ABSTRACT OF DISSERTATION

Daniel Thomas Thorne Jr.

The Graduate School

University of Kentucky

Dec 02, 2003

Multigrid with Cache Optimizations on Adaptive Mesh Refinement Hierarchies

ABSTRACT OF DISSERTATION

A dissertation submitted in partial fulfillment of the
requirements of the degree of dissertation
at the University of Kentucky

By

Daniel Thomas Thorne Jr.

Lexington, Kentucky

Director: Prof. Craig C. Douglas, Computer Science Department

Lexington, Kentucky

Dec 02, 2003

ABSTRACT OF DISSERTATION

Multigrid with Cache Optimizations on Adaptive Mesh Refinement Hierarchies

This dissertation presents a multilevel algorithm to solve constant and variable coefficient elliptic boundary value problems on adaptively refined structured meshes in 2D and 3D. Cache aware algorithms for optimizing the operations to exploit the cache memory subsystem are shown.

(Daniel Thomas Thorne Jr.)

(Date)

Multigrid with Cache Optimizations on Adaptive Mesh Refinement Hierarchies

By

Daniel Thomas Thorne Jr.

(Dissertation Director)

(Director of Graduate Studies)

(Date)

RULES FOR THE USE OF DISSERTATIONS

Unpublished dissertations submitted for the Master's and Doctor's degrees and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgments.

Extensive copying or publication of the dissertation in whole or in part requires also the consent of the Dean of the Graduate School of the University of Kentucky.

DISSERTATION

Daniel Thomas Thorne Jr.

The Graduate School

University of Kentucky

Dec 02, 2003

Multigrid with Cache Optimizations on Adaptive Mesh Refinement Hierarchies

DISSERTATION

A dissertation submitted in partial fulfillment of the requirements of the degree of

Ph.D.at the University of Kentucky By

Daniel Thomas Thorne Jr.

Lexington, Kentucky

Director: Prof. Craig C. Douglas, Computer Science Department

Lexington, Kentucky

Dec 02, 2003

Contents

Acknowledgments **iii**

List of Tables **vii**

List of Figures **iv**

1 Introduction **19**

1.1 Summary of Dissertation 19

1.2 Summary of Symbols 21

2 Background **23**

2.1 Grids 23

2.2 Discretization of Elliptic PDEs 23

2.2.1 1D 24

2.2.2 2D 25

2.2.3 3D 25

2.3 Iterative Solvers 26

2.4 Multigrid 27

2.4.1 Gauss-Seidel as Smoother 27

2.4.2 Transfer Operators 27

2.4.3 The V-Cycle 27

2.5 AMR 28

3 Tools for 2D AMRMG **31**

3.1 Tools For Constant Coefficient Problems 31

3.1.1 2D Stencils 31

3.1.2 Interpolation of Ghost Points in 2D 33

3.1.3 Flux Matching in 2D 36

3.2 Tools For Variable Coefficient Problems 38

3.2.1 Stencils 38

3.2.2 Interpolation of Ghost Points in 2D 40

3.2.3 Flux Matching 40

4 Analysis of Flux Matching **43**
4.1 Stencil Version, Using Ghost Coarse Grid Point. 43
4.2 Stencil Version, Averaging Two Coarse Grid Sized Fluxes 46
4.3 Flux Matching Version 47
4.4 Computing Flux Across Interface Directly Via Taylor's Series 48
4.4.1 Taylor Series Expansion With Three Points 49
4.4.2 Taylor Series Expansion With Five Points 50
4.4.3 Taylor Series Expansion With Seven Points 51
4.4.4 Example 53

5 Tools for 3D AMRMG **59**
5.1 Tools For Constant Coefficient Problems 59
5.1.1 3D Stencils 59
5.1.2 Interpolation of Ghost Points in 3D 60
5.1.3 Flux Matching in 3D 61
5.2 Tools For Variable Coefficient Problems 62
5.2.1 3D Stencils 62
5.2.2 Interpolation of Ghost Points in 3D 66
5.2.3 Flux Matching in 3D 67

6 Ghost Point Interpolation Revisited **75**
6.1 2D 75
6.1.1 Phase One (Intermediate) Ghost Points 75
6.1.2 Phase Two Ghost Points 77
6.2 3D 78
6.2.1 Phase One and Phase Two (Intermediate) Ghost Points 78
6.2.2 Phase Three Ghost Points 83

7 AMRMG **87**
7.1 Review of Notation 87
7.2 Comparison With Standard Multigrid Algorithm 87
7.3 The AMR Multigrid Algorithm 89
7.4 Post-Smoothing Only 93

8 Cache Optimizations **97**
8.1 Cache Aware Gauss-Seidel 97
8.2 Cache-Aware V-Cycle 98
8.2.1 Combined smoother 98
8.2.2 Effects of the coefficient matrix 98
8.3 Details 99
8.3.1 2D 100
8.3.2 3D 100
8.3.3 Update Pattern for Combined Smoother 102

9 Numerical Results **107**
9.1 Constant . . . 107
9.2 Variable . . . 110

10 Conclusions and Future Directions **113**

Appendix A **A–124**
.1 Introduction . . . A–124
.2 Interior Damping Factor . . . A–124
.3 Edge Damping Factor . . . A–124
.4 Corner Damping Factor . . . A–125
.5 Comments . . . A–125

Appendix B **B–126**
.1 Grids . . . B–126
.1.1 Grid Class . . . B–127
.1.2 Grid Level Class . . . B–129
.1.3 Grid Hierarchy Class . . . B–131
.2 Grid Functions . . . B–132
.2.1 Grid Function Class . . . B–133
.2.2 Grid Function Array Class . . . B–134
.2.3 Grid Function Level Class . . . B–148
.2.4 Grid Function Composite Class . . . B–151
.3 AMRMG Class . . . B–153
.4 Supporting Classes . . . B–155
.4.1 Coords Class . . . B–155
.4.2 Array Class . . . B–156

List of Tables

4.1 Combinations for expansions of points in Fig. 4.3 50

6.1 Coarse grid point weights for the first phase two ghost point. 81
6.2 Coarse grid point weights for the second phase two ghost point. 81
6.3 Coarse grid point weights for the third phase two ghost point. 83
6.4 Coarse grid point weights for the fourth phase two ghost point. 83

9.1 Itanium 1, Standard(0,4) Versus CA(0,4) and CAMG(0,4) 108
9.2 Itanium 2, Standard(0,4) Versus CA(0,4) and CAMG(0,4) 108
9.3 Pentium III, Standard(0,4) Versus CA(0,4) and CAMG(0,4) 109
9.4 Pentium IV, Standard(0,4) Versus CA(0,4) and CAMG(0,4) 109
9.5 CA(2,2) Versus CA(0,4) 109
9.6 Standard(2,2) Versus CAMG(0,4) 109
9.7 Itanium 1 speedups versus standard multigrid. 110
9.8 Itanium 2 speedups versus standard multigrid. 111
9.9 Pentium III speedups versus standard multigrid. 111
9.10 Pentium IV speedups versus standard multigrid. 111
9.11 Speedups for the cache aware smoother alone. 112

List of Figures

2.1 Grids. 24
2.2 Transfer Operators, P_ℓ and R_ℓ. 27
2.3 V-Cycle. 28
2.4 A simple patch based grid hierarchy. 30
2.5 Composite grid hierarchy corresponding to the patch hierarchy in Fig 2.4. 30

3.1 A sample 2D grid hierarchy. The dark lines denote the boundary. 32
3.2 Stencils. The dark lines in (b) and (c) represent boundaries. 32
3.3 Interpolation of Ghost Points. The dark edge in (c) indicates a boundary. . 34
3.4 2D flux matching with one coarse-fine interface. 36
3.5 2D flux matching with two coarse-fine interfaces. 37
3.6 2D flux matching adjacent to a boundary. 37
3.7 A sample 2D grid hierarchy. The dark lines denote the boundary. 38
3.8 Stencils. The dark lines in (b) and (c) represent boundaries. 39
3.9 Flux Matching. The dark edge in (c) represents a boundary. 41

4.1 Flux matching 44
4.2 Interface, Small 49
4.3 Interface, Medium 50
4.4 Interface, Large 52
4.5 Error in the ghost point interpolation. 54
4.6 Error in the fluxes. The red line is for the flux matching procedure, and the green line is for the small Taylor series expansion. The true fluxes are plotted in black in the two lower plots, but it is mostly covered by the flux matching approximation. 55
4.7 Error in the operator. The red line is for the flux matching based approximation to the operator, and the green line is for the small Taylor series expansion base approximation to the operator. The true values of the operator are plotted in black in the two lower plots, and they coincide very closely with the red line as in the flux plot. 56
4.8 Error in the fluxes as grid size increases. The horizontal axis is the grid size measured on the coarse grid and ranges over coarse grids of size 8×8 to $2^{10} \times 2^{10}$ with the dimensions doubling at each step. 57

4.9 Error in the operator as grid size increases. The horizontal axis is the grid size measured on the coarse grid and ranges over coarse grids of size 8×8 to $2^{10} \times 2^{10}$ with the dimensions doubling at each step. 58

5.1 3D Interpolation of Ghost Points, Step 1. 60
5.2 3D Interpolation of Ghost Points, Step 2. 61
5.3 Flux Matching in 3D. The coarse grid point where the operator is being applied is shown for reference. 62
5.4 Integration regions for variable coefficient stencil compution for interior cells. These are the boundaries of the control volume. 63
5.5 Integration regions for variable coefficient stencil compution for cells on the east face. These are the boundaries of the control volume. 64
5.6 Integration regions for variable coefficient stencil compution for cells on the UE edge. These are the boundaries of the control volume. 65
5.7 Integration regions for variable coefficient stencil compution for cells on the UNE corner. These are the boundaries of the control volume. 66
5.8 Integration regions for an interior cell with one coarse-fine interface (at the "west" cell wall) where flux matching is required. The little circles indicate the locations of the variable coefficient values. The labels α, β, γ, and δ are used to specify the corresponding ghost points used in the flux matching formula. 67
5.9 Integration regions for an interior cell with two coarse-fine interfaces (at the "west" and "south" cell walls) where flux matching is required. The little circles indicate the locations of the variable coefficient values. The labels α, β, γ, and δ are used to specify the corresponding ghost points used in the flux matching formula. 68
5.10 Integration regions for an interior cell with three coarse-fine interfaces (at the "west", "south", and "down" cell walls) where flux matching is required. The little circles indicate the locations of the variable coefficient values. The labels α, β, γ, and δ are used to specify the corresponding ghost points used in the flux matching formula. 69
5.11 Integration regions at one boundary (the "north" boundary) with one coarse-fine interface (at the "west" cell wall) where flux matching is required. The dashed lines indicate the location of the boundary. The little circles indicate the locations of the variable coefficient values. The labels α, β, γ, and δ are used to specify the corresponding ghost points used in the flux matching formula. 70
5.12 Integration regions at one boundary (the "north" boundary) with two coarse-fine interfaces (at the "west" cell wall and the "up" cell wall) where flux matching is required. The dashed lines indicate the location of the boundary. The little circles indicate the locations of the variable coefficient values. The labels α, β, γ, and δ are used to specify the corresponding ghost points used in the flux matching formula. 72

5.13 Integration regions at two boundaries (the "up" boundary and the "north" boundary) with one coarse-fine interface (at the "west" cell wall) where flux matching is required. The dashed lines indicate the location of the boundary. The little circles indicate the locations of the variable coefficient values. The labels α, β, γ, and δ are used to specify the corresponding ghost points used in the flux matching formula. 73

6.1 Phase one ghost points 76
6.2 Phase two ghost points 77
6.3 Phase one ghost points. 79
6.4 Phase two ghost points. 80
6.5 Template for interpolation of phase one ghost points. 81
6.6 Interpolation of the first phase two ghost point. 82
6.7 Template for interpolation of phase three ghost points, $i = 1..4$. Same as in Fig. 6.2 of the 2D case. 83

7.1 Illustration of $\Lambda_c^{\ell_{max}} - \mathcal{P}(\Lambda^{\ell+1})$. 90
7.2 Situation on Λ_c^3 before computation of the composite grid residual on Λ_c^2. The rest of the composite grids from Fig. 2.5 are shown for reference. . . . 91
7.3 The $\mathcal{L}^\ell$ operator shown for $1 \leq \ell \leq 3$. The shading indicates the computational domain $\Lambda^\ell - \mathcal{P}(\Lambda^{\ell+1})$. The hash marks denote the boundary. 91
7.4 The $\mathcal{L}^{nf,\ell}$ operator shown for $1 \leq \ell \leq 3$. The shading indicates the computational domain Λ^ℓ. The hash marks denote the boundary. 92
7.5 Illustration of $\Lambda^\ell - \mathcal{P}(\Lambda^{\ell+1})$. 93

8.1 2D Cache Block 100
8.2 2D Cache Block Seam 101
8.3 2D Cache Block Seams, Full 20×20 Patch Picture With Representative Cache Block. 101
8.4 2D Tall Cache Block 102
8.5 2D Tall Cache Block Seams, Full 20×20 Patch Picture With Representative Cache Block. 103
8.6 3D Cache Block 103
8.7 3D Cache Block Update Pattern. This shows xy-slices of the 3D cache block, starting in the negative direction (bottom of the cache block) and going toward positive (top of the cache block). 104
8.8 Template for interleaving ghost point interpolation and residual computation with cache aware smoother in 2D. The residual updates are denoted by the dark circles. The potential ghost points are denoted by the dashed boxes. . 104

9.1 Refinement patterns: (a) One refinement per patch. (b) Two refinements per patch. (c) Four refinements per patch. 107
9.2 (a) One refinement per patch. (b) Two refinements per patch. (c) Four refinements per patch. 110

1 C++ Class Hierarchy B–126

Chapter 1

Introduction

1.1 Summary of Dissertation

This document presents a combination of adaptive refinement [11, 12, 87, 92] and multilevel [21, 70, 73] procedures to solve variable coefficient elliptic boundary value problems of the form

$$\begin{cases} \mathcal{L}(\phi) = \rho \text{ in } \Omega, \\ \mathcal{B}(\phi) = \gamma \text{ on } \partial\Omega, \end{cases} \tag{1.1}$$

subject to standard conditions that ensure ellipticity and well posedness [3]. The solution procedure is derived from the adaptive mesh refinement process, not from the multigrid procedure. Hence, notation is borrowed from the *adaptive mesh refinement* (AMR) community. For some related work in multigrid and adaptive mesh refinement, see [14, 5, 6, 7, 8, 9, 10, 16, 15, 17, 18, 19, 20, 22, 25, 39, 41, 42, 43, 45, 46, 47, 48, 49, 52, 54, 64, 63, 65, 66, 67, 68, 69, 71, 72, 74, 75, 76, 77, 78, 85, 86, 88, 89, 95] .

Since the early 1980's, processors have sped up 5 times faster per year than memory. Multilevel memories, using *memory caches*, were developed to compensate for the uneven speedups in hardware. Essentially all computers today, from laptops to distributed memory supercomputers, use cache memories in an attempt to keep the processors busy. By the term *cache*, we mean a fast memory unit closely coupled to the processor [44, 81]. In the interesting cases, the cache is further divided into many *cache lines*. Each cache line holds copies of contiguous locations of main memory. Any given pair of cache lines may hold data from entirely separate regions of main memory. A good cache primer for solving PDEs can be found in [28, 30].

The focus of this research is on the effects of optimizations designed to minimize the number of times data goes through cache. Algorithms optimized in this way are called *cache aware*. Cache aware algorithms should be more efficient because cache memory is much faster than main memory, so the CPU can be kept more busy when it is getting data from cache memories. For some related work in cache aware algorithms, see [33, 35, 37, 50, 90, 93, 94, 91, 34, 99, 38, 32, 29, 31, 53, 82, 60, 98, 97, 62, 58, 102, 103] .

Chapter 2 provides some background material about grids, discretization of PDEs, iterative solvers, multigrid (MG), and adaptive mesh refinement (AMR).

Chapter 3 presents the basic tools used in the 2D version of the algorithm: stencils, ghost point computation, and flux matching.

Chapter 4 compares the flux matching approach from Chapter 3 with other ways of approximating the fluxes across the interface. In addition, a single formula (bypassing the ghost point interpolation and averaging of fluxes) is derived. This is done first by expanding

and simplifying the ghost point interpolation and avereaging computations. Then a Taylor series based derivation is also shown.

Chapter 5 presents the basic tools used in the 3D version of the algorithm: stencils, ghost point computation, and flux matching.

Chapter 7 describes the multilevel adaptive mesh refinement algorithm. It starts by stating a traditional (non-AMR) formulation of multigrid and then outlines the modification that need to be made in order to formulate the AMR version of multigrid.

Chapter 8 discusses cache optimizations. Processors are much faster than memory. Multilevel memory hierarchies, using cache memory, were developed to compensate for this. Cache optimizations modify the code to take better advantage of the cache memory mechanism. We refer to an algorithm that has been modified for cache in this way as cache aware.

Chapter 6 revisites the ghost point interpolation process. The process is introduced in Chapters 3 and 5 in a way that is useful for describing how the ghost points are computed. That is not efficient for the implementation, though, especially in 3D. This chapter derives a single, simple and efficient formula for computing the ghost points in both 2D and 3D.

Chapter 9 presents numerical results showing speedups associated with cache aware smoothers, integration of the residual computation with the cache aware smoother. and modifying the algorithm to do post-smoothing only.

1.2 Summary of Symbols

Following is a summary of the symbols used in this dissertation.

Symbol	Description
ℓ	Level index
$\ell = 1$	Coarsest level index
ℓ_{max}	Finest level index
$\ell_{max} + 1$	Null, fake level index
Λ	Patch
$\Lambda^{\ell,j}$	j^{th} Patch on level ℓ
Λ^{ℓ}	Union of patches $\Lambda^{\ell,j}$ on level ℓ
Ω	Domain
Ω^{ℓ}	Domain for level ℓ minimally covering Λ^{ℓ}
Λ_c^{ℓ}	Composite grid for levels $1 - \ell$
$\Lambda_c^{\ell_{max}}$	Composite grid for solution to (1.1)
ϕ, u	Solution
ϕ^{ℓ}	Solution on Λ^{ℓ}
ϕ_c^{ℓ}	Solution on Λ_c^{ℓ}
ρ	Right hand side
ρ^{ℓ}	Right hand side on Λ^{ℓ}
ρ_c^{ℓ}	Right hand side on Λ_c^{ℓ}
r	Residual
r^{ℓ}	Residual on Λ^{ℓ}
r_c^{ℓ}	Residual on Λ_c^{ℓ}
e	Correction
e^{ℓ}	Correction on Λ^{ℓ}
e_c^{ℓ}	Correction on Λ_c^{ℓ}
$\mathcal{L}_c$	Composite linear operator
$\mathcal{L}_c^{\ell}$	Composite linear operator on level ℓ
$\mathcal{L}$	Patch based linear operator
$\mathcal{L}^{\ell}$	Patch based linear operator on level ℓ
S_c	Composite smoother
S_c^{ℓ}	Composite smoother on level ℓ
S	Patch based smoother
S^{ℓ}	Patch based smoother on level ℓ
$\mathcal{P}$	Projection from a fine level to a coarser one
$\mathcal{P}^{\ell}$	Projection from level to level $\ell - 1$
$\mathcal{R}$	Projection from a coarse level to a finer one
$\mathcal{R}^{\ell}$	Projection from level ℓ to level $\ell + 1$

Chapter 2

Background

This chapter briefly introduces background concepts: grids, discretization of PDEs, iterative solvers, multigrid (MG), and adaptive mesh refinement (AMR). See [21, 56, 23] and the references given in §1 for more detail.

2.1 Grids

A *grid* is a partitioning of a region. It specifies a finite set of points, called *grid points*, in the region. The partitioning is usually illustrated with a set of *grid lines*. Intersections of grid lines are called *vertices*, and the interiors of individual partitions are called *cells*.

In the context of PDEs, a grid is chosen to specify a subset of points in the domain of the solution function of a PDE. There are different types of grids:

- *Structured* grids can be computed from a set of parameters like the dimensions of the grid and the mesh spacing. The simplest structured grid is a *rectilinear* grid of equally spaced, orthogonal grid lines, as illustrated in Fig. 2.1(a). More complicated structured grids (e.g., *curvalinear* grids [57, 96] can be defined as transformations of rectilinear grids.

- *Unstructured* grids [96, 80] are defined point-wise (or cell-wise). The coordinates of every grid point (or vertex) must be stored separately and explicitly. There is no regular relationship to corellate the locations of grid points in an unstructured grid. See Fig. 2.1(b)

- *Vertex-centered* grids [55, 96] are characterized by the coincidence of grid points with the intersections (vertices) of the grid lines, as illustrated in Fig. 2.1(c).

- *Cell-centered* grids [55, 96] are characterized by having grid points in the interior of partitions (cells) formed by the grid lines, as illustrated in Fig. 2.1(d).

The focus in the present work is on *structured* grids with *cell-centered* grid points.

2.2 Discretization of Elliptic PDEs

Usually, elliptic PDEs cannot be solved analytically. They are too complicated, and a classical analytical solution might not exist at all. Instead, they are solved numerically on a grid using a discrete version of the equations. The discrete equations determine approximate values of the solution function at the grid points. This section shows how to derive the discrete equations on 1-, 2- and 3-dimensional cell-centered structured grids.

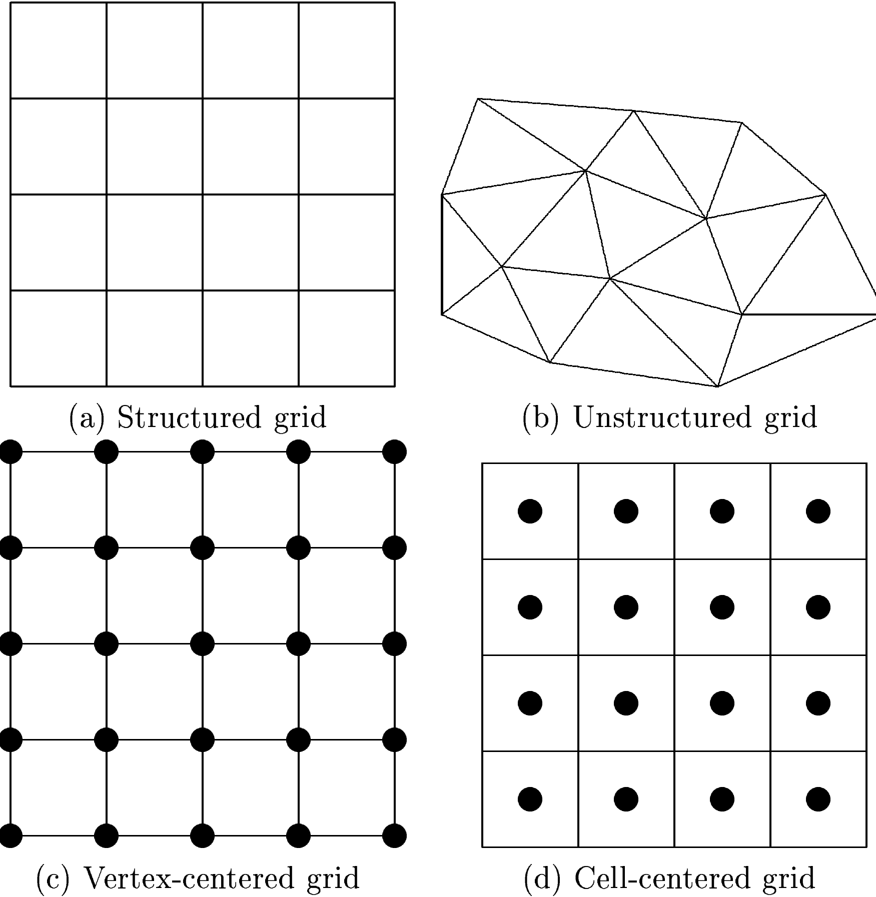


Figure 2.1: Grids.

2.2.1 1D

It is useful to convey the discretization process initially with 1D equations.

A simple 1D elliptic PDE looks like

$$-u'' = f \text{ on } [a, b]$$

with boundary conditions like, for example,

$$u(a) = u(b) = K,$$

where $u = u(x)$ is the solution function and $f = f(x)$ is a given right hand side, commonly called a *forcing* function. When $f = 0$, this is the *Laplace* equation. When $f \neq 0$, this is the *Poisson* equation.

The domain $[a, b]$ is partitioned into a grid with n grid points. For cell-centered grids, the *mesh spacing* is defined by $h = \frac{b-a}{n}$. Let $u_i = u(x_i)$.

Then approximating u'' by

$$u''(x_i) \approx \frac{1}{h^2}\left(u_{i-1} - 2u_i + u_{i+1}\right),$$

at the interior points gives a system of $n-2$ linear equations:

$$-u_{i-1} + 2u_i - u_{i+1} = h^2 f_i,$$

for $2 \le i \le n-1$. The boundary points give two more equations

$$2u_1 - u_2 = h^2 f_1 + K,$$

and

$$-u_{n-1} + 2u_n = h^2 f_n + K,$$

for a total of n equations which can be written in matrix form:

$$Au = f,$$

where $f(x_i) = h^2 f_i$ for $2 \le i \le n-1$, and $f(x_{1,n}) = h^2 f_{1,n} + K$. This example uses *Dirichlet* boundary conditions, meaning the solution values at the boundaries are constant. There exist other more complicated boundary conditions [79].

2.2.2 2D

Let $\Omega = [a,b] \times [c,d]$, and $\partial\Omega$ is the boundary of Ω. A 2D elliptic PDE like

$$-u_{xx} - u_{yy} = f \text{ on } \Omega,$$

with boundary conditions on $\partial\Omega$, is discretized by approximating the derivatives

$$\begin{aligned} u_{xx} &\approx \frac{1}{h^2}(u_{i-1,j} - 2u_{ij} + u_{i+1,j}) \\ u_{yy} &\approx \frac{1}{h^2}(u_{i,j-1} - 2u_{ij} + u_{i,j+1}) \end{aligned}$$

in the equation at interior grid points and applying boundary conditions at boundary points to build a linear system $Ax = b$. The equations at the interior points (i,j) are

$$4u_{ij} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1} = h^2 f_i,$$

and the equations at the boundary points depend on the boundary conditions. See [79] for more details.

2.2.3 3D

Let $\Omega = [a_1,b_1] \times [a_2,b_2] \times [a_3,b_3]$, and $\partial\Omega$ is the boundary of Ω. A 3D elliptic PDE like

$$-u_{xx} - u_{yy} - u_{zz} = f \text{ on } \Omega,$$

with boundary conditions on $\partial\Omega$, is discretized by approximating the derivatives

$$\begin{aligned} u_{xx} &\approx \frac{1}{h^2}\left(u_{i-1,j,k} - 2u_{ijk} + u_{i+1,j,k}\right) \\ u_{yy} &\approx \frac{1}{h^2}\left(u_{i,j-1,k} - 2u_{ijk} + u_{i,j+1,k}\right) \\ u_{zz} &\approx \frac{1}{h^2}\left(u_{i,j,k-1} - 2u_{ijk} + u_{i,j,k+1}\right) \end{aligned}$$

in the equation at interior grid points and applying boundary conditions at boundary points to build a linear system $Ax = b$. The equations at the interior points (i, j, k) are

$$6u_{ijk} - u_{i-1,j,k} - u_{i+1,j,k} - u_{i,j-1,k} - u_{i,j+1,k} - u_{i,j,k-1} - u_{i,j,k+1} = h^2 f_i,$$

and the equations at the boundary points depend on the boundary conditions.

2.3 Iterative Solvers

Iterative solvers solve $Ax = b$ by constructing a sequence of vectors $x^0, x^1, x^2, \ldots$ that converges to the solution vector x. There are two main classes of iterative solvers: *splitting methods* and *projection methods*. The type that is relevant here is splitting methods. The general form of a splitting method is

$$x^{(k+1)} \leftarrow Rx^{(k)} + c,$$

where $A = Q - K$ (which can be viewed as a *splitting* of A), $R = Q^{-1}K$, and $c = Q^{-1}b$. Note that Q must be non-singular. Then

$$\begin{aligned} e^{(k+1)} = x^{(k+1)} - x &= Rx^{(k)} + c - x \\ &= Q^{-1}Kx^{(k)} + Q^{-1}b - x \\ &= Q^{-1}(Q - A)x^{(k)} + Q^{-1}Ax - x \\ &= (I - Q^{-1}A)x^{(k)} - (I - Q^{-1}A)x \\ &= (I - Q^{-1}A)e^{(k)}, \end{aligned}$$

which shows that the $x^{(k)}$ converge to x as long as $I - Q^{-1}A$ is small (i.e., Q is a sufficiently close approximation of A).

The particular splitting method that we are concerned with is Gauss-Seidel. Let $A = D - C_L - C_U$, where D is the diagonal of A, C_L is the strictly lower triangular part of A and C_U is the strictly upper triangular part of A. In Gauss-Seidel, $Q = D - C_L$ and $K = C_U$, so

$$x^{(k+1)} \leftarrow R_{GS}x^{(k)} + c_{GS},$$

where $R_{GS} = (D - C_U)^{-1}C_U$ and $c_{GS} = (D - C_L)^{-1}b$. In a computer code, we update $x^{(k+1)}$ one element at a time. The element-wise form of the Gauss-Seidel iteration is

$$x_i^{(k+1)} \leftarrow \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij}x_j^k\right),$$

for $1 \le i \le n$, where $x = (x_i)_{n\times 1}$, $A = (a_{ij})_{n\times n}$ and $b = (b_i)_{n\times 1}$. In practice, we use just one vector x and overwrite it with the updated values during each iteration.

2.4 Multigrid

This section presents the two main pieces of a multigrid algorithm and then the algorithm itself. It is this algorithm that we adapt to work on complicated hierarchies of adaptively refined meshes in Ch. 7.

2.4.1 Gauss-Seidel as Smoother

Multigrid coordinates the application of an iterative method on a hierarchy of grids. A splitting method like Gauss-Seidel is referred to as a smoother in this context, because it damps high frequency error very quickly. This smoothing property is exploited by applying the smoother on a hierarchy of grids of different resolutions.

2.4.2 Transfer Operators

Multigrid involves moving data between grids. This movement of data is achieved with transfer operators. An interpolation operator moves data from a coarse grid to a fine grid, and a projection operator moves data from a find grid to a coarse grid. Fig. 2.2 illustrates the transfer operators, with P_ℓ denoting projection from level ℓ to level $\ell - 1$ and R_ℓ denoting interpolation from level $\ell - 1$ to level ℓ.

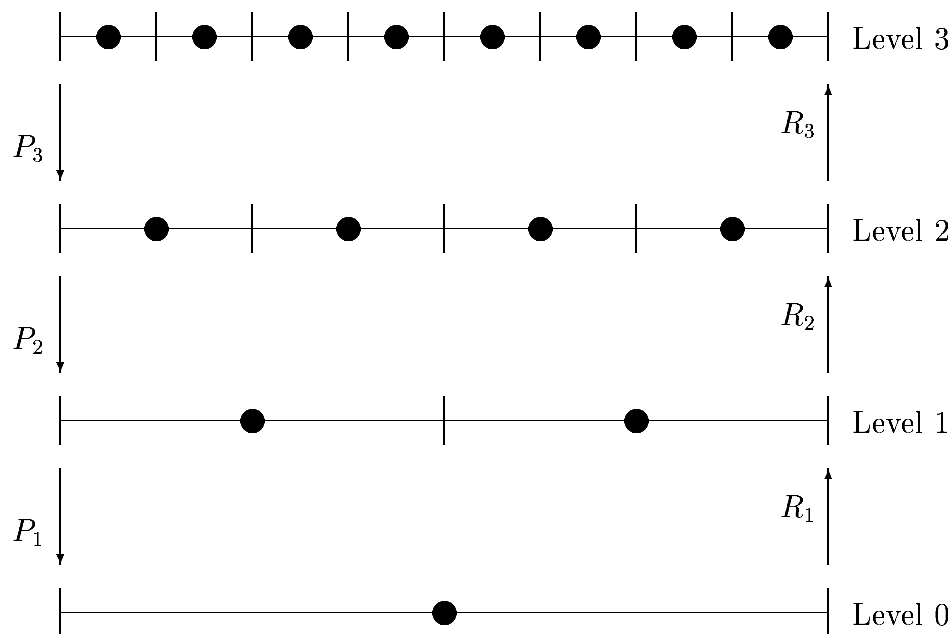


Figure 2.2: Transfer Operators, P_ℓ and R_ℓ.

2.4.3 The V-Cycle

The basic multigrid V-Cycle is illustrated in Fig. 2.3. A standard multigrid V-Cycle begins

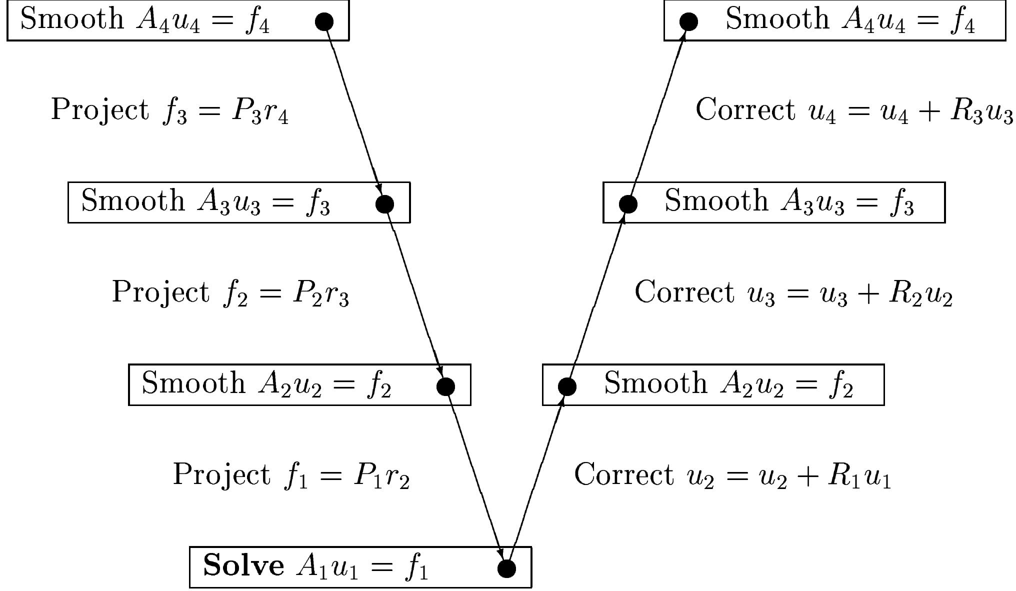


Figure 2.3: V-Cycle.

on the finest level with an application of the smoother, and then the residual, $r_i = f_i - A_iu_i$, is computed and projected as the right hand side of a correction problem on the next coarsest grid. The decrease in resolution exposes new high frequency modes which an application of the smoother on this level will quickly damp. This process is repeated recursively down to the coarsest grid. On the coarsest grid the correction problem is solved directly, and then the correction is recursively interpolated and applied to the finer levels, usually with an additional application of the smoother on each level. See [21] for more detailed background on multigrid.

2.5 AMR

Assume that Ω is overlaid by a union of tensor product meshes $\Lambda^{1,j}$, $j = 1, \ldots, n_1$ that forms a grid in Ω:

$$\Lambda^1 = \bigcup_{j=1}^{n_1} \Lambda^{1,j}, \quad \text{where } \Lambda^1 \subset \Omega.$$

Normally $n_1 = 1$. However, the method works for $n_1 > 1$, too. This is referred to as the level 1, or coarsest grid.

An adaptive mesh refinement procedure is used to define many *patches*. The set of local grid patches corresponding to $\ell - 1$ refinements $(1 < \ell \leq \ell_{max})$ is denoted

$$\Lambda^\ell = \bigcup_{j=1}^{n_\ell} \Lambda^{\ell,j} \quad \text{and} \quad \Lambda^{\ell_{max}+1} = \emptyset.$$

The $\Lambda^{\ell,j}$ are tensor product meshes that have been obtained by adaptively refining the $\Lambda^{\ell-1,j}$ meshes. The definition of $\Lambda^{\ell_{max}+1}$ is just for convenience in presenting algorithms

later. The domains Ω^ℓ and $\Omega^{\ell,j}$ are defined as the minimum domains that include Λ^ℓ and $\Lambda^{\ell,j}$, respectively. Normally, Ω^ℓ is a union of disconnected subdomains (one subdomain corresponding to each level ℓ patch). Note that within an adaptive grid refinement code ℓ_{max} can change (increase or decrease) during the course of solving an actual problem.

The AMR procedure defines a composite grid $\Lambda_c^{\ell_{max}}$ and, more generally, a composite grid hierarchy, $1 < \ell \leq \ell_{max}$, by

$$\Lambda_c^\ell = \bigcup_{i=1}^{\ell} (\Lambda^i - \mathcal{P}(\Lambda^{i+1})), \tag{2.1}$$

where $\mathcal{P}$ is the projection operator discussed below. The ℓth composite grid Λ_c^ℓ contains all points from the ℓth level patches Λ^ℓ as well as additional points from regions not covered by the patches. The new grid points correspond to mesh points from patches on lower levels, always taking from patches on the closest possible level.

Projection and refinement operators are denoted by $\mathcal{P}$ and $\mathcal{R}$, respectively. This notation is standard adaptive mesh refinement notation but is different from multigrid notation (where the symbols are unfortunately reversed). The operators are used interchangeably with either domains Ω^ℓ or grids Λ^ℓ. In terms of domain and grid superscripts, $\mathcal{P}$ *projects* from "fine to coarse," i.e., $\ell \rightarrow \ell - 1$ and $\mathcal{R}$ *refines* from "coarse to fine," i.e., $\ell \rightarrow \ell + 1$.

For the current work, domains must be properly nested

$$\Omega^{\ell_{max}} \subseteq \Omega^{\ell_{max}-1} \subseteq \cdots \subseteq \cdots \Omega^1 \equiv \Omega.$$

This can also be written as

$$\mathcal{R}(\mathcal{P}(\Omega^{\ell+1})) \subseteq \Omega^{\ell+1} \quad \text{and} \quad \mathcal{P}(\Omega^{\ell+1}) \subseteq \Omega^\ell$$

and

$$\Omega = \bigcup_{\ell=1}^{\ell_{max}} (\Omega^\ell - \mathcal{P}(\Omega^{\ell+1})), \quad \text{where } \Omega^{\ell_{max}+1} = \emptyset.$$

The use of tensor product meshes allows for fairly straightforward finite difference and finite volume stencils to define the discrete operator. At internal patch boundaries, however, some care must be taken. *Ghost points* are stored near internal patch boundaries, and quadratic interpolation is used to acquire values at these points so that locally equispaced unknowns are available for use with regular stencils within most of the computations. When computing composite grid residuals, however, more complicated stencils are needed for coarse points adjacent to finer grid patches. This is formally covered in detail in following chapters. The main idea is to use the same interpolated values for the stencils on both the fine grid side and the coarse grid side when updating points that are adjacent to a coarse-fine interface. Hence, the fluxes used to approximate the operator across the coarse-fine interfaces are matched and continuity of the first derivate (i.e., C^1 continuity) is enforced. This is referred to as *flux matching*. The C^1 continuity at the boundary between the coarse and fine subdomains can be precisely defined in terms of the derivatives of the quadratic functions that interpolate the ghost points. See Chs. 3, 5, and 6, for details of the ghost point interpolation procedure. The key point is that enforcing C^1 continuity preserves the

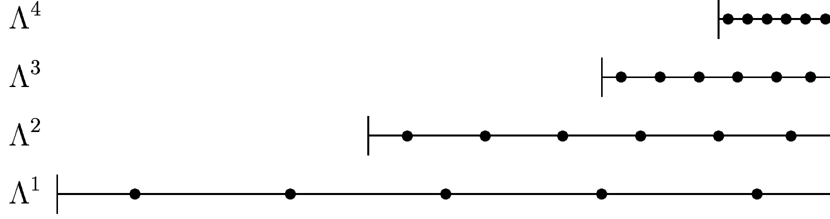


Figure 2.4: A simple patch based grid hierarchy.

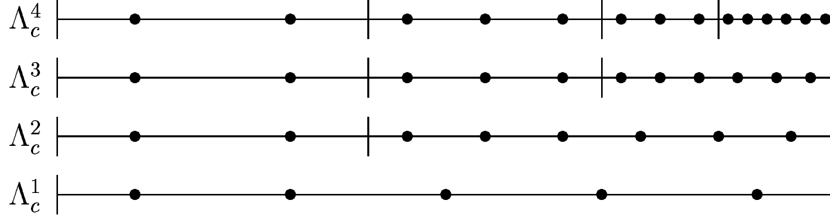


Figure 2.5: Composite grid hierarchy corresponding to the patch hierarchy in Fig 2.4.

second order convergence of the method. Especially for problems with severe fronts or near discontinuities, C^0 continuity alone is not always sufficient. The boundaries of the domains, $\{\partial\Omega^\ell\}$, are required to meet the condition

$$\partial\Omega^{\ell+1} \cap \partial\Omega^\ell \subseteq \partial\Omega$$

which ensures that the flux matching procedure is well defined and of the right approximation order near patch boundaries in the interior of Ω [70].

The solution to (1.1) is approximated using a multigrid algorithm to numerically solve the finite dimensional problem

$$\mathcal{L}_c^{\ell_{max}} \phi_c^{\ell_{max}} = \rho_c^{\ell_{max}}, \tag{2.2}$$

where $\mathcal{L}_c^{\ell_{max}}$ is a matrix representing the discretization on $\Lambda_c^{\ell_{max}}$, $\phi_c^{\ell_{max}}$ are the unknowns, and $\rho_c^{\ell_{max}}$ is the right hand side. Conceptually, the grid hierarchy used within the multigrid procedure is the composite grid hierarchy defined above. In practice, the implementation is patch based (see Ch. 7). This is an important distinction. A simple 1D AMR patch hierarchy is illustrated in Fig. 2.4. In the figure, dots represent grid points and vertical lines represent either physical boundaries or patch boundaries. A patch is a maximal set of contiguous grid points on a given level of the grid hierarchy. There are usually multiple patches per level, although the example hierarchy in Fig. 2.4 has only one patch per level. Fig. 2.5 illustrates the corresponding composite grid hierarchy for the simple 1D example. Each level in this composite grid hierarchy is a composite grid defined in (2.1). Equation (2.2) is defined on the highest level composite grid, namely $\Lambda_c^{\ell_{max}}$. Operators $\mathcal{L}_c^\ell$ are matrices representing the discretizations on composite grids Λ_c^ℓ for all ℓ. In Ch. 7, we define patch based versions of the discrete operators.

Chapter 3

Tools for 2D AMRMG

This chapter outlines the machinery needed to implement the AMRMG algorithm in 2D for both constant and variable coefficient.

3.1 Tools For Constant Coefficient Problems

This section details the tools needed for the 2D AMRMG algorithm using an example of a constant coefficient Poisson problem

$$\Delta\phi = \rho$$

on a square domain

$$\Omega = [0,1]^2$$

with Dirichlet boundary conditions

$$\phi = \gamma \text{ on } \partial\Omega.$$

A sample hierarchy for this example is illustrated in Figure 3.7. We derive the stencils, the ghost point interpolation, and the flux matching procedure. Each of these operations comes in a variety of forms depending on its location in the grid relative to boundaries and coarse-fine interfaces.

The purpose of this example is to nurture intuition that is useful for understanding the algorithms in Ch. 7. It shows, for a specific example, the details of the patch based operators that are used to implement multigrid on adaptive mesh hierarchies.

3.1.1 2D Stencils

Fig. 3.8 shows representatives of the three types of stencils that are required in 2D: interior, edge, and corner stencils. The formulae for these stencils are shown in the following subsections. The derivations of the edge and corner stencils are shown.

