Algebraic Multigrid Methods for Higher-Order Finite Element Discretization with Parallelization

A dissertation submitted for the degree of

Master of Science
in
Computational Mathematics with Modelling

by

Michael Wallner

September 2012

Supervisor:
Dr. Matthias Maischak

Brunel University London
School of Information Systems, Computation and Mathematics

Abstract

This thesis analyzes a new algebraic multigrid (AMG) method for algebraic systems arising from the discretization of second order elliptic boundary value problems by high-order finite element methods on quadrilaterals in two dimensions. The new AMG method is developed to analyze the sparse stiffness matrix from a discretization with biquadratic or bicubic Lagrangian finite elements and to recover the bilinear stiffness matrix associated with the same underlying mesh by algebraic means only. The method assumes the knowledge of degree and the usage of nodal Lagrangian basis functions, however the underlying grid's geometry stays unknown. This approach is motivated by far higher efficiency on discretizations of (bi)linear finite elements. The gained linear system can be solved with any classical AMG solver, where the smoothed aggregation method is chosen and carefully analyzed and adapted for parallel execution. Moreover, a brief introduction into all needed general AMG concepts is given. All developed algorithms are designed with special focus on fast and efficient parallel execution using OpenMP, if applicable. The efficiency of the new method is presented in numerical results by contrasting it with the classical AMG method which is directly applied to the high-order finite element matrix, showing a much shorter solution time.

Contents

Acknowledgements

First of all I want to thank my supervisor, Dr. Matthias Maischak, for arousing and deepening my interest in this cutting-edge field of research as well as his support in the preparation of this thesis. He provided me with the software package `maiprogs` which founded the basis of all my numerical work and enabled me to perform a vast range of experiments, which would not have been possible without.

Furthermore I want to thank my parents Margarete and Hans Wallner for their moral and financial support during my studies at Brunel University.

Last but not least I am greatly indebted to my girlfriend Birgit Ondra, for her constant support during all times of my studies and all the fruitful discussions which motivated me to go into more depth in various areas of my research.

Introduction

The numerical solution of partial differential equations (PDEs) lies at the heart of most mathematical problems arising in industry and throughout the natural sciences. Due to the rising complexity of realistic models and the constantly growing cost pressure in the market, efficient and fast solution strategies gain more and more importance. Computer simulations are the state of the art in nearly all new developments, as real experiments are too expensive and time consuming. The ever rising demand for more accurate answers cannot be met by the continuously increasing computing power of modern computer chips alone. A clear consequence is the need for highly efficient algorithms in all areas of research.

This work focuses on a new solution method for solving large scale algebraic systems arising form the discretization of elliptic PDEs. These are for instance used as models for the temperature distribution in solid or fluid media and as models for the concentration distribution of certain substances in diffusive media. A cutting edge technology for the solution of these problems are multigrid methods, which can be developed by two approaches: the geometric approach [2,3,14,33,44] and the algebraic approach [15,16,23,27,32–35,37]. Multigrid methods are well known to be by far the most efficient methods for solving this and other classes of PDEs [4].

The development of multigrid methods started in the 1960s with the geometric approach, where the geometry of the problem was used to define the various multigrid components. They were first introduced by Fedorenko in 1961 [11] but their actual efficiency was only realized more than 10 years later in the 1970s by the work of Brandt [3]. In the same decade the method was independently rediscovered and extended by Hackbusch [13]. Whereas the research on the algebraic approach started in the 1980s. Algebraic multigrid (AMG) methods only use the information available in the linear system of equations $Ax = b$, which makes them applicable as solvers on more complicated domains and unstructured grids, where geometric multigrid is often too difficult to apply. Another advantage is the possibility of an easy integration in existing software packages, as only the information of the linear system is needed. This makes them very popular in industry.

Out of all available papers this thesis has been motivated by the work of Shu et al. [31] and represents an extension of the methods on meshes of triangles onto meshes of quadrilaterals with the aim of a parallel implementation. It proposes to be a multigrid solver which combines both approaches and may be called “algebraic multigrid method based on geometric considerations” [31, p. 347].

The coarsening in a typical AMG method is done by examining the algebraic properties of the coefficient matrix. For doing so a vast range of algorithms exists, which work well on some

classes of problems, but are far away from being optimal on more complicated ones. Especially the agglomeration of elements brings difficulties and results in problems in the control of the coarse degrees of freedom. The idea in [31] is to utilize hidden geometric information which naturally comes with a specific class of problems. In particular, stiffness matrices from finite element discretizations reflect the structure of the underlying geometric grid in a more or less obvious way. In order to use this hidden information, additional input is necessary, which results in a hybrid method, as it relies not only on the linear system as sole input to the algorithm. However, the actual geometric data remains unknown.

The class of problems to be solved arises from the discretization with Lagrangian finite elements of different orders. In general classical AMG methods show good performance on (bi-)linear elements only and are far less efficient on higher-order elements. A main reason for this behavior is that the graph of the stiffness matrix is nearly the same as the graph of the finite element grid, whereas this property is lost for higher-order elements. Additionally, higher-order elements result in more interactions between basis functions and lead to more dense stiffness matrices. Hence, the coarsening process becomes more involved and requires special considerations.

Within the solving process the main idea is to algebraically recover the (bi-)linear finite element space and its associated stiffness matrix. For this purpose special methods have been introduced in [31] for general triangulations, which were adapted to meshes of general quadrilaterals. The recovered (bi-)linear system is solved with a classical AMG method, where the smoothed aggregation approach (cf. [35]) in combination with a SOR- or Gauss-Seidel-smoother was mainly used. Theoretically this technique can be visualized as a two-level method, where the coarse space is the (bi-)linear finite element space associated with the same geometrical grid.

The approaches mentioned will be described in accordance with the following structure. In chapter 1 we introduce mathematical and technical preliminaries, which are essential in the understanding of the presented theory. Chapter 2 introduces the concept of AMG in detail and discusses the smoothed aggregation method and the classical Ruge-Stüben coarsening strategy in combination with direct and standard interpolation. Special focus is laid on parallelization possibilities of the introduced algorithms, and the data structure is designed to enable this potential. Chapter 3 lies at the center of this work. It introduces the new method for biquadratic and bicubic finite elements including comments on the parallel implementation of the presented algorithms. Chapter 4 describes the implementations from a technical perspective by presenting the program architecture. Chapter 5 presents different numerical results, which show the efficiency of the new method, compared to classical AMG algorithms. The algorithms were implemented in `maiprogs` using OpenMP for the purpose of parallelization. Finally a summary of the achieved results is given and open problems with regard to this topic are suggested for future research.

Chapter 1

Mathematical Preliminaries

1.1 Notation and Problem Class for Higher-Order Finite Element Problems

In the following the notation for the higher-order finite element algorithm developed in chapter 3 is introduced. The algebraic multigrid method presented is designed for finite element discretization of second-order finite element boundary value problems. We are mostly going to use and extend the notation developed in [31]. The problem class we are going to solve can be generally described by the following model problem

$$\begin{cases} -\nabla\left(d(x)\nabla u\right) = f, & \text{in } \Omega, \\ u = 0, & \text{on } \partial\Omega, \end{cases} \tag{1.1}$$

where $\Omega \subset \mathbb{R}^2$ represents a polygonal domain, $x = (x_1, x_2)^\top \in \mathbb{R}^2$ and $d(x)$ is a function that is positive and bounded, but may contain large discontinuous jumps. For the algorithm presented in chapter 3.3 we assume $d(x)$ to be piecewise constant on the given grid.

Let R^h be a quadrilateral partition of the domain Ω and $\mathbb{P}_p$ the set of polynomials of degrees not more than p, where h is the longest side of all elements in R^h. We denote the finite element space by

$$V_h^p := \{\varphi \in H_0^1(\Omega) : \varphi|_R \in \mathbb{P}_p, \forall R \in R^h\}.$$

In particular we will work with bilinear ($p = 1$), biquadratic ($p = 2$) and bicubic ($p = 3$) elements. The finite element solution of equation (1.1) is denoted by $u_h^p \in V_h^p$ and satisfies the following finite element scheme

$$a(u_h^p, \varphi) = (f, \varphi), \qquad \forall \varphi \in V_h^p,$$

where we are using $a(u, v) = \int_\Omega d(x)\left(u_{x_1} v_{x_1} + u_{x_2} v_{x_2}\right) dx$, $(f, u) = \int_\Omega f u \, dx$ and the Sobolev space $H^m(\Omega) = \{v \mid \partial^\alpha v \in L^2(\Omega), |u| \leq m\}$.

Let $\{\phi_i\}_{i=1}^N$ be the standard nodal basisfunction of V_h^p, where N denotes the total degrees of freedom. Hence we can represent the solution of (1.1) in this basis by $u_h^p = \sum_{i=1}^N u_{h,i}^p \phi_i$ and substitute it into (1.1) to obtain the equivalent algebraic system

$$A_h^p u_h^p = f_h^p \tag{1.2}$$

where

$$A_h^p = (a(\phi_i, \phi_j))_{i,j\in I}, u_h^p = \left(u_{h,i}^p\right)_{i\in I} \text{ and } f_h^p = (f, \phi_i)_{i\in I}$$

with $I = \{1, \ldots, N\}$.

1.2 Sparse Matrix

The coefficient matrix A of any linear system $Ax = b$ dealt with in this thesis is stored in a sparse form in order to reduce the cost of memory and increase the efficiency. Let $A \in \mathbb{R}^{N\times N}$ be a sparse matrix with Nz nonzero entries. A popular sparse matrix data structure also used by `maiprogs` is the *Compressed Sparse Row* (CSR) format (cf. [28]). In this format the matrix A is stored in three one-dimensional arrays: AA, JA and IA. These are explained in table 1.1.

Array	Data type	Size	Usage
AA	Real	Nz	The real values a_{ij} of the matrix A are stored row by row, from row 1 to N
JA	Integer	Nz	Contains the column indices of the element a_{ij} as stored in the array AA
IA	Integer	$N+1$	Contains pointers to the beginning of each row in the arrays AA and JA. The element $IA(N+1) = IA(1)+Nz$ and can be interpreted as the beginning of the fictitious row number $N+1$. I.e. the elements of row i are stored at the positions $IA(i)$ to $IA(i+1)-1$ in AA and JA.

Table 1.1: Components of the CSR Format explained

Example 1.1: Consider the sparse matrix

$$A = \begin{pmatrix} 1. & 0. & 0. & 2. \\ 0. & 3. & 4. & 0. \\ 0. & 0. & 5. & 0. \\ 6. & 7. & 0. & 8. \end{pmatrix}.$$

For this example $N = 4$, $Nz = 8$ and the CSR arrays hold the following values:

$$\begin{array}{rcl} \text{AA} & = & (1. \quad 2. \quad 3. \quad 4. \quad 5. \quad 6. \quad 7. \quad 8.) \\ \text{JA} & = & (1 \quad 4 \quad 2 \quad 3 \quad 3 \quad 1 \quad 2 \quad 3\ ) \\ \text{IA} & = & (1 \quad 3 \quad 5 \quad 6 \quad 9\ ) \end{array}$$

■

According to Shu et al. [31] we introduce the following assumption.

Assumption 1.2 (Nonzero Assumption): If the system matrix arises as the stiffness matrix of a finite element discretization, we assume that for any entry of an element stiffness

matrix, if its row and column indices relate to nodes which are not on the Dirichlet boundary, it will be stored in the CSR format of A in the process of assembling the global stiffness matrix A. ■

From the above assumption it follows that there exists an entry with the row index i and the column index j in the CSR format, if and only if the corresponding non-Dirichlet boundary nodes x_i and x_j belong to the same element, that is

$$\operatorname{supp}\phi_i \cap \operatorname{supp}\phi_j \neq \emptyset \tag{1.3}$$

where ϕ_i and ϕ_j are the corresponding nodal basisfunctions.

1.3 Norms

For the theoretical discussions the system matrix A is always assumed to be symmetric and positive definite. Therefore we are able to define the following scalar products, where $D = \operatorname{diag}(A)$ denotes the diagonal of A:

$$\begin{aligned}(u,v)_{\mathcal{H}^0} &:= (Du, v)_2\\ (u,v)_{\mathcal{H}^1} &:= (Au, v)_2\\ (u,v)_{\mathcal{H}^2} &:= (D^{-1}Au, Av)_2\end{aligned}$$

The corresponding norms are $\|u\|_{\mathcal{H}^i} := \sqrt{(u,u)_{\mathcal{H}^i}}$, $i \in \{0,1,2\}$. In some sense these norms are discrete counterparts of the H^k-Sobolev (semi-)norms (see [33, 41]).

1.4 Lagrange Shape Functions on Rectangles in 2D

For the purpose of applying the finite element discretization, the definition of a set of basisfunctions $\phi_i(\xi)$, $i = 1, \ldots, N$, defined on a reference element is necessary. An introduction into the finite element method and a more detailed description can be found in [18,29,46]. We are going to introduce Lagrange shape functions on rectangles in two dimensions following [12], as they are needed for the theoretical discussions in chapter 3. On these rectangles we will work with bilinear, biquadratic and bicubic functions. For more details see [38, pp. 179].

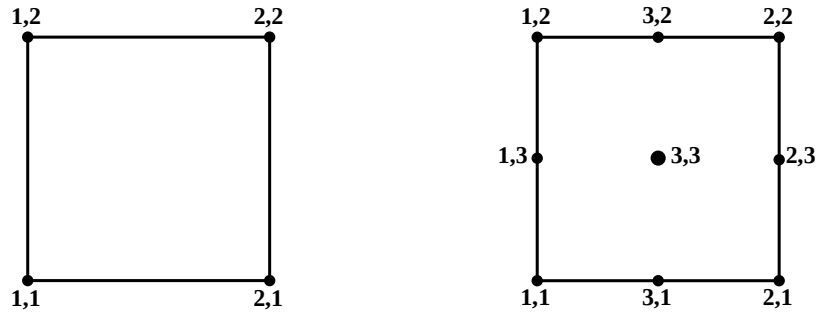
Lagrangian interpolants are simple to construct on rectangles by taking products of one dimensional Lagrange polynomials. As a reference element we choose the canonical 2×2 square $\{(\xi,\eta) \mid -1 \leq \xi, \eta \leq 1\}$. For our set of basisfunctions of degree d we choose $d+1$ equally spaced points along each unit coordinate axis, and form the Cartesian product of all possible pairs (ξ_i, η_j). This set is called the grid points and is shown in figure 1.1. Now we associate every element of our set of basisfunctions with one grid point. Consequently, element (i,j) represented by $N_{i,j}(\xi,\eta)$ is a polynomial with the properties that

- for every monomial term in $N_{i,j}(\xi,\eta)$, the exponents of ξ and η are between 0 and d,
- $N_{i,j}(\xi_i, \eta_j) = 1$,

- $N_{i,j}(\xi_k, \eta_l) = 0$, for all grid points $(\xi_k, \eta_l) \neq (\xi_i, \eta_j)$.

This properties clearly imply, that the following formula for the element (i, j) of d-th degree family of basisfunctions is given as

$$N_{i,j}(\xi, \eta) = \frac{\prod_{k=1;k\neq i}^{d+1}(\xi - \xi_k)\prod_{l=1;l\neq j}^{d+1}(\eta - \eta_l)}{\prod_{k=1;k\neq i}^{d+1}(\xi_i - \xi_k)\prod_{l=1;l\neq j}^{d+1}(\eta_j - \eta_l)}.$$



(a) Grid points for bilinear functions (b) Grid points for biquadratic functions

Figure 1.1: Grid points and indexing on rectangular reference element for bilinear and biquadratic basisfunctions

This construction leads us to the set of a *Lagrange polynomial basis*. It is defined over a set of polynomials and points where every polynomial is associated with exactly one point, at which it attains the value 1 but is zero at all other points [7].

The restriction of the bilinear Lagrange basisfunction $\psi_{i,j}(\xi, \eta)$ to the canonical reference element has the form

$$\psi(\xi, \eta) = c_{1,1}N_{1,1}(\xi, \eta) + c_{1,2}N_{1,2}(\xi, \eta) + c_{2,1}N_{2,1}(\xi, \eta) + c_{2,2}N_{2,2}(\xi, \eta).$$

The *reference bilinear shape functions* $N_{i,j}(\xi, \eta)$ satisfy

$$N_{i,j}(\xi_k, \eta_l) = \delta_{ik}\delta_{jl}, \qquad k, l = 1, 2.$$

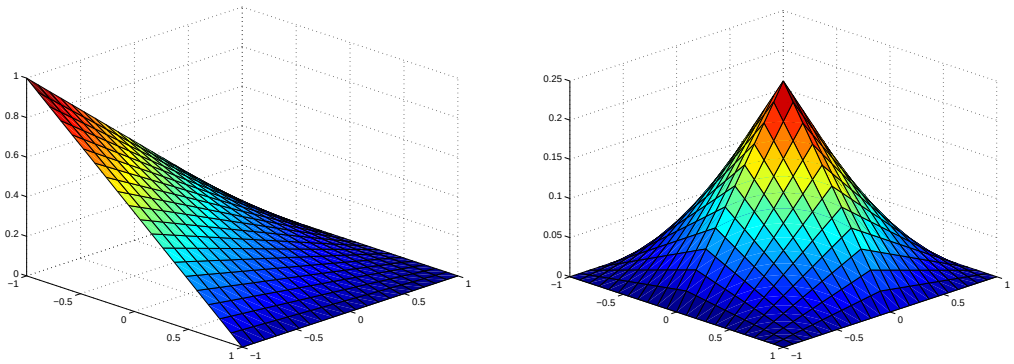
This implies $\psi(\xi_k, \eta_l) = c_{k,l}$ and we are able to write $N_{i,j}$ as the product of the one dimensional hat functions

$$N_{i,j}(\xi, \eta) = \hat{N}_i(\xi)\hat{M}_j(\eta), \qquad i, j = 1, 2,$$

with the following functions for $-1 \leq \xi, \eta \leq 1$

$$\hat{N}_1(\xi) = \frac{1-\xi}{2}, \qquad \hat{M}_1(\eta) = \frac{1-\eta}{2},$$
$$\hat{N}_2(\xi) = \frac{1+\xi}{2}, \qquad \hat{M}_2(\eta) = \frac{1+\eta}{2}.$$

The shape of function $N_{1,1}(\xi, \eta)$ is shown in figure 1.2a. It is a linear function along the edges containing node (ξ_i, η_j) and vanishes on the opposite edges.



(a) Bilinear basisfunction $N_{1,1}$ on $[-1,1] \times [-1,1]$ reference element

(b) Bilinear basisfunction on intersection of four squares

Figure 1.2: Bilinear basisfunction

The basisfunction $\psi_{i,j}$ can be constructed out of these reference shape function by combining them on elements sharing a node with the condition to create a continuous function. An example is shown in figure 1.2b where the support of the function $\psi_{i,j}$ consists of four square elements.

The construction of biquadratic shape functions follows a similar manner. As their degree is 2 we require 9 grid points on the reference element. Additionally to the four vertex points we use four points on the mid-sides and one at the center of the element (see figure 1.1b). By restricting the basisfunction $\phi(\xi, \eta)$ to this reference element we obtain the form

$$\phi(\xi, \eta) = \sum_{i=1}^{3} \sum_{j=1}^{3} c_{i,j} N_{i,j}(\xi, \eta),$$

where similar to the case before $N_{i,j}$, $i, j = 1, 2, 3$ are the *reference biquadratic shape functions* constructed as the product of one dimensional quadratic polynomial Lagrange shape functions

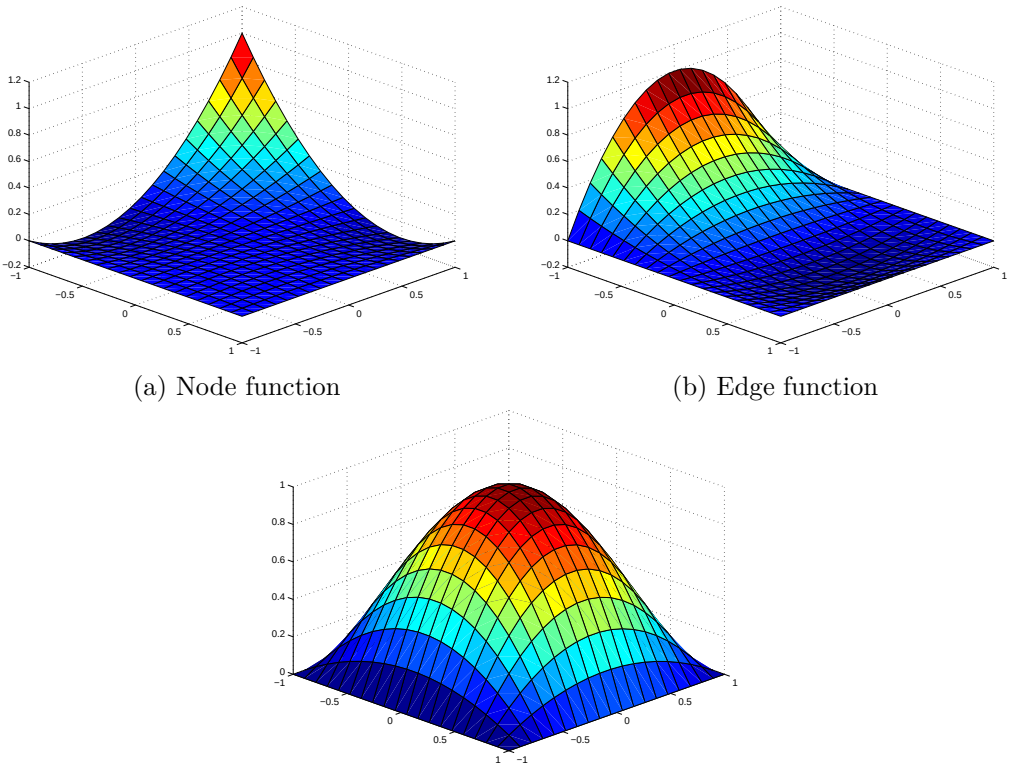
$$N_{i,j}(\xi, \eta) = \hat{N}_i(\xi)\hat{M}_j(\eta), \qquad i, j = 1, 2, 3,$$

with the following functions for $-1 \leq \xi, \eta \leq 1$

$$\begin{aligned} \hat{N}_1(\xi) &= -\frac{\xi(1-\xi)}{2}, & \hat{M}_1(\eta) &= -\frac{\eta(1-\eta)}{2}, \\ \hat{N}_2(\xi) &= \frac{\xi(1+\xi)}{2}, & \hat{M}_2(\eta) &= \frac{\eta(1+\eta)}{2}, \\ \hat{N}_3(\xi) &= (1-\xi^2), & \hat{M}_3(\eta) &= (1-\eta^2). \end{aligned}$$

Biquadratic basisfunctions can be constructed like in the bilinear case as union of reference shape functions. In this case three different types of reference basisfunctions for a vertex, an edge and the center, respectively are distinguished. They are shown in figure 1.3. All plots showing the shape of considered basisfunctions were constructed using Matlab R2010b.

