ_{ij} is the Kronecker delta. They are expressed as:

$$h_i(\xi) = -\frac{(1-\xi^2)L'_N(\xi)}{N(N+1)L_N(\xi_i)(\xi - \xi_i)},\ \ \xi \in \Lambda,\ \forall i \in \{0, N\}. \tag{2.23}$$

From (2.12) and (2.13) it follows that the weights ρ_i are the integrated values of the Lagrangian interpolants $h_i(\xi)$ over Λ.

It is convenient to write other functions that are not in X_h, such as the term f, in terms of Lagrangian interpolants but it is not necessary to impose the additional constraints for the continuity and homogeneous boundary conditions. To construct the right-hand side of the system (2.17), $f(x)$ is approximated by a collocation at the nodal points to produce $f_h(x)$. Rewriting equation (2.17) with the nodal basis expressions for u_h, v_h and f_h, and using that each test function v_h to be non zero at only one global collocation point, we arrive at the fully discrete matrix equation:

$$\sum_{k=1}^{K}{}' \sum_{j=0}^{N} C_{ij}^k u_j^k = \sum_{k=1}^{K}{}' \sum_{j=0}^{N} B_{ij}^k f_j^k, \tag{2.24}$$

$$\text{or} \qquad \mathbf{Cu} = \mathbf{Bf}, \tag{2.25}$$

$$\text{or} \qquad \mathbf{Au} = \mathbf{f}, \tag{2.26}$$

where

$$C_{ij}^k \quad = \quad \sum_{l=0}^{N} \frac{1}{J_l} \rho_{l,k} D_{li} D_{lj}, \forall i, j \in \{0, \cdots, N\}^2 \tag{2.27}$$

$$B_{ij}^k \quad = \quad J_j \rho_{i,k} \delta_{ij}, \forall i, j \in \{0, \cdots, N\}^2 \tag{2.28}$$

$$D_{ij} \quad = \quad \frac{dh_j}{d\xi}(\xi_i), \forall i, j \in \{0, \cdots, N\}^2 \tag{2.29}$$

$$A \quad = B^{-1}C. \tag{2.30}$$

Here $\sum'$ denotes elemental direct stiffness summation, in which the continuity and boundary conditions imposed on $u_h, v_h \in X_h$, are taken into account: the rows and columns corresponding to the same global degree-of-freedom are summed, and rows corresponding to Dirichlet boundary conditions are eliminated.

Since each basis function is not zero over a single element, the bilinear form $a(h_i, h_j)$ is non-zero only if h_i and h_j belong to the same element. To compute the global matrix equation, only the local elemental matrices are created:

$$\mathbf{A}^k\, \mathbf{u}^k = \mathbf{f}^k. \tag{2.31}$$

At the element level, the matrices $\mathbf{A}^k$ can be split into components containing boundary and interior contributions, that is:

$$\mathbf{A^k} = \begin{bmatrix} \mathbf{A}_{11}^k & \mathbf{A}_{12}^k \\ \\ \mathbf{A}_{21}^k & \mathbf{A}_{22}^k \end{bmatrix},$$

where $\mathbf{A}_{11}^k$ represents the components of $\mathbf{A}^k$ resulting from boundary-boundary mode interactions, $\mathbf{A}_{12}^k$ represents the components of $\mathbf{A^k}$ resulting from coupling between the boundary-interior modes, $\mathbf{A}_{21}^k$ represents the components of $\mathbf{A^k}$ resulting from coupling between the interior-boundary modes and $\mathbf{A_{22}^k}$ represents the components of $\mathbf{A^k}$ resulting from interior-interior mode interaction (see figure 2.1).

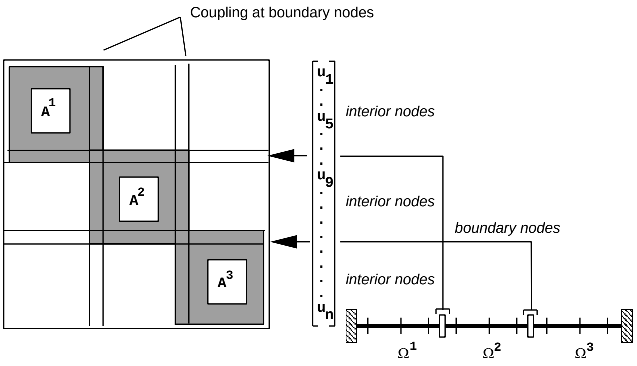


Fig. 2.1: Schematic of the direct stiffness summation of local matrices $\mathbf{A}^k$ to form the global matrix $\mathbf{A}$ [(Henderson and Karniadakis, 1991)].

The global matrix is computed by assembling contributions from the elemental matrices

$$\mathbf{A} = \sum_{k=1}^{K}{}' \mathbf{A}^k. \tag{2.32}$$

($\sum'$ represents ”direct stiffness summation”, see fig 2.1.)

Since $a(\cdot,\cdot)$ is symmetric and positive definite, the matrix $\mathbf{A}$ is symmetric: $\mathbf{A}_{21}^k = [\mathbf{A}_{12}^k]^T$. Also, $\mathbf{A}$ is banded as a result of the use of local basis functions, with all of its non-zero entries located in the N diagonals above the main diagonal. The GLL quadrature and interpolation offer some advantages. One of them is that elements only couple at element boundary nodes, resulting in a simple implementation and sparsity of the matrices. This minimal coupling also serves parallel implementations well, as it translates into a minimum of communication between sub-domains. Another advantage is that the mass matrix $\mathbf{B}$ is diagonal, resulting in rapid evaluation of the right hand side as well as time saving in the context of iterative and time dependent procedures.

2.1.2 Accuracy of the spectral method

The GLL quadrature is exact for any polynomial of degree less than or equal to $2N-1$, $a(u_h, v_h)_{GL}$ will be exact but $(f, v_h)_{GL}$ will not for arbitrary functions f. A theoretical bound for the error $\|u-u_h\|_1$ as $N \longrightarrow \infty$ for fixed K is given (Maday and Patera, 1988) to be

$$\|u-u_h\|_1 \leq C\{N^{1-\sigma}\|u\|_\sigma + N^{1-\rho}\|f\|_\rho\}, \tag{2.33}$$

where $\|\cdot\|_\sigma$ refers to the $\mathcal{H}^\sigma$ norm and C is a constant independent of $h=(N,K)$. This estimate for $u \in \mathcal{H}_0^\sigma(\Omega), f \in \mathcal{H}^\rho(\Omega)$ consists of approximation, interpolation and quadrature errors. This result indicates that if u, f are analytic, the spectral solution u_h converges to the exact solution u as $N \longrightarrow \infty$, K fixed, exponentially fast; that is faster than any order algebraic convergence. Herein lies the advantage of spectral methods over finite element methods which exhibits only algebraic convergence. The variational statement (2.17) is general and may be interpreted differently depending on how the discretization parameter is varied to achieve convergence. Requiring N fixed and K varying to infinity, correspond to the classical h-type finite element method, where low order methods would correspond to low N, typically $N \leq 4$. For $K=1$, N varying to infinity, the method corresponds to a global spectral method. When K is fixed ($K>1$), N varying to infinity, the case corresponds to a spectral element method (SEM) or a p-type finite element method. Varying the number of the elements K, and the polynomial order N, is still possible in the spectral element method using adaptive meshes. To improve the approximation of not infinitely smooth solutions or uniformly varying over the whole domain, a few options for refinement can be used:

1. $K \longrightarrow \infty$, N fixed, refinement by increasing the number of elements K
2. $N \longrightarrow \infty$, K fixed , refinement by increasing the polynomial N order
3. different N in different elements, refinement by changing the polynomial N order

4. changing element boundaries, refinement by moving elements and adjusting their relative size

or a combination of options. The adaptive method considered in this thesis will implement the first refinement strategy. For one-dimensional problems, the first strategy will be combined with a movement of the element boundaries. The flexibility to adapt the mesh to the solution, makes spectral elements methods quite robust. A good adaptive method should be able to simply determine the optimal combination of the four refinement strategies mentioned above to efficiently solve a physical problem.

2.2 Numerical examples of the one-dimensional spectral element method

In this section, we illustrate the performance of SEM and the mesh refinement technique for a few one-dimensional problems.We investigate the refinement by increasing the number of elements, combined with the movement of the elements boundaries by adjusting their relative size. For one-dimensional numerical examples, a common refinement criteria, based on the solution gradients, is used: refine everywhere where solution gradients are large. We require that:

$$\|\nabla u^{(k)}\|_{\mathcal{L}^2(\Omega^k)} \leq \epsilon \|u^h\|_{\mathcal{H}^1(\Omega)} \tag{2.34}$$

everywhere in the mesh, where $\|\cdot\|$ is the $\mathcal{L}^2$ norm, $\|\cdot\|_1$ is the $\mathcal{H}^1$ norm, and ϵ is the discretization tolerance. The criterion for moving elements relies on comparing neighbouring elemental solution gradients. Once the elements have been marked for refinement, if the gradients per element level are larger than an imposed tolerance, the refinement process starts. For refinement by increasing the number of elements K, the element to be refined is simply split in two. For refinement by moving elements, the elements with large relative errors with their neighbours are shrunk in size according to these errors.

In order to keep a uniform approach to one and two dimensional problems, presented in this thesis, we consider that a mesh is conforming, in one-dimensional case, if the mesh is uniform (the elements are equal-sized) and non-conforming if the mesh is not-uniform (the elements are not equal-sized). Since an adaption is based on repositioning refinement, or on creating new elements, a one-dimensional conforming mesh becomes, in general, a non-conforming mesh after adaption.

2.2.1 One-dimensional linear steady advection-diffusion with source term problem

The first test case we consider is the one-dimensional linear steady advection-diffusion problem with source term (one-dimensional steady cosine hill). Advection-diffusion in steady state is interesting in case of a spatially distributed source. The model problem we consider here (Vreugdenhil and Koren, 1993) can be described in a dimensionless form as:

$$u\frac{\partial c}{\partial x} - D\frac{\partial^2 c}{\partial x^2} = S(x)\ ,$$

$$S(x) = \frac{u\pi}{b-a}\sin 2\pi\frac{x-a}{b-a} - \frac{2\pi^2 D}{(b-a)^2}\cos 2\pi\frac{x-a}{b-a} \qquad (\ a \leq x \leq b)\ ,$$

$$S(x) = 0 \quad \text{elsewhere} \quad (\, x < a \ \text{ and } \ x > b) \,.$$

The exact solution for the source function we consider is

$$c(x) = \frac{1}{2}(1 - \cos 2\pi \frac{x-a}{b-a}) \quad (\, a \leq x \leq b),$$

$$c(x) = 0 \quad \text{elsewhere} \quad (\, x < a \ \text{ and } \ x > b\,) \,.$$

In our example, we solve the equation for

$$a = 0.2, b = 0.6, u = 1, D = 0.01.$$

The boundary conditions are for $x = 0$ and $x = 1$: $c(x) = 0$. Note that the source function, illustrated in figure 2.2 (a), is discontinuous at both $x = a$ and $x = b$ if $D \neq 0$. The exact solution to this problem in not smooth since the second derivative is discontinuous in $x = a$ and $x = b$. For this problem no exponential accuracy is achieved due to the discontinuities in the second derivatives. To improve the accuracy, we use adaption based on the solution gradients combined with the movement of the elements boundaries. Figure (2.3) illustrates the effect of adaption keeping the polynomial order and the number of elements order constant: $N = 14, K = 16$. We start the refinement using an initial grid that consists of four equally spaced elements, and we impose that the maximum number of elements that are created during the refinement equals $K = 16$, and the refinement tolerance is set $\epsilon \ = \ 1.0 \times 10^{-03}$. Figure 2.3 (a) shows the effect of the uniform refinement, where each element is split up in two children elements. Imposing the tolerance on the solution gradients, we now combine h-refinement with relocation of the element boundaries.
In figure 2.3 (b), we can see that moving the element boundaries influences the accuracy of the solution. The discrete minimum error is obtained for $N = 14, K = 16$, $\|c - c_h\|_{\infty,GL} = 1.72 \times 10^{-5}$, where the subscript $\|_{\infty,GL}$ means that the maximum error is evaluated in the GLL points. Despite of clustering of small elements near the sharp gradients, the error remains large.

2.2.2 One-dimensional steady Gaussian hill problem

We consider again the steady advection-diffusion problem as described by the first test case. However, in this case the source term, figure 2.2(b), is chosen such the exact solution to this problem is the Gaussian hill

$$c(x) = e^{-\dfrac{(x-x_0)^2}{2\sigma^2}}, \quad \text{with } \sigma = 0.04 \text{ and } x_0 = 0.4.$$

Since this problem has a well-defined (*infinitely smooth*) solution, we begin by computing the true error $\|c - c_h\|_{\infty,GL}$ on a uniformly refined grid (table 2.1). This table shows that

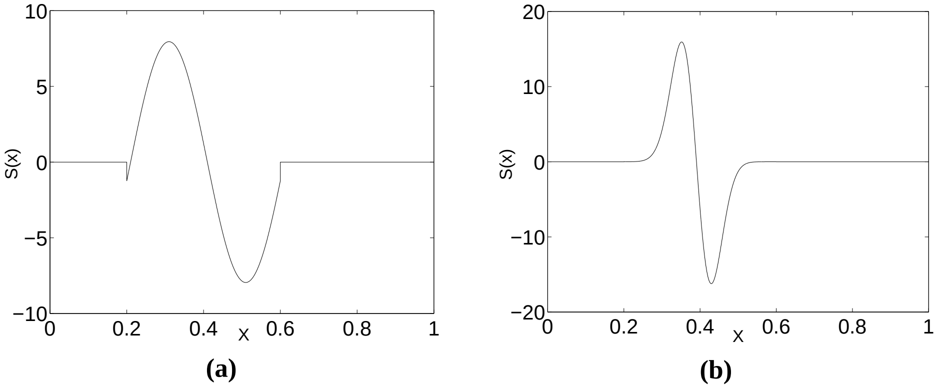
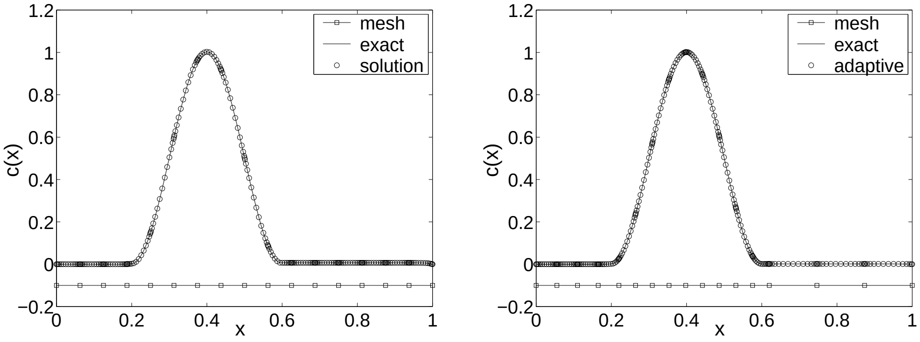


Fig. 2.2: Source functions for the one-dimensional linear steady advection-diffuse problem (a) and one-dimensional steady Gaussian hill problem (b).



(a) Uniform:
$N = 14, K = 16,$
$\|c - c_h\|_{\infty,GL} = 7.77 \times 10^{-03}$

(b) Adaptive:
$N = 14, K = 16,$
$\|c - c_h\|_{\infty,GL} = 1.72 \times 10^{-05}$

Fig. 2.3: Effect of refinement by moving the elements boundaries for the one-dimensional steady cosine hill. The tolerance $\epsilon = 1.0 \times 10^{-03}$.

the spectral convergence is obtained. In figure 2.4(a), the discrete maximum error $\epsilon = \|c - c_h\|_{\infty,GL}$ is plotted versus the degrees of freedom (*dofs*) obtained by increasing the degree of approximation N and the number of the elements K. For example: imposing a certain error for the approximation, we can determine how many *dofs* we need to obtain that error. Let us consider that we want the discrete maximum error to be $\epsilon = 1.0 \times 10^{-10}$. From figure 2.4(a), we can see that this error is achieved in three cases:

1. N=16 *with about 130* dofs

2. N=14 *with about 150* dofs

3. N=12 *with about 170* dofs.

Table 2.2 shows the errors for solving the one-dimensional steady Gaussian Hill problem using the adaption based on solution gradients. We see, by keeping the same number of elements K and adapting the polynomial order N, a better approximation for the solution is obtained. In figure 2.4(b), we illustrate the dependence of the error on the number of *dofs*. As in the uniform case, we impose the same error $\epsilon = 1.0 \times 10^{-10}$ to approximate the solution of the problem. In this case the number of *dofs* we need to achieve the imposed error is less than in the uniform case:

1. N=16 *with about 100* dofs

2. N=14 *with about 120* dofs

3. N=12 *with about 150* dofs.

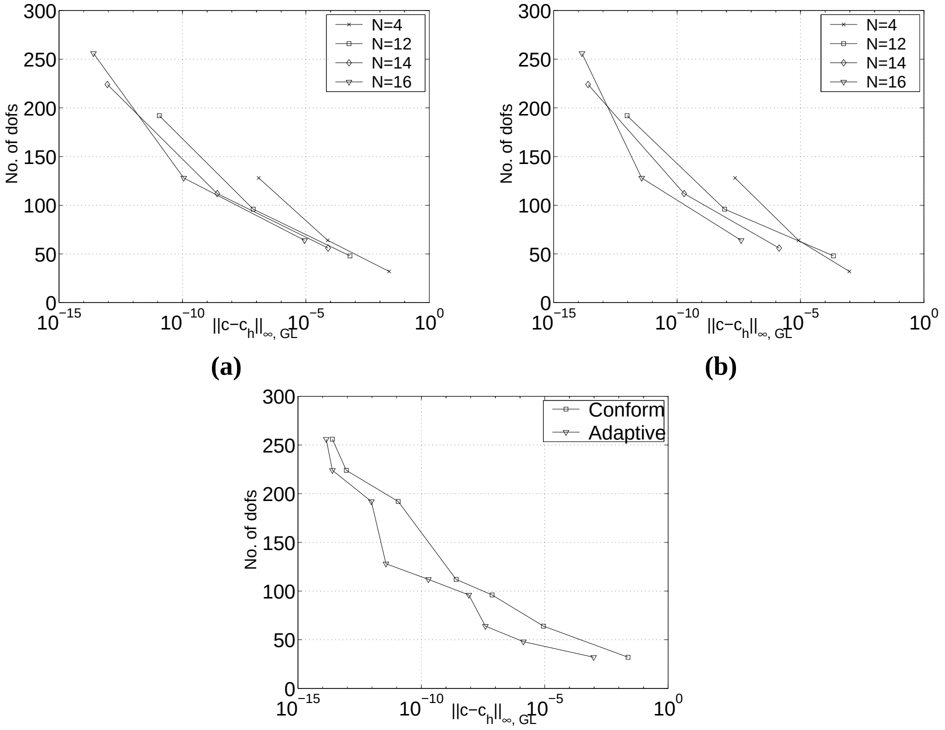
From tables 2.1 and 2.2 we choose the best approximation of the solution and plot the errors versus the number of *dofs* in both cases: *uniform* and *adaptive*. This is illustrated in figure 2.4(c). For all N, K combinations, we see that the best approximation of the solution is achieved in the adaptive case using less *dofs* than in the uniform case.

Table 2.1. Errors ($\|c - c_h\|_{\infty,GL}$) for solving the steady Gaussian hill, in the uniform case for polynomial orders N=$\{$8,12,14,16$\}$ elements:

K	N=8		N=12		N=14		N=16	
Elements	Error	dofs	Error	dofs	Error	dofs	Error	dofs
4	2.38E-02	33	6.14E-04	49	7.93E-05	56	8.82E-06	65
8	7.68E-05	65	7.36E-08	97	2.57E-09	113	1.12E-10	129
16	1.23E-07	129	1.15E-11	193	9.07E-14	225	2.48E-14	257

Table 2.2. Errors ($||c - c_h||_{\infty,GL}$) for solving the steady Gaussian hill, in the refined case for polynomial orders N=$\{$8,12,14,16$\}$ elements:

K	N=8		N=12		N=14		N=16	
Elements	Error	dofs	Error	dofs	Error	dofs	Error	dofs
4	9.57E-04	33	2.17E-04	49	1.36E-06	56	3.93E-08	65
8	8.37E-06	65	8.38E-09	97	1.93E-10	113	3.67E-12	129
16	2.22E-08	129	9.44E-13	193	2.48E-14	225	1.40E-14	257

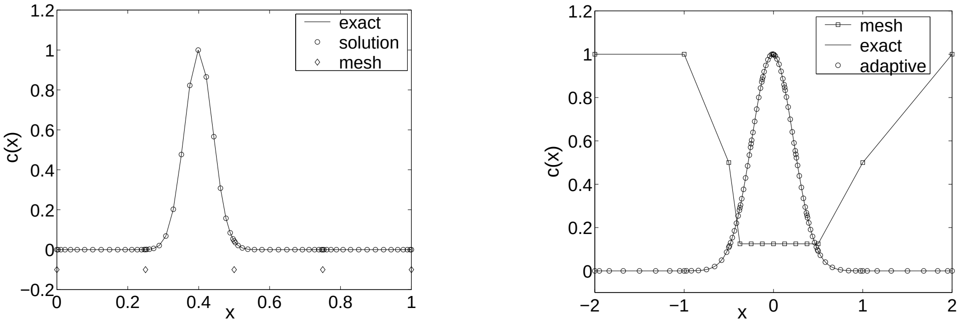


(a) **(b)**

(c)

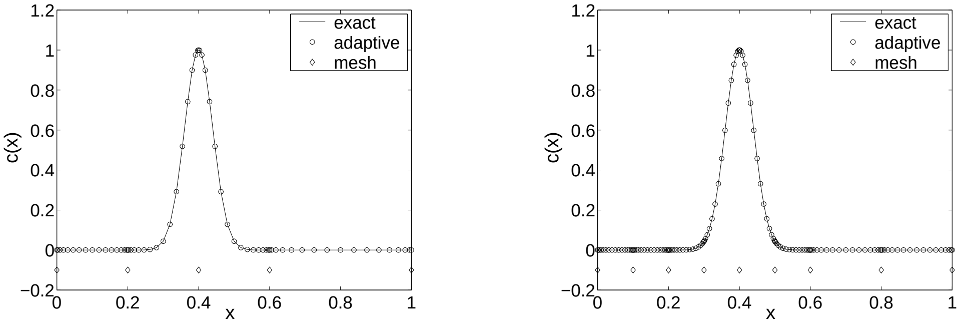
Fig. 2.4: The number of dofs for solving the one-dimensional steady Gaussian hill as a function of the required error:(a) uniform, (b) adaptive, (c) comparison between uniform and adaptive.

In figure 2.5 we illustrate the effect of the refinement by adaptively increasing the number of elements combined with a movement/relocation of the elements boundaries. The refinement starts with an initial grid of four equally spaced elements. Based on the solution gradients, we refine the grid and move the elements boundaries, trying to minimize the discrete error of the solution. When the imposed number of elements is reached, the refinement process stops. In this way, we can compare the errors computed in the uniform case, with the errors obtained in the adaptive case. On the left side of figure 2.5(a, c) we represent the uniform case N=16, K=$\{$4, 8$\}$. After refinement, the maximum errors are two order of magnitude smaller than in the uniform case (figure 2.5 b, d).



(a) Uniform:
$N = 16, K = 4,$
$\|c - c_h\|_{\infty,GL} = 8.82 \times 10^{-06}$

(b) Adaptive:
$N = 16, K = 4,$
$\|c - c_h\|_{\infty,GL} = 3.93 \times 10^{-08}$



(c) Uniform:
$N = 16, K = 8,$
$\|c - c_h\|_{\infty,GL} = 1.12 \times 10^{-10}$

(d) Adaptive:
$N = 16, K = 8,$
$\|c - c_h\|_{\infty,GL} = 3.67 \times 10^{-12}$

Fig. 2.5: Effect of refinement by adaptively increasing the number of elements and by moving the elements boundaries for the one-dimensional steady Gaussian hill.

2.2.3 One-dimensional linear unsteady advection problem

The last test case we consider in this chapter is the one-dimensional unsteady advection equation problem:

$$\frac{\partial c}{\partial t} + u\frac{\partial c}{\partial x} = 0,\ \text{-2} \leq \text{x} \leq \text{2, and } t \in [0,1],$$

where the velocity u is constant and equals $u = 1$. The initial condition is a Gaussian distribution:

$$c(x,0) = 0.01^{4r^2}, \quad r = \sqrt{(x+\frac{1}{2})^2}.$$

The exact solution is:

$$c(x,t) = 0.01^{4r^2}, \quad r = \sqrt{(x-t+\frac{1}{2})^2}.$$

We use a Crank-Nicolson scheme to solve the one-dimensional unsteady advection, which is unconditionally stable and second-order accurate in time. Table 2.3 shows the errors for solving the unsteady equation on a uniform grid (K=4, 8, 16, 32). For K=4, the errors due to the spatial resolution are dominant and we do not achieve the good solution accuracy in time. The same effect appears when the time step is kept constant and the polynomial order of the elements changed. In this case the errors due to the time discretization are dominant. We can improve the accuracy by refining the elements, based on the solution gradients.

Table 2.3. Errors ($\|c - c_h\|_{\infty,GL}$) for solving the one-dimensional unsteady Gaussian hill, in the uniform case for polynomial order $N = 8$ elements:

Time Steps	N=8, K=4	N=8, K=8	N=8, K=16	N=8, K=32
64	1.31E-01	6.12E-03	6.28E-03	6.28E-03
128	6.04E-02	1.56E-03	1.56E-03	1.56E-03
256	6.04E-02	9.32E-04	3.91E-04	3.91E-04

Table 2.4 shows the improvements in accuracy due to mesh refinement. Each time cycle starts with an initial number of elements represented by K_i. After refinement, the number of elements per time cycle is averaged to obtain the average number of elements used K_a. Even in this case the time discretization errors are very large. To eliminate them and prove that we get more accuracy in space, we consider *a modified exact solution*. From table 2.3, it is observed that the accuracy is not improving anymore, even when the time step = 256 and $N = 8, K = 32$. The same error $\|c - c_h\|_{\infty,GL} = 3.91 \times 10^{-04}$ is obtained when we solve the problem for $N = 16$ and $K = 32$. This is an indication that the error due to the time discretization is dominant. We compute the *modified exact solution* c_{mex} solving the

one-dimensional unsteady advection equation problem for $N = 20,\ K = 64$ and 2048 time steps. Computing the norm $\|c_{mex} - c_h\|_{\infty,GL}$ we are able to eliminate the error due the time discretization. The improvement in accuracy for a 64 time steps cycle is illustrated in the table 2.5.

Table 2.4. Errors ($\|c - c_h\|_{\infty,GL}$) for solving the one-dimensional unsteady Gaussian hill, in the non-uniform case for polynomial order $N = 8$ elements and an error tolerance $\epsilon = 0.01$ for the solution gradients:

Time Step	N=8, K_i=4	N=8, K_i=8	N=8, K_i=16
64	5.73E-03, K_a=20	1.3E-04, K_a=20	6.28E-05, K_a=20
128	2.58E-03, K_a=19	3.98E-05, K_a=19	1.56E-05, K_a=19
256	1.21E-03, K_a=18	1.76E-05, K_a=18	2.46E-06, K_a=18

consciously

Table 2.5. Errors ($\|c_{mex} - c_h\|_{\infty,GL}$) for solving the one-dimensional unsteady Gaussian hill, in the non-uniform case for polynomial order N=8 elements and an error tolerance $\epsilon = 0.01$ for the solution gradients:

Time Step	N=8, K_i=4	N=8, K_i=8	N=8, K_i=16
64	1.02E-04, K_a=20	7.65E-07, K_a=20	3.56E-08, K_a=20

A comparison between the relative errors in solving the one-dimensional linear unsteady advection problem, is illustrated in figure 2.6. For different time steps, there is an improvement of the solution accuracy due to refinement. After refinement, the errors are two orders of magnitude smaller than in the uniform case.

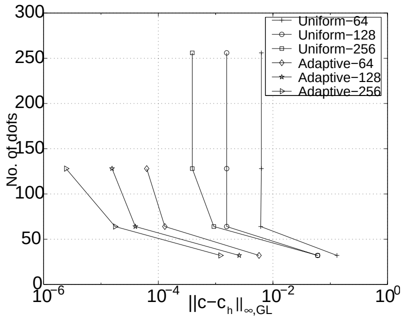
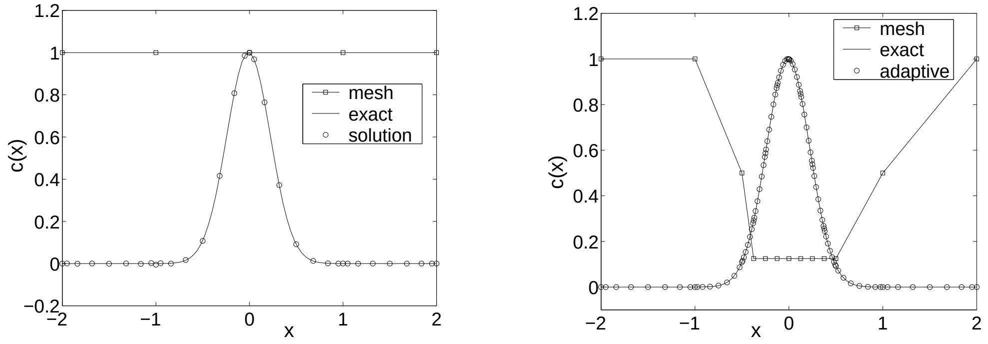


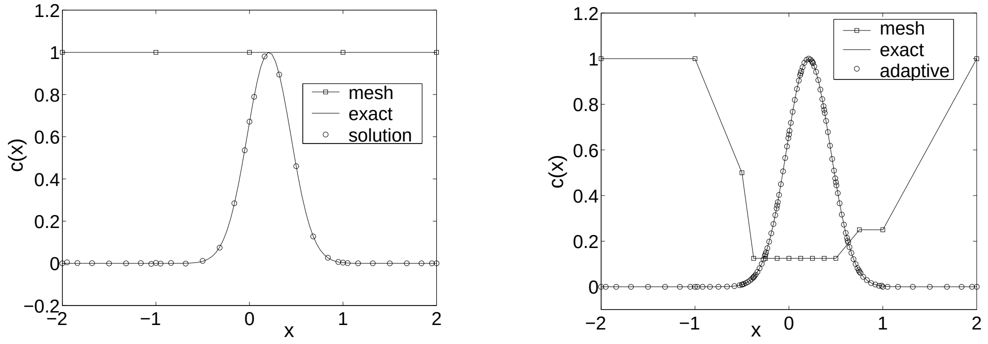
Fig. 2.6: The number of dofs for solving the one-dimensional linear unsteady advection problem, as a function of the required error for one-dimensional linear unsteady advection problem: comparison between uniform and adaptive , for $N = 8$ and different time steps: 64, 128, 256.

In figures 2.7, 2.8 and 2.9 is illustrated the accuracy of the solution in both, the uniform (left) and the non-uniform (right) cases for different time steps. The element size, the exact solution and the adapted solution are shown. Imposing different tolerances on the solution gradients, and choosing different time steps, the number of elements vary per simulation. In our example, we generate roughly the same number of elements ($K \approx 20$) using a uniform basis of order $N = 8$ and $N = 16$. The elements are generated around the Gaussian hill, and they ”move” together with it. The elements that are not in the vicinity of the hill are coarsened. We can conclude that in all time cycles an improvement in solution accuracy is achieved, due to the refinement.



(a) Uniform: time=0.25, steps=256,
$N = 8, K = 4$,
$\|c - c_h\|_{\infty,GL} = 4.21 \times 10^{-03}$

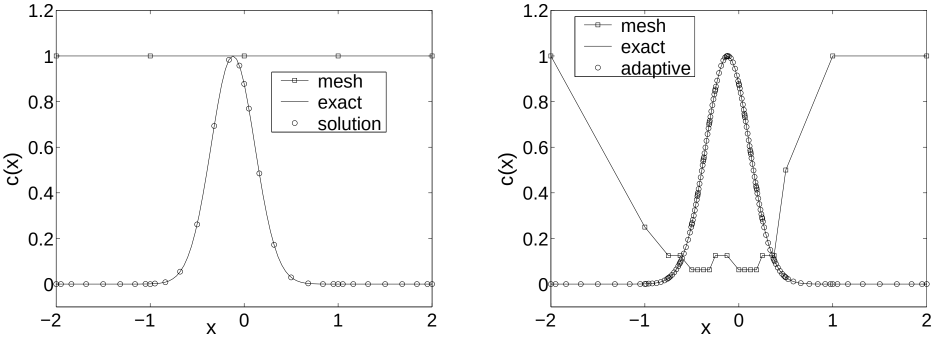
(b) Adaptive: time=0.25, steps=256,
$N = 8, K = 12$,
$\|c - c_h\|_{\infty,GL} = 1.19 \times 10^{-05}$



(c) Uniform: time=0.75, steps=256,
$N = 8, K = 4$,
$\|c - c_h\|_{\infty,GL} = 5.91 \times 10^{-02}$

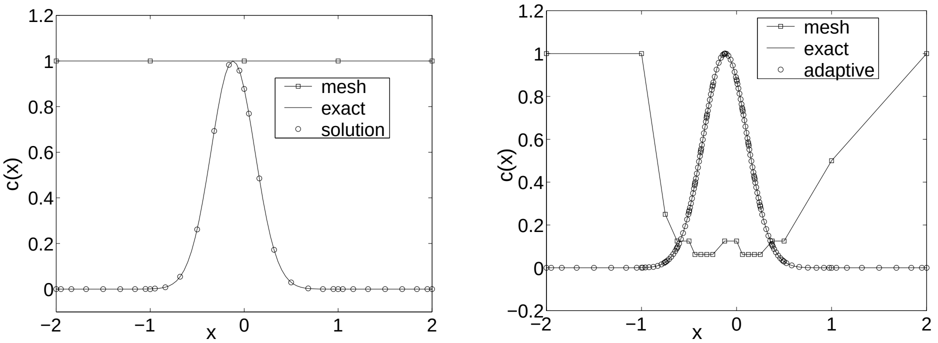
(d) Adaptive: time=0.75, steps=256,
$N = 8, K = 13$,
$\|c - c_h\|_{\infty,GL} = 3.15 \times 10^{-05}$

Fig. 2.7: Effect of refinement by adaptively increasing the number of elements for the one-dimensional unsteady Gaussian hill, based on the solution gradients with a tolerance of $\epsilon = 0.04$.



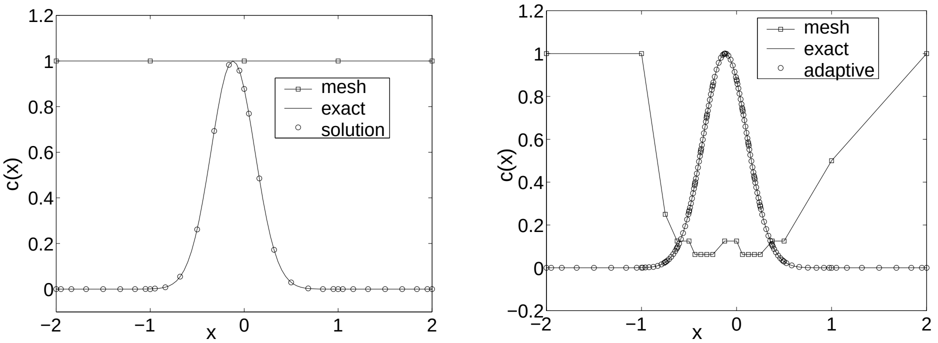
(a) Uniform: time=0.35, steps=64, $N = 8, K = 4$, $\|c - c_h\|_{\infty,GL} = 1.14 \times 10^{-01}$

(b) Adaptive: time=0.35, steps=64, $N = 8, K = 19$, $\|c - c_h\|_{\infty,GL} = 1.05 \times 10^{-04}$



(c) Uniform: time=0.35, steps=128, $N = 8, K = 4$, $\|c - c_h\|_{\infty,GL} = 1.22 \times 10^{-02}$

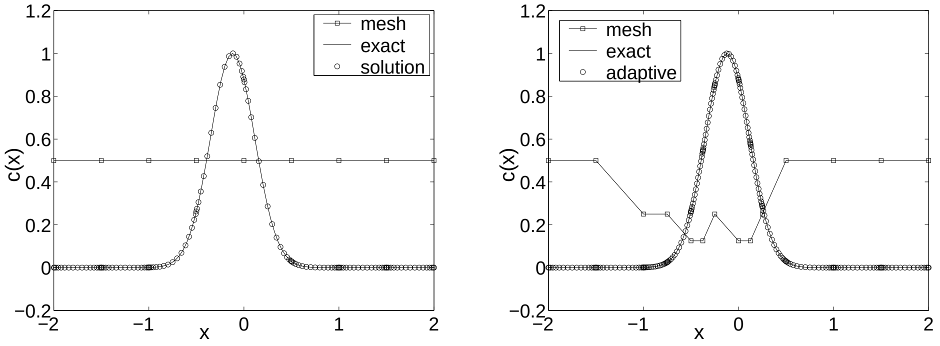
(d) Adaptive: time=0.35, steps=128, $N = 8, K = 18$, $\|c - c_h\|_{\infty,GL} = 2.48 \times 10^{-05}$



(e) Uniform: time=0.35, steps=256, $N = 8, K = 4$, $\|c - c_h\|_{\infty,GL} = 1.21 \times 10^{-02}$

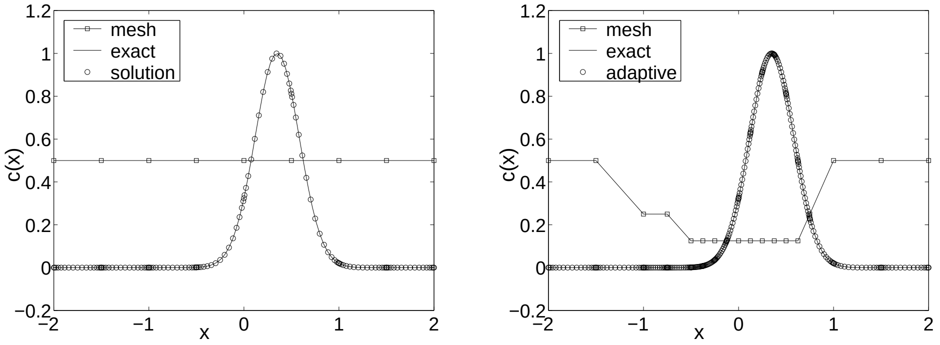
(f) Adaptive: time=0.35, steps=256, $N = 8, K = 18$, $\|c - c_h\|_{\infty,GL} = 1.01 \times 10^{-05}$

Fig. 2.8: Effect of refinement by adapting increasing the number of elements for the one-dimensional unsteady Gaussian hill based on the solution gradients with a tolerance of $\epsilon = 0.01$.



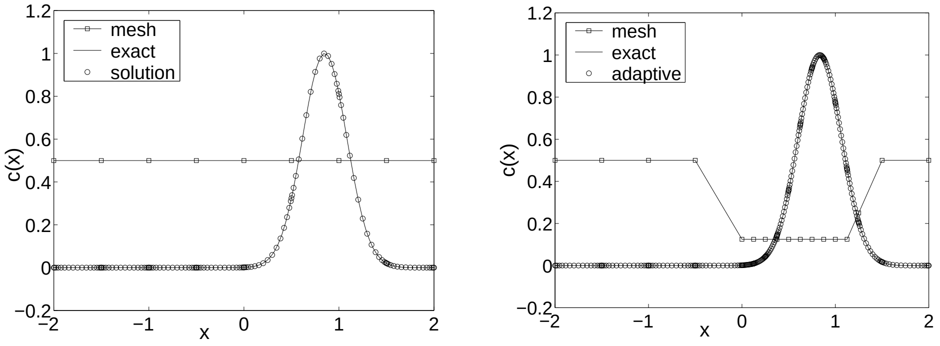
(a) Uniform: time=0.35, steps=64, $N = 16, K = 8$, $\|c - c_h\|_{\infty,GL} = 1.05 \times 10^{-03}$

(b) Adaptive: time=0.35, steps=64, $N = 16, K = 14$, $\|c - c_h\|_{\infty,GL} = 1.01 \times 10^{-05}$



(c) Uniform: time=0.50, steps=64, $N = 16, K = 8$, $\|c - c_h\|_{\infty,GL} = 1.11 \times 10^{-03}$

(d) Adaptive: time=0.50, steps=64, $N = 16, K = 18$, $\|c - c_h\|_{\infty,GL} = 1.48 \times 10^{-05}$



(e) Uniform: time=1.0, steps=64, $N = 16, K = 8$, $\|c - c_h\|_{\infty,GL} = 2.20 \times 10^{-03}$

(f) Adaptive: time=1.0, steps=64, $N = 16, K = 17$, $\|c - c_h\|_{\infty,GL} = 2.18 \times 10^{-05}$

Fig. 2.9: Effect of refinement by adapting increasing the number of elements for the one-dimensional unsteady Gaussian hill based on the solution gradients with a tolerance of $\epsilon = 0.04$.

2.3 Conclusions

The test cases presented clearly show that the adaptive formulation of the spectral element method increases the flexibility and capability of the method. Sharp gradients and regions of poor resolution can be solved optimally. Any refinement in the pure spectral element convergence context, namely K fixed, $N \longrightarrow \infty$, improves the solution's accuracy but at a slow rate. If the boundary layers were internal, as opposed to on the boundary, convergence would deteriorate rapidly, unless we know where the boundary layer was and could place elements on either side of it. A simple repositioning of the element boundaries (or equivalently a change in their size), however, results in an error two orders of magnitude lower for the same discretization parameter K. First, in practical calculations the spectral element method is not restricted to keep the number of elements K fixed and increasing N towards infinity. There is much more flexibility in the method due to the discretization parameter $h = (N, K)$ depending on two parameters. Second, we need to develop criteria and schemes to efficiently exploit this flexibility. The refinement criteria will be improved by introducing, in the next chapters, new *error estimators* which will indicate the quality of the resolution on each element. The test cases have shown that we can use many elements in regions of discontinuities or sharp structure. The refinement capabilities are very successful, but we have seen that the coarsening algorithm remains somewhat inefficient. We will address this problem again for the two dimensional problems.

Chapter 3

Mortar Element Method

In this chapter the fundamentals of the mortar element method are introduced, including theory and implementation. Single-mesh *a posteriori* error estimators for a spectral element solution u_h of a general partial differential equation $\mathcal{L}(u) = f$ will be presented. Several numerical examples are presented and analyzed, mainly based on work of Greengard and Lee (1996); Mavriplis (1989); Henderson and Karniadakis (1991).

3.1 Introduction

Mortar element methods were first introduced by Bernardi, Maday and Patera in Bernardi *et al.* (1994) for low-order and spectral/finite elements. Mortar spectral elements are non-conforming spectral elements that allow a non-conforming decomposition of the computational domain into sub-regions. With respect to accuracy, the optimal coupling of different variational approximations in different sub-regions is achieved. Since the method is non-conforming, the discrete space is not embedded in a continuous functional space suited to the numerical analysis of a given partial differential equation. However, the *'variational crime'* (Anagnostou *et al.*, 1989) committed in this strategy, does not pollute the accuracy of the original spectral method; it leads to more flexible discretizations and a better use of the discretization parameter h. The mortar element method preserves the element-based locality, distinguishing it from other, more global techniques such as functional minimization with Lagrange multiplier constrains (Dors, 1989). Another attractive property of the mortar element method is its good scalability characteristics on parallel computers due to dense computational kernels and sparse communication requirements. One of the main features of the mortar element method is that the order of the interpolation polynomial N and the number of elements K, can be adjusted to suit the problem at hand. In the next sections the fundamentals of the mortar element method are introduced.

3.2 Spectral element methods for two-dimensional problems

In Chapter 2, we introduced the basic operations of the one-dimensional spectral element method. In this section we present the procedure used to derive the spectral element method for two-dimensional problems. A key to the efficiency of high-order methods in higher dimensional problems is the formulation of a basis from the tensor product of one-dimensional basis functions. We consider the two-dimensional Poisson equation on a domain $\mathbf{\Omega} \in \mathbb{R}^2$, with homogeneous Dirichlet boundary conditions:

Problem 1: Find $u \in \mathcal{C}^2(\boldsymbol{\Omega}) \cap \mathcal{C}^1(\overline{\boldsymbol{\Omega}})$ such that

$$-\nabla^2 u = f \text{ in } \boldsymbol{\Omega}, \text{ and } u = 0 \text{ on } \partial\boldsymbol{\Omega}. \tag{3.1}$$

After integrating by parts and applying the boundary conditions, the variational statement for the problem 1 is:

Problem 2: Find $u \in \mathbf{X} = \mathcal{H}_0^1(\boldsymbol{\Omega})$ such that

$$\int_{\boldsymbol{\Omega}} \nabla u \cdot \nabla v \, d\mathbf{x} = \int_{\boldsymbol{\Omega}} f v \, d\mathbf{x}, \;\; \forall v \;\in\; \mathbf{X} \tag{3.2}$$

or

$$a(u, v) = (f, v) \;\; \forall v \;\in\; \mathbf{X}, \tag{3.3}$$

where a point in $\boldsymbol{\Omega}$ is denoted $\mathbf{x} = (x, y)$ and the bilinear form $a(\cdot, \cdot)$ is given by:

$$a(u, v) = \int_{\boldsymbol{\Omega}} \nabla u \cdot \nabla v \, d\mathbf{x}, \; u, v \in \mathbf{X}. \tag{3.4}$$

The spectral element discretization proceeds by breaking up the domain $\boldsymbol{\Omega}$ into K rectangular elements $\boldsymbol{\Omega}^k$. In conforming spectral element methods, the domain decomposition satisfies the constraint that the intersection of two adjacent elements is either an entire edge or a vertex and the order of approximation is equal for adjacent elements. Relaxing this constraint is the subject of the next section.

First, we require that the variational statement, equation (3.2), is satisfied for a piecewise polynomial subspace of $\mathcal{H}_0^1(\boldsymbol{\Omega})$. As in the one-dimensional case, we first define the space

$$\mathbf{P}_{N,K}(\boldsymbol{\Omega}) = \{\Phi \in \mathcal{L}^2(\boldsymbol{\Omega}); \Phi_{|\Omega^k} \in \mathbb{P}_N(\Omega^k) \times \mathbb{P}_N(\Omega^k)\}, \tag{3.5}$$

where $P_N(\boldsymbol{\Omega}^k)$ denotes the space of all polynomials of a degree less than or equal to N with respect to each variable $x, \; y$ on each subdomain $\boldsymbol{\Omega}^k$, and $\mathbf{P}_{N,K}(\boldsymbol{\Omega})$ is the tensor product space corresponding to (2.11). The spectral element space $\mathbf{X}_h$ consist of

$$\mathbf{X}_h = \mathcal{H}_0^1(\boldsymbol{\Omega}) \cap \mathbf{P}_{N,K}(\boldsymbol{\Omega}). \tag{3.6}$$

In two-dimensional problems, we can map a general curvilinear element to the standard element as shown in figure 3.1. The curved element must still verify the constrains imposed for the conforming case: the intersection of two adjacent elements is either an entire edge or a vertex and the interpolation points coincide. We may use arbitrary local elemental mappings:

$$(x, y)_h^k = (x, y)_{ij}^k h_i(\xi) h_j(\eta) \tag{3.7}$$

to map physical curved elements (x, y) onto the square computational domain (ξ, η) where $(x, y) \in \boldsymbol{\Omega}$ and $(\xi, \eta) \in \Lambda^2$.
The discrete problem is then given by:

Problem 3: Find $u_h \in X_h$ such that

$$\int_{\Omega} \nabla u_h \cdot \nabla v_h \, d\mathbf{x} = \int_{\Omega} f v_h \, d\mathbf{x}, \quad \forall v_h \;\in\; X_h. \tag{3.8}$$

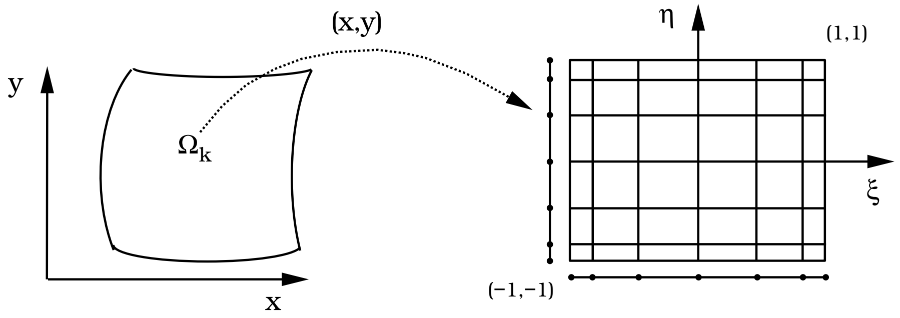


Fig. 3.1: Isoparametric mapping from physical curved geometry (x,y) to the computational rectangular grid (ξ, η) for a polynomial order $N = 6$.