Interior

The interior stencil is

$$(\Delta u)_{ij} = (u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{ij})\, h^{-2},$$

and is illustrated in Figure 3.8(a). The discrete equation is

$$4u_{ij} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1} = h^2\rho.$$

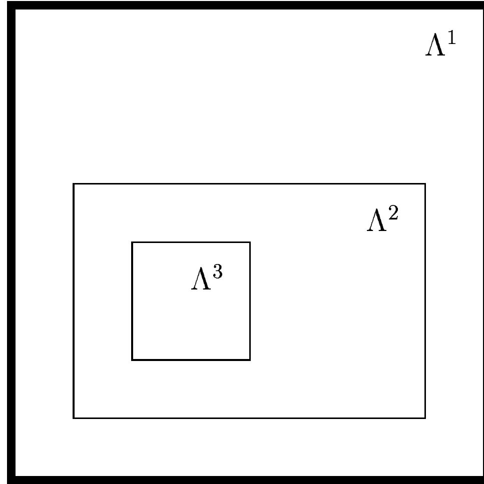


Figure 3.1: A sample 2D grid hierarchy. The dark lines denote the boundary.

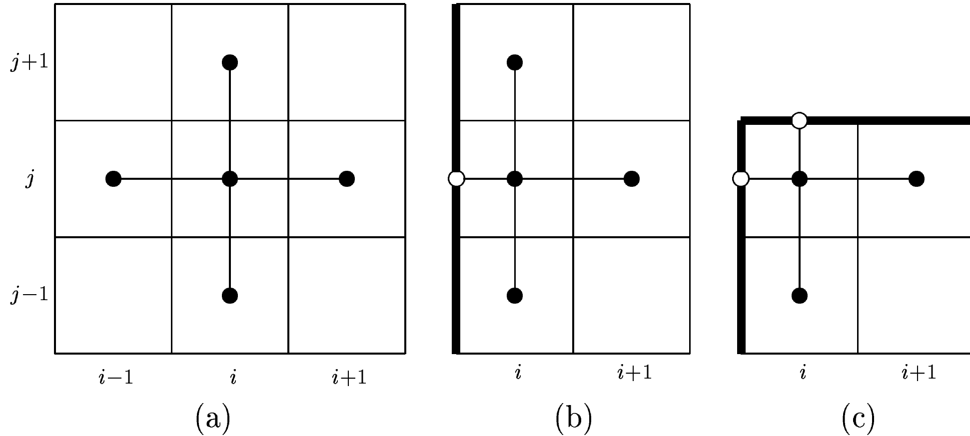


Figure 3.2: Stencils. The dark lines in (b) and (c) represent boundaries.

Edge Boundary

The edge stencil, corresponding to Figure 3.8(b), is derived by combining expansions in x and y. We write the expansions in x as

$$\left.\begin{array}{l} u_{i+1} \approx u_i + hu_i' + \frac{1}{2}h^2u_i'' \\ u_{i-\frac{1}{2}} \approx u_i - \frac{1}{2}hu_i' + \frac{1}{8}h^2u_i'' \end{array}\right\} \Rightarrow u_{i+1} + 2u_{i-\frac{1}{2}} \approx 3u_i + \frac{3}{4}h^2u_i''$$

using u_i, u_i', and u_i'' for the value of the function, its derivative in the x-direction, and its second derivative in the x-direction at the ith grid point in the x-direction. We write the expansions in y as

$$\left.\begin{array}{l} u_{j+1} \approx u_j + hu_j' + \frac{1}{2}h^2u_j'' \\ u_{j-1} \approx u_j - hu_j' + \frac{1}{2}h^2u_j'' \end{array}\right\} \Rightarrow u_{j+1} + u_{j-1} \approx 2u_j + h^2u_j''$$

using u_j, u'_j, and u''_j for the value of the function, its derivative in the y-direction, and its second derivative in the y-direction at the jth grid point in the y-direction. Combine these to obtain

$$(\Delta u)_{ij} \approx u''_i + u''_j \approx \left(\frac{4}{3}u_{i+1,j} + \frac{8}{3}u_{i-\frac{1}{2},j} + u_{i,j+1} + u_{i,j-1} - 6u_{ij}\right) h^{-2}.$$

The corresponding discrete equation is

$$6u_{ij} - \frac{4}{3}u_{i+1,j} - \frac{8}{3}u_{i-\frac{1}{2},j} - u_{i,j+1} - u_{i,j-1} = h^2 \rho.$$

Corner Boundary

The corner stencil, corresponding to Figure 3.8(c), is derived by combining expansions in x and y. Using the same notation as in §3.1.1, we write the expansions in x as

$$\left.\begin{array}{l} u_{i+1} \approx u_i + hu'_i + \frac{1}{2}h^2u''_i \\ u_{i-\frac{1}{2}} \approx u_i - \frac{1}{2}hu'_i + \frac{1}{8}h^2u''_i \end{array}\right\} \Rightarrow u_{i+1} + 2u_{i-\frac{1}{2}} \approx 3u_i + \frac{3}{4}h^2u''_i$$

and the expansions in y as

$$\left.\begin{array}{l} u_{j+\frac{1}{2}} \approx u_j + \frac{1}{2}hu'_j + \frac{1}{8}h^2u''_j \\ u_{j-1} \approx u_j - hu'_j + \frac{1}{2}h^2u''_j \end{array}\right\} \Rightarrow 2u_{j+\frac{1}{2}} + u_{j-1} \approx 3u_j + \frac{3}{4}h^2u''_j.$$

We combine them to obtain

$$(\Delta u)_{ij} \approx u''_i + u''_j \approx \left(\frac{4}{3}u_{i+1,j} + \frac{8}{3}u_{i-\frac{1}{2},j} + \frac{8}{3}u_{i,j+\frac{1}{2}} + \frac{4}{3}u_{i,j-1} - 8u_{ij}\right) h^{-2}.$$

The corresponding discrete equation is

$$8u_{ij} - \frac{4}{3}u_{i+1,j} - \frac{8}{3}u_{i-\frac{1}{2},j} - \frac{8}{3}u_{i,j+\frac{1}{2}} - \frac{4}{3}u_{i,j-1} = h^2 \rho.$$

3.1.2 Interpolation of Ghost Points in 2D

In order to apply regular stencils at the boundary points of patches and enforce flux matching at the coarse-fine interfaces, we need to interpolate values at ghost points around the edges of the patches. This section explains how to do these interpolations.

One Coarse-Fine Interface

Values are needed at the $\otimes$ ghost points for the one-sided coarse-fine interface illustrated in Figure 3.3(a). There are two steps. *Step 1*: Compute values for the the $\oslash$ points. Let $a = u(\bullet_A)$, $b = u(\bullet_B)$ and $c = u(\bullet_C)$, and let h be the mesh spacing on the finer grid. We derive a C^1 quadratic interpolation formula using

$$c_0 + c_1 x + c_2 x^2 = u(x),$$

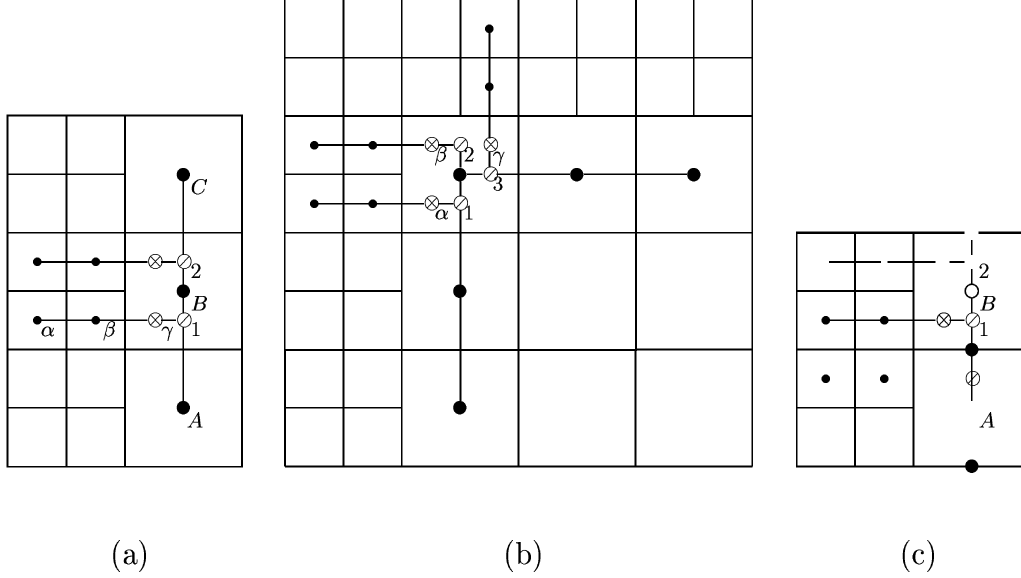


Figure 3.3: Interpolation of Ghost Points. The dark edge in (c) indicates a boundary.

where

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 2h & (2h)^2 \\ 1 & 4h & (4h)^2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} a \\ b \\ c \end{bmatrix} \Rightarrow \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} a \\ \frac{1}{h}(b - \frac{1}{4}c - \frac{3}{4}a) \\ \frac{1}{h^2}(\frac{1}{8}c - \frac{1}{4}b + \frac{1}{8}a) \end{bmatrix}.$$

Then

$$u(\oslash_1) = c_0 + c_1(\frac{3}{2}h) + c_2(\frac{3}{2}h)^2,$$

and

$$u(\oslash_2) = c_0 + c_1(\frac{5}{2}h) + c_2(\frac{5}{2}h)^2.$$

Step 2: Compute values for the $\otimes$ points. Let $a = u(\bullet_\alpha)$, $b = u(\bullet_\beta)$ and $c = u(\oslash_1)$, and let h be the mesh spacing on the finer grid. Again, we derive a C^1 quadratic interpolation formula using

$$c_0 + c_1 x + c_2 x^2 = u(x),$$

where

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & h & h^2 \\ 1 & \frac{5}{2}h & (\frac{5}{2}h)^2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} a \\ b \\ c \end{bmatrix} \Rightarrow \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} a \\ -\frac{1}{15h}(21a - 25b + 4c) \\ \frac{2}{15h^2}(3a - 5b + 2c) \end{bmatrix}.$$

Then

$$u(\otimes_\gamma) = c_0 + c_1(2h) + c_2(2h)^2.$$

The procedure is analogous for the other $\otimes$ point.

Two Coarse-Fine Interfaces

Ghost point interpolation at a two-sided coarse-fine interface is illustrated in Figure 3.3(b). Again, values are needed at the $\otimes$ points, and there are two steps. We use the notation

$c_{\{0,1,2\}}$ generically here to refer to the coefficients associated with each interpolation. The values of the coefficients are *not* the same for every interpolation.

Step 1: Use the same system as in *Step 1* of §3.1.2, once in each direction, to get the coefficents $c_{\{0,1,2\}}$ needed to compute values for the $\oslash$ points. In the y-direction,

$$u(\oslash_1) = c_0 + c_1\left(\frac{7}{2}h\right) + c_2\left(\frac{7}{2}h\right)^2$$

and

$$u(\oslash_2) = c_0 + c_1\left(\frac{9}{2}h\right) + c_2\left(\frac{9}{2}h\right)^2.$$

In the x-direction,

$$u(\oslash_3) = c_0 + c_1\left(\frac{1}{2}h\right) + c_2\left(\frac{1}{2}h\right)^2.$$

Step 2: Use the same system as in *Step 2* of §3.1.2, twice in the x-direction and once in the y-direction, to get the coefficents $c_{\{0,1,2\}}$ needed to compute values for the $\otimes$ points. In the x-direction,

$$u(\otimes_{\{\alpha,\beta\}}) = c_0 + c_1(2h) + c_2(2h)^2,$$

and in the y-direction,

$$u(\otimes_\gamma) = c_0 + c_1(2h) + c_2(2h)^2.$$

At a Boundary

Ghost point interpolation at a coarse-fine interface adjacent to a boundary is illustrated in Figure 3.3(c). Once again, values are needed at the $\otimes$ points, and there are two steps.

Step 1: Compute values for the $\oslash$ points. Let $a = u(\bullet_A)$, $b = u(\bullet_B)$ and $c = u(\circ)$, and let h be the mesh spacing on the finer grid. We want

$$c_0 + c_1 x + c_2 x^2 = u(x),$$

where

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 2h & (2h)^2 \\ 1 & 3h & (3h)^2 \end{bmatrix}\begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} a \\ b \\ c \end{bmatrix} \Rightarrow \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} a \\ -\frac{1}{24h}(5a - 9b + 4c) \\ -\frac{1}{24h^2}(a + 3b - 4c) \end{bmatrix}.$$

Then

$$u(\oslash_1) = c_0 + c_1(\frac{3}{2}h) + c_2(\frac{3}{2}h)^2,$$

and

$$u(\oslash_2) = c_0 + c_1(\frac{5}{2}h) + c_2(\frac{5}{2}h)^2.$$

Step 2: Compute values for the $\otimes$ points by the same system as in *Step 2* of §3.1.2.

See Ch. 6 for a more detailed look at the ghost point interpolation procedure and a derivation of a one step single formula for the $\otimes$ points.

Figure 3.4: 2D flux matching with one coarse-fine interface.

3.1.3 Flux Matching in 2D

This section introduces the flux-matching computations. Flux matching is used to avoid one-sided derivatives at grid points adjacent to refinement patches and preserve C^1 continuity across the interface between refinement levels.

The computations in this section involve values interpolated at the ghost points $\otimes$ from §3.2.2. The value of the solution at points $\otimes_{\{1,2,3\}}$ is represented by $u^{\ell+1}_{\otimes_{\{1,2,3\}}}$.

One Coarse-Fine Interface

Flux matching at a one-sided coarse-fine interface is illustrated in Figure 3.4. Notice that the fluxes used here are the same as, i.e., *match*, the fluxes computed by the stencils on the boundary of the next finer grid. We use

$$(\Delta u^{\ell})_{ij} \approx \frac{1}{h^{\ell}}\left(f^{\ell}_{i+\frac{1}{2},j} - f^{\ell}_{i-\frac{1}{2},j} + f^{\ell}_{i,j+\frac{1}{2}} - f^{\ell}_{i,j-\frac{1}{2}}\right),$$

where

$$\begin{aligned}
f^{\ell}_{i+\frac{1}{2},j} &= \tfrac{1}{h_\ell}\left(u^{\ell}_{i+1,j} - u^{\ell}_{i,j}\right), \\
f^{\ell}_{i-\frac{1}{2},j} &= \tfrac{1}{2}\left(\tfrac{1}{h_{\ell+1}}(u^{\ell+1}_{\otimes_1} - u^{\ell+1}_{2(i-1),2j-1}) + \tfrac{1}{h_{\ell+1}}(u^{\ell+1}_{\otimes_2} - u^{\ell+1}_{2(i-1),2j})\right) \\
f^{\ell}_{i,j+\frac{1}{2}} &= \tfrac{1}{h_\ell}\left(u^{\ell}_{i,j+1} - u^{\ell}_{ij}\right), \\
f^{\ell}_{i,j-\frac{1}{2}} &= \tfrac{1}{h_\ell}\left(u^{\ell}_{ij} - u^{\ell}_{i,j-1}\right).
\end{aligned}$$

Two Coarse-Fine Interfaces

Flux matching at a two-sided coarse-fine interface is illustrated in Figure 3.5. We use

$$(\Delta u^{\ell})_{ij} \approx \frac{1}{h_{\ell}}\left(f^{\ell}_{i+\frac{1}{2},j} - f^{\ell}_{i-\frac{1}{2},j} + f^{\ell}_{i,j+\frac{1}{2}} - f^{\ell}_{i,j-\frac{1}{2}}\right),$$

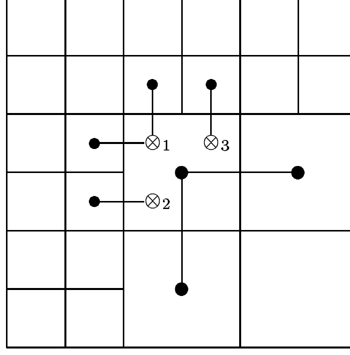


Figure 3.5: 2D flux matching with two coarse-fine interfaces.

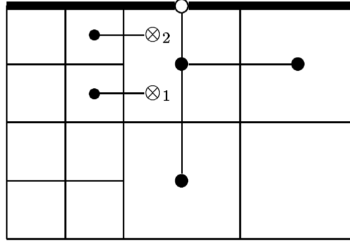


Figure 3.6: 2D flux matching adjacent to a boundary.

where

$$
\begin{aligned}
f^{\ell}_{i+\frac{1}{2},j} &= \tfrac{1}{h_\ell}\left(u^{\ell}_{i+1,j} - u^{\ell}_{ij}\right), \\
f^{\ell}_{i-\frac{1}{2},j} &= \tfrac{1}{2}\left(\tfrac{1}{h_{\ell+1}}(u^{\ell+1}_{\otimes_1} - u^{\ell+1}_{2(i-1),2j-1}) + \tfrac{1}{h_{\ell+1}}(u^{\ell+1}_{\otimes_2} - u^{\ell+1}_{2(i-1),2j})\right), \\
f^{\ell}_{i,j+\frac{1}{2}} &= \tfrac{1}{2}\left(\tfrac{1}{h_{\ell+1}}(u^{\ell+1}_{2i-1,2(j+1)-1} - u^{\ell+1}_{\otimes_2}) + \tfrac{1}{h_{\ell+1}}(u^{\ell+1}_{2i,2(j+1)-1} - u^{\ell+1}_{\otimes_3})\right), \\
f^{\ell}_{i,j-\frac{1}{2}} &= \tfrac{1}{h_\ell}\left(u^{\ell}_{ij} - u^{\ell}_{i,j-1}\right).
\end{aligned}
$$

At a Boundary

Flux matching at a coarse-fine interface that is adjacent to a boundary is illustrated in Figure 3.6. The differences are not obvious, so we show the derivation:

$$
\left.
\begin{aligned}
(u_{xx})_{ij} &\approx \tfrac{1}{h_\ell}\left(f_{i+\frac{1}{2},j} - f_{i-\frac{1}{2},j}\right) \\
(u_{yy})_{ij} &\approx \tfrac{1}{h_\ell^2}\left(\tfrac{8}{3}u_{i,j+\frac{1}{2}} + \tfrac{4}{3}u_{i,j-1} - 4u_{ij}\right)
\end{aligned}
\right\} \Rightarrow
$$

$$
(\Delta u)_{ij} \approx \frac{1}{h_\ell}\left(f_{i+\frac{1}{2},j} - f_{i-\frac{1}{2},j} + \frac{1}{h_\ell}\left(\frac{8}{3}u_{i,j+\frac{1}{2}} + \frac{4}{3}u_{i,j-1} - 4u_{ij}\right)\right),
$$

where $f_{i\pm\frac{1}{2}}$ is as in §3.2.3

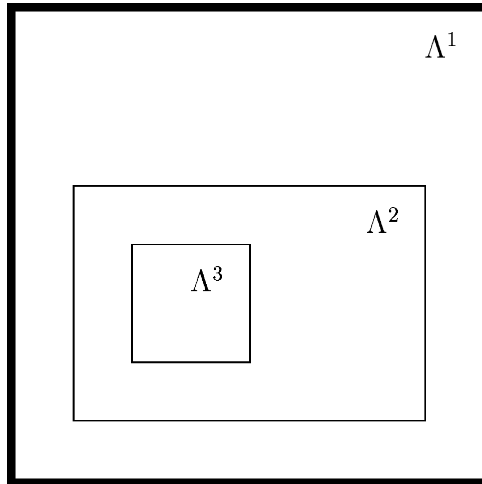


Figure 3.7: A sample 2D grid hierarchy. The dark lines denote the boundary.

3.2 Tools For Variable Coefficient Problems

This section details the tools needed for the 2D AMRMG algorithm using an example of a variable coefficient problem

$$-\nabla \cdot (a \nabla u) = \rho.$$

on a rectangular domain

$$\Omega = [a, b] \times [c, d]$$

with Dirichlet boundary conditions

$$\phi = \gamma \text{ on } \partial\Omega.$$

We will use a control volume approach [79] to discretize this problem.

A sample hierarchy for this example is illustrated in Figure 3.7. We derive the stencils, the ghost point interpolation, and the flux matching procedure. Each of these operations comes in a variety of forms depending on its location in the grid relative to boundaries and coarse-fine interfaces.

The purpose of this example is to nurture intuition that is useful for understanding the algorithms in Ch. 7. It shows, for a specific example, the details of the patch based operators that are used to implement multigrid on adaptive mesh hierarchies.

3.2.1 Stencils

Figure 3.8 shows representatives of the three types of stencils that are required in 2D: interior, edge, and corner stencils. The formulae for these stencils are shown in the following subsections.

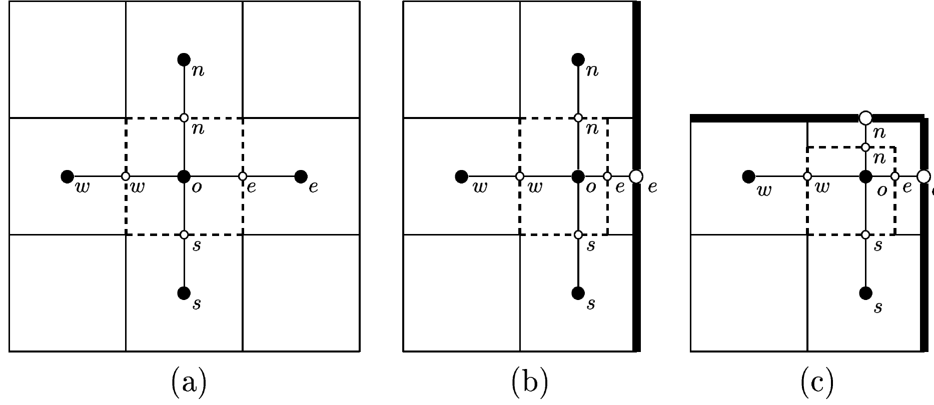


Figure 3.8: Stencils. The dark lines in (b) and (c) represent boundaries.

Interior

See Fig. 3.8(a) for an illustration of an interior stencil. The region within the dashed lines is the control volume, denoted by V. The boundary of the control volume, represented by the dashed lines, is denoted by ∂V.

Discretizing the integral form

$$-\int_{\partial V} a\frac{\partial u}{\partial n} = \int_V \rho$$

gives

$$\begin{aligned} \frac{a_n(u_n - u_o)}{\Delta y}\cdot \Delta x \quad &- \quad \frac{a_w(u_o - u_w)}{\Delta x}\cdot \Delta y + \\ \frac{a_e(u_e - u_o)}{\Delta x}\cdot \Delta y \quad &- \quad \frac{a_s(u_o - u_s)}{\Delta y}\cdot \Delta x = -\Delta x \Delta y \rho_o, \end{aligned}$$

which reduces to

$$(a_n + a_s + a_e + a_w)u_o - a_n u_n - a_s u_s - a_w u_w - a_e u_e = h^2 \rho_o,$$

when $h = \Delta x = \Delta y$.

Edge Boundary

See Fig. 3.8(b) for an illustration of an edge stencil. The region within the dashed lines is the control volume, denoted by V. The boundary of the control volume, represented by the dashed lines, is denoted by ∂V.

Discretizing the integral form

$$-\int_{\partial V} a\frac{\partial u}{\partial n} = \int_V \rho$$

gives

$$\frac{a_n(u_n - u_o)}{\Delta y} \cdot \frac{3}{4}\Delta x \quad - \quad \frac{a_s(u_o - u_s)}{\Delta y} \cdot \frac{3}{4}\Delta x +$$
$$\frac{a_e(u_e - u_o)}{\frac{1}{2}\Delta x} \cdot \Delta y \quad - \quad \frac{a_w(u_o - u_w)}{\Delta x} \cdot \Delta y = -\frac{3}{4}\Delta x \Delta y \rho_o,$$

which reduces to

$$(a_n + a_s + \frac{8}{3}a_e + \frac{4}{3}a_w)u_o - a_n u_n - a_s u_s - \frac{4}{3}a_w u_w - \frac{8}{3}a_e u_e = h^2 \rho_o,$$

when $h = \Delta x = \Delta y$.

Corner Boundary

See Fig. 3.8(c) for an illustration of a corner stencil. The region within the dashed lines is the control volume, denoted by V. The boundary of the control volume, represented by the dashed lines, is denoted by ∂V.

Discretizing the integral form

$$-\int_{\partial V} a \frac{\partial u}{\partial n} = \int_V \rho$$

gives

$$\frac{a_n(u_n - u_o)}{\frac{1}{2}\Delta y} \cdot \frac{3}{4}\Delta x \quad - \quad \frac{a_s(u_o - u_s)}{\Delta y} \cdot \frac{3}{4}\Delta x +$$
$$\frac{a_e(u_e - u_o)}{\frac{1}{2}\Delta x} \cdot \frac{3}{4}\Delta y \quad - \quad \frac{a_w(u_o - u_w)}{\Delta x} \cdot \frac{3}{4}\Delta y = -\frac{9}{16}\Delta x \Delta y \rho_o,$$

which reduces to

$$(\frac{8}{3}a_n + \frac{4}{3}a_s + \frac{8}{3}a_e + \frac{4}{3}a_w)u_o - \frac{8}{3}a_n u_n - \frac{4}{3}a_s u_s - \frac{4}{3}a_w u_w - \frac{8}{3}a_e u_e = h^2 \rho_o,$$

when $h = \Delta x = \Delta y$.

3.2.2 Interpolation of Ghost Points in 2D

The ghost point interpolation process for the variable coefficient case is the same as for the constant coefficient case discussed in §3.2.2.

3.2.3 Flux Matching

This section introduces the flux-matching computations. Flux matching is used to avoid one-sided derivatives and preserve C^1 continuity. It is necessary for the patch based operator $\mathcal{L}^\ell$ defined in Ch. 7.

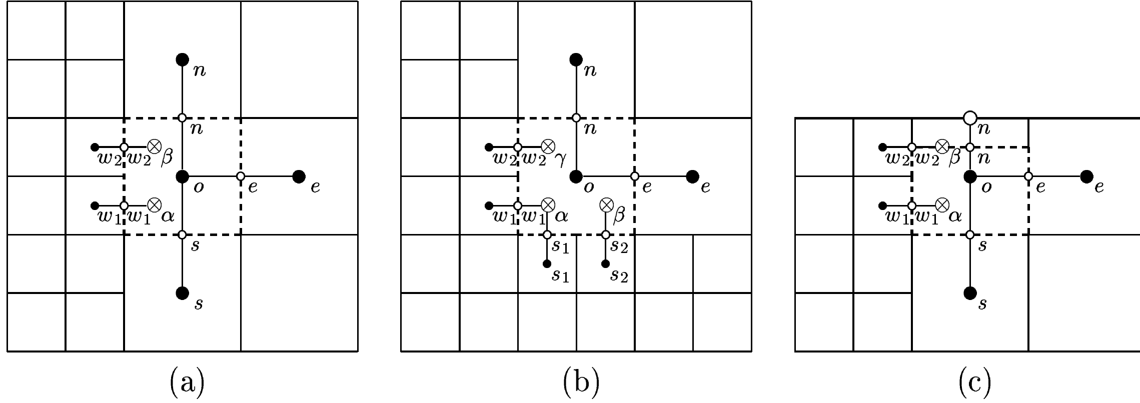


Figure 3.9: Flux Matching. The dark edge in (c) represents a boundary.

See Fig. 3.9 for illustrations. The regions with the dashed lines are the control volumes and is denoted by V. The boundaries of the control volumes, represented by dashed lines, is denoted by ∂V.

The grid points on edges represent coefficient values. The grid points in cells represent solution values, as usual. For example, $a_n = a(\circ_n)$ and $u_n = u(\bullet_n)$.

One Coarse-Fine Interface

See Fig. 3.9(a) for labels. The region within the dashed lines is the control volume, denoted by V. The boundary of the control volume, represented by the dashed lines, is denoted by ∂V.

Discretizing the integral form

$$-\int_{\partial V} a\frac{\partial u}{\partial n} = \int_V \rho$$

gives

$$-(f_n - f_s + f_e - f_w) = h^2\rho_o,$$

where

$$\begin{aligned}
f_n &= a_n(u_n - u_o) \\
f_s &= a_s(u_o - u_s) \\
f_e &= a_e(u_e - u_o) \\
f_w &= a_{w_1}(u_\alpha - u_{w_1}) + a_{w_2}(u_\beta - u_{w_2})
\end{aligned}$$

where $h = \Delta x = \Delta y$.

Two Coarse-Fine Interfaces

See Fig. 3.9(b) for labels. The region within the dashed lines is the control volume, denoted by V. The boundary of the control volume, represented by the dashed lines, is denoted by ∂V.

Discretizing the integral form

$$-\int_{\partial V} a\frac{\partial u}{\partial n} = \int_V \rho$$

gives

$$-(f_n - f_s + f_e - f_w) = h^2\rho_o,$$

where

$$\begin{aligned} f_n &= a_n(u_n - u_o) \\ f_s &= a_{s_1}(u_\alpha - u_{s_1}) + a_{s_2}(u_\beta - u_{s_2}) \\ f_e &= a_e(u_e - u_o) \\ f_w &= a_{w_1}(u_\alpha - u_{w_1}) + a_{w_2}(u_\gamma - u_{w_2}) \end{aligned}$$

where $h = \Delta x = \Delta y$.

At a Boundary

See Fig. 3.9(c) for labels. The region within the dashed lines is the control volume, denoted by V. The boundary of the control volume, represented by the dashed lines, is denoted by ∂V.

Discretizing the integral form

$$-\int_{\partial V} a\frac{\partial u}{\partial n} = \int_V \rho$$

gives

$$-(f_n - f_s + f_e - f_w) = \frac{3}{4}h^2\rho_o,$$

where

$$\begin{aligned} f_n &= \frac{a_n(u_n - u_o)}{\frac{1}{2}h} \cdot h = 2a_n(0 - u_o) = -2a_n u_o \\ f_s &= a_s(u_o - u_s) \\ f_e &= \frac{a_e(u_e - u_o)}{h} \cdot \frac{3}{4}h = \frac{3}{4}a_e(u_e - u_o) \\ f_w &= \frac{a_{w_1}(u_\alpha - u_{w_1})}{\frac{1}{2}h} \cdot \frac{1}{2}h + \frac{a_{w_2}(u_\beta - u_{w_2})}{\frac{1}{2}h} \cdot \frac{1}{4}h \\ &= a_{w_1}(u_\alpha - u_{w_1}) + \frac{1}{2}a_{w_2}(u_\beta - u_{w_2}) \end{aligned}$$

where $h = \Delta x = \Delta y$.

Chapter 4

Analysis of Flux Matching

This chapter shows an analysis of the flux matching procedure and a comparison with an alternate method using Taylor series expansion to approximate the fluxes across interfaces. Also shown is a Taylor series derivation of the flux matching formula itself.

We compare the flux matching style operator with alternative stencil based operators and with Taylor series based formulae. We would like to see what alternative formulations are equivalent to the flux matching procedure, and we would like to see how the error of alternative approaches compares with flux matching.

Note that in this chapter, the notation for grid points (e.g., $\bullet_1$) is used also as shorthand for representing the value of the function there. This simplifies the notation and is more readable.

The computations in the section are done in Matlab.

4.1 Stencil Version, Using Ghost Coarse Grid Point.

One obvious alternative to the flux matching approach that we implemented is to use ghost points on the fine grid side of the interface. These are located on the fine grid side where coarse grid points would be, and so they are referred to as ghost coarse grid points here. The notation here is based on the illustration in Fig. 4.1. Let $\bullet_a$ be the reference point, $\bullet_a = (0,0)$. The ghost coarse grid point is illustrated by the big $\bigotimes$. It is computed by

$$\bigotimes = \frac{1}{2}\left(\varphi_S\left(\frac{1}{2}h, \frac{1}{2}h\right) + \varphi_N\left(\frac{1}{2}h, \frac{1}{2}h\right)\right)$$

where φ_S is the 2D quadratic polynomial that interpolates the points $\bullet_a$, $\bullet_b$, $\bullet_c$, $\bullet_d$, $\bullet_S$, $\bullet_P$, and φ_N is the 2D quadratic polynomial that interpolates $\bullet_a$, $\bullet_b$, $\bullet_c$, $\bullet_d$, $\bullet_N$, $\bullet_P$. Quadratic interpolation in 2D requires six coefficients, so the grid points used in the interpolation are chosen as the closest six points to the interpolation point. We derive C^1 quadratic interpolation formula φ_S using

$$c_0 + c_1 x + c_2 y + c_3 xy + c_4 x^2 + c_5 y^2 = u(x,y),$$

where

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & h & 0 & 0 & h^2 & 0 \\ 1 & 0 & h & 0 & 0 & h^2 \\ 1 & h & h & h^2 & h^2 & h^2 \\ 1 & \frac{5}{2}h & -\frac{3}{2}h & -\frac{15}{4}h^2 & \left(\frac{5}{2}h\right)^2 & \left(\frac{3}{2}h\right)^2 \\ 1 & \frac{5}{2}h & \frac{1}{2}h & \frac{5}{4}h^2 & \left(\frac{5}{2}h\right)^2 & \left(\frac{1}{2}h\right)^2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{bmatrix} = \begin{bmatrix} \bullet_a \\ \bullet_b \\ \bullet_c \\ \bullet_d \\ \bullet_S \\ \bullet_P \end{bmatrix} \Longrightarrow$$

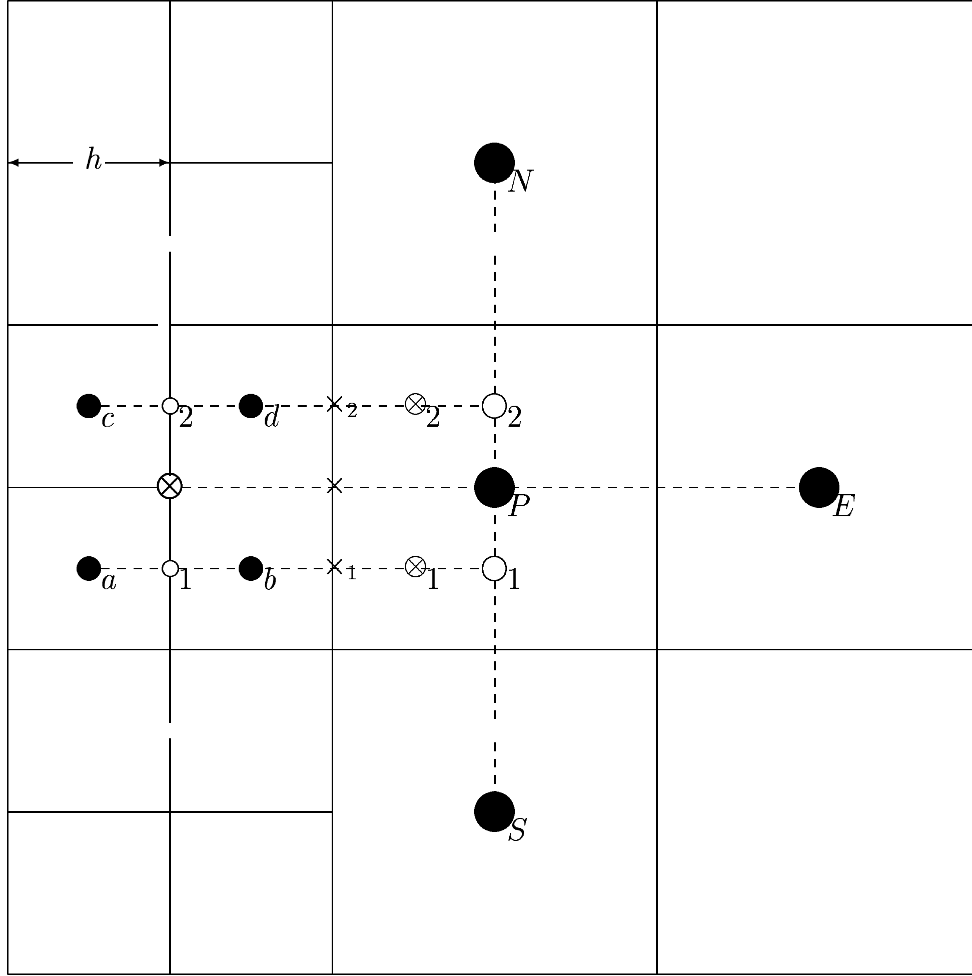


Figure 4.1: Flux matching

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{bmatrix} = \begin{bmatrix} \bullet_a \\ \frac{-1}{60h}(75\bullet_a - 15\bullet_d + \bullet_S - 85\bullet_b + 9\bullet_c + 15\bullet_P) \\ \frac{-1}{4h}(7\bullet_a + 5\bullet_d + \bullet_S - 5\bullet_b - 7\bullet_c - \bullet_P) \\ \frac{1}{h^2}(-\bullet_c + \bullet_d - \bullet_b + \bullet_a) \\ \frac{1}{60h^2}(-15\bullet_d + \bullet_S - 25\bullet_b + 9\bullet_c + 15\bullet_a + 15\bullet_P) \\ \frac{1}{4h^2}(3\bullet_a + 5\bullet_d + \bullet_S - 5\bullet_b - 3\bullet_c - \bullet_P) \end{bmatrix}.$$

We derive C^1 quadratic interpolation formula φ_N using

$$c_0 + c_1 x + c_2 y + c_3 xy + c_4 x^2 + c_5 y^2 = u(x, y),$$

where

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & h & 0 & 0 & h^2 & 0 \\ 1 & 0 & h & 0 & 0 & h^2 \\ 1 & h & h & h^2 & h^2 & h^2 \\ 1 & \frac{5}{2}h & \frac{5}{2}h & \frac{25}{4}h^2 & \left(\frac{5}{2}h\right)^2 & \left(\frac{5}{2}h\right)^2 \\ 1 & \frac{5}{2}h & \frac{1}{2}h & \frac{5}{4}h^2 & \left(\frac{5}{2}h\right)^2 & \left(\frac{1}{2}h\right)^2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{bmatrix} = \begin{bmatrix} \bullet_a \\ \bullet_b \\ \bullet_c \\ \bullet_d \\ \bullet_N \\ \bullet_P \end{bmatrix} \Longrightarrow$$

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{bmatrix} = \begin{bmatrix} \bullet_a \\ \frac{-1}{60h}(69\bullet_a - 25\bullet_d + \bullet_N - 75\bullet_b + 15\bullet_c + 15\bullet_P) \\ \frac{-1}{4h}(\bullet_a - 5\bullet_d + \bullet_N + 5\bullet_b - \bullet_c - \bullet_P) \\ \frac{1}{h^2}(-\bullet_c + \bullet_d - \bullet_b + \bullet_a) \\ \frac{1}{60h^2}(9\bullet_a - 25\bullet_d + \bullet_N - 15\bullet_b + 15\bullet_c + 15\bullet_P) \\ \frac{-1}{4h^2}(3\bullet_a + 5\bullet_d - \bullet_N - 5\bullet_b - 3\bullet_c + \bullet_P) \end{bmatrix}.$$

So

$$\begin{aligned} \varphi_S\left(\frac{1}{2}h, \frac{1}{2}h\right) &= c_0 + c_1(\frac{1}{2}h) + c_2(\frac{1}{2}h) + c_3(\frac{1}{2}h)(\frac{1}{2}h) + c_4(\frac{1}{2}h)^2 + c_5(\frac{1}{2}h)^2 \\ &= -\frac{1}{15}\bullet_P + \frac{2}{3}\bullet_b + \frac{2}{5}\bullet_c \end{aligned}$$

and

$$\begin{aligned} \varphi_N\left(\frac{1}{2}h, \frac{1}{2}h\right) &= c_0 + c_1(\frac{1}{2}h) + c_2(\frac{1}{2}h) + c_3(\frac{1}{2}h)(\frac{1}{2}h) + c_4(\frac{1}{2}h)^2 + c_5(\frac{1}{2}h)^2 \\ &= \frac{2}{5}\bullet_a + \frac{2}{3}\bullet_d - \frac{1}{15}\bullet_P. \end{aligned}$$

Then take the average

$$\otimes = \frac{1}{2}\left(\varphi_S\left(\frac{1}{2}h, \frac{1}{2}h\right) + \varphi_N\left(\frac{1}{2}h, \frac{1}{2}h\right)\right) = \frac{1}{5}\bullet_a + \frac{1}{3}\bullet_d - \frac{1}{15}\bullet_P + \frac{1}{3}\bullet_b + \frac{1}{5}\bullet_c.$$

And

$$\begin{aligned} u_x(\otimes) &= \frac{1}{2h}\left(\bullet_P - \left(\frac{1}{5}\bullet_a + \frac{1}{3}\bullet_d - \frac{1}{15}\bullet_P + \frac{1}{3}\bullet_b + \frac{1}{5}\bullet_c\right)\right) \\ &= \frac{32}{15}\bullet_P - \frac{2}{5}\bullet_a - \frac{2}{3}\bullet_d - \frac{2}{3}\bullet_b - \frac{2}{5}\bullet_c. \end{aligned}$$

This is very dissimilar to the flux matching formula (§4.3).

Alternatively, the ghost coarse grid point could be computed in stages by 1D interpolation. In particular, compute

$$\begin{aligned} \otimes &= \frac{1}{2}(\circ_1 + \circ_2) \\ &= \frac{1}{2}\left(\frac{1}{30}(12\bullet_a + 20\bullet_b - 2\bigcirc_1) + \frac{1}{30}(12\bullet_c + 20\bullet_d - 2\bigcirc_2)\right) \end{aligned}$$

$$
\begin{aligned}
&= \frac{1}{2}\left(\frac{2}{5}\bullet_a + \frac{2}{3}\bullet_b + \frac{2}{5}\bullet_c + \frac{2}{3}\bullet_d + \frac{-1}{15}\left(\frac{5}{32}\bullet_S + \frac{-3}{32}\bullet_N + \frac{15}{16}\bullet_P\right)\right. \\
&+ \left.\frac{-1}{15}\left(\frac{-3}{32}\bullet_S + \frac{5}{32}\bullet_N + \frac{15}{16}\bullet_P\right)\right) \\
&= \frac{1}{5}\bullet_a + \frac{1}{3}\bullet_b + \frac{1}{5}\bullet_c + \frac{1}{3}\bullet_d + \frac{-1}{480}\bullet_N + \frac{-1}{16}\bullet_P + \frac{-1}{480}\bullet_S
\end{aligned}
$$

and then

$$
\begin{aligned}
u_x(\times) &= \frac{1}{2h}\left(\bullet_P - \bigotimes\right) \\
&= \frac{1}{h}\left(\frac{-1}{10}\bullet_a + \frac{-1}{6}\bullet_b + \frac{-1}{10}\bullet_c + \frac{-1}{6}\bullet_d + \frac{1}{960}\bullet_N + \frac{17}{32}\bullet_P + \frac{1}{960}\bullet_S\right),
\end{aligned}
$$

but that is not quite the same as the flux matching formula (§4.3). Notice that $\frac{15}{960}ths$ of the weight from $\bullet_N$ and $\bullet_S$ has been shifted to $\bullet_P$ when compared with the flux matching formula.

4.2 Stencil Version, Averaging Two Coarse Grid Sized Fluxes

Another alternative to the flux matching procedure that we implemented is to average two coarse grid sized fluxes instead of two fine grid sized fluxes. That is, compute the flux $u_x(\times)$ as the average of

$$u_x(\times_1) = \frac{1}{2h}\left(\bigcirc_1 - \circ_1\right)$$

and

$$u_x(\times_2) = \frac{1}{2h}\left(\bigcirc_2 - \circ_2\right)$$

where

$$u(\bigcirc_1) = \frac{5}{32}\bullet_S - \frac{3}{32}\bullet_N + \frac{15}{16}\bullet_P$$

and

$$u(\bigcirc_2) = \frac{-3}{32}\bullet_S + \frac{5}{32}\bullet_N + \frac{15}{16}\bullet_P$$

and

$$u(\circ_1) = \frac{1}{30}\left(12\bullet_a + 20\bullet_b - 2\bigcirc_1\right)$$

and

$$u(\circ_2) = \frac{1}{30}\left(12\bullet_c + 20\bullet_d - 2\bigcirc_2\right).$$

So

$$
\begin{aligned}
u_x(\times) &= \frac{1}{2}\left(u_x(\times_1) + u_x(\times_2)\right) \\
&= \frac{1}{4h}\left(\left(\bigcirc_1 - \frac{1}{30}\left(12\bullet_a + 20\bullet_b - 2\bigcirc_1\right)\right) + \left(\bigcirc_2 - \frac{1}{30}\left(12\bullet_c + 20\bullet_d - 2\bigcirc_2\right)\right)\right)
\end{aligned}
$$

$$
\begin{aligned}
&= \frac{1}{h}\left(\frac{4}{15}\bigcirc_1 - \frac{1}{10}\bullet_a - \frac{1}{6}\bullet_b + \frac{4}{15}\bigcirc_2 - \frac{1}{10}\bullet_c - \frac{1}{6}\bullet_d\right)\\
&= \frac{1}{h}\left(\frac{-1}{10}\bullet_a + \frac{-1}{6}\bullet_b + \frac{-1}{10}\bullet_c + \frac{-1}{6}\bullet_d + \frac{4}{15}\left(\frac{5}{32}\bullet_S - \frac{3}{32}\bullet_N + \frac{15}{16}\bullet_P\right)\right.\\
&+ \left.\frac{4}{15}\left(\frac{-3}{32}\bullet_S + \frac{5}{32}\bullet_N + \frac{15}{16}\bullet_P\right)\right)\\
&= \frac{1}{h}\left(\frac{-1}{10}\bullet_a + \frac{-1}{6}\bullet_b + \frac{-1}{10}\bullet_c + \frac{-1}{6}\bullet_d + \frac{1}{60}\bullet_N + \frac{1}{2}\bullet_P + \frac{1}{60}\bullet_S\right),
\end{aligned}
$$

which is identical to the flux matching formula (see the next section, §4.3). It is not as useful in the implementation, though, because it does not address the operator on the fine grid side. The ghost points that are computed for the original flux matching procedure are still needed in order to apply the stencil on the fine grid side, and those grid points constitue a large part of what is involved in computing the flux matching formula, so it would be absurdly inefficient to recompute the fluxes from scratch.