Higher-order shape functions as bicubic ones are constructed in a similar way.



(a) Node function

(b) Edge function

(c) Bubble function

Figure 1.3: Biquadratic basisfunctions on reference element

1.5 Delaunay Triangulation and Voronoi Diagram

The mesh generation is of paramount importance in the finite element method. In particular the choice of a reasonable, which often means unstructured triangulation is a crucial step to secure convergence. Unstructured meshes could be necessary on irregularly shaped domains, which resist structured discretizations or in order to deal with certain geometrical singularities [30, p. 6]. Such meshes are often computed by Delaunay triangulations of point sets. Given a set of points fulfilling some assumptions, the Delaunay triangulation is the unique triangulation which maximizes the minimum angle in the triangulation [19, pp. 234]. As small interior angles lead to bad convergence in the finite element method, this property justifies the importance of Delaunay triangulations.

The following analysis is based on [10, chapter 7 and 9] and [19, chapter 5]. It should be remarked, that Delaunay triangulations are the dual graphs of Voronoi diagrams, which is why we are going to discuss these first.

Let $S = \{p, q, r, \ldots\} \subset \mathbb{R}^2$ be a set of n distinct points in a plane. As a measure for the influence between points $p = (p_1, p_2)$ and $x = (x_1, x_2)$, we use the *Euclidean distance*

$$\|px\| := \sqrt{|p_1 - x_1|^2 + |p_2 - x_2|^2}.$$

For $p \in S$ we define the *Voronoi region* as

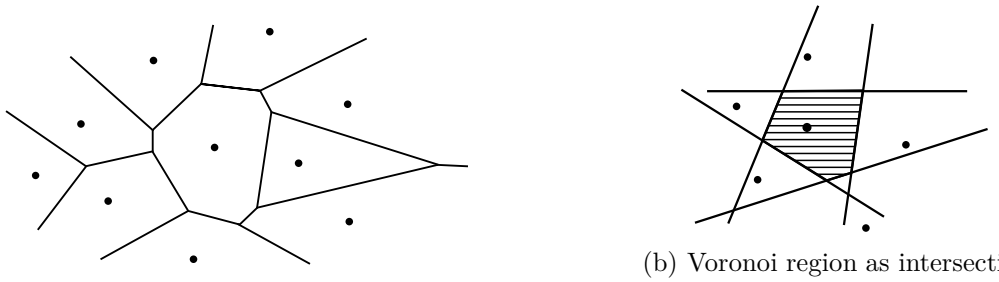
$$VR(p, S) := \{x \in \mathbb{R}^2 \mid \|px\| \leq \|qx\|,\ \forall q \in S\}.$$

Furthermore we define the *Voronoi diagram* $V(S)$ as the decomposition of $\mathbb{R}^2$ induced by $VR(p, S)$, $p \in S$ (see figure 1.4a). The *bisector* $B(p, q)$ of $p, q \in S$, is the set $B(p, q) := \{x \in \mathbb{R}^2 \mid \|px\| = \|qx\|\}$. It consists of all points located on the perpendicular line to the line segment $\overline{pq}$, which intersects with $\overline{pq}$ at its midpoint. $B(p, q)$ splits $\mathbb{R}^2$ into two open half-planes

$$D(p, q) := \{x \in \mathbb{R}^2 \mid \|px\| < \|qx\|\} \text{ and}$$
$$D(q, p) := \{x \in \mathbb{R}^2 \mid \|px\| > \|qx\|\}.$$

Obviously there is $p \in D(p, q)$ and $q \in D(q, p)$. Furthermore we deduce the following formula depicted in figure 1.4b

$$VR(p, S) = \bigcap_{q \in S \setminus \{p\}} D(p, q).$$



(a) Voronoi diagram

(b) Voronoi region as intersection of half planes

Figure 1.4: Construction of Voronoi diagrams

Next we want to look at the boundaries of Voronoi regions. First, consider the edge e_{pq} between $VR(p, S)$ and $VR(q, S)$:

$$\forall x \in e_{pq},\ \|px\| = \|qx\| < \|rx\|,\ \forall r \in S \setminus \{p, q\}$$

Second, consider the vertex v_{pqr}, which is the point where $VR(p, S)$, $VR(q, S)$ and $VR(r, S)$ intersect:

$$\|pv_{pqr}\| = \|qv_{pqr}\| = \|rv_{pqr}\| < \|sv_{pqr}\|,\ \forall s \in S \setminus \{p, q, r\}$$

Moreover, v_{pqr} is also the circumcenter of the circle through p, q and r, and the circle does not contain any other points of S.

The structure of the Voronoi diagram depends on the position of the points $p \in S$ to each other. Two degenerate cases should be avoided (see figure 1.5):

1. For a set of n collinear points, the Voronoi diagram consists of $n - 1$ parallel lines.
2. For a set of co-circular points, the Voronoi diagram looks like a "pie slice".

(a) Parallel points (b) Co-circular points

Figure 1.5: Degenerate cases of Voronoi diagrams

This motivates two general assumptions:

1. No three points are collinear.
2. No four points are co-circular.

This implies, that no Voronoi diagram has parallel edges and that the degree of each vertex in it is three.

Now we are able to define the *Delaunay triangulation* $DT(S)$ as the dual graph of the Voronoi diagram $V(S)$. Hence, $DT(S)$ has a node for every Voronoi cell (i.e. for every point $p \in S$) and two points p and q are connected by a line $\overline{pq}$, if their Voronoi regions $VR(p, S)$ and $VR(q, S)$ share a common edge in $V(S)$. Note, that the degree of each vertex in $V(S)$ is three, which implies that $DT(S)$ is a triangulation. An example is shown in figure 1.6.

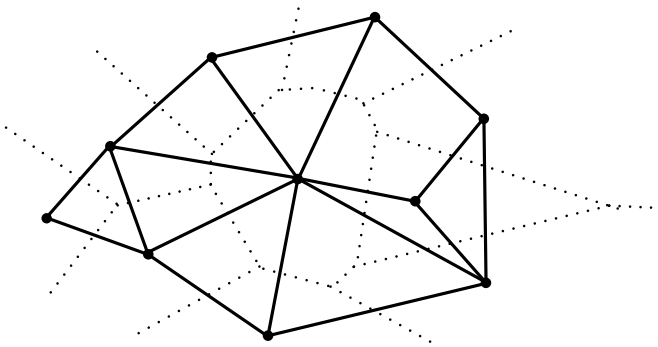


Figure 1.6: Delaunay triangulation $DT(S)$ as the dual grapf of Voronoi diagram $V(S)$

At the end, we want to give a summary of some basic properties of Delaunay triangulations. With C° we denote the interior of the set C.

- $\overline{pq} \in DT(S) \Leftrightarrow \partial V(p, S) \cap \partial V(q, S)$ is an edge in $V(S)$
- $\Delta pqr \in DT(S) \Leftrightarrow v_{pqr}$ is a vertex in $V(S)$.
- v_{pqr} is the circumcenter of Δpqr
- The circumcircle of each triangle in $DT(S)$ is empty, i.e. no point $p \in S$ lies inside.
- $\overline{pq} \in DT(S) \Leftrightarrow$ There exists a circle C_{pq} passing through p and q, such that $C_{pq}^\circ \cap S = \emptyset$.

Especially the last two properties are very important and can be used to characterize Delaunay triangulations [19, p. 235].

1.6 Remarks on Programming

The implementation of scientific software can be split into two phases. In the first phase the (mathematical) theory is converted into an algorithm which is possibly designed in pseudo code, where no attention is paid to syntax and semantic. The general properties of an algorithm, like complexity and parallelism are analyzed and if possible enhanced. All algorithms presented in the following chapters, are algorithms which are the result of this phase.

The second phase deals with the translation of these pieces of pseudo code into actual code of a programming language like Fortran 95. At this point issues like data structures, matrix-vector or matrix-matrix multiplications, organization of subprograms and many more have to be considered. Despite being mostly hidden behind fancy graphs and impressive tables, this work represents the foundation for all numerical experiments and is of very high importance for a successful implementation of the derived algorithms. In the following sections we want to mention some of the biggest and most frequently recurring challenges of this phase, with the aim of raising awareness for this crucial, but often underestimated phase of scientific work.

1.6.1 General Remarks

1. Set operations

   The considered algorithms will often rely on set operations. Hereby we understand, for example, the check if a given set is a subset or equal to another set. As the main data structure is a sparse matrix, and most sets are constructed from one individual row, they are generally very small. Hence the sets are saved in a vector. These operations can be performed in linear time, if the sets are stored in sorted order. As mentioned before, most sets are created from the sparse system matrix. Therefore the easiest and most efficient way to create sorted sets, is to ensure that the rows of the sparse matrix (stored in the CSR format) are sorted according to their column offsets (cf. example 1.1) and that this order is maintained when creating new sets.

2. Precompute constants

   Constants which are used more often should be precomputed and stored in a separate variable. This is particularly efficient within loops.

3. Usage of flag arrays with sparse data structures in nested loops

   Consider two nested loops over a large amount of elements. Some problems require a kind of array, like a flag array, which marks certain elements which were affected during each iteration of the inner loop. This could be necessary for postprocessing steps on these elements after each individual iteration of the outer loop. If this array needs to be set back to its default values (mostly back to zero) before the start of the next iteration, it would be very inefficient to perform this reset on all elements, as in general only a minor amount was changed. A possible solution is to use additional memory in a vector of fixed size, which remembers the positions of the saved parameters. If the vector is big enough, it will be sufficient to reset the changed values only. Otherwise the amount of allocated memory should be increased for the next iteration, since it is likely that the same case appears repeatedly. For this particular iteration the whole vector has to be

reset as the collected information is insufficient. The following table 1.2 illustrates this solution.

Note, that the algorithms discussed in the following chapters will not consider this problem. For brevity and simplicity we will always choose the slower variant in the presented pseudo codes, but the implementations will use the before discussed optimization.

Slow

```
1: for i ← 0 to n do
2:     I ← 0
3:     for j ← 0 to n do
4:         If condition then I(j) ← 1
5:     end for
6: end for
```

Efficient

```
1: I ← 0                                   ▷ Initialize
2: Allocate v(0, ..., M − 1)    ▷ M ≪ n
3: for i ← 0 to n do
4:     c = 0          ▷ Current position in v
5:     for j ← 0 to n do
6:         if condition then
7:             I(j) ← 1
8:             If c < M then v(c) ← j
9:             c = c + 1
10:        end if
11:    end for
12:    if c ≤ M then
13:        for j ← 0 to c do
14:            I(v(j)) ← 0
15:        end for
16:    else
17:        I ← 0
18:        Enlarge to v(0, ..., c − 1)
19:    end if
20: end for
```

Table 1.2: Efficient usage of a flag array with sparse data structures in nested loops

1.6.2 Parallel Programming

1. Frequently changing arrays

   If an element of an array changes frequently during execution, a temporary variable should be used to store intermediate results. After execution the final result is stored in the array. This is illustrated in table 1.3.

2. Calculating the size of a sparse data structure before constructing it

   Instead of allocating too much memory in advance, it is often advantageous to use an additional loop in order to determine the actual size. This is mostly done by counting how many elements of each row fulfill a certain condition. By doing so no memory is wasted and the surplus effort in determining the size can be compensated by the ability to execute the computation of the actual values in parallel.

   E.g.: Determine the strongly coupled neighbors for a given matrix in section 2.4.1.

Slow	**Efficient**
1: $x(k) \leftarrow 0$	1: $tmp \leftarrow 0$
2: **for** $i \leftarrow 0$ **to** n **do**	2: **for** $i \leftarrow 0$ **to** n **do**
3: $\quad x(k) \leftarrow x(k) + 1$	3: $\quad tmp \leftarrow tmp + 1$
4: **end for**	4: **end for**
	5: $x(k) \leftarrow tmp$

Table 1.3: Efficient usage of frequently changing arrays

3. Decouple iterations

   Parallel programming issues mainly address the parallel execution of expensive loops. In order to execute the iterations in parallel it is essential that every iteration is independent of any other, i.e. no iteration depends on the result of a previous one.

   Some algorithms are inherently serial, like the classical Gauss-Seidel method or algorithm 2.9. Though, in general it is possible to adapt these algorithms and develop parallel variants. But one has to consider the payoff between additional costs and gain in computation time. Often the preparatory costs are overwhelming the possibly won advantage (compare algorithm 2.9 with parallel remarks on page 35).

   However, many algorithms can be parallelized by some preparatory computations. In the current implementations it is often necessary to construct a sparse matrix saved in the CSR format (cf. chapter 1.2). These matrices are saved as vectors in a row-by-row format. Unfortunately it is often not known beforehand, how many entries each row will have and as the rows are stored one after each other, the location of an element in the next row depends on the length of the previous one. This dependency can easily be overcome, by counting the number of elements per row first. This structural information decouples the rows and enables a parallel computation. Examples are algorithm 2.8, where already computed information reveals the structure of the matrix and algorithm 3.2, where the number of entries per row is computed first.

Chapter 2

AMG Concepts

The development of multilevel methods basically followed two approaches: geometric and algebraic. In geometric multigrid the geometry of the underlying problem is used to define the various components. In contrast algebraic multigrid uses only the information in the linear (sparse) algebraic system of equations

$$Au = f \qquad \text{or} \qquad \sum_{j=1}^{n} a_{ij} u_j = f_i, \text{ for } i = 1, 2, \ldots, n. \tag{2.1}$$

Therefore the advantage of AMG is that it can be directly applied to more complicated domains and structured as well as unstructured grids. It can be shown that it provides very *robust* solution methods, i.e. it solves a large class of problems.

Every multigrid method consists of the following components which will be introduced in this chapter:

1. Smoother
2. Coarse Grid Hierarchy
3. Prolongation/Restriction
4. Coarse Grid Equation

The main difference between geometric and algebraic multigrid is the construction of the coarse grid hierarchy. In the geometric method, the coarse grids are predefined and chosen according to the geometric nature of the problem. Afterwards the smoother is adapted to the defined grids, in order to achieve optimal error reduction. In contrary, in the algebraic case the smoother is fixed beforehand and the coarse grids are adapted to the problem. To do so the geometric information is not available in the algebraic case. Therefore the hierarchy is created as part of the method in an additional *setup phase*. There are many approaches of how to handle this problem, but all of them have in common that the linear system matrix is the only source of information. This is illustrated in the figure 2.1 (see [33, p. 416]).

As the physical grid points are unknown the indices $\{1, 2, \ldots, n\}$ are used instead, as every grid point or node corresponds to one index. In the following the terms grid point or node always refer to the index and not to the physical point.

On these nodes, the connections within the grid are determined by the *undirected adjacency graph* of the matrix A. Every vertex of the graph is associated with a grid point and there

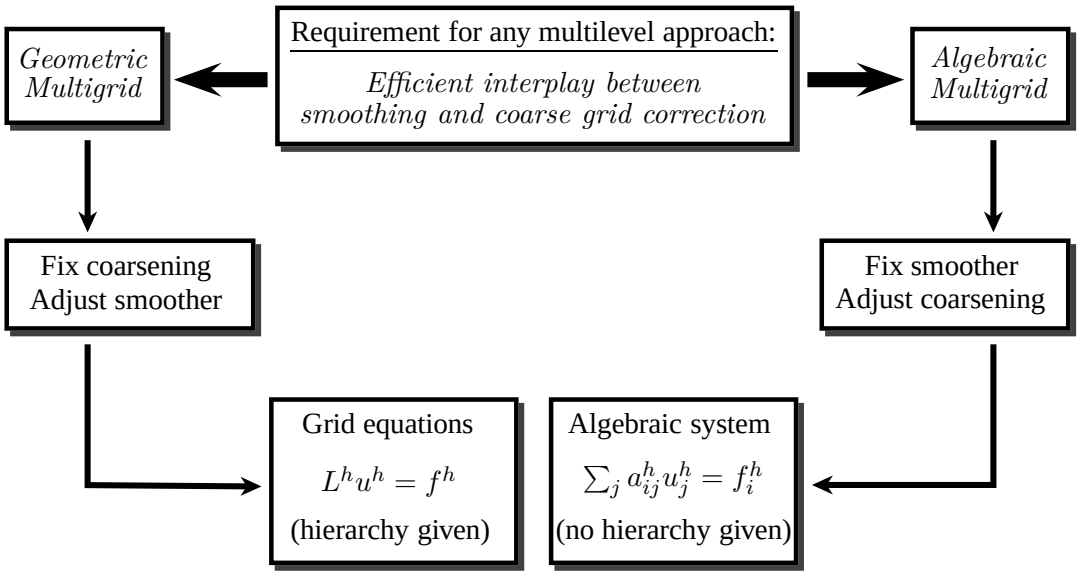


Figure 2.1: Geometric versus Algebraic Multigrid

exists an edge between the nodes i and j if either $a_{ij} \neq 0$ or $a_{ji} \neq 0$. The graph represents the connections in the grid and describes the interplay of the different nodes. A simple example of this relationship is shown in figure 2.2.

$$A = \begin{pmatrix} X & & & X & X & \\ & X & X & X & & \\ & X & X & X & & X \\ X & X & X & X & X & \\ X & & & X & X & X \\ & & X & & X & X \end{pmatrix}$$

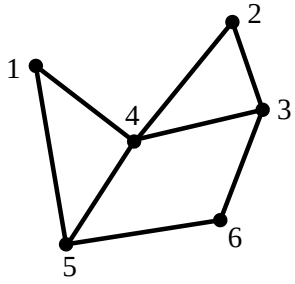


Figure 2.2: The nonzero structure of A and its adjacency graph [5, p. 138]

A very important concept in the development of multigrid methods is the concept of *smoothness*. We distinguish between geometric and algebraic smoothness. A function is geometrically or physically smooth if it has a low spatial frequency. In geometric multigrid it is assumed that the relaxation scheme smooths the error and that the smooth functions of a certain level are represented exactly on the coarser grid. Intergrid transfer operators are chosen in such a way, that smooth functions are transfered accurately between grids.

As in AMG the relaxation scheme is chosen first, it allows us to determine the nature of the smooth error. However, we have no access to the physical grid and for this reason the necessity arises to define the sense of smoothness algebraically. With the help of this sense of smoothness the coarse grids are selected and the intergrid transfer operators are defined. Finally a coarse grid version of the operator A is selected. The goal within this procedure stays the same as in the geometric case: Eliminate the error components in the range of the interpolation operator (see [5, 33, 45]).

2.1 Concept and Cost of AMG

This section gives an overview of the AMG concept. We follow the structure of [45], a detailed explanation can be found in [27] or any similar introduction to AMG.

Relaxation schemes like weighted Jacobi and Gauss-Seidel show the interesting property of fast initial convergence which stalls abruptly after a small number of iterations. The central idea in all multigrid methods is to use the fast initial convergence of these relaxation schemes. When performing a Fourier analysis in certain model problems it can be shown, that these relaxation schemes are very efficient on high frequency components of the error, but very poor on low frequency parts [2, 5]. Therefore the core idea in any multigrid method is that a smooth error e that is not eliminated by relaxation must be removed by coarse grid correction. This is done by solving the residual equation $Ae = r$ on a coarser grid, then interpolating the error back to the fine grid and using it to correct the fine-grid approximation by $u \leftarrow u + e$.