We use an affine mapping from (x^k, y^k) to $(\xi, \eta) \in \Lambda^2 = [-1,1] \times [-1,1]$ on each element k, and denote the Jacobian by:

$$J^k = \frac{\partial x^k}{\partial \xi}\frac{\partial y^k}{\partial \eta} - \frac{\partial x^k}{\partial \eta}\frac{\partial y^k}{\partial \xi}. \tag{3.9}$$

Performing GLL quadrature in (ξ, η) yields:

$$\sum_{k=1}^{K}{}' \frac{1}{|J^k|}\rho_i\rho_j \nabla u_h^k(\xi_i,\eta_j)\cdot\nabla v_h^k(\xi_i,\eta_j) = \sum_{k=1}^{K}{}' |J^k|\rho_i\rho_j f^k(\xi_i,\eta_j) v_h^k(\xi_i,\eta_j),\ \forall\, v_h \in \mathbf{X}_h \tag{3.10}$$

or

$$\sum_{k=1}^{K} a(u_h, v_h)_{GL} = \sum_{k=1}^{K} (f, v_h)_{GL}\ \ \forall v_h\ \in\ \mathbf{X}_h, \tag{3.11}$$

where ξ_i, η_j are the Gauss-Lobatto collocation points in the x, y directions respectively, and ρ_i, ρ_j the Gauss-Lobatto weights. The notation $\sum'$ denotes the direct stiffness summation between elements, in which continuity (physically coincident node contributions are summed) and boundary conditions (boundary node contributions are set to zero) on u_h and $v_h\ \in\ \mathbf{X}_h$ are taken into account. To complete the discretization, a basis must be chosen. The basis for $u_h\ \in\ \mathbf{X}_h$ then follows naturally from the one-dimensional case as:

$$u_h^k(x,y) = u_{ij}^k h_i(\xi) h_j(\eta),\ x, y \Longrightarrow \xi, \eta, \tag{3.12}$$

where the h_i and h_j are the one-dimensional Lagrangian interpolants defined in (2.23) and $u_{ij}^k = u_h^k(\xi_i, \eta_j)$.

Inserting (3.12) in (3.10) the tensor product form is retained and the resulting discrete equations can therefore efficiently be solved. We induce extra quadrature errors in the evaluation of both $a(u_h, v_h)_{GL}$ and $(f, v_h)_{GL}$; however, for smooth solutions and boundaries they are roughly of the same order as the approximation and interpolation errors (Rønquist, 1988).

As in the one-dimensional case we also require the polynomials to be $\mathcal{C}^0$ continuous across elemental boundaries and enforce homogeneous Dirichlet boundary conditions for

$u_h \in \mathbf{X}_h$. Then, we express the discrete solution $u_h \in \mathbf{X}_h$, the test functions $v_h \in \mathbf{X}_h$ and the prescribed force $f_h \in \mathbf{Y}_h$ in terms of the basis (3.12) and by choosing v_h to be nonzero at only one global collocation point, the discrete formulation (3.10) becomes

$$\sum_{k=1}^{K}{}' \sum_{m,n=0}^{N} (C^k_{im}B^k_{jn}+B^k_{im}C^k_{jn})u^k_{mn} = \sum_{k=1}^{K}{}' \sum_{m,n=0}^{N} B^k_{im}B^k_{jn}f^k_{mn} \qquad \forall i,j \in \{0,N\}^2 \tag{3.13}$$

or

$$\mathbf{Cu} = \mathbf{Bf}, \tag{3.14}$$

or

$$\mathbf{Au} = \mathbf{f}, \tag{3.15}$$

where C^k_{im}, B^k_{jn} are the matrices of the one-dimensional case (2.27, 2.28, 2.29) and $\mathbf{A} = \mathbf{B}^{-1}\mathbf{C}$.

There are two differences from the one-dimensional case that are worthwhile to be mentioned. First, by using the properties of GLL quadrature, the equivalence of (3.11) to a collocation procedure for points internal to an element can be obtained. The interface between the elements is more complex than in the one-dimensional case. The weak $\mathcal{C}^1$ condition, at element boundary points, is naturally generated by the variational approach.

The second observation is that the spectral method formulation for elliptic problems is directly compatible with h-type finite element methods, in that the projection operators and continuity are the same for both methods. This makes it possible to mix low-order with high-order elements in the same problem. For instance, low-order elements can be used in regions of the domain where the solution varies very slowly or near singularities. Since in that case the collocation points do not coincide on the internal elemental boundaries, at the interface of the two elements a projection is needed in order to make the connection between one elemental space and the other. This is addressed by the non-conforming formulation in the next section.

3.3 Mortar element basic concepts

The mortar formulation presented here is based on the approach presented in Maday *et al.* (1989); Bernardi *et al.* (1990); Levin *et al.* (2000); Anagnostou *et al.* (1989).

In our final implementation, the spectral expansions are the same in all elements and a non-conforming grid is obtained by h-refinement. This is a particular case of the general formulation presented in Levin *et al.* (2000) in which a two-dimensional mortar element formulation is introduced for geometrically non-conforming grids, where elements are allowed to have different spectral expansions. In the formulation presented in (Maday *et al.*, 1989; Bernardi *et al.*, 1990), the spectral expansions are the same in all elements and a non-conforming grid is obtained by *h*-refinement. Our formulation is designed for the general case: geometrically non-conforming grids where elements are allowed to have different spectral expansions. Due to this generality, a different formulation of a projection operator between interface and non-conforming element edges is obtained. In this section, we will first introduce a general two-dimensional mortar element formulation based on Levin *et al.* (2000) and then the particular case for *h*-refinement will be derived.

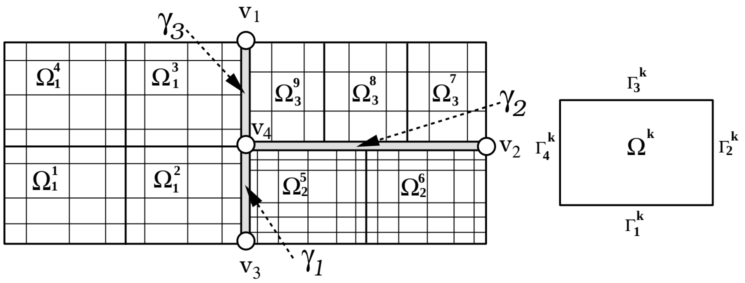


Fig. 3.2: An illustration of the non-conforming grid. The element edges are shown with thicker lines and the grid lines with thinner lines. The v_i represents the mortar endpoints and γ_i are the mortars. Each element k has four edges : $\Gamma_l^k, l = \{1, 2, 3, 4\}$. Mortars γ_i are represented by double thick lines.

The non-conforming formulation consists of a set of conforming spaces for the regions that permit a conforming formulation, a space of functions on a mortar interface and a space used in the projection between the mortar spaces and the conforming spaces.

Before we introduce the formulation of the two-dimensional mortar method, the basic concept of the method can be sketched as follows. We consider the domain $\mathbf{\Omega}$ on $\mathbb{R}^2$ such that $\mathbf{\Omega}$ consist of several subregions $\mathbf{\Omega}_p,\ p = \{1, ..., P\}$ and in each of $\mathbf{\Omega}_p$, the spectral grid is conforming. In figure 3.2 we illustrate this concept. The domain $\mathbf{\Omega}$ is covered completely by the union $\mathbf{\Omega}_p$: $\overline{\mathbf{\Omega}} = \bigcup_{p=1}^{P} \overline{\mathbf{\Omega}}_p$. The $\mathbf{\Omega}_p$ intersects with each other only along a collection of one-dimensional element edges. The union of these edges defines the skeleton S of the domain which is decomposed into mortars. These mortars are one-dimensional geometrical entities for two-dimensional problems. We require the mortars to coincide with a complete edge or with a union of edges. The choice of mortars is not unique. In the example, shown in figure 3.2, the domain $\mathbf{\Omega}$ has three subregions:

$$\mathbf{\Omega} = \mathbf{\Omega}_1 \cup \mathbf{\Omega}_2 \cup \mathbf{\Omega}_3, \tag{3.16}$$

where each subregion $\mathbf{\Omega}_p$ is covered by a collection of conforming spectral elements. Subregion $\mathbf{\Omega}_1$ has four elements, $\mathbf{\Omega}_2$ has two elements and $\mathbf{\Omega}_3$ has three elements:

$$\mathbf{\Omega}_1 = \mathbf{\Omega}_1^1 \cup \mathbf{\Omega}_1^2 \cup \mathbf{\Omega}_1^3 \cup \mathbf{\Omega}_1^4, \quad \mathbf{\Omega}_2 = \mathbf{\Omega}_2^5 \cup \mathbf{\Omega}_2^6, \quad \mathbf{\Omega}_3 = \mathbf{\Omega}_3^7 \cup \mathbf{\Omega}_3^8 \cup \mathbf{\Omega}_3^9. \tag{3.17}$$

Each subregion can have a different polynomial order of the spectral expansion. In this case the coupling between subregions is done by mortars. One important issue is the choice of the discrete space for the mortar functions. The global $\mathcal{C}^0$ continuity of the basis, for a geometrically and functionally non-conforming mesh, cannot be guaranteed. To make the basis as continuous as possible, we have to minimize the difference in function values across each mortar (non-conforming interface). We impose that the $\mathcal{L}^2$ projection of the jump across the mortar vanished:

$$\int_\gamma (u_1 - u_2)\Psi\, ds = 0,\ \forall \Psi \in \mathbb{P}_{N-2}(\Gamma), \tag{3.18}$$

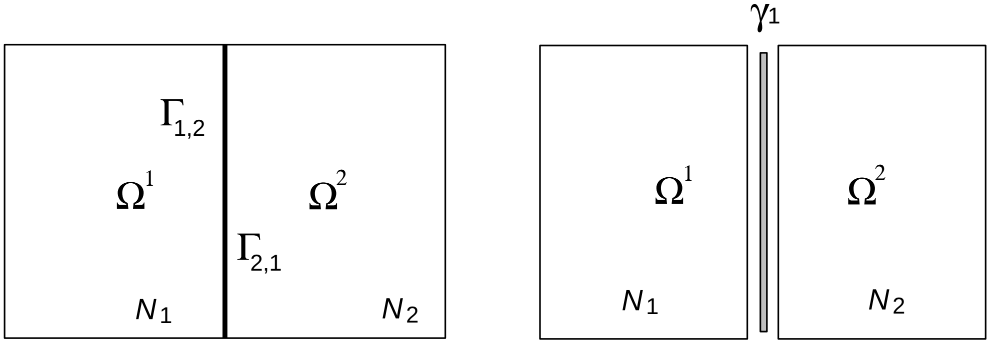


Fig. 3.3: Mortar decomposition in a functionally non-conforming case, one mortar. $\Gamma_{i,j}$ denotes the interface between elements $\boldsymbol{\Omega}^i$ and $\boldsymbol{\Omega}^j$.

and the continuity whenever the spectral element vertices coincided (at the end points of the mortars). Here, the two functions u_1 and u_2 are the functions that we would like to be continuous, and Ψ is the weight used to perform the minimization of the jump. Evaluating equation (3.18) using numerical quadrature, yields the algebraic form:

$$\mathbf{u_1} = \mathbf{Q}\,\mathbf{u_2}, \tag{3.19}$$

where $\mathbf{u_1}$ and $\mathbf{u_2}$ are the coefficients of the basis we choose to represent u_1 and u_2. We see that $\mathbf{Q}$ represent a set of relationships between the values of u_2, which are free, and the values of u_1, which are constrained to match them such that (3.18) is satisfied. In practice, we choose u_2 to be the solution along the mortar edges, and u_1 to be the solution along the edge of an adjacent non-conforming element. In this way, the u_1 degrees of freedom are eliminated in the mesh, and they are replaced with u_2.

To illustrate how the $\mathcal{L}^2$ projection is imposed, we consider a few examples (Deville *et al.*, 2002), where each subregion has only one element (we drop the subscript p in these examples).

In figure 3.3, the domain $\boldsymbol{\Omega}$ is split into two rectangular subdomains, $\boldsymbol{\Omega}^1$ and $\boldsymbol{\Omega}^2$, discretized by polynomials of degree N_1 and N_2, respectively. In this case the decomposition is functionally non-conforming. There are two mortars: $\gamma_1 = \Gamma_2^1$ and $\gamma_2 = \Gamma_4^2$, that can be considered. Without loss of generality, γ_1 is chosen to form the skeleton S of the domain. If we consider $\tilde{u}_1$ the restriction of u_1 to the interface $\Gamma_{1,2}$, and $\tilde{u}_2$ the restriction of u_2 to $\Gamma_{1,2}$, then $\tilde{u}_1 \equiv \Phi_1$ and $\tilde{u}_2$ becomes the dependent variable:

$$\int_{\Gamma_{1,2}} [\tilde{u}(s) - \Phi(s)]\Psi(s)\,ds = 0,\ \forall\Psi \in \mathbb{P}_{N_2-2}(\Gamma_{1,2}), \tag{3.20}$$

where $\Gamma_{1,2}$ is the interface between elements $\boldsymbol{\Omega}^1$ and $\boldsymbol{\Omega}^2$. If $N_1 \leq N_2$, $\tilde{u}_2(s)$ equation (3.20) can be evaluated by :

$$\tilde{u}_2(s) = \tilde{u}_1(s) + \alpha L_{N_2}(s) + \beta L_{N_2-1}(s), \tag{3.21}$$

where α and β can be expressed in terms of $\tilde{u}_{2,0}$ and $\tilde{u}_{2,N_2}$ by evaluating (3.21) at the vertices.

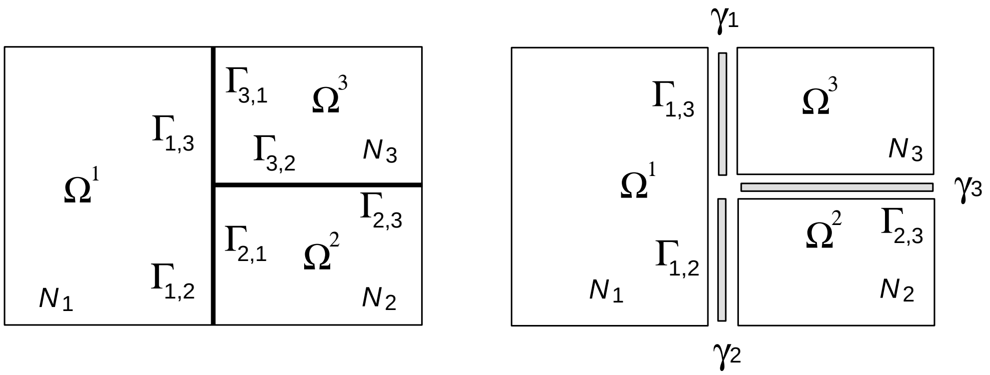


Fig. 3.4: Mortar decomposition in a geometrically non-conforming case, three mortars.

As a second example, we consider a geometrically non-conforming domain, see figures 3.4 and 3.5. In this case, possible choices for mortars are $\gamma_i \in \{\Gamma_{1,2}, \Gamma_{1,3}, \Gamma_{2,3}\}$ and $\gamma_i \in \{\Gamma_{1,2} \cup \Gamma_{1,3}, \Gamma_{2,3}\}$. To take into account the relative position and size of the interfaces, the integral constrain must be applied in physical coordinates. The long interface between $\mathbf{\Omega}^1$ and $\{\mathbf{\Omega}^2, \mathbf{\Omega}^3\}$, can be decomposed into either one or two mortars. First, we consider the skeleton $S = \bigcup_{i=1}^{3} \gamma_i$ (see figure 3.4). The restriction of $u(x)$ to the interface are $\tilde{u}_1, \tilde{u}_2, \tilde{u}_3$, where $\tilde{u}_2, \tilde{u}_3$ are the degrees of freedom. Applying integral matching condition yields:

$$\int_{\Gamma_2^1} \tilde{u}_1 \Psi \, ds = \int_{\Gamma_{1,2}} \tilde{u}_2 \Psi \, ds + \int_{\Gamma_{1,3}} \tilde{u}_3 \Psi \, ds, \tag{3.22}$$

$$\int_{\Gamma_2^1} \tilde{u}_1 \Psi \, ds = \mathbf{B} \tilde{\mathbf{u}}_1, \tag{3.23}$$

$$\int_{\Gamma_{1,2}} \tilde{u}_2 \Psi \, ds = \mathbf{P}_{1,2} \tilde{\mathbf{u}}_2, \tag{3.24}$$

$$\int_{\Gamma_{1,3}} \tilde{u}_3 \Psi \, ds = \mathbf{P}_{1,3} \tilde{\mathbf{u}}_3, \tag{3.25}$$

where the entries of matrices $\mathbf{B}$ and $\mathbf{P}$ are computed using equations (3.45) and (3.50). The dependent side is computed as:

$$\tilde{u}_1 = \mathbf{B}^{-1} \mathbf{P}_{1,2} \tilde{u}_2 + \mathbf{B}^{-1} \mathbf{P}_{1,3} \tilde{u}_3. \tag{3.26}$$

Since we impose the vertex matching conditions, the first and the last row of $\mathbf{B}$ and $\mathbf{P}_{1,2}, \mathbf{P}_{1,3}$ have to be modified accordingly (see equation (3.61)). Also, $\tilde{u}_{2,N_2}$ and $\tilde{u}_{3,0}$ have to match, because they represent the function values in the same point.

The case in figure 3.5 is similar. The degrees of freedom are associated with $\tilde{u}_1$ and the integral matching conditions yield:

$$\int_{\Gamma_{1,2}} (\tilde{u}_2 - \tilde{u}_1) \Psi \, ds = 0, \forall \Psi \in P_{N_2 - 2}, \tag{3.27}$$

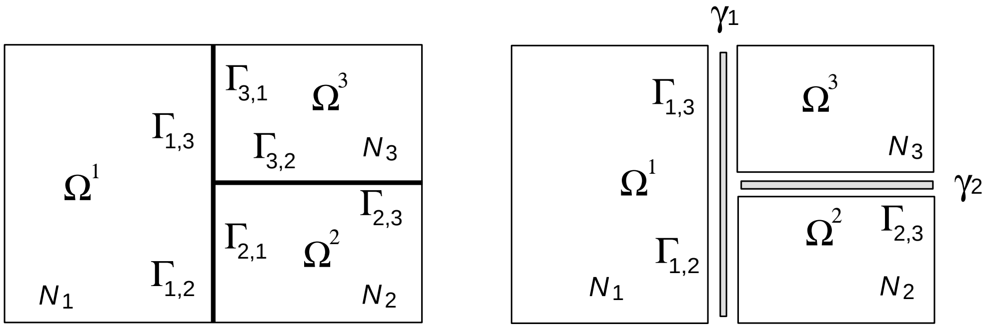


Fig. 3.5: Mortar decomposition in a geometrically non-conforming case, two mortars.

$$\int_{\Gamma_{1,3}} (\tilde{u}_3 - \tilde{u}_1)\Psi \, ds = 0, \forall \Psi \in P_{N_3-2}. \tag{3.28}$$

In this case, separate relationships are obtained for each of $\tilde{u}_2$ and $\tilde{u}_3$ by equations (3.27) and (3.28). To evaluate $\int_{\Gamma_{1,2}} \tilde{u}_1 \Psi \, ds$ and $\int_{\Gamma_{1,3}} \tilde{u}_1 \Psi \, ds$, we use a higher-order quadrature rule N, setting $N = \max(N_1, N_2)$ and $N = \max(N_1, N_3)$ respectively.

3.4 Mortar element formulation

We are now ready to introduce the mortar formulation, which is the main component of the adaptive mesh refinement process.The standard conforming discrete space $\mathbf{X}_p$ in each sub-domain $\mathbf{\Omega}_p$ is defined by:

$$\mathbf{X}_p(\mathbf{\Omega}_p) = \{v \in \mathcal{C}^0(\mathbf{\Omega}_p) : \forall k = 1, \ldots, K^p, v_{|\mathbf{\Omega}_p^k} \in \mathbb{P}_{N_p} \times \mathbb{P}_{N_p}(\mathbf{\Omega}_p^k), v_{|\partial\mathbf{\Omega}} = 0\} \tag{3.29}$$

where N_p is the polynomial order of the spectral expansion used in each element $\mathbf{\Omega}_p^k$ of a sub-region $\mathbf{\Omega}_p$. Here $\mathbb{P}_{N_p}(\mathbf{\Omega}_p^k)$ denotes the space of all polynomials on $\mathbf{\Omega}_p^k$ of order not greater than N_p in each spatial direction. Since each sub-domain $\mathbf{\Omega}_p$ contains only conforming elements, the conforming formulation (3.29) is valid everywhere, except on those element edges that belong to the mortar skeleton S. The skeleton S joins sub-domains $\mathbf{\Omega}_p$ with a different spectral expansion order (on S the nodal points that belong to the neighbouring elements do not coincide). Even in the particular case when the polynomial order of the neighbouring elements on S will be the same, there is a geometrically non-conforming coupling between the elements. In figure 3.2, the skeleton of the domain $\mathbf{\Omega}$ has three mortars:

$$S = \gamma_1 \cup \gamma_2 \cup \gamma_3, \tag{3.30}$$

where γ_i can be defined as a collection of edges: $\gamma_1 = \Gamma_2^2$ (edge two of element two), $\gamma_2 = \Gamma_3^5 \cup \Gamma_3^6$ (edge three of element five and six) and $\gamma_3 = \Gamma_2^3$ (edge two of element three).

In order to define the non-conforming space $\mathbf{X}_h$, we first introduce an auxiliary mortar space $\mathbf{W}_h$ that controls the error between the two representations of the solution on S and works as a link between the two representations (conforming and non-conforming).

In general, we define the skeleton S as a collection of line segments γ_i:

$$S = \bigcup_{i=1}^{\overline{M}} \overline{\gamma_i} = \bigcup_{p=1}^{P} \partial\mathbf{\Omega}_p, \tag{3.31}$$

where $\overline{M}$ represents the number of the mortar segments (see (3.30)). From this point on, unless noted otherwise, the parameters with an overbar will be related to the mortars.

Each end point of a mortar segment γ_i coincides with a vertex in all the sub-domains $\mathbf{\Omega}_p$, which share this portion of S. The space $\mathbf{W}_h$ is defined on the mortars γ_i and depends on the way the skeleton S is split into mortars. As illustrated in figure 3.2 the segments γ_i can be an edge of an element, or a union of edges.

The mortar auxiliary space $\mathbf{W}_h$ is now defined as:

$$\mathbf{W}_h = \{\Phi \in \mathcal{C}^0(S), \forall i = 1, \ldots, \overline{M},\ \Phi_{|\gamma_i} \in \mathbb{P}_{\overline{N}_i}(\gamma_i), \Phi_{|\partial\mathbf{\Omega}} = 0\ \}. \tag{3.32}$$

The order $\overline{N}_i$ of polynomial $\mathbb{P}_{\overline{N}_i}(\gamma_i)$ depends on the neighbouring spectral elements that are coupled to the mortar γ_i. We consider two sets of elements edges, the first set $\Gamma_j^S,\ j = 1, \ldots, K^S$ and the second set $\Gamma_j^M,\ j = 1, \ldots, K^M$ for each mortar γ_i, such that:

1. $\exists \mathbf{\Omega}^S \subset \mathbf{\Omega}_p,\ \exists\, \mathbf{\Omega}^M \subset \mathbf{\Omega}_p, \Gamma_j^S \in \mathbf{\Omega}^S \bigwedge \Gamma_j^M \in \mathbf{\Omega}^M, \forall\, j,$
2. $\gamma_i = \bigcup_{j=1}^{K^S} \Gamma_j^S$ and $\gamma_i = \bigcup_{j=1}^{K^M} \Gamma_j^M$, both sets cover the mortar completely,
3. $\gamma_i(v_i^1) \in \mathbf{\Omega}^S,\ \gamma_i(v_i^2) \in \mathbf{\Omega}^S$, and $\gamma_i(v_i^1) \in \mathbf{\Omega}^M,\ \gamma_i(v_i^2) \in \mathbf{\Omega}^M$, where v_i^1 and v_i^2 are the end points of the mortar γ_i.

The Γ^S and Γ^M are called the slave and the master sides of the mortar γ_i. The elements that contain the Γ^M edges set, form the master elements set. In the same way, the elements that contain the Γ^M edges set, form the slave elements set. These two sets are important for the definition of the data structure we use in the implementation. The polynomial order on the mortars sides can be different. In this case let N^S and N^M be the polynomial order of the expansions in $\mathbf{\Omega}^S$ and $\mathbf{\Omega}^M$ sub-domains. The order $\overline{N_i}$ of the polynomials $\mathbb{P}_{\overline{N}_i}(\gamma_i)$ in equation (3.32) is such that the number of degrees of freedom on the mortars is equal to the maximum number of degrees of freedom in the two sets Γ^S and Γ^M:

$$\overline{N}_i = \max(N^S K^S, N^M K^M). \tag{3.33}$$

On each interface γ_i, there are three different functions that belong to one of the following spaces:

$$\mathbf{W}_{|\gamma_i} = \{v \in \mathbb{P}_{\overline{N}_i}(\gamma_i)\}, \tag{3.34}$$

$$\mathbf{X}_p(\Omega^S)_{|\gamma_i} = \{v \in \mathcal{C}^0(\gamma_i), \forall j = 1, \ldots, K^S : v_{|\Gamma_j^S} \in \mathbb{P}_{N^S}(\Gamma_j^S)\ \}, \tag{3.35}$$

$$\mathbf{X}_p(\Omega^M)_{|\gamma_i} = \{v \in \mathcal{C}^0(\gamma_i), \forall j = 1, \ldots, K^M : v_{|\Gamma_j^M} \in \mathbb{P}_{N^M}(\Gamma_j^M)\ \}. \tag{3.36}$$

In order to minimize the difference between them, we introduce two additional constrains to the space $\mathbf{X}_p(\Omega^S)_{|\gamma_i}$ and $\mathbf{X}_p(\mathbf{\Omega}^M)_{|\gamma_i}$. Since $\gamma_i(v_i^1)$ and $\gamma_i(v_i^2)$, the end points of the mortar γ_i, coincide with a vertex in both $\mathbf{\Omega}^S$ and $\mathbf{\Omega}^M$, the function values in this points should be the same in all the representations. We call this condition, that ensures exact continuity at cross vertices, the vertex condition. The second condition we impose is the so-called integral matching condition: the jump in functions at internal boundaries on γ_i should be minimized. This is equivalent with the following: the error between the representations, on slave and master edges, is orthogonal to an appropriately chosen space.

With the two conditions imposed we can define the new spaces $\mathbf{X}_S(\gamma_i)$ and $\mathbf{X}_M(\gamma_i)$ as follows:

$$\mathbf{X}_S(\gamma_i) = \{v \in \mathcal{C}^0(\gamma_i),\ \forall j = \{1,\dots,K^S\} : v_{|_{\Gamma_j^S}} \in \mathbb{P}_{N^S}(\Gamma_j^S) \text{ such that} \tag{3.37}$$

$$\exists \Phi \in \mathbf{W}_h : \Phi(v_i^{1,2}) = v(v_i^{1,2}), \text{ and} \int_{\gamma_i} (v - \Phi_{|_{\gamma_i}})\Psi\, dS = 0,\ \forall\, \Psi \in \mathbf{P}^S(\gamma_i)\} \tag{3.38}$$

and

$$\mathbf{X}_M(\gamma_i) = \{v \in \mathcal{C}^0(\gamma_i),\ \forall j = \{1,\dots,K^M\} : v_{|_{\Gamma_j^M}} \in \mathbb{P}_{N^M}(\Gamma_j^M) \text{ such that} \tag{3.39}$$

$$\exists \Phi \in \mathbf{W}_h : \Phi(v_i^{1,2}) = v(v_i^{1,2}), \text{ and} \int_{\gamma_i} (v - \Phi_{|_{\gamma_i}})\Psi\, dS = 0,\ \forall\, \Psi \in \mathbf{P}^M(\gamma_i)\} \tag{3.40}$$

where $\mathbf{W}_h$ is defined in (3.32).

The integral matching conditions are represented by the integrals in equations (3.38 and 3.40) and can be viewed as projections from a mortar space onto its sides. Also, they represent a $\mathcal{L}^2$ minimization of the jump in functions at internal boundaries on the mortars segments. We have to construct now the projection spaces $\mathbf{P}^S(\gamma_i)$ and $\mathbf{P}^M(\gamma_i)$. The first constraint, for each mortar end point, imposes that the projection space should have at least two degrees of freedom less than the number of degrees of freedom on the sides. We can define now the projection spaces for the slaves and master sides:

$$\mathbf{P}^S(\gamma_i) = \{v \in \mathcal{C}^0(\gamma_i) : \forall j = \{1,\dots,K^S\},\ v_{|_{\Gamma_j^S}} \in \mathbb{P}_{N_{i,j}^S}(\Gamma_j^S)\ \} \tag{3.41}$$

$$\mathbf{P}^M(\gamma_i) = \{v \in \mathcal{C}^0(\gamma_i) : \forall j = \{1,\dots,K^M\},\ v_{|_{\Gamma_j^M}} \in \mathbb{P}_{N_{i,j}^M}(\Gamma_j^M)\ \} \tag{3.42}$$

where $N_{i,j}^S$ and $N_{i,j}^M$ are the orders that depend on the number of the end points shared by the mortar γ_i and the element edge Γ_j^S, Γ_j^M, respectively. The choice of the order is shown schematically in figure 3.6.

The non-conforming spectral element discretization space can now be defined as:

$$\begin{aligned} \mathbf{X}_h(\mathbf{\Omega}) \quad = \quad & \{v : \forall p = 1,\dots,P,\ v_{|\mathbf{\Omega}_p} \in \mathbf{X}_p(\mathbf{\Omega}_p), \forall \gamma_i \in S,\ i = 1,\dots,\overline{M}, \\ & \exists \Gamma_j^S, \Gamma_j^M, \gamma_i = \bigcup_{j=1}^{K^S} \Gamma_j^S = \bigcup_{j=1}^{K^M} \Gamma_j^M \text{ and} \\ & v_{|\Gamma^S} \in \mathbf{X}_S(\gamma_i),\ \ v_{|\Gamma^M} \in \mathbf{X}_M(\gamma_i)\}. \qquad (3.43) \\ & \qquad (3.44) \end{aligned}$$

To control the error between the two different spectral representations on γ_i, we make the solution continuous at the mortar end points and the error between the representations orthogonal to an appropriately chosen space. The definition of discrete spaces $\mathbf{X}_S(\gamma_i)$ and $\mathbf{X}_M(\gamma_i)$ that are given on the two sides of the mortar, differ from each other only in the number of elements K^S and K^M and the order of the polynomials N^S and N^M.

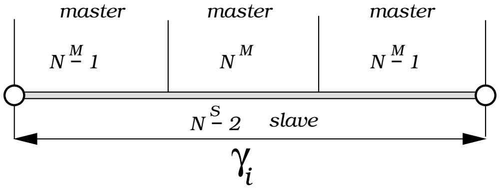


Fig. 3.6: Order $N_{i,j}^{S|M}$ in the definition of the projection space on the two sides of a mortar (slave and master). Here a single mortar contacts with three master elements on one side with one slave element on other side. The order $N_{i,j}^{M}$ on the master side is set to either $N^M - 1, N^M$ or $N^M - 1$, depending whether the mortar and an element share a vertex of not. On the slave side the order $N_{i,j}^{S}$ is set to $N - 2$.

3.4.1 *Definition of basis functions*

Since in each element $\mathbf{\Omega}_p^k$ of a subregion $\mathbf{\Omega}_p$ we have a conforming spectral expansion, a function $u \in \mathbf{X}_p(\mathbf{\Omega}_p)$ can be interpolated with a set of a Lagrangian interpolants of order N_p. According to the definition of the mortar space in equation (3.32), any function $\Phi \in \mathbf{W}_h$ can be interpolated on a mortar γ_i by a set of Lagrangian interpolants of order $\overline{N_i}$.

Since we specify the basis functions only for a single mortar γ_i and a basis for neighbouring elements on one of the mortar sides only, in the following we drop the subscripts in both N_p and $\overline{N}_i$. Therefore we introduce only a basis for the both mortar sides.

In each element $\mathbf{\Omega}^k$ that belongs to either $\mathbf{\Omega}^S$ or $\mathbf{\Omega}^M$, a variable u is interpolated as:

$$u(x(\xi,\eta), y(\xi,\eta)) = \sum_{i,j=0}^{N} u_{ij} h_i^N(\xi) h_j^N(\eta) \tag{3.45}$$

where h_i^N are the Lagrangian interpolants of order N:

$$h_i^N(\xi) = \frac{-L_N^{'}(\xi)(1-\xi^2)}{N(N+1)L_N(\xi_i^N)(\xi - \xi_i^N)}, \quad i = \{0, 1, ..., N\}, \tag{3.46}$$

$h_i^N \in \mathbb{P}_N([-1,1]), h_i^N(\xi_j) = \delta_{ij}, \forall i,j \in \{0,1,...,N\}^2$, L_N is the Legendre polynomial of order N. The ξ_i points are the GLL points of order N:

$$\xi_i^N = \begin{cases} -1 & i = 0 \\ \textit{roots} \text{ of } L_N^{'}(\xi) \; = \; 0 & i = 1, \dots, N-1 \\ 1 & i = N. \end{cases}$$

The functions $x(\xi,\eta), y(\xi,\eta)$ are the coordinate transformation functions from an isoparametric element into a square: $\xi,\eta \in [-1,1]$, see figure 3.1. For each element edge Γ_j a function u can be written as:

$$u(x(\xi), y(\xi)) = \sum_{n=0}^{N} u_n h_n^N(\xi) \tag{3.47}$$

where $x(\xi)$, $y(\xi)$ is a transformation between an element edge Γ_j and its representation in a computational space. According to the definition of mortar space $\mathbf{W}_h$, any function $\Phi \in \mathbf{W}_h$ can be interpolated on a mortar γ_i as:

$$\Phi(\overline{x}(\overline{\xi}), \overline{y}(\overline{\xi})) = \sum_{m=0}^{\overline{N}} \Phi_m h_m^{\overline{N}}(\overline{\xi}) \tag{3.48}$$

where the order $\overline{N}$ of polynomials basis is computed accordingly to equation (3.33).

The transformations $\overline{x}(\overline{\xi})$, $\overline{y}(\overline{\xi})$ will transform a line segment γ_i into its computational space, $\overline{\xi} \in [-1,1]$.

A basis for projection space $\mathbf{P}^{S|M}(\gamma_i)$ is a set of $K_{SM} = KN - 1$ functions. To define a basis for the projection space $\mathbf{P}^{S|M}(\gamma_i)$ we detect whether the end points of the mortar γ_i coincide with the end points of Γ_j, where $\mathbf{P}^{S|M}$ represents the $\mathbf{P}^S$ of $\mathbf{P}^M$ space. Since we impose the condition that the values of the functions, and the mortar, coincide at the mortar end-points, we have to use two projection polynomials of order one or two, less than the Legendre function of order N (equation (3.46)). To avoid loosing the diagonality in the left hand side of the projection, equation (3.54), we have to choose the polynomials of the form (3.49), (3.51), (3.52). After using quadrature of order N in (3.54), the only non-diagonal term introduced are those R_{q0} and $R_{q\overline{N}}$, that correspond to the element end points. Since we know that those element end points coincide with the mortar end points, we can move them to the right hand side (see equation (3.69). Now, we can define the basis on the mortars and its neighbouring elements.

If $\gamma_i = \Gamma_j$, the basis for $\mathbf{P}^{S|M}(\gamma_i)$ is a set of $N-1$ polynomials of order $N-2$, which has the form

$$\Psi_i^{N-2}(\xi) = \frac{-L_N^{'}(\xi)(1-\xi_i^N)(1+\xi_i^N)}{N(N+1)L_N(\xi_i^N)(\xi-\xi_i^N)}, \quad i = \{1, ..., N-1\}, \tag{3.49}$$

where $\xi_i^N,\ i = \{1, ..., N-1\}$ are the GLL points.

In the general case when $\gamma_i = \bigcup_{j=1}^{K} \Gamma_j,\ K > 1$, the basis for $\mathbf{P}^{S|M}(\gamma_i)$ consist of the union of basis functions for all edges Γ_j. For an edge that does not share any end points with the mortar, the basis is a set of $N+1$ Lagrangian interpolants of order N:

$$\Psi_i^N = h_i^N,\ i = \{0, 1, ..., N\}. \tag{3.50}$$

In the last case when an edge Γ_j shares only one end point with a mortar γ_i , the basis functions are polynomials of order $N-1$. If the end point corresponds to $\xi_0 = -1$, the polynomials are:

$$\Psi_i^{N-1}(\xi) = \frac{-L_N^{'}(\xi)(1-\xi)(1+\xi_i^N)}{N(N+1)L_N(\xi_i^N)(\xi-\xi_i^N)}, \quad i = \{1, ..., N\}. \tag{3.51}$$

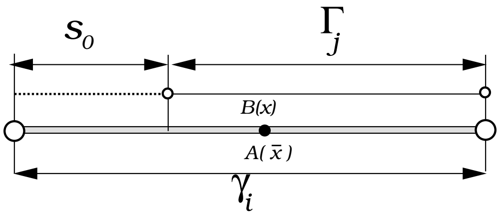


Fig. 3.7: Illustration of two different representations of the same point in physical space. The point $B(x)$ is represented in the coordinate system of an element. The point $A(\overline{x})$ is represented in the coordinate system of a mortar. The parameter s_0 is the offset between the mortar γ_i and the edge Γ_j.

Otherwise, if the end point corresponds to $\xi_N = 1$ the polynomials are:

$$\Psi_i^{N-1}(\xi) = \frac{-L_N'(\xi)(1+\xi)(1-\xi_i^N)}{N(N+1)L_N(\xi_i^N)(\xi-\xi_i^N)}, \quad i = \{0, 1, ..., N-1\}. \tag{3.52}$$

Since, we use the Lagrangian interpolants through GLL points, the functions Ψ_i defined above, satisfy the relation $\Psi_i(\xi_j) = \delta_{ij}$ for all nodal points ξ_j, except those that coincide with the end points of the mortar.

3.4.2 *Projection operator*

In order to express the integral matching conditions, expressed by equations (3.38), (3.40), we insert interpolation formulas (3.47), (3.48) in (3.38) and (3.40) and compute the integrals:

$$\int_{\gamma_i} (u - \Phi_{|\gamma_i})\Psi_q \, ds = 0, \tag{3.53}$$

for each basis function of the projection space $\Psi_q \in \mathbf{P}^{S|M}(\gamma_i), \ q = \{1, ..., K_{SM}\}$ defined in (3.49), (3.50), (3.51) and (3.52). To compute the integrals over the mortar γ_i we have to sum up the contributions from all the element edges $\Gamma_j \subset \gamma_i, j = \{1, ..., K\}$, while on each Γ_j, the integration is performed with Gauss-Lobatto quadrature. To integrate along the edges of the elements, the order of the quadrature for the $u\Psi_q$ term will be N. Since the terms that contain Φ as their polynomial expansion can be of much higher order than the quadrature order N, we use $\overline{N}$ as quadrature order for the integration of $\Phi_{|\gamma_i} \Psi_q$.

In case when Ψ_q is nonzero over an element edge Γ_j the integration of the $u\Psi_q$ term over Γ_j yields

$$\int_{\Gamma_j} u\Psi_q \, ds = \sum_{n=0}^{N} u_n \int_{-1}^{1} h_n^N(\xi)\Psi_q(\xi)|S(\xi)| \, d\xi = B_q u_q + R_{q0}\Phi_0 + R_{q\overline{N}}\Phi_{\overline{N}} \tag{3.54}$$

where

$$B_q = \rho_q^N |S_q^{\Gamma_j}|, \tag{3.55}$$

$$R_{q0} = \Psi_q(-1)\rho_0^N |S_0^{\Gamma_j}|, \tag{3.56}$$

$$R_{q\overline{N}} = \Psi_q(1)\rho_N^N |S_N^{\Gamma_j}|, \tag{3.57}$$

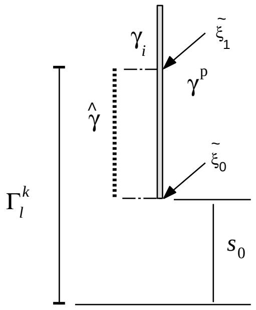


Fig. 3.8: Representation of a mortar offset s_0, edge Γ_l^k, mortar γ_i, and the integration strip $\hat{\gamma}$. The offset s_0 can be positive of negative. In this case s_0 is positive, $\tilde{\xi}_0$ and $\tilde{\xi}_1$ are the intersection points between the edge Γ_l^k and the mortar γ_i.

and ρ_i^N are the weights of the quadrature of order N.

The $|S_i^{\Gamma_j}|$ are metric terms, that are generated by the transformation of the coordinates between the edge Γ_j and their representation in the computational space. Since the edge Γ_j is a straight line on a two-dimensional plane, the metric terms are simply the length of an edge in the physical space, divided by the length of the edge in the computational space:

$$|S_q^{\Gamma_j}| = |S_0^{\Gamma_j}| = |S_N^{\Gamma_j}| = \frac{\sqrt{(x_{j,2} - x_{j,1})^2 + (y_{j,2} - y_{j,1})^2}}{1 - (-1)} = \frac{|\Gamma_j|}{2}. \tag{3.58}$$

If the test function Ψ_q is not zero at the end points $\xi = \pm 1$, the terms R_{q0} and $R_{q\overline{N}}$ will be not zero. In the case that an edge does not share any end point with a mortar, equation (3.50), R_{q0} and $R_{q\overline{N}}$ are zero. If $\gamma_i = \Gamma_j$, both terms are not zero and this is the case that we consider in our implementation. Next, we integrate $\Phi\Psi_q$ over an element edge Γ_j in a similar way as the integration of $u\Psi_q$:

$$\int_{\Gamma_j} \Phi\Psi_q \, ds = \sum_{m=0}^{\overline{N}} \Phi_m \int_{-1}^{1} h_m^{\overline{N}}(\overline{\xi}(\xi))\Psi_q(\xi)|S(\xi)| \, d\xi = \sum_{m=0}^{\overline{N}} P_{qm}\Phi_m \tag{3.59}$$

where

$$P_{qm} = \sum_{n=0}^{\overline{N}} h_m^{\overline{N}}(\overline{\xi}(\xi_n^{\overline{N}}))\Psi_q(\xi_n^{\overline{N}})\rho_n^{\overline{N}}|S_n^{\Gamma_j}|. \tag{3.60}$$

The function $\overline{\xi}(\xi)$ is a coordinate transformation between two different representations of the same point (x, y) in the computational space for a mortar and for an element. To compute the transformation $\overline{\xi}(\xi)$ the mortar offset s_0, is introduced as illustrated in figures 3.8 and 3.7. A point x is represented in two computational spaces, for a mortar, and for an element by:

$$A(\overline{x}(\overline{\xi})) \quad = \quad B(x(\xi)) + s_0 \tag{3.61}$$

$$\begin{aligned}\frac{1}{2}[(\overline{x}_2-\overline{x}_1)\overline{\xi}+(\overline{x}_2+\overline{x}_1)] &= \frac{1}{2}[(x_2-x_1)\xi+(x_2+x_1)]+s_0 && (3.62)\\ (|\gamma_i|\overline{\xi}+|\gamma_i|) &= (|\Gamma_j|\xi+|\Gamma_j|)+2s_0 && (3.63)\\ \overline{\xi}(\xi) &= \frac{2s_0}{|\gamma_i|}-1+\frac{|\Gamma_j|}{|\gamma_i|}(\xi+1). && (3.64)\end{aligned}$$

Now, we combine equations (3.54 and 3.59) and assemble the contributions from different element edges Γ_j together. We obtain:

$$B_q u_q + R_{q0}\Phi_0 + R_{q\overline{N}}\Phi_{\overline{N}} = \sum_{m=0}^{\overline{N}} P_{qm}\Phi_m, \quad q=\{1,...,K_{SM}\} \tag{3.65}$$

$$B_q u_q = P_{q0}\Phi_0 - R_{q0}\Phi_0 + \sum_{m=1}^{\overline{N}-1} P_{qm}\Phi_m + P_{q\overline{N}}\Phi_{\overline{N}} - R_{q\overline{N}}\Phi_{\overline{N}} \tag{3.66}$$

$$B_q u_q = (P_{q0}-R_{q0})\Phi_0 + \sum_{m=1}^{\overline{N}-1} P_{qm}\Phi_m + (P_{q\overline{N}}-R_{q\overline{N}})\Phi_{\overline{N}} \tag{3.67}$$

$$u_q = \frac{(P_{q0}-R_{q0})}{B_q}\Phi_0 + \frac{\sum_{m=1}^{\overline{N}-1} P_{qm}\Phi_m}{B_q} + \frac{(P_{q\overline{N}}-R_{q\overline{N}})}{B_q}\Phi_{\overline{N}} \tag{3.68}$$

Now, the projection operator is defined as:

$$u_q = Q_{q0}\Phi_0 + \sum_{m=1}^{\overline{N}-1} Q_{qm}\Phi_m + Q_{q\overline{N}}\Phi_{\overline{N}}, \ \ q=\{1,...,K_{SM}\} \tag{3.69}$$

where

$$Q_{qm} = \frac{P_{qm}}{B_q}, \quad Q_{qk} = \frac{(P_{qk}-R_{qk})}{B_q}, \ \ k=0,\overline{N}. \tag{3.70}$$

Term R_{qk} arises from the fact that we need to make the values of our functions, (on both side of the mortar, and on the mortar itself) the same at the end points of the mortar. In the general case, the order of the spectral expansions, on the two sides of the mortar are not the same. Therefore there are two projections of the form (3.69) for each mortar. To form the global linear projection operator, the projection operators are constructed for all mortars $\gamma_i, \ i=1,...,\overline{M}$, together with the conditions for the vertices $\Phi(v_i^1)=u(v_i^1)$ and $\Phi(v_i^2)=u(v_i^2)$. The global linear projection operator becomes:

$$\tilde{\mathbf{u}} = \mathbf{Q}\,\Phi, \tag{3.71}$$

where $\tilde{u}$ are nodal values on those element edges that require non-conforming matching and Φ are the nodal values on all the mortars $\gamma_i, i=1,...,\overline{M}$. The operator $\mathbf{Q}$ has a block structure, because the mortars always have two sides to connect. In general the matrix $\mathbf{Q}$ consist of two sub-matrices of similar structure, stacked on top of each other. For the particular case we consider (geometrically non-conforming), the matrix $\mathbf{Q}$ has the structure:

$$\mathbf{Q} = \begin{pmatrix} \mathbf{Q}_{Slave} \\ \mathbf{I}_{Master} \end{pmatrix}$$

where $\mathbf{Q}_{Slave}$ is the projector on the slave element side and $\mathbf{I}_{Master}$ is the identity matrix on the master side. Elements that share a common boundary segment, also share data in u_h. In a non-conforming method the following generalization is considered: the local solution $\tilde{u}^k$ is a projection of u_h onto the local basis:

$$\tilde{\mathbf{u}}^k = \mathbf{Q}^k \, \mathbf{u}_h. \tag{3.72}$$

The simplest interpretation of equation (3.72) is to think of $\mathbf{Q}^k$ as a sort of interpolation. In general the data values along the non-conforming boundary segments of each small element are not independent degrees of freedom but are obtained by projection of the solution along the adjacent (large) element onto the local polynomial basis. Figure 3.8 illustrates an arbitrary mortar/edge configuration, and introduces the notions of mortar offset s_0, mortar γ_i of length $|\gamma_i|$, elemental edge Γ_l^k of length $|\Gamma_l^k|$, and the integration strip $\hat{\gamma} = \Gamma_l^k \cap \gamma_i$ of length $|\hat{\gamma}|$. The intersection between the mortar γ_i with the top and bottom corners of the integration strip $\hat{\gamma}$, given in mortar-local coordinates, is represented by the points $\tilde{\xi}_0$ and $\tilde{\xi}_1$.

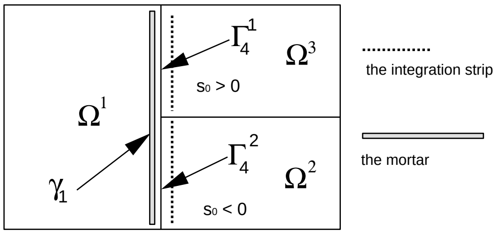


Fig. 3.9: Non-conforming mesh for a simple splitting, derived from the refinement of a conform mesh. The offset s_0 for the edge Γ_4^1 is positive. For the edge Γ_4^2 s_0 is negative. The mortar γ_1 is a union of the edges Γ_4^2 and Γ_4^1.

We consider only the cases where for each γ_i (the mortar) there exists an element $\mathbf{\Omega}^k$ that accepts γ_i as an entire edge. In this case, the refinement process starts with a mesh which is initially conforming. Figure (3.9) illustrates such a case in which the integration strip is $\hat{\gamma} = \Gamma_l^k/2$. Equation (3.72) gives as the procedure for construction of the local solution. In the non-conforming case, a corresponding change to equation (2.31) is made. In the standard conforming method $\mathbf{A}$ and $\mathbf{f}$ are formed by summing local contributions from K elements.

3.5 Solution techniques - static condensation

In this section we describe some implementation techniques, used to solve the large algebraic system that results from non-conforming spectral element discretizations. The static condensation algorithm is a method to reduce the complexity of the elemental matrices arising in spectral element methods. Because the elemental matrices $\mathbf{A}^k$ can be split into components containing only boundary and interior contributions, static condensation is particularly attractive for unstructured spectral element methods. In the mortar element method, there is a natural division of equations into those for boundaries (mortars) and element interiors (Maday *et al.*, 1989).

Static condensation techniques may be applied to general non-symmetric or symmetric matrix systems. We consider a non-symmetric matrix system to introduce the static condensation method. To apply this method to a discrete equation we begin by partitioning the elemental matrix into boundary and interior points, first for conforming elements, and second, for unstructured elements:

$$\begin{bmatrix} \mathbf{A}_{11}^k & \mathbf{A}_{12}^k \\ & \\ \mathbf{A}_{21}^k & \mathbf{A}_{22}^k \end{bmatrix} \begin{bmatrix} \mathbf{u}_b^k \\ \\ \mathbf{u}_i^k \end{bmatrix} = \begin{bmatrix} \mathbf{f}_b^k \\ \\ \mathbf{f}_i^k \end{bmatrix}, \tag{3.73}$$

where $\mathbf{A}_{11}^k$ is the boundary matrix: the components of $\mathbf{A}^k$ resulting from coupling between boundary-boundary nodes interactions, $\mathbf{A}_{12}^k$ is the coupling matrix: the components of $\mathbf{A}^k$ resulting from coupling between boundary-interior nodes interactions, $\mathbf{A}_{21}^k$ is the coupling interior-boundary matrix : the components of $\mathbf{A}^k$ resulting from coupling between interior-boundary nodes interactions and $\mathbf{A}_{22}^k$ is the interior matrix: the components of $\mathbf{A}^k$ resulting from coupling between interior-interior nodes interactions.