4.3 Flux Matching Version

Let $\bullet_a$ be the reference point, $\bullet_a = (0,0)$. We need to compute $\otimes_1$ and $\otimes_2$. For the $\bigcirc_1$ points, we derive a C^1 quadratic interpolation formula using

$$c_0 + c_1 x + c_2 x^2 = u(x),$$

where

$$
\begin{bmatrix} 1 & 0 & 0 \\ 1 & 2h & (2h)^2 \\ 1 & 4h & (4h)^2 \end{bmatrix}
\begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} =
\begin{bmatrix} \bullet_S \\ \bullet_P \\ \bullet_N \end{bmatrix}
\Rightarrow
\begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} =
\begin{bmatrix} \bullet_S \\ \frac{1}{h}\left(\bullet_P - \frac{1}{4}\bullet_N - \frac{3}{4}\bullet_S\right) \\ \frac{1}{h^2}\left(\frac{1}{8}\bullet_N - \frac{1}{4}\bullet_P + \frac{1}{8}\bullet_S\right) \end{bmatrix}.
$$

Then

$$u\left(\bigcirc_1\right) = c_0 + c_1\left(\frac{3}{2}h\right) + c_2\left(\frac{3}{2}h\right)^2 = \frac{5}{32}\bullet_S - \frac{3}{32}\bullet_N + \frac{15}{16}\bullet_P,$$

and

$$u\left(\bigcirc_2\right) = c_0 + c_1\left(\frac{5}{2}h\right) + c_2\left(\frac{5}{2}h\right)^2 = \frac{-3}{32}\bullet_S + \frac{5}{32}\bullet_N + \frac{15}{16}\bullet_P,$$

For the $\otimes$ points, we derive a C^1 quadratic interpolation formula using

$$c_0 + c_1 x + c_2 x^2 = u(x),$$

where

$$
\begin{bmatrix} 1 & 0 & 0 \\ 1 & h & h^2 \\ 1 & \frac{5}{2}h & \left(\frac{5}{2}h\right)^2 \end{bmatrix}
\begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} =
\begin{bmatrix} \bullet_a \\ \bullet_b \\ \bigcirc_i \end{bmatrix}
\Rightarrow
\begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} =
\begin{bmatrix} \bullet_a \\ -\frac{1}{15h}\left(21\bullet_a - 25\bullet_b + 4\bigcirc_i\right) \\ \frac{2}{15h^2}\left(3\bullet_a - 5\bullet_b + 2\bigcirc_i\right) \end{bmatrix},
$$

so

$$\begin{aligned} u(\otimes_1) &= c_0 + c_1\,(2h) + c_2\,(2h)^2 \\ &= \frac{-1}{5}\bullet_a + \frac{8}{15}\bigcirc_1 + \frac{2}{3}\bullet_b \\ &= -\frac{1}{5}\bullet_a + \frac{2}{3}\bullet_b + \frac{1}{12}\bullet_S - \frac{1}{20}\bullet_N + \frac{1}{2}\bullet_P \end{aligned}$$

and

$$\begin{aligned} u(\otimes_2) &= c_0 + c_1\,(2h) + c_2\,(2h)^2 \\ &= \frac{-1}{5}\bullet_c + \frac{8}{15}\bigcirc_2 + \frac{2}{3}\bullet_d \\ &= -\frac{1}{5}\bullet_c + \frac{2}{3}\bullet_d - \frac{1}{20}\bullet_S + \frac{1}{12}\bullet_N + \frac{1}{2}\bullet_P. \end{aligned}$$

so

$$\begin{aligned} u_x\,(\times) &= \frac{1}{2h}\left(u(\otimes_1) - \bullet_b + u(\otimes_2) - \bullet_d\right) \\ &= \frac{1}{h}\left(\frac{-1}{10}\bullet_a + \frac{-1}{6}\bullet_b + \frac{1}{24}\bullet_S + \frac{-1}{40}\bullet_N + \frac{1}{4}\bullet_P\right. \\ &+ \left.\frac{-1}{10}\bullet_c + \frac{-1}{6}\bullet_d + \frac{-1}{40}\bullet_S + \frac{1}{24}\bullet_N + \frac{1}{4}\bullet_P\right) \\ &= \frac{1}{h}\left(\frac{-1}{10}\bullet_a + \frac{-1}{6}\bullet_b + \frac{-1}{10}\bullet_c + \frac{-1}{6}\bullet_d + \frac{1}{60}\bullet_N + \frac{1}{2}\bullet_P + \frac{1}{60}\bullet_S\right) \end{aligned}$$

This is a single simple expression for computing the approximation of the flux that the flux matching process gives. It is not used in the code, though, for the same reason as in §4.2. The ghost points that are needed in order to apply the stencil on the fine grid side constitue a large part of what is involved in computing this flux matching formula, so it would be absurdly inefficient to recompute the fluxes from scratch.

4.4 Computing Flux Across Interface Directly Via Taylor's Series

There are a variety of ways to compute the fluxes using Taylor series. This section examines three ways. The main difference is in the number of grid points that are used. We begin with a case using only a few grid points and end up uith an example that uses all of the same grid points that are used in the flux matching formula (including those used in the ghost point computation) and show how the flux matching formula can be derived from Taylor series expansions. We assume, throughout this section, the boundedness of all relevant derivatives when using the big-O notation to express the order of the truncation error.

Note that, in this section, we more use rigorous notation than in the other sections, because it is helpful for clarity when writing out the Taylor series expansions. There are new illustrations to facilitate this, with the grid points labelled with indices. After each derivation, the corresponding expression in the notation from Fig. 4.1 is be given.

4.4.1 Taylor Series Expansion With Three Points

The most natural method for computing the flux across the interface is to use Taylor series expansions with respect to the two adjacent fine grid points and the one adjacent coarse grid point as illustrated in Fig. 4.2. For the south flux,

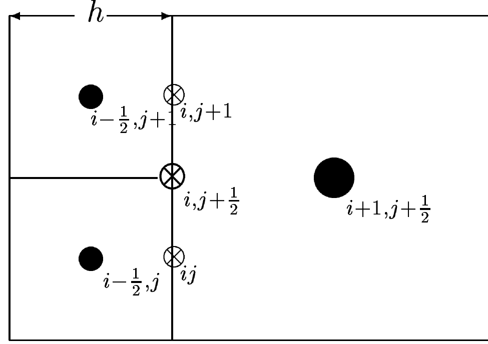


Figure 4.2: Interface, Small

$$\left.\begin{array}{l} u_{i-\frac{1}{2},j} = u_{ij} - \frac{1}{2}hu_x + 0 + \frac{1}{8}h^2u_{xx} + 0 + \cdots \\ u_{i+1,j+\frac{1}{2}} = u_{ij} + hu_x + \frac{1}{2}hu_y + \frac{1}{2}h^2u_{xx} + \frac{1}{8}h^2u_{yy} + \cdots \\ u_{i-\frac{1}{2},j+1} = u_{ij} - \frac{1}{2}hu_x + hu_y + \frac{1}{8}h^2u_{xx} + \frac{1}{2}h^2u_{yy} + \cdots \end{array}\right\} \Longrightarrow$$

$$2u_{i+1,j+\frac{1}{2}} - u_{i-\frac{1}{2},j} - u_{i-\frac{1}{2},j+1} = 3hu_x(\otimes_{ij}) + \frac{3}{4}h^2u_{xx} - \frac{1}{4}h^2u_{yy}$$

so

$$u_x(\otimes_{ij}) = \frac{2u_{i+1,j+\frac{1}{2}} - u_{i-\frac{1}{2},j} - u_{i-\frac{1}{2},j+1}}{3h} + O(h^2).$$

For the north flux,

$$\left.\begin{array}{l} u_{i-\frac{1}{2},j} = u_{i,j+1} - \frac{1}{2}hu_x - hu_y + \frac{1}{8}h^2u_{xx} + \frac{1}{2}h^2u_{yy} + \cdots \\ u_{i+1,j+\frac{1}{2}} = u_{i,j+1} + hu_x - \frac{1}{2}hu_y + \frac{1}{2}h^2u_{xx} + \frac{1}{8}h^2u_{yy} + \cdots \\ u_{i-\frac{1}{2},j+1} = u_{i,j+1} - \frac{1}{2}hu_x + 0 + \frac{1}{8}h^2u_{xx} + 0 + \cdots \end{array}\right\} \Longrightarrow$$

$$2u_{i+1,j+\frac{1}{2}} - u_{i-\frac{1}{2},j} - u_{i-\frac{1}{2},j+1} = 3hu_x(\otimes_{i,j+1}) + \frac{3}{4}h^2u_{xx} - \frac{1}{4}h^2u_{yy}$$

so

$$u_x(\otimes_{i,j+1}) = \frac{2u_{i+1,j+\frac{1}{2}} - u_{i-\frac{1}{2},j} - u_{i-\frac{1}{2},j+1}}{3h} + O(h^2).$$

Symmetry gives the same formula for both of the fine fluxes. The error is very large because of this, as illustrated in 4.6. There is nothing to do about it, though. Both fine grid points are needed in order to cancel out the u_y term, so there is no way to obtain asymmetric expansions for the two fine fluxes. The next section explores possibilities with a couple more grid points worth of information to work with.

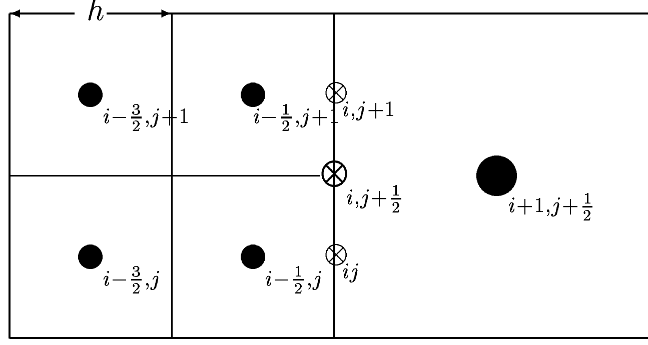


Figure 4.3: Interface, Medium

$\bullet_{i-\frac{3}{2},j+1}$	$\bullet_{i-\frac{1}{2},j+1}$	$\bullet_{i+\frac{1}{2},j+\frac{1}{2}}$	$\bullet_{i-\frac{1}{2},j}$	$\bullet_{i-\frac{3}{2},j}$	u_x	u_{xx}	u_{yy}
-1	-1	4	-1	-1	$8h$	$-\frac{1}{2}h^2$	$-\frac{1}{2}h^2$
$-\frac{1}{2}$	-1	3	-1	$-\frac{1}{2}$	$\frac{11}{2}h$	$\frac{1}{8}h^2$	$-\frac{3}{8}h^2$
-1	$-\frac{1}{2}$	3	$-\frac{1}{2}$	-1	$\frac{13}{2}h$	$-\frac{7}{8}h^2$	$-\frac{3}{8}h^2$
-1	$-\frac{1}{2}$	3	-1	$-\frac{1}{2}$	$6h$	$-\frac{3}{8}h^2$	$-\frac{3}{8}h^2$
$-\frac{1}{2}$	-1	3	$-\frac{1}{2}$	-1	$6h$	$-\frac{3}{8}h^2$	$-\frac{3}{8}h^2$

Table 4.1: Combinations for expansions of points in Fig. 4.3

4.4.2 Taylor Series Expansion With Five Points

In this section, a little more information from the fine grid is added, as shown in Fig. 4.3. If we use all the points as in the last section, symmetry gives the same formula for both of the fine fluxes, $u_x(\otimes_{ij})$ and $u_x(\otimes_{i,j+1})$. Consider the $u_x(\otimes_{ij})$ case. The expansions are

$$\left.\begin{array}{l} u_{i-\frac{3}{2},j} = u_{ij} - \frac{3}{2}hu_x + hu_y + \frac{9}{8}h^2u_{xx} + \frac{1}{2}h^2u_{yy} + \cdots \\ u_{i-\frac{1}{2},j} = u_{ij} - \frac{1}{2}hu_x + hu_y + \frac{1}{8}h^2u_{xx} + \frac{1}{2}h^2u_{yy} + \cdots \\ u_{i+1,j+\frac{1}{2}} = u_{ij} + hu_x + \frac{1}{2}hu_y + \frac{1}{2}h^2u_{xx} + \frac{1}{8}h^2u_{yy} + \cdots \\ u_{i-\frac{3}{2},j+1} = u_{ij} - \frac{1}{2}hu_x + 0 + \frac{1}{8}h^2u_{xx} + 0 + \cdots \\ u_{i-\frac{1}{2},j+1} = u_{ij} - \frac{3}{2}hu_x + 0 + \frac{9}{8}h^2u_{xx} + 0 + \cdots \end{array}\right\}.$$

There are several ways to combine these to get a valid formula. Table 4.1 shows the different combinations up to the second order terms. The second one has the smallest error:

$$-\frac{1}{2}u_{i-\frac{3}{2},j+1} - u_{i-\frac{1}{2},j+1}3u_{i+1,j+\frac{1}{2}} - u_{i-\frac{1}{2},j} - \frac{1}{2}u_{i-\frac{3}{2},j} = \frac{11}{2}hu_x + \frac{1}{8}h^2u_{xx} - \frac{3}{8}h^2u_{yy} \Longrightarrow$$

$$u_x(\otimes_{ij}) = \frac{2}{11h}\left(-\frac{1}{2}u_{i-\frac{3}{2},j+1} - u_{i-\frac{1}{2},j+1} + 3u_{i+1,j+\frac{1}{2}} - u_{i-\frac{1}{2},j} - \frac{1}{2}u_{i-\frac{3}{2},j}\right) + O(h^2).$$

The error terms are smaller than in the last section, but still of the same order.

We do not have to use all of the points to approximate each flux, now, though. We can choose a different set of points for the different fluxes. For the south flux, we could use

$$\left.\begin{array}{l} u_{i-\frac{1}{2},j} = u_{ij} - \frac{1}{2}hu_x + 0 + \frac{1}{8}h^2u_{xx} + 0 + \cdots \\ u_{i+1,j+\frac{1}{2}} = u_{ij} + hu_x + \frac{1}{2}hu_y + \frac{1}{2}h^2u_{xx} + \frac{1}{8}h^2u_{yy} + \cdots \\ u_{i-\frac{3}{2},j+1} = u_{ij} - \frac{3}{2}hu_x + hu_y + \frac{9}{8}h^2uxx + \frac{1}{2}h^2u_{yy} + \cdots \end{array}\right\} \Longrightarrow$$

$$2u_{i+1,j+\frac{1}{2}} - u_{i-\frac{1}{2},j} - u_{i-\frac{3}{2},j+1} = 4hu_x - \frac{1}{4}h^2u_{xx} - \frac{1}{4}h^2u_{yy} \Longrightarrow$$

$$u_x(\otimes_{ij}) = \frac{2u_{i+1,j+\frac{1}{2}} - u_{i-\frac{1}{2},j} - u_{i-\frac{3}{2},j+1}}{4h} + O(h^2).$$

And then for the north flux, we could use

$$\left.\begin{array}{l} u_{i-\frac{3}{2},j} = u_{i,j+1} - \frac{3}{2}hu_x - hu_y + \frac{9}{8}h^2u_{xx} + \frac{1}{2}h^2u_{yy} + \cdots \\ u_{i+1,j+\frac{1}{2}} = u_{i,j+1} + hu_x - \frac{1}{2}hu_y + \frac{1}{2}h^2u_{xx} + \frac{1}{8}h^2u_{yy} + \cdots \\ u_{i-\frac{1}{2},j+1} = u_{i,j+1} - \frac{1}{2}hu_x + 0 + \frac{1}{8}h^2u_{xx} + 0 + \cdots \end{array}\right\} \Longrightarrow$$

$$2u_{i+1,j+\frac{1}{2}} - u_{i-\frac{3}{2},j} - u_{i-\frac{1}{2},j+1} = 4hu_x - \frac{1}{4}h^2u_{xx} - \frac{1}{4}h^2u_{yy} \Longrightarrow$$

$$u_x(\otimes_{i,j+1}) = \frac{2u_{i+1,j+\frac{1}{2}} - u_{i-\frac{3}{2},j} - u_{i-\frac{1}{2},j+1}}{4h} + O(h^2).$$

Then the average is

$$\frac{1}{2}\left(u_x\left(\otimes_{i,j}\right) + u_x\left(\otimes_{i,j+1}\right)\right) = \frac{1}{8h}\left(4u_{i+1,j+\frac{1}{2}} - u_{i-\frac{3}{2},j+1} - u_{i-\frac{1}{2},j+1} - u_{i-\frac{3}{2},j} - u_{i-\frac{1}{2},j}\right).$$

This can be rewritten in the notation from Fig. 4.1 as

$$u_x(\times) = \frac{1}{8h}\left(4\bullet_P - \bullet_a - \bullet_b - \bullet_c - \bullet_d\right).$$

This gives unique approximations for both fine fluxes, but the error is still comparable to the previous approximations. We will have to go ahead and include expansions of all of the points from the flux matching formulation in order to see substantial improvement of the truncation error.

4.4.3 Taylor Series Expansion With Seven Points

In this section, we use additional coarse grid data, as shown in Fig. 4.4, and we choose the weighting for the expansions specifically for the purpose of obtaining exactly the flux matching formula. For the south flux, $u_x(\otimes_{ij})$,

$$\left.\begin{array}{l} u_{i-\frac{3}{2},j} = u_{ij} - \frac{3}{2}hu_x + 0 + \frac{27}{8}h^2u_{xx} + 0 - \frac{9}{48}h^3u_{xxx} + 0 + \cdots \\ u_{i-\frac{1}{2},j} = u_{ij} - \frac{1}{2}hu_x + 0 + \frac{1}{8}h^2u_{xx} + 0 - \frac{1}{48}h^3u_{xxx} + 0 + \cdots \\ u_{i+1,j+\frac{5}{2}} = u_{ij} + hu_x + \frac{5}{2}hu_y + \frac{1}{2}h^2u_{xx} + \frac{25}{8}u_{yy} + \frac{1}{6}h^3u_{xxx} + \frac{125}{48}h^3u_{yyy} + \cdots \\ u_{i+1,j+\frac{1}{2}} = u_{ij} + hu_x + \frac{1}{2}hu_y + \frac{1}{2}h^2u_{xx} + \frac{1}{8}h^2u_{yy} + \frac{1}{6}h^3u_{xxx} + \frac{1}{48}h^3u_{yyy} + \cdots \\ u_{i+1,j-\frac{3}{2}} = u_{ij} + hu_x - \frac{3}{2}hu_y + \frac{1}{2}h^2u_{xx} + \frac{9}{8}h^2u_{yy} + \frac{1}{6}h^3u_{xxx} - \frac{27}{48}h^3u_{yyy} + \cdots \end{array}\right\} \Longrightarrow$$

$$-12u_{i-\frac{3}{2},j} - 20u_{i-\frac{1}{2},j} - 3u_{i+1,j+\frac{5}{2}} + 30u_{i+1,j+\frac{1}{2}} + 5u_{i+1,j-\frac{3}{2}} = 60hu_x + \frac{25}{2}h^3u_{xxx} - 10h^3u_{yyy} \Longrightarrow$$

$$u_x\left(\otimes_{ij}\right) = \frac{1}{60h}\left(-12u_{i-\frac{3}{2},j} - 20u_{i-\frac{1}{2},j} - 3u_{i+1,j+\frac{5}{2}} + 30u_{i+1,j+\frac{1}{2}} + 5u_{i+1,j-\frac{3}{2}}\right) + O(h^3) \Longrightarrow$$

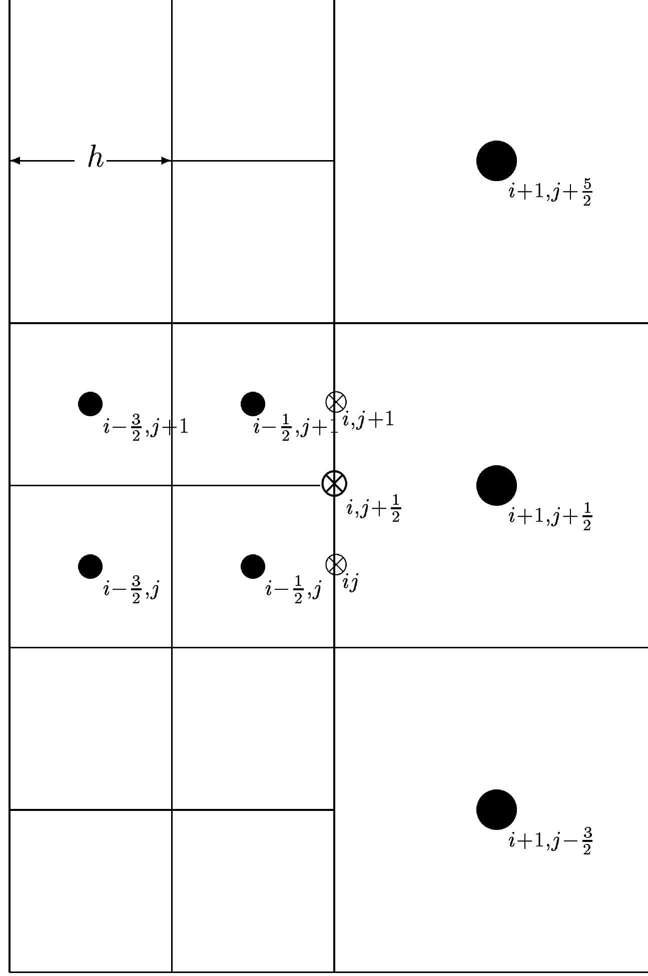


Figure 4.4: Interface, Large

$$u_x(\otimes_{ij}) = \frac{1}{h}\left(\frac{-1}{5}u_{i-\frac{3}{2},j} + \frac{-1}{3}u_{i-\frac{1}{2},j} + \frac{-1}{20}u_{i+1,j+\frac{5}{2}} + \frac{1}{2}u_{i+1,j+\frac{1}{2}} + \frac{1}{12}u_{i+1,j-\frac{3}{2}}\right) + O(h^3).$$

For the north flux, $u_x(\otimes_{i,j+1})$,

$$\left.\begin{array}{l}
u_{i+1,j+\frac{5}{2}} = u_{i,j+1} + hu_x + \frac{3}{2}hu_y + \frac{1}{2}h^2u_{xx} + \frac{9}{8}h^2u_{yy} + \frac{1}{6}h^3u_{xxx} + \frac{27}{48}h^3u_{yyy} + \cdots \\
u_{i+1,j+\frac{1}{2}} = u_{i,j+1} + hu_x - \frac{1}{2}hu_y + \frac{1}{2}h^2u_{xx} + \frac{1}{8}h^2u_{yy} + \frac{1}{6}h^3u_{xxx} - \frac{1}{48}h^3u_{yyy} + \cdots \\
u_{i+1,j-\frac{3}{2}} = u_{i,j+1} + hu_x - \frac{5}{2}hu_y + \frac{1}{2}h^2u_{xx} + \frac{25}{8}h^2u_{yy} + \frac{1}{6}h^3u_{xxx} - \frac{125}{48}h^3u_{yyy} + \cdots \\
u_{i-\frac{3}{2},j+1} = u_{i,j+1} - \frac{3}{2}hu_x + 0 + \frac{9}{8}h^2u_{xx} + 0 - \frac{27}{48}h^3u_{xxx} + 0 + \cdots \\
u_{i-\frac{1}{2},j+1} = u_{i,j+1} - \frac{1}{2}hu_x + 0 + \frac{1}{8}h^2u_{xx} + 0 - \frac{1}{48}h^3u_{xxx} + 0 + \cdots
\end{array}\right\} \Longrightarrow$$

$$+5u_{i+1,j+\frac{5}{2}} + 30u_{i+1,j+\frac{1}{2}} - 3u_{i+1,j-\frac{3}{2}} - 12u_{i-\frac{3}{2},j+1} - 20u_{i-\frac{1}{2},j+1} = 60hu_x + \frac{25}{2}h^3u_{xxx} + 10h^3u_{yyy} \Longrightarrow$$

$$u_x(\otimes_{i,j+1}) = \frac{1}{60h}\left(+5u_{i+1,j+\frac{5}{2}} + 30u_{i+1,j+\frac{1}{2}} - 3u_{i+1,j-\frac{3}{2}} - 12u_{i-\frac{3}{2},j+1} - 20u_{i-\frac{1}{2},j+1}\right) + O(h^3) \Longrightarrow$$

$$u_x(\otimes_{i,j+1}) = \frac{1}{h}\left(\frac{1}{12}u_{i+1,j+\frac{5}{2}} + \frac{1}{2}u_{i+1,j+\frac{1}{2}} + \frac{-1}{20}u_{i+1,j-\frac{3}{2}} + \frac{-1}{5}u_{i-\frac{3}{2},j+1} + \frac{-1}{3}u_{i-\frac{1}{2},j+1}\right) + O(h^3).$$

Then the average is

$$
\begin{aligned}
\frac{1}{2}\left(u_x\left(\otimes_{i,j}\right)+u_x\left(\otimes_{i,j+1}\right)\right) &= \frac{1}{2h}\left(\frac{-1}{5}u_{i-\frac{3}{2},j}+\frac{-1}{3}u_{i-\frac{1}{2},j}+\frac{-1}{20}u_{i+1,j+\frac{5}{2}}+\frac{1}{2}u_{i+1,j+\frac{1}{2}}+\frac{1}{12}u_{i+1,j-\frac{3}{2}}\right) \\
&+ \frac{1}{2h}\left(\frac{1}{12}u_{i+1,j+\frac{5}{2}}+\frac{1}{2}u_{i+1,j+\frac{1}{2}}+\frac{-1}{20}u_{i+1,j-\frac{3}{2}}+\frac{-1}{5}u_{i-\frac{3}{2},j+1}+\frac{-1}{3}u_{i-\frac{1}{2},j+1}\right) \\
&= \frac{1}{h}\left(\frac{-1}{10}u_{i-\frac{3}{2},j}+\frac{-1}{6}u_{i-\frac{1}{2},j}+\frac{-1}{40}u_{i+1,j+\frac{5}{2}}+\frac{1}{4}u_{i+1,j+\frac{1}{2}}+\frac{1}{24}u_{i+1,j-\frac{3}{2}}\right. \\
&+ \left.\frac{1}{24}u_{i+1,j+\frac{5}{2}}+\frac{1}{4}u_{i+1,j+\frac{1}{2}}+\frac{-1}{40}u_{i+1,j-\frac{3}{2}}+\frac{-1}{10}u_{i-\frac{3}{2},j+1}+\frac{-1}{6}u_{i-\frac{1}{2},j+1}\right) \\
&= \frac{1}{h}\left(\frac{-1}{10}u_{i-\frac{3}{2},j}+\frac{-1}{6}u_{i-\frac{1}{2},j}+\frac{-1}{10}u_{i-\frac{3}{2},j+1}+\frac{-1}{6}u_{i-\frac{1}{2},j+1}\right. \\
&+ \left.\frac{1}{60}u_{i+1,j+\frac{5}{2}}+\frac{1}{2}u_{i+1,j+\frac{1}{2}}+\frac{1}{60}u_{i+1,j-\frac{3}{2}}\right).
\end{aligned}
$$

This can be rewritten in the notation from Fig. 4.1 as

$$
u_x(\times)=\frac{1}{h}\left(\frac{-1}{10}\bullet_a+\frac{-1}{6}\bullet_b+\frac{-1}{10}\bullet_c+\frac{-1}{6}\bullet_d+\frac{1}{60}\bullet_N+\frac{1}{2}\bullet_P+\frac{1}{60}\bullet_S\right)
$$

which is exactly the flux matching formula as derived in §4.3. Notice how the second order terms cancel, thus improving the truncation error substantially over that of the other approximations.

4.4.4 Example

Here we compute the approximation of the fluxes across an interface and compare the results from the small Taylor series expasion from §4.4.1 and the flux matching formula of §4.3.

The coarse grid is 64×64 and the fine grid is 64×128. The fine grid is located all the way to one side of the coarse grid, so the interface runs down the entire length of the middle of the composite grid. The solutions function is $\sin(1.5\pi x)\sin(2\pi y)$ so the true flux across the interface (at $x = .5$) is $1.5\pi\cos(.75\pi)\sin(2\pi y)$.

First, Fig. 4.5 shows the error in the ghost point interpolation process which is the first step in the flux matching process.

The top frames in Fig. 4.6 show the error in the fluxes as approximated by the two procedures. The error is shown for the fine side fluxes and the coarse side fluxes. The red line is flux matching procedure, and the green line is the small Taylor series expansion. The bottom frames show the actual approximated fluxes. The true fluxes are plotted here in black, but it is mostly covered by the flux matching approximation.

We expect the error for the small Taylor series approach to be larger than flux matching based on the analysis of the previous sections, and the plots show that it is. In fact, the error in the approximation by the small Taylor series expansion is not only very large compared to the error in the flux matching approximation but is very oscillatory also, in the case of the fine side fluxes.

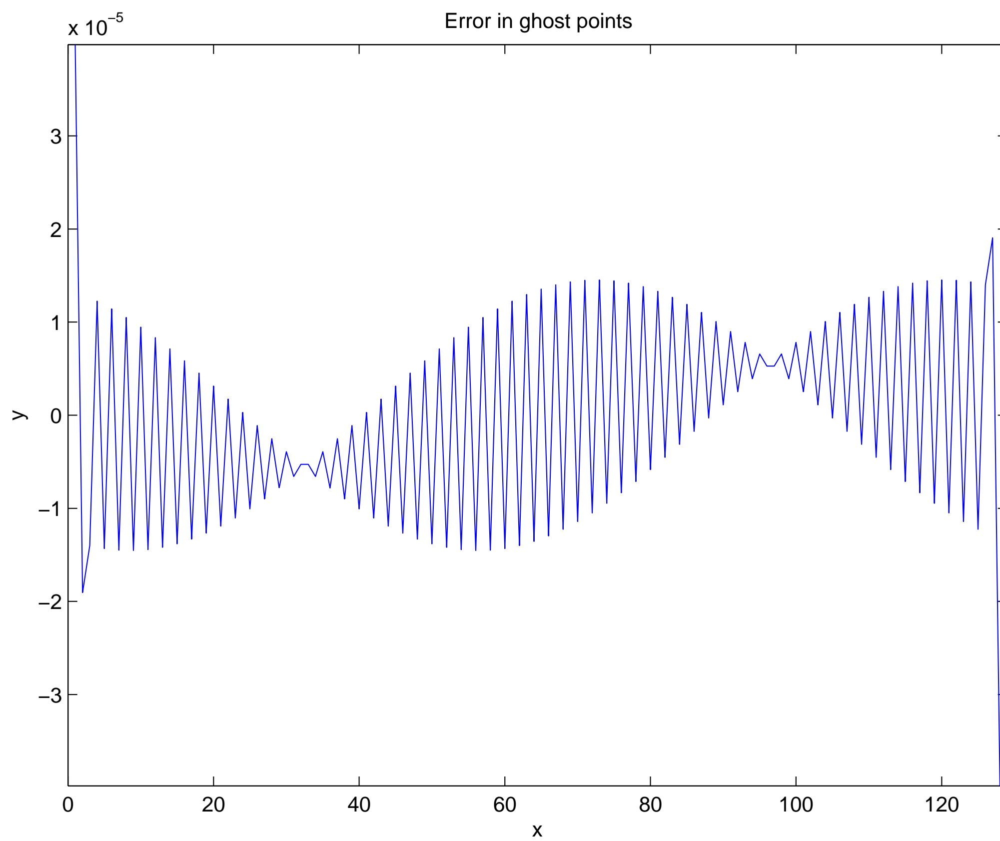


Figure 4.5: Error in the ghost point interpolation.

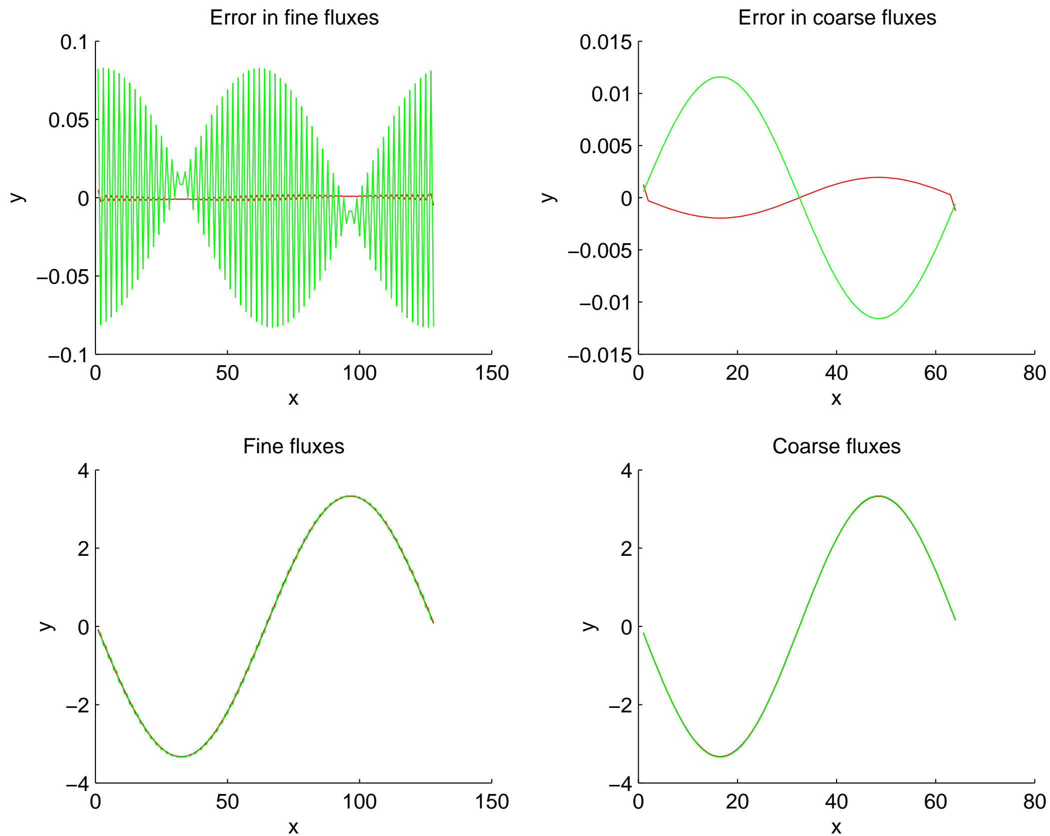


Figure 4.6: Error in the fluxes. The red line is for the flux matching procedure, and the green line is for the small Taylor series expansion. The true fluxes are plotted in black in the two lower plots, but it is mostly covered by the flux matching approximation.

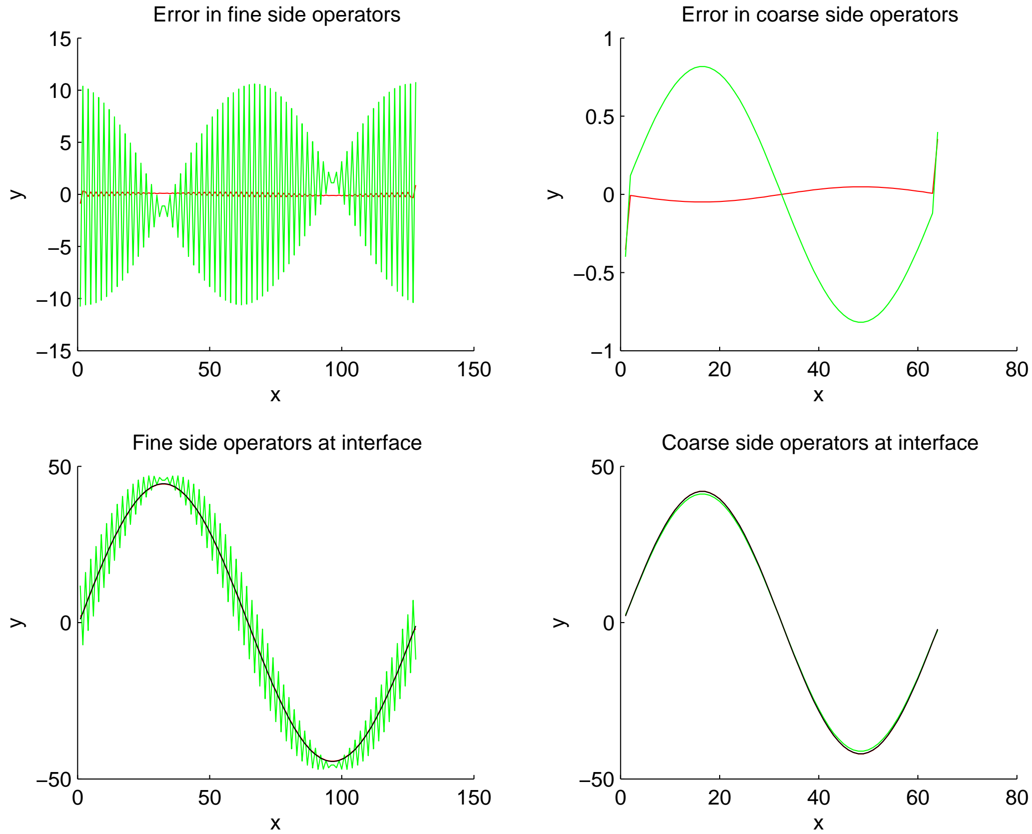


Figure 4.7: Error in the operator. The red line is for the flux matching based approximation to the operator, and the green line is for the small Taylor series expansion base approximation to the operator. The true values of the operator are plotted in black in the two lower plots, and they coincide very closely with the red line as in the flux plot.

This oscillation is exposed even more dramatically in the error for the operator using these flux approximations. The error in the operator is shown in the upper frames of Fig. 4.7. The true values of the operator are plotted in black in the lower frames. Again, it coincides very closely with the flux matching based approximation.

Fig. 4.8 shows the behavior of the error in the approximate fluxes as the grid resolution increases. The error decreases in all cases, and the error in the flux matching version decreases faster, as expected. The horizontal axis is the grid size measured on the coarse grid and ranges over coarse grids of size 8×8 to $2^{10} \times 2^{10}$ with the dimensions doubling at each step.

Fig. 4.9 shows the behavior of the error in the approximate operator values as the grid resolution increases. The grid resolutions are the same as in Fig. 4.8. The flux matching base approximations decrease nicely. Surprisingly, the small Taylor series based approximations do not decrease. This dispels any doubt about the merit of the more complicated flux matching procedure over the simple approach.

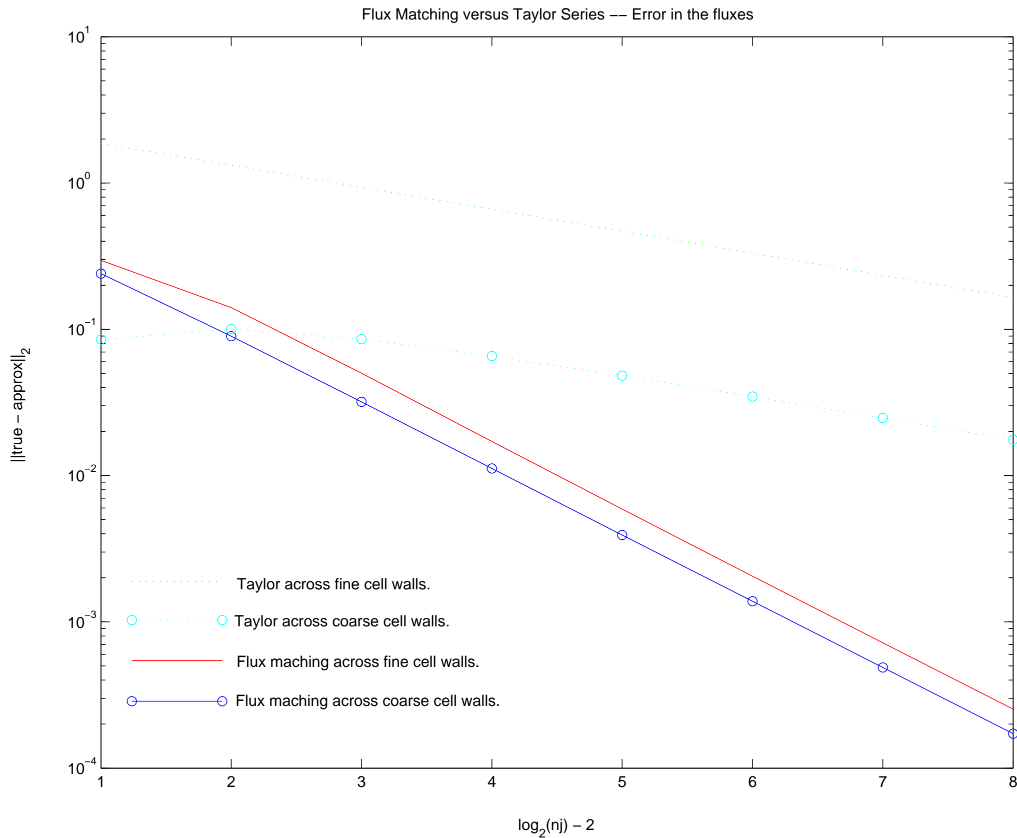


Figure 4.8: Error in the fluxes as grid size increases. The horizontal axis is the grid size measured on the coarse grid and ranges over coarse grids of size 8×8 to $2^{10} \times 2^{10}$ with the dimensions doubling at each step.

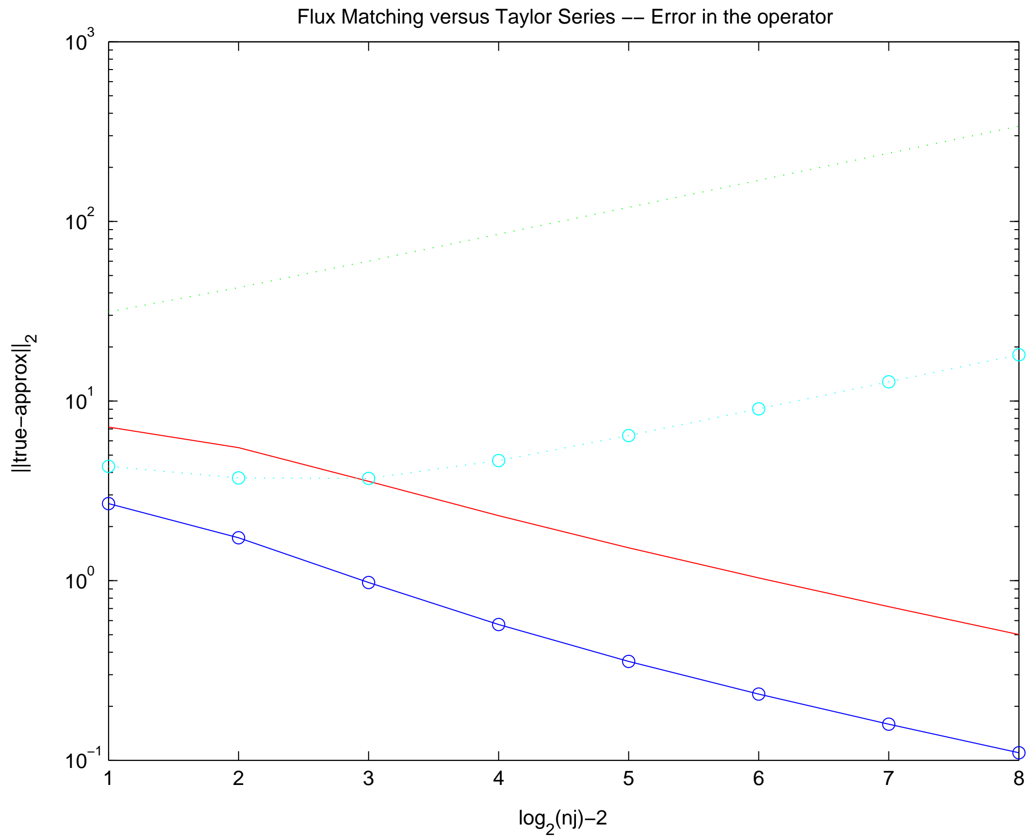


Figure 4.9: Error in the operator as grid size increases. The horizontal axis is the grid size measured on the coarse grid and ranges over coarse grids of size 8×8 to $2^{10} \times 2^{10}$ with the dimensions doubling at each step.