This procedure is best described in an algorithm. We use subscripts to indicate the level number, where 0 indicates the finest level (i.e. $A_0 = A$ and $\Omega_0 = \Omega$). The current level is denoted by k and the number of unknowns on one level by n_k. The following components are needed for AMG:

1. **Grids:** $\Omega_0 \supset \Omega_1 \supset \ldots \supset \Omega_M$ with the subsets:
   Set of coarse points or C-points C_k, $k = 0, 1, \ldots, M-1$
   Set of fine points or F-points F_k, $k = 0, 1, \ldots, M-1$
2. **Grid Operators:** $A_0, A_1, \ldots, A_M$
3. **Grid transfer operators:**
   Interpolation: $P_k : \Omega_{k+1} \to \Omega_k$, $k = 0, 1, \ldots, M-1$
   Restriction: $R_k : \Omega_k \to \Omega_{k+1}$, $k = 0, 1, \ldots, M-1$
4. **Smoothers:** S_k, $k = 0, 1, \ldots, M-1$

Note, that coarse grid selection and interpolation/restriction must go hand in hand and affect each other in many ways. In the following we will use the terms interpolation and prolongation synonymously as the operation performed by P_k, $k = 0, \ldots, M$.

In the first step, the *setup phase*, these components are constructed in algorithm 2.1. Clearly, if the same problem has to be solved more often, the setup phase needs to be performed only once.

Algorithm 2.1 AMG Setup Phase

1: $k \leftarrow 0$
2: **while** Ω_k is big **do**
3: Partition Ω_k into two disjoint sets C_k and F_k
4: $\Omega_{k+1} \leftarrow C_k$
5: Define interpolation P_k
6: Define restriction R_k (often $R_k \leftarrow (P_k)^\top$)
7: Setup A_{k+1} (often $A_{k+1} \leftarrow R_k A_k P_k$)
8: Setup S_k if necessary
9: **end while**
10: $M \leftarrow k$

After the setup phase is completed, the *solve phase* is performed recursively in algorithm 2.2. It represents a $V(m_1, m_2)$-cycle (i.e. a V-cycle with m_1 presmoothing and m_2 postsmoothing steps) which, along with other more complex variants such as a W-cycle, is described in detail in [5, pp. 40].

Algorithm 2.2 AMG Solve Phase

1: **procedure** MGV($A_k, R_k, P_k, S_k, u_k, f_k$)
2: **if** $k == M$ **then**
3: Solve $A_M u_M = f_M$ with a direct solver ▷ Coarsest level
4: **else**
5: **Presmoothing:** Apply smoother S_k m_1 times to $A_k u_k = f_k$
6: **Coarse grid correction:**
7: $r_k \leftarrow f_k - A_k u_k$ ▷ Residual
8: $r_{k+1} \leftarrow R_k r_k$ ▷ Restrict residual
9: Call MGV($A_{k+1}, R_{k+1}, P_{k+1}, S_{k+1}$,$\boldsymbol{e_{k+1}, r_{k+1}}$)
10: $e_k \leftarrow P_k e_{k+1}$ ▷ Interpolate
11: $u_k \leftarrow u_k + e_k$ ▷ Correct solution
12: **Postsmoothing:** Apply smoother S_k m_2 times to $A_k u_k = f_k$
13: **end if**
14: **end procedure**

The quality of AMG depends basically on two measures, which are both very important, but depending on the priorities of the user, one might be of more importance than the other one:

1. **Convergence factor:** gives an indication on how fast the method converges
2. **Complexity:** affects the number of operations per iteration and the memory usage

The complexity can be understood as the cost of AMG. The geometric case has regular and predictable costs, in terms of storage and floating-point operation count. By contrast, there are no predictive cost analyses for AMG. This is due to the fact, that we do not know in advance how many levels will be created and thus we do not know the ratio of coarse to fine grid points on each level.

We distinguish between two types of complexities [5, p. 154] shown in table 2.1:

1. **Grid complexity:**	the total number of grid points, on all grids, divided by the number of grid points on the finest grid
2. **Operator complexity:**	the total number of nonzero entries, in all matrices A_k, divided by the number of nonzero entries in the fine grid operator $A = A_0$

Table 2.1: Complexities

The grid complexity gives an accurate measure of the storage required for the right sides and approximation vectors and can be directly compared to geometric multigrid.

The operator complexity indicates the total storage space required by the operators A_k over all grids. Additionally it has another use. Similarly to the geometric case, the work in the solve phase of AMG is dominated by the relaxation and residual computations, which are directly

proportional to the number of nonzero entries in the operator. Hence, the work of a V-cycle turns out to be essentially proportional to the operator complexity. This proportionality is not perfect, but operator complexity is generally considered to be a good indication of the expense of one AMG V-cycle.

2.2 Smoother

The idea of multigrid started with a detailed analysis of classical iterative methods as solvers for the linear system of equations

$$Au = f. \tag{2.2}$$

In the following discussion we denote the exact solution as u and an approximation of it as v. With this notation the error e of an approximation is defined as $e := u - v$. Note that if the solution u is not known, this applies also for the error e. Furthermore we will need the following definition.

Definition 2.1: The *residual* r corresponding to an approximation v of the solution u of the linear system of equations $Au = f$ is defined as

$$r := f - Av. \qquad \square$$

If the solution of the linear system is unique, it can be shown, that $r = 0$ if and only if $e = 0$. Moreover there is another important connection between residual and error.

Lemma 2.2: For the residual r and the error e of an approximation v of (2.2), it holds that

$$Ae = r. \tag{2.3}$$

Proof: A substitution gives $r = f - Av = Au - Av = A(u - v) = Ae$. $\square$

This equation is often called *residual equation* and forms the basis of most iterative solution methods. If the approximation v is known, the following algorithm 2.3 presents the idea of improving v iteratively.

Algorithm 2.3 One step of a general iterative solution method

1: **Compute residual:** $r = f - Av$
2: **Compute approximation to error:** $\hat{e}$ approximately solves $Ae = r$
3: **Correct approximation to solution:** $v = v + \hat{e}$

In mathematical notation, the above algorithm 2.3 can be represented as

$$v_{k+1} = v_k + M^{-1}\left(f - Av_k\right),$$

where v_k denotes the k-th approximation of the exact solution u. Examples for iterative methods are the Jacobi or Gauß-Seidel scheme, in which the matrix M is assigned with the

values

$$\begin{aligned} M &= D && \text{(Jacobi)}, \\ M &= D + L && \text{(Gauß-Seidel)}, \end{aligned}$$

where $A = D + L + U$ and L (D) is a strict lower (upper) triangular matrix.

These methods show in general very fast convergence in the first iterations, but slow down dramatically afterwards. This is due to the fact, that these methods reduce only high frequency error components efficiently. Hence, instead of reducing the error $e_k = u - v_k$, they only *smooth* the error. This explains the origin of their equivalent name *smoother*, which is particularly used in the context of multigrid methods (see [41, pp. 13]). A more technical discussion can be found in [14, pp. 49].

These basis methods can be extended by introducing the weight parameter $\omega \in [0, 2]$. With this parameter we introduce the following generalizations of the above methods:

$$v_{k+1} = v_k + \omega \hat{M}^{-1} (f - A v_k)$$

The iteration method changes only in the case of the Gauss-Seidel method and we get the weighted Jacobi and the SOR (successive over-relaxation) schemes.

$$\begin{aligned} \hat{M} &= D && \text{(Weighted Jacobi)} \\ \hat{M} &= D + \omega L && \text{(SOR)} \end{aligned}$$

Note that for $\omega = 1$ we get the classical methods, however a typical choice is $\omega = 4/3$ (cf. [28, pp. 95]).

Remark 2.3: The representation formulas of the iteration steps over the residuals allows a compact representation and simplifies the notation, but for an explicit implementation there exist more efficient formulas. These are more technical and conceal the core ideas, but bring advantageous in computation time and memory usage. The interested reader is referred to [28, chapter 4] or [14]. ■

2.3 Algebraic Smoothness

As the concept of algebraic smoothness is very important to AMG it is worthwhile investigating it in more detail. Here we follow the ideas of [5,27,41]. For the theoretical discussion the system matrix A is always assumed to be symmetric and positive definite. In the following the norms of chapter 1.3 are needed.

In geometric multigrid the most important property of a smooth error is, that it is not effectively reduced by relaxation. This motivates a loose definition of an algebraically smooth error as an error slow to converge with respect to a smoother S. In the following we are going to consider the weighted point Jacobi method which can be expressed as

$$v_{k+1} \leftarrow v_k + \omega D^{-1}(f - A v_k) \tag{2.4}$$

where D is the diagonal of A. Here v_k denotes the k-th approximation to the exact solution u. We define the error as $e_k := u - v_k$ and look at the error propagation in this scheme. Note, that the exact solution u is a fixed point of the above iteration if ω is chosen appropriately, i.e. $u = u + \omega D^{-1}(f - Au)$. Subtracting the above equation (2.4) from this one we get a relation for the error propagation:

$$\begin{aligned} u - v_{k+1} &= u - v_k - \omega D^{-1} A(u - v_k) \\ e_{k+1} &= u - v_k - \omega D^{-1} A e_k \\ &= (I - \omega D^{-1} A) e_k \\ &= S e_k \end{aligned}$$

It can be shown, that the weighted Jacobi relaxation (as all smoothers), has the property that after making great progress towards convergence, it stalls and little improvement is made with successive iterations (see [28, chapter 4] or [24, pp. 9]). At this point we define the error to be *algebraically smooth*, i.e.:

$$\|Se_k\|_{\mathcal{H}^1} \approx \|e_k\|_{\mathcal{H}^1}.$$

Depending on A, an algebraically smooth error may well be highly oscillatory in the geometrical sense.

Ruge and Stüben have shown, that for typical relaxation schemes the inequality

$$\|Se_k\|_{\mathcal{H}^1}^2 \leq \|e_k\|_{\mathcal{H}^1}^2 - \alpha \|e_k\|_{\mathcal{H}^2}^2 \tag{2.5}$$

holds with $\alpha > 0$ (e.g. $\alpha = 0.25$) [27, pp. 82]. Therefore an algebraically smooth error has to satisfy $\|e_k\|_{\mathcal{H}^2} \ll \|e_k\|_{\mathcal{H}^1}$. We are able to us this result for two important implications.

First, writing this expression in components gives

$$\sum_i \frac{r_i^2}{a_{ii}} \ll \sum_i r_i e_i.$$

Fixing i with the motivation of considering the result on average yields

$$|r_i| \ll a_{ii} |e_i|.$$

This implies, that an algebraically smooth error can be characterized by *relatively small residuals*. Despite our analysis here for the weighted Jacobi method, the Gauss-Seidel relaxation is more commonly used for AMG. However, a similar analysis also leads to the above conclusion [5, p. 139].

Second, remark that the matrix D has full rank and all entries are greater than zero, because A is a positive definite matrix. Hence we are able to take the square root of every entry and to invert the matrix. As this matrix is trivially symmetric it is also self-adjoint. If it is clear from the context we will write e instead of e_k for easier readability. So this reasoning and the application of the Cauchy-Schwarz inequality shows

$$\begin{aligned} \|e\|_{\mathcal{H}^1}^2 &= (Ae, e)_2 = (D^{-1/2} Ae, D^{1/2} e)_2 \\ &\leq \|D^{-1/2} Ae\|_2 \|D^{1/2} e\|_2 \\ &= (D^{-1} Ae, Ae)_2 (De, e)_2 = \|e\|_{\mathcal{H}^2} \|e\|_{\mathcal{H}^0}. \end{aligned}$$

Consequently $\|e\|_{\mathcal{H}^2} \ll \|e\|_{\mathcal{H}^1}$ implies $\|e\|_{\mathcal{H}^1} \ll \|e\|_{\mathcal{H}^0}$. Writing the last relation more explicitly and using the symmetry of A again we get

$$\begin{aligned}(Ae,e)_2 &= \frac{1}{2}\sum_{i,j} -a_{ij}(e_i-e_j)^2 + \frac{1}{2}\sum_{i,j} a_{ij}e_i^2 + \frac{1}{2}\sum_{i,j} a_{ij}e_j^2 \\ &\overset{\text{sym.}}{=} \frac{1}{2}\sum_{i,j} -a_{ij}(e_i-e_j)^2 + \sum_i \left(\sum_j a_{ij}\right) e_i^2 \ll \sum_i a_{ii}e_i^2 \quad (= (De,e)_2)\,.\end{aligned}$$

If we consider the important case of $\sum_{j\neq i}|a_{ij}| \approx a_{ii}$ or $\sum_i a_{ij} \approx 0$, this yields, that *at least on average*, the algebraically smooth error e satisfies for each i

$$\begin{aligned}\frac{1}{2}\sum_{j\neq i} -a_{ij}(e_i-e_j)^2 &\ll a_{ii}e_i^2, \\ \sum_{j\neq i} \frac{|a_{ij}|}{a_{ii}} \frac{(e_i-e_j)^2}{e_i^2} &\ll 2.\end{aligned}$$

This means, that an algebraically smooth error varies generally slowly in the direction of strong connections, i.e. from e_i to e_j if $\frac{|a_{ij}|}{a_{ii}}$ is relatively large.

2.4 Coarse Grid Hierarchy

There are basically two ways of choosing a coarse grid. The first approach splits all points into two groups: coarse points (C-points) and fine points (F-points). The C-points form the next coarser level, while the F-points will be interpolated by these. This is the classical approach which was mainly developed by Ruge and Stüben [27,33]. The second approach is referred to as coarsening by (smoothed) aggregation or agglomeration and was developed by Vaněk, Mandel and Brezina [34,35,37]. It accumulates aggregates which represent the coarse points of the next coarser level.

Both have in common that they use a concept of strength to measure the dependencies in the undirected adjacency graph. By this measure, a neighborhood for every point i is defined, which is used to construct the coarse level in several steps.

We start our discussion with the approach of smoothed aggregation, because most experiments of chapter 5 use this one as basis AMG algorithm. Compared to the classical RS coarsening, it possesses a faster setup phase, as the coarsening is more aggressive, which results in sparser coarse system matrices, but slower convergence, as fine nodes are interpolated from less coarse nodes.

2.4.1 Smoothed Aggregation

Following [35] we describe in this section the construction of a system of aggregates $\{\mathcal{A}_i^k\}_{i=1}^{n_{k+1}}$ of the matrix A_k. The aggregates form a disjoint partitioning of the domain Ω_k, i.e. $\bigcup_i \mathcal{A}_i^k = \Omega_k$. In order to measure the dependencies of different nodes on each other, we give the

following definition of strength.

Definition 2.4: A point i is *strongly coupled* to a point j if

$$|a_{ij}| \geq \theta \sqrt{|a_{ii} a_{jj}|}.$$

The *strongly coupled neighborhood* of the node i on level k is defined as

$$N_i^k(\theta) = \{i \in \Omega_k \mid |a_{ij}| \geq \theta \sqrt{|a_{ii} a_{jj}|}\}$$

where $\theta \geq 0$. □

Therefore an aggregate $\mathcal{A}_i^k$ is defined by a root point $\hat{i}$ and its neighborhood $N_{\hat{i}}^k(\theta)$. Now the basic aggregation procedure consists of the following three phases. First a root point, that is not adjacent to any already existing aggregate is picked. The aggregate is defined by the root point and all its neighbors in the above sense. This procedure is repeated until all unaggregated points are adjacent to an aggregate. In the second phase all remaining unaggregated points are integrated into existing aggregates, if possible. Otherwise, in the third phase new aggregates are created for all points which are still unaggregated, which is done in the same manner as in phase one. Algorithm 2.4 describes this procedure.

Experiments have shown that step 3 is typically not performed, because $R = \emptyset$ after step 2.

In the following we discuss the individual steps in more detail and point out where parallelization can be used to enhance the performance. As a shorthand we refer to the strongly coupled neighbors as neighbors only.

Before the algorithm starts the neighborhoods $N_i^k(\theta)$ must be constructed. In a first step the diagonal elements are extracted from A_k. Then above condition is checked for every element a_{ij}. As the matrix A_k is a sparse matrix, the computed neighbors are best stored in a sparse format too. Therefore we use an adapted CSR-format (cf. section 1.2) without the vector AA to store real values. Hence the neighbors of node i are $JA(l)$, $l = IA(i), \ldots, IA(i+1) - 1$. Due to the sparse structure of the matrix, this needs only $\mathcal{O}(n_k)$ operations.

As a second data structure we introduce the vector M_k of length n_k. It maps every node i to its aggregate, which is identified by a unique number (in algorithm 2.4 j counts the total number of aggregates). The first aggregate starts with 1, we use 0 to mark unaggregated entries and $n_k + 1$ to mark isolated points. A node i is an isolated point, if the number of elements in row i of matrix A_k is equal one.

$$M_k(i) := \begin{cases} j, & i \in \mathcal{A}_j^k, \\ 0, & i \in R, \\ n_k + 1, & i \text{ isolated point.} \end{cases} \tag{2.6}$$

Parallel 2.5: With the right data structure, most parts can be performed in parallel. First every row is scanned for the diagonal entries which are stored in vector of length n_k. Second the number of neighbors in each row is computed using this information. Finally, the array of neighbors is initialized in parallel (see point 2 in section 1.6.2). ■

Algorithm 2.4 Aggregation

1: **Initialization:**
2: $R \leftarrow \{i \in \Omega_k \mid N_i^k(0) \neq \{i\}\}$
3: $j \leftarrow 0$
4: **Step 1 - Startup aggregation:**
5: **for all** $i \in R$ **do**
6: **if** $N_i^k(\theta) \subset R$ **then**
7: $j \leftarrow j + 1$ ▷ New Aggregate
8: $\mathcal{A}_j^k \leftarrow N_i^k(\theta)$
9: $R \leftarrow R \setminus \mathcal{A}_j^k$
10: **end if**
11: **end for**
12: **Step 2 - Enlarging the decomposition sets:**
13: **for all** $l \leq j$ **do**
14: $\bar{\mathcal{A}}_l^k \leftarrow \mathcal{A}_l^k$ ▷ Copy
15: **end for**
16: **for all** $i \in R$ **do**
17: **for all** $z \leq j$ **do**
18: **if** $N_i^k(\theta) \cap \bar{\mathcal{A}}_l^k \neq \emptyset$ **then**
19: $\mathcal{A}_l^k \leftarrow \mathcal{A}_l^k \cup \{i\}$ ▷ Add to existing aggregate
20: $R \leftarrow R \setminus \{i\}$
21: break
22: **end if**
23: **end for**
24: **end for**
25: **Step 3 - Handling remnants:**
26: **for all** $i \in R$ **do**
27: $j \leftarrow j + 1$ ▷ New Aggregate
28: $\mathcal{A}_j^k \leftarrow N_i^k(\theta) \cap R$
29: $R \leftarrow R \setminus \mathcal{A}_j^k$
30: **end for**

In the following we are going to discuss each step individually:

Init.: The set R is represented by all entries equal to 0 in M_k. It is created during the setup of the neighborhoods.
Complexity: $\mathcal{O}(n_k)$

Step 1: If an element i has not been aggregated yet (i.e. $M_k(i) == 0$), we check if all neighbors are still members of R (i.e. $M_k(j) == 0$, $\forall j \in N_i^k(\theta)$).
Complexity: $\mathcal{O}(n_k)$ due to sparsity of A_k

Step 2: As the vector M_k uses only positive integers to indicate aggregates, we can use the negative numbers of aggregates to save newly added elements during this phase. By this strategy memory and computation time are saved because the aggregates $\mathcal{A}_l^k$, $l = 1, \ldots, j$ do not need to be copied. At the end we take the absolute value of M_k to merge newly added elements with existing sets.
Complexity: $\mathcal{O}(n_k)$

Step 3: Similar to step 1 but often not even needed because $R = \emptyset$ after step 2.
Complexity: $\mathcal{O}(n_k)$

Parallel 2.6: Most steps can be performed only sequentially as the aggregation depends on the previously aggregated elements. Only step 2 can be performed in parallel, as it does not create new aggregates and the sets are not changed during the iterations. ■

The disadvantage of storing the aggregates in an array is that if only the elements of one specific aggregate are needed, all n_k elements must be checked. To increase the performance it is advisable to group elements at the end according to their aggregate number.

2.4.2 Classical RS-Coarsening

The classical coarsening strategy is usually called RS (Ruge-Stüben) approach. We are going to follow the explanation of [45, pp. 214], a more detailed description can be found in [33, pp. 472].

In contrast to the last approch of smoothed aggregation, the basic idea is to split all nodes into two disjoint sets $\Omega = C \cup F$. The C-points represent the degrees of freedom of the next coarser grid, while the F-points will be interpolated by the C-points. In order to obtain this splitting we need a concept of strength to identify matrix coefficients which have a stronger influence than others.

Definition 2.7: A point i *strongly depends* on a point j or the point j *strongly influences* a point i if

$$-a_{ij} \geq \theta \max_{l \neq i} \{-a_{il}\},$$

where $0 < \theta \leq 1$.