To solve this system (3.73), it is factored into one for the boundary nodes, and one for the interior nodes, for each element (sub-domain) $\mathbf{\Omega}^k$:

$$\mathbf{A}_{11}^k \mathbf{u}_b^k \quad + \quad \mathbf{A}_{12}^k \mathbf{u}_i^k \quad = \quad \mathbf{f}_b^k \tag{3.74}$$

$$\mathbf{A}_{21}^k \mathbf{u}_b^k \quad + \quad \mathbf{A}_{22}^k \mathbf{u}_i^k \quad = \quad \mathbf{f}_i^k. \tag{3.75}$$

After elimination of u_i from the second equation, and replacement in the first equation we obtain:

$$[\mathbf{A}_{11}^k \ - \ \mathbf{A}_{12}^k \, [\mathbf{A}_{22}^k]^{-1} \, \mathbf{A}_{21}^k] \, \mathbf{u}_b^k \ = \ \mathbf{f}_b^k \ - \ \mathbf{A}_{12}^k \, [\mathbf{A}_{22}^k]^{-1} \, \mathbf{f}_i^k \tag{3.76}$$

$$\mathbf{u}_i^k \ = \ [\mathbf{A}_{22}^k]^{-1} \, [\mathbf{f}_i^k \ - \ \mathbf{A}_{21}^k \, \mathbf{u}_b^k]. \tag{3.77}$$

The global boundary matrix is assembled by summing the elemental matrices:

$$\mathbf{A}_{11} = \sum_{k=1}^{K}{}' \, [\mathbf{A}_{11}^k \ - \ \mathbf{A}_{12}^k \, [\mathbf{A}_{22}^k]^{-1} \, \mathbf{A}_{21}^k]. \tag{3.78}$$

To evaluate $\mathbf{A}_{11}$, first, we evaluate and invert $[\mathbf{A}_{11}^k \ - \ \mathbf{A}_{12}^k \, [\mathbf{A}_{22}^k]^{-1} \, \mathbf{A}_{21}^k]$ which is also known as the Schur complement of $\mathbf{A}_{22}^k$ in $\mathbf{A}_{11}$. Subsequently, as part of this phase, we compute and store for each element the inverse of the interior matrix $[\mathbf{A}_{22}^k]^{-1}$ and its product with the coupling matrix $\mathbf{A}_{12}^k\ [\mathbf{A}_{22}^k]^{-1}$. The products $\mathbf{A}_{12}^k\ [\mathbf{A}_{22}^k]^{-1}\ \mathbf{f}_i^k$ and $[\mathbf{A}_{22}^k]^{-1}\ \mathbf{A}_{21}^k\ \mathbf{u}_b^k$ can be also treated as local operations, because they only involve the matrix-vector products of a known vector $\mathbf{f}_i^k$ and $\mathbf{u}_b^k$. The global assembly is only necessary for the boundary system when using static condensation. Once the boundary solution is known, the solution for the interior elemental nodes, given by equation (3.77), can be performed at elemental level. Because the coupling between elements is only $\mathcal{C}^0$, the element interiors are independent and on a multiprocessor system this final stage can be solved concurrently. The effect of constructing each local Schur complement matrices $[\mathbf{A}_{11}^k\ -\ \mathbf{A}_{12}^k\ [\mathbf{A}_{22}^k]^{-1}\ \mathbf{A}_{21}^k]$ is to separate the boundary nodes from the interior nodes. However, the inverse matrix $[\mathbf{A}_{22}^k]^{-1}$ is typically full, which means that the boundary nodes are tightly coupled. It is this coupling which dictates the bandwidth of the globally assembled Schur complement system. To reduce computational time and memory, we wish to find an optimal form of the discrete system corresponding to a minimum bandwidth for the matrix $\mathbf{A}_{11}$. To compute the bandwidth we simply need to find the maximum difference between the global numbering of the boundary nodes within each element, which is a Greedy algorithm (Saad, 1995). Even though the boundary nodes are coupled to all other boundary nodes of neighbouring elements, they are not coupled with boundary nodes within non-neighbouring elements. The reduction in bandwidth translates to direct savings in memory and computational cost. Some standard methods of bandwidth reduction used for finite elements, e.q. the reverse Cuthill-McKee algorithm can be also used, but they will be used only for the boundary system (Saad, 1995) .

Let us consider now the non-conforming case (mortar method). In the mortar method, the interface conditions, the integral matchings, are imposed variationally through an $\mathcal{L}^2$ minimization condition. After discretization, we can select some edges to become mortars. In the non-conforming case, we make the following generalization: the local solution $\tilde{\mathbf{u}}^k$ is a projection of $\mathbf{u}_h$ onto the local basis.
We can write this as follows:

$$\tilde{\mathbf{u}}^k = \mathbf{Q}\ \mathbf{u}_h. \tag{3.79}$$

For a master element, $\mathbf{Q}$ is the identity matrix, here $\mathbf{u}_h$ can be replaced in (3.79) by the solution along the corresponding master edge. For a ”slave” element, let $\tilde{\mathbf{u}}_b$ represents the values of the true degrees of freedom along the boundary (conforming edges plus mortars) and $\mathbf{u}_i$ the values on the interior of the element. The vector of nodal coefficients for a non-conforming element can be related to the standard (conforming) coefficients through the matrix equation:

$$\tilde{\mathbf{u}}^k = \begin{bmatrix} \mathbf{u}_b^k \\ \\ \mathbf{u}_i^k \end{bmatrix} = \begin{bmatrix} \mathbf{Q}^k & \mathbf{0} \\ \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \tilde{\mathbf{u}}_b^k \\ \\ \mathbf{u}_i^k \end{bmatrix} \tag{3.80}$$

where $\mathbf{0}$ is the zero matrix, $\mathbf{I}$ is the identity matrix and $\mathbf{Q}^k$ is the transformation (projection) matrix that relates the master nodal points to the original boundary points of the slave element. Some elements may have more than one mortar edge. In this case we have to sum the contributions of all intersections of the mortar γ_i with the particular Γ_l^k. Equation (3.80) provides the principal relation, required to form the elemental matrix system. If the local system for a standard conforming element is given by:

$$\mathbf{A}^k\,\mathbf{u}^k = \mathbf{f}^k \tag{3.81}$$

then the corresponding system for a non-conforming element is (Maday *et al.*, 1989):

$$[\mathbf{Q}^k]^T\,\mathbf{A}^k\,\mathbf{Q}^k\,\tilde{\mathbf{u}}^k = [\mathbf{Q}^k]^T\,\mathbf{f}^k \tag{3.82}$$

where $\tilde{\mathbf{u}}^k = [\tilde{\mathbf{u}}_b^k\ \mathbf{u}_i^k]^T$.

Using the static condensation method, the elemental system is obtained:

$$\begin{bmatrix} [\mathbf{Q}^k]^T & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{A}_{11}^k & \mathbf{A}_{12}^k \\ \mathbf{A}_{21}^k & \mathbf{A}_{22}^k \end{bmatrix} \begin{bmatrix} \mathbf{Q}^k & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \tilde{\mathbf{u}}_b^k \\ \mathbf{u}_i^k \end{bmatrix} = \begin{bmatrix} [\mathbf{Q}^k]^T & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{f}_b^k \\ \mathbf{f}_i^k \end{bmatrix} \tag{3.83}$$

After evaluation of the left side of the system (3.83), we obtain:

$$\begin{bmatrix} [\mathbf{Q}^k]^T\mathbf{A}_{11}^k\mathbf{Q}^k & [\mathbf{Q}^k]^T\mathbf{A}_{12} \\ \mathbf{A}_{21}^k\mathbf{Q}^k & \mathbf{A}_{22}^k \end{bmatrix} \begin{bmatrix} \tilde{\mathbf{u}}_b^k \\ \mathbf{u}_i^k \end{bmatrix} = \begin{bmatrix} [\mathbf{Q}^k]^T & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{f}_b^k \\ \mathbf{f}_i^k \end{bmatrix} \tag{3.84}$$

As in the conforming case (3.73), we have the same elemental system:

$$\begin{bmatrix} \tilde{\mathbf{A}}_{11}^k & \tilde{\mathbf{A}}_{12}^k \\ \tilde{\mathbf{A}}_{21}^k & \tilde{\mathbf{A}}_{22}^k \end{bmatrix} \begin{bmatrix} \mathbf{u}_b^k \\ \mathbf{u}_i^k \end{bmatrix} = \begin{bmatrix} \tilde{\mathbf{f}}_b^k \\ \tilde{\mathbf{f}}_i^k \end{bmatrix} \tag{3.85}$$

where

$$\tilde{\mathbf{A}}_{11}^k = [\mathbf{Q}^k]^T\,\mathbf{A}_{11}^k\,\mathbf{Q}^k, \tag{3.86}$$

$$\tilde{\mathbf{A}}_{12}^{k} = [\mathbf{Q}^{k}]^{T}\,\mathbf{A}_{12}^{k}, \tag{3.87}$$

$$\tilde{\mathbf{A}}_{21}^{k} = \mathbf{A}_{21}^{k}\,\mathbf{Q}^{k}, \tag{3.88}$$

$$\tilde{\mathbf{A}}_{22}^{k} = \mathbf{A}_{22}^{k}, \tag{3.89}$$

$$\tilde{\mathbf{f}}_{b}^{k} = [\mathbf{Q}^{k}]^{T}\,\mathbf{f}_{b}^{k}, \tag{3.90}$$

$$\tilde{\mathbf{f}}_{i}^{k} = \mathbf{f}_{i}^{k}. \tag{3.91}$$

(3.85) can be factorized into one for the boundary (mortars) nodes and one for the interior nodes, so that on $\mathbf{\Omega}^k$ we have:

$$[\tilde{\mathbf{A}}_{11}^{k} - \tilde{\mathbf{A}}_{12}^{k}\,[\tilde{\mathbf{A}}_{22}^{k}]^{-1}\,[\tilde{\mathbf{A}}_{21}^{k}]\,\tilde{\mathbf{u}}_{b}^{k} = \tilde{\mathbf{f}}_{b} - [\tilde{\mathbf{A}}_{12}\tilde{\mathbf{A}}_{22}^{-1}]\,\tilde{\mathbf{f}}_{i} \tag{3.92}$$

$$\tilde{\mathbf{A}}_{22}\,\mathbf{u}_{i} = \tilde{\mathbf{f}}_{i} - \tilde{\mathbf{A}}_{21}\,\mathbf{u}_{b}. \tag{3.93}$$

During a pre-processing phase, the global boundary matrix is assembled by summing the elemental matrices of the conforming and non-conforming elements:

$$\mathbf{A}_{11} = \sum_{k=1}^{K}{}'[\tilde{\mathbf{A}}_{11} - \tilde{\mathbf{A}}_{12}\,[\tilde{\mathbf{A}}_{22}^{k}]^{-1}\,\tilde{\mathbf{A}}_{21}]. \tag{3.94}$$

In the case when a direct solver is used to solve the system (3.93), the assembled boundary matrix is prepared for the solution phase by computing its LU factorization. Then the system is solved by setting up the right-hand side of the global boundary equations, solving the boundary equations, using a back-substitution, and accordingly computing the solution on the interior of each element, using matrix multiplication. To illustrate this process, first, we construct the global operator **A**. The block diagonal matrix is regrouped to yield a system of linear equations (Maday *et al.*, 1989):

$$\begin{bmatrix} \mathbf{A}_{SS} & \mathbf{A}_{SI} \\ \mathbf{A}_{IS} & \mathbf{A}_{II} \end{bmatrix} \begin{bmatrix} \mathbf{u}_{S} \\ \mathbf{u}_{I} \end{bmatrix} = \begin{bmatrix} \mathbf{f}_{S} \\ \mathbf{f}_{I} \end{bmatrix} \tag{3.95}$$

where the points that lie on the skeleton S (the skeleton contains the boundaries points of the elements) are ordered first in the global vector $\mathbf{u} = (\mathbf{u}_S, \mathbf{u}_I)$ and the right-hand side vector $\mathbf{f} = (\mathbf{f}_S, \mathbf{f}_I)$ that correspond to the nodal points in a conforming subregion $\mathbf{\Omega}_p$.

Since the solution of the linear system (3.95) has a discontinuity at the non-conforming interface S, in order to minimize the error (Maday *et al.*, 1989), the following problem has to be solved:

$$\begin{bmatrix} \mathbf{Q}^T & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{A}_{SS} & \mathbf{A}_{SI} \\ \mathbf{A}_{IS} & \mathbf{A}_{II} \end{bmatrix} \begin{bmatrix} \mathbf{Q} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{u}_S \\ \mathbf{u}_I \end{bmatrix} = \begin{bmatrix} \mathbf{Q}^T & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{f}_S \\ \mathbf{f}_I \end{bmatrix} \tag{3.96}$$

It is easy to see that a solution of (3.96) satisfies the sets of requirements we impose (3.38), (3.40), (3.37) and (3.39) on the non-conforming edges and on each conforming subregion $\mathbf{\Omega}_p$. Because we try to reduce the bandwidth of the boundary system, it is natural to take advantage of the zeros of the boundary matrix and to use a sparse matrix technique, coupled with an iterative solver. One of the key issues is to define data structures for these matrices that are well suited for efficient implementation of standard iterative methods. In our implementation, we used the sparse matrix format introduced in Saad (1995).

A direct solver is advantageous only when the cost of factoring the boundary matrix can be spread over a large number of solutions. In this case, only the cost of a back-substitution using the factored boundary matrix becomes dominant.

3.6 Refinement criteria

There are two key features in the development of adaptive meshes:

1. the non-conforming discretization
2. the error estimators that serve as criteria for refinement.

The first role of the error estimators is, of course, to provide an estimate of the actual error on a per element basis, as well as globally. Comparison of the elemental error estimates provides criteria as to which elements must be further refined. The estimate must be efficiently calculated, such that it may be used as a post-processing step as well as in the course of the calculation, such that time-dependent mesh refinement can proceed. Adaptive calculations involved decisions which must be taken ”on the fly”, based on some succinct information. This information can usually only be based on data from the last available step, but must be able, in some way, to predict how the solution converges, in order to direct refinement in the right direction.

We consider in a few types of refinement criteria based on the *a posteriori error estimators* introduced by Babusks and Dorr (1981), Berger and Oliger (1984) and Mavriplis (1990).

The first refinement criterion(gradient) we use is the simplest:

1. refine everywhere where the solution gradients exceed a certain tolerance.

It is required that

$$\|\nabla u^{(k)}\|_{\mathcal{L}^2(\mathbf{\Omega}^k)} \leq \epsilon \|u_h\|_{\mathcal{H}^1(\mathbf{\Omega})}, \tag{3.97}$$

where ϵ is the discretization tolerance. In high order methods, we have a locally global quality which provides sufficient information to estimate errors without excessive additional calculation. The gradient calculation in this case, is not limited to a few points but rather tied to the size of the element, which can be quite large for spectral elements.

The second refinement criterion(extrapolation) was proposed by Mavriplis (1990) and is based on the calculation and extrapolation of the spectrum of the Legendre discretization, to estimate the error as well as to predict convergence rates. The error estimators proposed are single mesh *a posteriori error estimators* in contrast to those of Babuška and Szabo. The concepts used by Mavriplis in the following error estimators are simple. Typically, if the numerical approximation to the spectral element Ω^k is written as:

$$u_h(x) = \sum_{n=0}^{N} a_n L_n(r), \tag{3.98}$$

where L_n is the Legendre polynomial of degree N and a_n's are the spectral coefficients we have:

$$\epsilon_{est} = \left(\frac{a_N^2}{\frac{1}{2}(2N+1)} + \int_{N+1}^{\infty} \frac{[a(n)]^2}{\frac{1}{2}(2N+1)} dn) \right)^{1/2}_{(\boldsymbol{\Omega}^k)} \leq \epsilon \|u_h\|_{\mathcal{L}^2(\boldsymbol{\Omega})}, \tag{3.99}$$

where $a(n)$ is the Legendre polynomial spectrum and $\boldsymbol{\Omega}$ is the domain. The two contributions to equation (3.99) are generated by the approximation error due to truncation, and the approximation error due to the quadrature and the best polynomial approximation (equation (A.28). The function $a(n)$ is a least squares best fit of the last six points of the spectrum to an exponential decay: $a(n) \sim c_e e^{-\sigma_e n}$. The decay rate σ_e indicates insufficient resolution if $\sigma_e < 1$ and good resolution if $\sigma_e > 1$. The refinement process uses the decay rate to decide whether it increases the number of elements and decreases accordingly the polynomial degree, or whether it has to move elements and reconstruct the grid.

The reader may be referred to Appendix A for an introduction of the error estimators and the derivation of the actual error incurred by the spectral element approximation $\|u - u_h\|$.

The third refinement criterion(spectrum) (Henderson, 1994) is based also on a local polynomial spectrum. In equation (3.99) the main contribution comes from coefficients of order N. By summing the tail of the spectrum, we can require that the spectrum satisfies the discretization tolerance. The tail of the spectrum can form an estimate of the approximation over $\|u - u_h\|$. First, we average over polynomials in x and y to produce an equivalent one-dimensional spectrum:

$$\text{trace spectrum} = \sum_{i=0}^{N-1} |a_{i,N-1}| + \sum_{j=0}^{N-1} |a_{N-1,j}| - |a_{N-1,N-1}|, \tag{3.100}$$

and subsequently the refinement criteria becomes:

$$\left(\sum_{i=0}^{N-1} |a_{i,N-1}| + \sum_{j=0}^{N-1} |a_{N-1,j}| - |a_{N-1,N-1}| \right)_{(\boldsymbol{\Omega}^k)} \leq \epsilon \|u_h\|_{\mathcal{L}^2(\boldsymbol{\Omega})}. \tag{3.101}$$

From the numerical results (see numerical results table 3.1), we can conclude that the spectrum-based (extrapolation) estimate is nearly equivalent to the trace spectrum, but the trace spectrum is easier and faster to compute.

The refinement decision is based on the above presented refinement criteria. First, we compare elemental error estimators to a globally acceptable level of error, set once for the whole run. Second, the elements with error over the acceptable level are marked for refinement. Another indicator for the refinement decision is the elemental error estimator. In order to determine which element has the greatest need for refinement, we compare the elemental error estimators of the neighbouring elements. Based on this comparison, we decide which element should next be refined. At this moment our implementation does not limit the number of elements K, or the total number of degrees of freedom created during the refinement process. In a future extension of the implementation, the maximum value of N, the order of polynomial, K, the number of elements, and $TOTAL\_DOF$ the total number of degrees of freedom, could be imposed. When the maximum values of parameters mentioned above parameters will be reached, refinement will be prohibited. Also, the decision to refine by adding elements, or increasing the order of the polynomial, could be based on the value of the σ_e decay rate of the spectrum in each element. If $\sigma_e > 1$ we should increase the polynomial order, and if $\sigma_e < 1$ we should increase the number of elements K. These are referred as p and h refinements to the finite element community. The decision to coarsen elements is not easy. The spectrum decay can predict convergence, but does not have the ability to predict convergence, if one remove elements. For now, coarsening is limited to remove the children elements and to activate the parent elements (see Voxel Data Base in Chapter 4).

3.7 Numerical results

The mortar method has been implemented, using a combination of C++ and Fortran77 languages. C++ for the adaptive bookkeeping, and Fortran77/C++ for the basic numerical modules. We illustrate the performance of the method for several steady and unsteady tow-dimension problems.

3.7.1 Gaussian distribution on a uniform grid

The first example is a Gaussian distribution on a uniform grid (Greengard and Lee, 1996). We consider the equation

$$-\Delta u = (400^2 r^2 - 800)e^{-r^2/\sigma^2}, \tag{3.102}$$

where $r^2 = x^2 + y^2$ and $\sigma = \sqrt{2}/20$, for which the exact solution is given by:

$$u(x,y) = e^{-r^2/\sigma^2}. \tag{3.103}$$

The computational domain is $\mathbf{\Omega} = [-0.5, 0.5] \times [-0.5, 0.5]$ and the homogeneous boundary conditions $u(x,y) = 0$ on $\partial\mathbf{\Omega}$. First, we compute the solution on a uniform refined grid, and then, by using the different error estimators defined in the previous section, we refine the mesh and compute the solution on the new grid using the mortar implementation.

In table 3.1 the relative errors and the estimators are shown. This table shows that the error estimates, spectrum and extrapolation, are nearly equivalent. Because the spectrum estimator is easier and faster to compute, this method will be used as the basis method for the mesh refinement. In figure 3.10 we illustrate the adaptive mesh refinement based on two different refinement criteria: polynomial spectrum and extrapolation. In this case both methods produce roughly equivalent discretizations, about 400 elements. The performance of the adaptive mesh generation method is illustrated through figures 3.10, 3.11, 3.12 and 3.13. For different polynomial orders we measure the relative errors in both conforming and non-conforming cases. We see that the number of the points used to achieve the same relative error, is less in the non-conforming case than in the conforming case. The evolution of the number of the mesh points based on the imposed relative error, for non-conforming and conforming mesh, for different polynomial orders, is illustrated in figure 3.14.

Table 3.1. Error estimates (spectrum, extrapolation) and relative errors ((∞, GL), $(\mathcal{L}^2)$, $\mathcal{H}^1$ norms) for solving the Poisson equation on a uniform grid with polynomial order N=9 elements:

Dofs	Spectrum	Extrapolation	$\|u-u_h\|_{\infty,GL}$	$\|u-u_h\|_{\mathcal{L}^2}$	$\|u-u_h\|_{\mathcal{H}^1}$
324	0.0896632	0.109066	0.0532076	0.0153128	0.132968
1296	0.00347407	0.00343821	0.000580771	9.04734e-05	0.00538864
5184	1.59027e-05	1.59011e-05	2.91117e-06	2.67009e-07	3.59727e-05
20736	5.94265e-09	5.94261e-09	2.51794e-09	2.54511e-10	6.94391e-08

Table 3.2. Relative errors ((∞, GL), $\mathcal{L}^2$, $\mathcal{H}^1$ norms) for solving the Poisson equation on a non-conforming mesh. The adaptive mesh generation is based on the local Legendre polynomial spectrum:

Dofs	$\|u-u_h\|_{\infty,GL}$	$\|u-u_h\|_{\mathcal{L}^2}$	$\|u-u_h\|_{\mathcal{H}_1}$
324	0.0532076	0.0153128	0.132968
1296	0.000580771	9.04734e-05	0.00538864
2268	2.91124e-06	2.67035e-07	3.59741e-05
8100	2.47615e-09	2.64522e-10	7.02348e-08

In table 3.2, we show for $N = 9$ the relative errors for the Gaussian distribution after the refinement of the grid based on the spectrum error estimator. A seven level quad-tree is used to describe the mesh (see chapter 4).

3.7.2 *Singularity problem*

The second example we consider is a singularity Poisson problem:

$$-\nabla^2 u = 1,$$

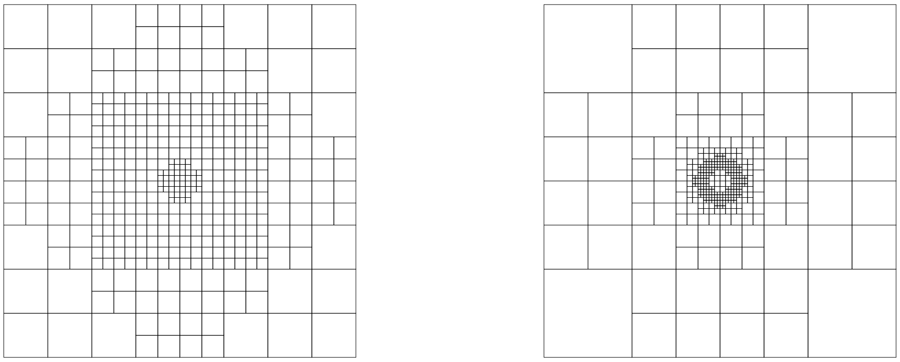


Fig. 3.10: Adaptive mesh generation: $N = 7$, (*left*) adaption based on the local Legendre polynomial spectrum with a tolerance of $\epsilon = 0.23 \times 10^{-10}$; (*right*) adaption based on solution gradients with a tolerance of $\epsilon = 0.09$.

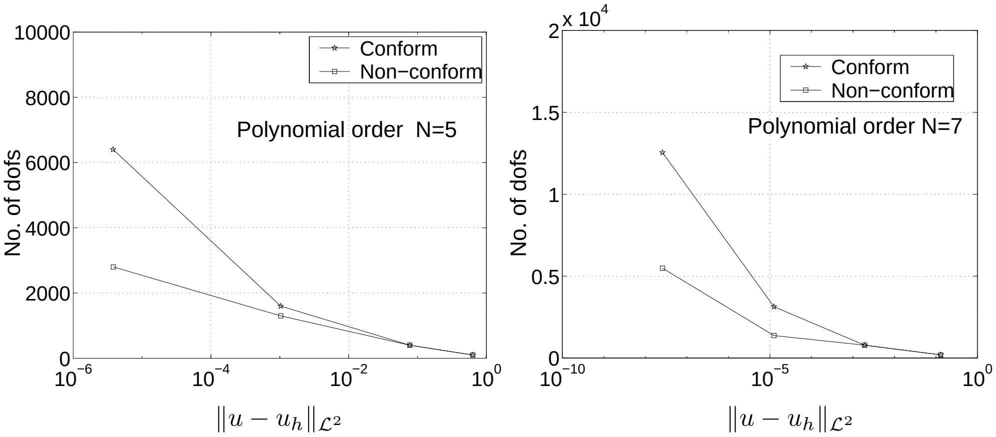


Fig. 3.11: Performance of refinement for $N = 5$ and $N = 7$; adaption based on local Legendre polynomial spectrum.

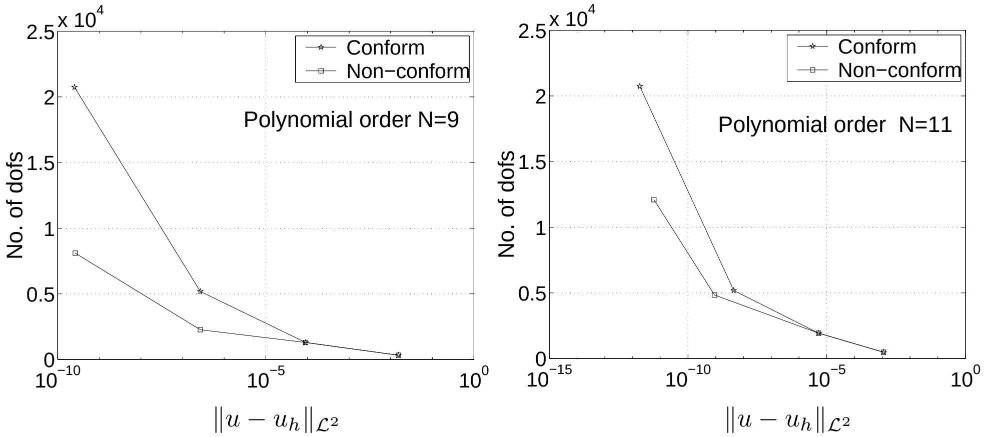


Fig. 3.12: Performance of refinement for $N = 9$ and $N = 11$; adaption based on local Legendre polynomial spectrum.

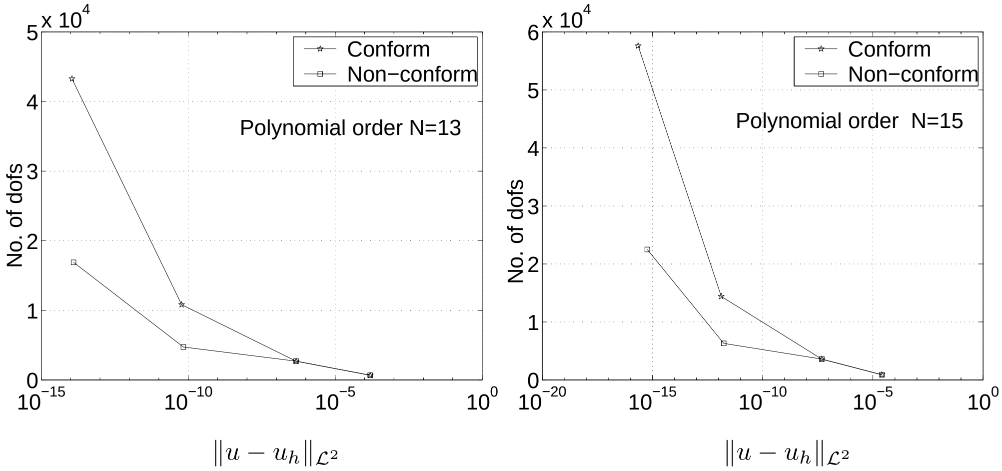


Fig. 3.13: Performance of refinement for $N = 13$ and $N = 15$; adaption based on local Legendre polynomial spectrum.

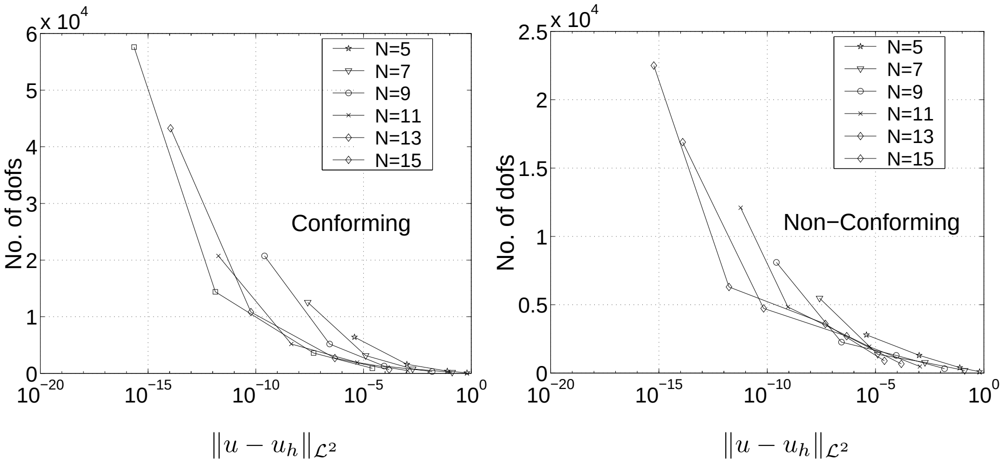


Fig. 3.14: Adaptive mesh generation: (*left*) conforming case , (*right*) non-conforming case. The adaption is based on local Legendre polynomial spectrum, using different polynomial orders.

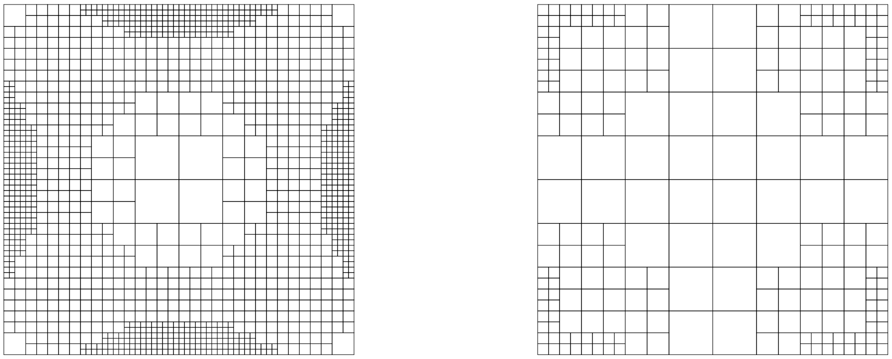


Fig. 3.15: Adaptive solution of the Poisson equation with corner singularities: (*left*), adaption based on the solution gradients with a tolerance of $\epsilon = 0.04$; (*right*), adaption based on the local Legendre polynomial spectrum with a tolerance $\epsilon = 0.8 \times 10^{-11}$.

with homogeneous Dirichlet boundary conditions on $\mathbf{\Omega} = [-1,1] \times [-1,1]$. The solution exhibits weak singularities of the form $r^2 log(r)$ as $r \longrightarrow 0$, where r is the radial distance from the corner. However, the solution gradients are largest along the edges of the domain, where the structure of u is rather simple. To reduce the error for this case, we have to rely only on the local polynomial spectrum refinement. The mesh refinement based on solution gradients completely misses the location where the errors are largest. The structure of the solution is shown in figure 3.15.

3.7.3 Smooth problem

The next example we consider, is the following Helmholtz smooth problem:

$$-\nabla^2 u + \lambda^2 u = f(x,y) \text{ on } \mathbf{\Omega} = [0,1] \times [0,1],$$

with Dirichlet boundary conditions:

$$u(x,y) = u_{exact}(x,y) = e^{\frac{\lambda}{\sqrt{2}}((x-1)+(y-1))} \text{ on } \partial\mathbf{\Omega}.$$

Since, the solution exhibits a sharp, but smooth, boundary layer near the corner (1, 1) (see figure 3.17), for large λ, we expect a smooth solution, and hence a smooth and exponentially decaying spectrum. In figure 3.16 we illustrate the mesh generated based on two refinement criteria for $\lambda = 50$. The grid is refined in approximately the same location, near the corner (1, 1), and to the same depth for a given discretization tolerance. The left one is based on the solution gradients and the right one is based on the local Legendre polynomial spectrum. Figure 3.17 illustrates the solution, before and after the refinement. The left images show that a further resolution is needed near the corner (1,1). Using *h-refinement*, the overall error has been reduced by at least two orders of magnitude. The right image shows the solution after refinement.

3.7.4 Two-dimensional linear unsteady convection problem

The last example presented here is a two-dimensional linear unsteady convection problem (Vreugdenhil and Koren, 1993). We consider the unsteady rotation of a Gaussian hill described by the convection equation in two dimensions in the domain $\mathbf{\Omega} = [-1,1] \times [-1,1]$:

$$\frac{\partial c}{\partial t} + u\frac{\partial c}{\partial x} + v\frac{\partial c}{\partial y} = 0, \text{ t } \in [0,1], \tag{3.104}$$

where the velocity field describes a pure rigid-body rotation:

$$u = -\omega y, v = \omega x, \omega = 2\pi.$$

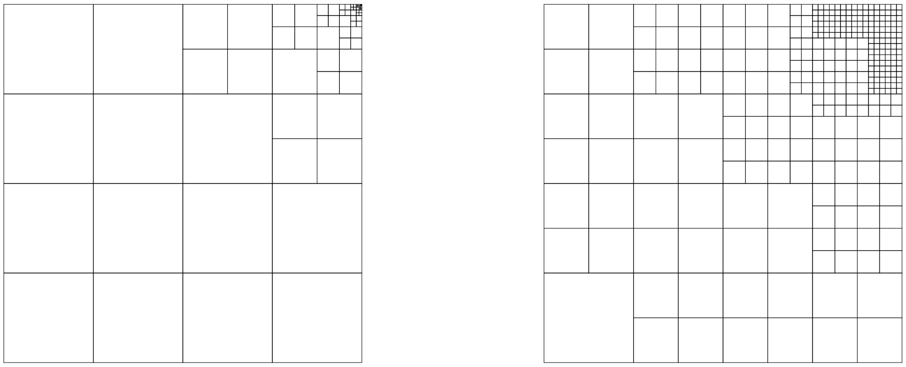


Fig. 3.16: Adaptive solution of the Helmholtz problem: (left) adaption based on the solution gradients with a tolerance $\epsilon = 0.015$; (right), adaption based on the local Legendre polynomial spectrum with a tolerance $\epsilon = 0.8E \times 10^{-12}$.

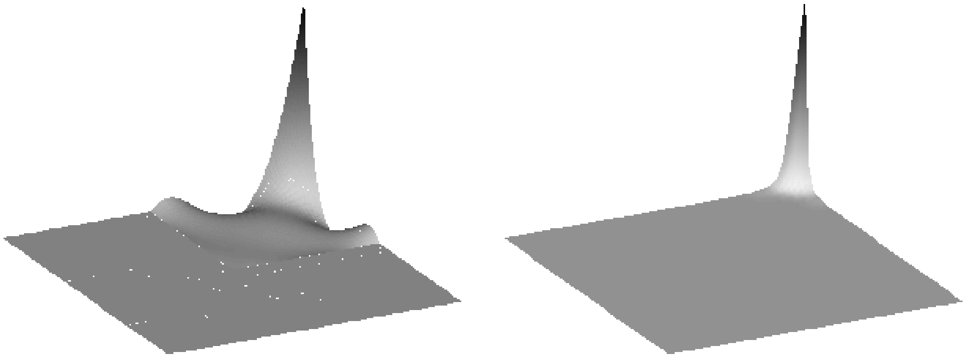


Fig. 3.17: Adaptive solution of the Helmholtz problem: (left) the solution before refinement; (right), the solution after *h*-refinement, adaption based on the local Legendre polynomial spectrum with a tolerance $\epsilon = 0.8 \times 10^{-12}$.

The initial condition is a Gaussian distribution, given by:

$$c(x, y, 0) = 0.01^{4r^2}, r = \sqrt{(x + \frac{1}{2})^2 + y^2}.$$

The exact solution is given by:

$$c(x, y, t) = 0.01^{4r^2}, r = \sqrt{(x + \frac{1}{2}\cos\omega t)^2 + (y + \frac{1}{2}\sin\omega t)^2}.$$

This problem is solved adaptively, starting with a coarse $K = 4$ equal-sized element grid shown in figure 3.19 (left). For the time discretization the Crank-Nicolson scheme is used.

Table 3.3. Discrete maximum error $\|c - c_h\|_{\infty,GL}$ for the rotation of a Gaussian hill, time $t \in [0, 1]$; N is the degree of approximation. The maximum number of elements is 22 for a tolerance $\epsilon = 4.0E \times 10^{-2}$:

Time-steps	$N = 8$	$N = 12$	$N = 14$	$N = 16$
128	7.60E-02	4.52E-03	1.50E-03	0.87E-03
256	4.36E-02	3.25E-03	2.50E-03	1.85E-03
512	2.35E-02	2.10E-03	1.50E-03	1.10E-03
1024	1.30E-02	1.1E-03	0.9E-04	0.7E-04

In table 3.3 the discrete maximum error $\epsilon = \|c - c_h\|_{\infty,GL}$ is given for different time steps. The results show that the solution becomes more accurate as the time step decreases. Even in the non-conforming case, the errors due to time discretization are still large (for conforming case see Vreugdenhil and Koren (1993)). To eliminate them and prove that we get more spatial accuracy, we will consider *a modified exact solution*. To compute the *modified exact solution*, the equation is solve for a polynomial order $N = 22$, a number of element $K = 64$ and time step=4096. Computing the norm $\|c_{mex} - c_h\|_{\infty,GL}$ we are able to eliminate the error due to time discretization. Table 3.4 illustrates the improvement in accuracy for the 128 time step cycle.

Table 3.4. Discrete maximum error $\|c_{mex} - c_h\|_{\infty,GL}$ for the rotation of a Gaussian hill; N is the degree of approximation. The maximum number of elements is 22 for a tolerance $\epsilon = 4.0E \times 10^{-2}$:

Time-steps	$N = 8$	$N = 12$	$N = 14$	$N = 16$
128	2.60E-08	2.12-08	1.6E-08	1.35E-08

Figures 3.19, 3.20 and 3.21 show several intermediate adaption steps. In figure 3.19 *(left)* the initial mesh is shown, together with the collocation points grid after the first refinement *(right)*. In this case, the tolerance is $\epsilon = 1.0E \times 10^{-3}$, which increases the number of refined elements. The Gaussian hill is rotated quite accurately, and there are no oscillations due to the low number of elements of the initial mesh.

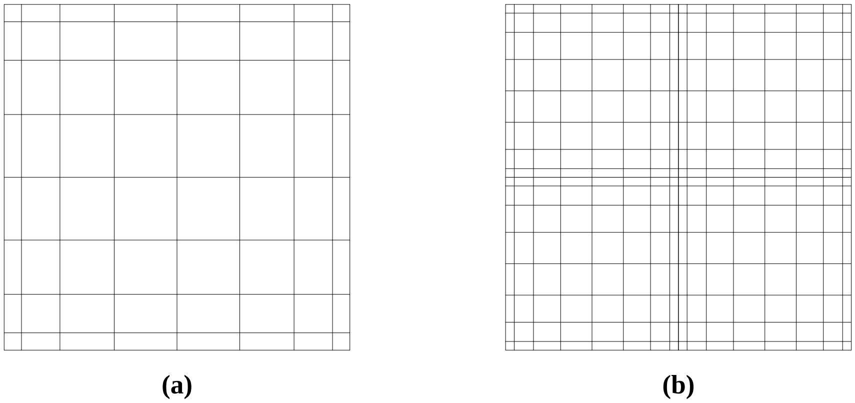

(a) **(b)**

Fig. 3.18: Spectral element mesh for $N = 9$; (a) the parent element, (b) the children elements.

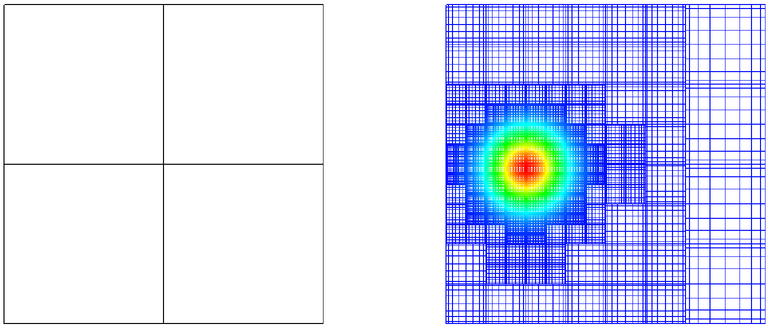


Fig. 3.19: Unsteady rotation of a Gaussian hill: (*left*) the initial conforming mesh; (*right*) $t = 0.00$, the spectral non-conforming mesh after the first refinement.

After the first time step, the mesh is dynamically adapted. The refinement algorithm tracks the movement of the Gaussian hill quite accurately. New elements are mainly added or removed from the mesh based on the refinement Legendre polynomial spectrum criteria. The new elements are added around the boundary of the hill, or where, if that is the case, the oscillations appear. To coarsen the mesh, we impose that elements that belong to the same parent element have to pass the coarsening test: the local spectrum of each child element has to be smaller than a specified threshold = $\epsilon = 1.125E \times 10^{-3}$. The regions with small solution gradients are candidates for coarsening. When we generate new elements , due to refinement, the mesh and the data of a parent element is interpolated to a new mesh and data that belong to its children elements. Also, when the children elements are pruned, due to coarsening, the mesh and data are interpolated back to their parent element. In both case, the interpolation is spectral in (x, y). Figure 3.18 shows the spectral mesh of a parent element (a), and of its children (b) for $N = 9$.

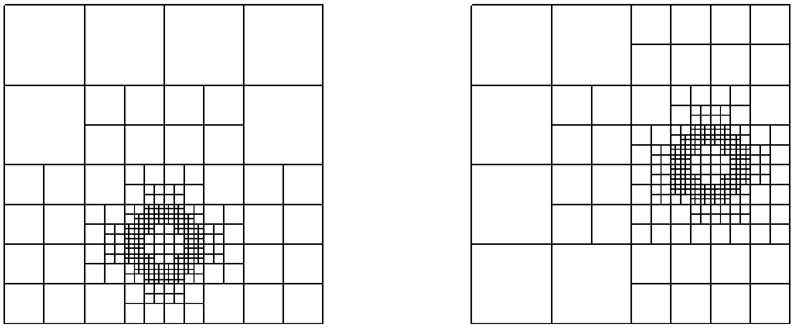


Fig. 3.20: Unsteady rotation of a Gaussian hill: (*left*) $t = 0.20$; (*right*) $t = 0.50$.

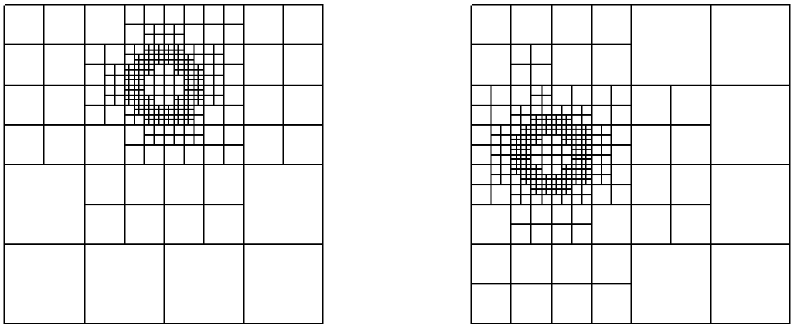


Fig. 3.21: Unsteady rotation of a Gaussian hill: (*left*) $t = 0.75$; (*right*) $t = 1.0$.

3.8 Conclusions

In this chapter, we outlined the basic features of the mortar discretization. The new non-conforming method relieves the spectral element method of its geometric and functional restrictions, namely that the interface between two adjacent elements must be conforming. Also, the mortar discretization enables functional disparity at element interfaces, by using a $\mathcal{L}^2$ projection from the mortar to the element edges, and by enforcing exact continuity at element vertices to maintain the jump in functions between elements small. Since the $\mathcal{H}_0^1(\mathbf{\Omega})$ continuity is still ensured, the mortar formulation guarantees optimality. In our test cases we achieved an exponential convergence, thus the optimality has been demonstrated.

An interesting part of the method is found in the refinement criteria that are based on posteriori error estimators for the spectral element method. This provides a heuristic error estimate that is independent of the system being solved. The estimators are based on an analysis and extrapolation of the Legendre spectrum of the numerical solution, as a post-processing step which takes very little computational effort. They are quite sharp, differing from the $\mathcal{L}^2$ error by only a small multiplicative factor. This can be a good indication of the error we want to achieve through adaptive mesh generation. For example, imposing a threshold of order 10^{-9} for the spectrum-based estimates, we can estimate an order of the solution's accuracy proportional to 10^{-9}. In the case of corner singularities, the adaptive solution based on the gradient does not produce the right result. The local polynomial spectrum indicates the correct location for refinement, while the magnitude of solution gradients can be misleading. In this case, mesh refinement based on solution gradients completely misses the location where the errors are largest. For unsteady problems, the errors due to the time discretization can be large, and to prove that the accuracy of the solution is improved, we have to eliminate them. A *modified exact solution* has to be computed.

In summary, the mortar discretizations represent a significant advance for spectral element methods, which offer new possibilities for time-dependent moving boundary calculations. The error estimators are powerful concepts, that can be used automatically in an adaptive refinement scheme. Using this non-conforming spectral element method, the mesh generation task in computational modelling is minimized. Generating new elements only in regions where the resolution is inadequate, it will have a huge impact on the memory usage and computational times. Less memory storage will be needed to store the elements and the computational time will decrease. The four test case problems illustrate the advantage of the present error estimators: the capability to predict convergence behavior is crucial to an efficient refinement process. The use of single mesh *a posteriori* error estimates, is an improvement to the multiple mesh error estimates. The estimates can be used as a post-processing step as well as in the course of the calculation, such that time-dependent mesh refinement can proceed.

Chapter 4

Software Implementation

In this chapter, the architecture of the software system and the main data structures, C++ classes, used for the implementation of the mortar method are introduced. The dynamic data structure, needed to keep the dynamically refined mesh, is also described. Since the number of the C++ classes is large, only a few classes will be presented here. Also, the basic operations required to implement the common procedures in spectral element methods, will be described.

4.1 Introduction

We present a software infrastructure for the implementation of the mortar method used to solve a system of partial differential equations and the integration of this infrastructure with an adaptive spectral h mesh-refinement method. The development of large codes for scientific computing is known to be a comprehensive and time consuming process. Moreover, large stand-alone Fortran codes dominate the field of scientific computing. Long-term evolvement of such codes is usually an error-prone and expensive process, unless the original software is carefully designed for future extensions. There is a continuing debate regarding the programming language of choice among competing languages such as C++, Fortran90 and their extensions. Not only must the software be portable across a wide variety of computing platforms , but other issues such a easy incorporation of new data structures, easy of programming interfacing via user interfaces and code maintenance must be considered, before choosing a particular programming language and style. Turning to the field of computer science, years of experience indicate that software reliability can be significantly improved by a modular design that encourages reuse of a code. Modularity and code reuse can be achieved by using traditional implementation in Fortran. However, this requires careful and complicated considerations. Object-Oriented design and Programming (OOP) techniques offer a much easier and more efficient methodology for obtaining the above mentioned goals. The improvement of C++ compilers makes the C++ language a reasonable alternative to Fortran, that offers the most important Object-Oriented constructs along with satisfactory computational efficiency. C++'s major features as :

1. support for data abstraction and object-oriented programming
2. classes and abstracted classes that encapsulate data and functions
3. hierarchies of classes, based on inheritance and multiple inheritance
4. support for real-time polymorphism via virtual functions

5. dynamic memory allocation and deallocation
6. overloading mechanism for operators and functions
7. support for the creation of templates and generic functions
8. the Standard Template Library (STL) which provides a collection of generic data structure and algorithms,

offer a powerful implementation mechanism for most of the numerical code. However, we advocate a hybrid approach of using an Object-Oriented programming language such as C++ for handling the data structure for adaptive spectral element methods, and a combination between Fortran (BLAS (Lawson *et al.*, 1979), LAPACK (Andersen *et al.*, 1999), SEPRAN (Segal, 1995)) and C++, for the linear solver and mesh adaptivity. Using this approach, we take optimal advantages of both programming languages. Interfacing C++ and Fortran for a heterogeneous cluster of machines, is not an easy task, but developing a few wrappers for the Fortran subroutines and functions make the task easier. This approach enables us to provide a flexible C++ interface by which one can incorporate new spectral element algorithms and new problem definitions, as requirements grow with time, without compromising the performance of Fortran77 needed in numerical calculations.