Chapter 5

Tools for 3D AMRMG

This chapter outlines the machinery needed to implement the AMRMG algorithm in 3D for both constant and variable coefficient.

5.1 Tools For Constant Coefficient Problems

This section illustrates the tools needed for the 3D AMRMG algorithm using an example of a constant coefficient Poisson problem

$$\Delta\phi = \rho$$

on a cubic domain

$$\Omega = [0,1]^3$$

with Dirichlet boundary conditions

$$\phi = \gamma \text{ on } \partial\Omega.$$

This material is analogous to the tools presented for the 2D example in Ch. 3.

5.1.1 3D Stencils

In the interior of Ω^ℓ, we have the standard seven point stencil:

$$(\Delta\phi)_{ijk} = \left(\phi_{i+1} + \phi_{i-1} + \phi_{j+1} + \phi_{j-1} + \phi_{k+1} + \phi_{k-1} - 6\phi_{ijk}\right) h^{-2}.$$

On the physical boundaries, we must incorporate boundary values. For instance, on the side boundary in the negative x-direction ($i = 0$), we have

$$(\Delta\phi)_{0jk} = \left(\frac{4}{3}\phi_1 + \frac{8}{3}\phi_{-\frac{1}{2}} + \phi_{j+1} + \phi_{j-1} + \phi_{k+1} + \phi_{k-1} - 8\phi_{0jk}\right) h^{-2},$$

at the edge boundary in the negative x-direction ($i = 0$) and positive y-direction ($j = N$) we have

$$(\Delta\phi)_{0Nk} = \left(\frac{4}{3}\phi_1 + \frac{8}{3}\phi_{-\frac{1}{2}} + \frac{8}{3}\phi_{N+\frac{1}{2}} + \frac{4}{3}\phi_{N-1} + \phi_{k+1} + \phi_{k-1} - 10\phi_{0Nk}\right) h^{-2},$$

and at the corner boundary where $(i,j,k) = (0,N,N)$, we have

$$(\Delta\phi)_{0NN} = \left(\frac{4}{3}\phi_1 + \frac{8}{3}\phi_{-\frac{1}{2}} + \frac{8}{3}\phi_{N+\frac{1}{2}} + \frac{4}{3}\phi_{N-1} + \frac{8}{3}\phi_{N+\frac{1}{2}} + \frac{4}{3}\phi_{N-1} - 12\phi_{0NN}\right) h^{-2}.$$

Symmetry gives the rest of the boundary stencils trivially.

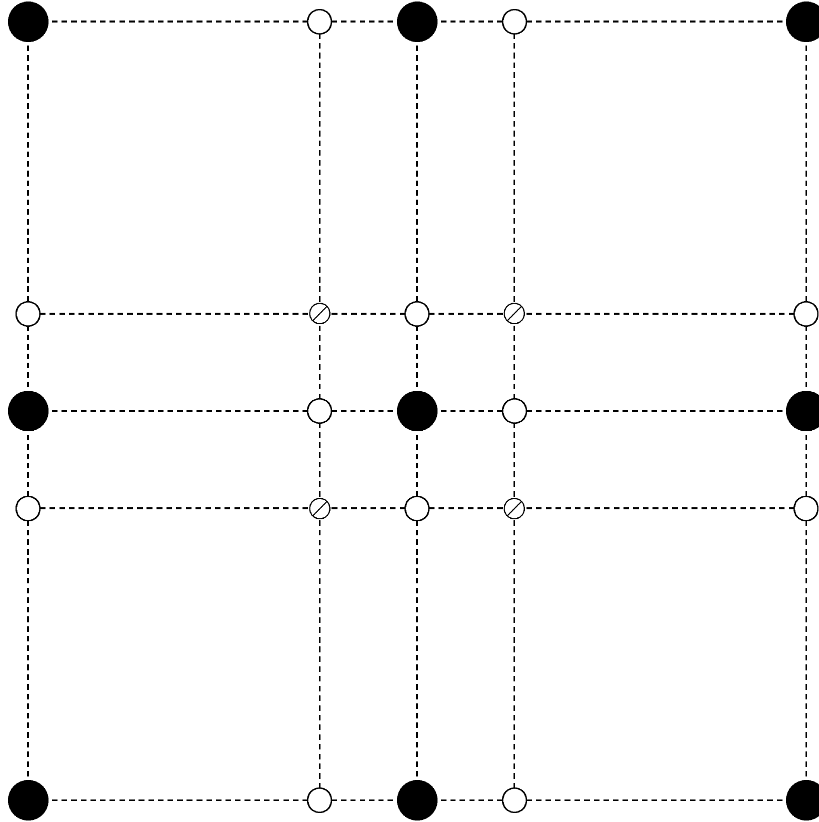


Figure 5.1: 3D Interpolation of Ghost Points, Step 1.

5.1.2 Interpolation of Ghost Points in 3D

Consider Figs. 5.1 and 5.2. Only one dimensional, quadratic interpolation and extrapolation is used. First (Fig. 5.1), coarse grid points are used to calculate values at the $\circ$ points, which are used in turn to calculate the values at the $\oslash$ points. The formula for these interpolations is the same as in *Step 1* of §3.1.2 for the 2D case of one coarse-fine interface. Second (Fig. 5.2), we use these $\oslash$ points and two existing fine grid points to get ghost point values at the $\otimes$ points, which correspond to where fine grid points would be if the fine grid extended that far. The formula for the interpolation of the $\otimes$ points is the same as in *Step 2* of §3.1.2 for the 2D case. Fig. 5.2 shows the stencil for the interpolation of just one of the four $\otimes$ points. The amount of illustration on the coarse grid side has been kept to a minimum with the intention of clarifying the context of the illustration without obfuscating its main purpose in conveying the $\otimes$ point interpolation stencil.

This procedure can be modified easily, in ways analogous to the 2D case, to accomodate cells that abut physical boundaries and cells that are adjacent to more than one coarse-fine interface.

See Ch. 6 for a more detailed look at the ghost point interpolation procedure and a derivation of a one step single formula for the $\otimes$ points.

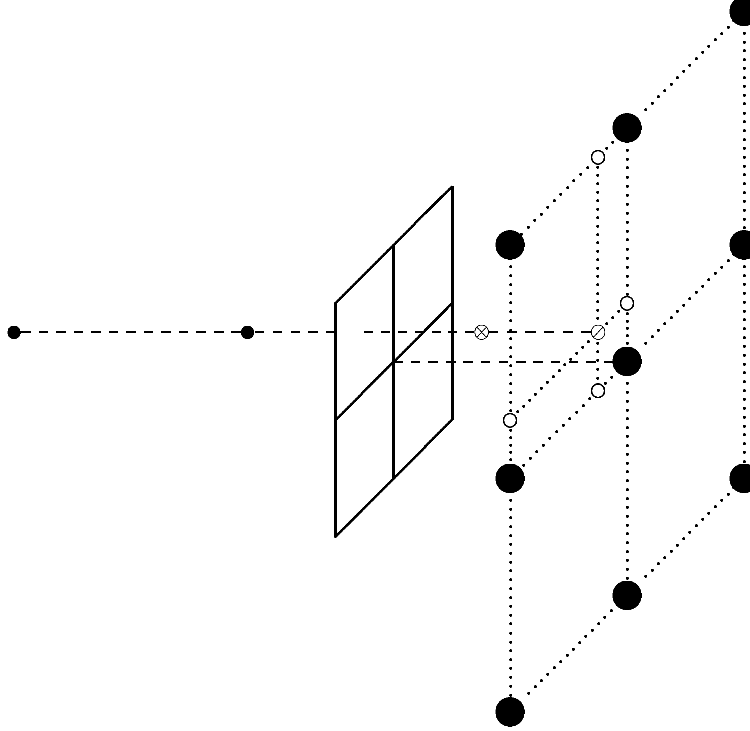


Figure 5.2: 3D Interpolation of Ghost Points, Step 2.

5.1.3 Flux Matching in 3D

Flux matching is used to avoid one-sided derivatives at grid points adjacent to refinement patches and preserve C^1 continuity across the interface between refinement levels. The computations in this section involve values interpolated at the ghost points $\otimes$ from §5.1.2. For the purposes of flux matching, the general form of the operator is written in terms of the fluxes across each of the six cell walls

$$(\Delta\phi^\ell)_{ijk} = \frac{1}{h_\ell}\left(f^\ell_{i+\frac{1}{2}} - f^\ell_{i-\frac{1}{2}} + f^\ell_{j+\frac{1}{2}} - f^\ell_{j-\frac{1}{2}} + f^\ell_{k+\frac{1}{2}} - f^\ell_{k-\frac{1}{2}}\right)$$

where

$$\begin{aligned}
f^\ell_{i+\frac{1}{2},j,k} &= (\phi^\ell_{i+1,j,k} - \phi^\ell_{ijk})h_\ell^{-1},\\
f^\ell_{i-\frac{1}{2},j,k} &= (\phi^\ell_{ijk} - \phi^\ell_{i-1,j,k})h_\ell^{-1},\\
f^\ell_{i,j+\frac{1}{2},k} &= (\phi^\ell_{i,j+1,k} - \phi^\ell_{ijk})h_\ell^{-1},\\
f^\ell_{i,j-\frac{1}{2},k} &= (\phi^\ell_{ijk} - \phi^\ell_{i,j-1,k})h_\ell^{-1},\\
f^\ell_{i,j,k+\frac{1}{2}} &= (\phi^\ell_{i,j,k+1} - \phi^\ell_{ijk})h_\ell^{-1},\\
f^\ell_{i,j,k-\frac{1}{2}} &= (\phi^\ell_{ijk} - \phi^\ell_{i,j,k-1})h_\ell^{-1}.
\end{aligned}$$

Suppose the coarse-fine interface is in the negative x-direction from the coarse grid point at which the operator is being applied. This is illustrated in Fig. 5.3. Then the flux across the interface is computed by

$$f^\ell_{i-\frac{1}{2},j,k} = \tfrac{1}{4}\left(\tfrac{1}{h^{\ell+1}}\left(\delta_1 + \delta_2 + \delta_3 + \delta_4\right)\right)$$

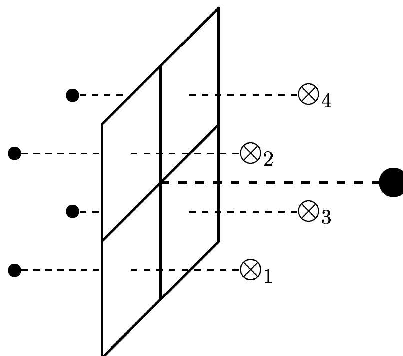


Figure 5.3: Flux Matching in 3D. The coarse grid point where the operator is being applied is shown for reference.

where

$$\begin{aligned}
\delta_1 &= \phi^{\ell+1}_{\otimes_1} - \phi^{\ell+1}_{2(i-1),2j-1,2k-1} \\
\delta_2 &= \phi^{\ell+1}_{\otimes_2} - \phi^{\ell+1}_{2(i-1),2j,2k-1} \\
\delta_3 &= \phi^{\ell+1}_{\otimes_3} - \phi^{\ell+1}_{2(i-1),2j-1,2k} \\
\delta_4 &= \phi^{\ell+1}_{\otimes_4} - \phi^{\ell+1}_{2(i-1),2j,2k}.
\end{aligned}$$

Application of the operator at cells that abut physical boundaries and cells that are adjacent to more than one coarse-fine interface are achieved by simple modifications analogous to those in the 2D case.

5.2 Tools For Variable Coefficient Problems

Here we show the tools needed for the 3D variable coefficient problem. This section uses the directions east, west, north, south, up, and down as the basis for subscripts in the stencils and formulae. East and west are the positive and negative x-directions. North and south are the positive and negative y-directions. Up and down are the positive and negative z-directions. The abbreviations e, w, n, s, u, and d are used as subscripts. The abbreviations E, W, N, S, U, and D are used as references to the faces of a cell and can be combined to refer to edges and corners (e.g., NW edge and DSW corner).

5.2.1 3D Stencils

This section gives a representative stencil for each case: interior, face, edge, and corner. The formulae for the other orientations of each case are symmetric.

The illustrations show the boundary regions of the control volume in each case. The axes are positioned to correspond to the DSW corner of the cell (see the above introduction to §5.2 for an explanation of the notation).

Each term on the left hand side of the discretizations comes from approximating the integral of the normal over one of the regions. The term on the right hand side comes from approximating the integral over the entire control volume.

Interior

See Fig. 5.4 for the integration regions corresponding to a control volume of an interior cell.

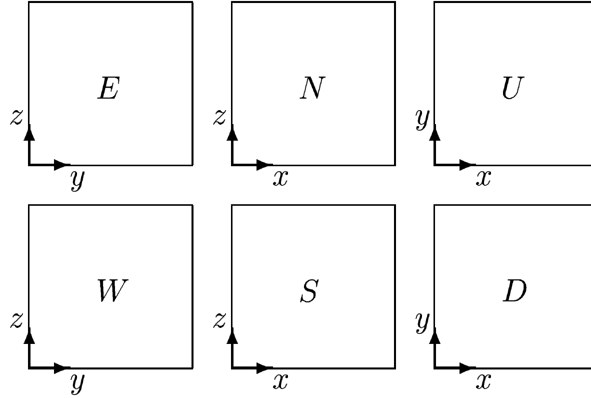


Figure 5.4: Integration regions for variable coefficient stencil compution for interior cells. These are the boundaries of the control volume.

Discretizing the integral form

$$-\int_{\partial V} a\frac{\partial \phi}{\partial n} = \int_V \rho$$

gives

$$\begin{array}{ccc}
\frac{a_n(\phi_n - \phi_o)}{\Delta y} \cdot \Delta x \Delta z & - & \frac{a_s(\phi_o - \phi_s)}{\Delta y} \cdot \Delta x \Delta z + \\
\frac{a_e(\phi_e - \phi_o)}{\Delta x} \cdot \Delta y \Delta z & - & \frac{a_w(\phi_o - \phi_w)}{\Delta x} \cdot \Delta y \Delta z + \\
\frac{a_u(\phi_u - \phi_o)}{\Delta z} \cdot \Delta x \Delta y & - & \frac{a_d(\phi_o - \phi_d)}{\Delta z} \cdot \Delta x \Delta y + \\
= -\Delta x \Delta y \Delta z \rho_o, & &
\end{array}$$

which reduces to

$$(a_n + a_s + a_e + a_w + a_u + a_d)\phi_o - a_n\phi_n - a_s\phi_s - a_w\phi_w - a_e\phi_e - a_u\phi_u - a_d\phi_d = h^2\rho_o,$$

when $h = \Delta x = \Delta y$.

Face

See Fig. 5.5 for the integration regions corresponding to a control volume for a representative face cell.

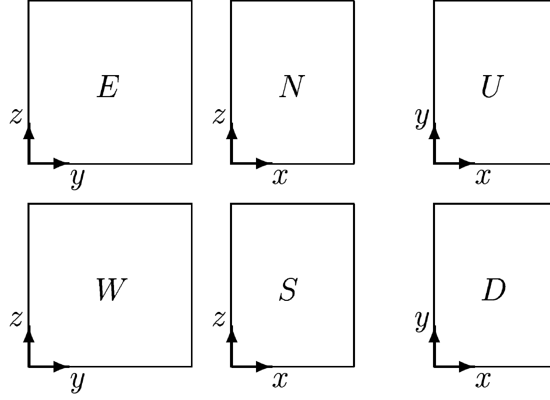


Figure 5.5: Integration regions for variable coefficient stencil compution for cells on the east face. These are the boundaries of the control volume.

Discretizing the integral form

$$-\int_{\partial V} a \frac{\partial \phi}{\partial n} = \int_V \rho$$

gives

$$\begin{aligned}
\frac{a_n(\phi_n - \phi_o)}{\Delta y} \cdot \frac{3}{4}\Delta x \Delta z \quad &- \quad \frac{a_s(\phi_o - \phi_s)}{\Delta y} \cdot \frac{3}{4}\Delta x \Delta z + \\
\frac{a_e(\phi_e - \phi_o)}{\frac{1}{2}\Delta x} \cdot \Delta y \Delta z \quad &- \quad \frac{a_w(\phi_o - \phi_w)}{\Delta x} \cdot \Delta y \Delta z + \\
\frac{a_u(\phi_u - \phi_o)}{\Delta z} \cdot \frac{3}{4}\Delta x \Delta y \quad &- \quad \frac{a_d(\phi_o - \phi_d)}{\Delta z} \cdot \frac{3}{4}\Delta x \Delta y + \\
= -\frac{3}{4}\Delta x \Delta y \Delta z \rho_o, &
\end{aligned}$$

which reduces to

$$\begin{aligned}
\left(a_n + a_s + \frac{8}{3}a_e + \frac{4}{3}a_w + a_u + a_d\right)\phi_o \quad &- \quad a_n\phi_n - a_s\phi_s \\
&- \quad \frac{4}{3}a_w\phi_w - \frac{8}{3}a_e\phi_e \\
&- \quad a_u\phi_u - a_d\phi_d = h^2\rho_o,
\end{aligned}$$

when $h = \Delta x = \Delta y$.

Edge

See Fig. 5.6 for the integration regions corresponding to a control volume for a representative edge cell.

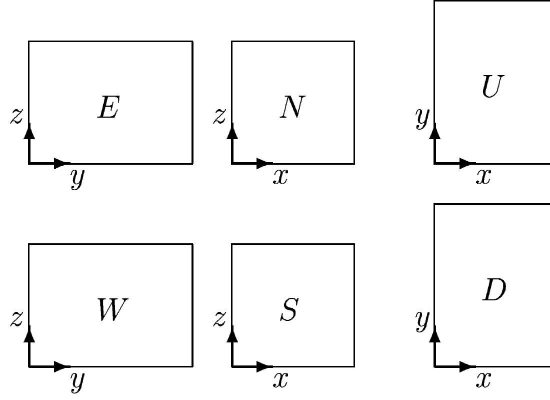


Figure 5.6: Integration regions for variable coefficient stencil compution for cells on the UE edge. These are the boundaries of the control volume.

Discretizing the integral form

$$-\int_{\partial V} a\frac{\partial\phi}{\partial n} = \int_V \rho$$

gives

$$\begin{aligned}
&\frac{a_n(\phi_n-\phi_o)}{\Delta y}\cdot\frac{3}{4}\Delta x\frac{3}{4}\Delta z \quad - \quad \frac{a_s(\phi_o-\phi_s)}{\Delta y}\cdot\frac{3}{4}\Delta x\frac{3}{4}\Delta z + \\
&\frac{a_e(\phi_e-\phi_o)}{\frac{1}{2}\Delta x}\cdot\Delta y\frac{3}{4}\Delta z \quad - \quad \frac{a_w(\phi_o-\phi_w)}{\Delta x}\cdot\Delta y\frac{3}{4}\Delta z + \\
&\frac{a_u(\phi_u-\phi_o)}{\frac{1}{2}\Delta z}\cdot\frac{3}{4}\Delta x\Delta y \quad - \quad \frac{a_d(\phi_o-\phi_d)}{\Delta z}\cdot\frac{3}{4}\Delta x\Delta y + \\
&= -\left(\frac{3}{4}\right)^2\Delta x\Delta y\Delta z\rho_o,
\end{aligned}$$

which reduces to

$$\begin{aligned}
\left(a_n + a_s + \frac{8}{3}a_e + \frac{4}{3}a_w + \frac{8}{3}a_u + \frac{4}{3}a_d\right)\phi_o \quad &- \quad a_n\phi_n - a_s\phi_s \\
&- \quad \frac{3}{4}a_w\phi_w - \frac{3}{8}a_e\phi_e \\
&- \quad \frac{3}{8}a_u\phi_u - \frac{3}{4}a_d\phi_d = h^2\rho_o,
\end{aligned}$$

when $h = \Delta x = \Delta y$.

Corner

See Fig. 5.7 for the integration regions corresponding to a control volume for a representative corner cell.

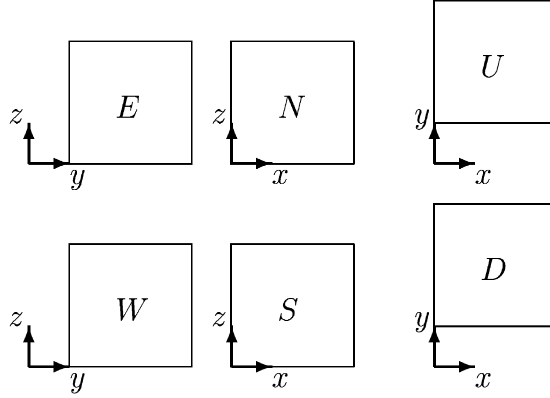


Figure 5.7: Integration regions for variable coefficient stencil compution for cells on the UNE corner. These are the boundaries of the control volume.

Discretizing the integral form

$$-\int_{\partial V} a\frac{\partial \phi}{\partial n} = \int_V \rho$$

gives

$$\begin{aligned}
&\frac{a_n(\phi_n-\phi_o)}{\frac{1}{2}\Delta y}\cdot\frac{3}{4}\Delta x\frac{3}{4}\Delta z &-& \quad\frac{a_s(\phi_o-\phi_s)}{\Delta y}\cdot\frac{3}{4}\Delta x\frac{3}{4}\Delta z+\\
&\frac{a_e(\phi_e-\phi_o)}{\frac{1}{2}\Delta x}\cdot\frac{3}{4}\Delta y\frac{3}{4}\Delta z &-& \quad\frac{a_w(\phi_o-\phi_w)}{\Delta x}\cdot\frac{3}{4}\Delta y\frac{3}{4}\Delta z+\\
&\frac{a_u(\phi_u-\phi_o)}{\frac{1}{2}\Delta z}\cdot\frac{3}{4}\Delta x\frac{3}{4}\Delta y &-& \quad\frac{a_d(\phi_o-\phi_d)}{\Delta z}\cdot\frac{3}{4}\Delta x\frac{3}{4}\Delta y+\\
&=-\left(\frac{3}{4}\right)^3\Delta x\Delta y\Delta z\rho_o,
\end{aligned}$$

which reduces to

$$\begin{aligned}
\left(\frac{8}{3}a_n+\frac{4}{3}a_s+\frac{8}{3}a_e+\frac{4}{3}a_w+\frac{8}{3}a_u+\frac{4}{3}a_d\right)\phi_o &-& \frac{8}{3}a_n\phi_n-\frac{3}{4}a_s\phi_s\\
&-& \frac{3}{4}a_w\phi_w-\frac{3}{8}a_e\phi_e\\
&-& \frac{3}{8}a_u\phi_u-\frac{3}{4}a_d\phi_d=h^2\rho_o,
\end{aligned}$$

when $h=\Delta x=\Delta y$.

5.2.2 Interpolation of Ghost Points in 3D

The ghost point interpolation procedure for the variable coefficient case is identical to the one shown in §5.1.2 for the constant coefficient case.

5.2.3 Flux Matching in 3D

This section introduces the flux-matching computations. Flux matching is used to avoid one-sided derivatives and preserve C^1 continuity.

The following examples in this section are representative of the basic types of configuration. For each example there are several other symmetric orientations for that configuration.

One Coarse-Fine Interface

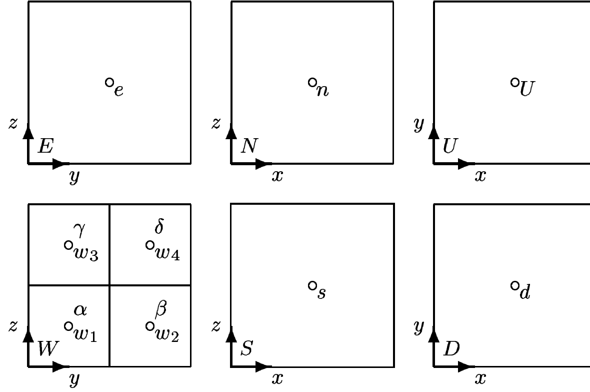


Figure 5.8: Integration regions for an interior cell with one coarse-fine interface (at the "west" cell wall) where flux matching is required. The little circles indicate the locations of the variable coefficient values. The labels α, β, γ, and δ are used to specify the corresponding ghost points used in the flux matching formula.

Discretizing the integral form

$$-\int_{\partial V} a\frac{\partial \phi}{\partial n} = \int_V \rho$$

gives

$$-(f_n - f_s + f_e - f_w + f_u - f_d) = h^3\rho_o,$$

where

$$\begin{aligned}
f_n &= \frac{a_n(\phi_n - \phi_o)}{h}h^2 = ha_n(\phi_n - \phi_o) \\
f_s &= \frac{a_s(\phi_o - \phi_s)}{h}h^2 = ha_s(\phi_o - \phi_s) \\
f_e &= \frac{a_e(\phi_e - \phi_o)}{h}h^2 = ha_e(\phi_e - \phi_o) \\
f_w &= \frac{a_{w_1}(\phi_\alpha - \phi_{w_1})}{\frac{1}{2}h}\frac{1}{4}h^2 + \frac{a_{w_2}(\phi_\beta - \phi_{w_2})}{\frac{1}{2}h}\frac{1}{4}h^2 + \frac{a_{w_3}(\phi_\gamma - \phi_{w_3})}{\frac{1}{2}h}\frac{1}{4}h^2 + \frac{a_{w_4}(\phi_\delta - \phi_{w_4})}{\frac{1}{2}h}\frac{1}{4}h^2 \\
&= \frac{1}{2}h\left(a_{w_1}(\phi_\alpha - \phi_{w_1}) + a_{w_2}(\phi_\beta - \phi_{w_2}) + a_{w_3}(\phi_\gamma - \phi_{w_3}) + a_{w_4}(\phi_\delta - \phi_{w_4})\right)
\end{aligned}$$

$$
\begin{aligned}
f_u &= \frac{a_u(\phi_u - \phi_o)}{h} h^2 = h a_u(\phi_u - \phi_o) \\
f_d &= \frac{a_d(\phi_o - \phi_d)}{h} h^2 = h a_d(\phi_o - \phi_d)
\end{aligned}
$$

and $h = \Delta x = \Delta y$.

Two Coarse-Fine Interfaces

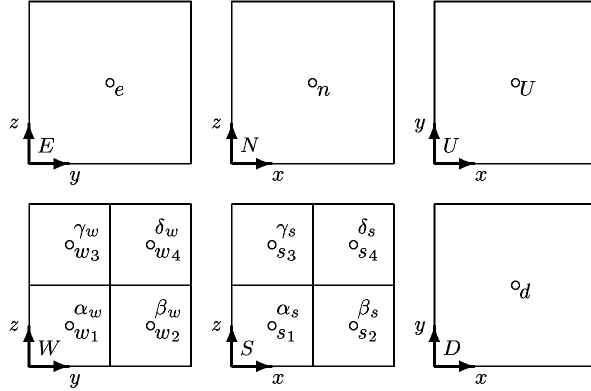


Figure 5.9: Integration regions for an interior cell with two coarse-fine interfaces (at the "west" and "south" cell walls) where flux matching is required. The little circles indicate the locations of the variable coefficient values. The labels α, β, γ, and δ are used to specify the corresponding ghost points used in the flux matching formula.

Discretizing the integral form

$$
-\int_{\partial V} a \frac{\partial \phi}{\partial n} = \int_V \rho
$$

gives

$$
-(f_n - f_s + f_e - f_w + f_u - f_d) = h^3 \rho_o,
$$

where

$$
\begin{aligned}
f_n &= \frac{a_n(\phi_n - \phi_o)}{h} h^2 = h a_n(\phi_n - \phi_o) \\
f_s &= \frac{a_{s_1}(\phi_{\alpha_s} - \phi_{s_1})}{\frac{1}{2}h} \frac{1}{4} h^2 + \frac{a_{s_2}(\phi_{\beta_s} - \phi_{s_2})}{\frac{1}{2}h} \frac{1}{4} h^2 + \frac{a_{s_3}(\phi_{\gamma_s} - \phi_{s_3})}{\frac{1}{2}h} \frac{1}{4} h^2 + \frac{a_{s_4}(\phi_{\delta_s} - \phi_{s_4})}{\frac{1}{2}h} \frac{1}{4} h^2 \\
&= \frac{1}{2} h \left(a_{s_1}(\phi_{\alpha_s} - \phi_{s_1}) + a_{s_2}(\phi_{\beta_s} - \phi_{s_2}) + a_{s_3}(\phi_{\gamma_s} - \phi_{s_3}) + a_{s_4}(\phi_{\delta_s} - \phi_{s_4}) \right) \\
f_e &= \frac{a_e(\phi_e - \phi_o)}{h} h^2 = h a_e(\phi_e - \phi_o)
\end{aligned}
$$

$$
\begin{aligned}
f_w &= \frac{a_{w_1}(\phi_{\alpha_w} - \phi_{w_1})}{\frac{1}{2}h}\frac{1}{4}h^2 + \frac{a_{w_2}(\phi_{\beta_w} - \phi_{w_2})}{\frac{1}{2}h}\frac{1}{4}h^2 + \frac{a_{w_3}(\phi_{\gamma_w} - \phi_{w_3})}{\frac{1}{2}h}\frac{1}{4}h^2 + \frac{a_{w_4}(\phi_{\delta_w} - \phi_{w_4})}{\frac{1}{2}h}\frac{1}{4}h^2 \\
&= \frac{1}{2}h\left(a_{w_1}(\phi_{\alpha_w} - \phi_{w_1}) + a_{w_2}(\phi_{\beta_w} - \phi_{w_2}) + a_{w_3}(\phi_{\gamma_w} - \phi_{w_3}) + a_{w_4}(\phi_{\delta_w} - \phi_{w_4})\right) \\
f_u &= \frac{a_u(\phi_u - \phi_o)}{h}h^2 = ha_u(\phi_u - \phi_o) \\
f_d &= \frac{a_d(\phi_o - \phi_d)}{h}h^2 = ha_d(\phi_o - \phi_d),
\end{aligned}
$$

and $h = \Delta x = \Delta y$.

Three Coarse-Fine Interfaces

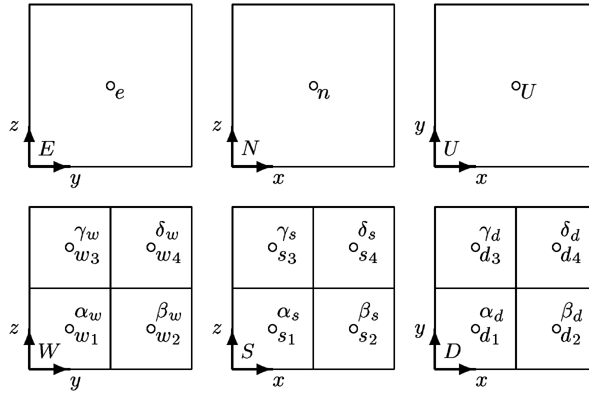


Figure 5.10: Integration regions for an interior cell with three coarse-fine interfaces (at the "west", "south", and "down" cell walls) where flux matching is required. The little circles indicate the locations of the variable coefficient values. The labels α, β, γ, and δ are used to specify the corresponding ghost points used in the flux matching formula.

Discretizing the integral form

$$
-\int_{\partial V} a\frac{\partial \phi}{\partial n} = \int_V \rho
$$

gives

$$
-(f_n - f_s + f_e - f_w + f_u - f_d) = h^3\rho_o,
$$

where

$$
\begin{aligned}
f_n &= \frac{a_n(\phi_n - \phi_o)}{h}h^2 = ha_n(\phi_n - \phi_o) \\
f_s &= \frac{a_{s_1}(\phi_{\alpha_s} - \phi_{s_1})}{\frac{1}{2}h}\frac{1}{4}h^2 + \frac{a_{s_2}(\phi_{\beta_s} - \phi_{s_2})}{\frac{1}{2}h}\frac{1}{4}h^2 + \frac{a_{s_3}(\phi_{\gamma_s} - \phi_{s_3})}{\frac{1}{2}h}\frac{1}{4}h^2 + \frac{a_{s_4}(\phi_{\delta_s} - \phi_{s_4})}{\frac{1}{2}h}\frac{1}{4}h^2 \\
&= \frac{1}{2}h\left(a_{s_1}(\phi_{\alpha_s} - \phi_{s_1}) + a_{s_2}(\phi_{\beta_s} - \phi_{s_2}) + a_{s_3}(\phi_{\gamma_s} - \phi_{s_3}) + a_{s_4}(\phi_{\delta_s} - \phi_{s_4})\right)
\end{aligned}
$$

$$
\begin{aligned}
f_e &= \frac{a_e(\phi_e-\phi_o)}{h}h^2 = ha_e(\phi_e-\phi_o) \\
f_w &= \frac{a_{w_1}(\phi_{\alpha_w}-\phi_{w_1})}{\frac{1}{2}h}\frac{1}{4}h^2 + \frac{a_{w_2}(\phi_{\beta_w}-\phi_{w_2})}{\frac{1}{2}h}\frac{1}{4}h^2 + \frac{a_{w_3}(\phi_{\gamma_w}-\phi_{w_3})}{\frac{1}{2}h}\frac{1}{4}h^2 + \frac{a_{w_4}(\phi_{\delta_w}-\phi_{w_4})}{\frac{1}{2}h}\frac{1}{4}h^2 \\
&= \frac{1}{2}h\left(a_{w_1}(\phi_{\alpha_w}-\phi_{w_1}) + a_{w_2}(\phi_{\beta_w}-\phi_{w_2}) + a_{w_3}(\phi_{\gamma_w}-\phi_{w_3}) + a_{w_4}(\phi_{\delta_w}-\phi_{w_4})\right) \\
f_u &= \frac{a_u(\phi_u-\phi_o)}{h}h^2 = ha_u(\phi_u-\phi_o) \\
f_d &= \frac{a_{d_1}(\phi_{\alpha_d}-\phi_{d_1})}{\frac{1}{2}h}\frac{1}{4}h^2 + \frac{a_{d_2}(\phi_{\beta_d}-\phi_{d_2})}{\frac{1}{2}h}\frac{1}{4}h^2 + \frac{a_{d_3}(\phi_{\gamma_d}-\phi_{d_3})}{\frac{1}{2}h}\frac{1}{4}h^2 + \frac{a_{d_4}(\phi_{\delta_d}-\phi_{d_4})}{\frac{1}{2}h}\frac{1}{4}h^2 \\
&= \frac{1}{2}h\left(a_{d_1}(\phi_{\alpha_d}-\phi_{d_1}) + a_{d_2}(\phi_{\beta_d}-\phi_{d_2}) + a_{d_3}(\phi_{\gamma_d}-\phi_{d_3}) + a_{d_4}(\phi_{\delta_d}-\phi_{d_4})\right),
\end{aligned}
$$

and $h = \Delta x = \Delta y$.

One Coarse-Fine Interface At a Boundary

See Fig. 5.11 for the control volume regions with labels. The dashed lines indicate the location of the boundary. The little circles indicate the locations of the variable coefficient values. The labels α, β, γ, and δ are used to specify the corresponding ghost points used in the flux matching formula.

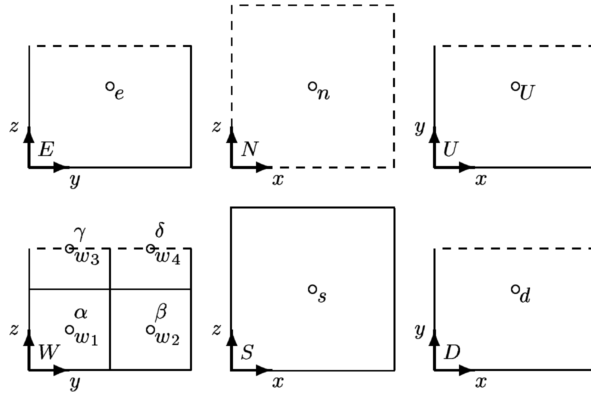


Figure 5.11: Integration regions at one boundary (the "north" boundary) with one coarse-fine interface (at the "west" cell wall) where flux matching is required. The dashed lines indicate the location of the boundary. The little circles indicate the locations of the variable coefficient values. The labels α, β, γ, and δ are used to specify the corresponding ghost points used in the flux matching formula.

Discretizing the integral form

$$
-\int_{\partial V} a\frac{\partial \phi}{\partial n} = \int_V \rho
$$

gives

$$-(f_n - f_s + f_e - f_w + f_u - f_d) = \frac{3}{4}h^3\rho_o,$$

where

$$\begin{aligned}
f_n &= \frac{a_n(\phi_n - \phi_o)}{\frac{1}{2}h}h^2 = 2ha_n(0 - \phi_o) = -2ha_n\phi_o \\
f_s &= \frac{a_s(\phi_o - \phi_s)}{h}h^2 = ha_s(\phi_o - \phi_s) \\
f_e &= \frac{a_e(\phi_e - \phi_o)}{h}\frac{3}{4}h^2 = \frac{3}{4}ha_e(\phi_e - \phi_o) \\
f_w &= \frac{a_{w_1}(\phi_\alpha - \phi_{w_1})}{\frac{1}{2}h}\frac{1}{4}h^2 + \frac{a_{w_2}(\phi_\beta - \phi_{w_2})}{\frac{1}{2}h}\frac{1}{4}h^2 + \frac{a_{w_3}(\phi_\gamma - \phi_{w_3})}{\frac{1}{2}h}\frac{1}{8}h^2 + \frac{a_{w_4}(\phi_\delta - \phi_{w_4})}{\frac{1}{2}h}\frac{1}{8}h^2 \\
&= \frac{1}{2}h\left(a_{w_1}(\phi_\alpha - \phi_{w_1}) + a_{w_2}(\phi_\beta - \phi_{w_2})\right) + \frac{1}{4}h\left(a_{w_3}(\phi_\gamma - \phi_{w_3}) + a_{w_4}(\phi_\delta - \phi_{w_4})\right) \\
f_u &= \frac{a_u(\phi_u - \phi_o)}{h}\frac{3}{4}h^2 = \frac{3}{4}ha_u(\phi_u - \phi_o) \\
f_d &= \frac{a_d(\phi_o - \phi_d)}{h}\frac{3}{4}h^2 = \frac{3}{4}ha_d(\phi_o - \phi_d),
\end{aligned}$$

and $h = \Delta x = \Delta y$.

Two Coarse-Fine Interfaces At a Boundary

See Fig. 5.12 for the control volume regions with labels. The dashed lines indicate the location of the boundary. The little circles indicate the locations of the variable coefficient values. The labels α, β, γ, and δ are used to specify the corresponding ghost points used in the flux matching formula.

Discretizing the integral form

$$-\int_{\partial V} a\frac{\partial\phi}{\partial n} = \int_V \rho$$

gives

$$-(f_n - f_s + f_e - f_w + f_u - f_d) = \frac{3}{4}h^3\rho_o,$$

where

$$\begin{aligned}
f_n &= \frac{a_n(\phi_n - \phi_o)}{\frac{1}{2}h}h^2 = 2ha_n(0 - \phi_o) = -2ha_n\phi_o \\
f_s &= \frac{a_s(\phi_o - \phi_s)}{h}h^2 = ha_s(\phi_o - \phi_s) \\
f_e &= \frac{a_e(\phi_e - \phi_o)}{h}\frac{3}{4}h^2 = \frac{3}{4}ha_e(\phi_e - \phi_o)
\end{aligned}$$

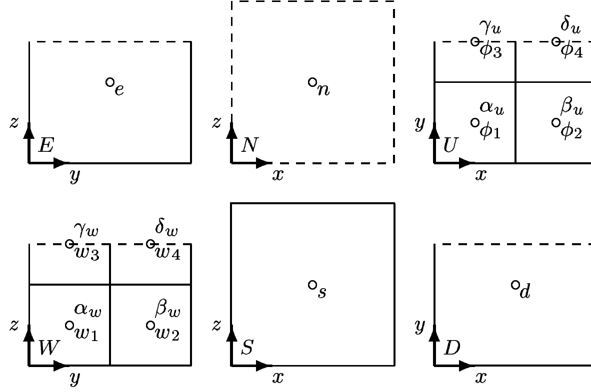


Figure 5.12: Integration regions at one boundary (the "north" boundary) with two coarse-fine interfaces (at the "west" cell wall and the "up" cell wall) where flux matching is required. The dashed lines indicate the location of the boundary. The little circles indicate the locations of the variable coefficient values. The labels α, β, γ, and δ are used to specify the corresponding ghost points used in the flux matching formula.

$$
\begin{aligned}
f_w &= \frac{a_{w_1}(\phi_{\alpha_w} - \phi_{w_1})}{\frac{1}{2}h}\frac{1}{4}h^2 + \frac{a_{w_2}(\phi_{\beta_w} - \phi_{w_2})}{\frac{1}{2}h}\frac{1}{4}h^2 + \frac{a_{w_3}(\phi_{\gamma_w} - \phi_{w_3})}{\frac{1}{2}h}\frac{1}{8}h^2 + \frac{a_{w_4}(\phi_{\delta_w} - \phi_{w_4})}{\frac{1}{2}h}\frac{1}{8}h^2 \\
&= \frac{1}{2}h\left(a_{w_1}(\phi_{\alpha_w} - \phi_{w_1}) + a_{w_2}(\phi_{\beta_w} - \phi_{w_2})\right) + \frac{1}{4}h\left(a_{w_3}(\phi_{\gamma_w} - \phi_{w_3}) + a_{w_4}(\phi_{\delta_w} - \phi_{w_4})\right) \\
f_u &= \frac{a_{\phi_1}(\phi_{\phi_1} - \phi_{\alpha_u})}{\frac{1}{2}h}\frac{1}{4}h^2 + \frac{a_{\phi_2}(\phi_{\phi_2} - \phi_{\beta_u})}{\frac{1}{2}h}\frac{1}{4}h^2 + \frac{a_{\phi_3}(\phi_{\phi_3} - \phi_{\gamma_u})}{\frac{1}{2}h}\frac{1}{8}h^2 + \frac{a_{\phi_4}(\phi_{\phi_4} - \phi_{\delta_u})}{\frac{1}{2}h}\frac{1}{8}h^2 \\
&= \frac{1}{2}h\left(a_{\phi_1}(\phi_{\phi_1} - \phi_{\alpha_u}) + a_{\phi_2}(\phi_{\phi_2} - \phi_{\beta_u})\right) + \frac{1}{4}h\left(a_{\phi_3}(\phi_{\phi_3} - \phi_{\gamma_u}) + a_{\phi_4}(\phi_{\phi_4} - \phi_{\delta_u})\right) \\
f_d &= \frac{a_d(\phi_o - \phi_d)}{h}\frac{3}{4}h^2 = \frac{3}{4}ha_d(\phi_o - \phi_d),
\end{aligned}
$$

and $h = \Delta x = \Delta y$.

One Coarse-Fine Interface At Two Boundaries

See Fig. 5.13 for the control volume regions with labels. The dashed lines indicate the location of the boundary. The little circles indicate the locations of the variable coefficient values. The labels α, β, γ, and δ are used to specify the corresponding ghost points used in the flux matching formula.