Denote by S_i the set of points that strongly influence the point i and by $S_i^\top$ the set of points that strongly depend on the point i, i.e.

$$S_i = \{j \in \Omega_k \mid -a_{ij} \geq \theta \max_{l \neq i} \{-a_{il}\}\},$$
$$S_i^\top = \{j \in \Omega_k \mid -a_{ji} \geq \theta \max_{l \neq j} \{-a_{jl}\}\}.$$ □

Remark 2.8: Definition 2.7 was originally motivated by the assumption that A is a symmetric *M-matrix*, i.e. a positive definite matrix with non-positive off-diagonal elements, but it can be applied to a more general class of matrices as well [45, p. 214]. ■

The initial selection of coarse grid points is guided by the following two heuristics [5, p. 146]:

(H1) For each F-point i, every point $j \in S_i$, that strongly influences i, j is either a C-point or it strongly depends on a C-point l that also strongly influences i, i.e. $l \in S_i$.

(H2) The set of C-points should be a maximal subset of all points with the property, that no C-point strongly depends on another C-point.

Heuristic (H1) ensures the quality of interpolation and heuristic (H2) should restrict the size of the coarse grids. In general it is impossible to fulfill both conditions, which is why we choose to enforce (H1), while (H2) serves as a guideline.

Now we are able to describe the coarsening algorithm, which proceeds in two passes. In the first pass C- and F-points are generated in such a way, that there are enough C-points for interpolation but as little as possible, to minimize the complexity. Algorithm 2.5 describes the first pass. In the initialization phase each point i is assigned a measure λ_i, which equals the number of points that are strongly influenced by i. Then the C/F-splitting starts. First a point with a maximal λ_i is selected as coarse point. Note that in general there will be several and each choice could generate a different coarse grid. Then all points that strongly depend on i become F-points (i.e. $S_i^\top$). For all points j that strongly influence these new F-points, their measure λ_j is increased by the number of new F-points that are strongly influenced by them. This increases their chances of becoming a C-point. The first pass finishes, when all points are either C- or F-points.

Remark 2.9: The measure λ_i can be interpreted as a "measure of importance", as it determines which node should become the next C-point. Instead of the chosen $\lambda_i = |S_i^\top|$ any reasonable formula can be applied. A different suggestion is $\lambda_i = |S_i^\top \cap R| + 2|S_i^\top \cap F|$, where R is like in algorithm 2.5, at any stage the current set of undecided nodes [33, p. 474]. ■

Algorithm 2.5 RS Coarsening, First pass

1: Associate every point i with a measure $\lambda_i \leftarrow |S_i^\top|$
2: $R \leftarrow \Omega$, $C \leftarrow \emptyset$, $F \leftarrow \emptyset$
3: **while** $R \neq \emptyset$ **do**
4: Select point $i \in R$ with $\lambda_i = \max_{j \neq i} \lambda_j$
5: $C \leftarrow C \cup \{i\}$ ▷ New coarse node
6: **for all** $j \in S_i^\top$ **do**
7: $F \leftarrow F \cup \{j\}$ ▷ New fine node
8: $\lambda_l \leftarrow \lambda_l + |S_l^\top|$ for all $l \in S_j$ ▷ Increase chances of becoming C-point
9: **end for**
10: $R \leftarrow R \setminus (S_i^\top \cup \{i\})$
11: **end while**

The second pass enforces condition (H1) by examining all strong F-F connections for common coarse neighbors and if heuristic (H1) is not satisfied new C-points will be added. It is described in detail in algorithm 2.6. Note that the choice of j as the new C-point in line 4 is arbitrary and could be replaced by node l. In that case leaving the inner loop in the next line is not allowed. The second pass is necessary, because strong connections between F-points without common C-point as neighbor worsen interpolation and therefore convergence.

Parallel 2.10: RS coarsening works very well for many applications, but it is in its nature an inherently sequential algorithm. Only the computation of the dependence sets S_i and $S_i^\top$ and the initialization phase of the first pass, can be performed in parallel. ■

The second pass often generates too many C-points and leads to large complexities and inefficiency. Hence, condition (H1) is modified to the following [45, p. 215]:

(H1′) Each F-point i needs to strongly depend on at least one C-point j.

Algorithm 2.6 RS Coarsening, Second pass

```
1: for all j ∈ F do
2:     for all l ∈ S_j ∩ F do                          ▷ Strong F-F-connection
3:         if S_j ∩ S_l ∩ C = ∅ then
4:             F ← F \ {j} and C ← C ∪ {j}             ▷ New coarse node
5:             exit                                    ▷ Now j ∉ F anymore
6:         end if
7:     end for
8: end for
```

Only the first pass of the RS coarsening fulfills this requirement. We call this method *weaker RS coarsening*. Yet another method in order to reduce complexities is *aggressive coarsening*, where condition (H1) is weakened to allow longer connections and not look at direct neighbors only [33, pp. 472].

The disadvantage of all previously mentioned methods is their sequential nature. Hence, generalizations for parallel architectures have been considered, like $CLJP$, RS0 and RS3 coarsening. As the focus of this work lies on an optimization of AMG for higher-order discretizations rather than on efficient classical AMG algorithms, these optimizations are not considered any further. The interested reader is referred to [33, chapter 7] or [15, p. 160-165].

2.5 Interpolation

Interpolation in AMG is used to transfer (prolongate) an error from a coarse level to the next finer level. The usual procedure is to use the same value as on the coarse level for C-points and represent F-points as a linear combination of C-points. This means for the i-th component of the interpolation operator applied on the error e

$$\left(P_H^h e\right)_i = \begin{cases} e_i, & \text{if } i \in C, \\ \sum_{j \in C} w_{ij} e_j, & \text{if } i \in F, \end{cases}$$

where w_{ij} denotes the interpolation weight, which has to be determined. The choice of interpolation weights depends on the chosen interpolation algorithm. In general we have to distinguish between interpolation for the smoothed aggregation approach and the classical methods. The smoothed aggregation algorithm builts a so called *tentative prolongator* of the above structure and applies a smoothing algorithm on it. By doing so the above structure does not hold any more for the final interpolation operator.

The basis for all interpolation schemes is an *algebraically smooth error*, which main characteristic is that the residual is small: $r \approx 0$ (see section 2.3). Hence, the residual equation (2.3) implies $Ae \approx 0$ and therefore the ith component of the error can be approximated as

$$a_{ii} e_i + \sum_{j \in N_i} a_{ij} e_j \approx 0, \tag{2.7}$$

where N_i is the set of all neighboring points of i. The interpolation weights are then set, so that the interpolation is as good as possible. Note, that for the interpolation from a coarse level only the C-points, and not all members of N_i, are available.

2.5.1 Interpolation for Smoothed Aggregation

In [34] a construction scheme for prolongators is introduced, which is extended and analyzed in more detail in [37]. Additionally, [36] gives an optimization method for the strategy of smoothed prolongation which turns out to be a generalization of the previous methods. In the applications of chapter 3 only the case of scalar equations is needed, to which we limit this discussion which is mostly orientated on the summary in [41].

In section 2.4.1 we constructed aggregates $\{\mathcal{A}_i^k\}_{i=1}^{n_{k+1}}$ for a given level k starting from the system matrix $A_k \in \mathbb{R}^{n_k \times n_k}$. Now we want to create a hierarchy of *tentative prolongators* (prolongation operators) $Y_{k+1}^k : \Omega_{k+1} \to \Omega_k$ such that for a given test vector $t_0 \in R^{n_0}$

$$t_0 \in \operatorname{ran}(Y_l^0), \qquad Y_l^0 = Y_1^0 Y_2^1 \cdots Y_{k+1}^k, \qquad Y_0^0 = I, \qquad k = 0, \ldots, M-1. \tag{2.8}$$

A typical example for t_0 is $t_0 = (1, \ldots, 1)^\top$ (what we are going to use) or $t_0 = \ker(A)$. The idea behind this choice is to ensure the exact interpolation of certain vectors. The interested reader is referred to [39, 40] where similar so-called *test vectors* have been introduced.

We are going to construct a vector t_{k+1} and the prolongator Y_{k+1}^k with the relation

$$Y_{k+1}^k t_{k+1} = t_k,$$

where t_k has been constructed during the setup of Y_{k+1}^k or is given for $k = 0$. This implies for (2.8)

$$Y_l^0 t_l = t_0.$$

In the more complex case of systems of partial differential equations, the test vectors t_l become matrices [37].

Every column of the tentative prolongator Y_{k+1}^k is associated with one aggregate $\mathcal{A}_i^k$, as every aggregate gives rise to one degree of freedom on the coarse grid. It is formed by restriction of the corresponding rows of t_k onto the aggregate $\mathcal{A}_i^k$ and is visualized in figure 2.3. For simplicity we assume that the fine level unknowns are numbered by consecutive numbers within each aggregate. This assumption may easily be obtained by possible renumbering. The detailed algorithm 2.7 follows.

Algorithm 2.7 Construction of tentative prolongator

1: **Partition** the vector $t_k \in \mathbb{R}^{n_k}$ into n_{k+1} blocks t_k^i, $i = 1, \ldots, n_{k+1}$
2: $q_k^i \leftarrow \frac{t_k^i}{\|t_l^i\|}$
3: $(t_{k+1})_i \leftarrow \|t_k^i\|$
4: **Construct** Y **and** t_{k+1}**:**
5: $$Y \leftarrow \begin{pmatrix} q_k^1 & & \\ & \ddots & \\ & & q_k^{n_{k+1}} \end{pmatrix}, \quad t_{k+1} \leftarrow \begin{pmatrix} (t_{k+1})_1 \\ \vdots \\ (t_{k+1})_{n_{k+1}} \end{pmatrix}$$

We use the tentative prolongators to define the final transfer operators P_k and R_k, $k = 0, \ldots, M-1$.

$$P_k = Z_k Y_{k+1}^k \qquad\qquad R_k = (P_k)^\top$$

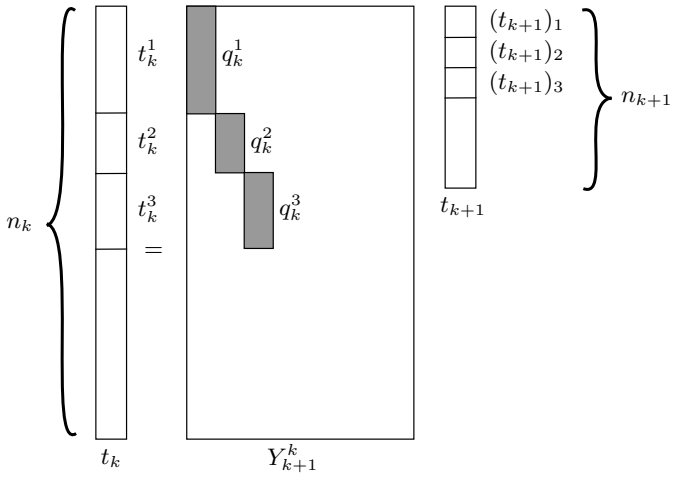


Figure 2.3: Structure of tentative prolongator Y_{k+1}^k

Here Z_k denotes a smoothing operator for Y_{k+1}^k. There are many choices for this operator. In [37] the following form is used to perform a convergence analysis:

$$Z_k = I - \frac{4}{3\bar{\lambda}_k^W} W_k^{-1} A_k$$
$$W_k = \left(Y_k^0\right)\left(Y_k^0\right)^\top, \quad \bar{\lambda}_k^W \geq \rho(W_k^{-1} A_k)$$

Furthermore they show that for system matrices A produced by a finite element discretization of an elliptic partial differential equation

$$\bar{\lambda}_k^W = 9^{-k}\bar{\lambda}$$

is a possible choice, where $\bar{\lambda} \geq \rho(A)$ is an upper bound for the spectral radius of A.

We are going to follow the original suggestions which were proposed in [35]. As a smoothing operator a simple damped Jacobi smoother

$$Z_K = I - \omega D^{-1} A_k^F \tag{2.9}$$

is used. $A_k^F = (a_{ij}^F)$ is the *filtered matrix* given by

$$a_{ij}^F = \left.\begin{cases} a_{ij}, & \text{if } j \in N_i^k(\theta), \\ 0, & \text{otherwise.} \end{cases}\right\} \text{if } i \neq j, \qquad a_{ii}^F = a_{ii} + \sum_{\substack{j=1 \\ j\neq i}}^{n_k} \left(a_{ij} - a_{ij}^F\right), \tag{2.10}$$

where D denotes the diagonal of A_k and $N_i^k(\theta)$ is given by definition 2.4.

Parallel 2.11: As a first fact note, that the filtered matrix A_k^F will be a sparse matrix. We are going to use the notation introduced in chapter 1.2.

In order to setup A_k^F we need the neighborhoods $N_i^k(\theta)$ first. These have been computed already for algorithm 2.4 to perform the aggregation procedure. According to the definition

in (2.10) the nonzero structure of A_k^F is completely determined by these neighborhoods. Hence we are able to allocate exactly the needed memory without any further computation.

The neighborhoods have been saved in a sparse format, similar to the one of a sparse matrix. It follows, that the nonzero structure of matrix A_k^F is given by the vectors JA and IA of the neighborhood data structure. Hence, we do not need to recompute this information and receive full knowledge over every position in this matrix.

Thus, the last step, the actual computation of the values in algorithm 2.8 stored in this vector AA can be performed completely in parallel and without any further preparatory computation. Every row can be computed in parallel as it is independent of any other. ■

Algorithm 2.8 Setup of filtered matrix A_h^F

```
1:  A_h^F ← 0
2:  for i ← 1 to n_k do                    ▷ All iterations can be performed in parallel
3:      s ← 0                              ▷ Sum for weak neighbors in row i
4:      for all j with a_ij ≠ 0 do
5:          if j ∈ N_i^k(θ) then
6:              a_ij^F ← a_ij^F + a_ij     ▷ Strong neighbor
7:          else
8:              s ← s + a_ij               ▷ Weak neighbor
9:          end if
10:     end for
11:     a_ii^F ← a_ii^F + s                ▷ Correct diagonal element
12: end for
```

After the setup of the filtered matrix A_k^F we are able to apply formula (2.9). We give an algorithm to construct the restriction operator R_k because our implementation needs this operator as an input. Formula (2.9) transforms to

$$R_k = {P_k}^\top = {Y_{k+1}^k}^\top \Big(I - \omega D^{-1} A_k^F\Big)^\top.$$

Let $Y_{k+1}^k = (y^1, \ldots, y^{n_{k+1}})$, where $y^i = (0, \ldots, 0, q_k^i, 0, \ldots, 0)^\top$ is the i-th column vector of Y_{k+1}^k. Then the i-th row r^i of R_k is given by

$$r^i = (y^i)^\top - \omega\Big(D^{-1} A_k^F y^i\Big)^\top$$

Therefore the smoothing process can be split in single tasks. The outermost subtraction and the multiplication with the constant ω can be performed individually, hence in parallel for every row. The diagonal values in D can be computed within algorithm 2.8 by an additional if-clause after line 6. The most involved task is the matrix-vector multiplication. Due to the special structure of the column vectors of Y_{k+1}^k, this process can be highly optimized.

The matrix Y_{k+1}^k has only n_k different nonzero values and here again every row has exactly one. Hence, the matrix-matrix multiplication, can be interpreted and performed as a matrix-vector multiplication, as every entry of the matrix A_k^F multiplies with exactly one nonzero

value of the matrix Y_{k+1}^k only. Thus, Y_{k+1}^k can be interpreted as a vector $q \in \mathbb{R}^{n_k}$, where q_i is the nonzero value in row i of Y_{k+1}^k.

The difficulty lies in the fact, that all matrices are stored in the CSR format (see chapter 1.2) and in particular in the fact, that they are saved row by row. Therefore computing the transposed matrix is quite expensive and we want to combine the computation of the values and the transposition into one operation. To do so we are able to exploit the previously mentioned structure. Let us consider the following example first, which explains the key ideas.

$$\left[\begin{pmatrix} a_1 & a_2 & a_3 & \dots \\ b_1 & b_2 & b_3 & \dots \\ \vdots & \vdots & & \end{pmatrix} \begin{pmatrix} q_1 & & \\ q_2 & & \\ & q_3 & \\ & q_4 & \\ & q_5 & \end{pmatrix}\right]^\top = \begin{pmatrix} a_1q_1 + a_2q_2 & a_3q_3 + a_4q_4 + a_5q_5 & \dots \\ b_1q_1 + b_2q_2 & b_3q_3 + b_4q_4 + b_5q_5 & \dots \\ \vdots & \vdots & \end{pmatrix}^\top$$

$$= \begin{pmatrix} \boldsymbol{a_1}q_1 + \boldsymbol{a_2}q_2 & b_1q_1 + b_2q_2 & \dots \\ \boldsymbol{a_3}q_3 + \boldsymbol{a_4}q_4 + \boldsymbol{a_5}q_5 & b_3q_3 + b_4q_4 + b_5q_5 & \dots \\ \vdots & \vdots & \end{pmatrix}$$

Here we see, that every a/b-value multiplies with exactly one q-value only and note, that the a-values of the first row influence only values of the first column in the result. The same statement holds for the b-values of the second row. Note, that due to the sparsity of A_k^F and Y_{k+1}^k most entries of the matrix-matrix product will be zero. Therefore, as a first step, we count the number of entries per row, by checking for every entry, if at least one a/b-value and one q-value not equal to zero multiply with each other. Due to the sparsity and the above mentioned fact, this task can be performed in $\mathcal{O}(n_k)$ operations. With this information we define the vector IA^R of R_k, which marks the start of every row.

The actual multiplication is then done by iterating over all entries of A_k^F, multiplying them with the corresponding q-value and adding them to their predefined position. In the following algorithm 2.9 we will need the data structure M_k defined in (2.6), which maps every node to its aggregate number.

Algorithm 2.9 Matrix-Matrix product $A_h^F Y_{k+1}^k$ to setup restriction operator R_k

1: $R_k = (r_{ij})_{n_k \times n_{k+1}} \leftarrow 0$
2: **for** $i \leftarrow 1$ **to** n_k **do**
3: **for all** j with $a_{i,j}^F \neq 0$ **do**
4: **if** $M_k(j) \leq n_k$ **then** ▷ $M_k(j) = n_k + 1$ for an isolated point
5: $r_{M_k(j),i} \leftarrow r_{M_k j,i} + a_{i,j}^F \cdot \frac{q_i}{a_{ii}}$
6: **end if**
7: **end for**
8: **end for**

Note, that algorithm 2.9 is a serial algorithm, which is because we are dealing with a sparse matrix in the CSR format. Every row can grow in each iteration, and it is not known beforehand, which iterations will influence which rows. Despite that, the algorithm is still very efficient, as it needs the minimal number of operations to compute the matrix-matrix product. It would be possible to compute the interpolation operator P_k in parallel, but in

order to obtain the restriction operator R_k, the previous one would need to be transposed, which results in similar steps as above and in the same complexity. Hence this strategy would be less efficient.

Remark 2.12: Experiments show that the application of algorithm 2.4 to uniformly elliptic problems results usually in a coarsening by a factor of 3 in each dimension. In this case the filtration (2.10) has little or no effect. In the case of anisotropic problems, the application of the smoother with the unfiltered matrix would make the supports of the basisfunctions overlap extensively in the direction of weak connections. Here the filtration prevents these undesired overlaps. The special treatment of the diagonal entries ensures, that the sum of entries in a row of A_k^F is zero whenever the sum of the entries in the corresponding row of A is zero. Hence, for the choice of a constant test vector $t_0 = (1, \ldots, 1)$, a constant remains in the local kernel of A_k^F at every point where a constant is in the local kernel of A_k. This ensures certain requirements for an efficient and robust AMG algorithm, developed in [35]. We are not going to pursue a detailed discussion of these requirements, as the aim of this work is not so much in the mathematical theory but rather in its numerical implementations and experiments. ■

Example 2.13: Consider the standard one dimensional model problem

$$Au = f, \quad A = [-1\ 2\ -1], \quad A \in R^{n \times n},$$

resulting from the discretization of a one dimensional Laplace operator. If the damped Jacobi smoother

$$S = I - \frac{2}{3} D^{-1} A, \quad D = \mathrm{diag}(A),$$

with damping factor $\omega = 2/3$ is applied to the vector $\vartheta = (0, \ldots, 0, 1, 1, 1, 0, \ldots, 0)^\top$ we get the smoothed vector

$$S\vartheta = \varphi = (0, \ldots, 0, \frac{1}{3}, \frac{2}{3}, 1, \frac{2}{3}, \frac{1}{3}, 0, \ldots, 0)^\top.$$

The vectors ϑ and φ can be interpreted as discretizations of coarse grid basisfunctions. The effect of smoothing these vectors is shown in figure 2.4. As mentioned above, this strategy reduces the number of degrees of freedom on the coarse grid to approximately 1/3 compared to the number of fine grid points. Note, that the prolongation induced by the vectors $(0, \ldots, 0, 1/2, 1, 1/2, 0, \ldots, 0)^\top$ which are discrete versions of the standard 1D linear basisfunctions leads to a coarsening of about 1/2. ■

Numerical experiments have shown that the suggested parameters

$$\theta = 0.08 \cdot 2^{-k} \quad \text{and} \quad \omega = \frac{2}{3}$$

achieve in general a reasonable balance between coarsening and interpolation. The choice of a fixed θ for all levels, results in general in two slow coarsening on lower levels. But many problems are very sensible to these values and require an independent analysis.