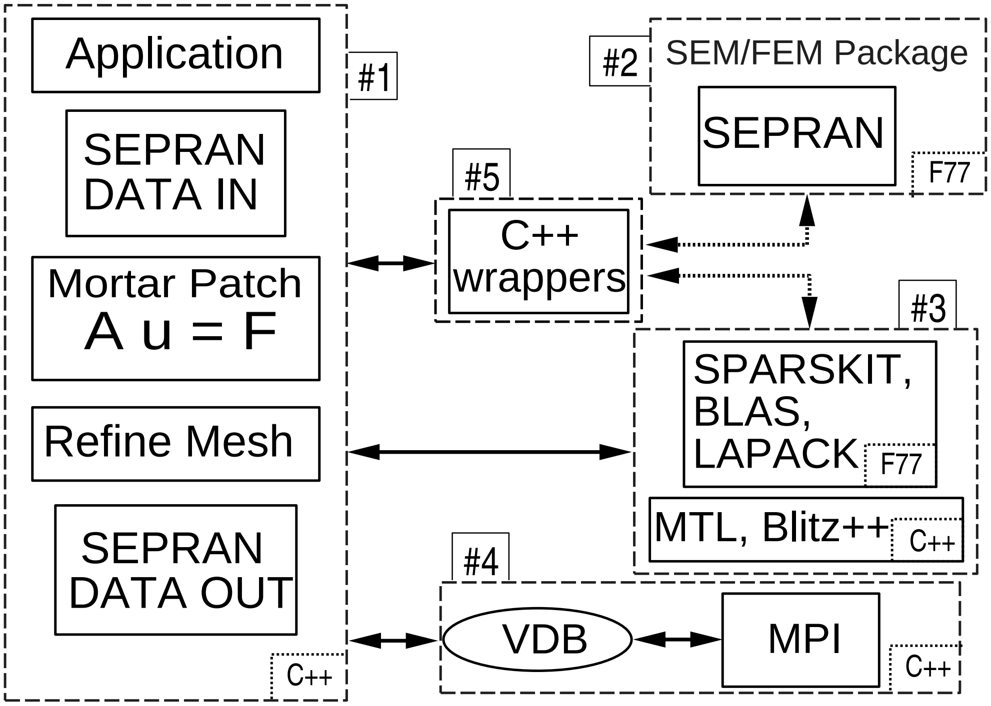


Fig. 4.1: Interaction of the components in our hybrid implementation.

Developing an adaptive spectral element package is a challenging and time consuming task. Using the hybrid approach proposed, will allow us to reuse an existing SEM/FEM package, called SEPRAN, which is implemented in Fortran77. Figure 4.1 shows the main components of our hybrid implementation. The component $\sharp 1$ implements:

1. the interface to the SEPRAN ($\sharp 2$) package,
2. the mortar spectral element method,
3. the refinement criteria and the mechanism of deciding if the mesh will be refine/pruned.

Also, the $\sharp 1$ component provides to SEPRAN, for each time step, the input mesh and the old solution, receiving back from SEPRAN only the elemental matrices. SEPRAN has already available a large number of discrete operators that can be used by the mortar extension, which make the hybrid implementation of the mortar method available to many applications. In chapter 5 we illustrate this, by applying the mortar method to two applications which have been implemented in SEPRAN. A few direct and iterative solvers are used: MTL (iterative), LAPACK (direct), SPARSEKIT (iterative), which are implemented by component $\sharp 3$. Because some solvers are implemented in Fortran77, and SEPRAN provides only a Fortran77 interface to its functions, a C++ wrapper is available, component $\sharp 5$, for Fortran77 functions calls. The dynamic mesh and the communication between the processors, in the parallel version of the mortar method, are implemented by component $\sharp 4$. Each time the mesh is updated, the voxel data base VDB (section Mesh Data Structure) will be updated and the information on the connectivity of the mesh will be spread between the processors. In the next sections, only the essential C++ classes that implement the above components are introduced.

4.2 Basic operations for the spectral element method

The operations described in section 3.5 were all local, as they involved a single spectral element and no information was coupled with any other element. In general, however, we are interested in solving second-order partial differential equations, which requires that some form of continuity is maintained between elemental regions. A sufficient, although not necessary, condition to satisfy the continuity requirements is to make the global approximation $\mathcal{C}^0$ continuous. In doing so, we couple information from one element to another and, therefore, operations involving the inverse of a matrix system such as the forward transformation become global. To set up matrix inverse, we need to perform what is known as a *direct stiffness summation* or a *global assembly* operation which constructs a $\mathcal{C}^0$ continuous global expansion basis from the elemental basis functions. In practice, we still perform most operations in a local fashion within each element and then sum the contributions to form the global system. However, to do this we need a mapping which assembles the global system from the local system. In spectral element methods, global data is stored as a flat, unstructured array. The basic data structure used to relate the mesh to entries in this array is a table that identifies the global node number of a local node within each element. We define the *local degree of freedom* as the elemental expansion coefficients over all elements. The table of indices can be stored as a two-dimensional array of integers:

`gmap[k][i]` = global index of local datum i in element k.

For the conforming case, the number of degrees of freedom in the mesh `ndof` and the number of degrees of freedom associated with each element `edof` are constant. To perform global operations, we need a layer of indirection between the global data and the local data. For example, if the matrix $\mathbf{A}$ is the assembly matrix (keeps the relation between the local and global degrees of freedom) the operations represented by $\mathbf{u}_l = \mathbf{A}\mathbf{u}_g$ are called the ”scatter” (Deville *et al.*, 2002) operation, where $\mathbf{u}_l$ are the local degrees of freedom and $\mathbf{u}_g$ are the global degrees of freedom. The operation $\mathbf{u}_g = \mathbf{A}^T\mathbf{u}_l$ is called a ”gather” operation. The following is a template for any such a computation:

```
// Loop over elements
for(k=0;  k < K ; k++) {

    // Copy global data to local
    for(i=0;    i <  edof;  i++) {

        // Scatter operation from global to local
        ul = ug [gmap[k][i]];

    }
} // end loop over elements

// Loop over elements

for (k=0;  k  < K  ;  k++) {

    // Accumulate the data to global
    for(i=0;    i <  edof;  i++) {

    // Gather operation from local to global
        ug [gmap[k][i]] += ul[i];
    }
} // end loop over elements
```

The global assembly procedure primarily involves boundary node connectivity, since the interior nodes may be independently numbered as global degrees of freedom. Due to the static condensation, the assembly procedure only involves the boundary nodes since the interior nodes may be removed from the full matrix problem. In this case, the global assembly procedure is numerically evaluated as:

```
for (k=0;  k  < K  ;  k++) {

    // Accumulate the data to global
    for(i=0;   i <  bedof;  i++) {

    // The gather operation AˆT (transpose of A)
    //  of the boundary nodes
        ug [bmap[k][i]] += ul[i];
```

```
        }
    } // end loop over elements
```

where `bmap[k][i]` is the mapping of the boundary nodes index to the global index, and `bedof` is the number of boundary degrees of freedom. In our example, we assumed that the local degrees of freedom are ordered, such that the boundary nodes are listed first. If we know how to construct `bgmap[k][i]`, it is a trivial extension to generate `gmap[k][i]`, simply by adding a unique block of global degrees of freedom equal in length to the number of interior nodes within the element. Figure 4.2 illustrates the global numbering of the boundary nodes. For example, the element Ω^1 has a `bmap[]` array that contains the values:

```
    // The boundary array
    bmap[8]={1,2,4,5,10,14,13,3} ;
```

which represent the global numbering of the boundaries nodes.

To make this data structure suitable for the non-conforming elements, we introduce two generalizations. In the non-conforming case, the number of degrees of freedom can be different per element. To allow this we have to introduce the array `edof[k]`, which contains the local degrees of freedom of the elements. Since we want to allow each local degree of freedom to depend on an arbitrary combination of the global degree of freedom, we need to introduce two new arrays `ldof`, `assembly` and a new dimension to the `gmap` array:

```
    ldof[k][i]     = number of global dependencies for
                     local datum i in element k,

    assembly[k][i] = array of coefficients for global to
                     local data mapping,

    gmap[k][i][j]  = global index of the jth dependency of
                     local datum i.
```

In the non-conforming case, we need also a transformation matrix $\mathbf{Q}^k$ between global and local degrees of freedom. The procedures to construct the scatter and the gather operations become:

```
    // Loop over elements
    for(k=0;   k   <   K ;  k++) {

    // Copy global data to local
        for(i=0;   i <   edof[k];  i++) {

            // Get the coefficients for the local data
            double  *Qk = assembly[k][i];

            for(j=0;    j <    ldof[k][i];   j++) {

                // Scatter operation
                ul[i]  += Qk[j] * ug [gmap[k][i][j]];
```

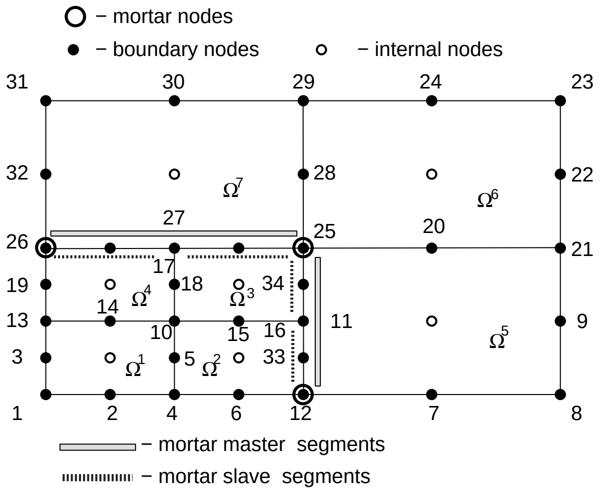


Fig. 4.2: The illustration of the master and slave segments together with the global numbering of the boundary nodes.

```
        } // end loop over ldof

    } // end loop over elements degrees of freedom

} // end loop over elementen


// Loop over elements
for(k=0;   k   <   K ;  k++) {

// Copy global data to local
    for(i=0;   i <   edof[k];  i++) {

        // Get the coefficients for the local data
        double  *Qk = assembly[k][i];

        for(j=0;    j <    ldof[k][i];   j++) {

            // Gather  operation
            ug [gmap[k][i][j]] += Qk[j] * ul[i];

        } // end loop over ldof

    } // end loop over elements degrees of freedom

} // end loop over elementen
```

We can conclude that the procedures to assemble the global system in both cases: conforming and non-conforming differ only in the way we transform between the local and global systems. The matrix $\mathbf{Q}^k$ offers a flexible scheme for storing the global solution and reconstructing the local one. The computational overhead and additional storage is the price we pay for new capabilities: arbitrary connectivity in the mesh and variable order of the local basis functions, which are the key ingredients for adaptive $h-p$ refinement techniques.

4.3 Data structure

In addition to the Object-Oriented Programming aspects, there are equally important issues to the underlying data structure that are essential to adaptive mesh strategies. The complexity of the data structure increases significantly when adaptive meshing is included. The first data structure, we present here, is the basic data structure required for the manipulation of the elements. For any element, we denote the polynomial degree of basis functions as `Nr, Ns, Nz` , an identification number `Elemid` unique to each element, the mesh coordinates in physical space `xmesh`, `ymesh`, a binary node key `RefKey` used to identify the expanded elements after refinement, a double-ended queue `EdgesList` that contains the link to the element's edges, a double-ended queue `VertexList` that contains the vertices, and a local map `ElemMap` that relates the local nodes of the element to the interior-boundary nodes structure, used for static condensation. Here, `deque` is used as a generic container. It can be replaced by a container defined by the **Blitz++** of **MTL** libraries (see section 4.5).

An abstraction for any higher order spectral element can be created considering common behavior and attributes of different types of spectral elements. The attributes of an element are identified in C++ by the **Element** class as follows (see figure 4.3):

```
class    Element {

  protected:
    int           Elemid;       // element id
    int           RefKey;       // element key after refinement
    int           Type;         // element type identification
    int           Nr, Ns, Nz;   // points in each direction
    int           NrSol;        // the number of DOF
    int           NrPoints;     // points in element, (icount)
    int           *Global;      // global numbering, index2
    int           NrPresc;      // prescribed points (icountp)
    int           *PrescOff;    // prescribed points (index4)
    int           NrTrueSol;    // nr of true unknown (icnt)
    int           *TrueOff;     // offset true unknown, index
    int           *ElemMap;     // map  (nr,ns) to (nb,ni)
    double        **ElemMat;    // element stiffness matrix
    double        *ElemVc;      // element vector
    double        *ElemBc;      // boundary values, bc[nrpresc]
    double        **Xmesh;      // x mesh  in physical space
    double        **Xmesh;      // y mesh in physical space
    deque<Edge>   *EdgesList;   // Array of edges
    deque<Vertex> *VertexList;  // Array of vertices
```

```
  public:
    Element (void):Elemid(-1), Nr(0), Ns(0), Nz(0), NrSol(0),
                   NrPoints(0), Global(NULL), NrPresc(0),
                   PrescOff(NULL), NrTrueSol(0), TrueOff(NULL),
                   ElemMap(NULL), ElemMat(NULL), ElemVc(NULL),
                   ElemBc(NULL), Xmesh(NULL), Ymesh(NULL),
                   EdgesList(NULL), VertexList(NULL) {}
    Element Element (const Element&  elmt);
    Element operator = (const Element& elmt);
    ~Element();

    int     GetId (void)   { return  ElemId; }  // Get ElemId
    int     GetType (void) { return  Type; }    // Get Type
    int     GetNr (void)   { return  Nr; }      // Get Nr
    void    SetId (void)   { int  ElemId; }     // Set ElemId
    void    SetType( void) { int Type; }        // Set Type
    void    SetNr (void)   { int Nr; }          //Set Nr
    ...

    ...
    double** GetXmCoord (void) { return  Xmesh; }

    double   GetXmCoord (const int i, const int j) {
                  return Xmesh[i][j];}

    double** GetYmCoord (void) { return Ymesh; }

    double   GetYmCoord (const int i, const int j) {
                  return Ymesh[i][j]; }

    ...
}
```

The class above defines all variables and the member functions that access them. To avoid direct access to the data members of the class, we choose to alter them via the functions calls. Note that the `protected` access specifies in the base **Element** class provides both data protection and inheritance. The variables defined with `protected` specifier are available only to class **Element** and any other derived from it. Most of the variables and functions defined above are self-explanatory and annotated with comments. However, we point out that the data type `deque <Edge>` construct a `deque` of class **Edge**. Once a basic **Element** class has been defined, it is relatively easy to derive specific element types from it.

For example we can derive a rectangular element (see figure 4.3):

```
class   ElementQuad: public Element {

  protected:
    static  double* Weights;         // weights array
```

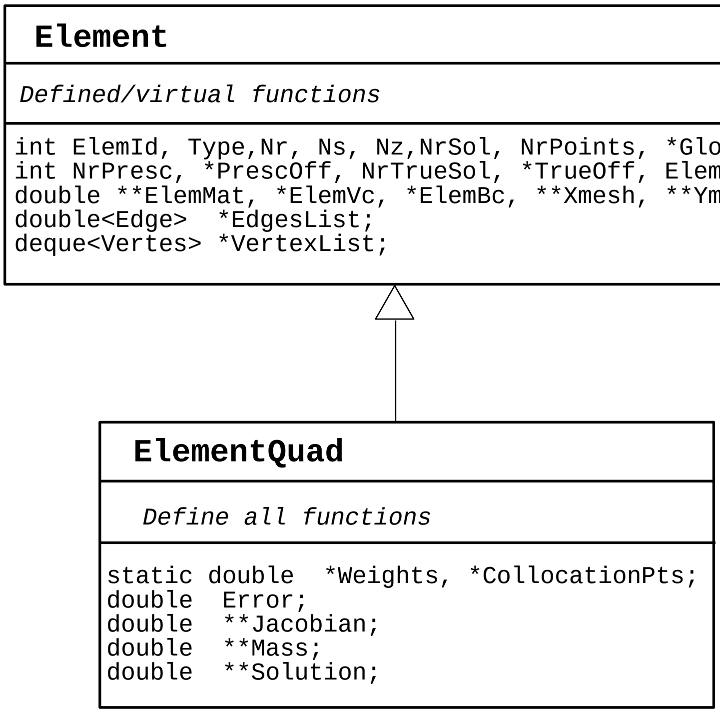


Fig. 4.3: An illustration of the derive procedure for the Element class hierarchy .

```
    static  double* ColloccationPts; // collocation points array
    double  Error;                   // error on the element
    double  **Jacobian;              // Jacobian matrix
    double  **Mass;                  // mass matrix
    double  **Solution;              // solutions matrix

  public:
    ElementQuad (int Id, int Nr, int Ns, int Type);
    ElementQuad (void);
    ˜ElementQuad ();

    void Weights (void);
    void CollocPoints (void);
    friend ostream& operator << (ostream& out, ElementQuad& e);
    ...

    ...
}
```

The overloaded `operator <<` is defined as a public member function to print out information about this particular element. Also, we define the arrays `weights` and `points` as `static`. In this case, all the elements **ElementQuad** have the same pointers that point to the same arrays with collocation points and weights.

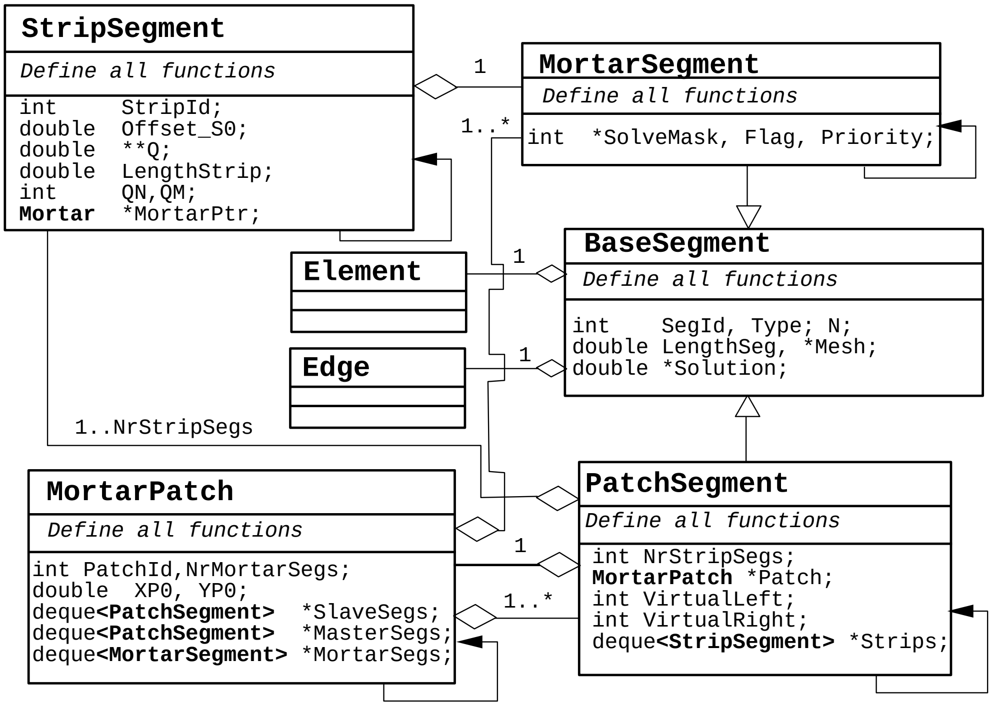


Fig. 4.4: The mortar class hierarchy in a UML diagram (Rumbauch *et al.*, 1998).

Another basic data structure, required for the implementation of the mortar method, is the class **BaseSegment** that describes the common attributes of the classes **MortarPatch**, and **MortarSegment**. The **BaseSegment** contains the basic structure of all the segments used in the mortar method as: the segment type `Type`, segment length `LengthSeg`, the parent edge `ParentEdge`, the solution on the segment `Solution`, the polynomial order on the segment `N`, see figure 4.4).Here is the definition of the **BaseSegment** class:

```
class   BaseSegment: {

  protected:
    int     SegId;       // segment id  number
    int     Type;        // segment type
    int     N;           // polynomial order - for the segment
    double  LengthSeg;   // segment length
    double  *Mesh;       // nodal cooordinates of the segment
    double  *Solution;   // solution on the segment
    Element *ParentElem; // parent element
    Edge    *ParentEdge; // parent edge

   public:
```

```
    BaseSegment (void): SegId(-1), Type(BI\_SECTION), N(0),
                        LengthSeg(0), Mesh(NULL), Sol(NULL),
                        ParentElem(NULL), ParentEdge(NULL) { }

    BaseSegment  operator = ( const     BaseSegment&  baseseg);
    BaseSegment  (const  BaseSegment& baseseg);
    ˜BaseSegment ();

    void CreateMesh (void);           // make mesh of segment
    void GatherSegment ( Edge* edge); // copy buffer to segment
    void ScatterSegment (Edge* edge); // copy buffer to edge

    bool operator <  (const BaseSegment &s)  const
                             { return SegId < s.ElemId; }
    bool operator == (const BaseSegment& s) const {
                             { return ElemId == s.ElemId; }
    bool operator != (const BaseSegment& s) const
                             { return ElemId !operator==(s); }
    bool operator >  (const BaseSegment& s)const
                             { return ElemId > s.ElemId; }
    ...

    ...
}
```

Another important class is the **MortarSegment** class. The structure required to describe a mortar edge can be represented with this class:

```
class   MortarSegment:   public BaseSegment {

  private:
    int  Flag;       // indicate the mortar status
    int  Priority;   // mortar priority
    int  *SolveMask; // solve mask of the mortar segment

  public:
    MortarSegment    (void);
    MortarSegment&   operator = (const MortarSegment& mp);
    MortarSegment&   (const MortarSegment& mp);
    ˜MortarSegment();
    ...

    ...
 }
```

For any mortar segment (edge), we need the polynomial degree of basis functions, the solve mask `SolveMask`, the length of the mortar $\hat{\gamma}$ in (x, y) coordinates, the solution on the mortar, the parent element the mortar belongs to, and the parent edge. Using the inheritance mechanism, a few attributes, of the above specified class, are defined by the **BaseSegment** class. The only new attributes that are declared in **MortarSegment** are: `Flag`, which indicates whether the mortar has been used, the mortar priority `Priority`, which indicates the order in which the mortar are used, and the solve mask `SolveMask`, which indicates the points on the mortar that are true degrees of freedom. For the sorting of the mortar edges, we use the boolean `operators` ”<”, ”==”, ”!=” defined in the **BaseSegment** class.

Since the refinement process create non-conforming edges, we need a class that implements this type of edges. The class is called **PatchSegment** and is derived from the **BaseSegment** class. The main attributes of the class are: the segment type `Type`, that indicates whether the current segment is a master of slave segment, the number of the integration strips associated with the current patch segment `NrStripSegs`, and the `deque` with the strip segments `Strips`. The class **PatchSegment** is defined as:

```
class   PatchSegment:  public BaseSegment {

  private:
    int                  NrStripSegs;  // nr. strip segments
    MortarPatch          *Patch;       // patch segment
    int                  VirtualLeft;  // nr. vertices, left
    int                  VirtualRight; // nr. vertices, right
    deque<StripSegment> *Strips;       // associated strips

  public:
    PatchSegment (int Id,  int  Type,  int  N,  double Len,
                 Element  *elmt,  Edge *edge , int VirtualLeft,
                 int VirtualRight, eque<StripSegment> StripsIn,
                 MortarPatch *PatchIn);

    PatchSegment (void);

    PatchSegment& operator = (const PatchSegment& mp);
    PatchSegment& (const PatchSegment&  mp);
    ~PatchSegment ();

    void ProjectQT  (Element *elmt, double *u); // Q^{T} * A
    void ProjectQ   (Element *elmt, double *u); // Q * A
    void MakeStrips (void);                     // make strips
    ...

    ...
}
```

Each mortar-slave segment combination is implemented with the **PatchSegment** class.

The next class that introduced is the **MortarPatch** class. In figure 3.8, we see that a patch can be described by three components Γ_l^k - the edge l of element k, γ_p - the mortar and $\hat{\gamma_p}$ - the integration strip. These components are represented by the **BaseSegment** class. The **MortarPatch** class is created from a file that contains the description of the mesh. Because we are using the Standard Template Library (**STL**) templates (Musser and Stepanov, 1994), in our case the `deque` container, a lot of information about the class **MortarPatch** can be obtained using the functions of the **STL** templates. For example: we want to know how many mortars we have in the current patch. The template has a function `size()` that returns the number of the elements it currently holds. For a more detailed description of `deque` operations see Timothy (1998); Stroustrup (1997). The class **MortarPatch** can be defined as:

```
class   MortarPatch: {

  private:
    int PatchId;    // patch identification number
    double  XP0;    // x coord. of the patch origin
    double  YP0;    // y coord. of the patch origin

    deque <PatchSegment>  **SlaveSegs;  // slave segments
    deque <PatchSegment>  **MasterSegs; // master segments list
    deque <MortarSegment> **MortarSegs; // mortar segments list

  public:
    MortarPath (int Id, double Xp0, double Yp0,
                deque <PatchSegment> *Mseg,
                deque <PatchSegment> *Sseg,
                deque <PatchSegment> *MrtSeg);

    MortarSegment (FILE *file) ;
    MortarSegment () ;
    MortarSegment& operator = (const MortarSegment& mp);
    MortarSegment& (const MortarSegment&  mp);
    ˜MortarSegment();

    void BuildMortars (Mesh  *mesh);
    void BuildSlaves  (Mesh  *mesh);
    void BuildMasters (Mesh  *mesh);

    bool operator <  (const MortarPatch &s) const
                         { return PatchId < s.PatchId; }

    bool operator == (const MortarPatch &s) const
                         { return PatchId == s.PatchId;}

    bool operator != (const MortarPatch &s) const
                         { return PatchId !operator==(s); }

    bool operator >  (const MortarPatch &s) const
                         { return PatchId > s.PatchId; }
    ...

    ...
}
```

The class **StripSegment** specifies the integration strips introduced by the mortar element method. To implement the integration strip concept, the class contains several attributes as (see 3.8): the identification number of the strip `StripId`, the mortar offset `OffsetS0`, the projection operator `Q`, the integration strip length `LengthStrip`, the polynomial orders on the two sides of the mortars `QN` and `QM` and a link to the mortar attached to the **StripSegment** `PtrMortar`.

The **StripSegment** has this definition:

```
class   StripSegment: {

  private:
    int             StripId;     // strip id number
    double          OffsetS0;    // mortar offset
    double          **Q;         // the Q projection operator
    double          LengthStrip; // integration  strip length
    int             QN;          // N on the slave  edge
    int             QM;          // N  on the master edge
    MortarSegment   *MortarPtr;  // pointer to a mortar

  public:
    StripSegment() : StripId(-1), OffsetS0(0), LengthStrip(0),
                     Q(NULL), QN(0), QM(0), MortarPtr(NULL) { }

    StripSegment (FILE * file) ;
    StripSegment (int Id, double S0, double len,
                                       MortarSegment *Ms);
    StripSegment () ;
    StripSegment& operator = (const StripSegment&  strip);
    StripSegment& (const StripSegment& strip);
    ~StripSegment();

    void BuildMatrixQ();   // build the projection matrix Q

    bool operator < (const StripSegment& s) const}
                     { return StripId < s.StripId; }

    bool operator == (const StripSegment& s) const
                      { return StripId == s.StripId; }

    bool operator != (const StripSegment& s) const
                      { return StripId !operator==(s); }

    bool operator >  (const StripSegment& s) const
                      { return StripId > s.StripId; }
    ...

    ...
}
```

The classes defined above, are the main structures we need to implement the mortar element method. For the Adaptive Mesh Refinement Mechanism, we define a dynamical data structure that is updated each time an element is `added/refined/pruned`. In the next subsection, the mesh data structure is introduced, which is the basic component of the adaptive mesh refinement process.

4.4 Mesh data structure

The representation of a spectral element mesh is a fundamental design decision. A traditional structure consisting of arrays of elements and nodes, and their connectivity is not sufficient for an adaptive computation. A fully entity hierarchy is needed to support efficient mesh enrichment, both through *refinement* and *coarsening* (h-refinement), and through order enrichment (p/N-refinement). Another aspect of the implementation is the development of an infrastructures that supports the development of a complex dynamic data structure in a distributed memory environment. More information on the subject is presented in Bose and Carey (1999); Williams (1992); Edwards and Browne (1998); Parashar and Browne (1995); Baden *et al.* (1998). The solution we use here is based on Voxel Data Base introduced by Williams (1992), who presented a detailed description of the design used here and compared it with others.

In the numerical solution of scientific and engineering problems, there is a trade-off between the quality of a solution and the computational cost of obtaining that solution. This trade-off can be attacked in two ways:

- development of sophisticated adaptive methods that increase accuracy for reduced computational cost
- use of a larger faster computational environment, parallel/distributed computing.

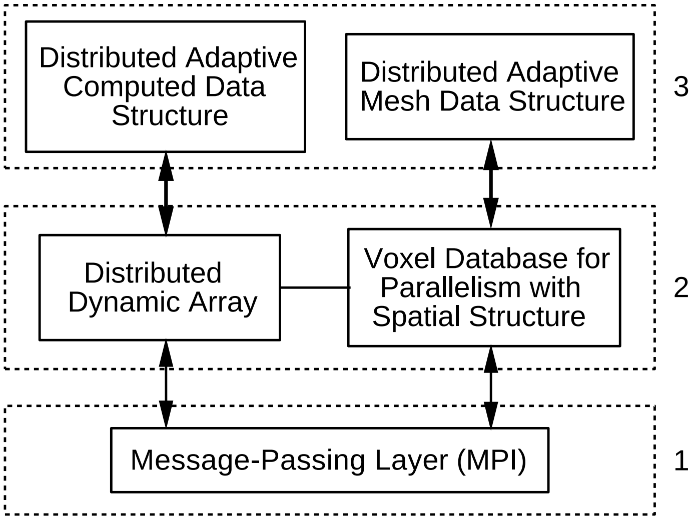


Fig. 4.5: Extended three-layer model for distributed dynamic data structures.

Each of these approaches leads to increased mathematical, algorithmic, and/or data management complexity. Because we try to combine both ways, adaptive methods and parallel/distributed computing, the implementation of a such approach has to take care of the computed data structure, and the adapted mesh data structure. A conventional parallel application's distributed dynamic data structures are layered directly upon a low level interprocess communication capability, such as a message-passing library MPI (MPI, 1994). This two-layer model does not adequately separate concerns and leads to complex data structures and parallel access mechanisms. To improve this model we introduce a generalized distributed data management layer: Voxel Database for Parallelism with Spatial Structure *VDB* and Distributed Dynamic Array (*DDA*), directly positioned above the message passing layer see figure 4.5. The *DDA* is a simple array structure, which couples the computed data, associated with the elements, to the *VDB*.

The *VDB* has been introduced by Williams (Williams, 1992) to provide a way to program a large variety of mesh computations in Fortran, C or C++, such that the execution of the program is independent of the distribution of data to processors, and the mesh may be topologically adapted and load balanced.

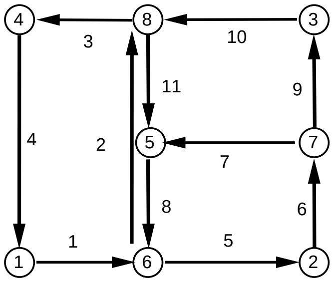
A voxel data base *VDB* is a distributed shared memory, where entities which share memory are those at the same geometric position. A *VDB* may be thought of as a dictionary of position-subscript pairs, so that data may be associated with points in space by using the subscription of that point as an index into data arrays. A voxel can be represented like a three-dimensional volume element in space, and can be addressed using by integer coordinates. Each voxel gets a unique integer key assigned to, and a count of hits that indicates how many times the position has been referenced.

The Spectral Element code may be split into two parts, one which deals with the properties and nature of the data stored with the elements and nodes and the calculation done with those data, and another part which maintains the shared memories at the nodes and supplies the mapping from local to global. The latter part may be done with a Voxel Database, producing a code to run on a distributed machine. The first part is done using a Distributed Dynamic Array, a structure that ia used to send/receive the data between the processors, based on the information stored in the *VDB*.

The *VDB* introduces the notion of *Set of Points*. The data structure for a *VDB* is a collection of subsets of points. For example, a quadrilateral mesh is a set of all the nodes together with a set of pointsets (elements) of size four. A graph may be represented as a set of nodes plus a collection of pointsets (edges), each of which is of size two. We can represent the pointsets as a table of subscripts, such as that for the graph shown in Fig 4.6. In this case each node of the graph is associated with a geometric point.

We can represent the pointsets as a table of subscripts, such as that for the quadrilateral mesh shown in figure 4.6. In this case, each node of the graph is associated with a geometric point. Since the structure is a graph, each pointset is of size two: the local numbering is at the left of the table. We can represent the eleven edges 1,...,11 as a eleven pointsets. The global numbering for the points is sequential, and it is easy to set up a loop over the points rather than always looping over pointsets and using the table of subscripts.

Using subscripts to implement a *VDB* can cause some cache misses on the processors caches and a loss of performance. Renumbering the points, we can improve this situation substantially, so that when the pointsets are used in order, most of the points required will already be in cache from the previous pointset. An application written with *VDBs* leaves control of communication with the application by explicit synchronization calls, keeping local



Pointset Number/Global Number

Local Number	1	2	3	4	5	6	7	8	9	10	11
1	1	6	8	4	6	2	7	5	7	3	8
2	6	8	4	1	2	7	5	6	3	8	5

Fig. 4.6: A graph represented as a collection of points and a collection of pointsets of size 2 (edges).

copies of data in all the processors that need them. The copies may have all the same status (the *combining* method), or may be copies of a master datum which are updated at synchronization. The assumption is that the data objects (pointsets) need only be communicated if they share a geometric point in space.

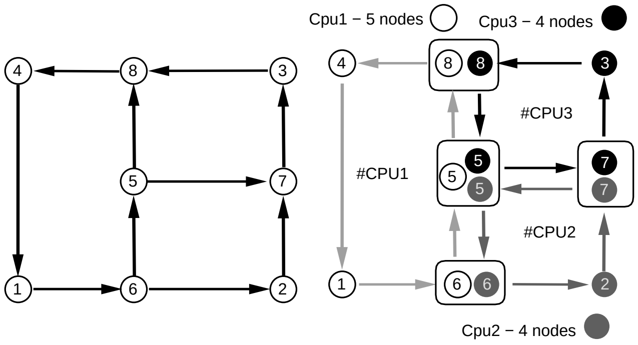


Fig. 4.7: A graph represented as a collection of points and a collection of pointsets of size 2 (edges).

For the distributed shared memory, *VDB* introduces the *alias* points. The concept of a point is replaced by an equivalence class of points which share memory, where two points are equivalent if they have the same position. Thus a point is actually a set of *alias* points, each of which has the same position. In fact an *alias* is a connection between a local point and a remote one. To access the points owned by the same processor, we have to read a copy of the points data stored locally in the memory of the processor. The same point may have different *aliases* that are distributed on different processors. To illustrate this see figure 4.7.

The left of figure 4.7 is shows a graph of eight nodes, which describes a non-conforming mesh. At the right we find the same graph distributed among three processors, with points assigned to different processors. Some points have been replaced by equivalence classes of *alias* points, and each processor has obtained, from the synchronization function, the number of nodes as shown. The sum of all these is 13, which is of course greater than the 8 equivalence class of *aliases*.

To represent a mesh, we have to create the graph seen as a collection of pointsets (edges),

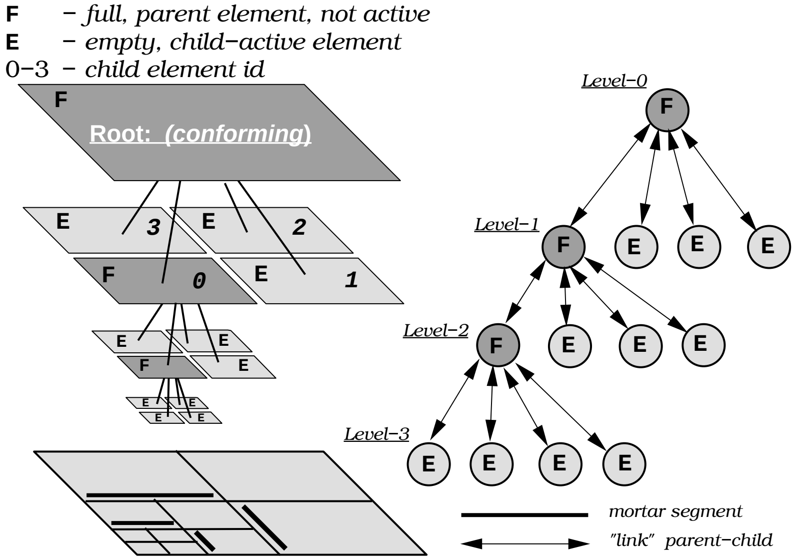


Fig. 4.8: A four-level quad-tree mesh. The parent refined elements, the ”F” elements, are not active. The active children elements are the ”E” elements and they make up the current discretization. The non-conforming edges are represented by the mortar segments. The mesh can be represented by a graph. There is a unique ”link” between the parents and the children such that the children can be identify by the parent’s id .

each owned uniquely by a processor. To create the connectivity of a mesh like the one illustrated in figure 4.8 we need to build four separate *VDBs*: one for the vertices, one for the edges, one for the midpoints of the edges, and one for the elements. The number of times a position is registered is called its *multiplicity*. If a vertex with multiplicity one, that does not lie along an external boundary, is not part of the true mesh true degree of freedom. In this case, the vertex is said to be *virtual* (Mavriplis, 1989). To detect if an edge is non-conforming, we have to query the *VDB* to find its multiplicity. If the edge’s multiplicity is one, and it does not lie along an external boundary, we query once again the vertices *VDB*, using its endpoints. If it is not a match, then the edge is non-conforming. The active elements (E) make up the current discretization of the mesh. Each leaf node (F) has a unique integer key that is a function of the parent’s key, which allows to identify the parent of the current active element.

The *VDB* implements weakly coherence for the shared memory. That implies that shared data may only be read outside a weak block code (the data values are not reliably read inside this block) in which the data is written, and may only be written inside such a block. Writing to a memory location implies replacing the data from that location by something else.

There is another way to ”write” to a shared memory location. The new data we want to write in the shared location, can be combined with the location. In this case, different processors can add data to a shared memory location in an independent order. At the end, we only need to synchronize the operations and get the right result. Thus pointsets may read from their points, but may only combine with, rather than write to a point. The *combining* block is ended with a synchronization, where *aliases* are combined with each other by *VDB* software, after which that dat is available for reading. Also, the *VDB* allows another traditional method of writing to shared memory, based on the idea of write permission that is called *updating*. Only one processor has write permission to the memory at any given time. Of all the *aliases* at a given point, exactly one may write; this special *alias* of all those at a given point is called the *secretary alias*. All the *aliases* may write, but after synchronization, the value in the memory is the value written by the *secretary* point. For a global *combining* operation we use only the *secretary* points, rather than all points.

For example, elements on separate processors with a common boundary segment share data along an edge. Because each processor may update its edge data independently, we need to call a synchronization routine that combines local and remote values to produce a globally consistent data set.

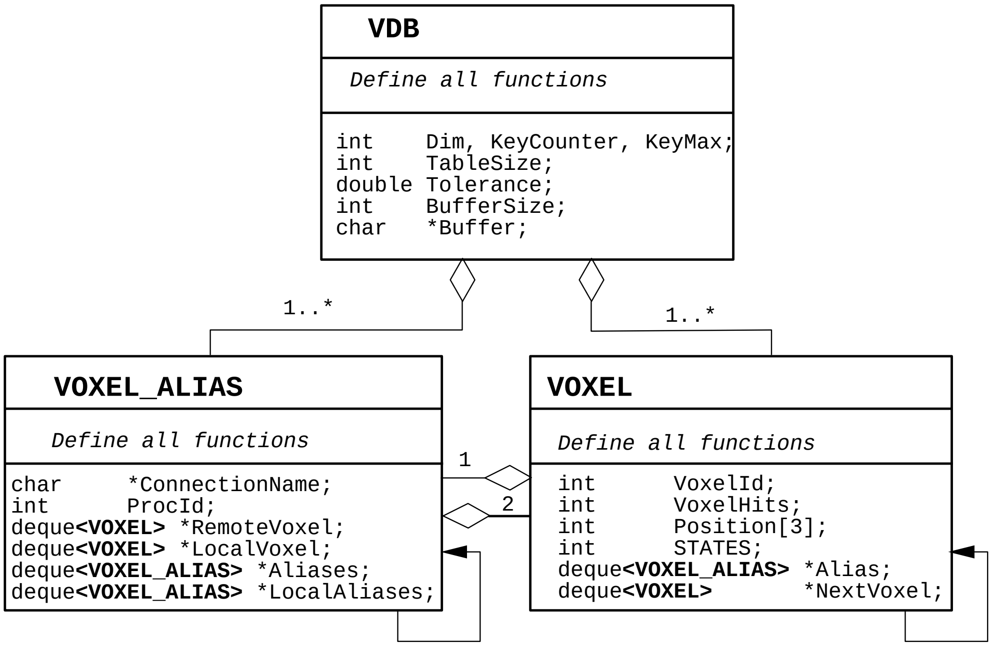


Fig. 4.9: The UML diagram of the **VDB** classes.

In figure 4.9 the **VDB** class hierarchy is shown. The main attributes of **VDB** class are: an array with the voxels positions, where each voxel is an instantiation of a **VOXEL** class and an array of remote voxel connections, where each connection is defined by a **VOXEL_ALIAS** class. The resolution of the voxels and the size of the hash table with the entries into the voxels' array are also defined. The communication handled by the **VDB**, *combining* local/remote data, is done via the buffer `Buffer`. Each voxel is addressed using integer coordinates stored in the `Position` attribute of class **VOXEL**. A reference to a remote voxel, that is stored on another processor is defined by **VOXEL_ALIAS** class.

4.5 A base environment for Object-Oriented scientific computing

Scientific codes are often large and complex, requiring vast amount of domain knowledge for their construction. They also process large data sets, so there is an additional requirement for efficiency and high performance. The last decades have seen significant advances in the area of software engineering. New techniques have been created for managing software complexity and building abstractions. Object-Oriented, generic (Stepanov, 1996), generative (Czarnecky *et al.*, 1998), meta-programming (Veldhuizen, 1995), are the new trends in scientific programming that points the way for constructing better software that is portable, maintainable and achieves high performances at a lower development cost. Every sub-domain in scientific computing has its own requirements: interval arithmetic, tensors, polynomials, automatic differentiation, sparse arrays, meshes, and so on. To provide these abstractions we need an Object-Oriented language, which allows library developers to construct them. In C++ and Fortran 77/90, we are seeing libraries for many applications which were previously solved by domain-specific languages. Unfortunately, such libraries are hard to optimize. Compilers have difficulty because they lack semantic knowledge of the abstractions: instead of seeing array operations, they see loops and pointers. Libraries also tend to have layers of abstractions and side effects which confound optimizations. It is doubtful that the optimization problems admit a general-purpose solution, since problem domain has its own tricks and peculiarities. What we really need are language features which allow library developers to define their own abstraction, and also to specify how these abstractions are optimized. In the literature, this solution is called *Active Libraries*. Active libraries combine the benefits of build-in language abstractions which those of library-level abstractions, adaptability, quick feature turnaround, easy to implement. In our implementation, we have considered two Active Libraries: $\mathbf{Blitz++}$ and $\mathbf{MTL}$.
The $\mathbf{Blitz++}$ library (Veldhuizen, 1998) provides generic array objects for C++ similar to those in Fortran 90, but with many additional features. In the past, C++ array libraries have been up to 10 times slower than Fortran, due to the temporary arrays which result from overloaded operators. $\mathbf{Blitz++}$ solves this problem using *expression templates* techniques (Veldhuizen, 1995) to generate custom evaluation kernels for array expressions. For example this code might represent the summation of three vectors:

```
w = x  +  y  +  z ;
```

Operator overloading in C++ is pairwise: to evaluate this expression, C++ forces first the evaluation of `tmp1=x+y` , then `tmp2=tmp1+z`, and final assign `w=tmp2`. The objects `tmp1` and `tmp2` are *temporaries* used to store intermediate results. Pairwise expression evaluation for the vectors is slow: instead of a single loop to evaluate an expression, multiple loops and temporary vectors are generated. These temporary vectors are usually allocated dynamically, which causes a severe performance loss for small vectors. Consequently, C++ vector/matrix libraries tended to be very slow for operations on small objects. To optimisme these operations for small objects, $\mathbf{Blitz++}$ uses the *template meta-program* techniques. It turns out that C++ compilers can be persuaded to do arbitrary computations at compile time by *meta-programs* which exploit template instantiation mechanisms. One good use of template meta-programs is creating *specialized algorithms*. For example, the following code calculates a 3x3 matrix-vector product:

```
Matrix < double > A(3,3);
```

```
Vector < double > b(3), c(3);
c = product(A,b);
```

Since, the `Matrix` and `Vector` objects must allocate their memory dynamically, which will take a lot of time, the nested loops, in *product()* function, are optimized by the compiler that assumes that the loop will be executed many times. The result is code which is fast for large matrices, but mediocre for small matrices. $\mathbf{Blitz}++$ solves this problem, by providing two versions of objects like `Matrix` and `Vector`. One version is optimized for large objects and the other is optimized for small objects (e.g. *TinyVector*). Here is the matrix-vector product implementation with *Tiny** objects:

```
TinyMatrix < double, 3, 3 >  A;
TinyVector < double, 3 > b, c;
c = product(A,b);
```

where the $\mathbf{TinyVector}$ can be declared as:

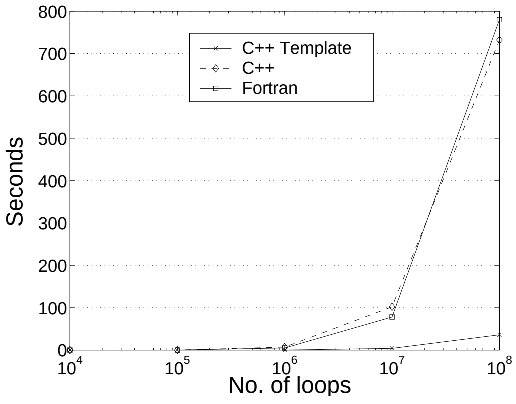
```
template <class T,  int N >
  class  TinyVector  {
    private:
        double data[N];
}
```

Putting the vector data inside the object allows it to reside on the stack, so the overhead of allocation memory is avoided.

As an example of a template meta-program, we consider a program that calculates factorials at compile time. When the **Factorial**$<$**N**$>$ template is instantiated, the value **enum** is set to **N*Factorial**$<$**N-1**$>$**::value**. This triggers the instantiation of **Factorial**$<$**N-1**$>$, which triggers the instantiation of **Factorial**$<$**N-2**$>$, and so on. The templates are instantiated recursively until the root case **Factorial**$<$**1**$>$ hit. The **Factorial** is defined as:

```
template<int  N >
    class Factorial {
      public:
        enum  { value = N * Factorial <N-1>::value } ;
    }

class Factorial<0> {
   public:
    enum    { value = 1 } ;
 } ;

const int x = Factorial<12>::value ;
```

In figure 4.10, the comparison between three implementations (C++ with expression templates, C++ and Fortran) of the function `Factorial` is illustrated. The function is evaluated iteratively with a loop . Increasing the number of iterations of the loop, will have a negative impact on the computation speed in the standard C++ and Fortran case. In the C++ with expression templates implementation the performances are very high even when the number of function calls is about 10^9.



(a)

Fig. 4.10: Speed increase for `Factorial<12>` template example. Timings were taken on a 250 MHZ, Onyx -SGI.

Another feature provided by $\mathbf{Blitz++}$ is array stencilling. Array stencils are common used for solving of partial differential equations. The performance of stencil operations is heavily constrained by memory accesses, so efficient cache use is critical. In $\mathbf{Blitz++}$, the stencilling operation is implemented by dividing the array into smaller sub-arrays, and applying the stencil to each sub-array in turn. Multidimensional arrays in $\mathbf{Blitz++}$ are provided by the class template $\mathbf{Array<T,N>}$. The template parameter $\mathbf{T}$ is the numeric type stored in the array, and $\mathbf{N}$ is its dimensionality. This class supports a variety of array models: arrays of scalar type, such as $\mathbf{Array<int,2>}$ and $\mathbf{Array<double,3>}$ or arrays of user-defended types. Let us consider the class $\mathbf{Element}$. The array $\mathbf{Array<Element,2>}$ is a two dimensional array of $\mathbf{Element}$ objects. Nested heterogeneous arrays, can be also defined, such as $\mathbf{Array<Array<double,1>,1>}$, in which each element is an array of variable length.

In figure 4.11 the results of an DAXPY (double precision $y = a * x + y$) operation are shown (Czarnecky *et al.*, 1998). This routine is used in many linear algebra operations. The class `TinyVector` is optimized for very small vectors, so its performances is much better than the other implementations for vectors of length 1 to 10. The `Vector` class and Fortran77 implementation have some loop overhead which makes their performance poorer for small vectors. For longer vectors their performance is very similar. The drop in performance around N=1000 occurs because the vectors are no longer small enough to fit in the cache. Due to the pairwise evaluation and dynamically allocated temporaries, the `valarray` is typical of older C++ class libraries (the performance of `valarray` is improved in the new C++ libraries, see 4.12).

We conclude that the $\mathbf{Blitz++}$ library provides a solid base environment of arrays, matrices and vectors for scientific computing in C++. The numeric arrays in $\mathbf{Blitz++}$ rival the efficiency of Fortran (Veldhuizen, 1995), but without any extension to the C++ language. The performances of the compilers on the chosen platform are crucial.

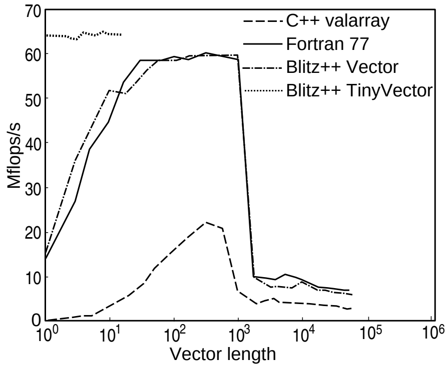


Fig. 4.11: The measure of the performance for this vector operation: `y = y + a*x`, where `x` and `y` are vectors, and `a` is a scalar. Timings were taken on a 100 MHz Octane (SGI) using a KAI++ (Intel) compiler with -K3 -O2 options and Fortran 77 code was compiled with MIPSpro 7.2 Fortran compiler with -O3 option.