Discretizing the integral form

$$-\int_{\partial V} a\frac{\partial \phi}{\partial n} = \int_V \rho$$

gives

$$-(f_n - f_s + f_e - f_w + f_u - f_d) = \frac{9}{16}h^3\rho_o,$$

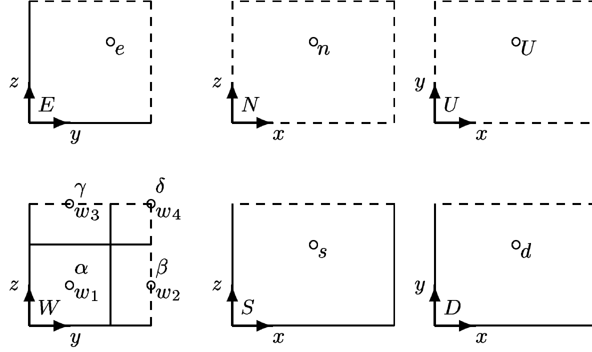


Figure 5.13: Integration regions at two boundaries (the "up" boundary and the "north" boundary) with one coarse-fine interface (at the "west" cell wall) where flux matching is required. The dashed lines indicate the location of the boundary. The little circles indicate the locations of the variable coefficient values. The labels α, β, γ, and δ are used to specify the corresponding ghost points used in the flux matching formula.

where

$$
\begin{aligned}
f_n &= \frac{a_n(\phi_n - \phi_o)}{\frac{1}{2}h}\frac{3}{4}h^2 = \frac{3}{2}ha_n(0-\phi_o) = -\frac{3}{2}ha_n\phi_o \\
f_s &= \frac{a_s(\phi_o - \phi_s)}{h}\frac{3}{4}h^2 = \frac{3}{4}ha_s(\phi_o - \phi_s) \\
f_e &= \frac{a_e(\phi_e - \phi_o)}{h}\frac{9}{16}h^2 = \frac{9}{16}ha_e(\phi_e - \phi_o) \\
f_w &= \frac{a_{w_1}(\phi_\alpha - \phi_{w_1})}{\frac{1}{2}h}\frac{1}{4}h^2 + \frac{a_{w_2}(\phi_\beta - \phi_{w_2})}{\frac{1}{2}h}\frac{1}{4}h^2 + \frac{a_{w_3}(\phi_\gamma - \phi_{w_3})}{\frac{1}{2}h}\frac{1}{8}h^2 + \frac{a_{w_4}(\phi_\delta - \phi_{w_4})}{\frac{1}{2}h}\frac{1}{16}h^2 \\
&= \frac{1}{4}h\left(a_{w_1}(\phi_\alpha - \phi_{w_1}) + a_{w_2}(\phi_\beta - \phi_{w_2})\right) + \frac{1}{8}ha_{w_3}(\phi_\gamma - \phi_{w_3}) + \frac{1}{16}ha_{w_4}(\phi_\delta - \phi_{w_4}) \\
f_u &= \frac{a_u(\phi_u - \phi_o)}{\frac{1}{2}h}\frac{3}{4}h^2 = \frac{3}{2}ha_u(0-\phi_o) = -\frac{3}{2}ha_u\phi_o \\
f_d &= \frac{a_d(\phi_o - \phi_d)}{h}\frac{3}{4}h^2 = \frac{3}{4}ha_d(\phi_o - \phi_d),
\end{aligned}
$$

and $h = \Delta x = \Delta y$.

Chapter 6

Ghost Point Interpolation Revisited

Ghost point interpolation was presented in §3 and §5 in a way that is useful for conveying the process. It is not efficient for the implementation, though, especially in 3D. In the code, we want one single simple expression for each ghost point without bothering with multiple steps and intermediate values. This chapter derives single expressions for the ghost point interpolation in both 2D and 3D. The systems of equations for the interpolation coefficients in this chapter were all solved with Matlab.

Before continuing with the following sections, it is important to note some shorthand that is employed to make the notation more readable. The notation for grid points, e.g., $\bullet_a$, is used not only to denote a grid point but also to represent the value of the function at that grid point. There is never any ambiguity regarding what function value is represented, because this section deals specifically with interpolation of function values for a single function, the function for which ghost points are required. In the code, this is usually the solution function, although in the post-smoothing only case §7.4 it can also be the correction function. For this discussion, the particular function involved is immaterial, because the derivation and formulae are the same in either case.

6.1 2D

This section derives the ghost point interpolation in 2D. Ghost point interpolation in 2D is derived in two phases. Phase one is described in §6.1.1. Phase two is described in §6.1.2. It is the phase two ghost points that are required by the algorithm to apply the discrete operator at interface points. In the final implementation, only phase two ghost points will be stored. The intermediate ghost points are used only in the derivation of the final formula.

6.1.1 Phase One (Intermediate) Ghost Points

The intermediate ghost points are illustrated in Fig. 6.1.

We want a quadratic interpolation formula of the form

$$u(x) = c_0 + c_1 x + c_2 x^2,$$

where

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 2h & (2h)^2 \\ 1 & 4h & (4h)^2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} \bullet_1 \\ \bullet_2 \\ \bullet_3 \end{bmatrix} \Rightarrow \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} \bullet_1 \\ \frac{1}{h}\left(\frac{-3}{4}\bullet_1 + \bullet_2 + \frac{-1}{4}\bullet_3\right) \\ \frac{1}{h^2}\left(\frac{1}{8}\bullet_1 - \frac{1}{4}\bullet_2 + \frac{1}{8}\bullet_3\right) \end{bmatrix}.$$

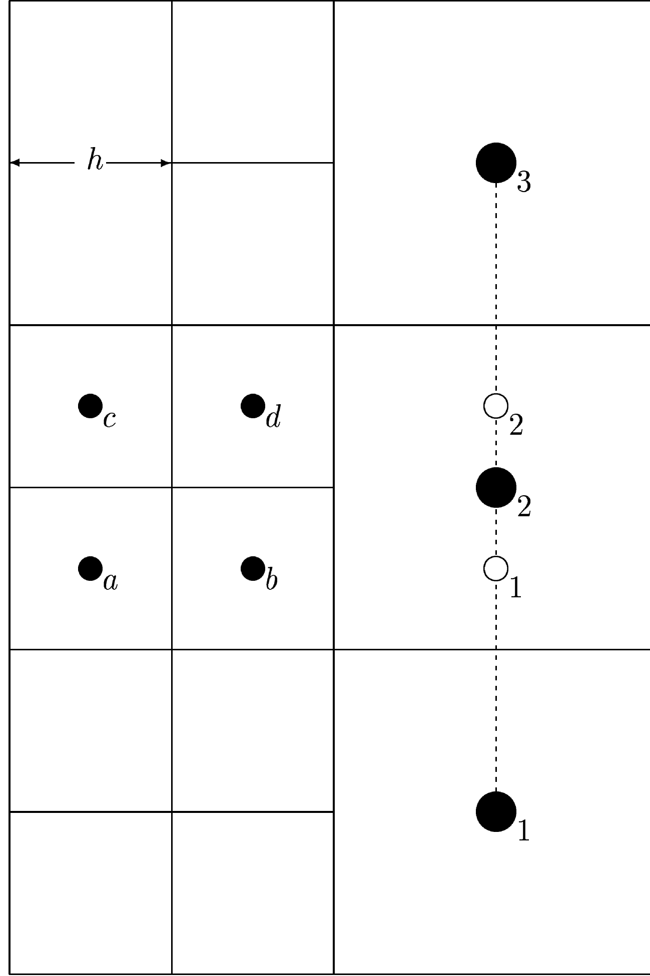


Figure 6.1: Phase one ghost points

So

$$\begin{aligned} u(\circ_1) &= u\left(\frac{3}{2}h\right) \\ &= c_0 + c_1\left(\frac{3}{2}h\right) + c_2\left(\frac{3}{2}h\right)^2 \\ &= \frac{5}{32}\bullet_1 + \frac{15}{16}\bullet_2 - \frac{3}{32}\bullet_3. \end{aligned}$$

and

$$\begin{aligned} u(\circ_2) &= u\left(\frac{5}{2}h\right) \\ &= c_0 + c_1\left(\frac{5}{2}h\right) + c_2\left(\frac{5}{2}h\right)^2 \\ &= \frac{-3}{32}\bullet_1 + \frac{15}{16}\bullet_2 + \frac{5}{32}\bullet_3. \end{aligned}$$

6.1.2 Phase Two Ghost Points

The phase two ghost points are illustrated in Fig. 6.2. These are the ghost points used by

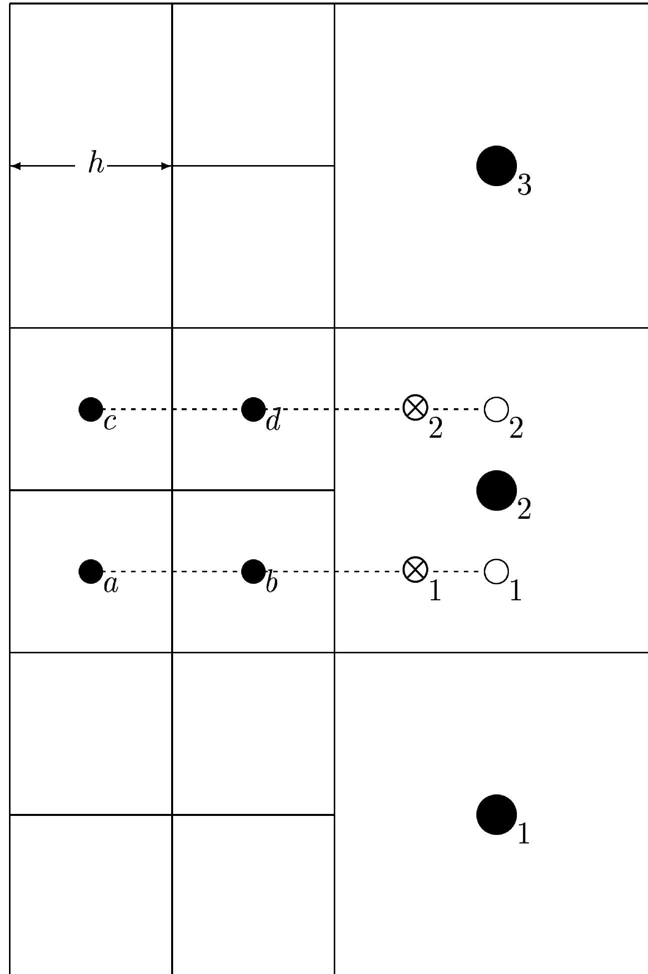


Figure 6.2: Phase two ghost points

the AMRMG algorithm for the discrete operator at coarse-fine interfaces.

We want a quadratic interpolation formula of the form

$$u(x) = c_0 + c_1 x + c_2 x^2,$$

where

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & h & h^2 \\ 1 & \frac{5}{2}h & \left(\frac{5}{2}h\right)^2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} \bullet_a \\ \bullet_b \\ \circ_1 \end{bmatrix} \Rightarrow \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} \bullet_a \\ \frac{1}{h}\left(\frac{-7}{5}\bullet_a - \frac{-5}{3}\bullet_b + \frac{-4}{15}\circ_1\right) \\ \frac{1}{h^2}\left(\frac{2}{5}\bullet_a - \frac{2}{3}\bullet_b + \frac{4}{15}\circ_1\right) \end{bmatrix}.$$

So

$$u(\otimes_1) = u\,(2h)$$

$$= c_0 + c_1 (2h) + c_2 (2h)^2$$
$$= \frac{-1}{5}\bullet_a + \frac{2}{3}\bullet_b + \frac{8}{15}\circ_1.$$

Likewise,

$$u(\otimes_2) = \frac{-1}{5}\bullet_c + \frac{2}{3}\bullet_d + \frac{8}{15}\circ_2.$$

Then substituting for $\circ_1$ and $\circ_2$ gives

$$u(\otimes_1) = \frac{-1}{5}\bullet_a + \frac{2}{3}\bullet_b + \frac{1}{12}\CIRCLE_1 + \frac{1}{2}\CIRCLE_2 + \frac{-1}{20}\CIRCLE_3$$

and

$$u(\otimes_2) = \frac{-1}{5}\bullet_c + \frac{2}{3}\bullet_d + \frac{-1}{20}\CIRCLE_1 + \frac{1}{2}\CIRCLE_2 + \frac{1}{12}\CIRCLE_3.$$

These final formulae are the ones used to implement the ghost point computations in the code.

6.2 3D

This section derives the ghost point interpolation in 3D. Ghost point interpolation in 3D is derived in three phases. Phase one and phase two are described in §6.2.1. Phase three is described in §6.2.2 It is the phase three ghost points that are required by the algorithm to apply the discrete operator at interface points. In the final implementation, only phase three ghost points will be stored. The intermediate ghost points are used only in the derivation of the final formula.

6.2.1 Phase One and Phase Two (Intermediate) Ghost Points

See Fig. 6.3 for the locations of phase one ghost points. The phase one ghost points are used to interpolate the phase two ghost points. See Fig. 6.4 for the locations of phase two ghost points. The phase two ghost points $\circ_1$, $\circ_2$, $\circ_3$, and $\circ_4$ are used in the §6.2.2, along with fine grid points, to interpolate phase three ghost points. Phase three ghost points are the only ghost points needed by the AMRMG algorithm. The intermediate ghost points are used in the process of deriving the final formula for the phase three ghost points.

First interpolate phase one ghost points $\bullet_a$, $\bullet_b$, $\bullet_c$, $\bullet_d$, $\bullet_e$, and $\bullet_f$ using the template in Fig. 6.5. The formula for $\bullet$ in Fig. 6.5 is of the form

$$u(x) = c_0 + c_1 x + c_2 x^2,$$

where

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & h & h^2 \\ 1 & 2h & (2h)^2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} \CIRCLE_1 \\ \CIRCLE_2 \\ \CIRCLE_3 \end{bmatrix} \Rightarrow \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} \CIRCLE_1 \\ \frac{1}{h}\left(\frac{-3}{2}\CIRCLE_1 + 2\CIRCLE_2 + \frac{-1}{2}\CIRCLE_3\right) \\ \frac{1}{h^2}\left(\frac{1}{2}\CIRCLE_1 - \CIRCLE_2 + \frac{1}{2}\CIRCLE_3\right) \end{bmatrix}.$$

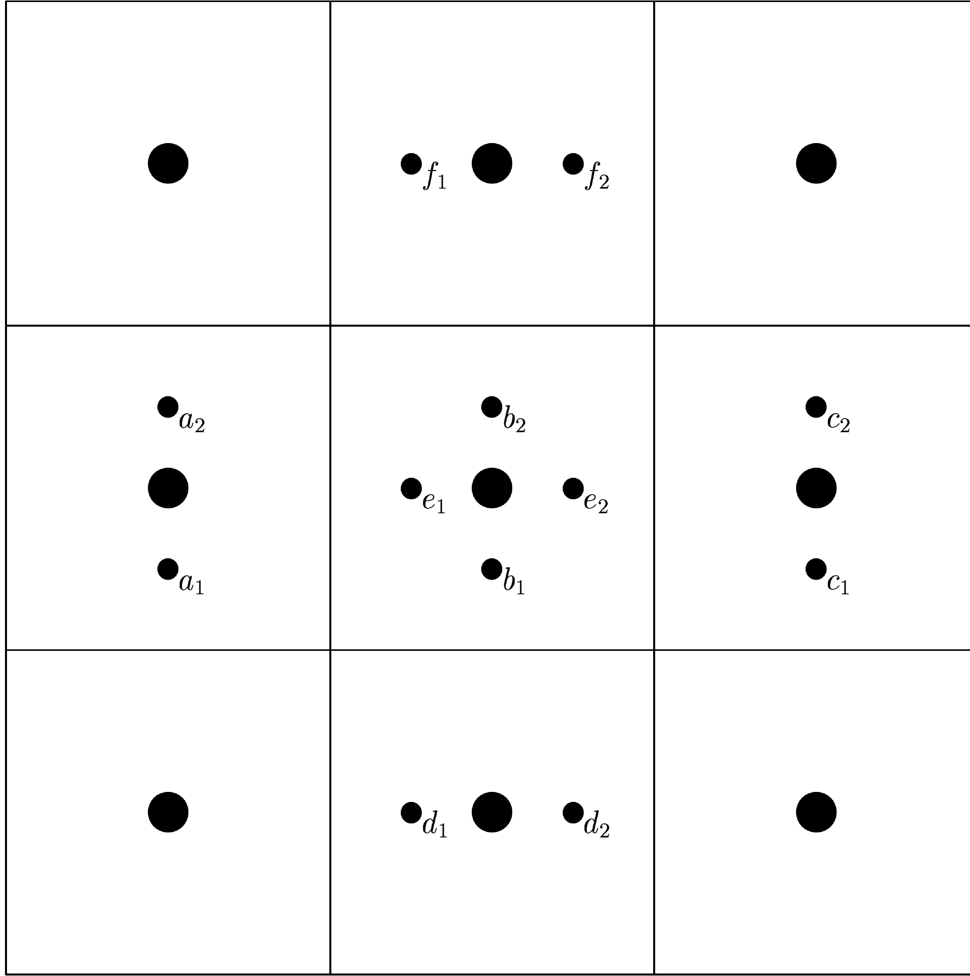


Figure 6.3: Phase one ghost points.

So

$$
\begin{aligned}
u(\bullet) &= u\left(\frac{3}{4}h\right) \\
&= c_0 + c_1\left(\frac{3}{4}h\right) + c_2\left(\frac{3}{4}h\right)^2 \\
&= \frac{5}{32}\bullet_1 + \frac{15}{16}\bullet_2 - \frac{3}{32}\bullet_3.
\end{aligned}
$$

This formula can be used to compute all the phase one ghost points. Just substitute the corresponding coarse grid points in place of $\bullet_1$, $\bullet_2$, and $\bullet_3$.

Then interpolate the phase two ghost points. The interpolation formulas for the phase two ghost points $\circ_1$, $\circ_2$, $\circ_3$, and $\circ_4$, are derived in the following separate subsections.

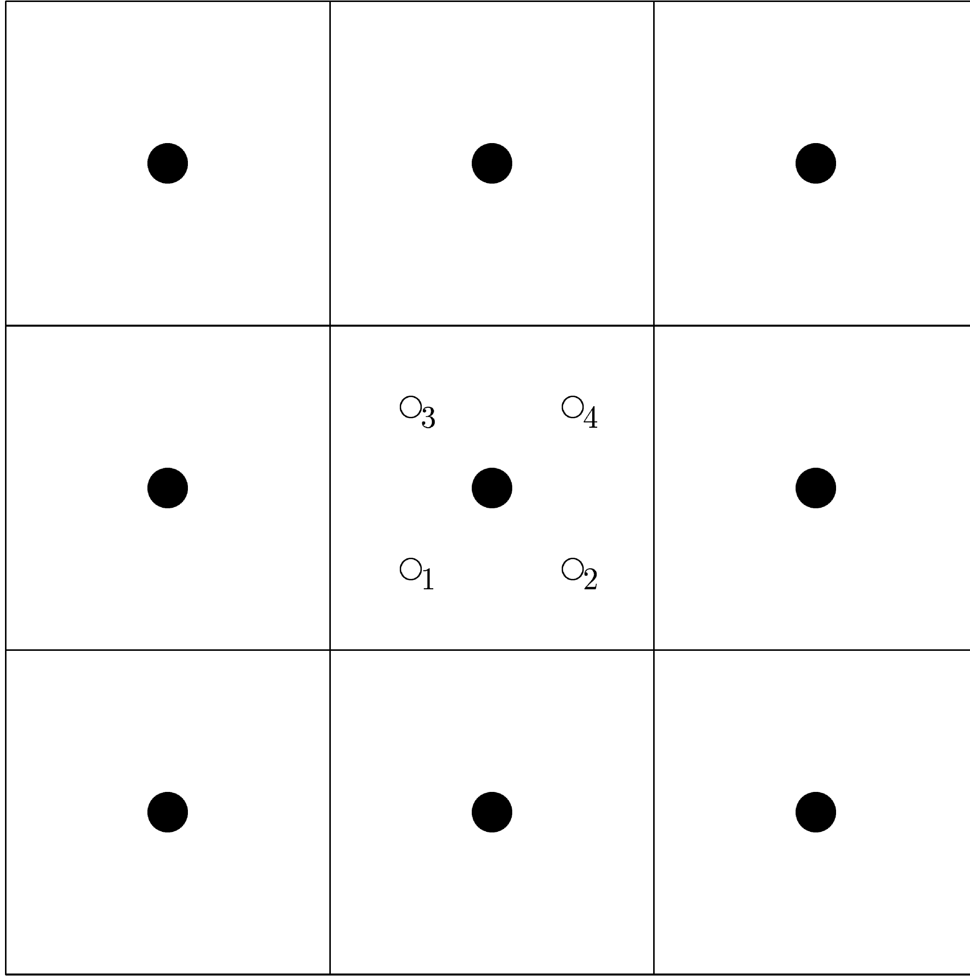


Figure 6.4: Phase two ghost points.

Phase two ghost point number one

Interpolate the phase two ghost point $\circ_1$ shown in Fig. 6.6. We will use the average of two quadratic interpolation formulas: one that interpolates the phase one ghost points $\bullet_a$, $\bullet_b$, and $\bullet_c$, and one that interpolates the phase one ghost points $\bullet_d$, $\bullet_e$, and $\bullet_f$. Individually, these conform to the same template as the phase one interpolation. So

$$\begin{aligned} \circ_1 &= \frac{1}{2}\left(\left(\frac{5}{32}\bullet_a + \frac{15}{16}\bullet_b - \frac{3}{32}\bullet_c\right) + \left(\frac{5}{32}\bullet_d + \frac{15}{16}\bullet_e - \frac{3}{32}\bullet_f\right)\right) \\ &= \frac{25}{1024}\bullet_1 + \frac{75}{512}\bullet_2 - \frac{15}{1024}\bullet_3 + \frac{75}{512}\bullet_4 + \frac{225}{256}\bullet_5 - \frac{45}{512}\bullet_6 - \frac{15}{1024}\bullet_7 - \frac{45}{512}\bullet_8 + \frac{9}{1024}\bullet_9 \end{aligned}$$

This motivates Table 6.1 which shows the weights of the coarse grid points in this formula. The table will be convenient for determining the formulas for the other three phase two ghost points by symmetry instead of deriving them all from scratch.

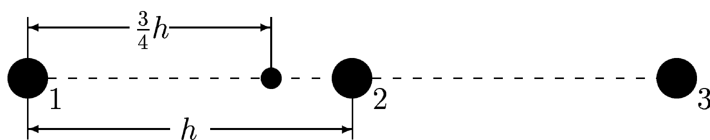


Figure 6.5: Template for interpolation of phase one ghost points.

$\frac{-15}{1024}$	$\frac{-45}{512}$	$\frac{9}{1024}$
$\frac{75}{512}$	$\frac{225}{256}$	$\frac{-45}{512}$
$\frac{25}{1024}$	$\frac{75}{512}$	$\frac{-15}{1024}$

Table 6.1: Coarse grid point weights for the first phase two ghost point.

Phase two ghost point number two

The symmetry associated with the phase two ghost points is radial symmetry with respect to the center coarse grid point $\bullet_5$. The second phase two ghost point $\circ_2$ is positioned 90 degrees counter-clockwise from $\circ_1$ with respect to $\bullet_5$, so the weights for the second phase two ghost point interpolation are given by rotating Table 6.1 90 degrees counter-clockwise with respect to $\bullet_5$. The resulting weights are shown in Table 6.2. The corresponding interpolation formula for $\circ_2$, then, is

$$\frac{-15}{1024}\bullet_1 + \frac{75}{512}\bullet_2 + \frac{25}{1024}\bullet_3 + \frac{-45}{512}\bullet_4 + \frac{225}{256}\bullet_5 + \frac{75}{512}\bullet_6 + \frac{9}{1024}\bullet_7 + \frac{-45}{512}\bullet_8 + \frac{-15}{1024}\bullet_9.$$

$\frac{9}{1024}$	$\frac{-45}{512}$	$\frac{-15}{1024}$
$\frac{-45}{512}$	$\frac{225}{256}$	$\frac{75}{512}$
$\frac{-15}{1024}$	$\frac{75}{512}$	$\frac{25}{1024}$

Table 6.2: Coarse grid point weights for the second phase two ghost point.

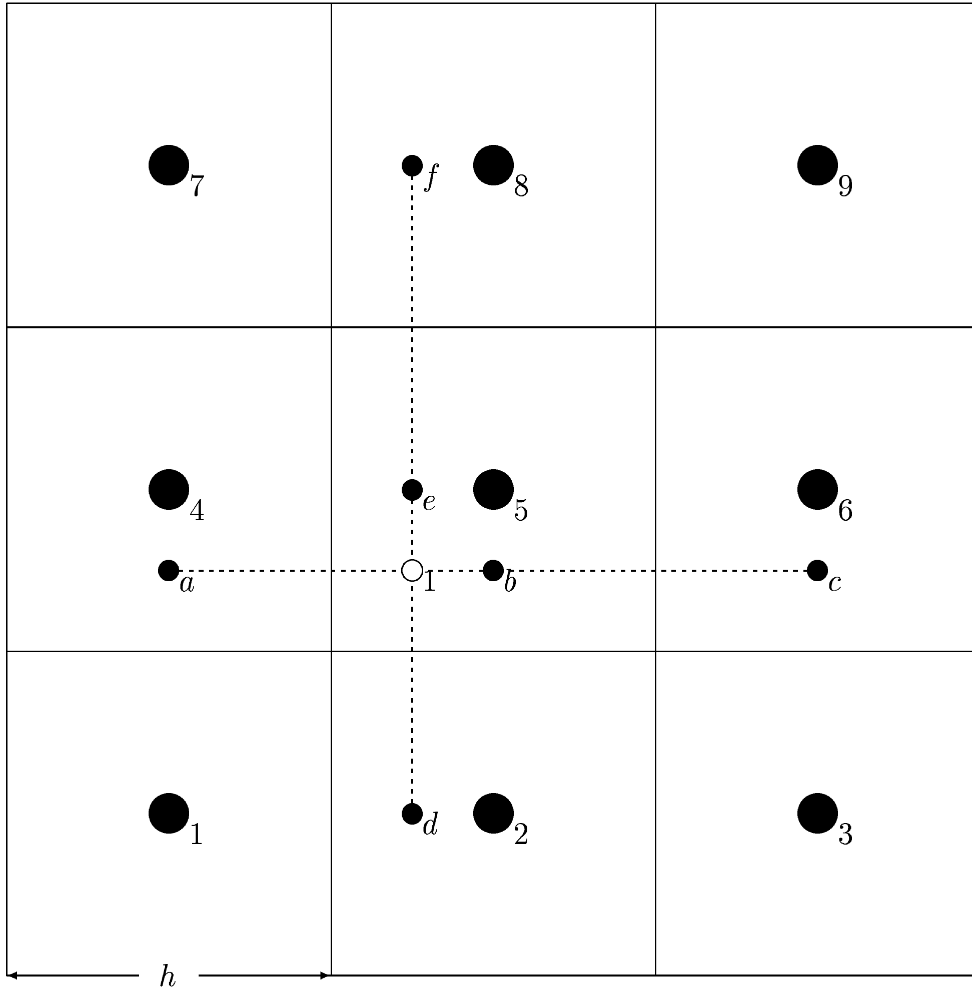


Figure 6.6: Interpolation of the first phase two ghost point.

Phase two ghost point number three

Following the same procedure as in §6.2.1 rotate Table 6.2 90 degrees counter-clockwise to get Table 6.3, the weights for ghost point $\circ_3$. The corresponding interpolation formula for $\circ_3$, then, is

$$\frac{9}{1024}\bullet_1 + \frac{-45}{512}\bullet_2 + \frac{-15}{1024}\bullet_3 + \frac{-45}{512}\bullet_4 + \frac{225}{256}\bullet_5 + \frac{75}{512}\bullet_6 + \frac{-15}{1024}\bullet_7 + \frac{75}{512}\bullet_8 + \frac{25}{1024}\bullet_9.$$

Phase two ghost point number four

Following the same procedure as in §6.2.1 rotate Table 6.3 90 degrees counter-clockwise to get Table 6.4, the weights for ghost point $\circ_4$. The corresponding interpolation formula for

$\frac{-15}{1024}$	$\frac{75}{512}$	$\frac{25}{1024}$
$\frac{-45}{512}$	$\frac{225}{256}$	$\frac{75}{512}$
$\frac{9}{1024}$	$\frac{-45}{512}$	$\frac{-15}{1024}$

Table 6.3: Coarse grid point weights for the third phase two ghost point.

$\frac{25}{1024}$	$\frac{75}{512}$	$\frac{-15}{1024}$
$\frac{75}{512}$	$\frac{225}{256}$	$\frac{-45}{512}$
$\frac{-15}{1024}$	$\frac{-45}{512}$	$\frac{9}{1024}$

Table 6.4: Coarse grid point weights for the fourth phase two ghost point.

$\circ_4$, then, is

$$\frac{-15}{1024}\bullet_1 + \frac{-45}{512}\bullet_2 + \frac{9}{1024}\bullet_3 + \frac{75}{512}\bullet_4 + \frac{225}{256}\bullet_5 + \frac{-45}{512}\bullet_6 + \frac{25}{1024}\bullet_7 + \frac{75}{512}\bullet_8 + \frac{-15}{1024}\bullet_9.$$

6.2.2 Phase Three Ghost Points

Main ghost points. These are used by the AMRMG algorithm for the discrete operator at coarse-fine interfaces.

With the phase two ghost points from §6.2.1, interpolate ghost points $\otimes_1$, $\otimes_2$, $\otimes_3$, and $\otimes_4$ using the template in Fig. 6.7. This is the same as for the phase two ghost points in

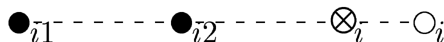


Figure 6.7: Template for interpolation of phase three ghost points, $i = 1..4$. Same as in Fig. 6.2 of the 2D case.

the 2D case, Fig. 6.2, except there are four $\otimes$ ghost points to interpolate here instead of

only two. *Notice that* **h** *is the mesh spacing on the fine grid side, in this section.* We want a quadratic interpolation formula of the form

$$u(x) = c_0 + c_1 x + c_2 x^2,$$

where

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & h & h^2 \\ 1 & \frac{5}{2}h & \left(\frac{5}{2}h\right)^2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} \bullet_{11} \\ \bullet_{12} \\ \circ_1 \end{bmatrix} \Rightarrow \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} \bullet_a \\ \frac{1}{h}\left(\frac{-7}{5}\bullet_a - \frac{-5}{3}\bullet_b + \frac{-4}{15}\circ_1\right) \\ \frac{1}{h^2}\left(\frac{2}{5}\bullet_a - \frac{2}{3}\bullet_b + \frac{4}{15}\circ_1\right) \end{bmatrix}.$$

So

$$\begin{aligned} u(\otimes_1) &= u\left(2h\right) \\ &= c_0 + c_1\left(2h\right) + c_2\left(2h\right)^2 \\ &= \frac{-1}{5}\bullet_{11} + \frac{2}{3}\bullet_{12} + \frac{8}{15}\circ_1. \end{aligned}$$

Likewise,

$$u(\otimes_2) = \frac{-1}{5}\bullet_{21} + \frac{2}{3}\bullet_{22} + \frac{8}{15}\circ_2$$

and

$$u(\otimes_3) = \frac{-1}{5}\bullet_{31} + \frac{2}{3}\bullet_{32} + \frac{8}{15}\circ_3$$

and

$$u(\otimes_4) = \frac{-1}{5}\bullet_{41} + \frac{2}{3}\bullet_{42} + \frac{8}{15}\circ_4.$$

Then substituting for $\circ_1$, $\circ_2$, $\circ_3$, and $\circ_4$ gives

$$\begin{aligned} \otimes_1 &= \frac{-1}{5}\bullet_{11} + \frac{2}{3}\bullet_{12} \\ &+ \frac{5}{384}\bullet_1 + \frac{5}{64}\bullet_2 + \frac{-1}{128}\bullet_3 \\ &+ \frac{5}{64}\bullet_4 + \frac{15}{32}\bullet_5 + \frac{-3}{64}\bullet_6 \\ &+ \frac{-1}{128}\bullet_7 + \frac{-3}{64}\bullet_8 + \frac{3}{640}\bullet_9 \end{aligned}$$

and

$$\begin{aligned} \otimes_2 &= \frac{-1}{5}\bullet_{21} + \frac{2}{3}\bullet_{22} \\ &+ \frac{-1}{128}\bullet_1 + \frac{5}{64}\bullet_2 + \frac{5}{384}\bullet_3 \\ &+ \frac{-3}{64}\bullet_4 + \frac{15}{32}\bullet_5 + \frac{5}{64}\bullet_6 \\ &+ \frac{3}{640}\bullet_7 + \frac{-3}{64}\bullet_8 + \frac{-1}{128}\bullet_9 \end{aligned}$$

$$
\begin{aligned}
\otimes_3 &= \frac{-1}{5}\bullet_{31} + \frac{2}{3}\bullet_{32} \\
&+ \frac{3}{640}\CIRCLE_1 + \frac{-3}{64}\CIRCLE_2 + \frac{-1}{128}\CIRCLE_3 \\
&+ \frac{-3}{64}\CIRCLE_4 + \frac{15}{32}\CIRCLE_5 + \frac{5}{64}\CIRCLE_6 \\
&+ \frac{-1}{128}\CIRCLE_7 + \frac{5}{64}\CIRCLE_8 + \frac{5}{384}\CIRCLE_9
\end{aligned}
$$

$$
\begin{aligned}
\otimes_4 &= \frac{-1}{5}\bullet_{41} + \frac{2}{3}\bullet_{42} \\
&+ \frac{-1}{128}\CIRCLE_1 + \frac{-3}{64}\CIRCLE_2 + \frac{3}{640}\CIRCLE_3 \\
&+ \frac{5}{64}\CIRCLE_4 + \frac{15}{32}\CIRCLE_5 + \frac{-3}{64}\CIRCLE_6 \\
&+ \frac{5}{384}\CIRCLE_7 + \frac{5}{64}\CIRCLE_8 + \frac{-1}{128}\CIRCLE_9
\end{aligned}
$$

These final four formulae are the ones used to implement the ghost point computations in the code.

Chapter 7

AMRMG

In this section, we define the operators, vectors, and algorithms needed to solve (1.1) numerically on an adaptive mesh hierarchy using a multilevel method.Much of the material is motivated by [70].

7.1 Review of Notation

As in §2.5, the ith single patch on grid level ℓ is denoted by $\Lambda^{\ell,i}$ and the union of all patches on grid level ℓ is denoted by Λ^{ℓ}. Review the example illustrated in Fig. 2.4.

Composite grid variables are denoted by a subscript c. The ℓth level composite grid is denoted by Λ_c^{ℓ}. The top level (most refined) composite grid is denoted by $\Lambda_c^{\ell_{max}}$. Review the example illustrated in Fig. 2.5.

The solution, ϕ_c, and right hand side, ρ_c, are ultimately needed only on $\Lambda_c^{\ell_{max}}$ (e.g., the top level in the example illustrated by Fig. 2.5). The residual, r_c, and correction, e_c, are needed on the entire composite grid hierarchy, although they only need to be stored on

$$\bigcup_{\ell=1}^{\ell_{max}} \Lambda^{\ell},$$

which is the patch based grid hierarchy as shown for a small example in Fig. 2.4. This patch based grid hierarchy versus the composite grid hierarchy is an important distinction, and it motivates definitions of new patch based versions of the operator $\mathcal{L}$ in §7.3.

7.2 Comparison With Standard Multigrid Algorithm

Alg. 7.1 gives a standard multigrid algorithm to solve (2.2). Multigrid methods always have at least one solver, called a smoother or rougher [26, 27] and sometimes more than one of each (for pre- and post-processing). For the multilevel algorithms that we define in the following sections, we define a smoother on each Λ^{ℓ} in terms of the new patch based discrete operators that are introduced. For now, we use the notion of a composite grid smoother $S_c^{\ell}(u_c^{\ell}, f_c^{\ell})$. To define the composite grid smoother, let $u_{c,ij}^{\ell}$ and $f_{c,ij}^{\ell}$ be the solution and right hand side values at the grid point indexed by (i,j), and let $\mathcal{L}_{c,ij}^{\ell}$ denote the row of matrix $\mathcal{L}_c^{\ell}$ that corresponds to the grid point indexed by (i,j). Then the composite grid smoother $S_c^{\ell}(u_c^{\ell}, f_c^{\ell})$ is defined pointwise by

$$u_{c,ij}^{\ell} \leftarrow u_{c,ij}^{\ell} + \lambda_{ij} \left(f_{c,ij}^{\ell} - \mathcal{L}_{c,ij}^{\ell} u_c^{\ell} \right),$$

where, the damping factor λ_{ij} is the reciprocoal of the diagonal entry of the operator at the grid point indexed by (i, j). The damping factor is discussed in more detail below.

Algorithm 7.1 Standard multigrid V cycle

MG($\ell, \phi_c^\ell, \rho_c^\ell$ **)**

1: **if** $\ell == 1$ **then**
2: $\quad \phi_c^\ell \leftarrow S_c^\ell(\phi_c^\ell, \rho_c^\ell)$ on Λ^ℓ
3: $\quad$ **return**
4: **end if**
5: $\phi_c^\ell \leftarrow S_c^\ell(\phi_c^\ell, \rho_c^\ell)$ on Λ^ℓ
6: $\rho_c^{\ell-1} \leftarrow \mathcal{P}^\ell(\rho_c^\ell - \mathcal{L}_c^\ell \phi_c^\ell)$
7: $\phi_c^{\ell-1} \leftarrow 0$
8: MG($\ell - 1, \phi_c^{\ell-1}, \rho_c^{\ell-1}$)
9: $\phi_c^\ell \leftarrow \phi_c^\ell + \mathcal{R}^\ell \phi_c^{\ell-1}$
10: $\phi_c^\ell \leftarrow S_c^\ell(\phi_c^\ell, \rho_c^\ell)$ on Λ^ℓ

The main difference between Alg. 7.1 and typical multigrid algorithms is that the smoothing, S_c^ℓ, is only employed on Λ^ℓ. This means that only values within the composite grid corresponding to the ℓth level patches are updated.

Algorithmically, the final method (see Alg. 7.2) is equivalent to Alg. 7.1.

However, a number of re-formulations are made to improve the efficiency and practicality of the scheme. These modifications are based on the observation that most of the time we only work on patches within a given level of the composite grid. In fact, many unknowns within other parts of the composite grid level are zero. Thus, we can avoid storing all unknowns from all composite grids within the grid hierarchy. Additionally, we do not want to work directly with $\mathcal{L}_c^\ell$. Instead, it is preferable to use only operators defined on patches at the different levels. These operators are defined in §7.3.

For reasons described below, it is convenient to maintain both the ℓth level multigrid correction *and* the ℓth-level composite grid solution on $\Lambda_c^{\ell_{max}} - \mathcal{P}(\Lambda^{\ell+1})$ (see Fig. 7.1). To do this, we introduce new notation. Specifically, define e_c^ℓ as a correction computed to the current solution. Notice that e_c^ℓ is identical to ϕ_c^ℓ except on the finest level. The composite grid solution on $\Lambda_c^{\ell_{max}} - \mathcal{P}(\Lambda^{\ell+1})$ is kept in the values of $\phi_c^{\ell_{max}}$ corresponding to $\Lambda_c^{\ell_{max}} - \mathcal{P}(\Lambda^{\ell+1})$. So, we replace all ϕ_c^ℓ on coarse grids with e_c^ℓ. On the finest grid we retain $\phi_c^{\ell_{max}}$ and also use $e_c^{\ell_{max}}$ for the finest grid correction.

We now rewrite the multigrid algorithm with the new notation in Alg. 7.3. Notice that during the multigrid V cycle on the ℓth level, $\phi_c^{\ell_{max}}$ is the current ℓ^{th}-level composite grid solution on $\Lambda_c^{\ell_{max}} - \mathcal{P}(\Lambda^{\ell+1})$.

Algorithm 7.2 AMR Multigrid V cycle.

MG($\ell, e^\ell, r^\ell, \rho^\ell, \phi^{\ell_{max}}$ **)**

1: **if** $\ell == 1$ **then**
2: $\quad e^\ell \leftarrow S^\ell(e^\ell, r^\ell)$ on Λ^ℓ
3: $\quad \phi^{\ell_{max}} \leftarrow \phi^{\ell_{max}} + e^\ell$ on $\Lambda^\ell - \mathcal{P}(\Lambda^{\ell+1})$
4: $\quad$ **return**
5: **end if**
6: **if** $\ell == \ell_{max}$ **then**
7: $\quad r^\ell = \rho^\ell - \mathcal{L}^{nf,\ell}(\phi^\ell, \phi^{\ell-1})$
8: **end if**
9: $e^\ell \leftarrow 0$; $e^{\ell-1} \leftarrow 0$
10: $e^\ell \leftarrow S^\ell(e^\ell, r^\ell)$ on Λ^ℓ
11: $\phi^{\ell,temp} \leftarrow e^\ell + \phi^{\ell_{max}}$ on $\Lambda^\ell - \mathcal{P}(\Lambda^{\ell+1})$
12: $\hat{r}^\ell \leftarrow r^\ell - \mathcal{L}^{nf,\ell}(e^\ell, e^{\ell-1})$ on Λ^ℓ
13: $r^{\ell-1} \leftarrow \mathcal{P}^\ell \hat{r}^\ell$ on $\mathcal{P}(\Lambda^\ell)$
14: $r^{\ell-1} \leftarrow \rho^{\ell-1} - \mathcal{L}^{\ell-1}(\phi^{\ell,temp}, \phi^{\ell-1}, \phi^{\ell-2})$ on $\Lambda^{\ell-1} - \mathcal{P}(\Lambda^\ell)$
15: MG($\ell - 1, e^{\ell-1}, r^{\ell-1}, \rho^{\ell-1}, \phi^{\ell_{max}}$)
16: $e^\ell \leftarrow e^\ell + \mathcal{R}^{\ell-1} e^{\ell-1}$
17: $r^\ell \leftarrow r^\ell - \mathcal{L}^{nf,\ell}(e^\ell, e^{\ell-1})$ on Λ^ℓ
18: $\bar{e}^\ell \leftarrow 0$
19: $\bar{e}^\ell \leftarrow S^\ell(\bar{e}^\ell, r^\ell)$ on Λ^ℓ
20: $e^\ell \leftarrow e^\ell + \bar{e}^\ell$ on Λ^ℓ
21: $\phi^{\ell_{max}} \leftarrow \phi^{\ell_{max}} + e^\ell$ on $\Lambda^\ell - \mathcal{P}(\Lambda^{\ell+1})$

7.3 The AMR Multigrid Algorithm

The main difference between Alg. 7.3 and the AMR version, Alg. 7.2, is the treatment of the residual

$$\rho_c^{\ell-1} \leftarrow \mathcal{P}^\ell(r_c^\ell - \mathcal{L}_c^\ell e_c^\ell). \tag{7.1}$$

We want to avoid updating the entire composite grid residual r_c^ℓ. The term in parentheses, $r_c^\ell - \mathcal{L}_c^\ell e_c^\ell$, is the residual of the current approximation on composite grid level ℓ. This residual is used in level $\ell-1$ computations and then propagated to computations on coarser levels. If we have the current solution ϕ_c^ℓ on composite grid ℓ, then we can calculate (7.1) by

$$\rho_c^{\ell-1} \leftarrow \mathcal{P}^\ell(\rho_c^\ell - \mathcal{L}_c^\ell \phi_c^\ell), \tag{7.2}$$

thus avoiding the need for an updated r_c^ℓ. The solution ϕ_c^ℓ is part of $\phi_c^{\ell_{max}}$ and is available on $\Lambda_c^{\ell_{max}} - \mathcal{P}(\Lambda^{\ell+1})$, so we can use this shortcut on $\Lambda_c^{\ell_{max}} - \Lambda^\ell$. In fact, since we also have

Algorithm 7.3 Multigrid V cycle with new notation

MG($\ell, e_c^\ell, \rho_c^\ell, \phi_c^{\ell_{max}}$ **)**

1: **if** $\ell == 1$ **then**
2: $\quad e_c^\ell \leftarrow S_c^\ell(e_c^\ell, \rho_c^\ell)$ on Λ^ℓ
3: $\quad$ **return**
4: **end if**
5: **if** $\ell == \ell_{max}$ **then**
6: $\quad r_c^\ell = \rho_c^\ell - \mathcal{L}_c^\ell \phi_c^{\ell_{max}}$
7: **else**
8: $\quad r_c^\ell = \rho_c^\ell$
9: **end if**
10: $e_c^\ell \leftarrow 0$; $e_c^{\ell-1} \leftarrow 0$
11: $e_c^\ell \leftarrow S_c^\ell(e_c^\ell, r_c^\ell)$ on Λ^ℓ
12: $\rho_c^{\ell-1} \leftarrow \mathcal{P}^\ell(r_c^\ell - \mathcal{L}_c^\ell e_c^\ell)$
13: MG($\ell-1, e_c^{\ell-1}, \rho_c^{\ell-1}, \phi_c^{\ell_{max}}$)
14: $e_c^\ell \leftarrow e_c^\ell + \mathcal{R}^\ell e_c^{\ell-1}$
15: $e_c^\ell \leftarrow S_c^\ell(e_c^\ell, r_c^\ell)$ on Λ^ℓ
16: $\phi_c^{\ell_{max}} \leftarrow \phi_c^{\ell_{max}} + e_c^\ell$ on $\Lambda^\ell - \mathcal{P}(\Lambda^{\ell+1})$

$\phi_c^{\ell-1}$ on $\Lambda_c^{\ell_{max}} - \Lambda^\ell$, we can compute the residual

$$\rho_c^{\ell-1} \leftarrow \rho_c^{\ell-1} - \mathcal{L}_c^{\ell-1}\phi_c^{\ell-1} \tag{7.3}$$

directly on $\Lambda_c^{\ell_{max}} - \mathcal{P}(\Lambda^\ell)$. Applying $\mathcal{L}_c^{\ell-1}$ on $\Lambda_c^{\ell_{max}} - \mathcal{P}(\Lambda^\ell)$ does not require any data from $\Lambda^{\ell+1}$, because $\Lambda_c^{\ell_{max}} - \Lambda^\ell$ and $\mathcal{P}(\Lambda^{\ell+1})$ are disjoint (i.e., refinement patches are properly nested within their parent patch), as discussed in §2.5. So, on Λ^ℓ we use (7.1) and project it onto $\mathcal{P}(\Lambda^\ell)$, and on $\Lambda_c^{\ell_{max}} - \Lambda^\ell$ we use (7.3) directly. This is illustrated in Fig. 7.2 for computing the composite residual on levels $\ell = 3$ and $\ell = 2$ before ρ_c^2 is computed. The part of ρ_c^2 on $\Lambda_c^{\ell_{max}} - \mathcal{P}(\Lambda^3)$ can be computed directly, and the part of ρ_c^2 on Λ^2 is projected from an updated residual r_c^3 on Λ^3.