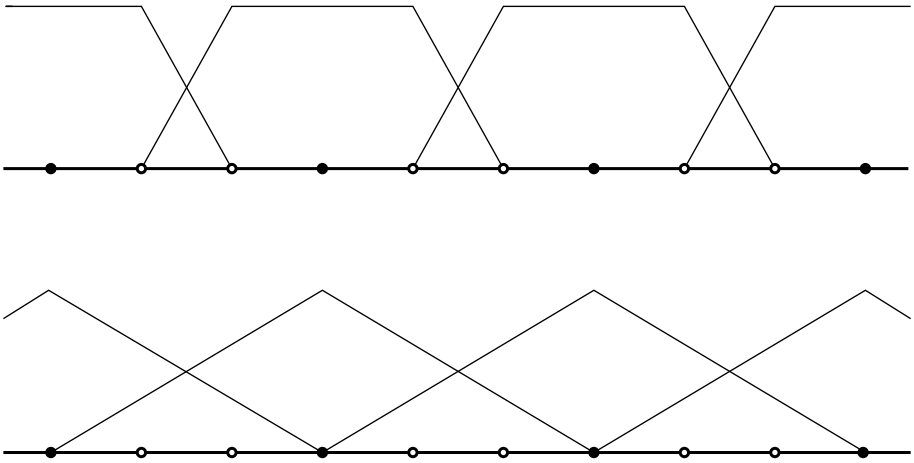


Figure 2.4: Smoothed basisfunctions

2.5.2 Direct Interpolation

Direct interpolation is one implemented approach, which applies in the case that the C/F-splitting has been constructed by means of classical coarsening procedures like RS coarsening [45, p. 223], [33, p. 479]. The idea behind this procedure is, that influences from F-points can be neglected and that interpolation can be done from neighboring C-points only. In the easiest case weights are defined as

$$w_{ij} = -\left(\frac{\sum_{l\in N_i} a_{il}}{\sum_{m\in C_i} a_{im}}\right)\frac{a_{ij}}{a_{ii}},$$

where $C_i = C \cap N_i$ are the coarse neighbors of point i.

This approach can be improved by distinguishing between positive and negative matrix elements. Therefore we define the following notation

$$a_{ij}^{+} = \begin{cases} a_{ij}, & \text{if } a_{ij} > 0, \\ 0, & \text{otherwise,} \end{cases} \qquad a_{ij}^{-} = \begin{cases} a_{ij}, & \text{if } a_{ij} < 0, \\ 0, & \text{otherwise.} \end{cases}$$

Additionally we define the set of interpolatory points P_i for point i as $P_i := C_i \cap S_i$, where S_i is defined like in definition 2.7. The definition of positive and negative connections is canonically extended onto sets, i.e. P_i^+ and P_i^-. Now we give a variant of the direct interpolation, whose implementation showed better convergence properties in most performed experiments.

$$w_{ij} = \begin{cases} -\alpha_i\, a_{ij}/a_{ii}, & \text{for } j \in P_i^-, \\ -\beta_i\, a_{ij}/a_{ii}, & \text{for } j \in P_i^+, \end{cases}$$

with

$$\alpha_i = \frac{\sum_{l\in N_i} a_{il}^-}{\sum_{m\in P_i} a_{im}^-} \qquad \text{and} \qquad \beta_i = \frac{\sum_{l\in N_i} a_{il}^+}{\sum_{m\in P_i} a_{im}^+}.$$

If $P_i^+ = \emptyset$, this formula is modified, such that we set $\beta_i = 0$ and add all positive entries, if there are any, to the diagonal.

Note, that for M-matrices (compare remark on page 29) both procedures will be equivalent. Since the formulas involve only direct connections of the variable i, this approach is referred to as *direct interpolation*.

This method obviously only makes sense if the interpolation of F-points from C-points only is reasonably good. This has to be guaranteed by the used coarsening method, otherwise interpolation and therefore convergence will deteriorate. This problem is a motivation for the definition of strong and weak connections for points with big or small influence. The idea is that strongly influencing points become C-points and guarantee a good interpolation.

Parallel 2.14: The rather simple structure of the formulas, enables a straightforward parallel implementation, which somehow compensates the loss in convergence for certain problems. However, our implementation in `maiprogs` requires the restriction instead of the interpolation matrix. Because this matrix is saved in the CSR format, it requires more effort to implement it in parallel, since the matrix is created column by column (compare algorithm 2.9). As our main AMG algorithm is the method of smoothed aggregation, we do not consider this case here. ■

2.5.3 Standard Interpolation

In order to achieve better convergence, more information has to be used. The problem of direct interpolation are strong F-F-connections, as it is assumed, that all F-points can be interpolated well enough from C-points only. Standard interpolation works like direct interpolation but eliminates components of strongly influencing F-points in formula (2.7) first by approximating these with the j-th equation

$$e_j = -\sum_{l \in N_j} \frac{a_{jl}}{a_{jj}}.$$

This results in a new equation for e_i:

$$\hat{a}_{ii} e_i + \sum_{j \in \hat{N}_i} \hat{a}_{ij} e_j \approx 0, \qquad \text{with} \qquad \hat{N}_i := \{j \neq i \mid \hat{a}_{ij} \neq 0\}$$

Defining the interpolatory set P_i as the union of $C_i \cap S_i$ and all $C_j \cap S_i$ where $j \in F \cap N_i \cap S_i$, we now define interpolation exactly as for direct interpolation with all as replaced by $\hat{a}$s and N_i replaced by $\hat{N}_i$.

Standard interpolation leads to very good convergence, but the computation of the extended neighborhoods results in higher setup times.

Parallel 2.15: The computation of the new matrix $\hat{A} = (\hat{a}_{ij})$ can be mostly done in parallel. Otherwise the the discussion at direct interpolation applies. ■

Chapter 3

Higher-Order Finite Element Discretizations on Quadrilaterals

Multigrid methods can be split into two approaches: the geometric-based approach and the algebraic approach (see figure 2.1). In this chapter we consider a hybrid method which can be seen as an algebraic multigrid algorithm based on geometric considerations. We follow the ideas of Shu et al. who introduced this ideas for triangulations [31] and extend them onto domains discretized by quadrilaterals. This approach is used to solve algebraic systems which arise from the discretization of second order elliptic partial differential equations obtained by high-order finite element discretization using Lagrangian finite elements or any hierarchical basis in two spacial dimensions.

Lagrangian finite elements of different orders are very popular choices for finite element bases. They are naturally defined on a mesh of triangles or quadrilaterals in which a definition can be found in section 1.4.

3.1 A Hybrid Method

The development of classical algebraic multigrid (AMG) methods was mainly focused on bilinear elements and they are much more efficient for these than for higher-order elements. One main reason for the higher efficiency when using bilinear elements is that the algebraic mesh graph corresponding to the stiffness matrix is very similar to the geometric mesh graph. This property is lost when using higher-order finite elements which lead to more dense stiffness matrices, and consequently to a more complicated coarsening process (cf. [16], [17], [43]).

The main idea is to reveal hidden geometric information of the stiffness matrix in order to algebraically construct the bilinear finite element stiffness matrix as first coarsening step and then apply the classical AMG method on the retrieved matrix. This process can be interpreted as a two-level method in which the coarse space is a bilinear finite element space. The main step in our algorithm is the identification of the bilinear finite element subspace by algebraic means only, but using further geometric information about the discretization.

Compared to the classical AMG a less algebraic method is obtained in the sense that more a-priori knowledge is necessary than the linear algebraic system all alone. In particular we

need to know

1. the type of the finite element space (i.e. biquadratic or bicubic) and
2. what type of basisfunctions are used in generating the stiffness matrices.

The goal is to use as little information as necessary in order to improve a given AMG solver by exploiting the high efficiency of the classical AMG methods on bilinear finite element spaces. This can be compared with the core idea of multigrid to utilize the excellent error damping properties of classical smoothers on high frequency errors via introducing hierarchies of coarser grids.

3.2 Biquadratic Lagrangian Finite Elements

In the following we are going to follow the structure of [31] who described the technical results for the case of triangles.

Let R^h be the underlying quadrilateral partition of the domain Ω as for example shown in figure 3.1a. The problem (1.1) is discretized by a quadratic Lagrangian FEM basis and results in the following discrete version

$$A_h^2 u_h^2 = f_h^2 \tag{3.1}$$

where $A_h^2 = (a_{ij})_{i,j\in I}$, $u_h^2 = (u_i)_{i\in I}$ and $f_h^2 = (f_i)_{i\in I}$ with $I = \{1, \ldots, N\}$ and N denotes the number of freedom in above equation (3.1).

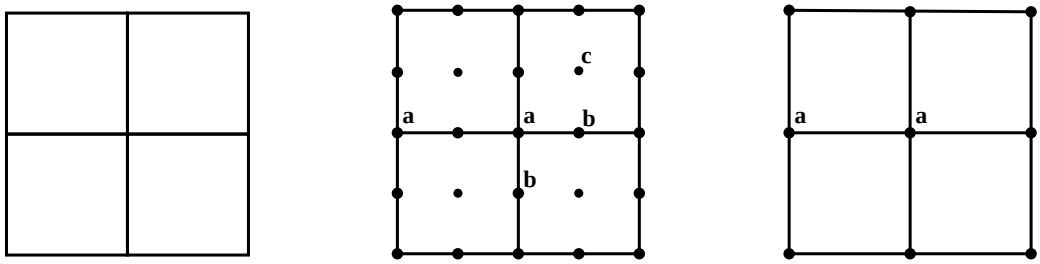
As an input to our algorithm we need the *stiffness matrix* A_h^2, the *right hand side vector* f_h^2 plus the knowledge of the *degree* $p = 2$ and the fact that *nodal basisfunctions* have been used. The first step is to split the nodes into three classes:

Definition 3.1: All vertices of the grid which do not lie on the Dirichlet boundary are called *type-a nodes*, the mid points of the edges which do not lie on the Dirichlet boundary are defined as *type-b nodes* and the mid points of the elements are defined as *type-c nodes* (see Fig. 3.1b). The sets of all type-a, type-b and type-c nodes are denoted by X_a, X_b and X_c, respectively and N_a, N_b and N_c are the cardinalities of theses sets. □

Remark 3.2: In the case of triangles two types of nodes (type-a and type-b) are sufficient, due to the fact that the mid point of a quadrilateral element is the mid point of the longest side of the triangle in that case and therefore a type-b node (cf. [31]). ■

As all nodes are considered it follows that $N = N_a + N_b + N_c$. With these definitions we divide the set S of all indices in (3.1) into the following three subsets where x_i is the node related to the component u_i:

$$S_a = \{i \mid i \in S,\ x_i \in X_a\}, \quad S_b = \{i \mid i \in S,\ x_i \in X_b\}, \quad S_c = \{i \mid i \in S,\ x_i \in X_c\} \tag{3.2}$$



(a) The grid R^h (b) Nodes in V_h^2 (c) Nodes in V_h^1

Figure 3.1: Type-a, type-b and type-c nodes on quadrilateral elements

As mentioned in chapter 1.1 let V_h^2 be the biquadratic Lagrangian finite element space on the grid R^h and $\{\phi_i\}_{i\in S}$ be the corresponding nodal basisfunctions which satisfy

$$\phi_i(x_j) = \delta_{i,j} \qquad \forall i,j \in S. \tag{3.3}$$

The nodal basisfunctions $\{\phi_i\}_{i\in S}$ of the biquadratic Lagrangian finite element space, are in the same manner divided into three groups which will be represented by the vectors $\Phi^a = (\phi_{i_1},\dots,\phi_{i_{N_a}})^\top$, $\Phi^b = (\phi_{j_1},\dots,\phi_{j_{N_b}})^\top$ and $\Phi^c = (\phi_{k_1},\dots,\phi_{k_{N_c}})^\top$ for all $i_p \in S_a$, $j_q \in S_b$ and $k_r \in S_c$. This gives rise to the following matrix notation:

$$\begin{aligned}
&a(\Phi^a,\Phi^a) := (a_{ij})_{i,j\in S_a}\,, && a(\Phi^a,\Phi^b) := (a_{ij})_{i\in S_a, j\in S_b}\,, && a(\Phi^a,\Phi^c) := (a_{ij})_{i\in S_a, j\in S_c}\,,\\
&a(\Phi^b,\Phi^a) := (a_{ij})_{i\in S_b, j\in S_a}\,, && a(\Phi^b,\Phi^b) := (a_{ij})_{i,j\in S_b}\,, && a(\Phi^b,\Phi^c) := (a_{ij})_{i\in S_b, j\in S_c}\,,\\
&a(\Phi^c,\Phi^a) := (a_{ij})_{i\in S_c, j\in S_a}\,, && a(\Phi^c,\Phi^b) := (a_{ij})_{i\in S_c, j\in S_b}\,, && a(\Phi^c,\Phi^c) := (a_{ij})_{i,j\in S_c}\,.
\end{aligned}$$

The interactions of basisfunctions corresponding to a certain class of nodes is represented by these nine parts of the stiffness matrix A_h^2. After possible reordering this interplay can be seen in the stiffness matrix which is a first step in revealing the hidden geometric information:

$$A_h^2 = \begin{pmatrix} a(\Phi^a,\Phi^a) & a(\Phi^a,\Phi^b) & a(\Phi^a,\Phi^c) \\ a(\Phi^b,\Phi^a) & a(\Phi^b,\Phi^b) & a(\Phi^b,\Phi^c) \\ a(\Phi^c,\Phi^a) & a(\Phi^c,\Phi^b) & a(\Phi^c,\Phi^c) \end{pmatrix}. \tag{3.4}$$

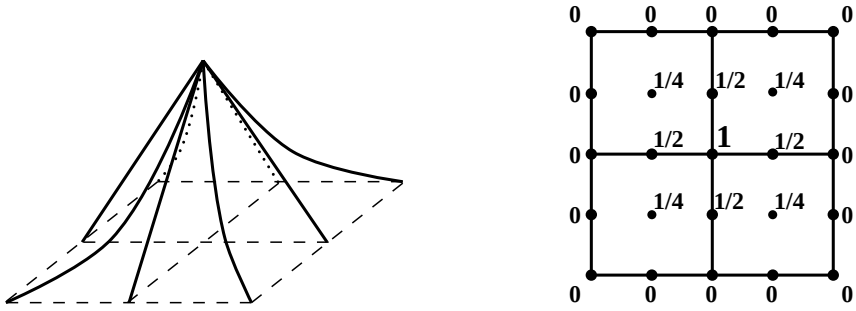
As a first step we introduce the bilinear finite element space V_h^1 on the grid R^h with the corresponding nodal basisfunctions $\{\psi_i\}_{i\in S_a}$ (see Fig. 3.1c) as the coarse space of the biquadratic finite element space V_h^2. The nodal basisfunctions satisfy

$$\psi_i(x_j) = \delta_{i,j} \qquad \forall i,j \in S_a. \tag{3.5}$$

Due to $V_h^1 \subset V_h^2$ we are able to represent every basisfunction of V_h^1 as a linear combination of basisfunctions of V_h^2. Using (3.3) and (3.5) and the compact support properties of $\{\psi_i\}_{i\in S_a}$ all we need to find are the nodal values of the bilinear basisfunctions at the interpolatory nodes of type-a, type-b and type-c. A straightforward calculation gives

$$\psi_i(x) = \phi_i(x) + \frac{1}{2}\sum_{j\in S_i^{b1}} \phi_j(x) + \frac{1}{4}\sum_{k\in S_i^{c1}} \phi_k(x), \qquad i \in S_a, \tag{3.6}$$

where $S_i^{b_1} := \{j \mid \psi_i(x_j) \neq 0,\ j \in S_b\} \subseteq S_b$ and $S_i^{c_1} := \{k \mid \psi_i(x_k) \neq 0,\ k \in S_c\} \subseteq S_c$ are the *interpolatory sets* of type-b and type-c nodes, respectively, for the type-a node i. This is also visualized in the following figure 3.2.



(a) Bilinear basisfunction (b) Values at interpolatory nodes

Figure 3.2: Interpolation of bilinear from biquadratic basisfunctions

In the same way as for the biquadratic basisfunction $\phi_i(x)$ we define $\Psi := (\psi_{i_1}, \ldots, \psi_{i_{N_a}})^\top$, $i_p \in S_a$. Hence we can write the last equation in matrix form

$$\Psi = P_h^H \begin{pmatrix} \Phi^a \\ \Phi^b \\ \Phi^c \end{pmatrix}, \tag{3.7}$$

where P_h^H is a $N_a \times N$-matrix.

This operator defines our *restriction operator* from V_h^2 to V_h^1 and we choose $P_H^h := (P_h^H)^\top$ as our *prolongation operator* from V_h^1 to V_h^2. The coarse grid operator is constructed using (3.4), (3.7) and the Galerkin condition:

$$A_h^1 := (a(\Psi, \Psi)) = P_h^H A_h^2 P_H^h \tag{3.8}$$

This enables us to construct a two-level algorithm for solving the linear system (3.1):

Algorithm 3.1 Two-Level Algorithm for the biquadratic Lagrangian FEM-Equation

1: **Presmoothing:** $u_h \leftarrow u_h + S(f - A_h^2 u_h)$, $j = 1, \ldots, m_1$
2: **Solving the coarse equation:** $e_{h,1} \leftarrow (A_h^1)^{-1} P_h^H \left(f - A_h^2 u_h\right)$
3: **Correcting:** $u_h \leftarrow u_h + P_H^h e_{h,1}$
4: **Postsmoothing:** $u_h \leftarrow u_h + S(f - A_h^2 u_h)$, $j = 1, \ldots, m_2$

Interpolatory Sets

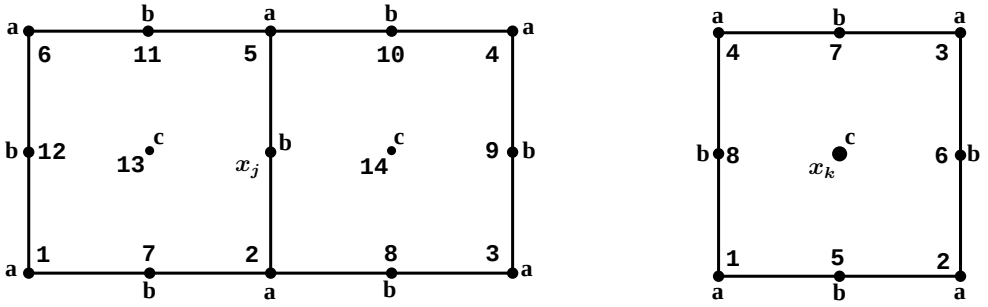
In order to apply algorithm 3.1 we need to find the interpolatory sets $S_i^{b_1}$ and $S_i^{c_1}$, which depend on the sets S_a, S_b and S_c. Hence, we have to solve the following two problems:

Problem *Q*1: Find the sets S_a, S_b and S_c from the coefficient matrix A_h^2.
Problem *Q*2: Find the interpolatory sets $S_i^{b_1}$ and $S_i^{c_1}$ for any fixed $i \in S_a$.

3.2.1 Algorithm for Problem $Q1$ – Find S_a, S_b and S_c

In the following, if not explicitly stated, for the indices i, j, k it holds that $i \in S_a$, $j \in S_b$ and $k \in S_c$. First we introduce index sets as defined below where $\operatorname{supp}\phi_i$ denotes the support of function ϕ_i.

$$
\begin{aligned}
S_i &= \{l \mid \operatorname{supp}\phi_i \cap \operatorname{supp}\phi_l \neq \emptyset,\ l \neq i,\ l \in S\} \\
S_i^b &= \{l \mid \operatorname{supp}\phi_i \cap \operatorname{supp}\phi_l \neq \emptyset,\ l \in S_b\} \\
S_i^c &= \{l \mid \operatorname{supp}\phi_i \cap \operatorname{supp}\phi_l \neq \emptyset,\ l \in S_c\} \\
R_j &= \{l \mid \operatorname{supp}\phi_i \cap \operatorname{supp}\phi_l \neq \emptyset,\ l \neq j,\ l \in S\} \\
R_j^{bc} &= \{l \mid \operatorname{supp}\phi_i \cap \operatorname{supp}\phi_l \neq \emptyset,\ l \neq j,\ l \in S_b \cup S_c\} \\
E_k &= \{l \mid \operatorname{supp}\phi_i \cap \operatorname{supp}\phi_l \neq \emptyset,\ l \neq k,\ l \in S\} \\
E_k^{bc} &= \{l \mid \operatorname{supp}\phi_i \cap \operatorname{supp}\phi_l \neq \emptyset,\ l \neq k,\ l \in S_b \cup S_c\}
\end{aligned}
$$



(a) Neighbors of $x_j \in X_b$ (b) Neighbors of $x_k \in X_c$

Figure 3.3: Maximal Number of neighbors for elements of X_b and X_c;
(a): $R_j^{bc} = \{7, 8, \ldots, 14\}$, $R_j = \{1, 2, \ldots, 14\}$, (b): $E_k^{bc} = \{5, 6, 7, 8\}$, $E_k = \{1, 2, \ldots, 8\}$

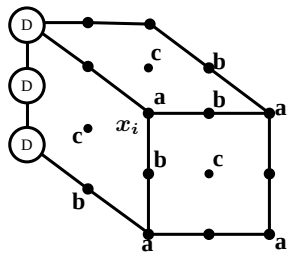
Additionally we give the following definitions.