The second library we consider in our implementation is the **Matrix Template Library** (**MTL**) (Siek and Lumsdaine, 1998). **MTL** is a C++ library which extends the ideas of **STL** to linear algebra. It handles both sparse and dense matrices. For dense matrices, **MTL** uses template meta-programs to generate tiled algorithms. Tiling is a crucial techniques for obtaining top performances from cache-based memory systems; **MTL** uses template meta-programs to tile on both the register and cache level. For register tiling, it uses templates meta-programs to completely unroll loops. **MTL** provides generic, high-performance algorithms which are competitive with vendor-supplied kernels. Also, it has an interface to the **Iterative Template Library** (**ITL**), a collection of sophisticated iterative methods written in C++. It contains methods for solving both symmetric and non-symmetric linear systems of equations. The **ITL** methods are constructed in a generic style, allowing for maximum flexibility and separation of concerns about matrix data structures, performance optimization, and algorithms. Presently, **ITL** contains routines for conjugate gradient (CG), conjugate gradient squared (CGS), biconjugate gradient (BiCG), biconjugate gradient stabilized (BiCGStab), generalized minimal residual (GMRES), quasi-minimal residual (QMR) without look-ahead, transpose-free QMR, and Chebyshev and Richardson iterations. In addition, **ITL** provides the following pre-conditioners: SSOR, incomplete Cholesky, incomplete LU and incomplete LU with thresholding.

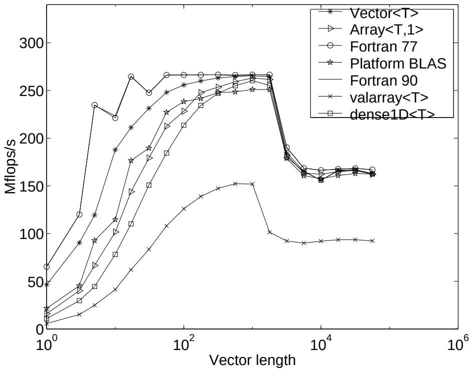
As mentioned, **MTL** provides generic algorithms. Generic programming has recently entered the spotlight with the introduction of the Standard Template Library (**STL**) into the C++ standard (Forum, 1995). The principal idea behind generic programming is that many algorithms can be abstracted away from the particular data structure on which they operate. Algorithms typically need the functionality of traversing through a data structure and accessing its elements. In this case the data structure provides a standard interface for this operations. The main difference in the separation of algorithms and containers is the *iterator* (a generalized pointer). Iterators provide a mechanism for traversing containers and accessing their elements. Generic algorithms are written solely in terms of iterators and never rely upon specifics of a particular container.

The next example shows the way **MTL** implements the generic matrix-vector product:

```
// Generic matrix-vector multiply
template < class Matrix, class IterX, class IterY >

void matvec_prod (Matrix A, IterX x, IterY y) {

    typename Matrix::const_iterator i;

    typename Matrix::OneD::const_iterator j ;

    for(i = A.begin(); i != A.end(); i++)

     for(j = i->begin(); j != i.end(); j++)

        y[j.row()] += *j * x[j.column()];
}

// BLAS-style dense matrix-vector multiply

    for( int i=0; i < m ; i++)

        for( int j=0; j < n ; j++)

            y[i] += a[i*n+j] * x[j] ;
```

The generic matrix-vector algorithm is flexible, and can be used with a wide variety of dense, sparse, and banded matrix types. The indexing in the **MTL** routines has been abstracted away. The traversal across a row goes from *begin()* to *end()*, instead of using explicit indices. The *row()* and *column()* methods provide a uniform way to access index information regardless of whether the matrix is dense, sparse or banded. The **MTL** provides a rich set of basic linear algebra operations, roughly equivalent to Level-1, Level-2 and Level-3 BLAS, though the **MTL** operates over a much wider set of data types. In **MTL** each algorithm is implemented with just one template function. Another aspect of the **MTL** implementation is that data encapsulation has been applied to the matrix and vector information, which makes the **MTL** interface simpler because input and output is in terms of matrix and vector objects, instead of integers, floating point numbers, and pointers. Figure 4.12 shows the comparison between a Fortran77, Fortran90, **MTL**, **BLAS**, C++ in the DAXPY benchmark test on an SGI-Onyx machine using the MIPsPRO C++ compiler. In these case, the compiler does perform enough optimization for **MTL** to be fast. Specifically, **MTL** makes heavy use of `iterators`. In order for `iterators` to be fast, an optimization called *lightweight object optimisation* must be performed by the compiler. The MIPsPRO C++ compiler does implement this. Since, **Blitz++** doesn't make as heavy use of iterators as **MTL**, it performs better than **MTL** for vectors with length $\leq 1 \times 10^3$.

It is obvious, from the test performed with **Blitz++** and **MTL**, that the performances we can obtain with this two libraries, depend heavily of the optimizations performed by the compilers we use. **Blitz++** (Veldhuizen, 1999) did not achieve great performance right from the start: it requires careful tuning for each compiler and platform. Another difficulty in tuning **Blitz++** is the inability of many compilers to optimisme small temporary objects, only the **KAI C++** compiler was an exception. **MTL** provides a good basis for creating higher level numerical linear algebra libraries. With performance benefits from **MTL** (Siek and Lumsdaine, 1999) , it would be good to gradually replace **LAPACK** functionality with versions that use **MTL**. In our implementation, we combine the features provided by both libraries, the support for high performance generic programming and meta-programming of **Blitx++** with the high-performance iterators of **MTL**.



(b)

Fig. 4.12: **Blitz**(`Vector<T> and Array<T,1>`), **MTL**(`dense1D<T>`), **Fortran77**, **Fortran90**, **C++**(`valarray`), **Fortran BLAS** performances for `y = y + a*x`. Timings were taken on a 250 Mhz Onyx-SGI(32 kb L1 data cache, 4 Mb L2 unified cache, and 1024 Mb main memory) using the CC (MIPsPRO C++ 7.3) compiler with -Ofast -64 options.

4.5.1 *Wrapping techniques for **SEPRAN**, **LAPACK** and **BLAS***

Scientists and engineers already recognized the benefits of Object-Oriented Programming (OOP) in their code development. Due to computational efficiency reasons, object-oriented constructs are usually restricted to a higher-level administration code, while the most CPU-time intensive computations take place in low-level code involving standard algorithms, loops and simple array data structures, which are easily recognized for optimization by the compiler. Many well-tested and documented non-C++ subroutines libraries exist for such algorithms. These libraries are mostly written in Fortran, with some newer libraries written in C or C++ (**LAPACK+**, **SparseLib** , **SL++**, **MET**,**uBLAS**). Although the C++ syntax is in many respects quite different from Fortran, the two languages apply practically the same basic data types, memory addressing and calling conventions. That makes the use of existing Fortran codes, when developing modern object-oriented numerical applications in C++, very easy. In this subsection we briefly detail the mechanism of how to call Fortran from C++, and subsequently present an interface class for a C++ ”wrapper” for existing Fortran codes. Calling Fortran functions from C++ and vice-versa is straightforward under the UNIX operating system and the technicalities are usually well-documented in language reference manuals and textbooks (Barton and Nackman, 1994). In order to accommodate the essential object-oriented features of polymorphism, C++ allows the creation of several functions with the same name, but different argument lists (*signature*). When calling Fortran functions from C++, it is essential that names for the Fortran functions, which will be prototyped in C++ header files, are not mangled. To prevent the usual name mangling carried out by the C++ compiler, declarations of Fortran subroutines in a C++ header file must be identified as requiring either Fortran or C linkage. This is done by using an extern ”C” wrapper:

```
// Use C linkage

extern "C" double ddot_ (int& n, double *x,
                  int& incx, double *y, int& incy);

// C++ function definition, calling the Fortran subroutine

inline void  ddot(int& n, double* x,int& incx,
                              double* y,int& incy) {

            ddot_(n,x,incx,y,incy);
}
```

When passing arrays of more than one dimension to Fortran, the data storage differences between Fortran and C++ have to be taken into account. In Fortran, two dimensional arrays are stored in ”column-major” format, whereas in C++ data is stored in ”row-major” format. As a consequence, the user must pass transposed data arrays to the Fortran compiler. To take advantage of polymorphism, sometimes we need to create base classes with virtual functions which are later defined in derived classes. The functions in the derived classed are then passed to the Fortran subroutines/function in the same manner as passing global functions defined in C++ to a Fortran subroutine/function. Here is an example of such a call:

```
// Use C linkage
typedef void (*CFunc_Ptr) (const double& x, double& value);

extern "C" integral_(CFunc_Ptr func, const double& x1,
                         const double& x2, double& result);

void CppFunction ( const double& x1, double&  value)  {
                value = exp(x) + 2 ;
}

// C++ function definition, calling the Fortran subroutine

inline void integral (CFunc_Ptr func, const double& x1,
                      const double& x2, double& result) {
        integral_((CFunc_Ptr) func, x1, x2, result);
}

// Main
int main(int argc, char* argv[]) {
```

```
    double  x1 = -1.0, x2 = 1.0, result = 0.0  ;

    integral ( (CFunc_Ptr) CppFunction, x1, x2, result);

    cout << "The value of integral: " << result << endl;

} // end main

C
C The Fortran function definition
C
    subroutine  INTEGRAL(f, x1,x2,result)
    REAL*8  result, x1, x2, val1, val2
    EXTERNAL  f

    call f(x1, val1)
    call f(x2, val2)
    result = 0.5 * (val1 + val2) * ( x2 - x1)
    return
    end
```

Using a Fortran library from C++ requires more work than using a C library because of the difference in build-in types and array layout. In our implementation, three Fortran subroutine libraries have been used: **SEPRAN**, **LAPACK** and **BLAS**. To use the Fortran subroutines of the libraries, we have to use C++ function calls via C++ wrappers to Fortran. One technique, to make such calls, is to wrap each C++ call to a **SEPRAN, LAPACK, BLAS** subroutine inside a C++ *inlined* function. In Appendix B this approach is presented.

Here is a short example with **SEPRAN** calls:

```
...

...
// Common bloks
////////////////////////////////////////////////////////
//      integer nbuffr, kbuffr, intlen, ibfree
//      common /cbuffr/ nbuffr, kbuffr, intlen, ibfree
//      save /cbuffr/

int&  nbuffr = cbuffr_.nbuffr;
int&  kbuffr = cbuffr_.kbuffr;
int&  intlen = cbuffr_.intlen;
int&  ibfree = cbuffr_.ibfree;

// Start SEPRAN

start ( istart, irotat, ioutp, itime);

// Generate mesh
```

```
kmesh[0]  = KMESH_LEN ;
kemesh[0] = KMESH_LEN ;
keqmesh[0] = KMESH_LEN ;
int ichois = 0;

mesh (ichois, idum, rdum, kmesh);

ichois = 0;
mesh (ichois, idum, rdum, keqmesh);

// Define problems

kprob[0] = KPROB_LEN;
ichois = 0;

probdf (ichois, kprob, kmesh, idum);
ichois = 6;
commat (ichois, kmesh, kprob, intmt1);

// Incorporate essential boundary conditions

int  ichcrv ;
int  ivec =0;
iu1[0] = 10;

ichois = 0; ichcrv = 1;
creavc (ichois, ichcrv, ivec, iQ, kmesh, kprob,
iu1,rdum,idum,rdum) ;

ichois = 0;
prestm (ichois,kmesh, kprob, iuni) ;


// Fill coefficients

int iprob ;

iuser1[0] = USER_LEN; user1[0] = USER_LEN; iprob = 1;
filcof (iuser1, user1, kprob, kmesh, iprob);

iuser2[0] = USER_LEN; user2[0]  = USER_LEN; iprob = 2;
filcof (iuser2, user2, kprob, kmesh, iprob);

...

...
```

Another approach, used only for **LAPACK, BLAS** subroutine calls is to reorganize the libraries and to group the related functions into corresponding class templates. Among the many excellent Fortran libraries for numerical processing, **LAPACK**, stands out as a well-designed package in wide use and applicable to a variety of scientific and engineering problems. Therefore, we use **LAPACK** to illustrate classes as mean of expressing the grouping of related subroutines with subsets of identical arguments. We use the same approach as in Barton and Nackman (1994). Consider the paired factoring and solving **LAPACK** subroutines for general matrices. The form of the factored matrix depends on the kind of matrix. For example, SGETRF (compute an LU factorization of a general M-by-N matrix) uses single precision to factor a general matrix into the product of a permutation, a unit lower triangular matrix, and an upper triangular matrix. To solve the system of liner equations factorized with SGETRF, the SGETRS subroutine must be called. By bringing the arguments together and combining them with the functions in an appropriate way, we can create a system of classes for factoring matrices. Unpacked matrices are passed to the factoring subroutines as a triple or quadruple:

1. the matrix in a Fortran array $\mathbf{A}$, the leading dimension LDA of $\mathbf{A}$
2. the number of columns N, and the number of rows M (for general matrices)
3. upon return, $\mathbf{A}$ contains the factored result and IPIV contains the pivots (where used).
4. the right-hand sides matrix $\mathbf{B}$, the leading dimension LDB of $\mathbf{B}$
5. the solution of the linear system is returned in $\mathbf{B}$.

Similar patterns, different in detail, appear in the calling sequence for **LAPACK** subroutines. Appendix B presents (B.3) details of class implementation for **LAPACK** and **BLAS**.

Here is an example on how to use the **LapackRect**<T>, which implements the class for rectangular matrices:

```
LapackRect <double> A(20,20) ;
// Fill the array A
...

...
LapackRect <double> b(20,1);
// Fill the array b
...

...
// Factor matrix A
FactoredLapackRect < double> factored_A= A.factor();

// Solve the system A x = b, b contains the solution
factored_A.solve(b);
```

The widely used **BLAS** subroutine libraries form the basis of many modern numerical libraries. **BLAS** comes in three levels: **BLAS-1** (Lawson *et al.*, 1979) for vector-vector operations, **BLAS-2** (Dongarra *et al.*, 1988) for matrix-vector operations, and **BLAS-3** (Dongarra *et al.*, 1990), (Kågström *et al.*, 1998a), (Kågström *et al.*, 1998b) for matrix-matrix operations.

The first example, implement a vector-scalar operation (**BLAS-1**) `x=a*x`, the multiplication of a vector by a scalar, using `xscal` subroutine:

```
template <class T>
ConcreteBlas <class T>&
  ConcreteBlas <T>::operator*=(const T& rhs) {

   // Obs. xscal is defined in BLAS level 1, here is inherited
   Blas3Calls::xscal (numElts(), rhs, firstDatum(), 1);

   return *this;

};
```

The second example is from **BLAS-2**, a matrix vector product `y=`α `Ax +` β `y` using the `xgemv` subroutine:

```
template <class T>
ConcreteBlas1d <class T>
 operator* (const ConcreteBlas <T> m,
                       const ConcreteBlas1d <T>& v) {

    ConcreteBlas1d <T> result (m.shape(0));

    // Obs. xgemv is define in BLAS level2, here is inherited
    Blas3Calls::xgemv (Blas2Calls::no_trans,
          m.shape(0), m.shape(1), T(1), m.firstDatum(),
          m.shape(0), v.firstDatum(), 1,T(0),
          result.firstDatum(), 1);
    return result;
};
```

For more details about the **LAPACK** and **BLAS** wrapping classes see Barton and Nackman (1994).

4.6 The architecture of the adaptive mesh refinement implementation

The classes presented so far, form the skeleton frame work of the implementation's class hierarchy. Based on the high-level data types and the computational modules, we are now able to define the overall architecture of the adaptive mesh refinement implementation. In figure 4.13 the class hierarchy diagram is illustrated. Each adaptive spectral element computation has an associated **Domain** data structure represented by the **Domain** class, which has the following components:

1. the adapted mesh `DomainMesh`, defined by the **Mesh** class
2. the error associated with the mesh and solutions `DomainErr`, defined by **Error** class
3. the solution fields `Sols[]`, defined by the **Field** class

4. the boundary conditions, `DomainBC`, defined by the **BCond** class
5. the matrix system `DomainMatrix`, defined by **Matrix** class.

The most important component of the implementation is the **Mesh** data structure, which is defined on the top of the **VDB** and **Element** classes and is stored as a collection of quad-trees. The leaves nodes of the quad-tree (see figure 4.8), the child-active elements, are kept in a list of current active elements `Elems`. The list is up-dated any time the mesh is adapted by refinement or coarsening. The connectivity of the mesh is maintained by using a **VDB** structure for the position of the vertices or edges and the non-conforming information necessary to up-date the mesh dynamically provided by the **MortarPatch** data structure. Every position in the data base is assigned a unique index that can be used to translate the position to an array index.

The common operations on a mesh are:

1. `traversal` of mesh entities implemented with `iterators`
2. `install/activate` an element
3. `refine` an element
4. `coarsen` an element of the entire mesh
5. `connect` the mesh.
6. `print/plot` the mesh.
7. `read/write` the mesh.

The public interface to the mesh includes a collection of different `iterators` types, introduced by the underlying representation. The level of details should not be exposed in the public interface, and has been hidden with the implementation of a more generic `iterator` class. `Iterators` over mesh entities have a simple, common interface: the `iterator` member function `int operator () (T *&)` return an integer indicating whether another entity is available to be return, and if so, return the next item in its argument. The internal data is organized such to provide an efficient access to inverse geometric classification information. Inverse classification information is useful, for example, when applying a boundary condition on a model edge. To dynamically adapt the mesh, the refinement criteria, which are implemented by the **Error** class are attached to each **Domain** data structure. An local error structure that contains the `NormInf, NormH1, NormL2` norms (infinity, H_1, L_2) is attached to each element of the `DomainMesh`. Based on the local errors, the mesh is dynamically `refine/coarsened`. To define a new refinement criteria, the public interface of the class **Error** provides the callback function `int (*user_criteria)()`, which is called during the computation to determine which element will be refined. The global matrix structure `DomainMatrix` is implemented by the **Matrix** class. Anytime the mesh is up-dated, the elemental matrices and various size parameters will be computed.

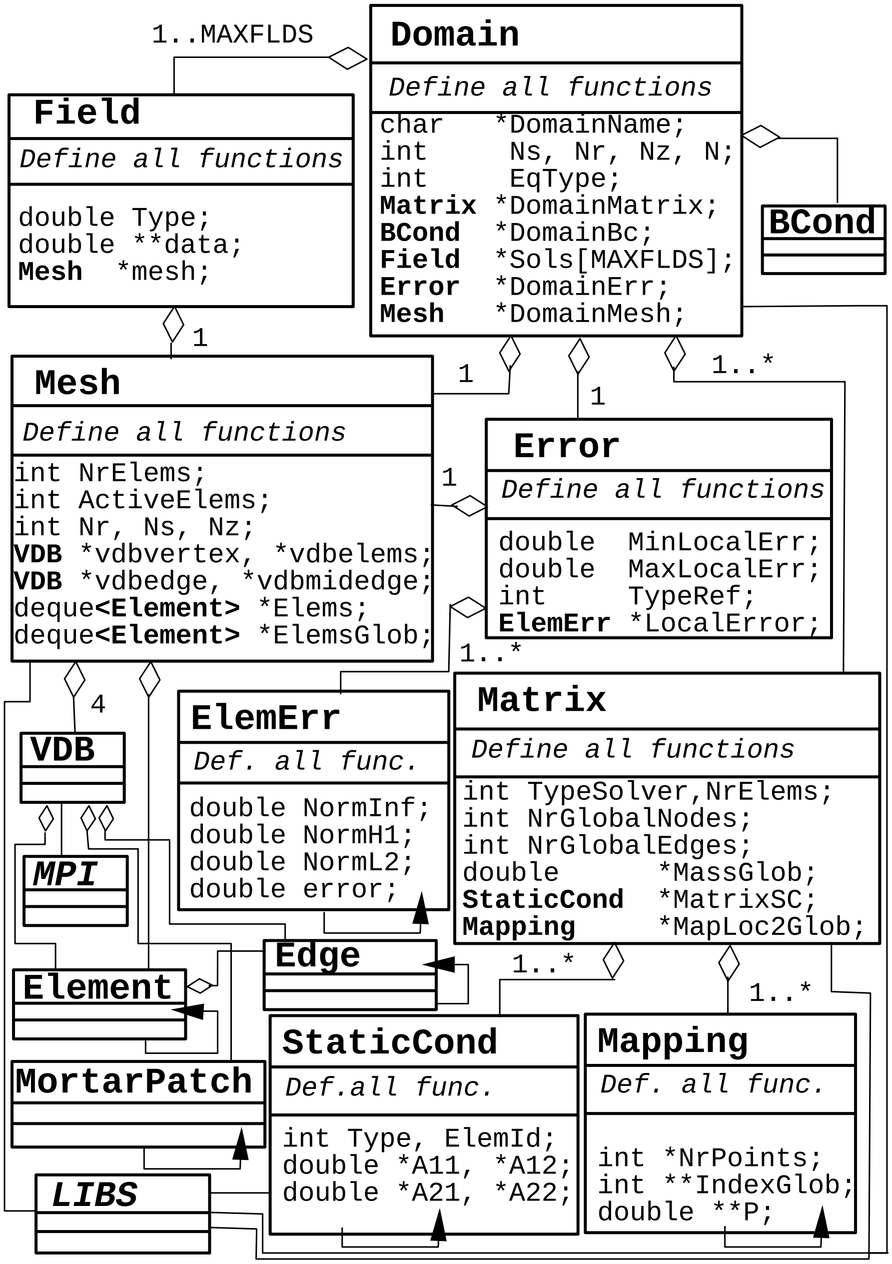


Fig. 4.13: The class hierarchy diagram of the adaptive mesh refinement technique based on the mortar elements method. The **MPI**_(Message-Passing Interface)_ and **LIBS** (**LAPACK**, **BLAS**, **SEPRAN** ,**Blitz** + +, **SPARSKIT**, **MTL**) are the class/wrapper libraries used in the implementation.

Applying the static condensation algorithm on an element level, implemented by **StaticCond** class, and assembly the global boundary system, by summing the elemental matrices, the boundary system is prepared for the solution phase by computing its **LU** factorisation. The system is solved by setting up the modified right-hand side of the global boundary equations, then solving the boundary and computing the solution on the interior of each element using direct matrix multiplication. The static condensation matrices are kept in `A11, A12, A21, A22` attributes of **StaticCond** class on element level. The solution of the system is stored in the `Sols[]` arrays of the **Domain** structure.

The computational part of the implementation is based on **BLAS, LAPACK, SPARSKIT, Blitz++** and **MTL** libraries. Using the wrapper classes defined in 4.5.1 the matrix and vector manipulations are implemented by **Blitz++** or **MTL** coupled with the computational **BLAS** and **LAPACK** classes.

The boundary conditions attached to the **Domain** are implemented by the **BCond** class. The essential (Dirichlet) and natural (Neumann) boundary condition can be attached to the element edges of the `DomainMesh` structure using the public interface of the **BCond** class. The periodic boundary conditions need a special treatment. The refinement of a periodic element trigger the refinement of the coupled element. In this way, the non-conforming edges are not allowed across periodic boundaries.

4.7 Example of a driver for adaptive mesh refinement

Based on the classes defined in the last section, we are now able to implement the driver for the adaptive mesh refinement process. In orde to implement the driver, the main components have to be defined:

1. The ”main time loop” component, that updates the mesh, computes the local errors, refine the mesh, interpolates the solution on the new mesh and assembles the global matrices for the solver.
2. The update-matrix component, which updates the local matrices each time step. This component is coupled to the spectral-element operators and facilitates the re-use of the old Fortran code.
3. The mortar solver, which uses the updated matrices and mesh to solve the linear system of equations for the current time step.

Instancing the appropriate classes for the above described components, the driver can have this form:

```
// Define the domain’s problem
Domain  *problem;

// Define the mesh associated with the problem
Mesh    *mesh;

// Define the error on the mesh
```

```
Error    *error;

// Define the matrix for the problem
Matrix   *matrix;

// Define the solution and the applied force
Field    *U;
Field    *F;

const double TIME_START = 0.0;
const double TIME_END   = 1.0;
const int    MAX_STEPS  = 1024;

int main(int argc, char* argv[]) {

    int i = 0;
    double time = 0;
    double dt = 0;
    char file = "problem.dat" ;

    dt = (TIME_END - TIME_INIT) / MAX_STEPS;

    problem = Domain (file) ;

    mesh = Mesh (problem) ;

    error = Error (mesh, SPECTRUM) ;

    U = Field (problem, SOLUTION) ;

    F = Field (problem, FORCE);     // initialize  applied force

    for (time =0; time < END_TIME ; time += dt) {

        matrix->Update  (mesh);

        problem->Solve  (matrix, mesh, U, ITERATIVE);

        error->Compute  (U);

        mesh->Refine  (error);

    } // end loop over time

} //end main
```

First, we define the domain of the problem we want to solve and the mesh associated with it. Since the adaption is used, an error class is instantiated for the defined mesh, which computes the local errors for the elements. To initialize the classes, the initial data is read in from an input file. The information related to the problem as: initial conforming mesh, boundary conditions, the coefficients of the partial differential equation, and the applied force, is contained in this input file. Each time step, the elemental matrices are generated by SEPRAN and assembled by the driver into the global matrix system, which is solved directly of iteratively. The local errors are computed and based on the refinement criteria the elements are flagged for refinement. The mesh is refined, if necessary, and the main time loop is ready for the next time step.

4.8 Conclusions

In this chapter, a tool for the solution of partial differential equations using the parallel adaptive spectral element method has been presented. The design and implementation of the adaptive mesh structure based on OO techniques is a complex task, but reusable software libraries are essential for the development of specific applications, to solve diverse problems without concern for details of the underlying mesh structure. After nearly a decade of being dismissed as too slow for scientific computing, C++ has caught up with Fortran and it is giving it hard competition (Veldhuizen, 1998; Siek and Lumsdaine, 1998). It provides powerful support for OO programming through language features such as virtual base classes, multiple inheritance, polymorphic functions, and operator overloading. However, since C++ uses dynamic memory allocation and deallocation, run-time binding and procedure calls to implement these features, it is difficult for the compiler to optimize the C++ code. To take advantage of the C++ features and to help the compiler with optimization of the C++ code, some issues have to be addressed when we design C++ numeric code. Inheritance presents an optimization challenge for the compiler. There are situations when C++ programs may have to dereference of system-defined pointers in order to access the numerical data. These indirections can only be done at run-time, and is difficult for the compiler to optimize such C++ code. Therefore, the use of the virtual classes must be done with care. Dynamic memory allocation is another C++ feature, which allows us, at run-time, to specify te exact amount of memory needed for the computations, but the associated overhead can be significant, especially when working with a large number of small data structure. To avoid this, we need to redefine the basic memory handling routines in C++. The attractive syntax provided by overloaded arithmetic operators for C++ vector classes, may drastically reduce the computational efficiency. To achieve maximum efficiency easily optimizable members functions should be used. Using expression templates to perform compile-time transformations will reduce the number of temporaries created by the overloading operator (Veldhuizen, 1998). C++ does provide many features necessary to support *generic programming* and *meta-programming*, but the implementation of such features is still cumbersome and restrictive.

Since our objective was to re-use Fortran code, the **SEPRAN** package, a hybrid approach of using an OO was proposed: C++ for handling data structures of the mortar element method, and a combination of legacy libraries (**BLAS**, **LAPACK**) and C++ for the linear solver with mesh-adaptivity. Implementing classes for use in scientific and engineering applications, involves numerical algorithms or other information processing using established algorithms. An object-oriented approach can use functions from the legacy libraries to improve the behavior of C++ objects. Wrapping these two libraries illustrates many of the goals of object-oriented programming. The resulting wrappers do not require more computations than the original code but they reorganize the computations and repackage it. Also, the wrappers will make the future use of the libraries easier and more robust. Class libraries can express more complex coupling inside of simpler boxes than can subroutine libraries.

With sufficiently powerful language features, such as templates in C++, it is possible to build libraries which both define abstractions and control how they are optimized (this is the idea of an *active library*). In the *active libraries* case, the responsibility for high-level optimization is shifted from the compiler to the library (Veldhuizen, 1999; Siek and Lumsdaine, 1999).

The conclude that: C++ written programs at a high abstraction level, and implemented carefully, so that the CPU intensive numerics take place in functions that are easily optimized by the C++ compilers can achieve a computation efficiency which comes close to Fortran. Replacing the CPU intensive numerics functions with Fortran code, approach used in our implementation, offers an opportunity to take advantage of the features offered by both programming languages.

Chapter 5

Application of Mortar Elements to Diffuse-Interface Methods

5.1 Introduction

In the previous chapters we described the construction of a mortar element method which will be used to tackle some length scale problems in diffuse-interface models. These diffuse interface models were applied successfully to situations in which the physical phenomena of interest had a length scale commensurable with the thickness of the inter-facial region. The challenge now is the area of mesoscopic fluid flows that involve large interface deformations and/or topological changes, such as droplet breakup and coalescence. Any diffuse-interface has a finite thickness, which is determined by the molecular force balance at the interface and its value is closely related to the finite range of molecular interactions (Rowlinson and Widom, 1989). The molecular force balance at the interface controls the topological transitions, therefore it allows to pass the topological transition in a physically justified way. The diffuse-interface approach has been used to study a wide range of phenomena involving topological changes: nucleation and growth, spinodal decomposition, droplet breakup. For a review on the subject see e.g. Anderson *et al.* (1998). Most of the studies on topological changes focus on small-scale systems, in which it is assumed that the numerical interface thickness is of the same order of magnitude as the real interface thickness. In general, for large systems, for which the droplet size is much larger than the physical value of the interface thickness, the real interfacial thickness can not be captured numerically. Scaling in such systems needs special attention, because if the real interfacial thickness is to be replaced by a numerically acceptable thickness, we have to make sure that we are still describing the same system with the same interfacial tension and diffusion. Different possibilities have been proposed in the literature, Anderson *et al.* (1998), Verschueren (1999), Lowengrub *et al.* (1998), but none of the proposed scaling strategies appears to be appropriate to cover all phenomena.

Our objective is to circumvent the scaling problem by using the adaptive mesh refinement method. Commonly, diffuse-interface models introduce a small length scale (the interface width), which places stringent conditions on the numerical solution methods. Based on the Cahn-Hilliard expression of the free energy (see Cahn-Hilliard model section), the critical size L_c is calculated to be:

$$L_c = \frac{\xi}{\mathcal{O}(C)} \quad , \tag{5.1}$$

with C the Cahn number and ξ the interface thickness. Small interfacial thicknesses would require the use of a smaller Cahn number in the simulations and, consequently, extremely small mesh sizes and, hence, require excessive computational time and computer memory. For Cahn numbers typically used in the simulations ($C = 0.02$), and a typical interface thickness, which is in the order of magnitude 10 nm, the computational domain used has a length of the order of 500 nm. If we want to extend to larger systems, the real interface thickness can not be captured numerically in general.

Using the adaptive mesh refinement (AMR) techniques based on the mortar element method, we try to relax this undesirable conditions. Since we are dealing with drastic topological changes, our scope is to track the movement of drop boundaries, by adding and removing elements around the boundaries of the drop for different, decreasing, values of the Cahn number C.

The diffuse-interface model is already implemented in SEPRAN, but it can be simulated only with conforming elements. Due to this limitation, the computational domain has to be decomposed between 1200 to 5000 elements, depending of the problem we want to solve. The method has been applied on a variety of problems ranging from a single drop to a large set of drops. In the next sections, we illustrate the benefits of the mortar method applied to diffuse-interface problem: with less elements than in the conforming case, we can achieve the same accuracy of the solution.

5.2 Cahn-Hilliard model

Diffuse interface models have a long history in fluid mechanics (see e.g. the review by Anderson *et al.* (1998), Verschueren (1999), Lowengrub *et al.* (1998) and Naumann and He (2001)), especially in the field of phase separation and structure development in solidifying metal (alloys) and polymer blends. The Cahn-Hilliard theory, also called gradient free energy theory, is the basic approach to express the specific Helmholtz free energy used in diffuse-interface modelling (Cahn and Hilliard, 1958):

$$f(c,\nabla c) = f_0(c) + \frac{1}{2}\varepsilon\left|\nabla c\right|^2 = -\frac{1}{2}\alpha c^2 + \frac{1}{4}\beta c^4 + \frac{1}{2}\varepsilon\left|\nabla c\right|^2 \ , \tag{5.2}$$

where α and β are positive constants and ε is the gradient energy parameter, that is proportional to the interaction parameter χ, and c is the mass fraction of one of the two components.

The chemical potential is defined as the change in f upon addition of an amount of component c. Mathematically, this is represented by a functional differentiation of f:

$$\mu = \frac{\delta f}{\delta c} = -\alpha c + \beta c^3 - \varepsilon \nabla^2 c \ . \tag{5.3}$$

This equation allows the computation of equilibrium concentration profiles. In order to comply with mass conservation for both components, the balance equation requires:

$$\frac{dc}{dt} = \frac{\partial c}{\partial t} + \nabla\cdot(c\mathbf{v}) = M\nabla^2\mu \ , \tag{5.4}$$

where M is the mobility coefficient, here taken constant.

Equations (5.3) and (5.4) are known as the Cahn-Hilliard equations. Cahn has used these relations to model spinodal decomposition. Due to the truncation of the gradient expansion of the second order, the Cahn-Hilliard theory was originally thought to be only valid for the initial stages of spinodal decomposition or for near-critical systems, where concentration gradients are small. However, equation (5.2) is generally assumed to be also valid when concentration gradients are large (Kikuchi and Cahn, 1962).

Momentum conservation, a generalized Navier-Stokes equation, can be derived yielding the velocity field (Lowengrub *et al.*, 1998):

$$\rho\left[\frac{\partial \mathbf{v}}{\partial t}+(\mathbf{v}\cdot\nabla)\,\mathbf{v}\right]=-\rho\nabla g+\nabla\cdot\eta\left(\nabla\mathbf{v}+\nabla\mathbf{v}^{T}\right)+\rho\mu\nabla c\,. \tag{5.5}$$

where g is the Gibbs free energy $g = f + p/\rho$, with p the local pressure and ρ the density. Generally, the viscosity η depends on c but, without any serious restrictions, the iso-viscous case will be considered here.

In equation (5.5) the interfacial tension Γ is reflected via the capillary term $\rho\mu\nabla c$. From this point forward, we consider only viscous fluids at moderate velocities, and hence the left-hand side of equation (5.5) can be neglected (the inertia-forces).

To obtain a more convenient form of the Stokes equations, for viscosity matched fluids, and in the absence of the inertia forces, we can use the stream function ($\mathbf{v} = (\frac{\partial\psi}{\partial y}, -\frac{\partial\psi}{\partial x})$). Conservation of mass is then automatically satisfied and the equations can be rewritten as:

$$\eta\nabla^4\psi=\rho\nabla\times\mu\nabla\mathrm{c}\,. \tag{5.6}$$

5.2.1 *Scaling of the Cahn-Hilliard equations*

To write the governing equations (5.3), (5.4) and (5.6) in non-dimensional form, the following dimensionless variables are introduced: $c^* = c/c_B$, $\mathbf{v}^* = \mathbf{v}/V$, $\mu^* = \mu\xi^2/(\varepsilon c_B)$, $t^* = tV/L$, with $c_B = \sqrt{\alpha/\beta}$ the bulk concentration, V is a characteristic velocity, and L is a characteristic domain size. Omitting the asterisk notation, the dimensionless equations are:

$$\frac{dc}{dt}=\frac{1}{Pe}\nabla^2\mu\,, \tag{5.7}$$

$$\mu=c^3-c-C^2\nabla^2 c\,, \tag{5.8}$$

$$\nabla^4\psi=\frac{1}{Ca}\frac{1}{C}\nabla\times\mu\nabla c\,, \tag{5.9}$$

with the Péclet number Pe, the capillary number Ca and the Cahn number C defined as:

$$Pe=\frac{\xi^2 LV}{M\varepsilon}\ ;\ \ Ca=\frac{\xi\eta V}{\rho\varepsilon c_B^2}\ ;\ \ C=\frac{\xi}{L}\,.$$

The capillary number can be related to the more classical definition $(\eta\mathrm{V}/\Gamma)$ (Davis and Scriven, 1982):

$$Ca=\frac{2\sqrt{2}}{3}\frac{\eta V}{\Gamma}\,.$$

This system of three partial differential equations, completed with proper initial and boundary conditions is capable of describing the dynamics of viscous two phase systems (like polymer blends) in the case of phases separating of separated systems in the presence of flow. The optimal way to scale the set of equations is still an important research challenge, which is beyond the scope of this thesis. Interested readers are referred to Verschueren (1999) and Keestra *et al.* (2003).

5.3 Numerical approach

To complete the set of Cahn-Hilliard equations, a relation between the chemical potential μ and the concentration c is required. We use the so-called 'c^4' approximation for the homogeneous part of the free energy (Gunton *et al.*, 1983):

$$f_0(c) = \frac{1}{4}\beta c^4 - \frac{1}{2}\alpha c^2 \ , \tag{5.10}$$

also called the Ginzburg-Landau approximation, which is a Taylor expansion around the critical point of the Flory Huggins equation:

$$f_0(c) \propto c \ln c + (1-c)\ln(1-c) + \chi c(1-c) \ , \tag{5.11}$$

with χ the Flory-Huggins interaction parameter.

To discretize the governing equations, the mortar-spectral element method introduced in chapter 3 is used, since this method is suitable for capturing interfaces with a small interfacial thickness. This method relaxes $\mathcal{H}^1$ continuity requirements of the conforming spectral-method. We consider each element individually and achieve a matching conditions through a variational process. As in the conforming case, the entire domain $\mathbf{\Omega}$ is subdivided into K non-overlapping sub-domains $\mathbf{\Omega}^k, \ \ k = 1, \dots, K$. The skeleton S of the domain decomposition comprises all interfaces between sub-domains and is also decomposed into mortars. We require the mortars to coincide with a complete edge of one of the sub-domains and the intersection of the mortars is empty space. Each mortar γ_m coincides with an element edge $\Gamma_l^k, l = \{1, 2, 3, 4\}$ of $\mathbf{\Omega}^k$.

Next, we define the mortars auxiliary space $\mathbf{W}_h(\mathbf{\Omega})$ (3.32) and the non-conforming spectral space $\mathbf{X}_h(\mathbf{\Omega})$ (3.44), that imposes the mortar conditions: the vertex conditions (3.37, 3.39) and the integral conditions (3.38, 3.40).

The momentum equation (5.9), is a fourth-order differential equation in ψ. Since the basis functions ϕ are elements of $\mathcal{H}^1$, that is $\mathcal{H}^1(\mathbf{\Omega}) = \{\phi \,|\, \phi \in \mathcal{L}^2(\Omega), \nabla\phi \in \mathcal{L}^2(\mathbf{\Omega}) \times \mathcal{L}^2(\mathbf{\Omega})\}$, we split equation (5.9) into two second-order differential equations:

$$-\nabla^2\omega = h \ , \tag{5.12}$$
$$-\nabla^2\psi = \omega \ , \tag{5.13}$$

where $h = -Ca^{-1}C^{-1}\nabla \times \mu\nabla c$. Using the inner product $(u, v)_{\mathbf{\Omega}} = \int_{\mathbf{\Omega}} uv \, d\Omega$, and u the standard Galerkin test function, the partial integration of the Galerkin residual representation of equations (5.12) - (5.13) yields the weak or variational forms:

$$(\nabla\omega, \nabla v)_{\mathbf{\Omega}} \ = \ (h, v)_{\mathbf{\Omega}} \ , \tag{5.14}$$
$$(\nabla\psi, \nabla v)_{\mathbf{\Omega}} \ = \ (\omega, v)_{\mathbf{\Omega}} \ , \tag{5.15}$$

where the boundary integrals vanish because of the homogeneous boundary conditions. Next, the domain $\boldsymbol{\Omega}$ is decomposed into K non-overlapping sub-domains $\boldsymbol{\Omega}^k$ and a spectral approximation is applied on each element, where $\mathbf{S}$ is the Laplacian stiffness matrix, $\mathbf{M}$ is the mass matrix and $\widetilde{\psi}$, $\widetilde{\omega}$ and $\widetilde{h}$ are the discrete vector representations of ψ, ω and h, respectively.

$$\mathbf{S}\widetilde{\omega} = \mathbf{M}\widetilde{h}\,, \tag{5.16}$$

$$\mathbf{S}\widetilde{\psi} = \mathbf{M}\widetilde{\omega}\,. \tag{5.17}$$

The local balance equation for c and the chemical potential μ form a set of two second-order differential equations, which are solved in a coupled way. Using Euler implicit time discretisation we obtain:

$$\begin{bmatrix} \mathbf{Q^T} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{M} + \Delta t \mathbf{N}^{\mathrm{n}} & \dfrac{\Delta t}{Pe}\mathbf{S} \\ [1 - (\mathbf{c}_{\mathrm{i}}^{\mathrm{n+1}})^2]\mathbf{M} - C^2\mathbf{S} & \mathbf{M} \end{bmatrix} \begin{bmatrix} \mathbf{Q} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{c}_{\mathrm{i+1}}^{\mathrm{n+1}} \\ \mu_{\mathrm{i+1}}^{\mathrm{n+1}} \end{bmatrix} = \begin{bmatrix} \mathbf{Q^T} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{M}\mathbf{c}_0^{\mathrm{n}} \\ \mathbf{0} \end{bmatrix}, \tag{5.18}$$

where $\mathbf{N}$ is the convection matrix and $\mathbf{Q}$ is the mortar projection matrix. Superscript n denotes time t and $\mathrm{n}+1$ denotes $t + \Delta t$. A Picard iteration is used to deal with the non-linearity in the c term ($i = 1 \ldots I$): the iteration starts using $\mathbf{c}_1^{\mathrm{n+1}} = \mathbf{c}_0^{\mathrm{n}}$ and as a stopping criterion we use $\max|\mathbf{c}_{\mathrm{i+1}}^{\mathrm{n+1}} - \mathbf{c}_{\mathrm{i}}^{\mathrm{n+1}}| < \delta$, in which δ is typically of the order 10^{-4}. After convergence, $\mu_{\mathrm{i+1}}^{\mathrm{n+1}}$ and $\mathbf{c}_{\mathrm{i+1}}^{\mathrm{n+1}}$ are used to update h and we can move to the next time step. Details can be found in Verschueren (1999).

In this model, there are two degrees of freedom (*dof*) per node c and μ. The mortar method introduced in chapter 3, treats only one *dof* per node. To keep the same structure of $\mathbf{Q}$ and $\mathbf{Q^T}$, as for one *dof*, the two *dof*s have to be sorted in a sequential order per node. Only in this case, the previous used equation (3.84) is still valid.

5.4 Results

In this section the mortar element method is applied to the diffuse interface modelling of the morphology and rheology of immiscible polymer blends. Blending of immiscible polymers offers an attractive route to produce new materials with tailor made properties. The mechanical properties of such two-phase or more phases polymer blends are intimately connected with the morphology imparted during processing. Hence, understanding the connection between flow fields applied and morphology development is vital to optimize the processing and, therefore, the resulting properties of blends. During the years, a number of comprehensive experimental and theoretical studies of morphology development in simple (shear)

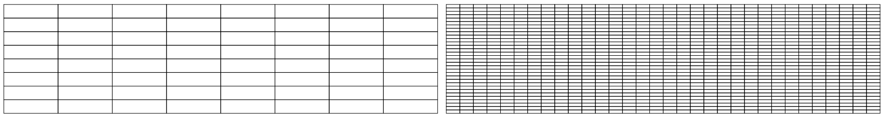


Fig. 5.1: The initial coarse grid: (*left*) adaption case, 8×8 elements ; (*right*) conform case, 32×32 elements.

flow fields has been reported. Some of this work is summarized in a recent review by Tucker and Moldenaers (2002), and although considerable fundamental understanding of morphology changes during (shear) flow has been obtained already, the prediction of the (transient) rheology coupled with the micro-structure development still remains a challenge.

We use a direct approach based on the framework of diffuse-interface models to predict the dynamics of the morphology. Interfaces are not modelled explicitly, but result implicitly from the composition field. Hence dramatic changes in topology of complex interfaces, occurring during coalescence and breakup, are present in the model without the need of making any additional assumptions about the underlying structure.

5.4.1 Single-drop problem

First, we consider a two-dimensional simulation with one drop and two refinement criteria, the gradient and the Legendre polynomial spectrum, are applied to the simulations. The simulations are performed on a two-dimensional rectangular mesh, with the dimensions (-2, -0.5) in the left bottom corner and (2, 0.5) in the top right corner, generated by successive refinement based on the concentration solution, with periodic boundary conditions on the left and right side of the domain. On the top and bottom side, boundary conditions are prescribed to introduce shear. Since equation (5.9) is split into two second-order differential equations, a set of two boundary conditions is applied and, bearing in mind the stream function $\mathbf{v} = (\frac{\partial \psi}{\partial y}, -\frac{\partial \psi}{\partial x})$, on the top and bottom wall $\psi = \frac{1}{2}ay^2$ and $\omega = a$ are prescribed.

We will look at one type of parameter variation for this problem: the evolution of the grid with decreasing the Cahn number C from 0.04 to 0.01. Note that temporal refinement is necessary as well; a suitable time step is chosen for each new Cahn number. The time steps in these calculations vary from $\Delta t = 0.001$ to $\Delta t = 0.0005$. The initial coarse grid has simply 8×8 elements (figure 5.1, left) with the polynomial order equal to $N = 4$. After each time step, local error estimators are calculated. These error estimators govern the refinement and coarsening process of the mesh. For the solution gradients criteria the imposed refinement tolerance is $\epsilon = 1.0 \times 10^{-1}$. In the Legendre polynomial spectrum case, the tolerance $\epsilon = 1.0 \times 10^{-6}$ is used. The Péclet number will vary depending of the Cahn number $P_e = 0.10/C$, and the capillary number will be fixed $Ca = 10$.

Figures 5.2, 5.5 and 5.8 show the adaptively generated grids and the evolution of the drop deformation for the concentration gradients. The adaptive procedure tracks the changes in topology of the interface and refines the grid to an appropriate level at each value of the Cahn number $C = \{0.04, 0.02, 0.01\}$. The same procedure is applied in the Legendre polynomial spectrum refinement criteria. In this case, the results are shown in figures 5.3, 5.6 and 5.9. Although there is obviously no exact solution for this problem, we compare the concentrations obtained in the non-conforming case, with the concentrations obtained in the conforming high-resolution numerical simulations of the same flow, to demonstrate that the adaptive procedure produces an accurate approximation. Figures 5.4, 5.7 and 5.10 compare the profile of the concentration c in the conforming and non-conforming case. The comparison shows that the two calculations are in extremely close agreement, and demonstrates that the adaptive procedure results in a highly accurate solution for small Cahn numbers C with an intelligent distribution of the elements. In table 5.1 the average number of elements used for each calculation is shown. For both refinement criteria the mesh has $K \times N^2 \approx 420 \times 25 = 10500$ points, far less that in the conforming case where the mesh has 25600 points (figure 5.1, right). Due to the refinement, less than 50% of the elements of the conforming case, are used to track the boundary interface. Since the periodic boundary conditions are imposed on the left, and right side of the domain, a few extra elements are refined on the left/right side of the domain, when the opposite side element is refined.

Table 5.1. The average number of active elements generated for different Cahn numbers $C = 0.04, 0.2, 0.01$ and polynomial order $N = 4$.

Tolerance	$C = 0.04$	$C = 0.02$	$C = 0.01$	Refinement Criteria
$\epsilon = 1.75 \times 10^{-1}$	370	425	516	Gradient
$\epsilon = 1.00 \times 10^{-6}$	360	401	475	Spectrum

To see the influence of a higher approximation order than $N = 4$, we consider the simulation for $C = 0.02, Ca = 10.0, P_e = 5.0$ and $N = \{8, 16\}$. Table 5.2 shows the number of active elements generated by the refinement based on the Legendre polynomial spectrum criteria with a tolerance $\epsilon = 1.0 \times 10^{-6}$. Also, in figure 5.11, the effect of the refinement for $N = 8$ is illustrated. The adaption generates a mesh with about $K = 280$ elements, which uses 22680 points. Using the same polynomial order $N = 8$ for the conforming case, it will generate $1024 \times 81 = 82944$ points. However, comparing the results illustrated in figure 5.6 and 5.11, we can conclude that, increasing the polynomial order of the approximation does not improve significantly the solution accuracy to justify the use of a large number of grid points.

Table 5.2. The average number of active elements generated for different polynomial order $N = 4, 8, 16$ and $C = 0.02, Ca = 10.0, P_e = 5.0$ for the Legendre polynomial spectrum refinement criteria.