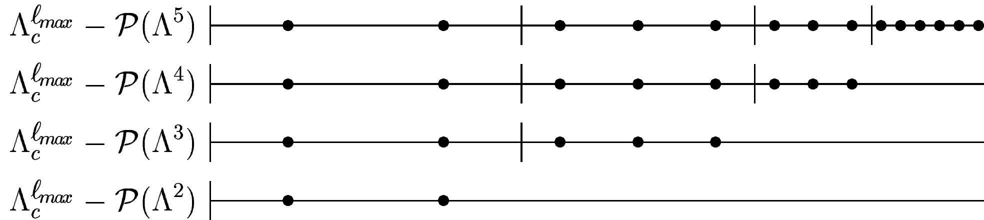


Figure 7.1: Illustration of $\Lambda_c^{\ell_{max}} - \mathcal{P}(\Lambda^{\ell+1})$.

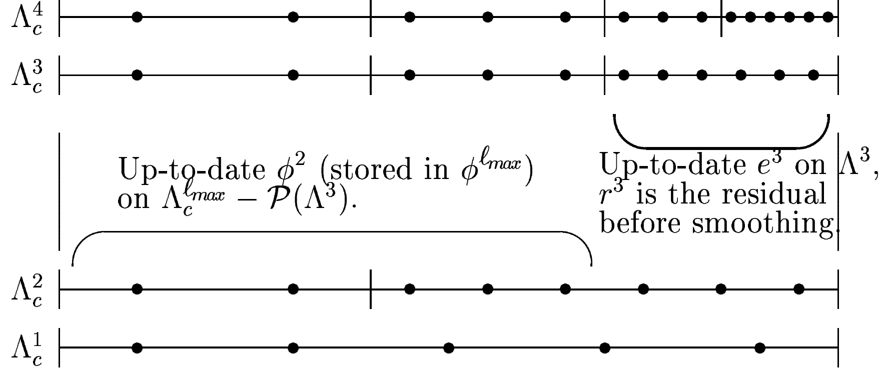


Figure 7.2: Situation on Λ_c^3 before computation of the composite grid residual on Λ_c^2. The rest of the composite grids from Fig. 2.5 are shown for reference.

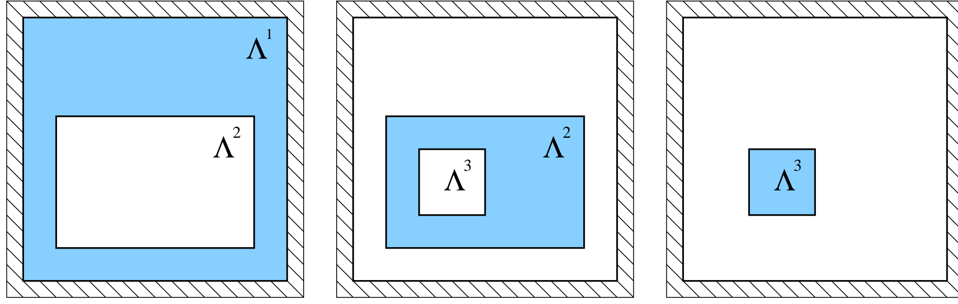


Figure 7.3: The $\mathcal{L}^\ell$ operator shown for $1 \leq \ell \leq 3$. The shading indicates the computational domain $\Lambda^\ell - \mathcal{P}(\Lambda^{\ell+1})$. The hash marks denote the boundary.

So far in the section we have defined things in terms of the composite grid. We want to view the algorithm in terms of patches. So we define two new versions of $\mathcal{L}$ for patches: one that is defined on $\Lambda^\ell - P(\Lambda^{\ell+1})$ and another that is defined on all of Λ^ℓ.

- We first define $\mathcal{L}^\ell(u^{\ell+1}, u^\ell, u^{\ell-1})$ on $\Lambda^\ell - \mathcal{P}(\Lambda^{\ell+1})$ as illustrated in Figure 7.3. We think of $\mathcal{L}^\ell$ as the restriction of $\mathcal{L}_c^\ell$ to the region $\Lambda^\ell - \mathcal{P}(\Lambda^{\ell+1})$. In the interior of Λ^ℓ away from $\mathcal{P}(\Lambda^{\ell+1})$ this is the standard discretization on Λ_c^ℓ. Near the coarse-fine interface between $\Lambda^{\ell-1}$ and Λ^ℓ we interpolate ghost points for u^ℓ using $u^{\ell-1}$ as described in §3.2.2. Near the coarse-fine interface between Λ^ℓ and $\Lambda^{\ell+1}$ we do flux matching using information from $u^{\ell+1}$ to compute the operator as described in §3.2.3.

- We next define $\mathcal{L}^{nf,\ell}(u^\ell, u^{\ell-1})$ on Λ^ℓ, illustrated in Figure 7.4. We think of $\mathcal{L}^{nf,\ell}$ as the restriction of $\mathcal{L}_c^\ell$ to the region Λ^ℓ. In the interior of Λ^ℓ this is the standard discretization on Λ_c^ℓ without any regard for the existence of finer levels. On Λ_c^1 we use the standard discretization in all of Λ^1. On Λ_c^ℓ, $\ell > 1$, we use data from the coarse grid, $\Lambda_c^{\ell-1}$, to interpolate ghost point information as we did for $\mathcal{L}^\ell$, but we do *not* use fine grid data to do flux matching.

It is worth repeating that the motivation for these two new versions of the operator is the fact that the solution and right hand side are stored only on $\Lambda_c^{\ell_{max}}$ whereas the residual and

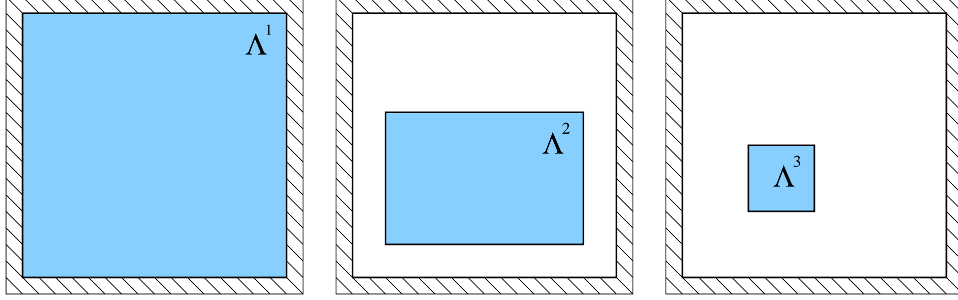


Figure 7.4: The $\mathcal{L}^{nf,\ell}$ operator shown for $1 \leq \ell \leq 3$. The shading indicates the computational domain Λ^ℓ. The hash marks denote the boundary.

correction are stored on the union of all patches,

$$\bigcup_{\ell=1}^{\ell_{max}} \Lambda^\ell.$$

Notice that in the AMR-based multigrid algorithms we present, *nothing* is stored on the entire composite grid hierarchy

$$\bigcup_{\ell=1}^{\ell_{max}} \Lambda_c^\ell.$$

The composite grid hierarchy is a conceptual mechanism used to facilitate the presentation of the algorithms. Hence, in the new definitions of $\mathcal{L}$, we dropped the subscript c and assume the patch based domain restrictions on the variables. This marks the shift from a composite grid based discussion to a patch based discussion. The former was to provide intuition starting from a conventional form multigrid algorithm. The latter is to convey the true form of the implementation of multigrid on an AMR hierarchy.

The split residual computation that was discussed in terms of composite grids above can now be rewritten in terms of patches:

$$r^{\ell-1} \leftarrow r^{\ell-1} - \left(r^\ell - \mathcal{L}^\ell(e^\ell, e^{\ell-1})\right) \text{ on } \mathcal{P}(\Lambda^\ell) \tag{7.4}$$

and

$$r^{\ell-1} \leftarrow \rho^{\ell-1} - \mathcal{L}^{nf,\ell-1}(\phi^\ell, \phi^{\ell-1}, \phi^{\ell-2}) \text{ on } \Lambda^{\ell-1} - \mathcal{P}(\Lambda^\ell). \tag{7.5}$$

In the algorithm, we use $\hat{r}^\ell$ to denote the residual $r^\ell - \mathcal{L}^\ell(e^\ell, e^{\ell-1})$ computed on Λ^ℓ which is projected onto $\mathcal{P}(\Lambda^{\ell-1})$. We have introduced the term $\phi^{\ell,temp}$, a temporary variable to store the ℓth-level solution on $\Lambda^\ell - \mathcal{P}(\Lambda^{\ell+1})$ after the pre-smoothing correction. It is used in the application of (7.5). See Fig. 7.5 for an illustration of $\Lambda^\ell - \mathcal{P}(\Lambda^{\ell+1})$. Note that in (7.4) and (7.5) we assign the residuals to separate residual vectors instead of assigning them to the right hand side. That is necessary in the patch based version because the original right hand side is needed on all of the levels, and because the right hand side is stored in only a subset of the hierarchy, whereas the residual is stored on the entire (but *not* composite) hierarchy, as emphasized above.

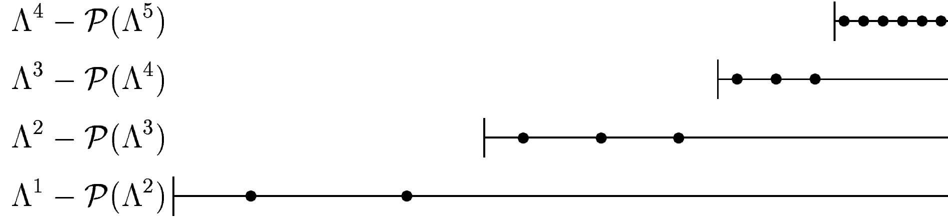


Figure 7.5: Illustration of $\Lambda^\ell - \mathcal{P}(\Lambda^{\ell+1})$.

Now we define a patch based smoother $S^\ell(u^\ell, f^\ell)$ using the new patch based operator $\mathcal{L}^{nf,\ell}$. The smoother S^ℓ is typically a damped Gauss-Seidel iteration using either the natural, red-black, or a multi-color ordering. For a one sided multilevel algorithm, one side uses the identity operator which implies no smoothing. An alternative not investigated here is to use a Krylov subspace rougher.

When smoothing to get the solution to (1.1), we compute S^ℓ pointwise on level ℓ:

$$\phi_{ij}^\ell \leftarrow \phi_{ij}^\ell + \lambda \left(\rho_{ij}^\ell - \mathcal{L}_{ij}^{nf,\ell}(\phi^\ell, \phi^{\ell-1}) \right).$$

When smoothing to get a correction, we compute S^ℓ pointwise on level ℓ without regard for any other level:

$$e_{ij}^\ell \leftarrow e_{ij}^\ell + \lambda \left(r_{ij}^\ell - \mathcal{L}_{ij}^{nf,\ell}(e^\ell, 0) \right),$$

where the damping factor λ is different depending on whether the grid point indexed by (i, j) is on the interior, an edge boundary, or a corner boundary. For a Gauss-Seidel smoother S^ℓ, we use the following damping factors:

$$\lambda_{interior} = \frac{1}{4}h^2, \quad \lambda_{edge} = \frac{1}{6}h^2, \quad \text{and} \quad \lambda_{corner} = \frac{1}{8}h^2.$$

These damping factors are the reciprocals of the corresponding diagonal entries of the discrete operator.

Another mechanism that we introduce in the AMR version of the algorithm is an intermediate correction step. This is for convenience and efficiency. In the first smoothing step of Alg. 7.3, the initial guess for e^ℓ is always zero. This implies that smoothing on Λ^ℓ does not involve data on patches other than Λ^ℓ. This is not the case for the second smoothing step which normally has a nonzero initial guess. In this case, it is necessary to use data from $\Lambda^{\ell-1}$ in order to update the ghost points around the Λ^ℓ patches before updating points on the boundary of the Λ^ℓ patches. To avoid this within the smoother, we reformulate the second smoothing step as a correction. This is a correction $\bar{e}$ to the correction e. Now the smoother can be written so that no data is needed from other patches.

The AMR version of the algorithm is shown in Alg. 7.2.

7.4 Post-Smoothing Only

Alg. 7.2 can be sped up considerably by only doing post-smoothing. This is illustrated in Alg. 7.4. The advantages are three-fold:

1. The initial guess on most coarse patches does not have to be set (and is 0).
2. The residual on most patches is just the right hand side.
3. Interpolation and add of a correction on a fine patch is replaced by just interpolation.

Each of these improvements seems trivial. However, each forces a large amount of data to pass through cache, which takes a considerable percentage of the total run time. In addition, nearly half of the computation is eliminated:

1. There is no need for a temporary correction $\phi^{\ell,temp}$ on the projection side of the V cycle.
2. $\hat{r}_c$ does not need to be computed.
3. The residual is only computed on one side of the V cycle.
4. There is no correction after interpolation and no need for an intermediate correction step before updating $\phi^{\ell_{max}}$.
5. The residual is not updated on the interpolation side of the V cycle.

Doing post-smoothing only is a substantial change over [70] and Alg. 7.2. It is especially useful when implementing cache aware algorithms as discussed in Ch. 8, because there should be better cache effects when more smoothing iterations are done consecutively.

Except for the projection side of the first V cycle and the correction side of the last V cycle, the case of doing post-smoothing only can be viewed as the usual pre-/post-smoothing case but with the post-smoothing iterations of one V cycle combined with the pre-smoothing iterations of the following V cycle. Hence, if we choose to have the same number of smoothing iterations per level in the post-smoothing only case as in the pre-/post-smoothing case, then there are twice as many *contiguous* smoothing iterations per level in the post-smoothing only case (albeit the same *total* number of smoothing iterations per level). That leads to better cache effects.

Algorithm 7.4 Multigrid V cycle with post-smoothing only

MG $(\ell, e^\ell, r^\ell, \rho^\ell, \phi^{\ell_{max}})$

1: **if** $\ell == \ell_{max}$ **then**
2: $\quad r^\ell = \rho^\ell - \mathcal{L}^{nf,\ell}(\phi^\ell, \phi^{\ell-1})$
3: **end if**
4: **if** $\ell == 1$ **then**
5: $\quad e^\ell \leftarrow 0$
6: $\quad e^\ell \leftarrow S^\ell(e^\ell, r^\ell)$ on Λ^ℓ
7: **else**
8: $\quad r^{\ell-1} \leftarrow \mathcal{P}^\ell r^\ell$ on $\mathcal{P}(\Lambda^\ell)$
9: $\quad r^{\ell-1} \leftarrow \rho^{\ell-1} - \mathcal{L}^{\ell-1}(\phi^\ell, \phi^{\ell-1}, \phi^{\ell-2})$ on $\Lambda^{\ell-1} - \mathcal{P}(\Lambda^\ell)$
10: $\quad$ MG $(\ell-1, e^{\ell-1}, r^{\ell-1}, \rho^{\ell-1}, \phi^{\ell_{max}})$
11: $\quad e^\ell \leftarrow \mathcal{R}^\ell e^{\ell-1}$ {Includes interpolation of ghost points.}
12: $\quad e^\ell \leftarrow S^\ell(e^\ell, r^\ell)$ on Λ^ℓ
13: **end if**
14: $\phi^{\ell_{max}} \leftarrow \phi^{\ell_{max}} + e^\ell$ on $\Lambda^\ell - \mathcal{P}(\Lambda^{\ell+1})$

Chapter 8

Cache Optimizations

Since the early 1980's, processors have sped up 5 times faster per year than memory. Multilevel memories, using *memory caches*, were developed to compensate for the uneven speedups in hardware. Essentially all computers today, from laptops to distributed memory supercomputers, use cache memories in an attempt to keep the processors busy. By the term *cache*, we mean a fast memory unit closely coupled to the processor [44, 81]. In the interesting cases, the cache is further divided into many *cache lines*. Each cache line holds copies of contiguous locations of main memory. Any given pair of cache lines may hold data from entirely separate regions of main memory. A good cache primer for solving PDEs can be found in [28, 30, 61]. Also, see [24, 1, 2, 40, 51, 101] for similar work and related topics.

Tiling is the process of decomposing a computation into smaller blocks and doing all of the computing in each block one at a time. Tiling is an attractive method for improving data locality. In some cases, compilers can do this automatically [104, 84, 105, 4]. However, this is rarely the case for realistic scientific codes. In fact, even for simple examples, manual help from the programmers is necessary [24].

Language standards interfere with compiler optimizations. Due to requirements about loop variable values during computation, compilers are not allowed to fuse nested loops into a single loop. In part, it is due to coding styles that make very high level code optimization (nearly) impossible [83].

8.1 Cache Aware Gauss-Seidel

Consider naturally ordered Gauss-Seidel restricted to matrices A_j which are based on discretization methods which are local to only 3 neighboring rows of the grid. Partition the grid into blocks of ℓ rows, and let m be the number of smoothing iterations required. It is necessary that $\ell + m - 1$ rows of an $N \times N$ grid G fit entirely into cache simultaneously and that $m < \ell$.

There are two special cases to the cache aware algorithm: the first block of rows and the rest of the blocks.

The first case is for the first ℓ rows of the grid. The data associated with rows 1 to ℓ is brought into cache. The data in rows 1 to $\ell - m + 1$ are updated m times, and the data in rows j, $\ell - m + 2 \leq j \leq \ell$, are updated $\ell - j + 1$ times.

The second case is for the rest of the blocks of ℓ rows of the grid. Once the first block of grid rows is partially updated, we have a second block to update and must also finish updating the first block of grid rows. After the ith update in the second block, we can go back and update rows ℓ down to $\ell - i + 1$ in the first block of rows, always performing the updates in the order that preserves the dependencies in the standard iteration (so that the

cache aware iteration achieves bitwise the same answer as the standard iteration). This procedure is repeated in the remaining blocks of rows until all the blocks are updated. In effect, this is a domain decomposition methodology applied to the standard iteration. The result is that m updates are done while bringing all the data through cache only one time.

8.2 Cache-Aware V-Cycle

The cache aware smoother discussed above is just one component of multigrid. The smoother can be made cache aware and plugged into the multigrid algorithm without changing other parts of the algorithm. This section describes how three components of the algorithm (the smoother, the residual computation, and the ghost point interpolation) can be combined and interleaved in a cache aware manner to give even better performance.

8.2.1 Combined smoother

Integrating the residual computation with the cache-aware smoother can give better cache-effects in multigrid. Call this the *combined smoother*. The combined smoother is implemented for the post-smoothing only version of the algorithm. The combined smoother must also interleave the ghost point interpolation so that the residual computation at patch boundaries has up-to-date ghost points based on the latest smoothing updates.

The combined smoother computes the residual to be used in the next V cycle. It can only compute the part of the residual that is not covered by finer patches. On the projection side of the V cycle, the projection of the part of the residual that is from finer patches must now include the application of the flux matching procedure because information needed for the flux matching is not available when the combined smoother is called.

A complication to interleaving the residual computation with the cache aware smoother is that, in the V cycle, the residual is updated *and* concurrently used as the right hand side for the Gauss-Seidel updates. So the residual updates need to be postponed long enough to avoid changing the right hand side for a subsequent Gauss-Seidel update.

8.2.2 Effects of the coefficient matrix

The coefficient matrix has an effect on the cache optimizations since it has to go through cache along with the solution and right hand side. In an attempt to minimize the effect, we tried storing the right hand side in the coefficient matrix in an interleaved manner. Hence, for a given grid point, the coefficients and the right hand side needed to update that point are contiguous in memory. However, the split residual computation complicates this approach. On every level, there are parts of the domain where the residual has to be computed from the right hand side, *and* there are parts of the domain where the residual is projected from finer levels. Depending on the context, the right hand side entries in the coefficient matrix might need to be either the original right hand side or the current residual. Updating the state for a given context involves copies which slow the algorithm.

Algorithm 8.1 Multigrid V cycle with integrated interpolation and residual computation.

MG($\ell, e^\ell, r^\ell, \rho^\ell, \phi^{\ell_{max}}$ **)**

1: **if** $\ell == \ell_{max}$ **then**
2: $\quad r^\ell = \rho^\ell - \mathcal{L}^{nf,\ell}(\phi^\ell, \phi^{\ell-1})$
3: **end if**
4: **if** $\ell == 1$ **then**
5: $\quad e^\ell \leftarrow 0$
6: $\quad e^\ell \leftarrow S^\ell(e^\ell, r^\ell)$ on Λ^ℓ
7: $\quad \phi^{\ell_{max}} \leftarrow \phi^{\ell_{max}} + e^\ell$ on $\Lambda^\ell - \mathcal{P}(\Lambda^{\ell+1})$
8: **else**
9: $\quad r^{\ell-1} \leftarrow \mathcal{P}^\ell r^\ell$ on $\mathcal{P}(\Lambda^\ell)$ {Includes flux matching at the interfaces.}
10: $\quad$ MG($\ell - 1, e^{\ell-1}, r^{\ell-1}, \rho^{\ell-1}, \phi^{\ell_{max}}$)
11: $\quad CombinedSmoother(\ell, e^{\ell-1}, e^\ell, \rho^\ell, r^\ell, r^{\ell-1})$
12: **end if**

Algorithm 8.2 Combined Smoother.

$CombinedSmoother(\ \ell, e^{\ell-1}, e^\ell, \rho^\ell, r^\ell, r^{\ell-1}\ )$

1: {The following operations are blocked and interleaved in a cache aware manner.}
2: $e^\ell \leftarrow \mathcal{R}^\ell e^{\ell-1}$ {Includes interpolation of ghost points.}
3: $e^\ell \leftarrow S^\ell(e^\ell, r^\ell)$ on Λ^ℓ
4: $r^\ell \leftarrow \rho^\ell - \mathcal{L}^\ell(\phi^{\ell+1}, \phi^\ell, \phi^{\ell-1})$ on $\Lambda^\ell - \mathcal{P}(\Lambda^{\ell+1})$ {With the flux matching procedure omitted.}
5: $\phi^{\ell_{max}} \leftarrow \phi^{\ell_{max}} + e^\ell$ on $\Lambda^\ell - \mathcal{P}(\Lambda^{\ell+1})$

Results in the AMR context are better when the right hand side and residual are separate from the coefficient matrix.

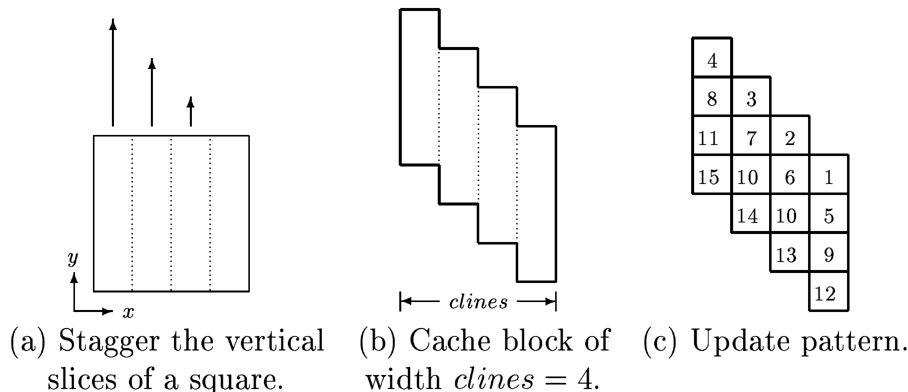
8.3 Details

This section presents the cache optimization technique in more detail. The focus here is to analyse the more sophisticated multidimensional blocking approach. This approach is necessary in 2D, as opposed to the simple row-based blocking discussed above, when patches are large enough that a few rows, or even one row, does not fit into cache. Multidimensional blocking is almost always needed in 3D, as opposed to layer-based blocking, because realistic problems are rarely small enough that several layers of a 3D patch fit into cache. Note that the skewed 3D blocking approach described in the 3D section below has never been

formulated before, to the knowledge of the author.

8.3.1 2D

In 2D, row based partitioning as discussed in §8.1 is sometimes sufficient. For very large patches, however, even one single row might not fit into cache. Then partitioning in both dimensions is necessary. We use a skewed block partitioning as illustrated in Fig. 8.1 [36, 59, 100].



(a) Stagger the vertical slices of a square. (b) Cache block of width $clines = 4$. (c) Update pattern.

Figure 8.1: 2D Cache Block

Notice that this is less efficient for small patches because some grid points have to be fetched into cache twice. In particular, the $NumIts - 1$ rows between sweeps of the cache block are fetched into cache twice. Call these rows a *cache seam*. A cache seam is illustrated in Fig. 8.2 by the cells with dotted boundaries. The cache block shown in this illustration is designed to produce four updates. It is shown toward the beginning of a sweep across the grid in the x-direction. The numbers printed in the cells in Fig. 8.2 specify how many updates have occured prior to the application of updates in the current cache block. Only the seam associated with the current rows being updated is shown in Fig. 8.2. The full set of seams for a 20×20 patch is shown in Fig. 8.3. The seam is denoted by the dotted cells. The *seam points* have to be fetched into cache twice, versus the other points which only have to be fetched into cache once. We want to minimize the number of seam points. This can be done by making the cache block taller as in Fig. 8.4. We maintain the staggered shape of the cache block. The width of the seams is constant at $NumIts - 1$. Taller cache blocks have a smaller proportion of seam points to total points, so the total number of seam points in the patch is less. Fig. 8.5 shows the full pattern of seams in the case of a cache block with 5 non-seam rows. The ratio of seam points to non-seam points is much lower. Most points have to be fetched into cache only one time in this case.

8.3.2 3D

In 3D, layer based partitioning (analogous to row based partitioning in 2D) usually requires cache blocks that are too large to fit into cache. Even one layer is often too large to fit into cache. Therefore, it is necessary to partition in more than one dimension. This gives cache

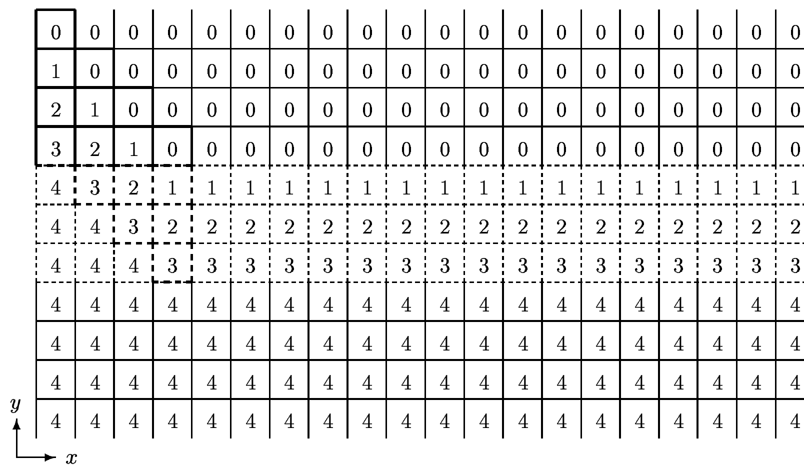


Figure 8.2: 2D Cache Block Seam

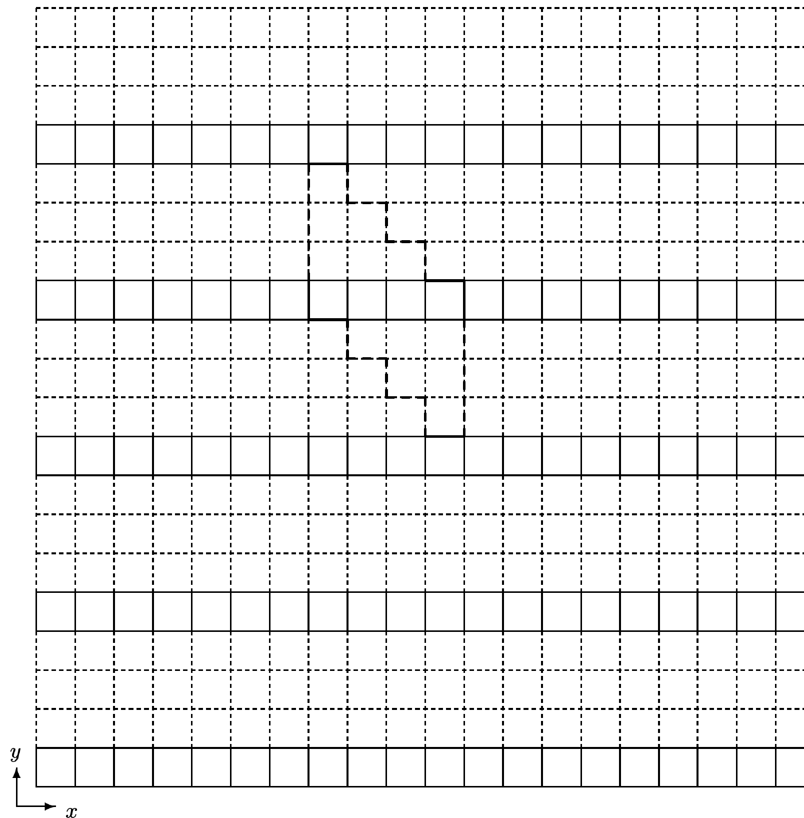
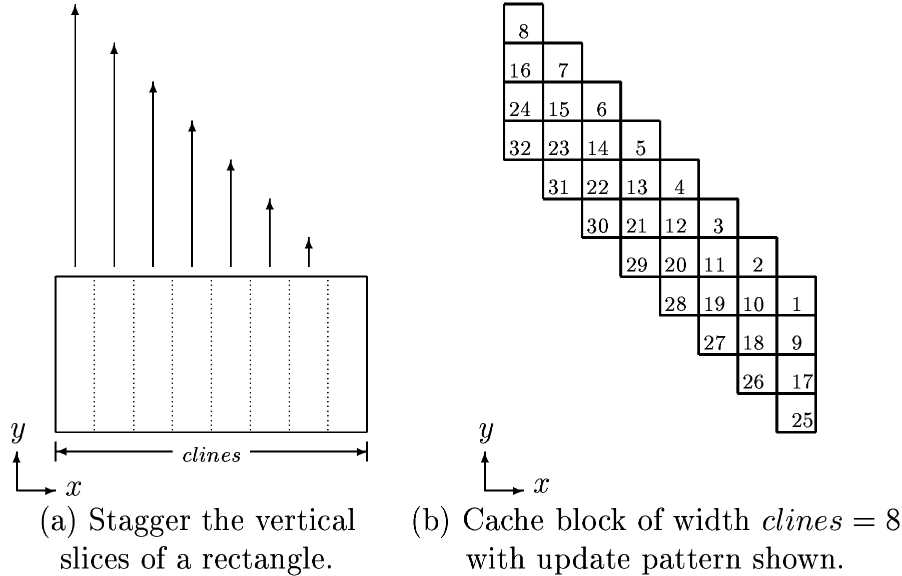


Figure 8.3: 2D Cache Block Seams, Full 20×20 Patch Picture With Representative Cache Block.



(a) Stagger the vertical slices of a rectangle.

(b) Cache block of width $clines = 8$ with update pattern shown.

Figure 8.4: 2D Tall Cache Block

blocks that are small chunks of the domain. A proper update schedule can be accomplished with a cache block like the one illustrated in Fig. 8.6. The illustration shows how the cache block is formed starting from a cube. The slices of the cube are staggered to correspond to the staggering of the updates. The order that the points in the cache block are updated is illustrated, slice by slice, in Fig. 8.7. The cache block sweeps the patch in the directions corresponding to a natural ordering of the grid points. Notice that there are grid points in the seams that are fetched into cache three times. In order to minimize the number of seam points, we allow the dimensions of the cache block to be non-isotropic. Assume that the natural ordering is the x-direction first followed by the y-direction and then the z-direction. There are no seams between successive cache blocks in the x-direction, so the span of the cache block in the x-direction $clinesx$ does not matter, in terms of cache misses. Hence, $clinesx$ should be as small as possible. That leaves as much space as possible to extend the cache block in the other directions where there are seams. A greater span in the y- and z-directions, $clinesy$ and $clinesz$, leads to fewer seams and, thus, fewer grid points that get fetched into cache multiple times.

8.3.3 Update Pattern for Combined Smoother

The update pattern for the combined smoother is shown in Fig. 8.8. It is illustrated only for the 2D case. Ghost points need to be interpolated when there is a residual computation that depends on them but only after all the smoothing updates have been applied to the fine grid points that are used in the interpolation. The residual updates are denoted by the dark circles ● in the illustration, and the potential ghost points are denoted by the dashed boxes ⬚ . The template traverses a patch applying updates as it goes, and, depending on where it is in the domain, the actual ghost points that need to be computed vary. In particular, it is only when the template is at a patch boundary that ghost points need to

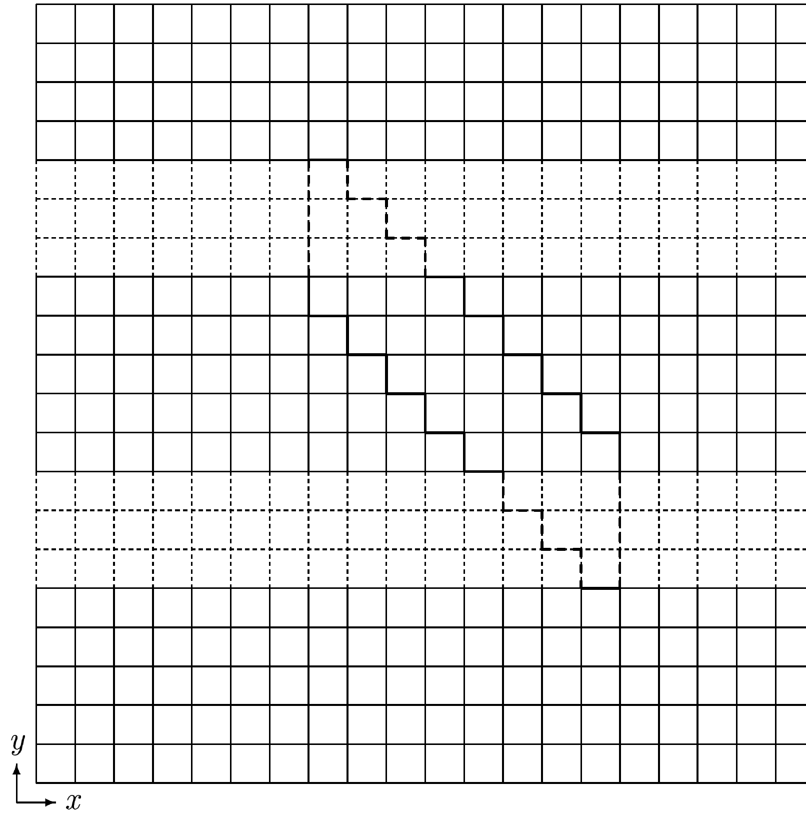


Figure 8.5: 2D Tall Cache Block Seams, Full 20×20 Patch Picture With Representative Cache Block.

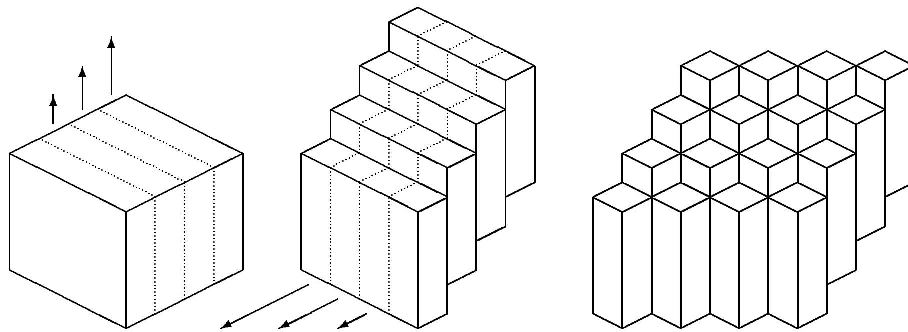


Figure 8.6: 3D Cache Block

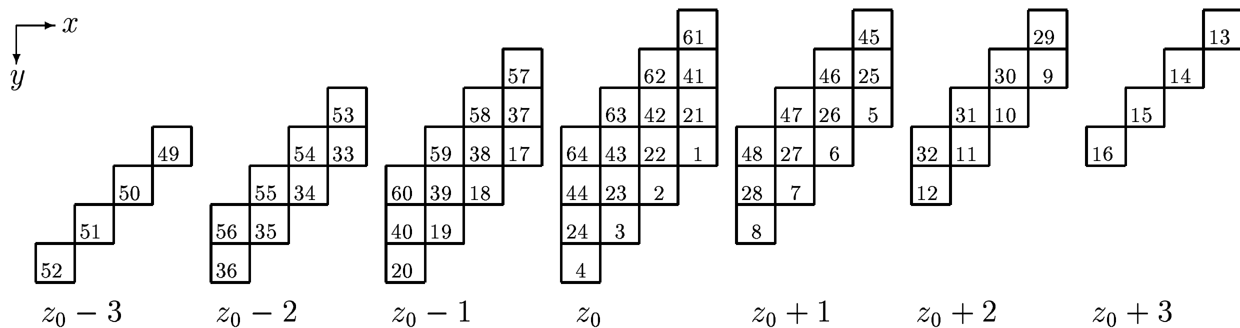


Figure 8.7: 3D Cache Block Update Pattern. This shows xy-slices of the 3D cache block, starting in the negative direction (bottom of the cache block) and going toward positive (top of the cache block).

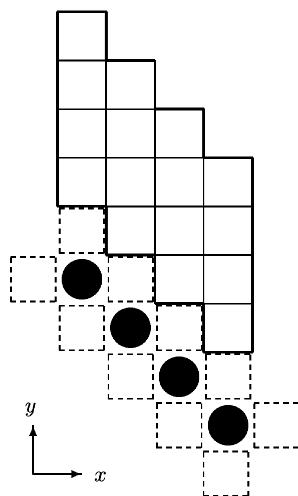


Figure 8.8: Template for interleaving ghost point interpolation and residual computation with cache aware smoother in 2D. The residual updates are denoted by the dark circles. The potential ghost points are denoted by the dashed boxes.

be computed, and then only for residual updates that are on the boundary. This pattern of inserting ghost point computations just prior to residual computations is trivial to extend to 3D.

Chapter 9

Numerical Results

This chapter gives numerical results. Results are shown for both constant and variable coefficient problems.

9.1 Constant

Results are shown for the hierarchies illustrated in Fig. 9.2 and also on a full domain refinement hierarchy (i.e., regular, non-AMR multigrid). The refinement patterns are not meaningful to the nature of the problem being solved here. They are contrived merely to demonstrate the behavior of the algorithm.

For the adaptive refinement cases, we do full domain refinements from a coarse grid for a few levels before applying the adaptive refinement. The coarse grid is 8×32, in our example. The first adaptive refinement is on level 6 which is a 512×2048 grid in our example. Another way of explaining this is that we do adaptive refinement on a 512×2048 base grid and then use geometric multigrid as the solver on that base grid.

The base grid discretizes the unit square $[0, 1]^2$. We solve the Poisson equation with the right hand side chosen so that the solution is

$$u(x, y) = \sin(\pi x) \sin(\pi y/4) x e^{x^2+(y/4)^2}.$$

The initial guess on the base grid is $u = 0$, and we iterate until the composite residual is smaller than 10^{-6} times the norm of the composite right hand side:

$$||r_c|| < 10^{-6} ||\rho_c||.$$

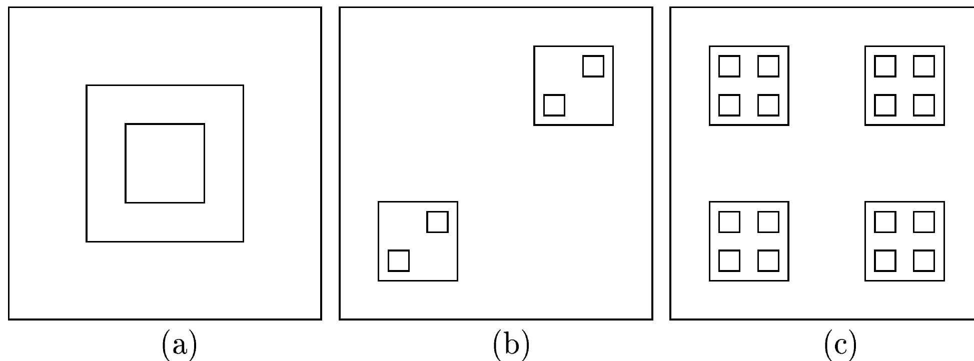


Figure 9.1: Refinement patterns: (a) One refinement per patch. (b) Two refinements per patch. (c) Four refinements per patch.

2D	CA	CARES
Full	1.1554	1.3350
One	1.1032	1.4807
Two	1.0981	1.5501
Four	1.0807	1.4226

Table 9.1: Itanium 1, Standard(0,4) Versus CA(0,4) and CAMG(0,4)

2D	CA	CARES
Full	1.1123	1.2754
One	1.0788	1.4564
Two	1.0766	1.5346
Four	1.0578	1.4246

Table 9.2: Itanium 2, Standard(0,4) Versus CA(0,4) and CAMG(0,4)

The tables show results for the AMR multilevel method employing the cache aware smoother, labeled *CA*, and the combined smoother, labeled *CAMG*, compared with a standard implementation of the smoother. Speedups are based on elapsed wall clock time. The timings include only the solution procedure. They do not include initialization of the hierarchy and right hand sides.

The main speedups we are interested in are shown in Table 9.6. This table shows the speedups of the original standard AMR multigrid V-Cycle, Standard(2,2), compared to the cache aware post-smoothing only version, CAMG(0,4). We see speedups consistently over a factor of 2 and even close to a factor of 3 in some cases.