Definition 3.3: An edge is called an *interior edge* if its two vertices do not belong to the Dirichlet boundary. If an edge has only one vertex on the Dirichlet boundary, then we call it a *half-interior edge*. If four edges of an element in R^h are all interior edges, then this element is called an *interior element* otherwise *boundary element*. For a given node $x_i \in X_a$ an element is called a *neighboring element* of x_i, if x_i is a vertex of the element. □

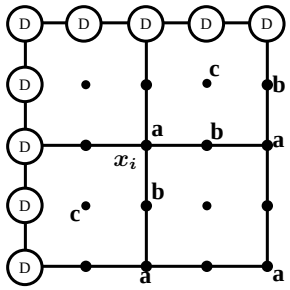
Furthermore we assume that for the given grid R^h every element has at least one type-a node (i.e. not all four vertices on the Dirichlet boundary) and that every node in X_a fulfills at least one of the following assumptions.

Assumption 3.4: There exists at least one interior element of the node $x_i \in X_a$ and

(1) two other neighboring elements which possess in total at most two half-interior edges,
(2) at least three other neighboring elements. ■



(a) Three neighboring elements, two half-interior edges, one interior element; $p = 2$



(b) Four neighboring elements, one interior element; $p = 2$

Figure 3.4: Minimal number of neighbors for elements of X_a under Assumption 3.4 for $p = 2$

Under the Nonzero Assumption 1.2, for any $i \in S_a$, $j \in S_b$ and $k \in S_c$, $|S_i| + 1$, $|R_j| + 1$ and $|E_k| + 1$ are exactly the nonzero entries of the i-th, j-th and k-th row in the matrix A_h^2, respectively.

Combining implication (1.3) of the Nonzero Assumption 1.2 and the compact support properties of the basisfunction ϕ_i we are able to split the set S into the two groups of type-a nodes and type-b/c nodes. This is done by counting the neighbors in Fig. 3.3a and Fig. 3.4.

Proposition 3.5: For a given grid R^h, under the Nonzero Assumption 1.2, we have

$$|S_i| \geq 15, \qquad \text{for any } i \in S_a$$

and

$$\begin{aligned} |R_j| &\leq 14, && \text{for any } j \in S_b, \\ |E_k| &\leq 8, && \text{for any } k \in S_c. \end{aligned}$$

Therefore we are able to uniquely identify type-a nodes among all other nodes, by only algebraic means.

Criterion 3.6: For any index $i \in S$, if there are more than 15 nonzero entries in the row i of the matrix A_h^2, then the node x_i related to the index i is a type-a node. ∎

In the same way as in [31] we introduce a flag array $I_a(k)$, $k = 1, \ldots, N$ for the matrix A_h^2, which satisfies

$$I_a(k) := \begin{cases} 1, & x_k \in X_a, \\ 0, & x_k \in X_b \cup X_c. \end{cases}$$

Parallel 3.7: By criterion 3.6 the process of checking if a row i corresponds to a type-a node is independent of all the other rows $j \neq i$ in the matrix A_h^2. Hence this process can be

performed in parallel. If the matrix is stored in the CSR format (chapter 1.2) the complexity of setting up I_a is $\mathcal{O}(N)$. ∎

With the purpose of distinguishing between type-b and type-c nodes we investigate certain types of neighborhoods which represent up to permutation all possible ones. In particular their layout on the Dirichlet-boundary is of interest.

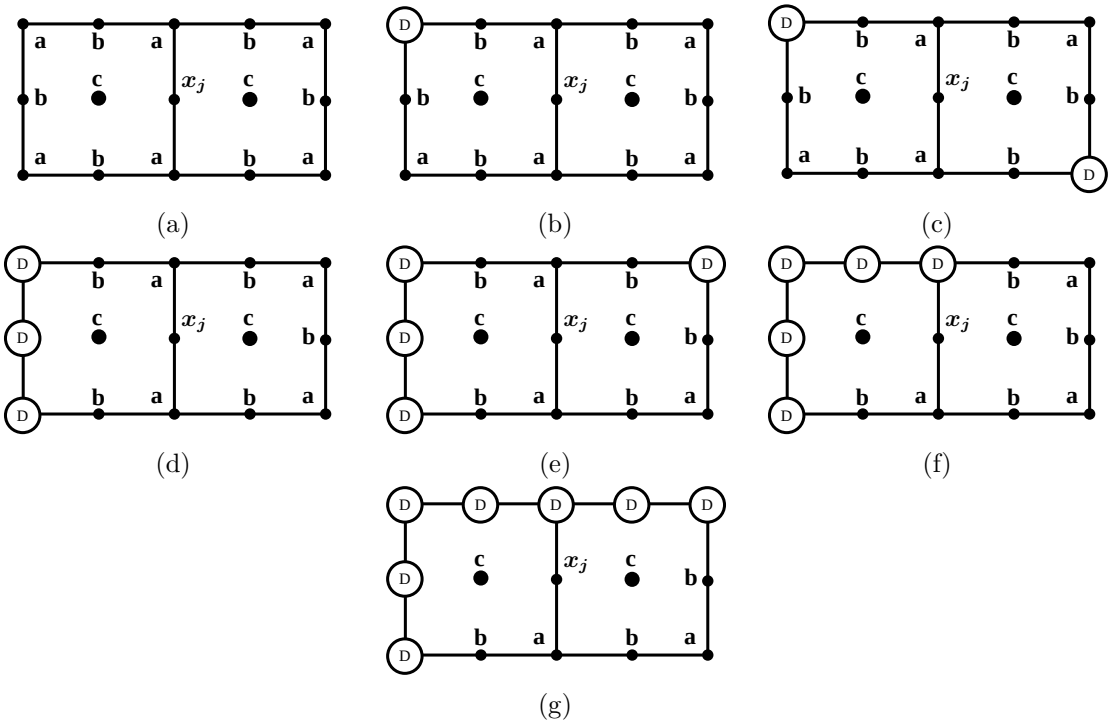


Figure 3.5: Neighborhoods of type-b nodes; (a): $|R_j|=14, |R_j^{bc}|=8$; (b): $|R_j|=13, |R_j^{bc}|=8$; (c): $|R_j|=12, |R_j^{bc}|=8$; (d): $|R_j|=11, |R_j^{bc}|=7$; (e): $|R_j|=10, |R_j^{bc}|=7$; (f): $|R_j|=9, |R_j^{bc}|=6$; (g): $|R_j|=7, |R_j^{bc}|=5$

As we see in Fig. 3.5 and Fig. 3.6, is the number of nonzero entries in a row (i.e. $|R_j|$ or $|E_k|$) no unique identifier on its own. In particular the cardinality 7 appears for both types of these sets. We could distinguish this case by checking a second parameter, like e.g. the number of type-a nodes in their neighborhood.

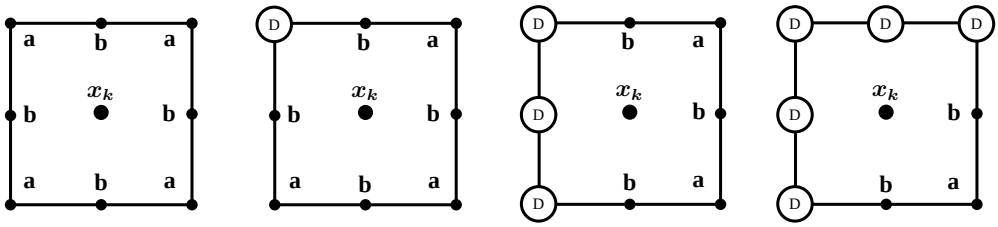
But in this geometry the size of the type-b and type-c neighborhoods R_j^{bc} and E_k^{bc} provides a unique way of distinguishing between these two nodes. This leads us to the following proposition and splits the set $X_b \cup X_c$.

Proposition 3.8: For a given grid R^h, under the Nonzero Assumption 1.2, we have

$$|R_j^{bc}| \geq 5, \qquad \text{for any } j \in S_b$$

and

$$|E_k^{bc}| \leq 4, \qquad \text{for any } k \in S_c.$$



(a) (b) (c) (d)

Figure 3.6: Neighborhoods of type-c nodes; (a): $|E_k|=8, |E_k^{bc}|=4$; (b): $|E_k|=7, |E_k^{bc}|=4$; (c): $|E_k|=5, |E_k^{bc}|=3$; (d): $|E_k|=4, |E_k^{bc}|=2$

Hence we obtain the following criterion for type-c nodes.

Criterion 3.9: For any index $i \in S$, which is not a type-a node, if there are less than six nonzero entries whose column indices are not related to type-a nodes, then the node x_i related to the index i is a type-c node. ■

As a final step we have to identify the neighbor index sets S_i^b, S_i^c, R_j^{bc} and E_k^{bc} and give an algorithm to construct them. To do so the following criterion 3.10 will be used.

Criterion 3.10: For any fixed index $i \in S_a$, $j \in S_b$ and $k \in S_c$ under the Nonzero Assumption 1.2, there holds:

1. The index $l \in S_i^b$ or $l \in S_i^c$, iff $l \in S_b$ or $l \in S_c$, respectively, and $\operatorname{supp}\phi_i \cap \operatorname{supp}\phi_l \neq \emptyset$.
2. The index $l \in R_j^{bc}$ or $l \in E_k^{bc}$, iff $l \neq j$ or $l \neq k$, respectively, $l \in S_b \cup S_c$ and $\operatorname{supp}\phi_i \cap \operatorname{supp}\phi_l \neq \emptyset$. ■

We will give an algorithm to create $S_i^{bc} := S_i^b \cup S_i^c$, R_j^{bc} and E_k^{bc}, then the other two sets can be extracted out of S_i^{bc} by criterion 3.9. In the following the last two sets are treated equivalently, as from the algorithmic point of view, there is no difference in creating them. First we introduce a vector nc of length N, where each entry counts the size of the bc-neighborhoods for the corresponding entry. This can be done using the vector I_a. Then we allocate a sparse structure consisting of nbc, $inbc$ that is stored in the CSR format, similar to a sparse matrix but without the array for the values, as only their offsets are of interest.

Parallel 3.11: Algorithm 3.2 is designed for parallel computation. Every iteration in the first and the last loop is independent from any other. Hence, it can be performed in parallel. The second loop must be performed iteratively, as every value depends on the previous one. However, by determining the storage size of the array nbc with the first two loops, the last loop to run can be performed in parallel. ■

Remark 3.12: The splitting of the set S_i^{bc} (cf. criterion 3.9) can be postponed till the point where the restriction operator is created because type-b and type-c nodes are treated in the

Algorithm 3.2 Generate R_j^{bc} and E_k^{bc} for $p = 2$

```
1: for i ← 1 to N do                                          ▷ Count members
2:     for all j with a_{i,j} ≠ 0 and I_a(j) == 0 do
3:         nc(i) ← nc(i) + 1                                  ▷ type-b/c neighbor
4:     end for
5: end for
6:
7: inbc(1) ← 0                                                ▷ Initialize inbc
8: for i ← 2 to N+1 do
9:     inbc(i) ← inbc(i − 1) + nc(i − 1)
10: end for
11:
12: for i ← 1 to N do                                         ▷ Initialize nbc
13:     cnt ← inbc(i)
14:     for all j with a_{i,j} ≠ 0 and i ≠ j and I_a(j) == 0 do
15:         nbc(cnt) ← j
16:         cnt ← cnt + 1
17:     end for
18: end for
```

same way as they do not correspond to a bilinear basisfunction like the type-a nodes. ■

3.2.2 Algorithm for Problem $Q2$ – Find $S_i^{b_1}$ and $S_i^{c_1}$

Now we have to identify the interpolatory sets $S_i^{b_1}$ and $S_i^{c_1}$ for every $i \in S_a$. It is easy to see that $S_i^c = S_i^{c_1}$ because the values of the bilinear basisfunction ψ_i at the type-c nodes cannot be trivial (cf. figure 3.2). In particular the value is always 1/4.

For the type-b node the case is more complicated, because ψ_i can be equal zero at neighboring type-b nodes. We revealed above that the set S_i is always bigger than the sets R_j and E_k of its neighboring nodes. But there holds another important relationship.

Criterion 3.13: For any index i related to a type-a node, and for any index $j \in S_i^b$, we have $j \in S_i^{b_1}$, iff $R_j^{bc} \subseteq S_i^{bc}$. ■

This must always hold, as every type-b node has at least two type-c neighbors and is shown by figure 3.5, which shows all possible cases if we consider the type-a node at the bottom right corner. We give an algorithm for this last algebraic step of the setup phase.

Parallel 3.14: The difficulty of a parallel execution of algorithm 3.3 is the growing set $S_i^{b_1}$. As we are dealing with a partition of the domain every type-a node has only a fraction of all possible elements as neighboring ones. Therefore the size of the set S_i^b is bounded by a small constant which again bounds the maximal size of $S_i^{b_1}$. This holds because every neighboring element contains at most 4 type-b nodes. Hence, a possible solution is to determine the

Algorithm 3.3 Generate $S_i^{b_1}$ for $p = 2$

1: **for all** i with $I_a(i) == 1$ **do** ▷ Coarse Nodes
2: $S_i^{b_1} = \emptyset$
3: **for all** $j \in S_i^b$ **do** ▷ type-b neighbors
4: **if** $R_j^{bc} \subseteq S_i^{bc}$ **then**
5: $S_i^{b_1} = S_i^{b_1} \cup \{j\}$
6: **end if**
7: **end for**
8: **end for**

maximal size M of $S_i^{b_1}$ for all $i \in S_a$ first and then store the values in a $N_a \times M$-matrix. Here every row corresponds to an index $i \in S_a$. Furthermore we need a vector of length N_a which stores the actual size of the row i. This creates independent memory for every iteration and enables them to be performed in parallel. ■

With the obtained sets S_a and $S_i^{b_1}$, $S_i^{c_1}$ for any $i \in S_a$ and the equation (3.6) we are able to construct the restriction operator P_h^H. The prolongation operator is defined as $P_H^h = \left(P_h^H\right)^\top$ and the coarse grid matrix is calculated by $A_h^1 = P_H^h A_h^2 P_h^H$. By adapting algorithm 3.1 we give the multigrid algorithm for the biquadratic FEM equation (3.1):

Algorithm 3.4 Modified Algorithm for the biquadratic Lagrangian FEM-Equation

Setup:
Find the flag array I_a by criterion 3.6
Split $S_b \cup S_c$ by criterion 3.9
Find sets S_i^b, S_i^c, R_j^{bc} and E_k^{bc} by criterion 3.10 and algorithm 3.2
Find sets $S_i^{b_1}$ and $S_i^{c_1}$ by criterion 3.13 and algorithm 3.3
Construct the restriction operator P_h^H by (3.6)
Construct the prolongation operator $P_H^h = \left(P_h^H\right)^\top$
Construct the coarse matrix $A_h^1 = P_H^h A_h^2 P_h^H$
Presmoothing: $u_h \leftarrow u_h + S(f - A_h^2 u_h)$, $j = 1, \ldots, m_1$
Solving the coarse equation: $e_{h,1} \leftarrow {A_h^1}^{-1} P_h^H \left(f - A_h^2 u_h\right)$
Correcting: $u_h \leftarrow u_h + P_H^h e_{h,1}$
Postsmoothing: $u_h \leftarrow u_h + S(f - A_h^2 u_h)$, $j = 1, \ldots, m_2$

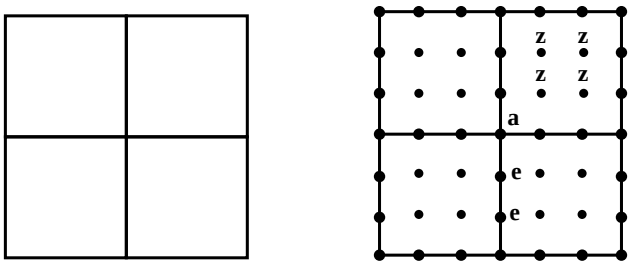
3.3 Bicubic Lagrangian Finite Elements

An extension to higher-order elements has also been presented in [31] for triangular meshes. This extension is not trivial, because as the order of the elements increases, the algebraic identification of different types of nodes becomes more complicated due to the fact that more nodes per element are present and new types of nodes emerge. In this section we are going to consider the extension to bicubic elements on quadrilateral meshes.

Let R^h be the underlying quadrilateral partition of the domain Ω as shown in figure 3.7a. The problem (1.1) is discretized by a bicubic Lagrangian FEM basis and results in the following discrete version

$$A_h^3 u_h^3 = f_h^3 \tag{3.9}$$

where $A_h^3 = (a_{ij})_{i,j\in I}$, $u_h^3 = (u_i)_{i\in I}$, $f_h^3 = (f_i)_{i\in I}$ with $I = \{1, \ldots, N\}$ and N denotes the number of freedom in (3.9).



(a) The grid R^h (b) Nodes in V_h^3

Figure 3.7: Type-a, type-e and type-z nodes on quadrilateral elements

Alike to the biquadratic case we need the *stiffness matrix* A_h^3, the *right hand side vector* f_h^3 plus the knowledge of the *degree* $p = 3$ and the fact that *nodal basisfunctions* have been used as input to our algorithm. The first step is to split the nodes into three classes:

Definition 3.15: All vertices of the grid are called *type-a nodes*, points on the edges are defined as *type-e nodes* and points in the interior of the elements are defined as *type-z nodes* (see Fig. 3.1b). The set of all type-a, type-e and type-z nodes are denoted by X_a, X_e and X_z, respectively, and N_a, N_e and N_z are the cardinalities of theses sets. □

Trivially it holds that $N = N_a + N_e + N_z$. Next the set S of all indices in (3.9) is divided into the following three subsets where x_i is the node related to u_i:

$$S_a = \{i \mid i \in S,\ x_i \in X_a\}, \quad S_e = \{i \mid i \in S,\ x_i \in X_e\}, \quad S_z = \{i \mid i \in S,\ x_i \in X_z\} \tag{3.10}$$

Let V_h^3 be the bicubic Lagrangian finite element space on the grid R^h and $\{\gamma_i\}_{i\in S}$ be the corresponding nodal basisfunctions which satisfy

$$\gamma_i(x_j) = \delta_{i,j} \qquad \forall i, j \in S. \tag{3.11}$$

To unveil the structure of the system matrix A_h^3 we introduce basisfunction vectors $\Upsilon^a = (\gamma_{i_1}, \ldots, \gamma_{i_{N_a}})^\top$, $\Upsilon^e = (\gamma_{j_1}, \ldots, \gamma_{j_{N_e}})^\top$ and $\Upsilon^z = (\gamma_{k_1}, \ldots, \gamma_{k_{N_z}})^\top$ for all $i_p \in S_a$, $j_q \in S_e$ and $k_r \in S_z$. This gives rise to the following matrix notation:

$$\begin{aligned}
&a(\Upsilon^a, \Upsilon^a) := (a_{ij})_{i,j\in S_a}, && a(\Upsilon^a, \Upsilon^e) := (a_{ij})_{i\in S_a, j\in S_e}, && a(\Upsilon^a, \Upsilon^z) := (a_{ij})_{i\in S_a, j\in S_z},\\
&a(\Upsilon^e, \Upsilon^a) := (a_{ij})_{i\in S_e, j\in S_a}, && a(\Upsilon^e, \Upsilon^e) := (a_{ij})_{i,j\in S_e}, && a(\Upsilon^e, \Upsilon^z) := (a_{ij})_{i\in S_e, j\in S_z},\\
&a(\Upsilon^z, \Upsilon^a) := (a_{ij})_{i\in S_z, j\in S_a}, && a(\Upsilon^z, \Upsilon^e) := (a_{ij})_{i\in S_z, j\in S_e}, && a(\Upsilon^z, \Upsilon^z) := (a_{ij})_{i,j\in S_z}.
\end{aligned}$$

Analogously to the biquadratic case the interaction of basisfunctions corresponding to a class of nodes is represented by these nine parts of the stiffness matrix A_h^3. After possible reordering this interplay can be seen:

$$A_h^3 = \begin{pmatrix} a(\Upsilon^a, \Upsilon^a) & a(\Upsilon^a, \Upsilon^e) & a(\Upsilon^a, \Upsilon^z) \\ a(\Upsilon^e, \Upsilon^a) & a(\Upsilon^e, \Upsilon^e) & a(\Upsilon^e, \Upsilon^z)) \\ a(\Upsilon^z, \Upsilon^a) & a(\Upsilon^z, \Upsilon^e) & a(\Upsilon^z, \Upsilon^z) \end{pmatrix}. \tag{3.12}$$