Tolerance	$N = 4$	$N = 8$	$N = 16$	Refinement Criteria
$\epsilon = 1.00 \times 10^{-6}$	360	280	150	Spectrum

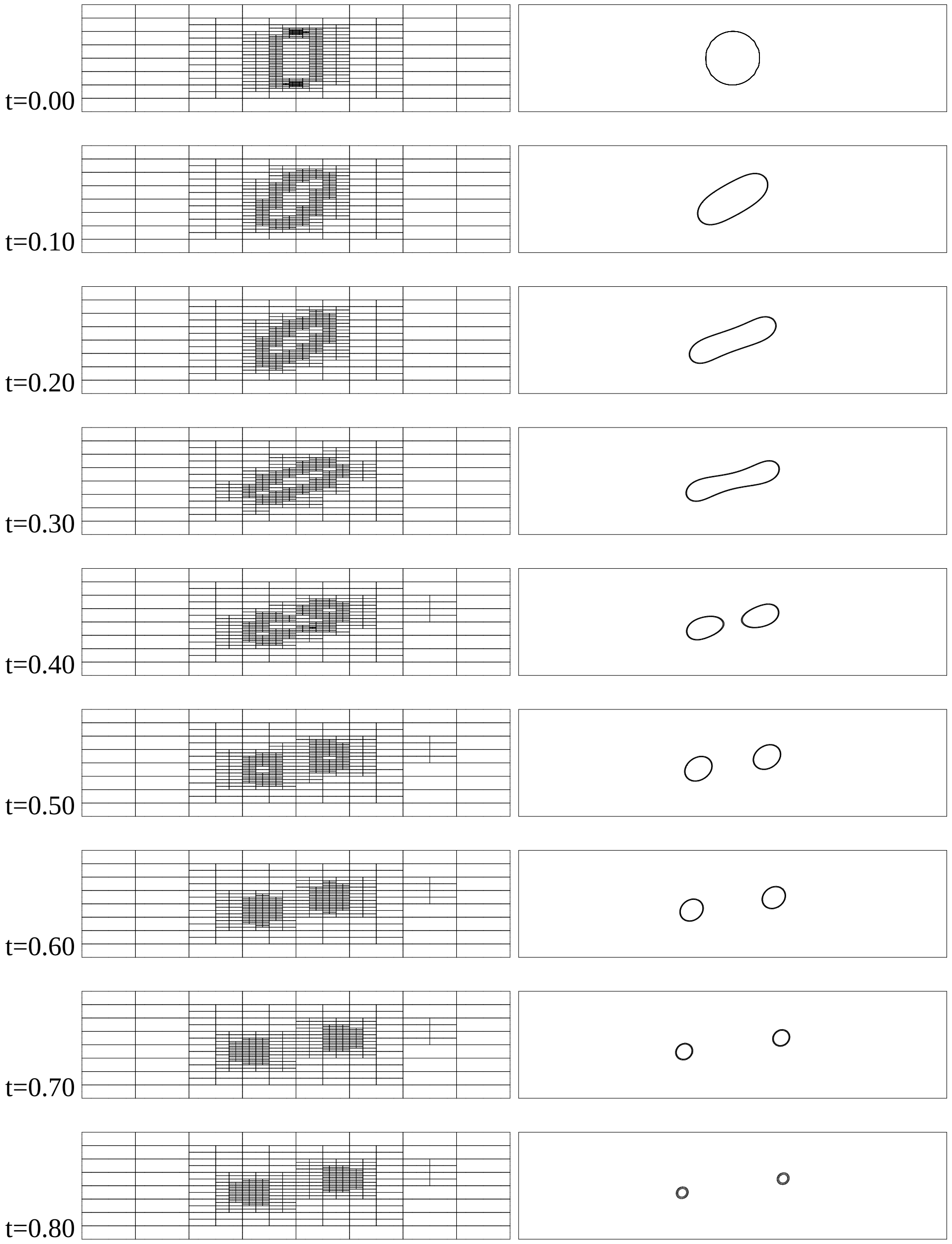


Fig. 5.2: Deformation of a drop for $C = 0.04, Ca = 10.0, P_e = 2.5, N = 4, \Delta t = 1.0 \times 10^{-3}$, adaption based on the solution gradients with a tolerance of $\epsilon = 1.0 \times 10^{-01}$: (*left*) adapted mesh; (*right*) contours of $c = 0.0$.

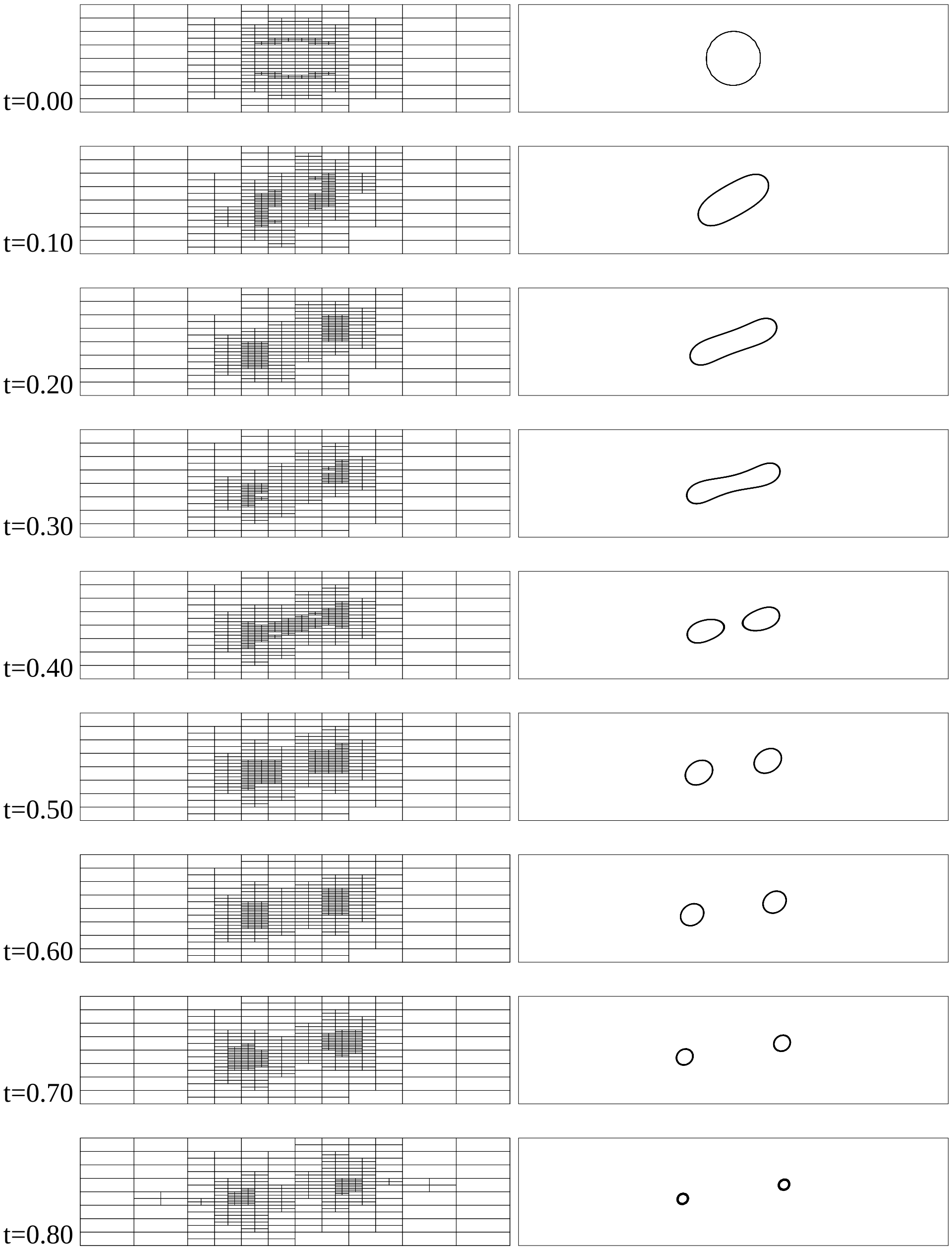


Fig. 5.3: Deformation of a drop for $C = 0.04, Ca = 10.0, P_e = 2.5, N = 4, \Delta t = 1.0 \times 10^{-3}$, adaption based on the local Legendre polynomial spectrum with a tolerance of $\epsilon = 1.0E \times 10^{-06}$: (*left*) adapted mesh; (*right*) contours of $c = 0.0$.

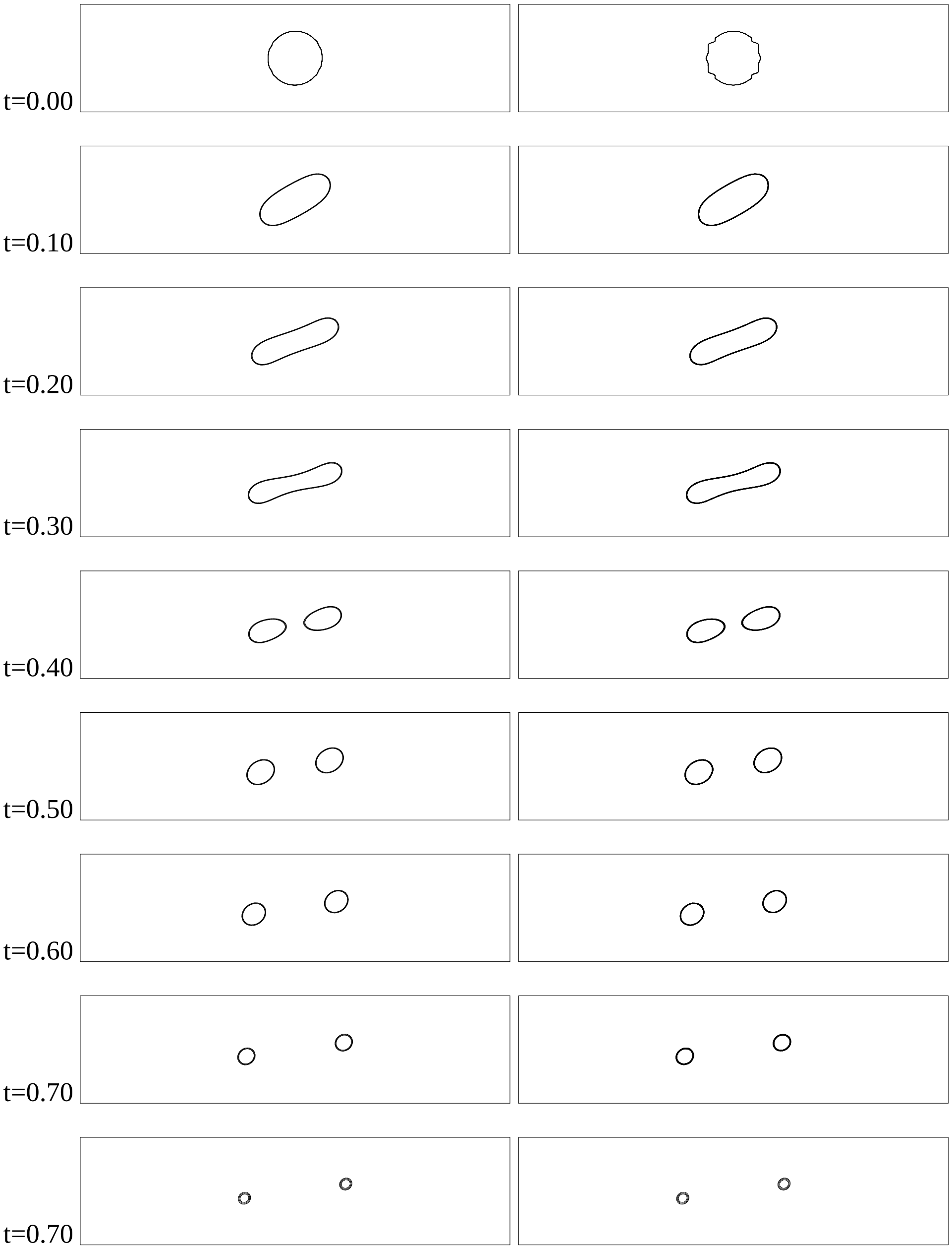


Fig. 5.4: Comparison of a drop deformation for $C = 0.04, Ca = 10.0, P_e = 2.5, N = 4, \Delta t = 1.0 \times 10^{-3}$: (*left*) adaptive, adaption based on the Legendre polynomial spectrum with a tolerance of $\epsilon = 1.0 \times 10^{-06}$; (*right*) conform.

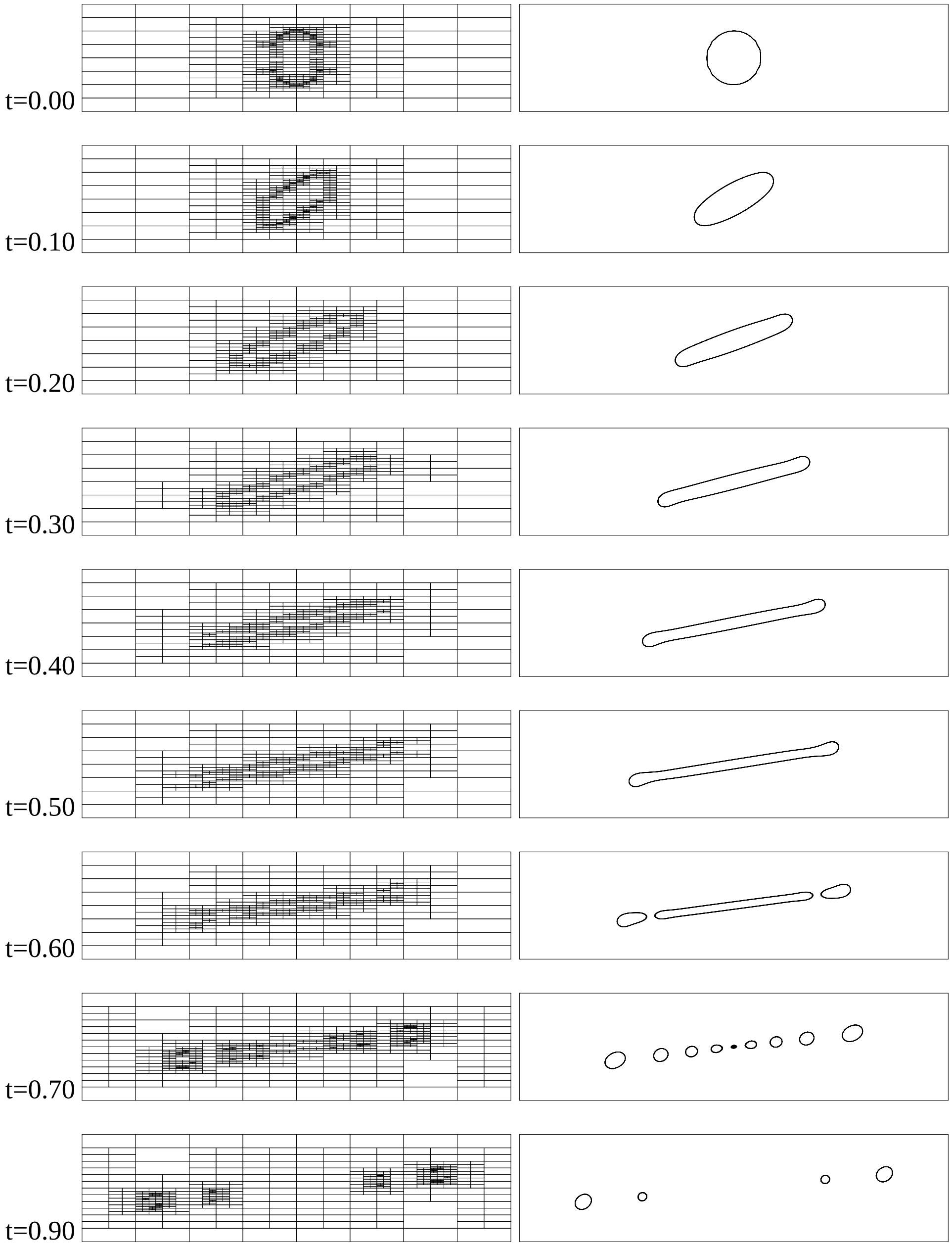


Fig. 5.5: Deformation of a drop for $C = 0.02, Ca = 10.0, P_e = 5.0, N = 4, \Delta t = 1.0 \times 10^{-3}$, adaption based on the solution gradients with a tolerance of $\epsilon = 1.0 \times 10^{-01}$: (*left*) adapted mesh; (*right*) contours of $c = 0.0$.

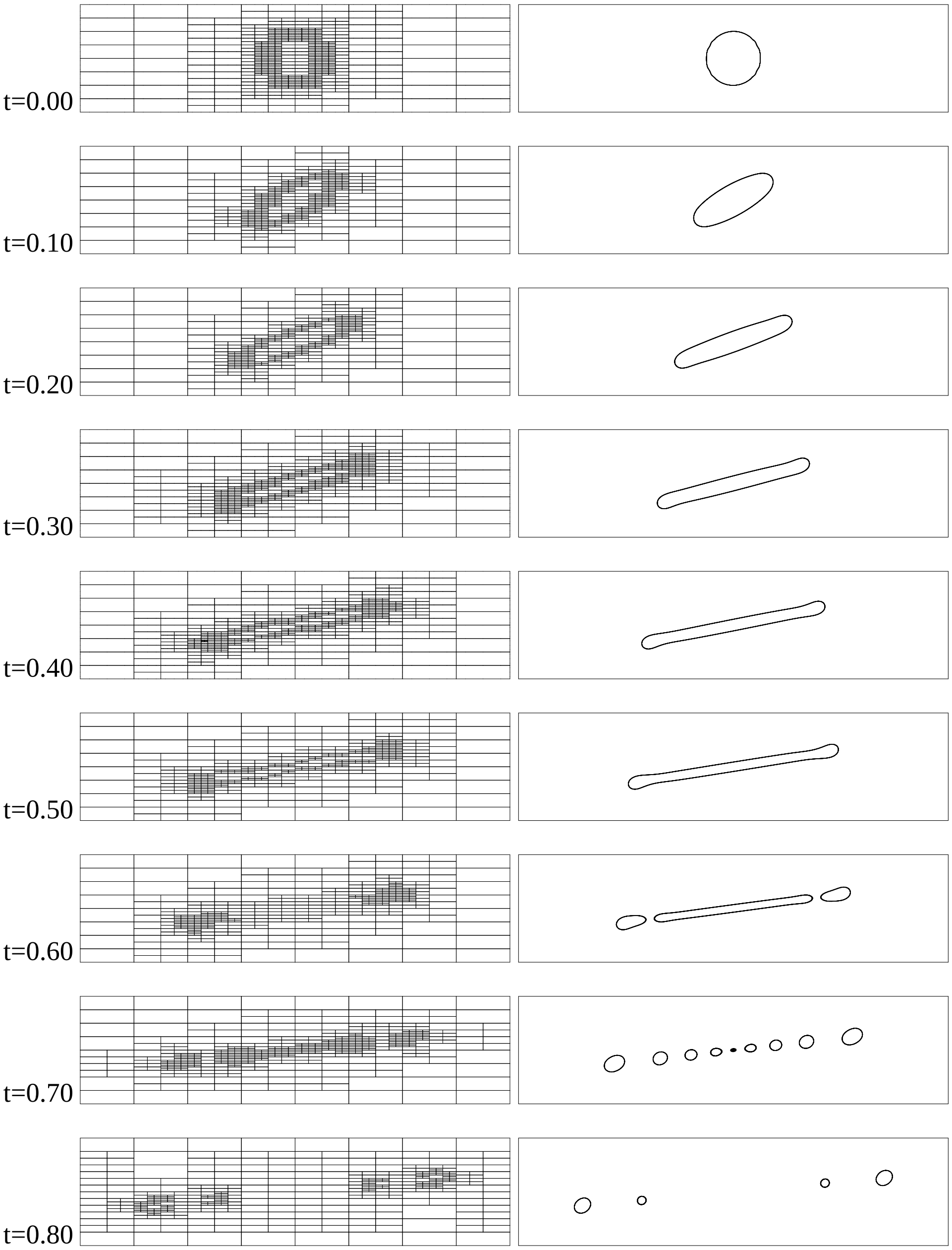


Fig. 5.6: Deformation of a drop for $C = 0.02, Ca = 10.0, P_e = 5.0, N = 4, \Delta t = 1.0 \times 10^{-3}$, adaption based on the local Legendre polynomial spectrum with a tolerance of $\epsilon = 1.0 \times 10^{-06}$: (*left*) adapted mesh; (*right*) contours of $c = 0.0$.

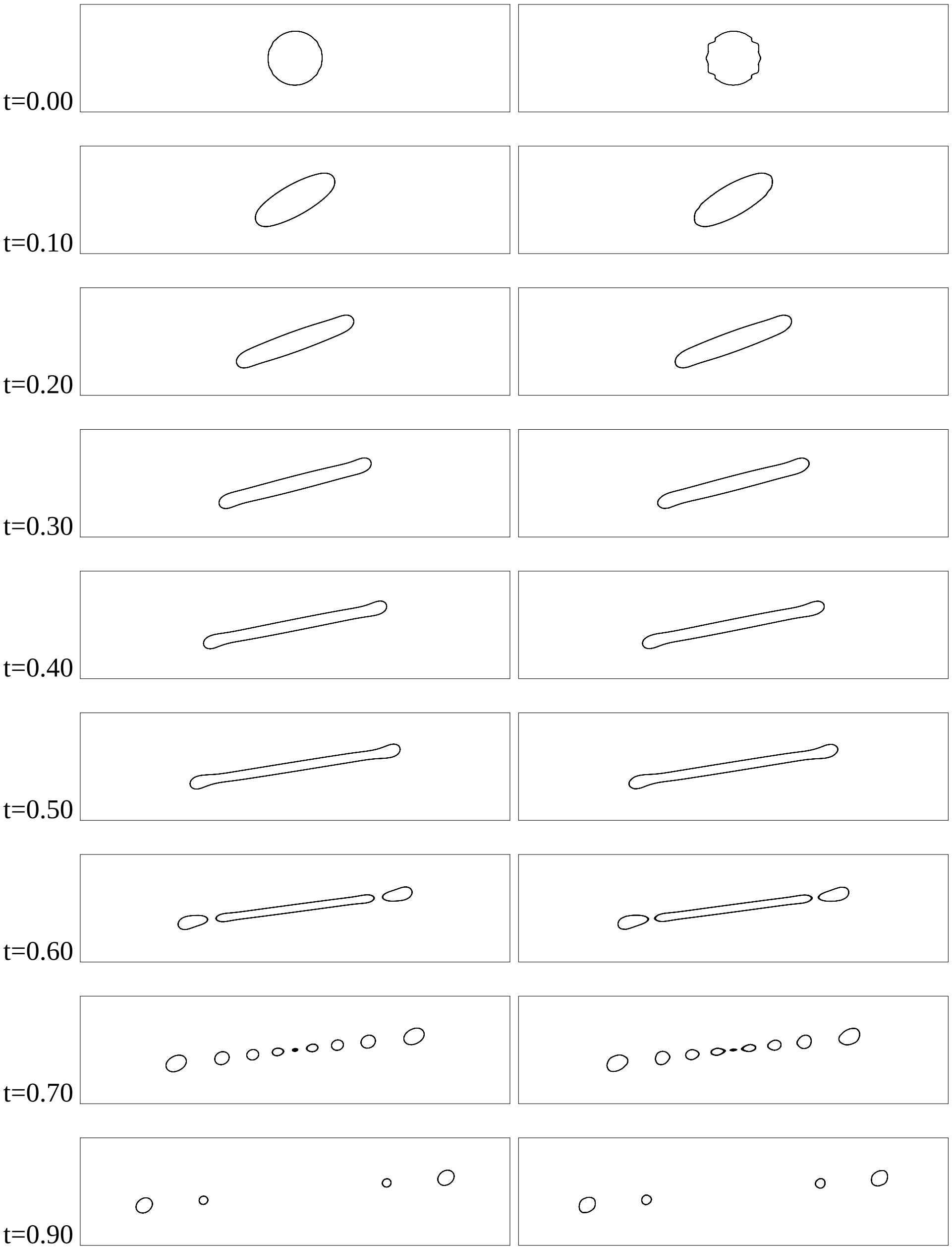


Fig. 5.7: Comparison of a drop deformation for $C = 0.02, Ca = 10.0, P_e = 5.0, N = 4, \Delta t = 1.0 \times 10^{-3}$: (*left*) adaptive, adaption based on the Legendre polynomial spectrum with a tolerance of $\epsilon = 1.0 \times 10^{-06}$; (*right*) conform.

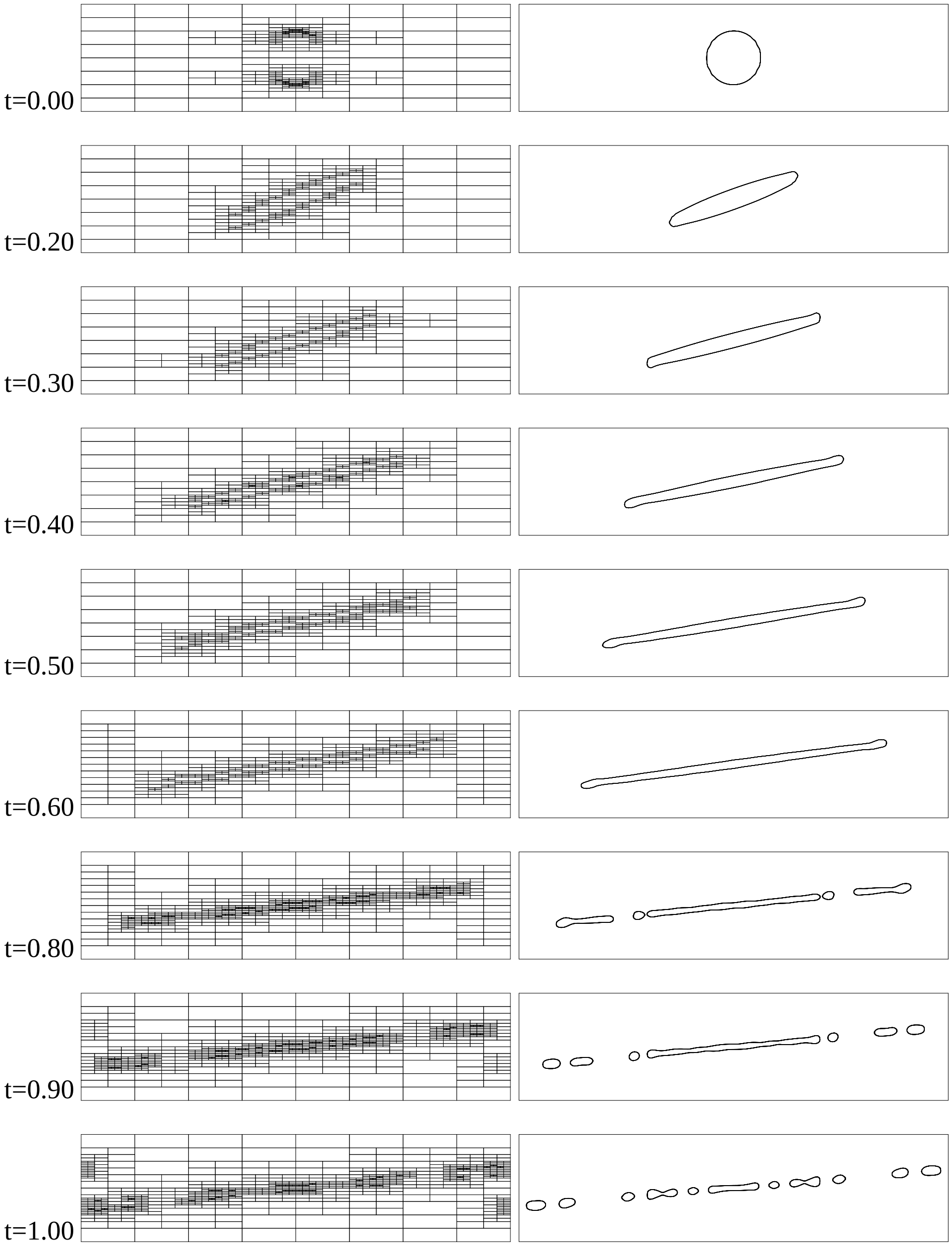


Fig. 5.8: Deformation of a drop for $C = 0.01, Ca = 10.0, P_e = 10.0, N = 4, \Delta t = 5.0 \times 10^{-4}$, adaption based on the solution gradients with a tolerance of $\epsilon = 1.0 \times 10^{-01}$: (*left*) adapted mesh; (*right*) contours of $c = 0.0$.

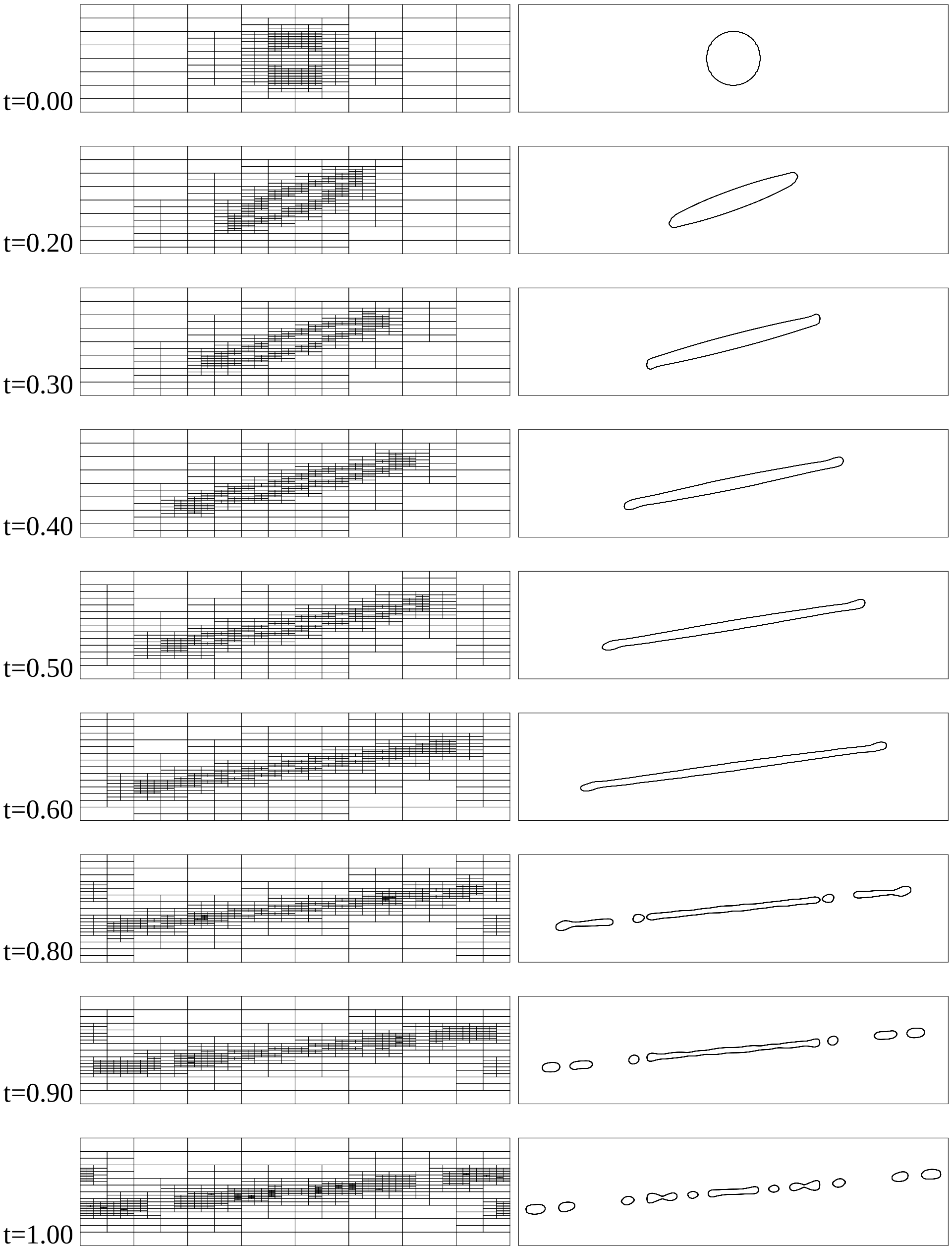


Fig. 5.9: Deformation of a drop for $C = 0.01, Ca = 10.0, P_e = 10.0, N = 4, \Delta t = 5.0 \times 10^{-4}$, adaption based on the Legendre polynomial spectrum with a tolerance of $\epsilon = 1.0 \times 10^{-06}$: (*left*) adapted mesh; (*right*) contours of $c = 0.0$.

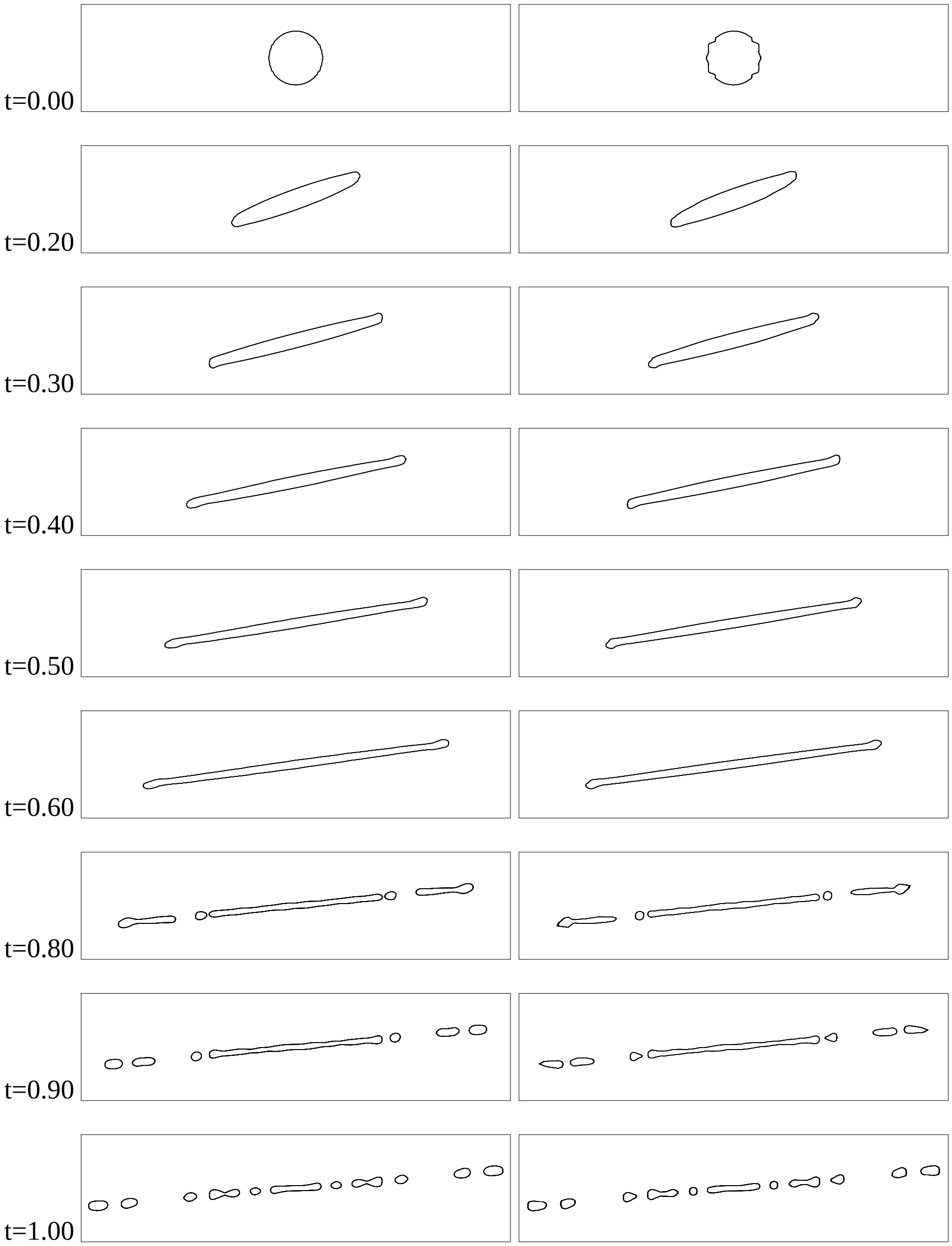


Fig. 5.10: Comparison of a drop deformation for $C = 0.01, Ca = 10.0, P_e = 10.0, N = 4, \Delta t = 5.0 \times 10^{-4}$: (*left*) adaptive, adaption based on the Legendre polynomial spectrum with a tolerance of $\epsilon = 1.0 \times 10^{-06}$; (*right*) conform.

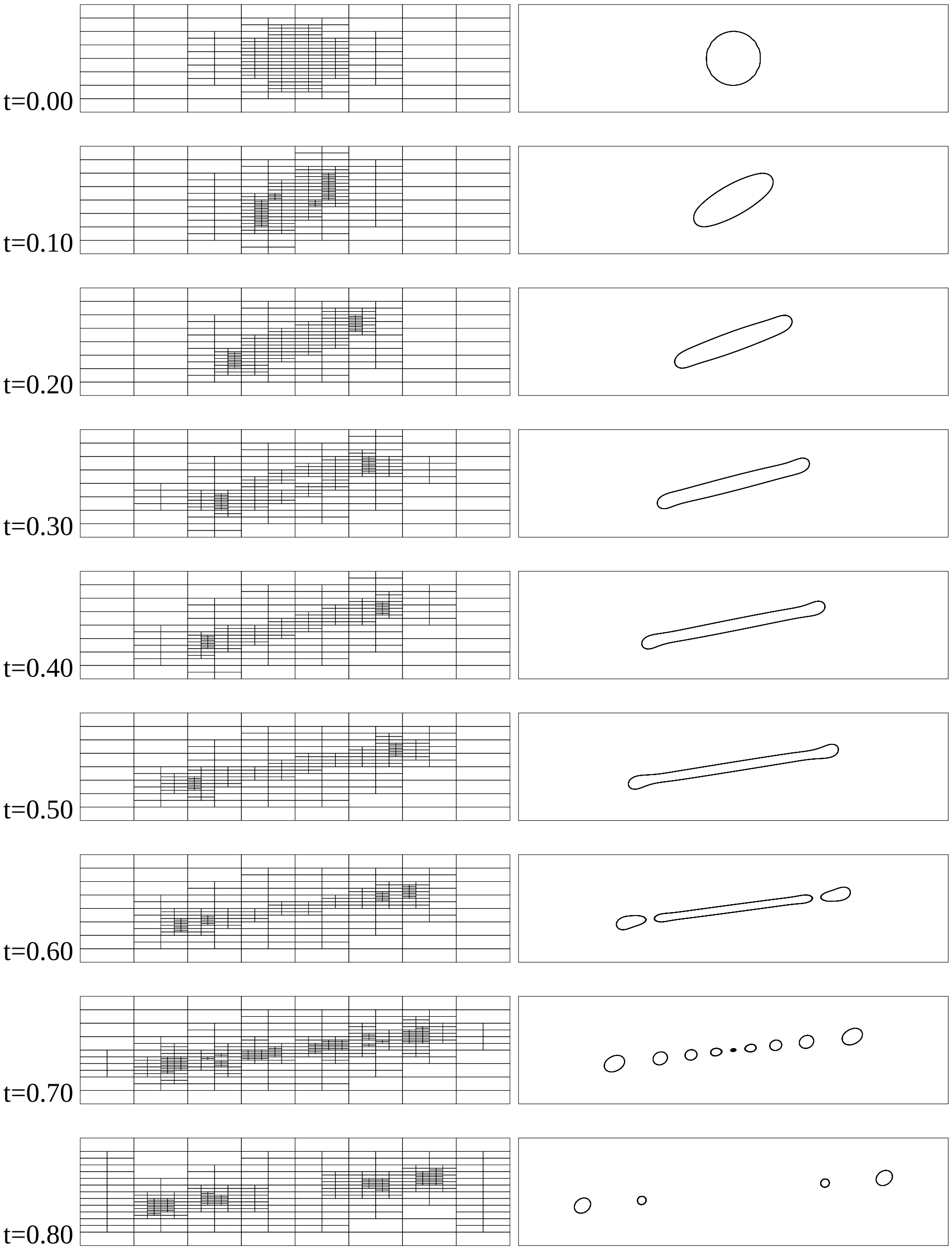


Fig. 5.11: Deformation of a drop for $C = 0.02, Ca = 10.0, P_e = 5.0, N = 9, \Delta t = 1.0 \times 10^{-3}$, adaption based on the local Legendre polynomial spectrum with a tolerance of $\epsilon = 1.0E \times 10^{-06}$: (*left*) adapted mesh; (*right*) contours of $c = 0.0$.

5.4.2 Coalescence of two drops

The second example we present is the coalescence of two drops. To study the capability of our mortar code for the simulation of interactions of these drops in close approach, we consider two drops in the shear flow for different Cahn numbers $C = \{0.04, 0.02, 0.01, 0.005\}$, $P_e = 1.0/C$ and $Ca = 0.10$. The domain has the dimensions (-1, -1) in the left bottom corner and (1, 1) in the top right corner. We consider only refinement based on solution gradients with a tolerance $\epsilon = 1.0 \times 10^{-2}$.

The evolution of the two drops for the Cahn number $C = 0.04$ are shown in figures 5.12 and 5.13. Figures 5.14 and 5.15 illustrate the topological changes of the drops in the $C = 0.02$ case. In figures 5.12(b), 5.14(b), 5.17(b) and 5.19(b), the boundary of the drop is enlarged to show the location of smallest elements created by the refinement. The depth of the refined mesh vary from $d = 6$ to $d = 9$, which indicates that the interfacial thickness is becoming smaller, depending of the value of C. The coalescence of the two drops based on the solution gradients for $C = 0.01$ is shown in figures 5.17 and 5.18. The last case we consider is for $C = 0.005$, which is illustrated in figures 5.19 and 5.20. The profile of the concentration for $c = 0.0$ is shown in figures 5.16 and 5.21.

In all the two-drop cases presented so far, it is seen that the refinement process detects the boundary of the drops, and refines the regions around it during coalescence and breakup.

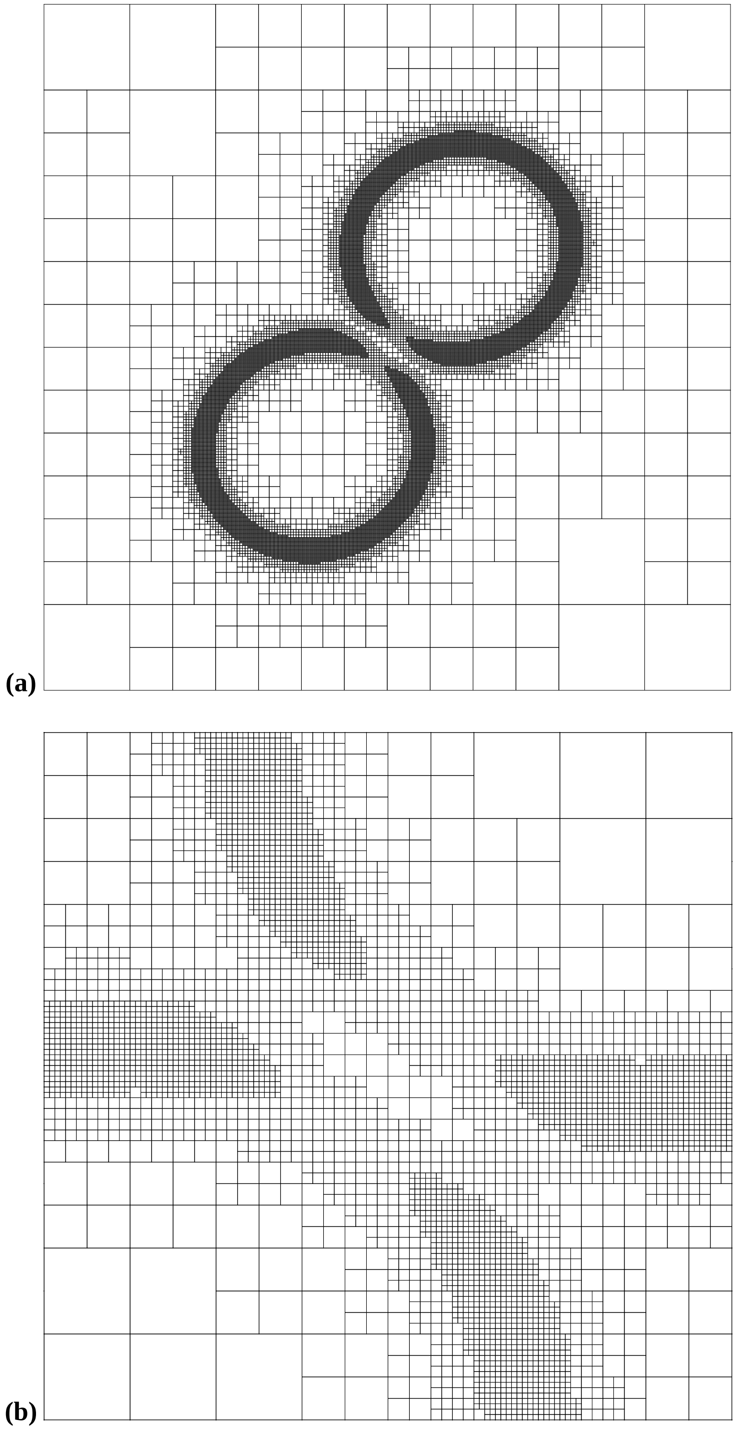


Fig. 5.12: Coalescence of two drops based on the solution gradients with a tolerance of $\epsilon = 1.0 \times 10^{-02}$ and $C = 0.04, Ca = 0.1, P_e = 25.0, N = 7, \Delta t = 1.0 \times 10^{-03}$: (a) adaptive for $t = 0.10$; (b) zoomed adapted mesh for $t = 0.10$.

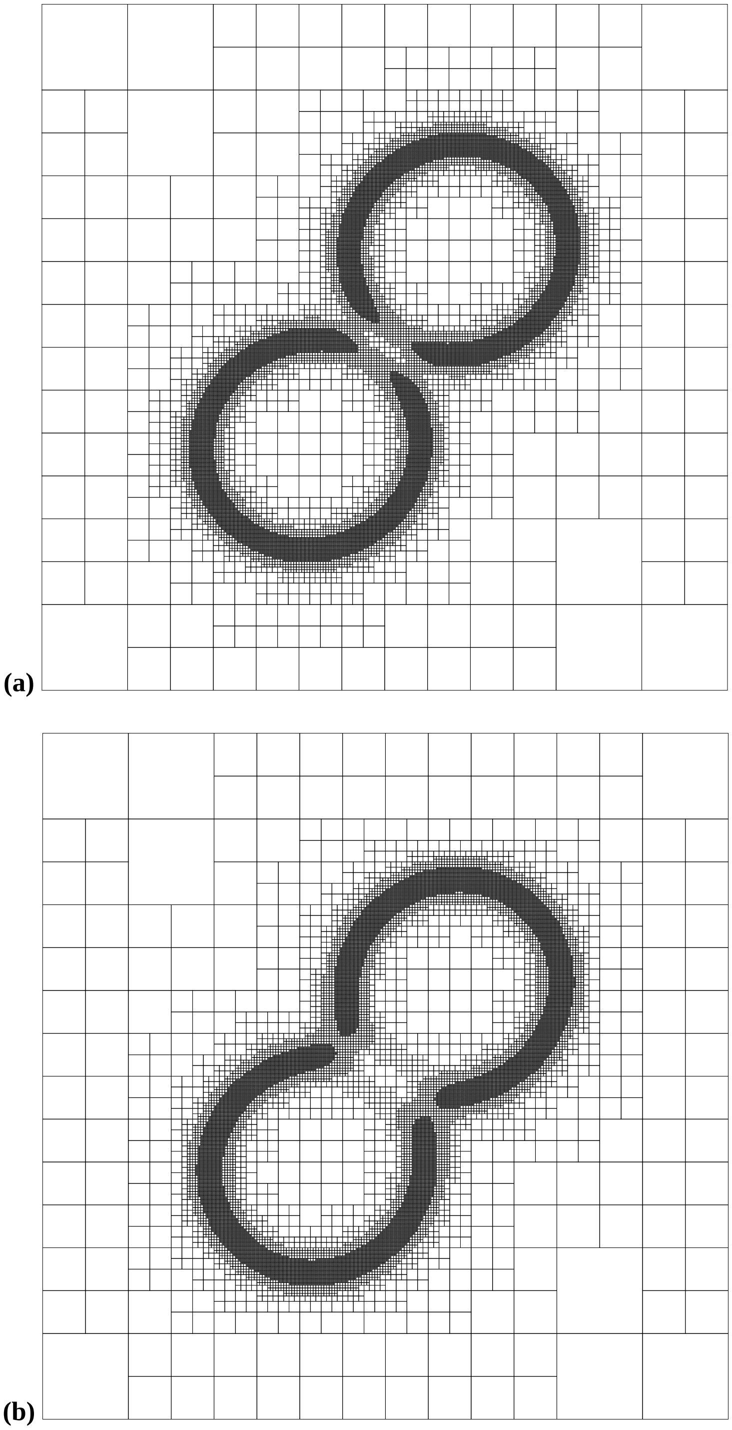


Fig. 5.13: Coalescence of two drops based on the solution gradients with a tolerance of $\epsilon = 1.0 \times 10^{-02}$ and $C = 0.04, Ca = 0.1, P_e = 25.0, N = 7, \Delta t = 1.0 \times 10^{-03}$: (a) adaptive for $t = 0.15$; (b) adaptive for $t = 0.175$.