The contribution from just the cache optimizations alone is shown for the Itanium 1, Itanium 2, Pentium III and Pentium IV in Tables 9.1, 9.2, 9.3 and 9.4. These experiments show the speedups associated with the cache optimizations for the V(0,4) cycle. The CA(0,4) cycle uses just the cache aware smoother plugged into the multigrid algorithm. Those speedups are fairly insignificant in most cases. The CAMG(0,4) cycle shows the speedups associated with interleaving the residual computation with the smoother in a cache aware manner. These speedups are more substantial, up to nearly a factor of 2.

The contribution from merging the smoothing steps on each level that is accomplished by the post-smoothing only algorithm is shown in Table 9.5. This shows the speedup associated with doing post-smoothing only, CA(0,4), versus doing pre-smoothing *and* post-smoothing, CA(2,2). Note that we do the same total number of smoothing iterations per level in both cases. In particular, the pre-/post-smoothing case uses 2 smoothing iterations on each side. The post-smoothing only case uses 4 smoothing iterations only on the correction side of the V-Cycle. These examples were run with the cache aware smoothers. The average time per V-Cycle is about 1.4 to 2 times faster in the post-smoothing only runs.

2D	CA	CARES
Full	1.3574	1.6344
One	1.2399	1.7575
Two	1.2221	1.8417
Four	1.2031	1.7002

Table 9.3: Pentium III, Standard(0,4) Versus CA(0,4) and CAMG(0,4)

2D	CA	CARES
Full	1.1432	1.3355
One	1.1310	1.3907
Two	1.1040	1.4356
Four	1.0849	1.2824

Table 9.4: Pentium IV, Standard(0,4) Versus CA(0,4) and CAMG(0,4)

2D	IA1	IA2	PIII	PIV
Full	1.7606	1.7797	2.0714	1.8841
One	1.4810	1.5388	1.6885	1.4752
Two	1.4933	1.5111	1.6513	1.4808
Four	1.4715	1.4956	1.6367	1.4409

Table 9.5: CA(2,2) Versus CA(0,4)

2D	IA1	IA2	PIII	PIV
Full	2.1824	2.1442	2.8288	2.3216
One	2.1038	2.1615	2.6543	1.9278
Two	2.2170	2.2448	2.7539	2.0469
Four	2.0226	2.0887	2.5493	1.7878

Table 9.6: Standard(2,2) Versus CAMG(0,4)

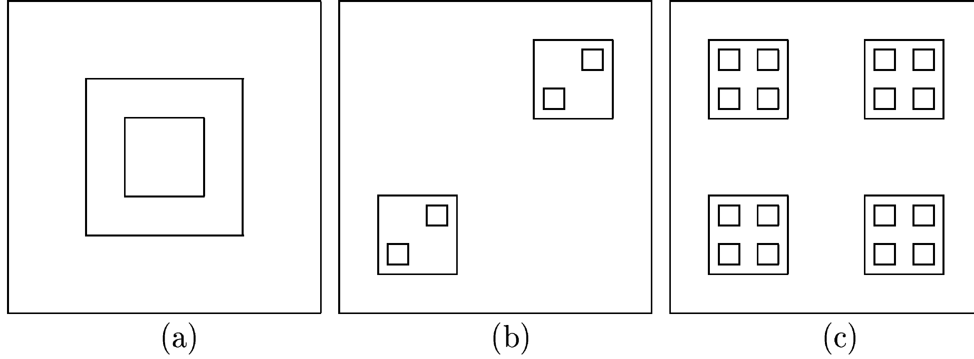


Figure 9.2: (a) One refinement per patch. (b) Two refinements per patch. (c) Four refinements per patch.

	Cache Aware Smoother	Cache Aware MG
Full Domain	1.3530	1.6335
One Patch	1.2522	1.7120
Two Patches	1.2550	1.8221
Four Patches	1.2415	1.6780

Table 9.7: Itanium 1 speedups versus standard multigrid.

9.2 Variable

This section shows results on the hierarchies illustrated in Fig. 9.2 and also on a full domain refinement hierarchy (i.e., regular, non-AMR multigrid). The refinement patterns are not meaningful to the nature of the problem being solved here. They are contrived merely to demonstrate the behavior of the algorithm. The AMR base grid in 9.2(a) is 128×512 grid points, and there are three levels of refinement above that (although only two are illustrated). The AMR base grid in 9.2(b) and 9.2(c) is 256×1024 grid points, and there are four levels of refinement above that (although only two are illustrated). Geometric multigrid is used as the solver on the AMR base grid. The full domain refinement case starts on an 8×32 base grid and has a total of 7 grid levels.

The tables show results for the AMR multilevel method employing the *cache-aware* smoother and the *combined smoother* (as described in §8.2.1) compared with a standard implementation of the smoother. We call the combined smoother case *cache aware multigrid.*

The base grid discretizes the rectangle $[0, 1] \times [0, 4]$. The right hand side of the Poisson equation is chosen so that the solution is $u(x, y) = \sin(\pi x) \sin(\pi y/4) x e^{x^2+(y/4)^2}$ and the coefficient is $a(x, y) = 1 + \sin(\pi x) \sin(\pi y/4) x e^{x^2+(y/4)^2}$. The initial guess on the base grid is $u = 0$, and the convergence criteria is $||r_c|| < 10^{-6} ||\rho_c||$.

Table 9.7 shows the results of the set of experiments run on an Itanium 1. Table 9.8 shows the results of the set of experiments run on a Itanium 2. Table 9.9 shows the results of the set of experiments run on a Pentium III. Table 9.10 shows the results of the set of experiments run on a Pentium IV. These experiments show the speedups associated with

	Cache Aware Smoother	Cache Aware MG
Full Domain	1.2080	1.4057
One Patch	1.1697	1.5791
Two Patches	1.1519	1.6245
Four Patches	1.1352	1.5176

Table 9.8: Itanium 2 speedups versus standard multigrid.

	Cache Aware Smoother	Cache Aware MG
Full Domain	1.4339	1.6402
One Patch	1.3500	1.8728
Two Patches	1.3161	1.8257
Four Patches	1.1478	2.1175

Table 9.9: Pentium III speedups versus standard multigrid.

	Cache Aware Smoother	Cache Aware MG
Full Domain	1.1434	1.3000
One Patch	1.0675	1.3379
Two Patches	1.0713	1.4019
Four Patches	1.0548	1.2758

Table 9.10: Pentium IV speedups versus standard multigrid.

Machine	Speedup
Pentium III	2.3431
Pentium IV	1.2298
Itanium 1	1.8721
Itanium 2	1.5209

Table 9.11: Speedups for the cache aware smoother alone.

the cache optimizations. The timings measure only the solution procedure. They do not include initialization of the hierarchy, coefficient matrix and right hand side. The total speedups are up to about a factor of two. Table 9.11 shows the speedups associated with performing smoothing alone, outside of the context of a multilevel method.

The speedups (not shown) for doing post-smoothing only versus doing pre-smoothing *and* post-smoothing are around 20% with the cache aware smoother. Note that both cases do the same total number of smoothing iterations per level. In particular, the pre-/post-smoothing case uses 2 smoothing iterations on each side of the V. The post-smoothing only case uses 4 smoothing iterations on only one side of the V.

Chapter 10

Conclusions and Future Directions

This document presented a combination of adaptive refinement [11, 12, 87, 92] and multilevel [21, 70, 73] procedures to solve variable coefficient elliptic boundary value problems of the form

$$\begin{cases} \mathcal{L}(\phi) = \rho \text{ in } \Omega, \\ \mathcal{B}(\phi) = \gamma \text{ on } \partial\Omega. \end{cases}$$

The focus of this research is on the effects of cache aware algorithms, i.e., algorithms designed to minimize the number of times data goes through cache. Cache aware algorithms should be more efficient because cache memory is much faster than main memory, so the CPU can be kept more busy when it is getting data from cache memories.

Chapter 2 provided some background material about grids, discretization of PDEs, iterative solvers, multigrid (MG), and adaptive mesh refinement (AMR).

Chapter 3 presented the basic tools used in the 2D version of the algorithm: stencils, ghost point computation, and flux matching.

Chapter 4 compared the flux matching approach from Chapter 3 with other ways of approximating the fluxes across the interface. In addition, a single formula (bypassing the ghost point interpolation and averaging of fluxes) is derived. This is done first by expanding and simplifying the ghost point interpolation and avereaging computations. Then a Taylor series based derivation is also shown.

Chapter 5 presented the basic tools used in the 3D version of the algorithm: stencils, ghost point computation, and flux matching.

Chapter 7 described the multilevel adaptive mesh refinement algorithm. It started by stating a traditional (non-AMR) formulation of multigrid and then outlined the modification that need to be made in order to formulate the AMR version of multigrid.

Chapter 8 discussed cache optimizations. Processors are much faster than memory. Multilevel memory hierarchies, using cache memory, were developed to compensate for this. Cache aware algorithms modify the code to take better advantage of the cache memory mechanism.

Chapter 6 revisited the ghost point interpolation process. The process was introduced in Chapters 3 and 5 in a way that is useful for describing how the ghost points are computed. That is not efficient for the implementation, though, especially in 3D. This chapter derives a single, simple and efficient formula for computing the ghost points in both 2D and 3D.

Chapter 9 presents numerical results showing speedups associated with cache aware smoothers, integration of the residual computation with the cache aware smoother. and modifying the algorithm to do post-smoothing only. Both the cache aware smoothers and the integration of the residual computation give good speedups in many cases. The integration of the residual computation is especially useful for getting good speedups on AMR

hierarchies. Modifying the algorithm to do post-smoothing only is also key to realizing good performance in the AMR context. It takes better advantage of the cache aware smoother since the smoothing iterations on each level are all contiguous.

Future plans for this research include 3D optimizations, parallelization, clustering algorithms and load balancing, and support for more general types of grids.

A lot of work has been done on a 3D version of the code, as shown in previous chapters. Getting speedups from the cache optimizations will require considerably more work beyond the time frame of this thesis.

Parallelization is a high priority, after the 3D code is ready. There are already some simple parallel mechanisms in the code, and Zoltan [13] has been incorporated and tested for a trivial problem. More sophisticated and general parallel support will be the next big phase of this project.

One topic that has not been considered at all in this project, but which we would like to consider in the future, is the process of erecting a hierarchy of adaptively refined grids in the first place. To date, we are assuming that the hierarchy exists and are concerned with the solution procedure on that hierarchy. To develop a more comprehensive package, we may incorporate clustering algorithms and mechanisms for building a hierarchy from scratch and modifying the hierarchy during the solution procedure based on the convergence properties of the solution.

We would also like to extend the notions described in this dissertation to support non-orthogonal grid lines (e.g. skewed or rotated patches) and curvalinear grids.

Bibliography

[1] *Blas (basic linear algebra subprograms)*. In URL http://www.netlib.org/blas.

[2] *Lapack (linear algebra package)*. http://www.netlib.org/lapack/.

[3] S. Agmon, *Lectures on Elliptic Boundary Value Problems*, Van Nostrand Reinhold, New York, 1965.

[4] D. F. Bacon, S. L. Graham, and O. J. Sharp, *Compiler transformations for high-performance computing*, ACM Computing Surveys, 26 (1994), pp. 345–420.

[5] S. B. Baden, *Structured Adaptive Mesh Refinement (Samr) Grid Methods*, vol. 117 of Ima Volumes in Mathematics and Its Applications, Springer– Verlag, New York, 1999.

[6] R. E. Bank and A. H. Sherman, *The use of adaptive grid refinement for badly behaved elliptic partial differential equations*, in Mathematics and Computers in Simulation, XXII, North-Holland, Amsterdam, 1980, pp. 18–24.

[7] ——, *An adaptive multi–level method for elliptic boundary value problems*, Computing, 26 (1981), pp. 91–105.

[8] P. Bastian and G. Wittum, *On robust and adaptive multi–grid methods*, in Multigrid Methods IV, Proceedings of the Fourth European Multigrid Conference, Amsterdam, July 6-9, 1993, vol. 116 of ISNM, Basel, 1994, Birkhäuser, pp. 1–17.

[9] M. J. Berger, *Data structures for adaptive mesh refinement*, in Adaptive Computational Methods for Partial Differential Equations, I. Babuška, J. Chandra, and J. E. Flaherty, eds., SIAM, Philadelphia, 1984.

[10] ——, *Data structures for adaptive grid generation*, SIAM J. Sci. Stat. Comp., 7 (1986), pp. 904–916.

[11] M. J. Berger and P. Colella, *Local adaptive mesh refinement for shock hydrodynamics*, J. Comput. Phys., 82 (1989), pp. 64–84.

[12] M. J. Berger and J. Oliger, *An adaptive mesh refinement for hyperbolic partial differential equations*, J. Comput. Phys., 53 (1984), pp. 484–512.

[13] E. Boman, K. Devine, R. Heaphy, B. Hendrickson, W. F. Mitchell, M. S. John, and C. Vaughan, *Zoltan*. In URL http://www.cs.sandia.gov/Zoltan.

[14] A. Brandt, *Multi–level adaptive technique (MLAT) for fast numerical solution to boundary value problems*, in Proceedings of the Third International Conference on Numerical Methods in Fluid Mechanics, H. Cabannes and R. Teman, eds., vol. 18 of Lecture Notes in Physics, Berlin, 1973, Springer–Verlag, pp. 82–89.

[15] ——, *Multi–level adaptive solutions to boundary–value problems*, Math. Comp., 31 (1977), pp. 333–390.

[16] ——, *Multi–level adaptive techniques (MLAT) for partial differential equations: ideas and software*, in Mathematical Software III, J. R. Rice, ed., Academic Press, New York, 1977, pp. 277–318.

[17] ——, *Multi–level adaptive finite–element methods. I. Variational problems*, in Special Topics of Applied Mathematics, J. Frehse, D. Pallaschke, and U. Trotenberg, eds., North–Holland, Amsterdam, 1979, pp. 91–128.

[18] ——, *Multi–level adaptive techniques (MLAT) for singular–perturbation problems*, in Numerical Analysis of Singular Perturbation Problems, P. W. Hemker and J. J. H. Miller, eds., Academic Press, New York, 1979, pp. 53–142.

[19] ——, *Multi–level adaptive computations in fluid dynamics*, AIAA J., 18 (1980), pp. 1165–1172.

[20] ——, *Multi–level adaptive finite–element methods I: Variational problems*, in Special Topics of Applied Mathematics, North–Holland, Amsterdam, 1991, pp. 91–128.

[21] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A Multigrid Tutorial*, SIAM Books, Philadelphia, 2000. Second edition.

[22] H.-J. Bungartz, *An adaptive Poisson solver using hierarchical bases and sparse grids*, in Proceedings of the IMACS International Symposium on Iterative Methods in Linear Algebra, Brussels, April, 1991, Amsterdam, 1992, Elsevier.

[23] G. F. Carey, *Grid Generation, Refinement, and Redistribution*, Wiley, 1993.

[24] J. Dongarra and R. C. Whaley, *Automatically tuned linear algebra software.* In URL http://www.netlib.org/atlas, 1999.

[25] W. Dörfler, *A robust adaptive strategy for the non-linear Poisson equation*, Computing, 55 (1995), pp. 289–304.

[26] C. C. Douglas and J. Douglas, *A unified convergence theory for abstract multigrid or multilevel algorithms, serial and parallel*, SIAM J. Numer. Anal., 30 (1993), pp. 136–158.

[27] C. C. Douglas, J. Douglas, and D. E. Fyfe, *A multigrid unified theory for non-nested grids and/or quadrature*, E. W. J. Numer. Math., 2 (1994), pp. 285–294.

[28] C. C. Douglas, G. Haase, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss, *Portable memory hierarchy techniques for pde solvers, part i*, SIAM News, 33 (2000), pp. 1, 8–9.

[29] C. C. Douglas, G. Haase, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss, *Portable Memory Hierarchy Techniques For PDE Solvers: Part I*, Siam News, 33 (2000).

[30] C. C. Douglas, G. Haase, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss, *Portable memory hierarchy techniques for pde solvers, part ii*, SIAM News, 33 (2000), pp. 1, 10–11, 16.

[31] C. C. Douglas, G. Haase, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss, *Portable Memory Hierarchy Techniques For PDE Solvers: Part II*, Siam News, 33 (2000).

[32] C. C. Douglas, J. Hu, M. Iskandarani, M. Kowarschik, U. Rüde, and C. Weiss, *Maximizing Cache Memory Usage for Multigrid Algorithms*, in Numerical Treatment of Multiphase Flows in Porous Media. Proc. of the Int. Workshop Held at Beijing, China, 2-6 August, 1999, Z. Chen, R. Ewing, and Z.-C. Shi, eds., Lecture Notes in Physics, Springer, Aug. 2000.

[33] C. C. Douglas, J. Hu, W. Karl, M. Kowarschik, U. Rüde, and C. Weiss, *Fixed and adaptive cache aware algorithms for multigrid methods*, in Multigrid Methods VI, vol. 14 of Lecture Notes in Computational Science and Engineering, Berlin, 2000, Springer–Verlag, pp. 87–93.

[34] C. C. Douglas, J. Hu, W. Karl, M. Kowarschik, U. Rüde, and C. Weiss, *Fixed and Adaptive Cache Aware Algorithms for Multigrid Methods*, in Multigrid Methods VI. Proc. of the Sixth European Multigrid Conference held in Gent, Belgium, September 27-30, 1999, E. Dick, K. Riemslagh, and J. Vierendeels, eds., vol. 14 of Lecture Notes in Computational Science and Engineering, Springer, July 2000.

[35] C. C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss, *Cache optimization for structured and unstructured grid multigrid*, Elect. Trans. Numer. Anal., 10 (2000), pp. 21–40.

[36] C. C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss, *Cache Optimization for Structured and Unstructured Grid Multigrid*, Electronic Transactions on Numerical Analysis, 10 (2000), pp. 21–40.

[37] C. C. Douglas, J. Hu, J. Ray, D. T. Thorne, and R. S. Tuminaro, *Fast, adaptively refined computational elements in 3D*, in Computational Science – ICCS 2002, vol. 3, Springer–Verlag, Berlin, 2002, pp. 774–783.

[38] C. C. Douglas, U. Rüde, J. Hu, and M. Bittencourt, *A Guide to Designing Cache Aware Multigrid Algorithms*, in Concepts of Numerical Software, W. Hackbusch and G. Wittum, eds., Notes on Numerical Fluid Mechanics, Vieweg-Verlag, 2000. to appear.

[39] R. E. Ewing, S. F. McCormick, and J. W. Thomas, *The fast adaptive composite grid method for solving differential boundary value problems*, in Proc. Fifth ASCE–EMD Speciality Conference, 1984, pp. 1453–1456.

[40] M. Frigo and S. G. Johnson, *Fftw (fastest fourier transform in the west).* In URL http://www.fftw.org.

[41] S. R. Fulton, *An adaptive multigrid model for hurricane track prediction*, in Sixth Copper Mountain Conference on Multigrid Methods, N. D. Melson, T. A. Manteuffel, and S. F. McCormick, eds., vol. CP 3224, Hampton, VA, 1993, NASA, pp. 207–214.

[42] ——, *A comparison of multilevel adaptive methods for hurricane track prediction*, Elect. Trans. Numer. Anal., 6 (1997), pp. 120–132.

[43] ——, *An adaptive multigrid barotropic tropical cyclone track model*, Mon. Wea. Rev., 129 (2001), pp. 138–151.

[44] J. Handy, *The Cache Memory Book*, Academic Press, New York, 1998.

[45] L. Hart and S. F. McCormick, *Asynchronous multilevel adaptive methods for solving partial differential equations on multiprocessors: basic ideas*, Parallel Comput., 12 (1989), pp. 131–144.

[46] L. Hart, S. F. McCormick, A. O'Gallagher, and J. W. Thomas, *The fast adaptive composite grid method (FAC): Algorithms for advanced computers*, Appl. Math. Comput., 19 (1986), pp. 103–126.

[47] P. W. Hemker, *On the structure of an adaptive multi–level algorithm*, BIT, 20 (1980), pp. 289–301.

[48] P. W. Hemker and J. Molenaar, *An adaptive multigrid approach for the solution of the 2D semiconductor equations*, in Multigrid Methods III, W. Hackbusch and U. Trottenberg, eds., vol. 98 of International Series of Numerical Mathematics, Basel, 1991, Birkhäuser Verlag, pp. 41–60.

[49] M. A. Heroux, S. F. McCormick, S. McKay, and J. W. Thomas, *Applications of the fast adaptive composite grid method*, in Multigrid Methods: Theory, Applications, and Supercomputing, S. F. McCormick, ed., vol. 110 of Lecture Notes in Pure and Applied Mathematics, Marcel Dekker, New York, 1988, pp. 251–265.

[50] J. Hu, *Cache Based Multigrid on Unstructured Grids in Two and Three Dimensions*, PhD thesis, University of Kentucky, Department of Mathematics, Lexington, KY, 2000.

[51] IBM, *Essl (engineering and scientific subroutine library)*. http://www-1.ibm.com/servers/eserver/pseries/library/sp_books/essl.html.

[52] M. Jung, *On adaptive grids in multilevel methods*, in GAMM–Seminar on Multigrid–Methods, Gosen, Germany, September 21-25, 1992, S. Hengst, ed., Berlin, 1993, IAAS, pp. 67–80. Report No. 5.

[53] W. Karl, M. Kowarschik, U. Rüde, and C. Weiss, *DiMEPACK — A Cache–Aware Multigrid Library: User Manual*, Tech. Rep. 01–1, Lehrstuhl für Informatik 10 (Systemsimulation), University of Erlangen–Nuremberg, Germany, 2001.

[54] K. Khadra, P. Angot, and J.-P. Caltagirone, *Comparison of locally adaptive multigrid methods: L.D.C. F.A.C. and F.I.C.*, in Sixth Copper Mountain Conference on Multigrid Methods, N. D. Melson, T. A. Manteuffel, and S. F. McCormick, eds., vol. CP 3224, Hampton, VA, 1993, NASA, pp. 275–292.

[55] M. Khalil and P. Wesseling, *Vertex-centered and cell-centered multigrid for interface problems*, J. Comp. Phys., 32 (1992), pp. 1–10.

[56] P. Knupp and S. Steinberg, *Fundamentals of GRID GENERATION*, CRC Press, 1993.

[57] P. M. Knupp and S. Steinburg, *Fundamentals of Grid Generation*, CRC Press, 1993.

[58] M. Kowarschik, U. Rüde, N. Thürey, and C. Weiss, *Performance Optimization of 3D Multigrid on Hierarchical Memory Architectures*, in Proc. of the 6th Int. Conference on Applied Parallel Computing (PARA 2002), vol. 2367 of Lecture Notes in Computer Science, Espoo, Finland, June 2002, Springer, pp. 307–316.

[59] M. Kowarschik, U. Rüde, C. Weiss, and W. Karl, *Cache-Aware Multigrid Methods for Solving Poisson's Equation in Two Dimensions*, Computing, 64 (2000), pp. 381–399.

[60] M. Kowarschik and C. Weiss, *DiMEPACK — A Cache–Optimized Multigrid Library*, in Proc. of the Int. Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001), H. Arabnia, ed., vol. I, Las Vegas, NV, USA, 2001, CSREA, CSREA Press.

[61] ——, *An Overview of Cache Optimization Techniques and Cache–Aware Numerical Algorithms*, in Algorithms for Memory Hierarchies — Advanced Lectures, U. Meyer, P. Sanders, and J. Sibeyn, eds., vol. 2625 of Lecture Notes in Computer Science, Springer, Mar. 2003, pp. 213–232.

[62] M. Kowarschik, C. Weiss, and U. Rüde, *Data Layout Optimizations for Variable Coefficient Multigrid*, in Proc. of the 2002 Int. Conference on Computational Science (ICCS2002), Part III, vol. 2331 of Lecture Notes in Computer Science, Amsterdam, The Netherlands, Apr. 2002, Springer, pp. 642–651.

[63] M. Lemke and D. Quinlan, *Fast adaptive composite grid methods on distributed parallel architectures*, in Preliminary Proceedings of the Fifth Copper Mountain Conference on Multigrid Methods, T. A. Manteuffel and S. F. McCormick, eds., vol. 2, Denver, 1991, University of Colorado, pp. 61–75.

[64] ——, *Local refinement based fast adaptive composite grid methods onSUPRENUM, multigrid methods: special topics and applications II*, in GMD Studien Nr. 189, W. Hackbusch and U. Trottenberg, eds., GMD, Sankt Augustin, 1991, pp. 179–189.

[65] ——, *Fast adaptive composite grid methods on distributed parallel architectures*, Comm. Appl. Num. Methods, 8 (1992), pp. 609–619.

[66] C. Liu, Z. Liu, and S. F. McCormick, *Multilevel adaptive methods for incompressible flow in grooved channels*, J. Comput. Appl. Math., 38 (1991), pp. 283–295.

[67] ——, *Multilevel adaptive methods for incompressible flow in grooved channels*, in Preliminary Proceedings of the Fifth Copper Mountain Conference on Multigrid Methods, T. A. Manteuffel and S. F. McCormick, eds., vol. 2, Denver, 1991, University of Colorado, pp. 103–120.

[68] ——, *Multilevel adaptive methods for laminar diffusion flames*, J. Sci. Comput., 8 (1993), pp. 341–355.

[69] C. Liu and S. F. McCormick, *Multigrid, elliptic grid generation and the fast adaptive composite grid method for solving transonic potential flow equations*, in Multigrid Methods: Theory, Applications, and Supercomputing, S. F. McCormick, ed., vol. 110 of Lecture Notes in Pure and Applied Mathematics, Marcel Dekker, New York, 1988, pp. 365–387.

[70] D. Martin and K. Cartwright, *Solving Poisson's equation using adaptive mesh refinement.* http://seesar.lbl.gov/anag/staff/martin/tar/AMR.ps, 1996.

[71] S. F. McCormick, *Fast adaptive composite grid (FAC) methods: Theory for the variational case*, in Defect Correction Methods: Theory and Applications, K. Böhmer and H. J. Stetter, eds., Computing Suppl. 5, Springer–Verlag, Vienna, 1984, pp. 115–121.

[72] ——, *A variational theory for multi–level adaptive techniques (MLAT)*, in Multigrid Methods for Integral and Differential Equations, D. Paddon and H. Holstein, eds., The Institute for Integral and Differential Equations, Clarendon Press, Oxford, 1985, pp. 225–230.

[73] ——, *The fast adaptive composite (FAC) method for elliptic equations*, Math. Comp., 46 (1986), pp. 439–456.

[74] S. F. McCormick, S. M. McKay, and J. W. Thomas, *Computational complexity of the fast adaptive composite grid (fac) method*, Appl. Numer. Math., 6 (1990).

[75] S. F. McCormick and D. Quinlan, *Asynchronous multilevel adaptive methods for solving partial differential equations on multiprocessors: performance results*, Parallel Comput., 12 (1989), pp. 145–156.

[76] S. F. McCormick and U. Rüde, *A finite volume convergence theory for the fast adaptive composite grid method*, Appl. Numer. Math., 14 (1994), pp. 91–103.

[77] S. F. McCormick and J. W. Thomas, *The fast adaptive composite grid (FAC) method for elliptic equations*, Math. Comp., 46 (1986), pp. 439–456.

[78] W. F. Mitchell, *Optimal multilevel iterative methods for adaptive grids*, SIAM J. Sci. Stat. Comput., 13 (1992), pp. 146–167.

[79] K. W. Morton and D. F. Mayers, *Numerical Solution of Partial Differential Equations*, Cambridge University Press, 1994.

[80] S. Owen, *Meshing research corner.* In URL http://www.andrew.cmu.edu/user/sowen/mesh.html, 2003.

[81] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantative Approach*, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1996.

[82] H. Pfänder, *Cache–optimierte Mehrgitterverfahren mit variablen Koeffizienten auf strukturierten Gittern*, Master's thesis, Department of Computer Science, University of Erlangen–Nuremberg, Germany, 2000.

[83] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li, *Thread scheduling for cache locality*, in Proceedings of the Seventh ACM Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, 1996, ACM, pp. 60–73.

[84] J. Ramanujam and P. Sadayappan, *Nested loop tiling for distributed memory machines*, in dmcc90, Charleston, S.C., Apr 1990, pp. 1088–1096.

[85] J. R. Rosendale, *Algorithms and data structures for adaptive multigrid elliptic solvers*, Appl. Math. Comput., 13 (1983), pp. 453–470.

[86] U. Rüde, *Fully adaptive multigrid methods*, SIAM J. Numer. Anal., 30 (1993), pp. 230–248.

[87] ——, *Mathematical and Computational Techniques for Multilevel Adaptive Methods*, vol. 13 of Frontiers in Applied Mathematics, SIAM, Philadelphia, 1993.

[88] ——, *On the robustness and efficiency of the fully adaptive multigrid method*, in Domain Decomposition Methods in Science and Engineering: The Sixth International Conference on Domain Decomposition, vol. 157 of Contemporary Mathematics, Providence, Rhode Island, 1994, American Mathematical Society, pp. 121–126.

[89] ——, *On the V–cycle of the fully adaptive multigrid method*, in Adaptive Methods – Algorithms, Theory and Applications, vol. 46 of Notes on Numerical Fluid Mechanics, Braunschweig, 1994, Vieweg, pp. 251–260.

[90] U. Rüde, *Iterative Algorithms on High Performance Architectures*, in Proc. of the EuroPar97 Conference, Lecture Notes in Computer Science, Springer, Aug. 1997, pp. 26–29.

[91] ——, *Technological Trends and their Impact on the Future of Supercomputing*, in High Performance Scientific and Engineering Computing, Proc. of the Int. FORTWIHR Conference on HPSEC, H.-J. Bungartz, F. Durst, and C. Zenger, eds., vol. 8 of Lecture Notes in Computer Science and Engineering, Springer, Mar. 1998, pp. 459–471.

[92] W. Skamarock, J. Oliger, and R. L. Street, *Adaptive grid refinement for numerical weather prediction*, J. Comput. Phys., 80 (1989), p. 27.

[93] L. Stals and U. Rüde, *Techniques for Improving The Data Locality of Iterative Methods*, Tech. Rep. MRR97–038, School of Mathematical Science, Australian National University, Oct. 1997.

[94] L. Stals, U. Rüde, C. Weiss, and H. Hellwagner, *Data Local Iterative Methods for the Efficient Solution of Partial Differential Equations*, in Proc. of the Eighth Biennial Conference Computational Techniques and Applications: CTAC97, J. Noye, M. Teubner, and A. Gill, eds., Adelaide, Australia, Sept. 1997, pp. 655–662.

[95] J. W. Thomas, R. Schweitzer, M. A. Heroux, S. F. McCormick, and A. M. Thomas, *Application of the fast adaptive composite grid method to computation fluid dynamics*, in Numerical Methods in Laminar and Turbulent Flow, C. Taylor, W. G. Habashi, and M. M. Hafez, eds., Pineridge Press, Swansea, U.K., 1987, pp. 1071–1082.

[96] J. Thompson, B. Soni, and N. Weatherill, *Handbook of Grid Generation*, CRC Press, 1999.

[97] N. Thürey, *Cache Optimizations for Multigrid in 3D.* Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, University of Erlangen-Nuremberg, Germany, June 2002.

[98] C. Weiss, *Data Locality Optimizations for Multigrid Methods on Structured Grids*, PhD thesis, Lehrstuhl für Rechnertechnik und Rechnerorganisation, Institut für Informatik, Technische Universität München, Munich, Germany, Dec. 2001.

[99] C. Weiss, H. Hellwagner, L. Stals, and U. Rüde, *Data Locality Optimizations to Improve The Efficiency of Multigrid Methods*, Tech. Rep. 02–1, Lehrstuhl für Informatik 10 (Systemsimulation), University of Erlangen–Nuremberg, Germany, Jan. 2002.

[100] C. Weiss, W. Karl, M. Kowarschik, and U. Rüde, *Memory Characteristics of Iterative Methods*, in Proc. of the ACM/IEEE SC99 Conference, Portland, Oregon, Nov. 1999.

[101] C. Weiss et al, *Dimepack*. http://wwwbode.cs.tum.edu/Par/arch/cache.

[102] J. Wilke, *Cache Optimizations for the Lattice Boltzmann Method in 2D*. Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, University of Erlangen-Nuremberg, Germany, Feb. 2003.

[103] J. Wilke, T. Pohl, M. Kowarschik, and U. Rüde, *Cache Performance Optimizations for Parallel Lattice Boltzmann Codes*, in Proc. of the EuroPar03 Conference, Lecture Notes in Computer Science, Springer. to appear.

[104] M. Wolfe, *More iteration space tiling*, in super89, Reno, Nev., nov 1989, pp. 655–664.

[105] M. Wolfe, *High Performance Compilers for Parallel Computing*, Addison Wesley, 1996.

Appendix A

.1 Introduction

In the following sections, the smoother is derived from the discretizations for the interior, edge, and corner points, and the corresponding damping factors are shown. The edge and corner damping factors differ a little bit from [70].

The equation being discretized is $-\Delta u = f$, and $\mathcal{L}$ represents the discrete operator.

.2 Interior Damping Factor

In the interior,

$$\frac{1}{h^2}(4u_{ij} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1}) = f_{ij},$$

so a Gauss-Seidel update looks like this:

$$\begin{aligned} u_{ij} &\leftarrow \frac{1}{4}\left(h^2 f_{ij} + u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1}\right) \\ &= u_{ij} + \frac{1}{4}\left(h^2 f_{ij} + u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{ij}\right) \\ &= u_{ij} + \frac{1}{4}h^2\left(f_{ij} + \frac{1}{h^2}\left(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{ij}\right)\right) \\ &= u_{ij} + \lambda_{interior}\left(f_{ij} - \mathcal{L}u\right), \end{aligned}$$

where the damping factor $\lambda_{interior} = \frac{1}{4}h^2$.

.3 Edge Damping Factor

At an edge,

$$\frac{1}{h^2}\left(6u_{ij} - \frac{4}{3}u_{i+1,j} - \frac{8}{3}u_{i-\frac{1}{2},j} - u_{i,j+1} + u_{i,j-1}\right) = f_{ij},$$

so a Gauss-Seidel update looks like this:

$$\begin{aligned} u_{ij} &\leftarrow \frac{1}{6}\left(h^2 f_{ij} + \frac{4}{3}u_{i+1,j} + \frac{8}{3}u_{i-\frac{1}{2},j} + u_{i,j+1} + u_{i,j-1}\right) \\ &= u_{ij} + \frac{1}{6}\left(h^2 f_{ij} + \frac{4}{3}u_{i+1,j} + \frac{8}{3}u_{i-\frac{1}{2},j} + u_{i,j+1} + u_{i,j-1} - 6u_{ij}\right) \\ &= u_{ij} + \frac{1}{6}h^2\left(f_{ij} + \frac{1}{h^2}\left(\frac{4}{3}u_{i+1,j} + \frac{8}{3}u_{i-\frac{1}{2},j} + u_{i,j+1} + u_{i,j-1} - 6u_{ij}\right)\right) \\ &= u_{ij} + \lambda_{edge}(f_{ij} - \mathcal{L}u), \end{aligned}$$

where the damping factor $\lambda_{edge} = \frac{1}{6}h^2$.

.4 Corner Damping Factor

At a corner,

$$\frac{1}{h^2}\left(8u_{ij}-\frac{4}{3}u_{i+1,j}-\frac{8}{3}u_{i-\frac{1}{2},j}-\frac{8}{3}u_{i,j+\frac{1}{2}}-\frac{4}{3}u_{i,j-1}\right)=f_{ij},$$

so a Gauss-Seidel update looks like this:

$$\begin{aligned}
u_{ij} &\leftarrow \frac{1}{8}\left(h^2 f_{ij}+\frac{4}{3}u_{i+1,j}+\frac{8}{3}u_{i-\frac{1}{2},j}+\frac{8}{3}u_{i,j+\frac{1}{2}}+\frac{4}{3}u_{i,j-1}\right)\\
&= u_{ij}+\frac{1}{8}\left(h^2 f_{ij}+\frac{4}{3}u_{i+1,j}+\frac{8}{3}u_{i-\frac{1}{2},j}+\frac{8}{3}u_{i,j+\frac{1}{2}}+\frac{4}{3}u_{i,j-1}-8u_{ij}\right)\\
&= u_{ij}+\frac{1}{8}h^2\left(f_{ij}+\frac{1}{h^2}\left(+\frac{4}{3}u_{i+1,j}+\frac{8}{3}u_{i-\frac{1}{2},j}+\frac{8}{3}u_{i,j+\frac{1}{2}}+\frac{4}{3}u_{i,j-1}-8u_{ij}\right)\right)\\
&= u_{ij}+\lambda_{corner}(f_{ij}-\mathcal{L}u),
\end{aligned}$$

where the damping factor $\lambda_{corner}=\frac{1}{8}h^2$.

.5 Comments

So the damping factors are

$$\lambda_{interior}=\frac{1}{4}h^2,$$

$$\lambda_{edge}=\frac{1}{6}h^2,$$

and

$$\lambda_{corner}=\frac{1}{8}h^2.$$

The interior damping factor here matches the interior damping factor stated in [70]. The boundary damping factors do not, although the boundary damping factor

$$\lambda_{boundary}=\frac{3}{4}\lambda_{interior}=\frac{3}{16}h^2=0.1875h^2$$

stated in [70] is close to the edge damping factor $\lambda_{edge}\approx 0.1667h^2$ computed here.

Appendix B

The code is written in C++. There is a hierarchy of C++ classes corresponding to the AMR hierarchy

Grids	Grid Functions
	GridFunctionClass
GridClass	GridFunctionArrayClass
GridLevelClass	GridFunctionLevelClass
GridHierarchyClass	GridFunctionCompositeClass

and there is a interface class, AMRMGClass.

The relationships between the classes are outlined in Fig. 1. This chapter describes

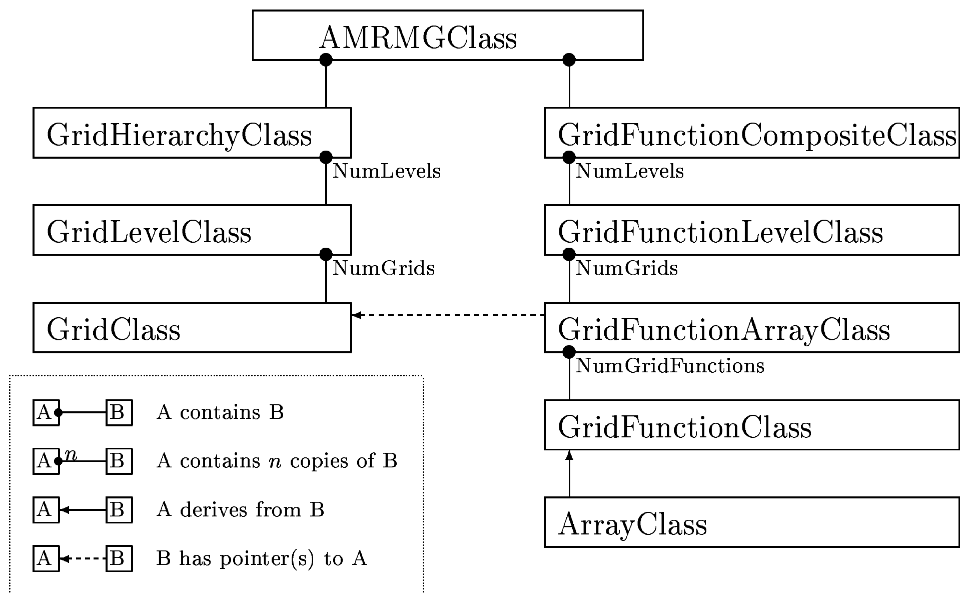


Figure 1: C++ Class Hierarchy

the grids and grid functions and the methods that are defined for each. Note the parallel between grid classes and grid function classes. Grid classes store specifications about the hierarchy and grid functions store the corresponding function values. The methods for the discrete operators, iterative solvers, residual computations, and composite multilevel algorithm are defined in the grid functions. The interface class, AMRMGClass, stores the grid hierarchy and composite grid function and defines methods for initializing a problem and coordinating the solution procedure.

.1 Grids

Information about the AMR hierarchy is managed by grid classes.

.1.1 Grid Class

A grid is, first of all, a bounding box. The grid class contains the beginning and ending coordinates of a rectangular grid stored as the integer coordinates of the beginning and ending grid points from the discretization. We also store the real coordinates of the corresponding region from the domain. In addition, the grid class includes the mesh spacing in each direction and the number of grid points in each direction. The local indexing on a grid is always $(0, 0, 0)..(ni - 1, nj - 1, nk - 1)$, where ni, nj and nk are the number of grid points in the x, y and z directions. Global indexing is computed based on the real coordinates and mesh spacings.

The Uniform Modeling Language (UML) diagram for this class is

```
+-----------------------------------------------------------------+
  GridClass
+-----------------------------------------------------------------+
  - si,  ei,  ni: GridCoordsClass
  - GridLevelPtr: GridLevelClass*
  - LevelIndex: int
  - NumLevels: int
  - RefinementFactor: int
  - ChildPtrArray: ArrayClass<GridClass*>
  - ParentPtr: GridClass*
  - GridFunctionArrayPtr: GridFunctionArrayClass*
  - Index: int
  - ShadowGrid: ArrayClass<int>
  - TooBigForCache: int
  - clinesx, clinesy: int
  - OnlyPostSmoothing: int
  - NumSmootherStepsPerLevel: int
  - NumSmootherSteps: int
+-----------------------------------------------------------------+
  + GridClass()
  + GridClass(
      inout arg_name: char*)
  + GridClass(
      inout arg_name: char*
      in arg_sx: const RealCoordsClass&
      in arg_ex: const RealCoordsClass&
      in arg_si: const GridCoordsClass&
      in arg_ei: const GridCoordsClass&
      in arg_LevelIndex: const int
      in arg_NumLevels: const int )
  + GridClass(
      inout arg_name: char*
      in arg_sx: const RealCoordsClass&
```

```
    in arg_ex: const RealCoordsClass&
    in arg_si: const GridCoordsClass&
    in arg_ei: const GridCoordsClass&
    in arg_ni: const GridCoordsClass&
    in arg_LevelIndex: const int
    in arg_NumLevels: const int )
+ ~GridClass()
+ GridClass(
    in arg_GridClass: GridClass&)
+ operator=(
    in arg_GridClass: GridClass&): GridClass&
+ operator<<(
    in o: ostream&
    in arg_GridClass: GridClass&): friend ostream&
+ AddChildPtr(
    inout arg_GridPtr: GridClass*): void
+ isLeaf(): int
+ HasChildren(): int
+ InitShadowGrid(): void
+ get_sx(): RealCoordsClass&
+ get_ex(): RealCoordsClass&
+ get_dx(): RealCoordsClass&
+ get_sx(
    in i: int ): real
+ get_ex(
    in i: int ): real
+ get_dx(
    in i: int ): real
+ get_si(): GridCoordsClass&
+ get_ei(): GridCoordsClass&
+ get_ni(): GridCoordsClass&
+ get_si(
    in i: int ): int
+ get_ei(
    in i: int ): int
+ get_ni(
    in i: int ): int
+ get_n(): int
+ set_GridLevelPtr(
    inout arg_GridLevelPtr: GridLevelClass*): void
+ get_GridLevelPtr(): GridLevelClass*
+ get_NumChildren(): int
+ get_ChildPtr(
    in i: int ): GridClass*
```

```
  + get_ChildPtrArray(): ArrayClass<GridClass*>&
  + set_ParentPtr(
     inout arg_ParentPtr: GridClass*): void
  + get_ParentPtr(): GridClass*
  + set_GridFunctionArrayPtr(
     inout arg_GridFunctionArrayPtr: GridFunctionArrayClass*): void
  + get_GridFunctionArrayPtr(): GridFunctionArrayClass*
  + set_LevelIndex(
     in arg_LevelIndex: int ): void
  + get_LevelIndex(): int
  + set_NumLevels(
     in arg_NumLevels: int ): void
  + get_NumLevels(): int
  + set_RefinementFactor(
     in arg_RefinementFactor: int ): void
  + get_RefinementFactor(): int
  + set_Index(
     in arg_Index: int ): void
  + get_Index(): int
  + get_ShadowGrid(): ArrayClass<int>&
  + get_TooBigForCache(): int
  + get_clinesx(): int
  + get_clinesy(): int
  + get_OnlyPostSmoothing(): int
  + get_NumSmootherStepsPerLevel(): int
+-----------------------------------------------------------------+
```

.1.2 Grid Level Class

Multiple grid classes are combined to form a grid level class. A grid level is characterized by the mesh spacing of the grids on that level. The mesh spacings between adjacent levels differ by a factor, usually two or four, called the refinement factor.