In the following, if not explicitly stated, for the indices i, j, r it holds that $i \in S_a$, $j \in S_e$ and $r \in S_z$. We recap and extend the index sets from section 3.2:

$$\begin{aligned}
S_i &= \{l \mid \operatorname{supp}\gamma_i \cap \operatorname{supp}\gamma_l \neq \emptyset,\ l \neq i,\ l \in S\} \\
S_i^e &= \{l \mid \operatorname{supp}\gamma_i \cap \operatorname{supp}\gamma_l \neq \emptyset,\ l \in S_e\} \\
S_i^z &= \{l \mid \operatorname{supp}\gamma_i \cap \operatorname{supp}\gamma_l \neq \emptyset,\ l \in S_z\} \\
R_j &= \{l \mid \operatorname{supp}\gamma_i \cap \operatorname{supp}\gamma_l \neq \emptyset,\ l \neq j,\ l \in S\} \\
R_j^a &= \{l \mid \operatorname{supp}\gamma_i \cap \operatorname{supp}\gamma_l \neq \emptyset,\ l \in S_a\} \\
R_j^e &= \{l \mid \operatorname{supp}\gamma_i \cap \operatorname{supp}\gamma_l \neq \emptyset,\ l \neq j,\ l \in S_e\} \\
R_j^z &= \{l \mid \operatorname{supp}\gamma_i \cap \operatorname{supp}\gamma_l \neq \emptyset,\ l \in S_z\} \\
E_r &= \{l \mid \operatorname{supp}\gamma_i \cap \operatorname{supp}\gamma_l \neq \emptyset,\ l \neq k,\ l \in S\} \\
E_k^a &= \{l \mid \operatorname{supp}\gamma_i \cap \operatorname{supp}\gamma_l \neq \emptyset,\ l \in S_a\} \\
E_k^e &= \{l \mid \operatorname{supp}\gamma_i \cap \operatorname{supp}\gamma_l \neq \emptyset,\ l \in S_e\} \\
E_k^z &= \{l \mid \operatorname{supp}\gamma_i \cap \operatorname{supp}\gamma_l \neq \emptyset,\ l \neq k,\ l \in S_z\}
\end{aligned}$$

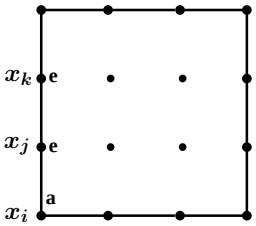
Finally we use the same idea as in the biquadratic case: Let $\{\psi_i\}_{i \in S_a}$ be the nodal basisfunctions of the bilinear FEM space A_h^1 on R^h. As a consequence of $V_h^1 \subset V_h^3$, (3.11), (3.5) and the compact support properties of $\{\psi_i\}_{i \in S_a}$ we get

$$\begin{aligned}
\psi_i(x) = \gamma_i(x) + &\sum_{(j,k) \in S_i^{e12}} \left(\frac{2}{3}\gamma_j(x) + \frac{1}{3}\gamma_k(x)\right) + \dots \\
&\sum_{(r,s,t,u) \in S_i^{z123}} \left(\frac{4}{9}\gamma_r(x) + \frac{2}{9}\gamma_s(x) + \frac{2}{9}\gamma_t(x) + \frac{1}{9}\gamma_u(x)\right), \qquad i \in S_a,
\end{aligned} \tag{3.13}$$

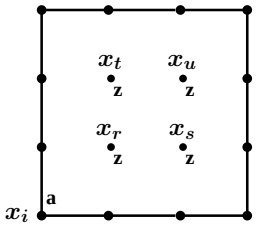
where $S_i^{e12} \subseteq S_e$ is the set of indices (j,k) related to two type-e nodes x_j and x_k with the property that x_j and x_k are on the same neighboring edge of x_i (i.e. $\psi_i(x_j) \neq 0$ and $\psi_i(x_k) \neq 0$) and x_j is geometrically closer to x_i than x_k (see Fig. 3.8a). The second set S_i^{z123} is the set of indices (r,s,t,u) related to four type-z nodes x_r, x_s, x_t and x_u in which these are in the interior of one neighboring element where

- x_r is geometrically the closest,
- x_s and x_t are farther away and
- x_u is the farthest away

compared to x_i (see Fig. 3.8b). We neglect to distinguish between x_s and x_t as it is not necessary for our algorithm.



(a) x_j, x_k on a neighboring edge of x_i



(b) x_r, x_r, x_s and x_t on a neighboring element of x_i

Figure 3.8: Geometry in S_i^{e12} and S_i^{z123}

Let $\Psi := (\psi_{i_1}, \ldots, \psi_{i_{N_a}})^\top$, $i_p \in S_a$ and rewrite (3.13) as

$$\Psi = P_h^H \begin{pmatrix} \Upsilon^a \\ \Upsilon^e \\ \Upsilon^z \end{pmatrix}, \tag{3.14}$$

where P_h^H is an $N_a \times N$ matrix.

Then P_h^H defines our *restriction operator* from V_h^3 to V_h^1 and $P_H^h := (P_h^H)^\top$ defines our *prolongation operator* from V_h^1 to V_h^3. The coarse grid operator is constructed using (3.12), (3.14) and the Galerkin condition:

$$A_h^1 := (a(\Psi, \Psi)) = P_h^H A_h^3 P_H^h \tag{3.15}$$

Interpolatory Sets

In order to develop an AMG method for the problem (3.9), we need to solve the following two problems by algebraic means.

Problem *C*1: Find the sets S_a, S_e and S_z from the coefficient matrix A_h^3.
Problem *C*2: Find the interpolatory sets S_i^{e12} and S_i^{z123} for any fixed $i \in S_a$.

3.3.1 Algorithm for Problem $C1$ – Find S_a, S_e and S_z

We assume that all nodes in X_a satisfy Assumption 3.4 (see Fig. 3.10) and the Nonzero Assumption 1.2.

Hence, for any $i \in S_a$, $j \in S_e$ and $r \in S_z$, $|S_i|+1$, $|R_j|+1$ and $|E_r|+1$ are exactly the nonzero entries of the i-th, j-th and r-th row in the matrix A_h^3, respectively.

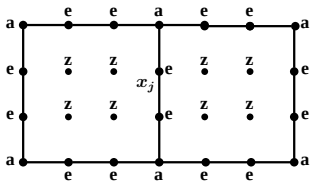
By implication (1.3) and the compact support properties of the basisfunctions γ_i we are able to split the set S. This is done by counting the neighbors in Fig. 3.9a and Fig. 3.10.

Proposition 3.16: For a given grid R^h, under the Nonzero Assumption 1.2, we have

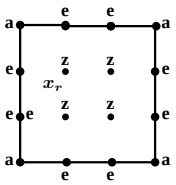
$$|S_i| \geq 32, \qquad \text{for any } i \in S_a$$

and

$$|R_j| \leq 27, \qquad \text{for any } j \in S_e,$$
$$|E_r| \leq 15, \qquad \text{for any } r \in S_z.$$

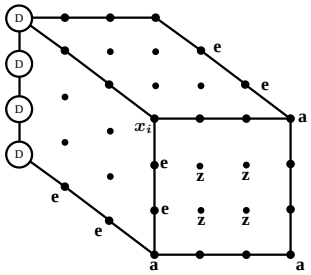


(a) Neighbors of $x_j \in X_e$

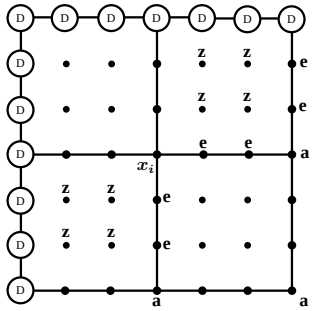


(b) Neighbors of $x_r \in X_z$

Figure 3.9: Maximal Number of neighbors for elements of X_e and X_e; $p = 3$



(a) Three neighboring elements, two half-interior edges, one interior element; $p = 3$



(b) Four neighboring elements, one interior element; $p = 3$

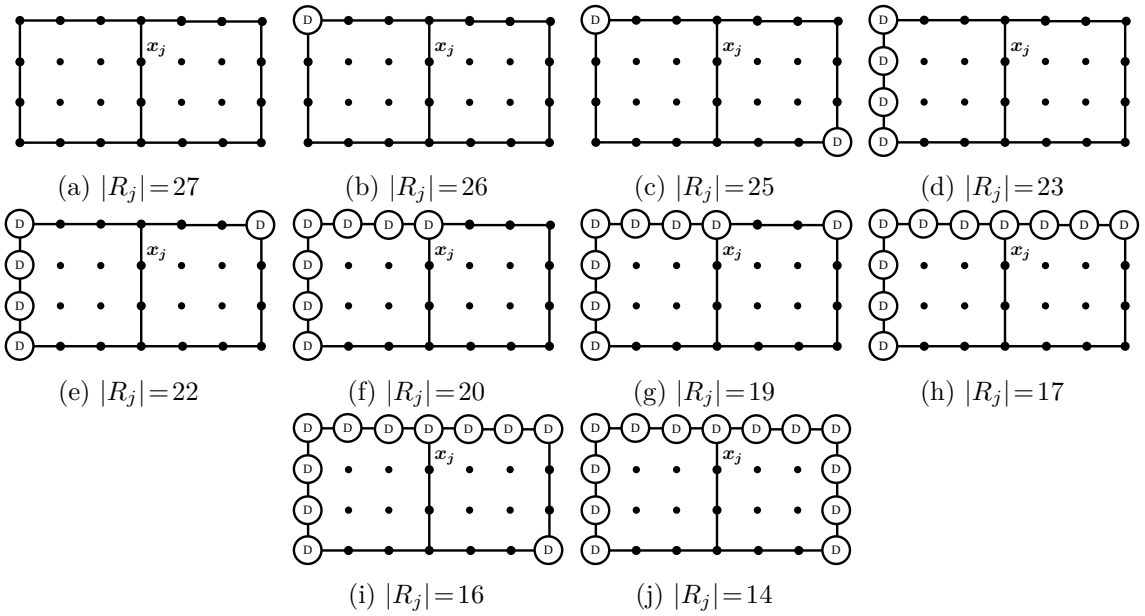
Figure 3.10: Minimal number of neighbors for elements of X_a under Assumption 3.4 for $p = 3$

Therefore we are able to uniquely identify type-a nodes among all other nodes in a similar way as in the biquadratic case.

Criterion 3.17: For any index $i \in S$, if there are more than 27 nonzero entries in the row i of the matrix A_h^3, then the node x_i related to the index i is a type-a node. ■

Parallel 3.18: By Criterion 3.17 the process of checking if a row i corresponds to a type-a node is independent of all the other rows $j \neq i$ in the matrix A_h^3. Hence this process can be performed in parallel. If the matrix is stored in the CSR format (cf. section 1.2) the complexity of this task is $\mathcal{O}(N)$. ■

With the purpose of distinguishing between type-e and type-z nodes we investigate certain types of neighborhoods which represent up to permutation all possible ones. In particular their layout on the Dirichlet-boundary is of interest.

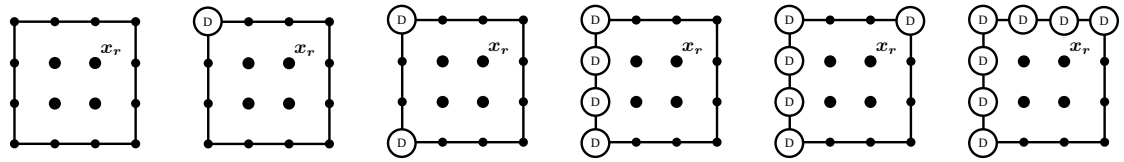


(a) $|R_j|=27$ (b) $|R_j|=26$ (c) $|R_j|=25$ (d) $|R_j|=23$

(e) $|R_j|=22$ (f) $|R_j|=20$ (g) $|R_j|=19$ (h) $|R_j|=17$

(i) $|R_j|=16$ (j) $|R_j|=14$

Figure 3.11: Neighborhoods of type-e nodes



(a) $|E_k|=15$ (b) $|E_k|=14$ (c) $|E_k|=13$ (d) $|E_k|=11$ (e) $|E_k|=10$ (f) $|E_k|=8$

Figure 3.12: Neighborhoods of type-z nodes

Similar to the biquadratic case we see in Fig. 3.11 and Fig. 3.12 that the number of nonzero entries in a row (i.e. $|R_j|$ or $|E_r|$) is no unique identifier on its own. In particular the cardinality 14 appears for both types of sets . We distinguish these cases by checking the number of type-a nodes in their neighborhood. Hence, we present the following proposition.

Proposition 3.19: For a given grid R^h, under the Nonzero Assumption 1.2, we have for any $j \in S_e$

1. $16 \leq |R_j| \leq 27$ (see Fig. 3.11a-i),
2. $|R_j| = 14$, $|R_j^a| = 1$ (see Fig. 3.11j),

and for any $r \in S_z$,

1. $|E_r| < 14$ (see Fig. 3.12c-f),
2. $|E_r| = 14$, $|E_r^a| > 1$ (see Fig. 3.12b),
3. $|E_r| = 15$ (see Fig. 3.12a).

Thus, by proposition 3.19 we get the following criterion to distinguish between type-e and

type-z nodes.

Criterion 3.20: For any index $i \in S$, in the row i of matrix A_h^3, if one of the following conditions holds, then the node x_i related to the index i belongs to type-z.

1. There are less than 14 nonzero entries or just 15 nonzero entries.
2. There are just 14 nonzero entries and and among them are more than one nonzero entries which belong to type-a. ∎

With the last to criteria 3.17 and 3.20 we are able to identify type-a and type-z nodes. The remaining nodes are of type-e. In order to associate all nodes with their type we introduce the following flag array $I_{aez}(k)$, $k = 1, \ldots, N$ for the matrix A_h^3, which satisfies

$$I_{aez}(k) := \begin{cases} 1, & x_k \in X_a, \\ 2, & x_k \in X_z, \\ 3, & x_k \in X_e. \end{cases}$$

Using the flag array, we can easily obtain an algorithm to get $N_a,\ N_e,\ N_z,\ N$ and the index sets $S_a,\ S_e,\ S_z$.

As a final step we have to identify the neighbor index sets S_i^e, S_i^z, R_j^e, R_j^z, E_r^e and E_r^z. To do so the following criterion 3.21 will be used.

Criterion 3.21: For any fixed index $i \in S_a$, $j \in S_e$ and $r \in S_z$ under the Nonzero Assumption 1.2, there holds:

1. The index $l \in S_i^e$ or $l \in S_i^z$, iff $l \in S_e$ or $l \in S_z$, respectively, and $\operatorname{supp}\gamma_i \cap \operatorname{supp}\gamma_l \neq \emptyset$.
2. The index $l \in R_j^e$, iff $l \neq j$ and $l \in S_e$ and $\operatorname{supp}\gamma_i \cap \operatorname{supp}\gamma_l \neq \emptyset$.
3. The index $l \in R_j^z$, iff $l \neq j$ and $l \in S_z$ and $\operatorname{supp}\gamma_i \cap \operatorname{supp}\gamma_l \neq \emptyset$.
4. The index $l \in E_r^e$, iff $l \neq r$ and $l \in S_e$ and $\operatorname{supp}\gamma_i \cap \operatorname{supp}\gamma_l \neq \emptyset$.
5. The index $l \in E_r^z$, iff $l \neq r$ and $l \in S_z$ and $\operatorname{supp}\gamma_i \cap \operatorname{supp}\gamma_l \neq \emptyset$. ∎

Using criterion 3.21 we can obtain the above index sets which is analogous to algorithm 3.2. Instead of only for type-a nodes the algorithm is now performed for all three types and we have to distinguish between type-e and type-z neighbors. This generalization is straightforward and the same data structures can be used.

3.3.2 Algorithm for Problem $C2$ on squares – Find S_i^{e12} and S_i^{z123}

For any $i \in S_a$ we have to identify the interpolatory sets S_i^{e12} and S_i^{z123}. As before we generalize the definitions and methods derived in [31].

The size and structure of the neighboring elements will not be enough to uniquely distinguish between nodes which lie on the same edge or on the same element. We will need to utilize more information from the problem. The following section will solve problems of type (1.1) where for simplicity we restrict $d(x)$ to be piecewise constant.

By geometric consideration illustrated in Fig. 3.8 we motivate the following definitions. For any $i \in S_a$ we set

$$S_i^{ee} = \{(j,k) \mid R_j = R_k \subseteq S_i,\ j,k \in S_e\},$$
$$S_i^{zz} = \{(r,s,t,u) \mid E_r = E_s = E_t = E_u \subseteq S_i,\ r,s,t,u \in S_z\}$$

where (j,k) and (k,j) are viewed as the same pair. Analogous for (r,s,t,u) and its permutations.

The construction of theses sets is easily done by considering the relations of the supports of different basisfunctions. Functions which are associated with nodes that lie on the same edge or element have the same support.

Criterion 3.22: For any $i \in S_a$ under the Nonzero Assumption 1.2 there holds, that the index $(j,k) \in S_i^{ee}$ iff $\operatorname{supp}\gamma_j = \operatorname{supp}\gamma_k \subseteq \operatorname{supp}\gamma_i$. Under the same assumptions there holds, that $(r,s,t,u) \in S_i^{zz}$, iff $\operatorname{supp}\gamma_r = \operatorname{supp}\gamma_s = \operatorname{supp}\gamma_t = \operatorname{supp}\gamma_u \subseteq \operatorname{supp}\gamma_i$. ∎

In order to generate S_i^{ee} we need to identify all nodes which are part of an edge that does not lie on the Dirichlet boundary. In the following algorithm 3.5 we denote this set be $\mathcal{E}$. Afterwards pairs of nodes which are part of one and the same edge are identified. Note that it is enough to compare the type-z neighbors of a type-e node to check if they share the same support (see line 11).

Algorithm 3.5 Generate S_i^{ee} for $p = 3$

```
1:  S_i^ee ← ∅
2:  E ← ∅
3:  for all j ∈ S_i^e do                          ▷ type-e neighbors
4:      if R_j ⊆ S_i then                         ▷ Part of an edge
5:          E ← E ∪ {j}
6:      end if
7:  end for
8:  while E ≠ ∅ do
9:      j ∈ E
10:     for all k ∈ E \ {j} do
11:         if R_j^z == R_k^z then                ▷ Same support
12:             S_i^ee ← S_i^ee ∪ (j,k)
13:             E ← E \ {j,k}
14:             break
15:         end if
16:     end for
17: end while
```

Algorithm 3.6 describes the generation of S_i^{zz} for a fixed $i \in S_a$. Note, that the support of type-z functions is given by only one element and therefore is $E_j \subseteq S_i$ for all $j \in S_i^z$. As every element has exactly four type-z nodes, which cannot lie on the Dirichlet boundary, the set E_j^z will always consist of exactly three elements.

Parallel 3.23: Algorithms 3.5 and 3.6 can be performed in parallel for every $i \in S_a$ as there is no dependence between S_i^{ee} or S_i^{zz} for different type-a nodes i. The algorithms themselves are iterative, as the set $\mathcal{E}$ changes during each iteration. ∎

The only remaining task is to identify the correct order in the pairs (j,k) and quadruples (r,s,t,u) which corresponds to the correct geometric position of the associated basisfunctions.

Algorithm 3.6 Generate S_i^{zz} for $p = 3$

1: $S_i^{zz} \leftarrow \emptyset$
2: $\mathcal{E} \leftarrow S_i^z$
3: **while** $\mathcal{E} \neq \emptyset$ **do**
4: $\quad j \in \mathcal{E}$ ▷ type-z neighbor of i
5: $\quad E_j^z \leftarrow \{r, s, t\}$ ▷ Support is just one element
6: $\quad S_i^{zz} \leftarrow S_i^{zz} \cup (r, s, t, j)$
7: $\quad \mathcal{E} \leftarrow \mathcal{E} \setminus (E_j^z \cup \{j\})$
8: **end while**

In order to make the presentation of the theory clearer we assume a mesh of squares first. This assumption prevents the introduction of many similar cases. Furthermore it enables us to use a heuristic based on the symmetry of our basisfunctions to identify the nodes s and t which correspond to the basisfunctions with the value 2/9 in the linear combination of $\psi(x)$ in (3.13). The differences to the general case are discussed in section 3.3.3.