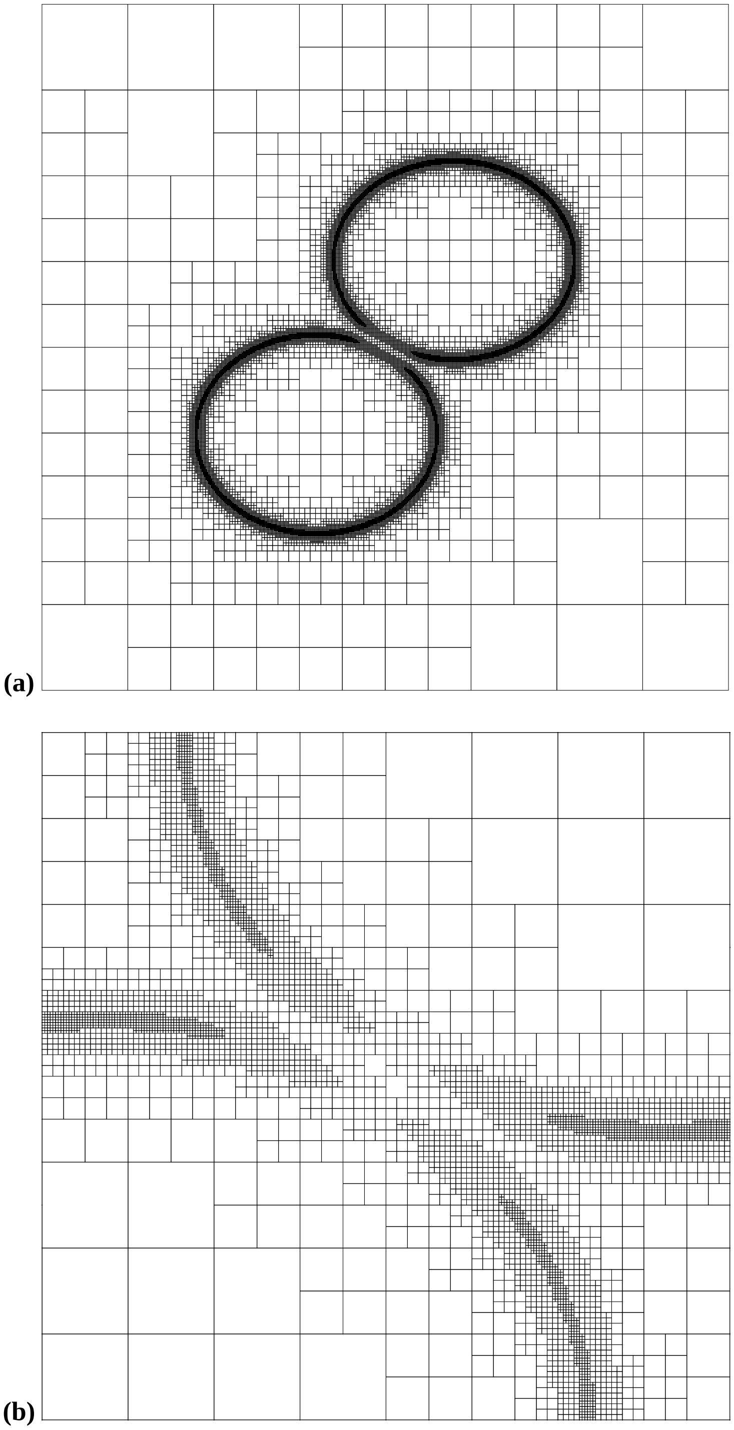


Fig. 5.14: Coalescence of two drops based on the solution gradients with a tolerance of $\epsilon = 1.0 \times 10^{-02}$ and $C = 0.02, Ca = 0.1, P_e = 50.0, N = 7, \Delta t = 1.0 \times 10^{-03}$: (a) adaptive for $t = 0.15$; (b) zoomed adapted mesh for $t = 0.15$.

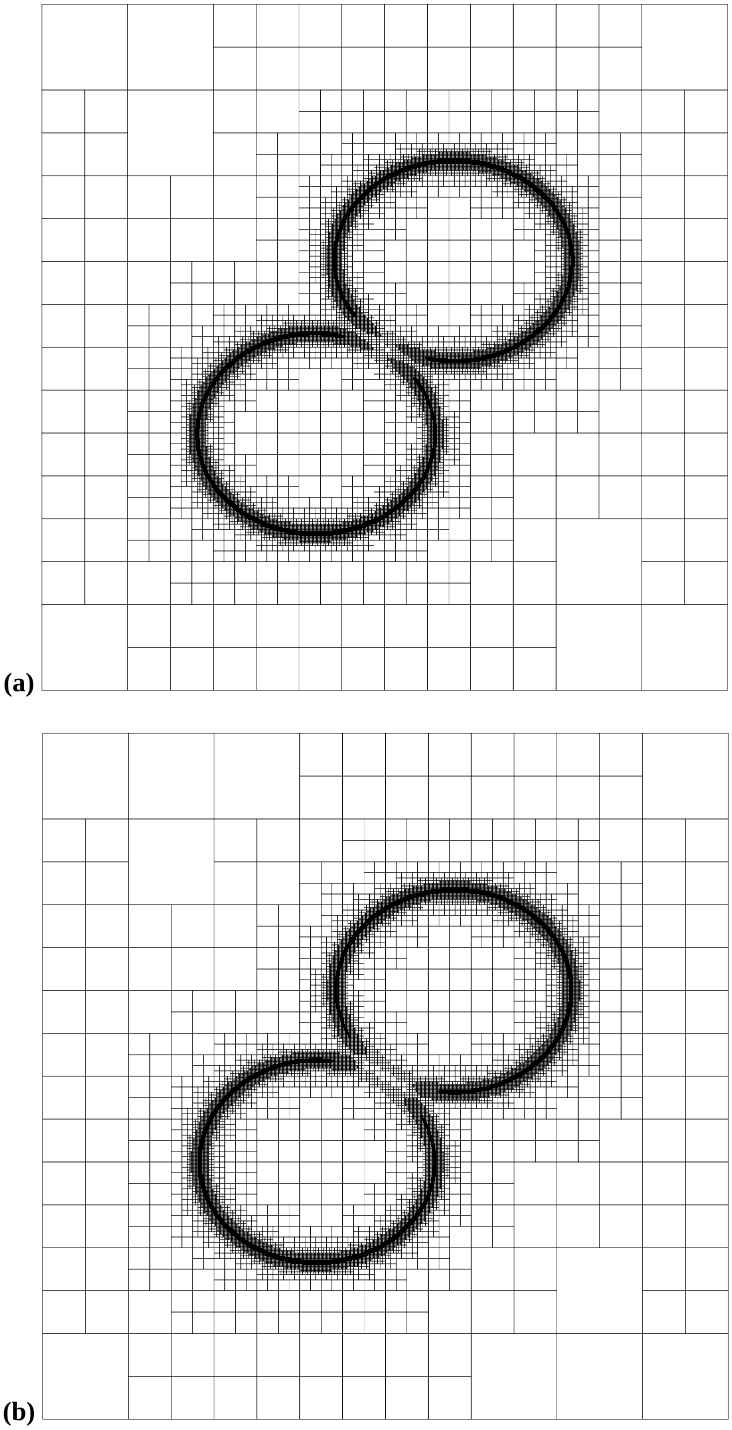


Fig. 5.15: Coalescence of two drops based on the solution gradients with a tolerance of $\epsilon = 1.0 \times 10^{-02}$ and $C = 0.02, Ca = 0.1, P_e = 50.0, N = 7, \Delta t = 1.0 \times 10^{-03}$: (a) adaptive for $t = 0.175$; (b) adaptive for $t = 0.25$.

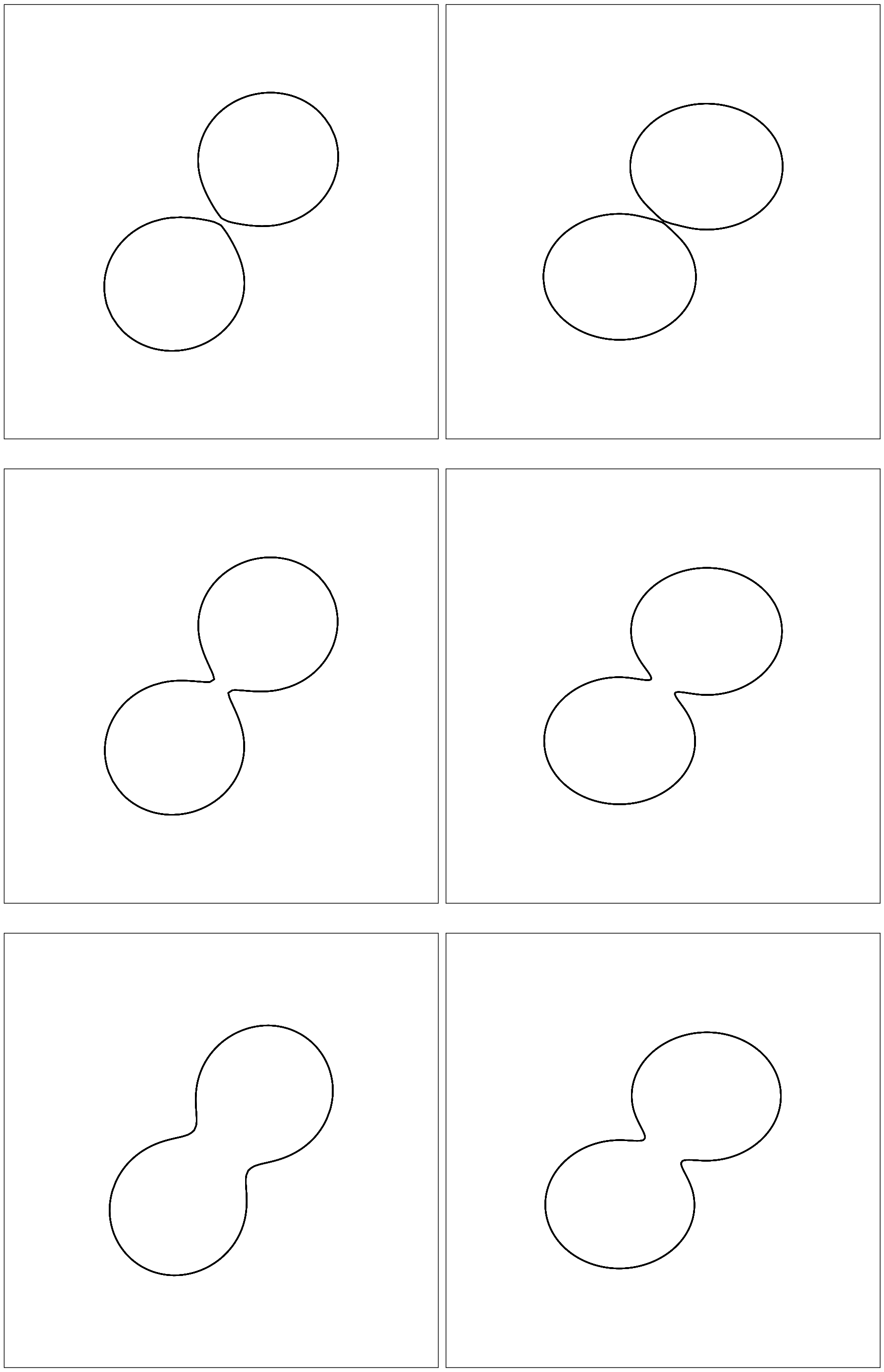


Fig. 5.16: The contour of the concentration $c = 0.0$: (*left*) $Ca = 0.04, t = 0.10, t = 0.15, t = 0.175$; (*right*) $C = 0.02, t = 0.15, t = 0.175, t = 0.25$.

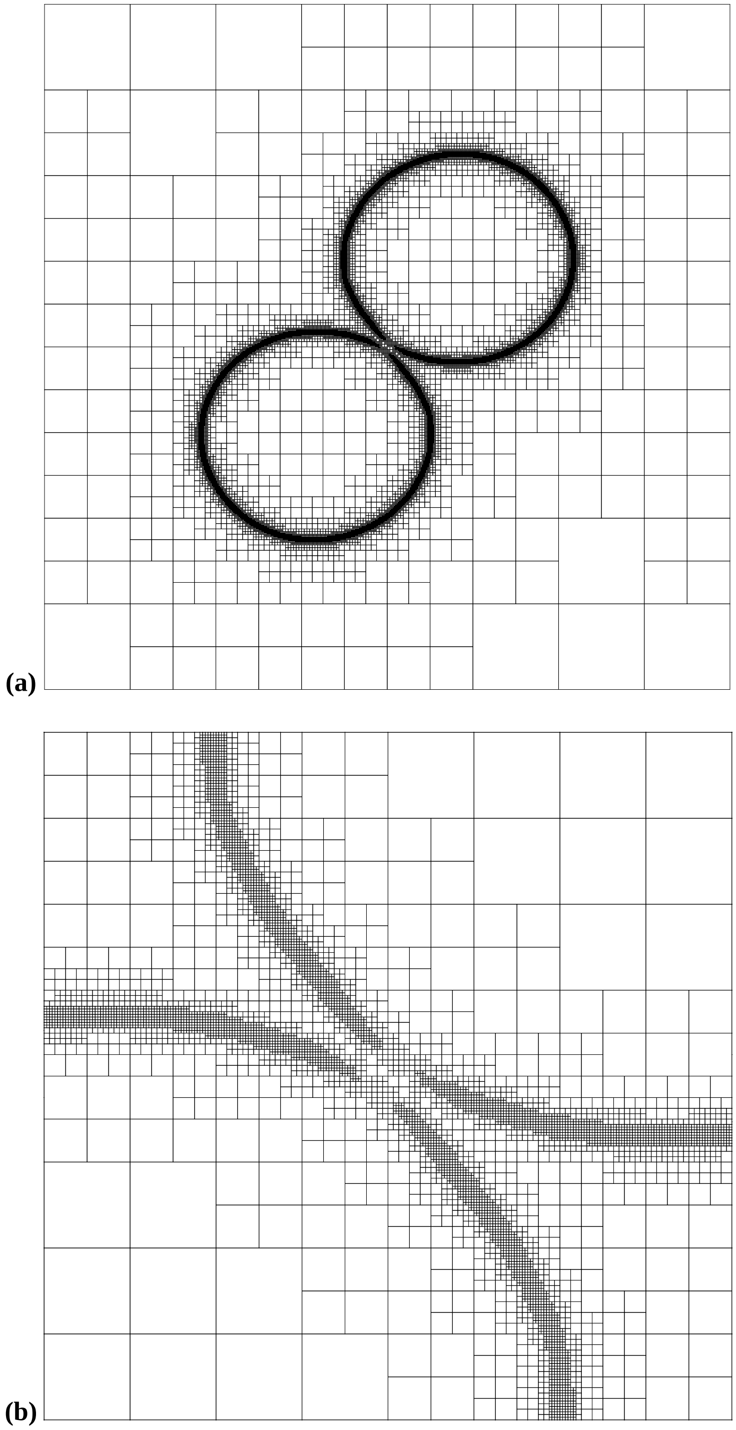


Fig. 5.17: Coalescence of two drops based on the solution gradients with a tolerance of $\epsilon = 1.0 \times 10^{-02}$ and $C = 0.01, Ca = 0.1, P_e = 100.0, N = 7, \Delta t = 5.0 \times 10^{-04}$: (a) adaptive for $t = 0.10$; (b) zoomed adapted mesh for $t = 0.10$.

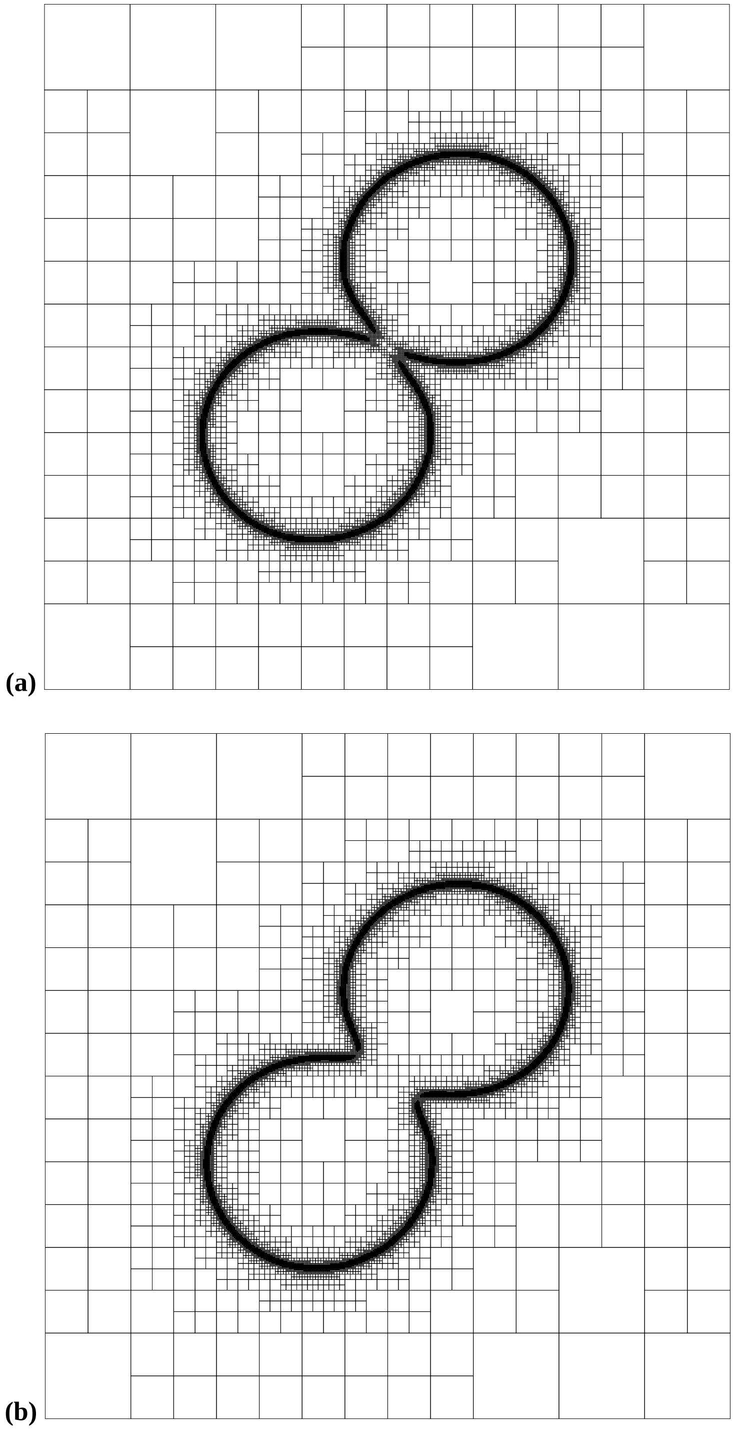


Fig. 5.18: Coalescence of two drops based on the solution gradients with a tolerance of $\epsilon = 1.0 \times 10^{-02}$ and $C = 0.01, Ca = 0.1, P_e = 100.0, N = 7, \Delta t = 5.0 \times 10^{-04}$: (a) adaptive for $t = 0.125$; (b) adaptive for $t = 0.20$.

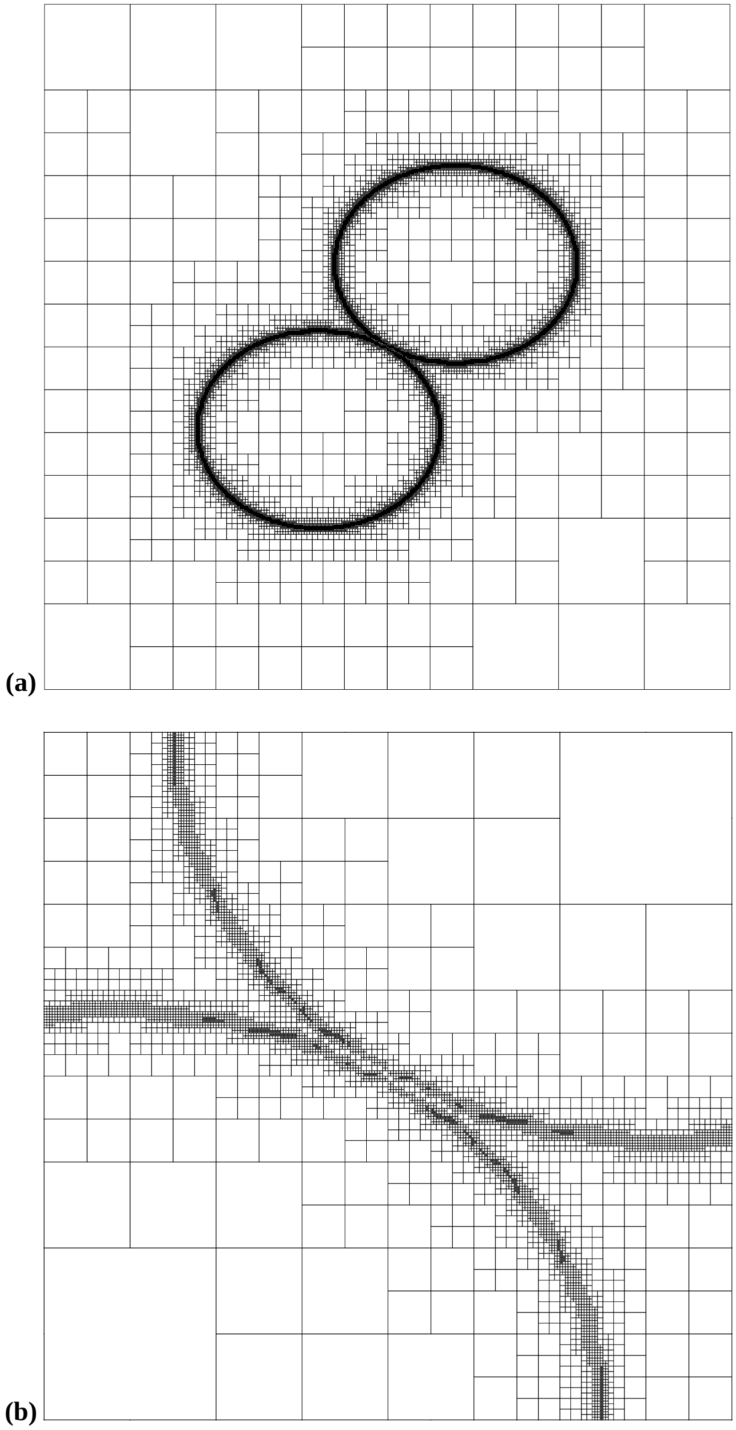


Fig. 5.19: Coalescence of two drops based on the solution gradients with a tolerance of $\epsilon = 1.0 \times 10^{-02}$ and $C = 0.005, Ca = 0.1, P_e = 200.0, N = 7, \Delta t = 5.0 \times 10^{-04}$: (a) adaptive for $t = 0.125$; (b) zoomed adapted mesh for $t = 0.125$.

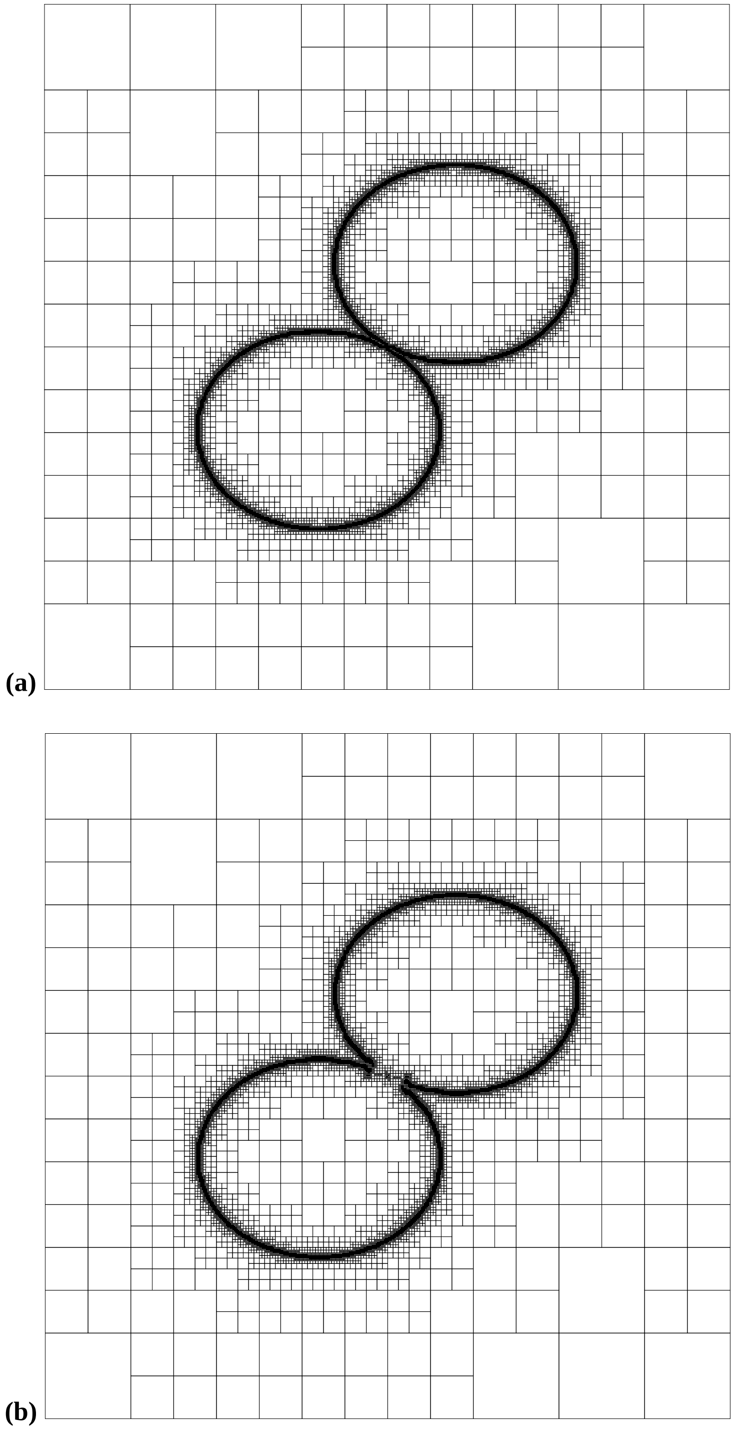


Fig. 5.20: Coalescence of two drops based on the solution gradients with a tolerance of $\epsilon = 1.0 \times 10^{-02}$ and $C = 0.005, Ca = 0.1, P_e = 200.0, N = 7, \Delta t = 5.0 \times 10^{-04}$: (a) adaptive for $t = 0.13$; (b) adaptive for $t = 0.14$.

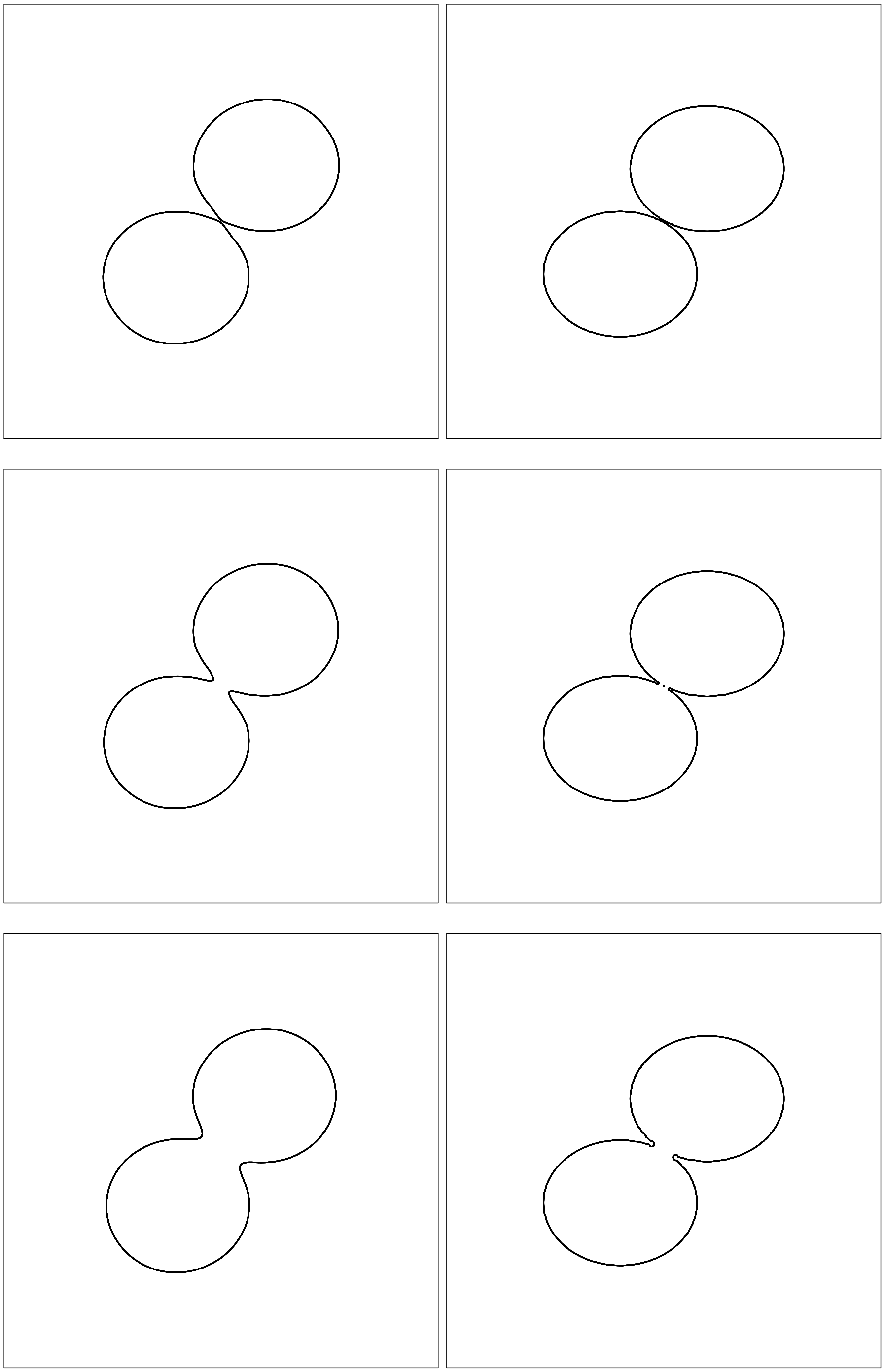


Fig. 5.21: The contour of the concentration $c = 0.0$: (*left*) $Ca = 0.01, t = 0.10, t = 0.125, t = 0.20$; (*right*) $C = 0.005, t = 0.125, t = 0.13, t = 0.14$.

5.4.3 Multi-drop problem

Finally, a more complex example is handled, where a large number of drops is present. For this problem, the initial coarse grid had the same number of elements 8×8 as in the one drop simulation, but the polynomial order was higher ($N = 7$). The simulations are performed on a rectangular mesh with periodic boundary conditions on the left and right side of the domain, with the dimensions (-4, -1) in the left bottom corner and (4, 1) in the top right corner. In the conforming case the grid had 100×50 elements. Figure 5.22 show a few snapshots of the morphology development of a blend consisting of a drop-matrix morphology during shear flow for 18 drops with $C = 0.02, Ca = 10,$ and $P_e = 5$.

Topological changes such as breakup and coalescence are present, and they are captured by the refinement process. With a tolerance $\epsilon = 3.1E \times 10^{-01}$, the adaption based on solution gradients generated about 2500 elements, see figure 5.22(a). Legendre spectrum, figure 5.22(b), produced about 2000 elements with a tolerance $\epsilon = 4.1E \times 10^{-07}$. The gradient criterion detects the boundary of the drops very well, even when they are close to each other. Legendre spectrum groups the drops together in sub-regions, and then refines these sub-regions. For the imposed tolerances, the profile of the concentration is the same in both cases.

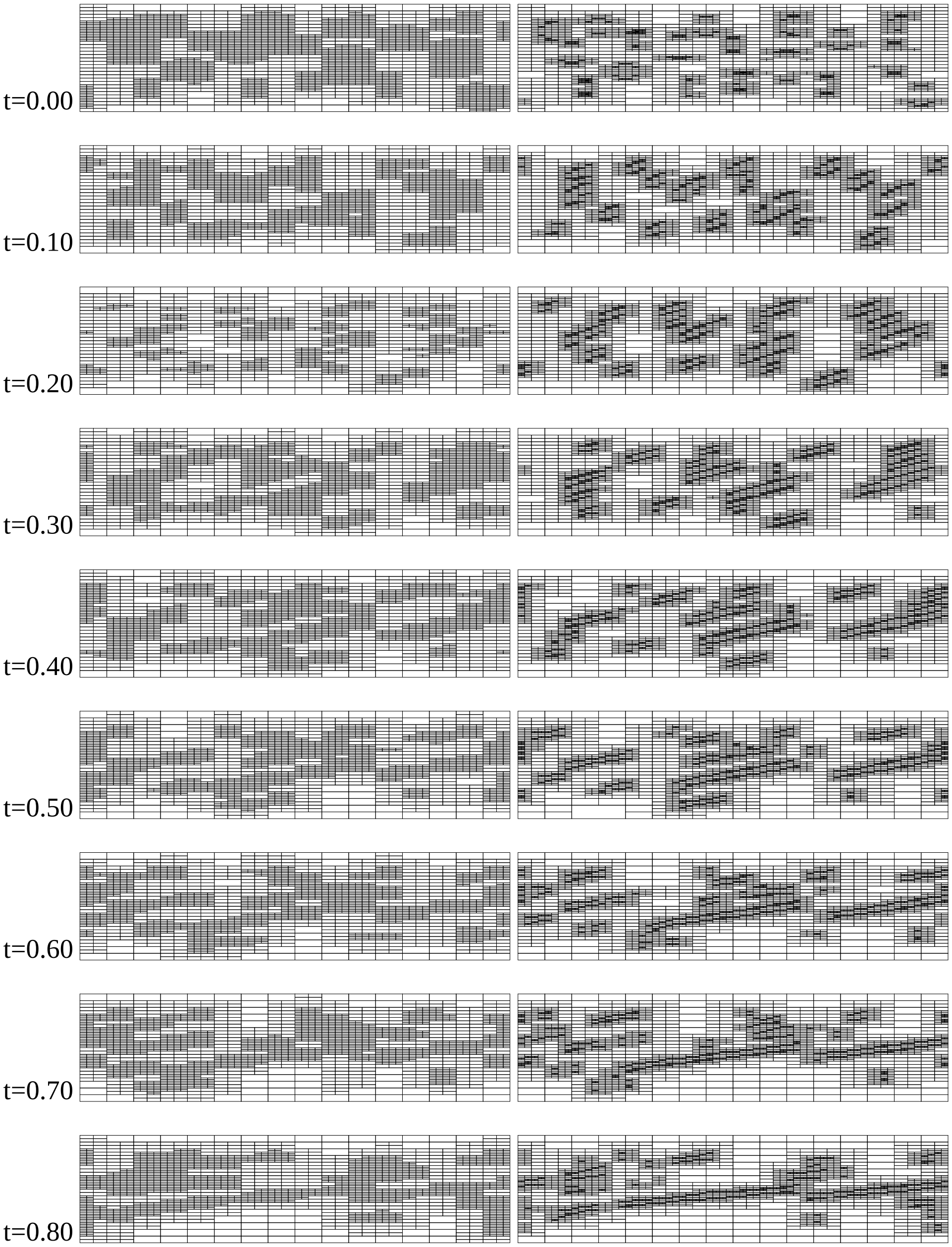
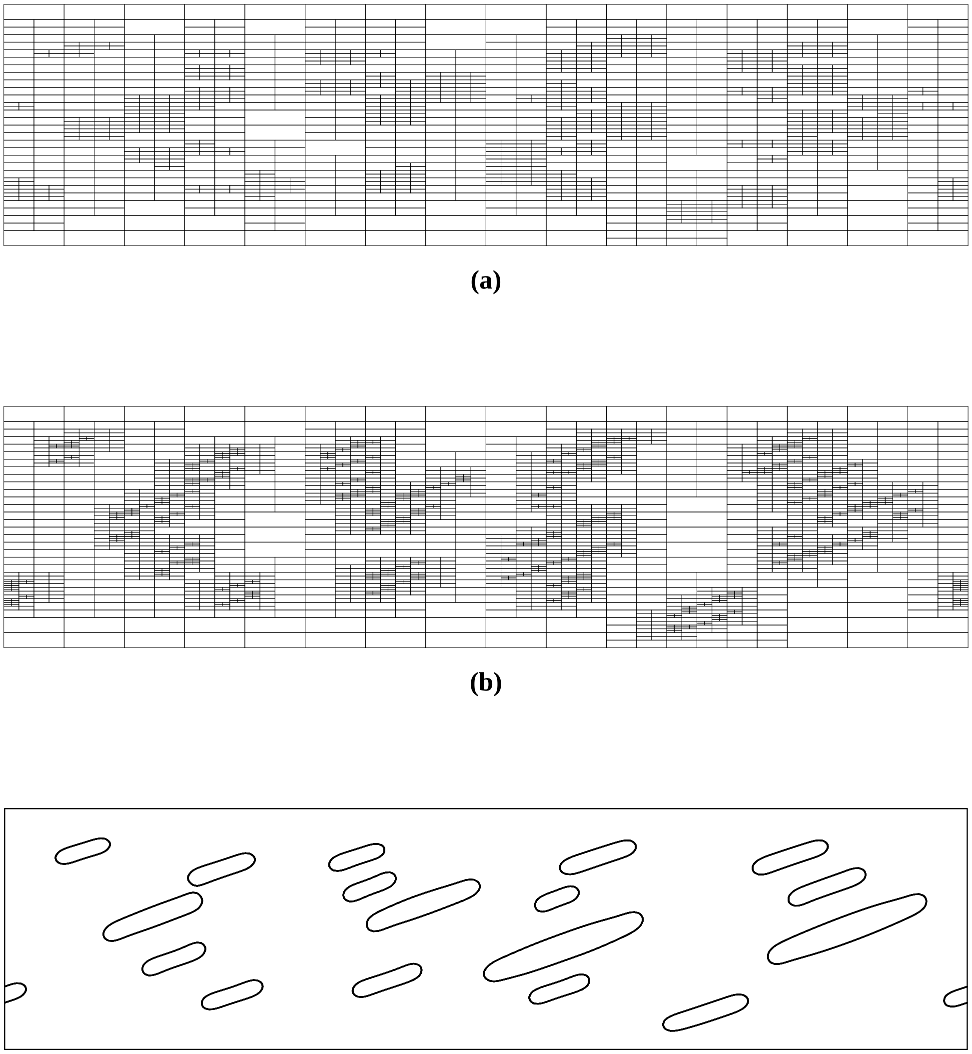


Fig. 5.22: Morphology development of a blending for 18 drops for $C = 0.02, Ca = 10.0, P_e = 5.0, N = 7, \Delta t = 1.0 \times 10^{-3}$: (*left*) adaption based on the local Legendre polynomial spectrum with a tolerance of $\epsilon = 4.1 \times 10^{-07}$; (*right*) adaption based on the solution gradients with a tolerance of $\epsilon = 3.1 \times 10^{-01}$.

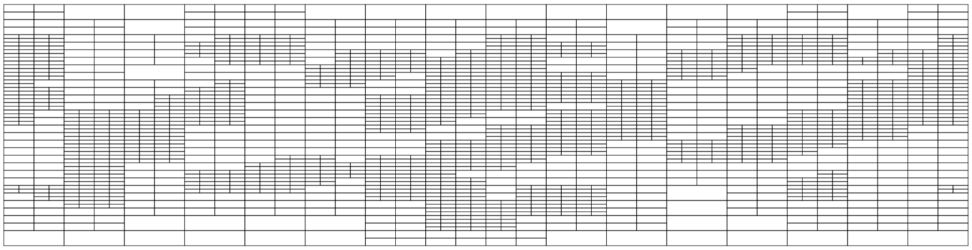


(a)

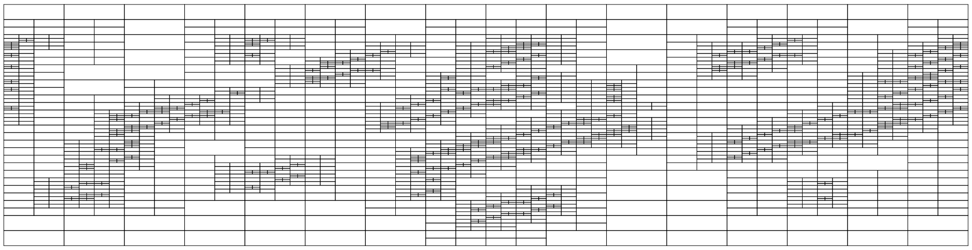
(b)

(c)

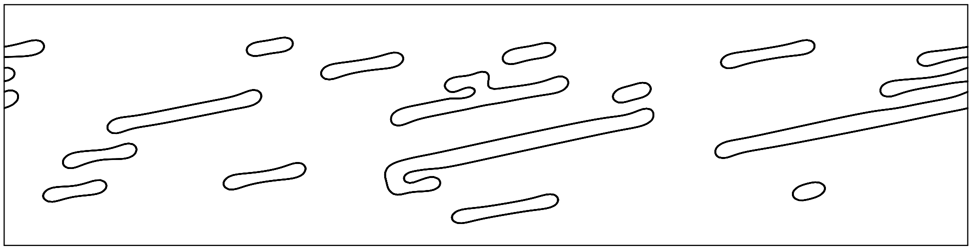
Fig. 5.23: Multi-drop example, $t = 0.1, C = 0.02, Ca = 10.0, P_e = 5.0, N = 7$: (a) adaption based on the local Legendre polynomial spectrum with a tolerance of $\epsilon = 4.1 \times 10^{-07}$; (b) adaption based on the solution gradients with a tolerance of $\epsilon = 3.1 \times 10^{-01}$; (c) profile of the concentration for $c = 0.0$.



(a)



(b)



(c)

Fig. 5.24: Multi-drop example, $t = 0.4, C = 0.02, Ca = 10.0, P_e = 5.0, N = 7$: (a) adaption based on the local Legendre polynomial spectrum with a tolerance of $\epsilon = 4.1 \times 10^{-07}$; (b) adaption based on the solution gradients with a tolerance of $\epsilon = 3.1 \times 10^{-01}$; (c) profile of the concentration for $c = 0.0$.

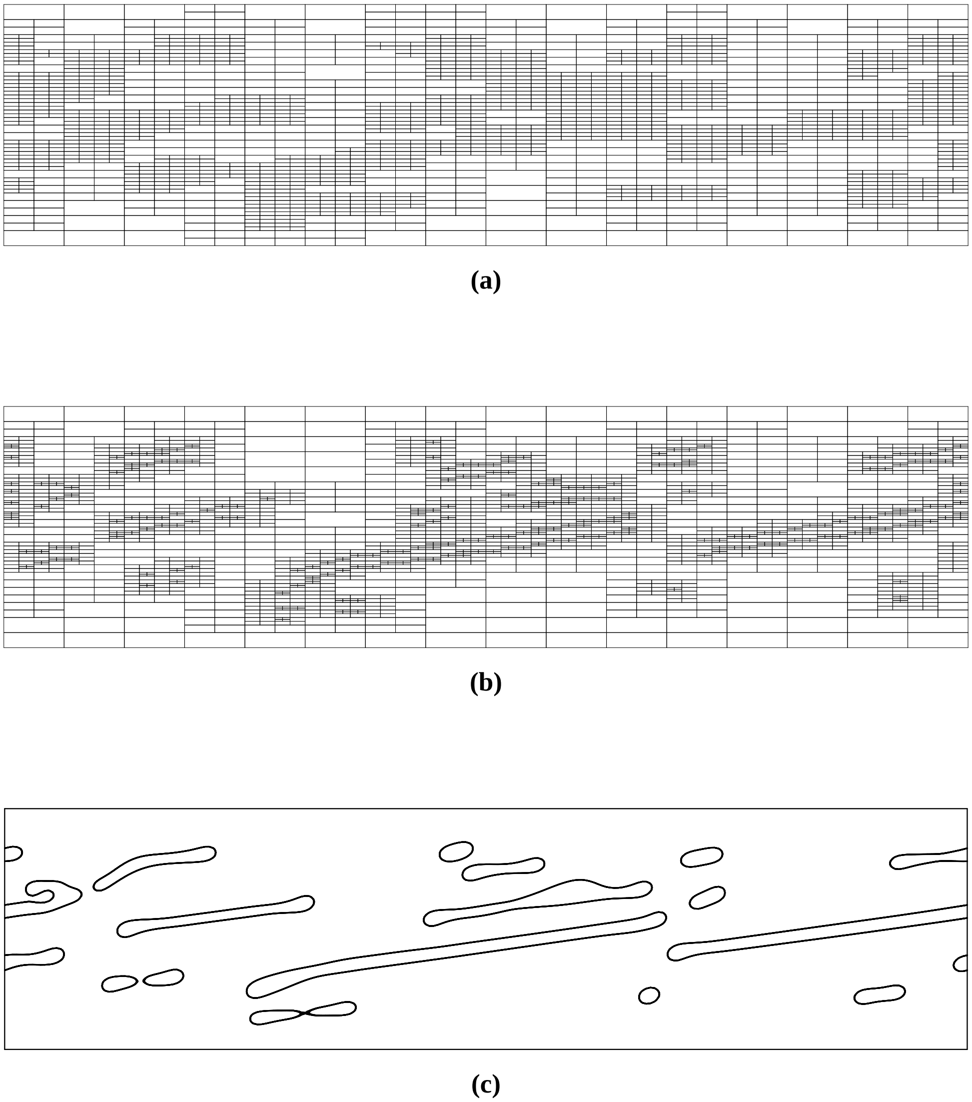


Fig. 5.25: Multi-drop example, $t = 0.6, C = 0.02, Ca = 10.0, P_e = 5.0, N = 7$: (a) adaption based on the local Legendre polynomial spectrum with a tolerance of $\epsilon = 4.1 \times 10^{-07}$; (b) adaption based on the solution gradients with a tolerance of $\epsilon = 3.1 \times 10^{-01}$; (c) profile of the concentration for $c = 0.0$.

5.5 Conclusions

In this chapter the diffuse-interface method was applied to model the morphology evolution of immiscible polymer blends. The method has been applied on a variety of problems ranging from the deformation and breakup of a single drop in a shear flow to simulations with tens of drops. It shown that topological changes such as breakup of drops is implicitly present in the model. The simulations can contribute to the understanding of the structure formation and rheological properties of immiscible blends (Keestra *et al.*, 2003).

For the one drop problem, both refinement criteria, the solution gradients and the Legendre polynomial spectrum, generate a six-level quad-tree mesh with roughly the same number of active elements ($K \approx 420$) using a uniform basis of order $N = 4$. The number of the elements generated by the refinement process depends of the tolerance ϵ that is used. In the two drop simulations, for a tolerance $\epsilon = 1.0 \times 10^{-02}$, approximatively 22000 elements are generated, far more than necessary to solve the problem with a good solution accuracy. In the multi-drop case, the gradient criterion detects the boundary of the drops much better than the Legendre spectrum, which groups the drops together in sub-regions, and then refines these sub-regions.

We can conclude that the adaptive mesh refinement technique, based on the mortar spectral elements, has proven to be an efficient method, which overcomes the problem of proper scaling for large systems. Using less elements than in the conforming case, we were able to track the boundary of the drops and analyse processes such as breakup and coalescence. Since we can identify the transition zone of the interface, based on the gradient of the solution, larger systems can be analyzed. For the particular case of coalescence of two drops a Cahn number could be used which was almost a factor ten smaller as in the conforming case.

Chapter 6

Concluding Remarks and Recommendations

6.1 Conclusions

In this thesis we presented the implementation of a fully adaptive mesh refinement method based on the mortar element method. The conforming spectral element method is limited by its geometric and functional restrictions: the interface between two adjacent elements must be conforming and the order of discretization must be the same. A first attempt to circumvent these restrictions was presented in Chapter 2. The cases presented show that a simple repositioning of the element boundaries (or equivalently a change in their size) results in an error two order sof magnitude lower for the same discretization parameters K. Also, increasing the order of discretization N improves the solution's accuracy but at a slow rate.

The new mortar discretization, presented in Chapter 3, relives the spectral method of the above limitations, relaxing the interface matching conditions to allow local refinement. The refinement can be achieved by varying the polynomial degree for one element to the next and/or by allowing multiple elements to share a single edge of an adjacent element. Such strategies improve the geometrical flexibility of the spectral discretization, allow for mesh adaptivity, and help circumvent the loss of accuracy near singularities. The single mesh *a posteriori* error estimators for the spectral element methods are important for the implementation of a fully adaptive mesh refinement process.

Three criteria have been implemented based on: the solution gradients, the exponential fit of the Legendre polynomial spectrum, and the trace of the Legendre polynomial spectrum. The error estimators prove to be accurate, as demonstrated in Chapter 3. However, in some cases, the gradient refinement criteria and the trace of the polynomial spectrum criteria lead to nearly complementary grids. Clearly, the local trace spectrum indicates the correct location for the refinement. Since the mortar element method allows localized refinement, the adaptive mesh process is efficient. The same solution accuracy can be achieved in the non-conforming case with roughly less than 50% points than in the conforming case.

Chapter 4 illustrates the Object Oriented (OO) approach of the implementation. The development of a large codes for scientific computing is generally an error-prone and time-consuming process. Modularity and code reuse can be achieved by using OO design and programming techniques. Since Fortran codes dominated the field of scientific computing, and since there are many well-tested and documented non-C++ subroutine and function libraries, we used a hybrid approach for the implementation: Fortran and C++. Interfacing C++ and Fortran for a heterogeneous cluster of machines was not an easy task, but the wrapper developed for the Fortran subroutines made the task easier. An important observation is that, the code implemented has little connection with the physics or engineering problem being solved. It is commonly the case that about 70% of an adaptive code, written in a conventional programming system, is concerned with procedurally realizing dynamic distributed data structures on top of static data structures, such as Fortran arrays.

To support parallelism, the Voxel Data Base (VDB) structure has been implemented. It allows the dynamic adapted mesh to migrate between different processors.

Finally, Chapter 5 shows the benefits of the non-conforming formulation and error estimators with numerical solutions to the diffuse-interface modelling of the morphology and rheology of immiscible polymer blends. Based of the adaptive mesh refinement technique, small interface thickness was tracked, avoiding excessive computational times as in the conforming case. Because the systems we solved were coupled, the mortar element implementation was extended to support more that one dof per node.

In summary, we implemented three key elements needed to make the spectral element method fully automatically adaptive: the mortar element method, the error estimators (refinement criteria) and the dynamic mesh structure (VDB). The most interesting parts of the implementation are:

1. the build-in refinement criteria, which provide a heuristic error estimate that is independent of the system being solved,
2. the dynamic mesh structure, that updates the entire refined mesh during the computation,
3. the integration of the **SEPRAN** package into the adaptive refinement process, which facilitates the re-use of the existing Fortran code.

6.2 Recommendations

Since only h-refinement has been implemented, and we want to take full advantage of the non-conforming formulation, the next step will be the implementation of the functional non-conforming case: the order N of the discretization within adjacent elements is different. The combination of h-refinement and N-refinement improves on the geometrical flexibility of the non-conform spectral discretization. The least squares best fit to the decay of the Legendre spectrum of each element discretization provides a decay rate, which can in turn be used to extrapolate the spectrum to infinity. The decay rate σ_e indicates insufficient resolution if $\sigma_e < 1$ and good resolution if $\sigma_e > 1$. The refinement process can use the decay rate to decide whether it increases the number of elements and decreases accordingly the polynomial degree, or whether is has to move elements and reconstruct the grid.