The Uniform Modeling Language (UML) diagram for this class is

```
+-----------------------------------------------------------------+
  GridLevelClass
+-----------------------------------------------------------------+
  - n: int
  - GridPtrArray: ArrayClass<GridClass*>
  - LevelIndex: int
  - NumLevels: int
  - RefinementFactor: int
  - GridHierarchyPtr: GridHierarchyClass*
+-----------------------------------------------------------------+
```

```
+ GridLevelClass()
+ GridLevelClass(
    inout arg_name: char*)
+ GridLevelClass(
    in arg_NumGrids: int )
+ GridLevelClass(
    inout arg_name: char*
    in arg_NumGrids: int )
+ GridLevelClass(
    inout arg_GridPtr: GridClass*)
+ GridLevelClass(
    inout arg_name: char*
    inout arg_GridPtr: GridClass*)
+ GridLevelClass(
    inout  arg_GridPtrs: ArrayClass<GridClass*>&)
+ GridLevelClass(
    inout arg_name: char*
    inout  arg_GridPtrs: ArrayClass<GridClass*>&)
+ GridLevelClass(
    in arg_GridLevel: GridLevelClass&)
+ operator=(
    in arg_GridLevel: GridLevelClass&): GridLevelClass&
+ operator()(
    in i: int): GridClass*
+ operator()(
    in i: int) const: const GridClass*
+ ~GridLevelClass()
+ operator<<(
    in o: ostream&
    in arg_GridLevel: GridLevelClass&): friend ostream&
+ InitBaseGrid(
    inout arg_BaseGridPtr: GridClass*): void
+ AddGrid(
    inout arg_GridPtr: GridClass*
    inout arg_ParentGridPtr: GridClass*): void
+ InitShadowGrid(): void
+ get_NumGrids(): int
+ get_n(): int
+ get_GridLevel(): ArrayClass<GridClass*>&
+ set_LevelIndex(
    in arg_LevelIndex: int ): void
+ get_LevelIndex(): int
+ set_NumLevels(
    in arg_NumLevels: int ): void
```

```
  + get_NumLevels(): int
  + set_RefinementFactor(
      in arg_RefinementFactor: int ): void
  + get_RefinementFactor(): int
  + set_GridHierarchyPtr(
      inout arg_GridPtrHierarchyPtr: GridHierarchyClass*): void
  + get_GridHierarchyPtr(): GridHierarchyClass*
+----------------------------------------------------------------+
```

.1.3 Grid Hierarchy Class

Multiple grid level classes are combined to form a grid hierarchy class. Finer grids can be thought of as either nested within or hovering above a coarser grid. See Fig. 3.7 for an illustration of the former and Fig. 2.4 for an illustration of the latter. In either case, the finer grids can be called *child grids* of a coarse grid (or *parent grid*). Grids maintain pointers to their children and parent. A grid has at most one parent.

The Uniform Modeling Language (UML) diagram for this class is

```
+----------------------------------------------------------------+
  GridHierarchyClass
+----------------------------------------------------------------+
  - n: int
  - RefinementFactor, RefinementFactor2: int
  - GridLevelPtrArray: ArrayClass<GridLevelClass*>
+----------------------------------------------------------------+
  + GridHierarchyClass()
  + GridHierarchyClass(
      inout arg_name: char*)
  + GridHierarchyClass(
      in arg_NumLevels: int )
  + GridHierarchyClass(
      inout arg_name: char*
      in arg_NumLevels: int )
  + GridHierarchyClass(
      inout arg_GridLevelPtr: GridLevelClass*)
  + GridHierarchyClass(
      inout arg_name: char*
      inout arg_GridLevelPtr: GridLevelClass*)
  + GridHierarchyClass(
      inout  arg_GridHierarchy: ArrayClass<GridLevelClass*>&)
  + GridHierarchyClass(
      inout arg_name: char*
      in:
      inout  arg_GridHierarchy: ArrayClass<GridLevelClass*>&)
```

```
  + ~GridHierarchyClass()
  + GridHierarchyClass(
      in arg_GridHierarchy: GridHierarchyClass&)
  + operator=(
      in arg_GridHierarchy: GridHierarchyClass&): GridHierarchyClass&
  + operator<<(
      in o: ostream&
      in:
      in arg_GridHierarchyClass: GridHierarchyClass&): friend ostream&
  + InitBaseGrid(
      inout arg_BaseGrid: GridClass*): void
  + AddLevel(): void
  + AddGrid(
      in arg_LevelIndex: int
      in:
      inout arg_GridPtr: GridClass*
      in:
      inout arg_ParentGridPtr: GridClass*): void
  + InitShadowGrid(): void
  + get_NumLevels(): int
  + set_NumLevels(
      in arg_NumLevels: int ): void
  + get_n(): int
  + set_n(
      in arg_n: int ): void
  + get_RefinementFactor(): int
  + get_RefinementFactor2(): int
  + set_RefinementFactor(
      in arg_RefinementFactor: int ): void
  + get_GridHierarchy(): ArrayClass<GridLevelClass*>&
  + get_GridLevel(
      in i: int ): GridLevelClass&
  + get_GridLevelPtr(
      in i: int ): GridLevelClass*
+-----------------------------------------------------------------+
```

.2 Grid Functions

Grid functions and operations on grid functions are managed by grid function classes.

.2.1 Grid Function Class

Grid functions store the values associated with each grid point on a grid. A grid function is initialized with a pointer to the corresponding grid. The grid function class is derived from an array class allowing easy access to the data either by an overloaded Fortran-style indexing operator or by a conventional C-style pointer.

The Uniform Modeling Language (UML) diagram for this class is

```
+----------------------------------------------------------------+
  GridFunctionClass
+----------------------------------------------------------------+
  - GridPtr: GridClass*
  - BaseGridPtr: GridClass*
  - GridFunctionIndex: int
+----------------------------------------------------------------+
  + GridFunctionClass(
      in arg_GridFunctionIndex = 0: int )
  + GridFunctionClass(
      inout arg_name: char*
      in arg_GridFunctionIndex = 0: int )
  + GridFunctionClass(
      inout arg_GridPtr: GridClass*
      inout arg_BaseGridPtr: GridClass*
      in arg_GridFunctionIndex = 0: int )
  + GridFunctionClass(
      inout arg_name: char*
      inout arg_GridPtr: GridClass*
      inout arg_BaseGridPtr: GridClass*
      in arg_GridFunctionIndex = 0: int )
  + GridFunctionClass(
      in arg_GridFunction: GridFunctionClass&)
  + operator*(
      in arg_GridFunction: GridFunctionClass& ): real
  + operator*(
      in arg: real ): GridFunctionClass
  + operator*=(
      in arg: real ): GridFunctionClass&
  + operator*(
      in arg: real
      in arg_GridFunction: GridFunctionClass& ): friend const GridFunctionClass
  + operator=(
      in arg_GridFunction: GridFunctionClass&): GridFunctionClass&
  + operator=(
      in ArrayClass<real> arg_Array: const&): GridFunctionClass&
  + operator<<(
```

```
      in o: ostream&
      in arg_GridFunctionClass: GridFunctionClass&): friend ostream&
  + ~GridFunctionClass()
  + Init(
      inout arg_GridPtr: GridClass*
      inout arg_BaseGridPtr: GridClass*
      in arg_myProcID: int ): void
  + SubtractPComplement(): void
  + SubtractP(): void
  + ComputeChildSumOfAbsoluteValues(): real
  + get_myProcID(): int
  + set_myProcID(
      in arg_myProcID: int ): void
  + get_GridPtr(): GridClass*
  + set_GridPtr(
      inout arg_GridPtr: GridClass*): void
  + get_BaseGridPtr(): GridClass*
  + set_BaseGridPtr(
      inout arg_BaseGridPtr: GridClass*): void
  + get_GridFunctionIndex(): int
  + set_GridFunctionIndex(
      in arg_GridFunctionIndex: int ): void
+-----------------------------------------------------------------+
```

.2.2 Grid Function Array Class

Each grid generally has many functions associated with it, e.g., the solution, right hand side, residual, and correction. The grid function array class stores an array of those grid functions. This is where most of the work is done. Most of the tools for implementing multigrid on the AMR hierarchy are defined here. However, operations are coordinated from the grid function level class and the grid function composite class.

The Uniform Modeling Language (UML) diagram for this class is

```
+-----------------------------------------------------------------+
  GridFunctionArrayClass
+-----------------------------------------------------------------+
  - BaseGridPtr: GridClass*
  - NumGridFunctions: int
  - GridFunctionPtrArray: ArrayClass<GridFunctionClass*>
  - CoefficientMatrixPtr: CoefficientMatrixClass*
  - gpn, gps, gpe, gpw: ArrayClass<real>
  - gpn_flag, gps_flag, gpe_flag, gpw_flag: ArrayClass<real>
  - gpsw, gpse, gpnw, gpne: real
  - gpsw_flag, gpse_flag, gpnw_flag, gpne_flag: real
```

```
  - OnlyPostSmoothing: int
  - NumSmootherStepsPerLevel: int
  - NumSmootherSteps: int
  - UseStandardSmoother: int
  - NumCacheLines: int
  - NumGridPointsInCache: int
  - UseCacheAwareSmoother: int
  - UseCacheAwareSmootherXY: int
  - ComputeResidualInSmoother: int
  - ParentPtr: GridFunctionArrayClass*
  - ChildPtrArray: ArrayClass<GridFunctionArrayClass*>
  - phisavePtr: GridFunctionClass*
  - LePtr: GridFunctionClass*
  - eTempPtr: GridFunctionClass*
  - interpd_red_east: ArrayClass<real>
  - interpd_red_west: ArrayClass<real>
  - interpd_red_north: ArrayClass<real>
  - interpd_red_south: ArrayClass<real>
+----------------------------------------------------------------+
  + GridFunctionArrayClass(
      in arg_myProcID: int
      in arg_NumGridFunctions: int )
  + GridFunctionArrayClass(
      inout arg_name: char*
      in arg_myProcID: int
      in arg_NumGridFunctions: int )
  + GridFunctionArrayClass(
      inout arg_GridPtr: GridClass*
      inout arg_BaseGridPtr: GridClass*
      in arg_myProcID: int
      in arg_NumGridFunctions: int )
  + GridFunctionArrayClass(
      inout arg_name: char*
      inout arg_GridPtr: GridClass*
      inout arg_BaseGridPtr: GridClass*
      in arg_myProcID: int
      in arg_NumGridFunctions: int )
  + GridFunctionArrayClass(
      in arg_GridFunctionArrayClass: GridFunctionArrayClass&)
  + operator=(
      in arg_GridFunctionArrayClass: GridFunctionArrayClass&):&
    GridFunctionArrayClass
  + operator()( in i: int ): GridFunctionClass*
  + operator()( in i: int ) const: const GridFunctionClass*
```

```
 + operator<<(
     in o: ostream&
     in arg_GridFunctionArrayClass: GridFunctionArrayClass&): friend ostream&
 + ~GridFunctionArrayClass()
 + Init(
     inout arg_GridPtr: GridClass*
     inout arg_BaseGridPtr: GridClass*
     in arg_myProcID: int
     in arg_NumGridFunctions: int ): void
 + AddChildPtr(
     inout arg_GridFunctionArrayPtr: GridFunctionArrayClass*): void
 + isLeaf(): int
 + HasChildren(): int
 + SolutionFunction(
     in x: real
     in y: real
     in z: real ): real
 + CoefficientFunction(
     in x: real
     in y: real
     in z: real ): real
 + InitPoisson(): void
 + InitPoissonCoeffs(): void
 + InitPoissonRHS(): void
 + InitRandom(): void
 + Smooth(
     inout arg_solPtr: GridFunctionClass*
     inout arg_rhsPtr: GridFunctionClass*
     inout arg_resPtr: GridFunctionClass*
     in arg_NumIts: int ): void
 + SmoothStandard(
     inout arg_solPtr: GridFunctionClass*
     inout arg_rhsPtr: GridFunctionClass*
     inout arg_resPtr: GridFunctionClass*
     in arg_NumIts: int ): void
+ oid GridFunctionArrayClass: : SmoothStandard_WithResidualUpdate(
     inout arg_solPtr: GridFunctionClass*
     inout arg_rhsPtr: GridFunctionClass*
     in arg_NumIts: int )
 + SmoothCacheAware(
     inout arg_solPtr: GridFunctionClass*
     inout arg_rhsPtr: GridFunctionClass*
     inout arg_resPtr: GridFunctionClass*
     in arg_NumIts: int ): void
```

```
+ SmoothCacheAwareXY(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothNSEWCacheAware(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothNSEWCacheAwareXY(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothSEWCacheAware(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothNEWCacheAware(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothNSECacheAware(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothNSWCacheAware(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothNWCacheAware(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothNECacheAware(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothSWCacheAware(
```

```
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothSECacheAware(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothNorthCacheAware(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothSouthCacheAware(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothEastCacheAware(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothWestCacheAware(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothInteriorCacheAware(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothInteriorCacheAwareXY(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothInteriorCacheAwareXYxy(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
```

```
+ SmoothCacheAware(
    in arg_SOL: int
    in arg_RHS: int
    in arg_NumIts: int ): void
+ SmoothCacheAwareXY(
    in arg_SOL: int
    in arg_RHS: int
    in arg_NumIts: int ): void
+ SmoothStandard(
    in arg_SOL: int
    in arg_RHS: int
    in arg_NumIts: int ): void
+ Smooth(
    in arg_SOL: int
    in arg_RHS: int
    in arg_NumIts: int ): void
+ SmoothNSEW(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothSW(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothSE(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothNE(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothNW(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothN(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothS(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    in arg_NumIts: int ): void
```

```
+ SmoothE(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothW(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothInterior(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothNSEW_WithIntegratedInterpolation(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    inout arg_cvecPtr: GridFunctionClass*): void
+ SmoothNSEW_WithIntegratedInterpolationAndFullResidualComputation(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    inout arg_cvecPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*): void
+ SmoothNSEW_WithIntegratedInterpolationAndResidualUpdate(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    inout arg_cvecPtr: GridFunctionClass*): void
+ Smooth_WithFullResidualComputation(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothNSEW_WithFullResidualComputation(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothInterior_WithFullResidualComputation(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ Smooth_WithResidualUpdate(
    inout arg_solPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
```

```
+ SmoothNSEW_WithResidualUpdate(
    inout arg_solPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothInterior_WithResidualUpdate(
    inout arg_solPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothCacheAware_WithResidualUpdate(
    inout arg_solPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothNSEWCacheAware_WithFullResidualComputation(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothInteriorCacheAware_WithFullResidualComputation(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothNSEWCacheAware_WithResidualUpdate(
    inout arg_solPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothInteriorCacheAware_WithResidualUpdate(
    inout arg_solPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothCacheAware_WithResidualUpdateXY(
    inout arg_solPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothNSEWCacheAware_WithFullResidualComputationXY(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothInteriorCacheAware_WithFullResidualComputationXY(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
```

```
+ SmoothNSEWCacheAware_WithResidualUpdateXY(
    inout arg_solPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ SmoothInteriorCacheAware_WithResidualUpdateXY(
    inout arg_solPtr: GridFunctionClass*
    inout arg_resPtr: GridFunctionClass*
    in arg_NumIts: int ): void
+ ComputeResidual(
    in arg_SOL: int
    in arg_RHS: int
    in arg_lfinest: int ): void
+ ComputeResidualInterior(
    in arg_SOL: int
    in arg_RHS: int
    in arg_lfinest: int ): void
+ ComputeResidualNSEW(
    in arg_SOL: int
    in arg_RHS: int
    in arg_lfinest: int ): void
+ ComputeResidualNSEW(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    in arg_lfinest: int ): void
+ ComputeResidualSW(
    in arg_SOL: int
    in arg_RHS: int
    in arg_lfinest: int ): void
+ ComputeResidualSW(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    in arg_lfinest: int ): void
+ ComputeResidualSE(
    in arg_SOL: int
    in arg_RHS: int
    in arg_lfinest: int ): void
+ ComputeResidualSE(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    in arg_lfinest: int ): void
+ ComputeResidualNE(
    in arg_SOL: int
    in arg_RHS: int
    in arg_lfinest: int ): void
```

```
+ ComputeResidualNE(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    in arg_lfinest: int ): void
+ ComputeResidualNW(
    in arg_SOL: int
    in arg_RHS: int
    in arg_lfinest: int ): void
+ ComputeResidualNW(
    inout arg_solPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    in arg_lfinest: int ): void
+ Vcycle_Project_Stage00(): void
+ Vcycle_Project_Stage01a(): void
+ Vcycle_Project_Stage01b(): void
+ Vcycle_Project_Stage02(): void
+ Vcycle_CoarseGridSolve(
    in arg_RHS: int ): void
+ Vcycle_Correct_Stage01(): void
+ Vcycle_Correct_Stage02(): void
+ Lpatch(
    inout arg_lopPtr: GridFunctionClass*
    in arg_SOL: int ): void
+ Lpatch(
    inout arg_lopPtr: GridFunctionClass*
    inout arg_solPtr: GridFunctionClass*): void
+ LpatchInterior(
    inout arg_lopPtr: GridFunctionClass*
    inout arg_solPtr: GridFunctionClass*): void
+ LpatchNSEW(
    inout arg_lopPtr: GridFunctionClass*
    inout arg_solPtr: GridFunctionClass*): void
+ LpatchSW(
    inout arg_lopPtr: GridFunctionClass*
    inout arg_solPtr: GridFunctionClass*): void
+ LpatchSE(
    inout arg_lopPtr: GridFunctionClass*
    inout arg_solPtr: GridFunctionClass*): void
+ LpatchNE(
    inout arg_lopPtr: GridFunctionClass*
    inout arg_solPtr: GridFunctionClass*): void
+ LpatchNW(
    inout arg_lopPtr: GridFunctionClass*
    inout arg_solPtr: GridFunctionClass*): void
```

```
+ Lsuball(
    inout arg_lopPtr: GridFunctionClass*
    in arg_lfinest: int
    in arg_SOL: int ): void
+ FluxMatch(
    inout arg_lopPtr: GridFunctionClass*
    in arg_lfinest: int
    in arg_SOL: int ): void
+ FluxMatchResidual(
    inout arg_resPtr: GridFunctionClass*
    inout arg_rhsPtr: GridFunctionClass*
    in arg_lfinest: int
    in arg_SOL: int ): void
+ InterpolateGhostPoints(
    inout arg_lopPtr: GridFunctionClass*
    in arg_lfinest: int
    in arg_SOL: int ): void
+ ComputeTrueSolution(
    in arg_IND: int ): void
+ ComputeError(): void
+ quad_interp(
    in a: double
    in b: double
    in c: double ): double
+ quad_interp_blue01(
    in a: double
    in b: double
    in c: double ): double
+ quad_interp_blue02(
    in a: double
    in b: double
    in c: double ): double
+ quad_interp_red(
    in fleft: double
    in fmiddle: double
    in cright: double ): double
+ quad_interp_blue01_backward(
    in a: double
    in b: double
    in c: double ): double
+ quad_interp_blue02_backward(
    in a: double
    in b: double
    in c: double ): double
```

```
+ quad_interp_blue01_forward(
    in a: double
    in b: double
    in c: double ): double
+ quad_interp_blue02_forward(
    in a: double
    in b: double
    in c: double ): double
+ get_interpd_red_east(): ArrayClass<real>&
+ get_interpd_red_west(): ArrayClass<real>&
+ get_interpd_red_north(): ArrayClass<real>&
+ get_interpd_red_south(): ArrayClass<real>&
+ Interpolate(
    in arg_FVEC: int
    in arg_CVEC: int ): void
+ InterpInterior(
    in fvec: GridFunctionClass&
    in cvec: GridFunctionClass&
    inout ChildGridPtr: GridClass*): void
+ InterpNSEW(
    in fvec: GridFunctionClass&
    in cvec: GridFunctionClass&
    inout ChildGridPtr: GridClass*): void
+ InterpSW(
    in fvec: GridFunctionClass&
    in cvec: GridFunctionClass&
    inout ChildGridPtr: GridClass*): void
+ InterpSE(
    in fvec: GridFunctionClass&
    in cvec: GridFunctionClass&
    inout ChildGridPtr: GridClass*): void
+ InterpNW(
    in fvec: GridFunctionClass&
    in cvec: GridFunctionClass&
    inout ChildGridPtr: GridClass*): void
+ InterpNE(
    in fvec: GridFunctionClass&
    in cvec: GridFunctionClass&
    inout ChildGridPtr: GridClass*): void
+ InterpolateAndCorrect(
    in arg_FVEC: int
    in arg_CVEC: int ): void
+ InterpAndCorrectInterior(
    in fvec: GridFunctionClass&
```

```
    in cvec: GridFunctionClass&
    inout ChildGridPtr: GridClass*): void
+ InterpAndCorrectNSEW(
    in fvec: GridFunctionClass&
    in cvec: GridFunctionClass&
    inout ChildGridPtr: GridClass*): void
+ InterpAndCorrectSW(
    in fvec: GridFunctionClass&
    in cvec: GridFunctionClass&
    inout ChildGridPtr: GridClass*): void
+ InterpAndCorrectSE(
    in fvec: GridFunctionClass&
    in cvec: GridFunctionClass&
    inout ChildGridPtr: GridClass*): void
+ InterpAndCorrectNW(
    in fvec: GridFunctionClass&
    in cvec: GridFunctionClass&
    inout ChildGridPtr: GridClass*): void
+ InterpAndCorrectNE(
    in fvec: GridFunctionClass&
    in cvec: GridFunctionClass&
    inout ChildGridPtr: GridClass*): void
+ Project(
    inout arg_FinePtr: GridFunctionClass*
    inout arg_CoarsePtr: GridFunctionClass*): void
+ Project(
    in arg_FVEC: int
    in arg_CVEC: int ): void
+ ProjectResidual(): void
+ InitGPN(): void
+ InitGPS(): void
+ InitGPE(): void
+ InitGPW(): void
+ InterpGPN(
    inout arg_fvecPtr: GridFunctionClass*
    inout arg_cvecPtr: GridFunctionClass*
    in i: int ): void
+ InterpGPS(
    inout arg_fvecPtr: GridFunctionClass*
    inout arg_cvecPtr: GridFunctionClass*
    in i: int ): void
+ InterpGPE(
    inout arg_fvecPtr: GridFunctionClass*
    inout arg_cvecPtr: GridFunctionClass*
```

```
    in j: int ): void
+ InterpGPW(
    inout arg_fvecPtr: GridFunctionClass*
    inout arg_cvecPtr: GridFunctionClass*
    in j: int ): void
+ InitGPsWithTrueSoln(): void
+ ComputeSumOfAbsoluteValues(
    in arg_VEC: int ): real
+ ComputeSumOfAbsoluteValues(
    inout arg_vecPtr: GridFunctionClass*): real
+ ComputeSumOfSquares(
    in arg_VEC: int ): real
+ ClearGhostPoints(): void
+ ResetCOERHS(): void
+ get_myProcID(): int
+ set_myProcID(
    in arg_myProcID: int ): void
+ get_GridPtr(): GridClass*
+ get_BaseGridPtr(): GridClass*
+ set_GridPtr(
    inout arg_GridPtr: GridClass*): void
+ set_BaseGridPtr(
    inout arg_BaseGridPtr: GridClass*): void
+ get_NumGridFunctions(): int
+ get_GridFunctionPtrArray(): ArrayClass<GridFunctionClass*>&
+ get_GridFunctionPtrArray(
    in i: int ): GridFunctionClass*
+ CoefficientMatrixClass get_CoefficientMatrixPtr()*
+ get_gpn(): ArrayClass<real>&
+ get_gps(): ArrayClass<real>&
+ get_gpe(): ArrayClass<real>&
+ get_gpw(): ArrayClass<real>&
+ get_gpn_ptr(): ArrayClass<real>*
+ get_gps_ptr(): ArrayClass<real>*
+ get_gpe_ptr(): ArrayClass<real>*
+ get_gpw_ptr(): ArrayClass<real>*
+ get_gpn_flag(): ArrayClass<real>&
+ get_gps_flag(): ArrayClass<real>&
+ get_gpe_flag(): ArrayClass<real>&
+ get_gpw_flag(): ArrayClass<real>&
+ get_gpn_flag_ptr(): ArrayClass<real>*
+ get_gps_flag_ptr(): ArrayClass<real>*
+ get_gpe_flag_ptr(): ArrayClass<real>*
+ get_gpw_flag_ptr(): ArrayClass<real>*
```

```
  + get_gpsw(): real
  + get_gpse(): real
  + get_gpnw(): real
  + get_gpne(): real
  + get_gpsw_flag(): real
  + get_gpse_flag(): real
  + get_gpnw_flag(): real
  + get_gpne_flag(): real
  + set_GridFunctionLevelPtr(
      inout arg_GridFunctionLevelPtr: GridFunctionLevelClass*): void
  + get_GridFunctionLevelPtr(): GridFunctionLevelClass*
  + get_NumChildren(): int
  + get_ChildPtr(
      in i: int ): GridFunctionArrayClass*
  + get_ChildPtrArray(): ArrayClass<GridFunctionArrayClass*>&
  + set_ParentPtr(
      inout arg_ParentPtr: GridFunctionArrayClass*): void
  + get_ParentPtr(): GridFunctionArrayClass*
  + get_phisavePtr(): GridFunctionClass*
  + get_LePtr(): GridFunctionClass*
  + get_eTempPtr(): GridFunctionClass*
  + get_OnlyPostSmoothing(): int
  + get_NumSmootherStepsPerLevel(): int
  + get_UseStandardSmoother(): int
  + get_NumCacheLines(): int
  + get_NumGridPointsInCache(): int
  + get_UseCacheAwareSmoother(): int
  + get_UseCacheAwareSmootherXY(): int
  + get_ComputeResidualInSmoother(): int
+-----------------------------------------------------------------+
```

.2.3 Grid Function Level Class

A union of grid function classes are combined to form a grid level function class. This class has methods for coordinating operations applied to an entire level.

The Uniform Modeling Language (UML) diagram for this class is

```
+-----------------------------------------------------------------+
  GridFunctionLevelClass
+-----------------------------------------------------------------+
  - NumFunctionArrays: int
  - NumGridFunctions: int
  - UseStandardSmoother: int
  - UseCacheAwareSmoother: int
```

```
  - OnlyPostSmoothing: int
  - NumSmootherStepsPerLevel: int
  - NumSmootherSteps: int
  - GridFunctionLevelPtrArray: ArrayClass<GridFunctionArrayClass*>
  - GridFunctionLevelParentPtr: GridFunctionLevelClass*
  - GridFunctionCompositePtr: GridFunctionCompositeClass*
  - e_rms: double
+----------------------------------------------------------------+
  + GridFunctionLevelClass(
      in arg_NumGridFunctions=NUM_GF: int )
  + GridFunctionLevelClass(
      inout arg_name: char*
      in arg_NumGridFunctions=NUM_GF: int )
  + GridFunctionLevelClass(
      in arg_GridFunctionLevel: ArrayClass<GridFunctionArrayClass*>&)
  + GridFunctionLevelClass(
      inout arg_name: char*
      in arg_GridFunctionLevel: ArrayClass<GridFunctionArrayClass*>&)
  + GridFunctionLevelClass(
      in arg_GridFunctionLevel: GridFunctionLevelClass&)
  + operator=(
      in arg_GridFunctionLevel: GridFunctionLevelClass&): GridFunctionLevelClass&
  + operator()(
      in i: int): GridFunctionArrayClass*
  + operator()(
      in i: int) const: const GridFunctionArrayClass*
  + operator<<(
      in o: ostream&
      in arg_GridFunctionLevel: GridFunctionLevelClass&): friend ostream&
  + ~GridFunctionLevelClass()
  + InitBaseGridFunctionArray(
      inout arg_BaseGridPtr: GridClass*
      in arg_myProcID=0: int
      in arg_NumGridFunctions=NUM_GF: int ): void
  + AddGridFunctionArray(
      inout arg_GridPtr: GridClass*
      inout arg_BaseGridPtr: GridClass*
      inout arg_ParentGridPtr: GridClass*
      in arg_myProcID=0: int ): void
  + Vcycle_Project_Stage00(): void
  + Vcycle_Project_Stage01(): void
  + Vcycle_Project_Stage02(): void
  + Vcycle_CoarseGridSolve(
      in arg_RHS = RES: int ): void
```

```
+ Vcycle_Correct_Stage01(): void
+ Vcycle_Correct_Stage02(): void
+ Lnfl(): void
+ Lsuball(
    in lfinest = -1: int ): void
+ FluxMatch(): void
+ FluxMatchResidual(): void
+ InitGPFlags(): void
+ InitNeighborGPs(
    in arg_IND = SOL: int
    in arg_RHS = RHS: int ): void
+ InitGPs(): void
+ ComputeResidual(
    in arg_SOL = SOL: int
    in arg_RHS = RHS: int
    in arg_lfinest = -1: int ): void
+ InitGPsWithTrueSoln(): void
+ ComputeTrueSolution(
    in arg_SOL = SOL: int ): void
+ Interpolate(): void
+ Project(): void
+ Smooth(
    in arg_NumIts = -1: int ): void
+ SmoothNSEW_WithIntegratedInterpolation(): void
+ SmoothNSEW_WithIntegratedInterpolationAndFullResidualComputation(): void
+ Smooth_WithFullResidualComputation(): void
+ ComputeError(): void
+ ComputeSumOfAbsoluteValues(
    in arg_VEC: int ): real
+ ComputeSumOfSquares(
    in arg_VEC: int ): real
+ InterpolateGhostPoints(
    in arg_lfinest = -1: int
    in arg_SOL = SOL: int ): void
+ ClearGhostPoints(): void
+ ResetCOERHS(): void
+ get_BaseGridPtr(): GridClass*
+ get_NumFunctionArrays(): int
+ get_GridFunctionLevelPtrArray(): ArrayClass<GridFunctionArrayClass*>&
+ get_GridFunctionLevelPtrArray(
    in arg: int ): GridFunctionArrayClass*
+ get_NumGridFunctions(): int
+ set_NumGridFunctions(
    in arg_NumGridFunctions: int ): void
```

```
  + get_GridFunctionLevelParentPtr(): GridFunctionLevelClass*
  + set_GridFunctionLevelParentPtr(
      inout arg_GridFunctionLevelParentPtr: GridFunctionLevelClass*): void
  + get_GridFunctionCompositePtr(): GridFunctionCompositeClass*
  + set_GridFunctionCompositePtr(
      inout arg_GridFunctionCompositePtr: GridFunctionCompositeClass*): void
  + get_e_rms(): double
  + set_e_rms(
      in arg_e_rms: double ): void
  + get_UseStandardSmoother(): int
  + get_UseCacheAwareSmoother(): int
  + get_OnlyPostSmoothing(): int
  + get_NumSmootherStepsPerLevel(): int
+-----------------------------------------------------------------+
```

.2.4 Grid Function Composite Class

The union of grid level function classes are combined into a composite grid function class. This class has methods for applying operations over all the grid level functions of the grid hierarchy as a whole.

The Uniform Modeling Language (UML) diagram for this class is

```
+-----------------------------------------------------------------+
  GridFunctionCompositeClass
+-----------------------------------------------------------------+
  - NumGridFunctions: int
  - GridFunctionLevelPtrArray: ArrayClass<GridFunctionLevelClass*>
  - AMRMGPtr: AMRMGClass*
  - OnlyPostSmoothing: int
  - ComputeResidualInSmoother: int
+-----------------------------------------------------------------+
  + GridFunctionCompositeClass()
  + GridFunctionCompositeClass(
      inout arg_name: char*
      in arg_NumLevels=1: int )
  + GridFunctionCompositeClass(
      inout: ArrayClass<GridFunctionLevelClass*>
      in rg_GridFunctionLevelPtrArray: a&)
  + GridFunctionCompositeClass(
      inout arg_name: char*
      inout: ArrayClass<GridFunctionLevelClass*>
      in rg_GridFunctionLevelPtrArray: a&)
  + GridFunctionCompositeClass(
      in: GridFunctionCompositeClass
```

```
    in rg_GridFunctionComposite: a&)
+ operator=(
    in: GridFunctionCompositeClass
    in rg_GridFunctionComposite: a&): GridFunctionCompositeClass&
+ operator()(
    in i: int): GridFunctionLevelClass*
+ operator()(
    in i: int) const: const GridFunctionLevelClass*
+ operator<<(
    in o: ostream&
    in arg_GridFunctionComposite: GridFunctionCompositeClass&): friend ostream&
+ ~GridFunctionCompositeClass()
+ InitBaseGridFunctionArray(
    inout arg_BaseGridPtr: GridClass*
    in arg_myProcID=0: int
    in arg_NumGridFunctions=NUM_GF: int ): void
+ AddLevel(
    inout arg_BaseGridPtr: GridClass*): void
+ AddGridFunctionArray(
    inout arg_GridPtr: GridClass*
    inout arg_BaseGridPtr: GridClass*
    inout arg_ParentGridPtr: GridClass*
    in arg_myProcID=0: int ): void
+ Vcycle(
    in arg_FineLevelIndex = -1: int ): void
+ Lsuball(
    in lfinest = -1: int ): void
+ InitGPFlags(): void
+ InitNeighborGPs(
    in arg_IND = SOL: int ): void
+ Interpolate(): void
+ Project(): void
+ ComputeResidual(
    in arg_SOL = SOL: int
    in arg_RHS = RHS: int
    in arg_lfinest = -1: int ): void
+ ComputeError(): void
+ ComputeNorm(
    in arg_VEC: int
    in arg = 2: int ): real
+ Smooth(
    in arg_NumIts = -1: int ): void
+ ResetCOERHS(): void
+ get_NumLevels(): int
```

```
  + set_NumLevels(
      in arg_NumLevels: int ): void
  + get_GridFunctionLevelPtrArray(): ArrayClass<GridFunctionLevelClass*>&
  + get_GridFunctionLevelPtrArray(
      in i: int ): GridFunctionLevelClass*
  + set_NumGridFunctions(
      in arg_NumGridFunctions: int ): void
  + get_NumGridFunctions(): int
  + get_AMRMGPtr(): AMRMGClass*
  + set_AMRMGPtr(
      inout arg_AMRMGPtr: AMRMGClass*): void
  + get_OnlyPostSmoothing(): int
  + get_ComputeResidualInSmoother(): int
+----------------------------------------------------------------+
```

.3 AMRMG Class

There is an interface class, AMRMGClass, that is composed of the grid hierarchy class and the composite grid function class. This class has methods for initializing the grid hierarchy and grid function(s) and for coordinating a solution method (e.g., multigrid V-Cycle).

The Uniform Modeling Language (UML) diagram for this class is

```
+----------------------------------------------------------------+
  AMRMGClass
+----------------------------------------------------------------+
  - NumGrids: int
  - GridHierarchy: GridHierarchyClass*
  - GridFunctionComposite: GridFunctionCompositeClass*
  - NumGridFunctions: int
  - RefinementFactor: int
  - MinPatchSize: int
  - ProcID: int
  - ProcX: int
  - ProcY: int
  - OnlyPostSmoothing: int
  - NumSmootherStepsPerLevel: int
  - NumSmootherSteps: int
  - ComputeResidualInSmoother: int
  - private_create(): void
+----------------------------------------------------------------+
  + AMRMGClass()
  + AMRMGClass(
      in argc: int
      inout argv: char**)
```

```
+ AMRMGClass(
    inout arg_name: char*)
+ AMRMGClass(
    inout arg_name: char*
    in arg_ProcID: int )
+ ~AMRMGClass()
+ AMRMGClass(
    in arg_AMRMGClass: AMRMGClass&)
+ operator=(
    in arg_AMRMGClass: AMRMGClass&): AMRMGClass&
+ operator<<(
    in o: ostream&
    in arg_AMRMG: AMRMGClass&): friend ostream&
+ Bing(): void
+ InitBaseGrid(): void
+ Refine(): void
+ Vcycle(): int
+ Lsuball(): void
+ Init(): void
+ ComputeError(): void
+ Vcycle_CoarseGridSolve(
    in arg_RHS = RES: int ): void
+ ComputeNorm(
    in arg_VEC: int
    in arg = 2: int ): real
+ AddGrid(
    inout GridPtr: GridClass*
    in ParentIndex: int
    in arg_procID: int
    in arg_procX: int
    in arg_procY: int
    in arg_sx: RealCoordsClass&
    in arg_ex: RealCoordsClass&
    in arg_si: GridCoordsClass&
    in arg_ei: GridCoordsClass&
    in arg_ni: GridCoordsClass&): void
+ Smooth(
    in arg_NumIts: int ): void
+ get_GridArray(): ArrayClass<GridClass*>&
+ set_GridArray(
    inout  arg_GridArray: ArrayClass<GridClass*>&): void
+ get_NumGrids(): int
+ set_NumGrids(
    in arg_NumGrids: int ): void
```

```
  + get_GridHierarchy(): GridHierarchyClass*
  + get_GridFunctionComposite(): GridFunctionCompositeClass*
  + get_NumGridFunctions(): int
  + set_NumGridFunctions(
      in arg_NumGridFunctions: int ): void
  + get_RefinementFactor(): int
  + set_RefinementFactor(
      in arg_RefinementFactor: int ): void
  + get_MinPatchSize(): int
  + set_MinPatchSize(
      in arg_MinPatchSize: int ): void
  + get_OnlyPostSmoothing(): int
  + get_NumSmootherStepsPerLevel(): int
  + get_ComputeResidualInSmoother(): int
+-----------------------------------------------------------------+
```

.4 Supporting Classes

.4.1 Coords Class

```
+-----------------------------------------------------------------+
  CoordsClass
+-----------------------------------------------------------------+
  # NumDims: int
+-----------------------------------------------------------------+
  + CoordsClass()
  + CoordsClass(
      inout arg_name: char*)
  + CoordsClass(
      in arg_NumDims: int )
  + CoordsClass(
      inout arg_name: char*
      in arg_NumDims: int )
  + ~CoordsClass()
  + CoordsClass(
      in arg_CoordsClass: CoordsClass&)
  + CoordsClass(
      in CoordsClass arg_CoordsClass: const&)
  + operator=(
      in CoordsClass arg_CoordsClass: const&): CoordsClass&
  + operator<<(
      in o: ostream&
      in arg_CoordsClass: CoordsClass&): friend ostream&
```

```
  + get_NumDims(): int
  + NumDims(): int get_const
  + set_NumDims(
      in arg_NumDims: int ): void
+----------------------------------------------------------------+
```

.4.2 Array Class

```
+----------------------------------------------------------------+
  ArrayClass
+----------------------------------------------------------------+
  # v1d: Type*
  # v2d: Type**
  # v3d: Type***
  # v4d: Type****
  # nx, ny, nz, nw, n: int
  # bx, by, bz, bw: int
  # numDimns: int
  # isEmptyFlag: int
  # isPtr: int
  # isShell: int
  # isFlat: int
  # FortranIndexingFlag: int
  # private_create(): void
  # private_copy(
      in A: ArrayClass<Type> const&): void
  # private_destroy(): void
+----------------------------------------------------------------+
  + ArrayClass(
      in nx_=0: int
      in ny_=0: int
      in nz_=0: int
      in nw_=0: int )
  + ArrayClass(
      inout name_: char*
      in nx_=0: int
      in ny_=0: int
      in nz_=0: int
      in nw_=0: int )
  + ArrayClass(
      inout v_: Type*
      in nx_: int
      in ny_=0: int
      in nz_=0: int
```

```
    in nw_=0: int )
+ ArrayClass(
    inout name_: char*
    inout v_: Type*
    in nx_: int
    in ny_=0: int
    in nz_=0: int
    in nw_=0: int )
+ ArrayClass(
    in  A: ArrayClass<Type>&)
+ operator=(
    in A: ArrayClass<Type> const&): ArrayClass&
+ operator=(
    in Type a: const&): ArrayClass&
+ ~ArrayClass()
+ allocate(
    in nx_=0: int
    in ny_=0: int
    in nz_=0: int
    in nw_=0: int ): void
+ allocate(
    inout v_: Type*
    in nx_: int
    in ny_=0: int
    in nz_=0: int
    in nw_=0: int ): void
+ append(
    in v_: Type ): void
+ append(): void
+ createPtr1d(
    in  A: ArrayClass<Type>&): void
+ createPtr2d(
    in  A: ArrayClass<Type>&): void
+ createPtr3d(
    in  A: ArrayClass<Type>&): void
+ createPtr4d(
    in  A: ArrayClass<Type>&): void
+ deallocate(): void
+ FortranIndexing(): void
+ CIndexing(): void
+ isFortranIndexing(): int
+ operator()(
    in x: int ): Type&
+ operator()(
```

```
    in x: int ) const: const Type&
+ operator()(
    in x: int
    in y: int ): Type&
+ operator()(
    in x: int
    in y: int
    in z: int ): Type&
+ operator()(
    in x: int
    in y: int
    in z: int
    in w: int ): Type&
+ operator+(
    in  A_: ArrayClass<Type>&): ArrayClass<Type>
+ operator+(
    in _A: const ArrayClass<Type>&): ArrayClass<Type>
+ operator+=(
    in  A_: ArrayClass<Type>&): ArrayClass<Type>&
+ operator+=(
    in _A: const ArrayClass<Type>&): ArrayClass<Type>&
+ operator+(
    in a: Type ): ArrayClass<Type>
+ operator+=(
    in a: Type ): ArrayClass<Type>&
+ operator-(
    in  A_: ArrayClass<Type>&): ArrayClass<Type>
+ operator-(
    in _A: const ArrayClass<Type>&): ArrayClass<Type>
+ operator-=(
    in  A_: ArrayClass<Type>&): ArrayClass<Type>&
+ operator-=(
    in _A: const ArrayClass<Type>&): ArrayClass<Type>&
+ operator-(
    in a: Type ): ArrayClass<Type>
+ operator-=(
    in a: Type ): ArrayClass<Type>&
+ operator*( in a: Type ): ArrayClass<Type>
+ operator*=( in a: Type ): ArrayClass<Type>&
+ operator*(
    in arg: T
    in arg_Array: ArrayClass<T>&): template < class T> friend ArrayClass<T>&
+ norm(): Type
+ maxval(): Type
```

```
+ minval(): Type
+ zero(): void
+ randfill(): void
+ setall(
    in a: Type ): void
+ incall(
    in a: Type ): void
+ constMult(
    in a: Type ): void
+ abs(): void
+ transpose(): void
+ printDimns(
    inout msg=NULL: char*): void
+ printData(
    in k0=0: int ): void
+ printData(
    in o: ostream&
    in k0=0: int ): void
+ friend ostream operator<<(
    in o: ostream&
    in  arg_ArrayClass: ArrayClass<T>&): template < class T>&
+ fout(
    in fio: fstream&): void
+ fout(
    in fout: ofstream&): void
+ fin(
    in fio: fstream&): void
+ fin(
    in fin: ifstream&): void
+ ptr(
    in nx_=0: int
    in ny_=0: int
    in nz_=0: int
    in nw_=0: int ): Type*
+ ptr(
    in nx_=0: int
    in ny_=0: int
    in nz_=0: int
    int nw_=0): Type const*
+ ptr2d(
    in nx_=0: int
    in ny_=0: int
    in nz_=0: int
    in nw_=0: int ): Type**
```

```
  + get_nx(): int
  + get_ny(): int
  + get_nz(): int
  + get_nw(): int
  + get_nx(): const int
  + get_ny(): const int
  + get_nz(): const int
  + get_nw(): const int
  + get_bx(): int
  + get_by(): int
  + get_bz(): int
  + get_bw(): int
  + set_bx(
      in b: int ): int
  + set_by(
      in b: int ): int
  + set_bz(
      in b: int ): int
  + set_bw(
      in b: int ): int
  + get_n(): int
  + size(): int
  + isEmpty(): int
  + get_numDimns(): int
  + get_numDimns(): const int
  + extractDataPtr1d(): Type*
  + extractDataPtr2d(): Type**
  + extractDataPtr3d(): Type***
  + extractDataPtr4d(): Type****
+----------------------------------------------------------------+
```