Another issue that should be addressed is parallelism. The basis for parallelism was implemented in the Voxel Data base -VDB component. In a parallel adaptive computation, the mesh changes during the computations, necessitating a dynamic redistribution of data. Mesh data must not only support the adaptivity, but also its dynamic redistribution. In order to support the parallelism, algorithms for mesh partitioning and dynamic load balancing have to be implemented. Our goal is to extend the software implementation to manage and distribute data across the processors of a parallel computer as part of an adaptive scientific computation, using a partitioning model containing the actual dissection of the domain into sub-domains. The partitioning model has to be independent of the mesh structure and can be used to manage other types of distributed data by deriving appropriate C++ classes. Load balancing will be another issue to be addressed. Dynamic load balancing in general has progressed significantly in recent years, but many challenges remain. Like the computational costs of elements, the communications costs associated with boundary entities may not be uniform. There could be a performance penalty for having entities on the inter-processor boundaries, and this information needs to be available to a load balancer.

Appendix A

Refinement Criteria

A.1 The error estimators

In this Appendix we derive the error estimators, first for the one-dimensional case and then for the two-dimensional case. Let us consider the one-dimensional case in which the numerical solution for the one-dimensional boundary layer problem :

$$-u_{xx} + \lambda u_x = 0 \;\; on\, \Omega = [0,1],\; u(0) = 0,\; u(1) = 1 \;, \tag{A.1}$$

corresponds to the truncated sum $u_h|\Omega^k = \sum\limits_{n=0}^{N} a_n^k L_n(r)$ where L_n is the n^{th} order Legendre polynomial.

For a geometrically converging series, the error is the order of the first (leading order) missing term, in this case a_{N+1}^k. We know that the quadrature is only exact for polynomials of order $\leq 2N-1$ and for a_N^k the order is $> 2N-1$, since $a_N^k = \int L_N L_N$. Due to the error introduced by the quadrature, we consider a_N^k as the leading error term since it is inexactly calculated. The coefficient a_N^k can be computed using the following formula:

$$a_N^k = \frac{2N+1}{4} \sum_{n=0}^{N} \rho_n u_n^k L_N(\xi_n), \tag{A.2}$$

where ρ_n and ξ_n are the Gauss-Lobatto weights and collocation points respectively, u_n^k are the collocation points values of u. The coefficients a_N^k are calculated in the computational space, rather that the physical, and it is independent of the element size. This is an important observation, because we want to determine how well the solution is approximated, and not to produce an absolute error relative to the domain size. As an error estimate is generated in each element, the a_N^k may be used to determine relative resolutions in different elements. In order to be efficient in finding the necessary refinements, we must provide more detailed information than a simple estimate of the error.

A.2 Approximation errors

Consider any arbitrary problem:
Find $u_h \in X_h$ solution to $\mathcal{L}u = f$ by a spectral element method.

We know that the error between the exact solution u and the spectral element approximation u_h in bounded by :

$$\|u - u_h\| < C \inf_{v_h \in X_h(\Omega_k)} \|u - v_h\|. \tag{A.3}$$

The best polynomial approximation to the function u is the projection $\Pi_h u$ of u onto X_h, the function which minimizes the error. We know that u_h is an element of X_h but it is not equal to the projection $\Pi_h u$ as illustrated in figure A.1.

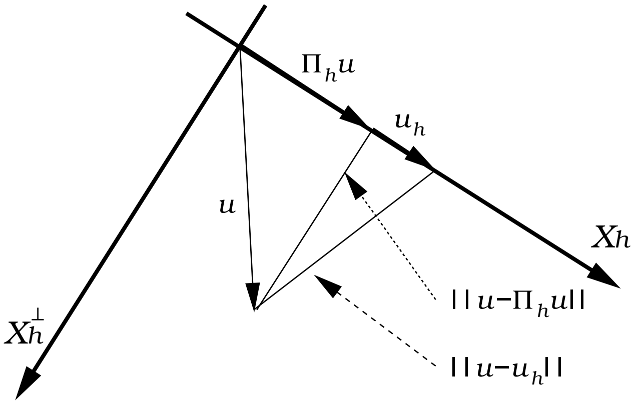


Fig. A.1: Illustration of the numerical and exact solution in the approximation space and its orthogonal complement [Mavriplis (1990)].

The error introduced by the spectral element approximation $\|u - u_h\|$ is the contribution of two error terms

$$\|u - u_h\| \leq \|u - \Pi_h u\| + \|u_h - \Pi_h u\|. \tag{A.4}$$

In the inequality (A.3) the constant C is close to 1 for the $\mathcal{H}^1$ norm. This implies that the term $\|u - \Pi_h u\|$ in A.4 dominates the actual error. Since, we don't want to estimate the error based of the smoothness of the solution, we have to devise an error estimate which can be derived from the calculated solution alone. The spectral element discretization can be seen as a series representation of orthogonal increasing order polynomials. The exact solution is an infinite sum and the spectral element solution is a truncated version of the sum, therefore the error can be estimated by the missing term of the sum.

Since an exact solution is not available, we have to estimate this extra terms. The estimation is based on the extrapolation of the available terms of the sum, which can be computed directly from the numerical solution. Due to extrapolation an extra error is added as follows :

$$\|u - u_h\| \leq \|u - \tilde{u}\| + \|\tilde{u} - u_h\| \tag{A.5}$$

where $\tilde{u}$ is the extrapolated approximation to u. The error between the exact and numerical solution is bounded by

$$\|u - u_h\| \leq \|u - \tilde{u}\| + \|\tilde{u} - \Pi_h \tilde{u}\| + \|u_h - \Pi_h \tilde{u}\| \tag{A.6}$$

where

$$\|u - \tilde{u}\| \quad - \quad \text{is the extrapolation error} \tag{A.7}$$

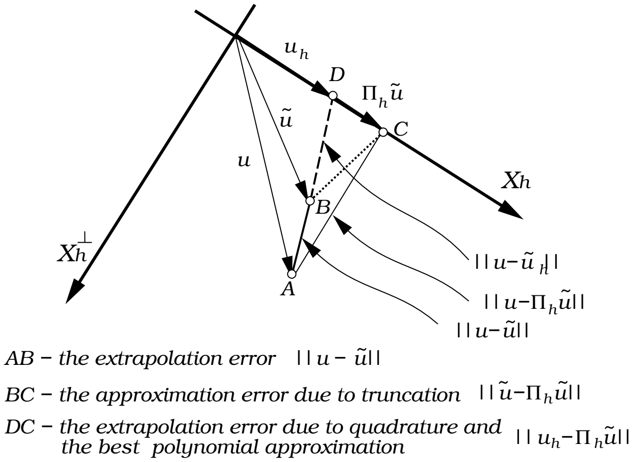


Fig. A.2: Illustration of the approximation error contributions [Mavriplis (1990)].

$\|\tilde{u}-\Pi_h\tilde{u}\|$ – is the approximation error due to truncation (A.8)

$\|u_h-\Pi_h\tilde{u}\|$ – is the approximation error due to quadrature and the best polynomial approximation. (A.9)

The latter two contributions can be grouped together in

$$\epsilon_{est}=\|\tilde{u}-\Pi_h\tilde{u}\|+\|u_h-\Pi_h\tilde{u}\|. \tag{A.10}$$

The terms are shown geometrically in figure A.2. We try to approximate ϵ_{est} and to minimize the extrapolation error $\|u-\tilde{u}\|$. In the $\mathcal{H}^1$ norm the term $\|\tilde{u}-\Pi_h\tilde{u}\|$ dominates so that the second term may be neglected for $\mathcal{H}^1$ error estimates. To calculate the above defined errors, we determine first the error due to truncation only, assuming for the moment that $\tilde{u}=u$. In the one-dimensional case we write the solution as:

$$u_h(x)|_{\Omega^k}=\sum_{n=0}^{N}a_n^kL_n(r), \tag{A.11}$$

while the exact solution is written as:

$$u(x)|_{\Omega^k}=\sum_{n=0}^{\infty}a_n^kL_n(r). \tag{A.12}$$

On an element basis the error can be written as following:

$$(u-u_h)^k(x)=\sum_{n=N+1}^{\infty}a_n^kL_n(r),\in X_h^{\perp}, \tag{A.13}$$

where $X_h^{\perp}$ is the orthogonal complement space, defined as the space of functions for which the inner product with any function of the original space X_h is zero. Henceforth, the superscript k is dropped, assuming that the calculation is on an element basis. Since the norm of Legendre polynomials is defined by:

$$\|L_n\| = (\int\limits_{-1}^{1} L_n^2(r)dr)^{1/2} = \frac{2}{2n+1} \tag{A.14}$$

the expression for the a_n coefficients can be obtained with:

$$a_n = \frac{2n+1}{2}\int_{-1}^{1} u_h(x(r))L_n(r)dr. \tag{A.15}$$

These coefficients a_n form the Legendre spectrum of the solution u_h. Since, we need the $a_n^k, n = N+1, \ldots, \infty$ coefficients and they are not available to estimate the error truncation , we must *predict* them. They are given by extrapolation of the exact coefficients, a_n as $\tilde{a}_n, n \geq N+1$. We assume that the extrapolation scheme is adequate and we can estimate the error:

$$\|\tilde{u} - \Pi_h\tilde{u}\| = \left(\int\limits_{-1}^{1} \sum_{n=N+1}^{\infty} (\tilde{a}_n L_n(r))^2 dr\right)^{1/2}. \tag{A.16}$$

Based on the norm of Legendre polynomials we estimate the approximation error due to truncation as:

$$\|\tilde{u} - \Pi_h\tilde{u}\| = \left(\sum_{n=N+1}^{\infty} \frac{\tilde{a}_n^2}{\frac{2n+1}{2}}\right)^{1/2}, \tag{A.17}$$

which is independent of the element size since the coefficients $\tilde{a}_n$ are calculated in the computational space (r). For the two-dimensional case, u can be expanded in terms of Legendre polynomials:

$$u(x,y) = \sum_{n=0}^{\infty}\sum_{m=0}^{\infty} a_{n,m}L_n(x)L_m(y) \tag{A.18}$$

and the expansion coefficients are given by:

$$a_{nm} = \frac{(2n+1)(2m+1)}{2^2}\int\limits_{-1}^{1}\int\limits_{-1}^{1} u_h L_n(r)L_m(s)drds \;, m, n \in \{0, N\}^2. \tag{A.19}$$

In case of the $\mathcal{H}^1$ norm, we need also the derivative coefficients:

$$\tilde{a}_{nm}^i = \frac{(2n+1)(2m+1)}{2^2}\int\limits_{-1}^{1}\int\limits_{-1}^{1} u_h^i L_n(r)L_m(s)drds \tag{A.20}$$

where u_h^i terms are the derivatives of u:

$$u_h^i = \frac{\partial u_h}{\partial x_i},\ x_1,\ x_2\ =\ x,\ y.$$

The estimate of the approximation error due to truncation, for two-dimensional case, is therefore:

$$\|\tilde{u} - \Pi_h \tilde{u}\| = \left(\sum_{n=N+1}^{\infty} \sum_{m=M+1}^{\infty} \frac{(\tilde{a}_{nm}^i)^2}{\frac{(2n+1)(2m+1)}{2^2}} \right)^{1/2} . \tag{A.21}$$

Derivative coefficients are used for the $\mathcal{H}^1$ norm error estimate whereas the a_n coefficients are used for the $\mathcal{L}^2$ norm error estimate.

Next, we compute the approximation error due to quadrature and the fact that the solution is a constant away from the best polynomial fit. If we denote the exact solution as:

$$u = \sum_{n=0}^{N} a_n L_n(r), \tag{A.22}$$

and the discretized numerical solution as:

$$u_h = \sum_{n=0}^{N} c_n L_n(r), \tag{A.23}$$

then the error is computed by:

$$\|u_h - \Pi_h \tilde{u}\| = \left(\sum_{n=0}^{N} \frac{(c_n - a_n)^2}{\frac{2n+1}{2}} \right)^{1/2} . \tag{A.24}$$

Since the quadrature is exact for all polynomials of degree $\leq 2N-1$, and the a_n coefficients are exactly for $n \leq N-1$, and approximately for all $n \geq N$, the error reduces to:

$$\|u_h - \Pi_h \tilde{u}\| = \left(\frac{(c_N - a_N)^2}{\frac{2N+1}{2}} \right)^{1/2} \simeq \left(\frac{2c_N^2}{2N+1} \right)^{1/2} . \tag{A.25}$$

In two dimensions there are $N + M + 2$ terms and the error is computed by:

$$\|u_h - \Pi_h \tilde{u}\| \simeq \left(\sum_{j=0}^{M} \frac{2(c_{Nj})^2}{2M+1} + \sum_{i=0}^{N} \frac{2(c_{iM})^2}{2N+1} \right)^{1/2} . \tag{A.26}$$

The two error estimates (A.10) contributions have equal importance in the $\mathcal{L}^2$ norm, whereas in $\mathcal{H}^1$ norm $\|\tilde{u} - \Pi_h \tilde{u}\| >> \|u_h - \Pi_h \tilde{u}\|$. Therefore, Mavriplis proposes as a rigorous error estimate, an approximation to the two conditions, namely :

$$\epsilon_{est} = \left(\sum_{n=N}^{\infty} \frac{c_n^2}{\frac{2n+1}{2}} \right)^{1/2} , \tag{A.27}$$

which we approximate as:

$$\epsilon_{est} \simeq \left(\frac{c_N^2}{\frac{2N+1}{2}} + \sum_{n=N+1}^{\infty} \frac{\tilde{c}_n^2}{\frac{2n+1}{2}} \right)^{1/2} \tag{A.28}$$

in the one-dimensional case.

The two-dimensional case is similarly and the error is computed by:

$$\epsilon_{est} \simeq \left(\sum_{n=N}^{\infty} \sum_{n=M}^{\infty} \frac{c_{nm}^2}{\frac{(2n+1)(2m+1)}{2^2}} \right)^{1/2}, \tag{A.29}$$

which we approximate as:

$$\epsilon_{est} \simeq \left(\sum_{j=0}^{M} \frac{(c_{Nj}^2)}{\frac{2M+1}{2}} + \sum_{j=0}^{N} \frac{(c_{iM}^2)}{\frac{2N+1}{2}} + \sum_{n=N+1}^{\infty} \sum_{m=M+1}^{\infty} \frac{\tilde{c}_{nm}^2}{\frac{(2n+1)(2m+1)}{2^2}} \right)^{1/2}, \tag{A.30}$$

where c refers to the coefficients of the solution for the $\mathcal{L}^2$ error norm and those of the derivative for $\mathcal{H}^1$ norm.

Since we need to calculate the missing coefficients $\tilde{a}_n, n \geq N+1$, the sum of which forms the approximation error due to truncation, the next step is to find a good extrapolation method for the coefficients $\tilde{a}_n, n \leq N$. Babuška proposed a scheme to approximate the missing coefficients by actually calculating them on a finer mesh. Mavriplis proposed another approach, based on the least squares best fit to their decay to obtain $a_n, n \geq N+1$. The fit is qualified by a regression factor:

$$r^2 = \frac{S_t - S_r}{S_t}, \tag{A.31}$$

where $S_t = \sum_{i=0}^{N} (y_i - f_i)^2,\ S_r = \sum_{i=0}^{N} (y_i - \overline{y})^2$, y_i being the raw data, f_i the fitted data and $\overline{y}$ the average of the y_i.

Using a six point least square best fit to the $a(n)$ spectrum we solve for c_e and σ_e the exponential decay approximation:

$$a(n) \sim c_e e^{-\sigma_e n}, \tag{A.32}$$

which in a log-linear plot corresponds to a straight line since :

$$\frac{d}{dt}(\log(a(n))) \sim -\sigma_e \log n. \tag{A.33}$$

To fit the $\log a(n)$ to a straight line a linear regression is used:

$$f(n) = \sigma_e n + \log c_e. \tag{A.34}$$

As expected the fit to an exponential decay is better with a larger number of terms. To obtain reliable results, a number of 6 and higher terms is a better fit to an exponential decay than 3 to 5 terms (Kreyszig, 1993).

For the one-dimensional case, the extrapolation errors can safely be assumed to be negligible for smooth functions, provided the fit is adequate. For non-smooth functions, the spectrum of the Legendre polynomials decomposition is not necessarily exponentially decaying. At worst, we can expect an algebraically decaying function, for which we need to perform a fit. We solve for c_e and σ_e in the approximation to $a(n)$:

$$a(n) \sim c_e n^{-\sigma_e}. \tag{A.35}$$

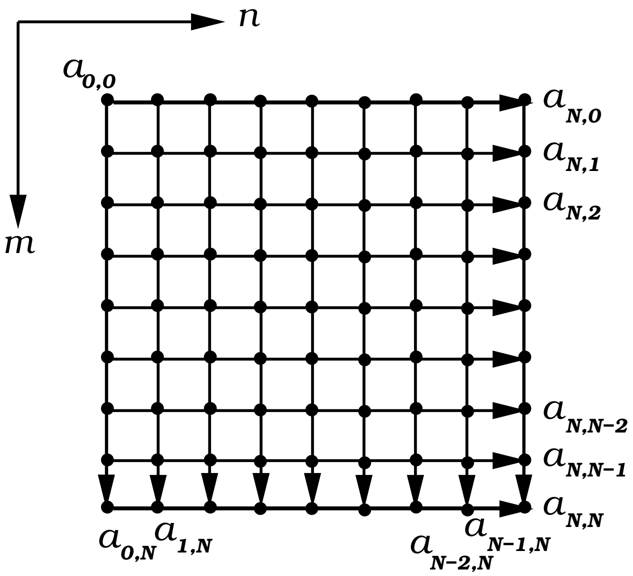


Fig. A.3: Two-dimensional spectrum of the Legendre discretization and "line-by-line" extrapolation procedure [Mavriplis (1990)].

In two dimensions, extrapolation becomes more complex. In this case we have a matrix with coefficients a_{nm}. We illustrate this in figure A.3. Using the same approach as in the one-dimensional case it does not produce a good extrapolation. Mavriplis concluded that we can minimize the extrapolation error $\|u-\tilde{u}\|$ by using line by line extrapolation (in the matrix a_{nm}), that is extrapolating $a^i_{nm},\ m = 0,\ N$ for each n fixed and vice-versa. We include, as usual, the last a^i_{nM} and a^i_{Nm} in the error estimate and integrate $\int_{N+1}^{\infty}(a^i_{nm})^2dn$, for each m fixed and vice-versa.

To summarize, the error estimates we use here are single mesh *a posteriori* local per element error estimates consisting of:

1. $\|\tilde{u} - \Pi_h \tilde{u}\|$ - the norm error due to truncation
2. $\|u_h - \Pi_h \tilde{u}\|$ - the error due to approximating the exact coefficients numerically by quadrature and the best polynomial approximation.

In the two-dimensional case the extrapolation error is minimized by performing line by line extrapolation of a_{nm} for n and m successively fixed.

Appendix B

C++ Wrappers

B.1 SEPRAN wrappers

The wrapping technique for SEPRAN calls is very simple. Each SEPRAN function has a corresponding C++ `inline` function definitions. The following we present some example of these definitions:

```
 // Raw SEPRAN prototypes
 extern "C" void start_ (int&, int&, int&, int&);
 extern "C" void presdf_(int*, const int *, int * );
 extern "C" void prestm_(int&, int*, int *, int *);
 extern "C" void commat_(int&, int*, int * , int *);
 extern "C" void mesh_  (int&, int*, double*, int * );
 extern "C" void filcof_(int*, double*, int* , int *, int&);
 extern "C" void build_ (int*, int*, int*, int*, int* ,int*,
                        int*, int*, int*, int*, double*);
 extern "C" void solve_ (int&, int *, int *, int *, int *, int *);
 ...

 // Define the SEPRAN wrapper class

class SepranCalls {

  public:

     static void start (int&, int&, int&, int&);
     static void presdf(int*, int *, int * );
     static void prestm(int&, int*, int *, int *);
     static void commat(int&, int*, int * , int *);
     static void mesh  (int&, int*, double*, int * );
     static void filcof(int*, double*, int* , int *, int&);
     static void build (int*, int*, int*, int*, int* ,int*,
                        int*, int*, int*, int*, double*);
     static void solve (int&, int *, int *, int *, int *, int *);
 ...

 };

// define the inline functions
```

```
inline void
    SepranCalls::start (int& jstart, int& i1, int& i2, int& i3) {
                  start_(jstart, i1, i2, i3);
}
inline void
    SepranCalls::presdf (int* kmesh, int* kprob, int* isol ) {
                  presdf_(  kmesh, kprob, isol );
}
...
```

B.2 BLAS and LAPACK wrappers

The wrapping technique used for SEPRAN calls, can also be implemented for LAPACK and BLAS. Here is an example of BLAS and LAPACK calls in C++ :

```
extern "C" double ddot_  (int& n, double *x, int& incx,
                                 double *y, int& incy);
extern "C" double dasum_ (int& n, double *x, int& incx);
extern "C" void   dgemv_ (const char *t, const int& m,
                          const int& n, const double& alpha,
                          double *a, const int& lda,
                          double *x, const int& incx,
                          const double& beta, double *y,
                          const int& incy);



...

// BLAS level 1
class Blas1Calls {
  public:
    static void  ddot  (int& n, double* x, int& incx,
                        double* y, int& incy);
    static void  dasum (int& n, double* x, int& incx);
    ...
};

inline float
   Blas1Calls::xdot(int n, double* x, int incx, double* y,
                  int incy){
    return ddot_(n, x, incx,y, incy);
}
...

// BLAS level 2
 class Blas2Subroutines :public Blas1Calls {
   public:
    enum Trans {no_T, T, Conj};
    static char T_char[];
```

```
    static void xgemv(
        Trans t,
        int m, int n, double alpha, const double* a, int lda,
        const double* x, int incx, double beta,
        double* y, int incy);

    static void xgemv(
        Trans t, int m, int n, float alpha, const float* a,
        int lda, const float* x, int incx, float beta,
        float* y, int incy);
   ...
}

// BLAS level 2 calls
inline void
    Blas2Calls::xgemv(Blas2Calls::Trans t, int m, int n,
               double alpha, const double* a, int lda,
               const double* x, int incx, double beta,
               double* y, int incy) {
     dgemv_(&trans_char[t], m, n, alpha, a, lda, x, incx,
             beta, y, incy);
}

// BLAS level 3 calls
class Blas3Calls :  public Blas2Calls {
 public:
    static char trans_char[];
    static void xgemm(
        Trans ta, Trans tb, int m, int n, int k,
        double alpha, const double* a, int lda,
        const double* b, int ldb, double beta,
        double* c, int ldc
    );
    static void xgemm(
        Trans ta, Trans tb, int m, int n, int k,
        float alpha, const float* a, int lda,
        const float* b, int ldb, float beta,
        float* c, int ldc
    );
    ...
};

// BLAS level 3 calls
inline void
    Blas3Calls::
        xgemm(Blas3Calls::Trans ta, Blas3Calls::Trans tb,
              int m, int n, int k, double alpha,
              const double* a, int lda, const double* b,
```

```
              int ldb, double beta, double* c, int ldc) {

            xgemm_(&trans_char[ta], &trans_char[tb], m, n, k,
                   alpha, a, lda, b, ldb, beta, c, ldc);
}
inline void
    Blas3Calls::
        xgemm(Blas3Calls::Trans ta, Blas3Calls::Trans tb,
              int m, int n, int k, float alpha,
              const float* a, int lda, const float* b,
              int ldb, float beta, float* c, int ldc) {

            xgemm_(&trans_char[ta], &trans_char[tb], m, n, k,
                   alpha, a, lda, b, ldb, beta, c, ldc);
}
...

// LAPACK subroutines
void sgetrf_(const int& M, const int& N, float A[],
              const int& LDA,  int IPIV[], int& INFO);
void dgetrf_ (const int& M, const int& N, double A[],
              const int& LDA, int IPIV[], int& INFO);

void sgetrs_ (const char TRANS[], const int& N,
              const int& NRHS, float A[], const int& LDA,
              const int IPIV[], float B[], const int& LDB,
              int& INFO);
...

// LAPACK class
class LapackCalls {

  public:
    // Factoring general matrices
    static void xgetrf (const int& M, const int& N, float* A,
                       const int& LDA, int* IPIV, int& INFO);

    static void xgetrf (const int& M, const int& N, double* A,
                        const int& LDA, int* IPIV, int& INFO);

    //Solving general factored matrices

    static void xgetrs (const char TRANS[], const int& N,
                        const int& NRHS, float* A,
                        const int& LDA, const int IPIV[],
                        float* B, const int& LDB,
                        int& INFO);

    static void xgetrs (const char TRANS[], const int& N,
```

```
                        const int& NRHS, double* A,
                        const int& LDA, const int* IPIV,
                        double* B, const int& LDB,
                        int& INFO);

    static void dgetrs (const char TRANS[], const int& N,
                        const int& NRHS, double A[],
                        const int& LDA, const int IPIV[],
                        double B[], const int& LDB, int& INFO);

// Factoring general matrices
inline
    void LapackCalls::
            xgetrf(const int& M, const int& N, float* A,
                 const int& LDA, int* IPIV, int& INFO) {
                    sgetrf_(M, N, A, LDA, IPIV, INFO);
}

inline
    void LapackCalls::
            xgetrf(const int& M, const int& N, double* A,
                  const int& LDA, int* IPIV, int& INFO) {
                dgetrf_(M, N, A, LDA, IPIV, INFO);
}


//Solving general factored matrices

inline
  void LapackSCalls::xgetrs(const char TRANS[], const int& N,
                  const int& NRHS, float* A, const int& LDA,
                  const int IPIV[], float* B, const int& LDB,
                  int& INFO) {

        sgetrs_(TRANS, N, NRHS, A, LDA, IPIV, B, LDB, INFO);
}

inline
  void LapackCalls::xgetrs(const char TRANS[], const int& N,
                 const int& NRHS, double* A, const int& LDA,
                 const int* IPIV, double* B, const int& LDB,
                 int& INFO) {

    dgetrs_(TRANS, N, NRHS, A, LDA, IPIV, B, LDB, INFO);
}
...

...
```

B.3 LAPACK, BLAS classes

Here we present an example, how we can group the LAPACK subroutines into a class, based on their functionality. The class template **FactoredLapackRect**<T> is defined for the factored matrices. For example, the factored form of a conventionally stored rectangular matrix might be represented by the following class:

```
template<class  T >
class FactoredLapackRect  {

  public:
    LapackRect <T>&  solve (LapackRect<T>&);

  private:
    friend FactoredLapackRect <T> LapackRect<T>::factor();

    FactoredLapackRect (FortranArray2d<T> *amatr);

    CopiedObjPtr <FortranArray2d<T>> fmatr; // Lapack A

    FortranArray1d <int> pivots; // Lapack ipiv - pivot
}
```

This implementation allows in-place factorization of a matrix while retaining full type checking. Class **FortranArray2d**<T> represents a Fortran array. Since **FactoredLapackRect**<T> does not behave like an ordinary matrix, it is not derived from **Array2**<T> (that implements an array template class). This is the reason that the constructor of the class is made `private`. The user will get access to it via the `friend` declaration and thus the **LapackRect**<T> controls when objects **FactoredLapackRect**<T> are created. With this scheme, the `factor()` member function would be called with an instance of an unfactored object and the `solve()` member function would be called with the factored object in order to solve the linear equations.

To store rectangular matrices, we can define the class **LapackRect** as:

```
template<class T >
class LapackRect:
  public  virtual  Array2d <T> {

    public:
      LapackRect (Subscript nrows, Subscript ncols);
      FactoredLapackRect <T> factor();  // factor matrix
      ...

      ...
      // Array  interface  declarations
    private:
      CopiedObjPtr <FortranArray2d<T>> amatr; // Lapack A
      Boolean valid ;
      // Check validity -- throw exception if not valid
```

```
      void CheckValidity() const ;
};
```

To wrap **BLAS** for use in C++, we use the same techniques as for **LAPACK**. First, we have to figure out what kind of algebraic structure to give it. The matrices with floating point elements form a division algebra under matrix addition, subtraction, multiplication, and inverse, with scalar multiplication by floating point numbers. Based on the **Abelian** (contains the user-must-define functions) and **Algebra** classes defined in (Barton and Nackman, 1994), the **BLAS** class can be defined as:

```
template <class T>
class ConcreteBlas:
    public Algebra <ConcreteBlas <T>, T>,
    public Abelian <ConcreteBlass <T>, T>,
    public FortranArray2d <T> {
  public:
    ConcreteBlas    (const  FortranArray2d <T> & a);
    ConcreteBlas    (Subscript nrows,  Subscript ncols);

    typedef  ConstBlasProjection1d <T> ConstProjectionT;
    typedef  BlasProjection1d <T> ProjectionT;

    // Algebra operations not implemented
    //  by Abelian class

    ConcreteBlas  <T>& operator*= (const T& rhs);
    ConcreteBlas <T>& operator*=
                         (const ConcreteBlas <T>& rhs);

    ConcreteBlas <T>& operator/= (const T& rhs);
    ConcreteBlas <T>& setToOne();

    ConcreteBlas <T>& operator=
                         (const ConcreteBlas<T>& rhs);
    ConcreteBlas <T>& operator= (const T& rhs);

    ConstProjectionT
        project ( Subscript i, Dimension d = 0) const ;
    ProjectionT
        project ( Subscript i, Dimension d = 0) const ;

    ConstProjectionT    operator[] ( Subscript i) const
                               { return project(i,0); }
    Projection          operator[] ( Subscript i)
                               { return project(i,0); }

    ConstProjectionT row (Subscript i) const
                               { return project(i,0); }
    ProjectionT      row (Subscript i)
                               { return project(i,0); }
```

```
        ConstProjectionT column (Subscript i) const
                                  { return project(i,1); }
        ProjectionT      column (Subscript i)
                                  { return project(i,1); }

        // Matrix-Vector (Blas level 2) operation
        friend ConcreteBlas1d <T>
          operator*( const ConcreteBlas <T>& m,
                             const ConcreteBlas1d <T>& v);

};
```

The constructors and assignment operators call the corresponding functions in the **FortranArray2d**<T> base; they are necessary because constructors and assignment operators are not inherited. The **ConcreteBlas1d**<T> class template represents vectors in a linear space compatible with **ConcreteBlas**<T>. The matrix-matrix operator `operator*=()` function can not be implemented as an in-place operation because the shape of the product matrix can be different from the shape of the original left-hand matrix. Thus we have to copy the left-hand side and adjust the size of the matrix before computing the product with a call to **BLAS**. The arithmetic computations are implemented by calling **BLAS** subroutines.

Bibliography

Adams, R. (1985). *Sobolev Spaces*. Academic Press, New York.

Adjerid, S., Flaherty, J., Moore, P., and Wang, Y. (1992). High-order adaptive methodes for parabolic systems. *Physica-D*, **111**, 60–94.

Anagnostou, G., Maday, Y., Mavripilis, C., and Patera, A. (1989). On the mortar element method: Generalizations and implementation. *Third International Symposium on Domain Decomposition Methods-SIAM*, **7**, 157.

Andersen, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., and Dongarra, J. (1999). *Lapack Users's Guide*. SIAM, Philadelphia, PA 19104-2688.

Anderson, D., McFadden, G., and Wheeler, A. (1998). *Annu. Rev. Fluid Mech.*, **30**, 139.

Armstrong, C. e. (1991). Advance in engng. software. *Computational Mechanics Publ., Southhampton*.

Babusks, I. and Dorr, M. (1981). Error estimates for the combined h and p versions of the finite element method. *Numerische Mathematik*, **25**, 257–277.

Baden, S., Fink, S., and Kohn, S. (1998). Efficient run-time support for irregular block-structured applications. *J. Parallel and Distributed Computing*, **April-May**, 61–82.

Barnard, S. and Simon, H. (1994). Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency:Practice and experience*, **6(2)**, 101–117.

Barrett, R. e. a. (1994). *Templates for the solution of linear systems: Building blocks for iterative methods.* http://www.netlib.org:templates.

Barton, J. and Nackman, L. (1994). *Scientific and Engineering C++*. Addison Wesley.

Berger, M. and Oliger, J. (1984). Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, **52**, 484–512.

Berger, M. and Saltzman, J. (1993). Structured adaptive mesh refinement on the connection machines. *SIAM*, **March**.

Bernardi, C., Maday, Y., and G., S. L. (1990). Non-conforming matching conditions for coupling spectral and finite element methods. *Applied Numerical Mathematics*, **6**, 65–84.

Bernardi, C., Maday, Y., and Patera, A. (1994). *A new nonconforming approach to domain decomposition: the mortar element method.* In Non-linear Partial Differential Equations and their Applications, vol. 11, Pitman/Wiley:London/New York.

Bose, A. and Carey, G. (1999). A class of data structures and object-oriented implementation for finite element method on distributed memory systems. *CMAME*, **171**, 109–121.

Cahn, J. and Hilliard, J. (1958). *J. Chem. Phys.*, **28**, 258.

Canuto, C., Hussaini, M., Quarteroni, A., and Zang, T. (1988). *Spectral methods in fluid dynamics*. Springer−Verlag, New York, Berlin.

Chan, T. and Mathew, T. (1994). *Domain Decomposition Algorithms, Acta Numerica, pages 61-143*. Acta Numerica.

Ciarlet, P. (1978). *The Finite Element Method*. North Holland.

Clark, K., Flaherty, J., and Shephard, M. (1994). Special ed. on adaptive methods for partial differential equations. *Appl. Numeric. Math.*, **14**.

Czarnecky, K., Eisenecker, U., Gluck, R., Vandevoorde, D., and Veldhuizen, T. (1998). *Generative programming and active libraries.* Lecture Notes in Computer Science, Dagstuhl-Seminar on Generic Programming.

Davis, H. and Scriven, L. (1982). *Adv. Chem. Phys.*, **49**, 358.

Davis, P. and Rabinowitz, P. (1984). *Methods of Numerical Integration*. Academic Press, Inc.

Deville, M., Fischer, P., and Mund, E. (2002). *High-Order Methods for Incompressible Fluid Flow*. Cambridge University Press, Cambridge.

Dongarra, J., Du Croz, J., Hammarling, S., and Duff, I. (1988). An extended set of fortran basic linear algebra subprograms. *ACM Tran. Math Software*, **14(1)**, 1–17.

Dongarra, J., Du Croz, J., Hammarling, S., and Duff, I. (1990). An extended set of level 3 basic linear algebra subprograms. *ACM Tran. Math Software*, **16(1)**, 1–17.

Dors, M. (1989). Domain decomposition via lagrange multipliers. *Numerische Mathematik*, **65**, 120–145.

Edwards, H. and Browne, J. (1998). Scalable distributed dynamic array (sdda) and its application to a distributed adaptive mesh data structure. *TICAM Repor*, **98-04**.

Farhat, C. and Lesoinne, M. (1993). Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics. *Int. J. Numer. Meth. Engng.*, **36**, 745–764.

Forum, C. (1995). *Working paper for draft proposed international standard for information systems- programming language C++*. American National Standard Institute.

Gervasion, L. (1998). Octree load balancing techniques for the dynamic load balancing library. *Master's thesis, Computer Science Dept., Rensselaer Polytechnic Institute, Troy, NY.*

Gottlieb, D. and Orszag, S. (1977). *Numerical Analysis of Spectral Methods*. SIAM, Philadelphia.

Greengard, L. and Lee, J. (1996). A direct adaptive poisson solver of arbitrary order accuracy. *J. Comput. Phys.*, **125**, 415–424.

Gunton, D., Miguel, M., and Sahni, P. (1983). *The dynamics of first-order phase transitions, vol8. of Phase transitions and critical phenomena*. Academic Press, London.

Henderson, R. (1994). *Unstructured Spectral Element Methods: Parallel Algorithms and Simulations*. PhD thesis, Princeton University.

Henderson, R. and Karniadakis, G. (1991). Hybrid spectral element-low order methods for incompressible flows. *Journal of Scientific Computing*, **6**, 2–79.

Hendrickson, B. and Leland, R. (1993). The chaco user's guid, version 1.0. *Tech. Report SAND93-2339, Sandia National Laboratories, Albuquerque*.

Kågström, B., Ling, P., and Van Loan, C. (1998a). GEMM-Based Level 3 BLAS: High-Performance Model Implementations and Performance Evaluation Benchmark. *ACM Trans. Math. Software*, **24**(3), 268–302.

Kågström, B., Ling, P., and Van Loan, C. (1998b). Algorithm 784: GEMM-Based Level 3 BLAS: Portability and Optimization Issues. *ACM Trans. Math. Software*, **24**(3), 303–316.

Keestra, B., Van Puyvelde, J., Anderson, P., and Meijer, H. (2003). Diffuse interface modelling of the morphology and rheology of immiscible polymer blends. *Physics of Fluids*, **15(9)**, 2567–2575.

Kikuchi, R. and Cahn, J. (1962). Theory of domain walls in ordered structures ii. pair approximation for nonzero temperatures. *J. Phys. Chem. Solids*, **23**, 137–151.

Kreyszig, E. (1993). *Advanced Engineering Mathematics, Seventh edition*. Wiley, New York.

Lawson, C., Hanson, R., Kincaid, D., and Krogh, F. (1979). Basic linear algebra subprograms for fortran usage. *ACM Tran. Math Software*, **5(3)**, 308–323.

Levin, J., Iskandarani, M., and Haidvogel, B. (2000). A nonconforming spectral element ocean model. *IBM*, **34**, 495–525.

Lowengrub, J., Goodman, J., Lee, H., Longmire, E., Shelley, M., and Truskinovsky, L. (1998). *Proceedings of the 1997 International Congress on Free Boundary Problems*. Addison-Wesley Longman, Reading Massachusetts.

Maday, Y. and Patera, A. (1988). *Spectral Element methods for Navier-Stokes equations*. State-of-the-art surveys in computational mechanics, ASME, New York.

Maday, Y., Mavriplis, C., and Patera, A. (1989). Non-conforming mortar element methods: application to spectral discretization. *SIAM*, **7**, 392–418.

Maday, Y., Patera, A., and Rønquist, E. (87). Optimal legendre spectral element methods for the stokes semi-periodic problem. *ICASE report*, **48**.

Mavriplis, C. (1989). *Nonconforming Discretizations and a Posteriori Error Estimates for Adaptive Spectral Element Techniques*. PhD thesis, MIT.

Mavriplis, C. (1990). A posteriori error estimators for adaptive spectral element techniques. *Notes on Numerical Fluid Mechanics*, **39**, 333–342.

MPI (1994). *Message Passing Interface Forum*. Knoxville, Tennessee, Knoxville, Tennessee.

Musser, D. and Stepanov, A. (1994). Algorithm-oriented generic libraries. *Software: Practice and Experience*, **24**, 632–642.

Naumann, E. and He, D. (2001). *Chem. Eng. Sci.*, **56**, 1999.

Parashar, M. and Browne, J. (1995). Distributed dynamic data-structures for parallel adaptive mesh refinement. *Proceeding of the International Conference for High Performance Computing*.

Patera, A. (1984). A spectral element method for fluid dynamics: laminar flow in a channel expansion. *Journal of Comp. Physics*, **54**, 468.

Patra, A. and Oden, J. (1995). Problem decomposition for adaptive hp finite element methods. *Comp. SyS. Engng.*, **6**, 97.

Rønquist, E. (1988). *Optimal Spectral Element Methods for the Unsteady Three-Dimensional Incompressible Navier-Stokes*. PhD thesis, Massachusetts Institute of Technology.

Rowlinson, J. and Widom, B. (1989). *Molecular Theory of Capillarity*. Clarendon, Oxford.

Rumbauch, J., Jacobson, I., and Booch, G. (1998). *The Unified Modelling Language Reference Manual*. Addison Wesley, Reading, Massachusetts.

Saad, Y. (1995). *Iterative Methods for Sparse Linear Systems*. PWS, Boston.

Schwab, C. (1998). *$p-$ and hp-FEM*. Oxford University Press.

Segal, A. (1995). *Sepran user manual and programmers guide*. Ingenieursbureau Sepra, Leidschendam.

Shephard, M. (1988a). Approaches to the automatic generation and control of finite element methods. *Applied Mechanics Reviews*, **41(4)**, 169–185.

Shephard, M. (1988b). Update to:approaches to the automatic generation and control of finite element methods. *Applied Mechanics Reviews*, **49(10)**, 5–14.

Shephard, M., Flaherty, J., de Cougny, H., Ozturan, C., Bottasso, C., and Befall, M. (1995). Parallel automated adaptive procedures for unstructured meshes. *Parallel Comput. in CFD.*, **R-807**, 6.1–6.49.

Siek, J. and Lumsdaine, A. (1998). *A rational approach to portable high performance: The basic linear algebra instruction set (BLAIS) and the fixed algorithm site template (FAST) library.* Parallel Object Oriented Scientific Computing, ECOOP.

Siek, J. and Lumsdaine, A. (1999). A modern framework for portable high-performance numerical linear algebra. *Advances in Software Tools for Scientific Computing;Springer−Verlag*, pages 1–56.

Sohn, A., Biswaas, R., and Simon, H. (1996). Impact of load balancing on unstructured adaptive computations for distributed-memory multiprocessors. *In Proc. Eighth IEEE Symp. on Parallel and Distrib. Proc., New Orleans, LA.*

Stepanov, A. (1996). *Generic programming.* Lecture Notes in Computer Science,1181.

Stroustrup, B. (1997). *C++*. Addison-Wesley, Inc.

Timothy, B. (1998). *Data Structures In C++*. Addison-Wesley, Inc.

Tucker, C. and Moldenaers, P. (2002). *Annu. Rev. Fluid Mech.*, **34**, 177.

Veldhuizen, T. (1995). Expression templates. *C++ Report*, **7**, 36–43.

Veldhuizen, T. (1998). Arrays in blitz++. *ISCOPE'98*, **1505**.

Veldhuizen, T. (1999). Blitz++: The library that think it is a compiler. *Advances in Software Tools for Scientific Computing;Springer−Verlag*, pages 57–89.

Verschueren, M. (1999). *A diffuse-Interface Model for Structure Development in Flow*. PhD thesis, Eindhoven University of Technology, the Netherlands.

Vreugdenhil, C. and Koren, B. (1993). *Numerical Methods for Advection-Diffusion Problems*. Vieweg Braunschweig, Wiesbaden.

Williams, R. (1992). Voxel databases: A paradigm for parallelism with spatial structure. *Concurrency*, **4**, 619–636.

Summary

In this thesis, we investigate the implementation of an adaptive mesh technique based on the mortar element method. A common bottleneck in the implementation of a numerical tool-box is the complexity of the code that has to be developed. The procedural programming approach demands a lot of programming effort and is a time consuming task. To avoid such problems, an object-oriented approach model is proposed and tested on a series of problems in one and two dimensions. Based on a set of C++ classes and Fortran libraries, the implementation offers an efficient software interface to an application code. Because the strength of C++ lies in new kind of features that assist in formulating more complex programs, we can build a framework for more sophisticated and reliable programs.

The mortar discretization is applied here to the spectral element method. It allows local mesh refinement, which simplifies grid generations for a problem in a complicated domain. Also, the mortar discretization represents a significant advance for spectral element methods , which offers new possibilities to time-dependent moving boundary problems. One of the major advantages of unstructured meshes is the ability to adapt the mesh to improve resolution at a place in the simulation which needs it. The error estimators provide powerful information to be used directly in an adaptive refinement scheme. Together with the mortar discretization, they form the basic components for a fully adaptive mesh refinement environment.

To provide a clear and efficient way to program a large variety of mesh computations in Fortran or C++, the Voxel Data Base (**VDB**) has been implemented, which makes the execution of a program independent of the distribution of data to processors. This allows an application to read in a mesh of just a few elements, and adapt it to the necessary resolution in parallel and, furthermore, adapt it locally once the simulation demands more or less resolution based on the values of the error estimators.

Several examples were presented that show that the convergence rates are similar in the uniform and non-conforming cases. However, in the non-conforming case, we use less elements (points) than in the uniform case, because the mortars allow local mesh refinement in the regions where a good accuracy is needed. Overall, the non-conforming discretization proves to be a flexible and a reliable method that eliminates the limitations of the standard conforming spectral discretizations.

Samenvatting

Dit proefschrift beschrijft de implementatie van een numeriek gereedschap voor *adaptive mesh* technieken, gebaseerd op de mortar element methode. Een veel voorkomende bottleneck in de implementatie van numerieke gereedschap is de complexiteit van de code die ontwikkeld moet worden. Een procedurele benadering van programmeren eist veel inspanning en kost veel tijd. Om deze problemen te vermijden, is een object-georiënteerde benadering gevolgd. Gebaseerd op een aantal C++ klassen en Fortranbibliotheken, biedt de implementatie een efficiënte software interface aan de applicatiecode. C++ kent veel object-georiënteerde mogelijkheden, die we kunnen gebruiken om complexe en ingewikkelde programma's te ontwikkelen. Mortar-discretisatie is hier toegepast voor de spectrale elementen methode. Het maakt locale roosterverfijning mogelijk waardoor de roostervorming voor een probleem in een ingewikkeld domein sterkt vereenvoudigt.

Mortar-discretisatie betekent een waardevolle aanwinst voor de spectrale elementen methode. Gegeven de nieuwe mogelijkheden voor tijdsafhankelijke problemen met bewegende grenzen, biedt de mortar-discretisatie een waardevolle bijdrage aan de spectrale element methoden. Een van de voornaamste voordelen van non-conforming roosters is de mogelijkheid het rooster aan te passen, op een plek in de simulatie, waar een hogere resolutie nodig is. De foutschatters leveren krachtige informatie die direct gebruikt kan worden bij deze adaptieve verfijning. Samen met de mortar-discretisatie, vormen ze de basiscomponenten voor een volledige omgeving voor adaptieve roosterverfijning.

Om een grote verscheidenheid aan roosterberekeningen in Fortran of C++ op een duidelijke en doeltreffende wijze te kunnen leveren, is de Voxel Data Base (**VDB**) gëimplementeerd. Dit maakt de werking van het programma onafhankelijk van de dataverdeling tussen de processoren. Hierdoor kan een applicatie, na slechts een aantal elementen te hebben gelezen, deze parallel aanpassen aan de juiste resolutie en ze daarna zo nodig lokaal aanpassen, gebaseerd op de foutschatters.

Er zijn enkele voorbeelden gepresenteerd die laten zien dat de convergentiesnelheden gelijk zijn in conforming en non-conforming gevallen. Echter, we gebruiken in het non-conforming geval minder elementen dan in het conforming geval, omdat de mortars lokale roosterverfijning toestaan in gebieden waar betere nauwkeurigheid vereist is. In het algemeen blijkt de non-conforming discretisatie een soepele en betrouwbare methode te zijn, die beperkingen van de standaard conforming discretisaties elimineert.

Acknowledgements

This thesis would not have been possible without the help of many people. To all of them who contributed to this work, I express my sincere gratitude. First and foremost, I would like to thank Frans van de Vosse, Han Meijer and Jan Dijkstra. Frans, thank you for introducing me to the MATE group and the beautiful world of spectral elements. Your support in the first years of my study was of great importance. Han, the freedom, support and understanding you gave me, were crucial to the continuity of my work. I am deeply grateful and thankful to you. Jan, thank you for the opportunity you offered me to continue with my study and the enjoyable atmosphere you have created in ICTOO group.

I would like to thank Patrick Anderson, for his support and guidance. His expertise in SEPRAN and spectral elements were very helpful to me.

I wish to thank Marleen Rieken for her kindness, understanding and positive thinking. She had all the time, when I was in trouble, a good advise for me.

Many thanks to my room mates Ivan, Erwan, Juan and Christian, who have tried to create all the time a very inspiring and enjoyable atmosphere. Our room was unique, we had the highest number of citizenships per square meter from the university. Ivan thank you for the fruitful discussions over the numerical implementation of the mortar methods and for remembering me the balkan way of life, I was missing so much. Bert and Viny, many thanks for the applications you offered me.

Several people, I am indebted to: Prof. M. O. Deville, Prof C. Mavriplis, Prof. G. Karniadakis, dr. R. D. Henderson for helping me to understand the world of high-order methods and offered me support in the implementation of the mortar method.

I thank my colleagues from ICTOO and Computer Graphics group for all the enjoyable moments we spent together. Thieu thank you for your advise and for solving a lot problems I was confronted to in these years.

To my friends Joshua, Michael, Rafael, Gabriel, Ouriel and Daskalos: thank you for being with me all the time, and for the guidance you give me in life.

I would like to express here my warmest thanks to my family for support and understanding during my work on this thesis. Dear Diana, I will not be able to thank you ever for the way you have supported me through these heavy years. You have been all the time, not only my loving wife but also, the third promotor. Your remarks were very valuable to me, even when they were very direct and critical.

To my children Alina, Julia and Stefan: I promise you that I will be very early at home, each day, and every weekends will be only a family matter.

Least but not last, I would like to thank my parents Ilinca, Simona, Marin and Ioan. Oriunde va aflati acum, va transmit toata dragostea mea. Va multumesc pentru sacrificiile pe care le-ati facut in viata, pentru iubirea si dragostea cu care m-ati inconjurat.

Ion Barosan,
Eindhoven, September 5, 2003.

Curriculum Vitae

Jun. 27, 1961	Born in Galateni, Romania .
1968-1976	Primary school education, Galateni.
1977-1980	Secondary school education, Bucharest.
1981-1986	”Politechnica” University Bucharest, Faculty of Computer Science .
1986-1989	Software Engineer, ”6 Martie” Company Zarnesti, Romania.
1989-1990	Software Engineer - researcher associate at the Automation Technological Engineering and Scientific Research Institute - Bucharest, Romania.
1991- *	Visualization and Virtual Reality Specialist, Eindhoven University of Technology.