Criterion 3.24: Let R^h be a mesh of squares. For any $i \in S_a$, $(r, s, t, u) \in S_i^{zz}$ and piecewise constant function $d(x)$ holds that $a_{i,s} = a_{i,t}$. ■

Proof: All basisfunctions $\gamma_r, \gamma_s, \gamma_t$ and γ_u share the same support $\omega \in \Omega$. On this element we denote the constant parts of $d(x)$ as d_ω. Using the notation of section 1.1 we will show that

$$\begin{aligned} a_{i,s} = a(\gamma_i, \gamma_s) &= \int_\omega d_\omega \nabla\gamma_i(x,y)\nabla\gamma_s(x,y)\, d(x,y) \\ &\overset{!}{=} d_\omega \int_\omega \nabla\gamma_i(x,y)\nabla\gamma_t(x,y)\, d(x,y) = a_{i,t}. \end{aligned}$$

First we consider the construction of bicubic basisfunctions (cf. section 1.4). We need four reference basisfunctions $\hat{N}_i(\xi)$, $i = 1, \ldots, 4$ and $\hat{M}_j(\eta)$, $i = 1, \ldots, 4$ for each axis. It holds that $\hat{M}_j(\eta) = \hat{N}_i(\eta)$. Then the basisfunctions are constructed as

$$N_{i,j}(\xi, \eta) = \hat{N}_i(\xi)\hat{M}_j(\eta) \qquad i, j = 1, \ldots, 4.$$

It is enough to consider the case of the reference square, as a transformation into a different square does not change any proportions. WLOG we set associate $\gamma_i(x, y)$ with $N_{1,1}(x, y)$, $\gamma_s(x, y)$ with $N_{3,4}(x, y)$ and $\gamma_t(x, y)$ with $N_{4,3}(x, y)$. Note that the points $(3, 4)$ and $(4, 3)$ correspond to the points on the element which do not lie on the diagonal through point $(1, 1)$. Finally we obtain

$$\begin{aligned} \int_\omega \nabla N_{1,1}(x,y)\nabla N_{3,4}(x,y)\, d(x,y) &= \int_\omega \nabla\left(N_1(x)M_1(y)\right)\nabla\left(N_3(x)M_4(y)\right)\, d(x,y) \\ &= \int_\omega \nabla\left(M_1(x)N_1(y)\right)\nabla\left(M_3(x)N_4(y)\right)\, d(x,y) \\ &= \int_\omega \nabla N_{1,1}(\tilde{x},\tilde{y})\nabla N_{4,3}(\tilde{x},\tilde{y})\, d(\tilde{x},\tilde{y}) \end{aligned}$$

where $N_i(x), M_j(y)$ denote onto the square ω transformed functions. The last equality holds because the integration boundaries for x and y are the same. □

By criterion 3.24 we can obtain an algebraic algorithm to identify the nodes s and t in every quadruple $(r,s,t,u) \in S_i^{zz}$.

In the following x is a shorthand for the vector $(x,y)^\top$. Let $i \in S_a$ be fixed and $\psi_i(x)$ be the corresponding bilinear basisfunction. We define n_i and m_i as the number of elements in the set S_i^{ee} and S_i^{zz}, respectively. Motivated by (3.13) and assuming the knowledge of the position of the nodes s and t by criterion 3.24, we can represent $\psi_i(x)$ as

$$\begin{aligned}\psi_i(x) = \gamma_i(x) + \sum_{n=1}^{n_i} \alpha_n \gamma_{j_n}(x) + (1-\alpha_n)\gamma_{k_n}(x) + \dots \\ \sum_{m=1}^{m_i} \beta_m \gamma_{r_m}(x) + (1-\beta_m)\gamma_{u_m}(x) + \frac{2}{9}\left(\gamma_{s_m}(x) + \gamma_{t_m}(x),\right)\end{aligned} \tag{3.16}$$

where (j_n, k_n) is the n-th index in S_i^{ee}, (r_m, s_m, t_m, u_m) is the m-th index in S_i^{zz} and the variables $\alpha_l, \beta_m \in [0,1]$ for $n = 1, \dots, n_i$, $m = 1, \dots, m_i$ are the unknown parameters.

We are going to analyze the equation (3.16) with respect to $\{\alpha_n\}_{n=1}^{n_i}$ and $\{\beta\}_{m=1}^{m_i}$. By reordering we get the following representation:

$$\begin{aligned}\psi_i(x) &= \underbrace{\gamma_i(x) + \sum_{n=1}^{n_i} \gamma_{k_n} + \sum_{m=1}^{m_i} \frac{2}{9}\left(\gamma_{s_m}(x) + \gamma_{t_m}(x)\right)}_{=:\varphi(x)} + \dots \\ &\quad \sum_{n=1}^{n_i} \alpha_n \underbrace{\left(\gamma_{j_n} - \gamma_{k_n}\right)}_{=:\tilde{\varphi}_n(x)} + \sum_{m=1}^{m_i} \beta_m \underbrace{\left(\gamma_{r_m} - \gamma_{u_m}\right)}_{=:\theta_m(x)} \\ &= \varphi(x) + \sum_{n=1}^{n_i} \alpha_n \tilde{\varphi}_n(x) + \sum_{m=1}^{m_i} \beta_m \theta_m(x).\end{aligned} \tag{3.17}$$

Let $\lambda := (\alpha_1, \dots, \alpha_{n_i}, \beta_1, \dots, \beta_{m_i})^\top$ and

$$J(\lambda) = a(\psi_i, \psi_i), \tag{3.18}$$

where $a(\cdot,\cdot)$ is the bilinear form of our model problem (1.1). In order to simplify the notation we define $M_i := n_i + m_i$, $\eta_j := \tilde{\varphi}_j$ for $j = 1, \dots, n_i$ and $\eta_j := \theta_{j-n_i}$ for $j = n_i+1, \dots, M_i$. This results in a shorter notation for (3.17):

$$\psi_i(x) = \varphi(x) + \sum_{j=1}^{M_i} \lambda_j \eta_j(x). \tag{3.19}$$

With this notation we introduce the following minimization problem: Find $\lambda^* \in \mathbb{R}^{M_i}$, such that

$$J(\lambda^*) = \min_{\lambda \in \mathbb{R}^{M_i}} J(\lambda). \tag{3.20}$$

By inserting (3.19) into (3.18), we obtain

$$\begin{aligned} J(\lambda) &= a(\varphi(x) + \sum_{j=1}^{M_i} \lambda_j \eta_j(x), \varphi(x) + \sum_{j=1}^{M_i} \lambda_j \eta_j(x)) \\ &= \sum_{j=1}^{M_i} \sum_{m=1}^{M_i} \lambda_j \lambda_m a(\eta_j, \eta_m) + 2 \sum_{j=1}^{M_i} \lambda_j a(\eta_j, \varphi) + a(\varphi, \varphi). \end{aligned} \tag{3.21}$$

Theorem 3.25 [31]: There exists a unique solution vector $\lambda^* = (\alpha_1^*, \ldots, \alpha_{n_i}^*, \beta_1^*, \ldots, \beta_{m_i}^*)^\top$ of the minimization problem (3.20). Moreover, the solution

$$\begin{aligned} \psi_i^*(x) = \gamma_i(x) &+ \sum_{n=1}^{n_i} \alpha_n^* \gamma_{j_n}(x) + (1 - \alpha_n^*)\gamma_{k_n}(x) + \ldots \\ &\sum_{m=1}^{m_i} \beta_m^* \gamma_{r_m}(x) + (1 - \beta_m^*)\gamma_{u_m}(x) + \frac{2}{9}\left(\gamma_{s_m}(x) + \gamma_{t_m}(x)\right) \end{aligned} \tag{3.22}$$

must be the basisfunction $\psi_i(x)$ of the bilinear finite element related to the node x_i.

Proof: The proof is analogous to [31, p. 368].

First we prove the existence and uniqueness of a solution of the minimization problem (3.20). As the functions $\{\eta_j(x)\}_{j=1}^{M_i}$ have all compact support, it is easy to derive their linear independence. We skip this step here. Looking closely at (3.21) we see that $J(\lambda)$ is a quadratic function which can be written as

$$J(\lambda) = \lambda^\top \hat{A} \lambda + b^\top \lambda + c$$

with the SPD-matrix $\hat{A} = (\hat{a}_{jm})_{n_i \times m_i}$, $\hat{a}_{jm} = a(\eta_j, \eta_m)$, the vector $b = (b_j)_{n_i}$, $b_j = a(\eta_j, \varphi)$ and $c = a(\varphi, \varphi)$. Applying the results of multivariate optimization (see e.g. [9, chapter 22]), we conclude that there exists a unique solution vector λ^* with

$$\hat{A}\lambda^* = \hat{F} \tag{3.23}$$

where $\hat{F} = -b$. For the following discussion the following equivalent form is used:

$$a(\eta_j, \psi_i^*) = 0, \qquad j = 1, \ldots, M_i \tag{3.24}$$

where $\psi_i^*(x) = \varphi(x) + \sum_{j=1}^{M_i} \lambda_j^* \eta_j(x)$. Due to the fact that the bilinear basisfunction $\psi_i(x)$ associated with the node x_i can be expressed as (3.17), the only thing we need to verify in order to proof theorem 3.25, is that the nodal basisfunction $\psi_i(x)$ satisfies (3.24).

We are going to discuss the function $\tilde{\varphi}$ and θ separately. Let e_l, $l = 1, \ldots, n_i$ be all rectangular elements neighboring node x_i. By the compact support property of ψ_i, for any fixed j, $j = 1, \ldots, n_i$ we get

$$a(\tilde{\varphi}_j, \psi_i) = \sum_{l=1}^{n_i} \int_{e_l} d_{e_l} \nabla\tilde{\varphi}_j \nabla\psi_i \, d(x, y) \tag{3.25}$$

$$= d_{e_{l_1}} \int_{e_{l_1}} \nabla\tilde{\varphi}_j \nabla\psi_i \, d(x, y) + d_{e_{l_2}} \int_{e_{l_2}} \nabla\tilde{\varphi}_j \nabla\psi_i \, d(x, y), \tag{3.26}$$

where $\tilde{\varphi}_j$ is defined by (3.17) and e_{l_1}, e_{l_2} are two elements neighboring to the edge $e_{(j_l,k_l)}$ with the starting point x_{j_l} and the ending point x_{k_l}.

For the next step note that $\psi_i(x)$ is linear with respect to each of its variables. Using Green's formula, we obtain

$$0 = \int_{e_l} \Delta\psi_i\tilde{\varphi}_j \, d(x,y) = \int_{\partial e_l} \frac{\partial\psi_i}{\partial n}\tilde{\varphi}_j \, ds - \int_{e_l} \nabla\tilde{\varphi}_j\nabla\psi_i \, d(x,y), \tag{3.27}$$

where n is the unit outward normal vector and ∂e_{l_1} is the boundary of the rectangular element e_{l_1}.

Substituting the definition of $\tilde{\varphi}$ from (3.17) into (3.27) and using the compact support property of $\tilde{\varphi}$ we obtain

$$\int_{e_l} \nabla\tilde{\varphi}_j\nabla\psi_i \, d(x,y) = \int_{\partial e_l} \frac{\partial\psi_i}{\partial n}\tilde{\varphi}_j \, ds = \int_{e_{(j_l,k_l)}} \frac{\partial\psi_i}{\partial n}\tilde{\varphi}_j \, ds \tag{3.28}$$

$$= \frac{\partial\psi_i}{\partial n}\int_{e_{(j_l,k_l)}} \tilde{\varphi}_j \, ds = \frac{\partial\psi_i}{\partial n}\int_{e_{(j_l,k_l)}} (\gamma_{j_l} - \gamma_{k_l}) \, ds, \tag{3.29}$$

where we use, that $\frac{\partial\psi_i}{\partial n}|_{e_{(j_l,k_l)}}$ is a constant, because $\psi_i|_{e_{(j_l,k_l)}}$ is a linear polynomial.

Computing the last integral explicitly we get

$$\int_{e_{(j_l,k_l)}} (\gamma_{j_l} - \gamma_{k_l}) \, ds = C\int_{-1}^{1} \xi(1+\xi)(1-\xi) \, d\xi = 0,$$

where C is a constant, which depends on the length $|e_{(j_l,k_l)}|$ of the edge $e_{(j_l,k_l)}$ and the height of the bicubic basisfunctions.

Substituting the previous result back into (3.28) for e_{l_1} and e_{l_2}, we get

$$\int_{e_{l_1}} \nabla\tilde{\varphi}_j\nabla\psi_i \, d(x,y) = 0 \qquad \text{and} \qquad \int_{e_{l_2}} \nabla\tilde{\varphi}_j\nabla\psi_i \, d(x,y) = 0.$$

Substituting these results into (3.25), we get

$$a(\tilde{\varphi}_j, \psi_i) = 0, \qquad j = 1, \ldots, n_i.$$

In the case of $j = n_i + 1, \ldots, M_i$ equation (3.25) simplifies even further:

$$a(\theta_j, \psi_i) = d_{e_l}\int_{e_l} \nabla\theta_j\nabla\psi_i \, d(x,y)$$

where θ_j is define by (3.17) and e_l is the element $e_{(r_l,s_l,t_l,u_l)}$ with the four vertices x_{r_l}, x_{s_l}, x_{t_l} and x_{u_l}.

Analogously we apply the result (3.27) and use the compact support property of θ_j to derive

$$\int_{e_l} \nabla\theta_j\nabla\psi_i \, d(x,y) = \int_{e_{(j_l,k_l)}} \frac{\partial\psi_i}{\partial n}\theta_j \, ds = 0. \tag{3.30}$$

The last equality holds, because $\theta(x)|_{\partial e_l} \equiv 0$, due to its definition $\theta(x) = \gamma_{r_m}(x) - \gamma_{u_m}(x)$ and the fact that $\gamma_{r_m}(x)$ and $\gamma_{u_m}(x)$ are zero on the boundary (cf. section 1.4).

Therefore we get the same result as in the previous case:

$$a(\theta_j, \psi_i) = 0, \qquad j = n_i + 1, \dots, M_i.$$

This completes the proof of theorem 3.25. □

For any fixed $i \in S_a$ we know from theorem 3.25 how to assemble linear system (3.23). First we investigate the structure of the coefficient matrix $\hat{A}$ and the the right hand side vector $\hat{F}$. In the following we will use the index sets $I_\alpha := \{1, \dots, n_i\}$ and $I_\beta := \{n_i + 1, \dots, M_i\}$ as a shorthand. Furthermore we define the vectors $\tilde{\varphi} = (\tilde{\varphi}_1, \dots, \tilde{\varphi}_{n_i})^\top$ and $\theta = (\theta_1, \dots, \theta_{m_i})^\top$ which are used in the following matrix notation:

$$\begin{aligned} a(\tilde{\varphi}, \tilde{\varphi}) &= (\hat{a}_{jm})_{j,m \in I_\alpha}, & a(\tilde{\varphi}, \varphi) &= (a(\tilde{\varphi}_j, \varphi))_{j \in \{1,\dots,n_i\}}, \\ a(\tilde{\varphi}, \theta) &= (\hat{a}_{jm})_{j \in I_\alpha, m \in I_\beta}, & a(\theta, \varphi) &= (a(\theta_j, \varphi))_{j \in \{1,\dots,m_i\}}, \\ a(\theta, \theta) &= (\hat{a}_{jm})_{j,m \in I_\beta}. \end{aligned}$$

By these definitions we see, that

$$\hat{A} = \begin{pmatrix} a(\tilde{\varphi}, \tilde{\varphi}) & a(\tilde{\varphi}, \theta) \\ a(\theta, \tilde{\varphi}) & a(\theta, \theta) \end{pmatrix}, \qquad \hat{F} = \begin{pmatrix} -a(\tilde{\varphi}, \varphi) \\ -a(\theta, \varphi) \end{pmatrix}.$$

Due to symmetry of the bilinear form it holds, that $a(\tilde{\varphi}, \theta) = a(\theta, \tilde{\varphi})$. The entries of the coefficient matrix $\hat{A}$ and the entries of the right hand side vector $\hat{F}$ can be expressed as

$$\hat{a}_{\mathfrak{l}\mathfrak{m}} = a(\eta_{\mathfrak{l}}, \eta_{\mathfrak{m}}) = \begin{cases} a(\tilde{\varphi}_l, \tilde{\varphi}_m) = a_{j_l,j_m} - a_{j_l,k_m} - a_{k_l,j_m} + a_{k_l,k_m}, & \mathfrak{l}, \mathfrak{m} \in I_\alpha, \\ a(\tilde{\varphi}_l, \theta_m) = a_{j_l,r_m} - a_{j_l,u_m} - a_{k_l,r_m} + a_{k_l,u_m}, & \mathfrak{l} \in I_\alpha, \mathfrak{m} \in I_\beta, \\ a(\theta_l, \theta_m) = a_{r_l,r_m} - a_{r_l,u_m} - a_{u_l,r_m} + a_{u_l,u_m}, & \mathfrak{l}, \mathfrak{m} \in I_\beta, \end{cases}$$

$$\begin{aligned} \hat{f}_l = -a(\tilde{\varphi}_l, \varphi) = -\Bigg[& (a_{j_l,i} - a_{k_l,i}) + \sum_{n=1}^{n_i} (a_{j_l,k_n} - a_{k_l,k_n}) + \dots \\ & \sum_{m=1}^{m_i} \left(\frac{5}{9} (a_{j_l,u_m} - a_{k_l,u_m}) + \frac{2}{9} (a_{j_l,s_m} - a_{k_l,s_m}) + \frac{2}{9} (a_{j_l,t_m} - a_{k_l,t_m}) \right) \Bigg], \quad l \in \{1, \dots, n_i\}, \end{aligned} \tag{3.31}$$

$$\begin{aligned} \hat{f}_{l+n_i} = -a(\theta_l, \varphi) = -\Bigg[& (a_{r_l,i} - a_{u_l,i}) + \sum_{n=1}^{n_i} (a_{r_l,k_n} - a_{u_l,k_n}) + \dots \\ & \sum_{m=1}^{m_i} \left(\frac{5}{9} (a_{r_l,u_m} - a_{u_l,u_m}) + \frac{2}{9} (a_{r_l,s_m} - a_{u_l,s_m}) + \frac{2}{9} (a_{r_l,t_m} - a_{u_l,t_m}) \right) \Bigg], \ l \in \{1, \dots, m_i\}, \end{aligned}$$

where a_{nm}, $n, m = 1, \dots, N$ are the entries of the coefficient matrix A_h^3 in (3.9). As the enumeration of θ_m starts with 1 and not with $n_i + 1$, the indices have to be converted. This is shown by the use of different indices $\mathfrak{l}, \mathfrak{m}$ for l, m. The mapping is given by

$$l = \begin{cases} \mathfrak{l} & \text{if } \mathfrak{l} \in I_\alpha, \\ \mathfrak{l} - n_i & \text{if } \mathfrak{l} \in I_\beta, \end{cases}$$

and analogous for m and $\mathfrak{m}$.

In the following we describe an algorithm to implement the setup of the linear system (3.23).

A closer inspection of (3.31) shows a pattern in the definition of the entries of matrix $\hat{A}$ and right hand side vector $\hat{F}$. We are giving an algorithm which uses this pattern in order to neglect the distinction of many different cases. By doing so it is possible to alter this algorithm with very little changes to a general mesh (chapter 3.3.3).

For every element j_l, k_l where $(j_l, k_l) \in S_i^{e12}$ and r_l, u_l where $(r_l, *, *, u_l) \in S_i^{z123}$ we define a mapping from its global index in A_h^3 to the row $\mathfrak{l}$ that is influenced by it via formula (3.31). Additionally it is important to save its current position, i.e. if it is the first or last element. We will indicate the first position with a positive row-index $\mathfrak{l}$ and the last position with a negative row-index $\mathfrak{l}$. For this purpose we define the array $\mathcal{M}_i(j)$, $j = 1, \ldots, N$. All values which are not effected are set to zero.

Definition 3.26: For any fixed $i \in S_a$ the array $\mathcal{M}_i(j)$, $j = 1, \ldots, N$ is defined as follows:

$$\begin{array}{rcllll} (j_l, k_l) \in S_i^{e12} & : & \mathcal{M}_i(j_l) = l & \text{and} & \mathcal{M}_i(k_l) = -l & \forall\, l \in \{1, \ldots, n_i\} \\ (r_l, *, *, u_l) \in S_i^{z123} & : & \mathcal{M}_i(r_l) = l + n_i & \text{and} & \mathcal{M}_i(u_l) = -(l + n_i) & \forall\, l \in \{1, \ldots, m_i\} \end{array}$$

All other indices are set to zero. □

Every entry in $\hat{A}$ is defined over a formula like $a_{j_l,j_m} - a_{j_l,k_m} - a_{k_l,j_m} + a_{k_l,k_m}$ which can also be written as

$$(a_{j_l,j_m} - a_{j_l,k_m}) - (a_{k_l,j_m} - a_{k_l,k_m})\,. \tag{3.32}$$

We see that the first and the second term are very similar. In fact, the first term becomes the second term if we replace j_l by k_l. We are going to use this symmetry by introducing the variables p_1 and p_2 which are defined in the first part of our algorithm 3.7.

Algorithm 3.7 Setup of linear system to compute S_i^{e12} and S_i^{z123} for $p = 3$ – Part 1

```
1: for l ← 1 to M_i do
2:     if l ∈ I_α then
3:         p_1 ← j_l; p_2 ← k_l
4:     else                                          ▷ l ∈ I_β
5:         p_1 ← r_l; p_2 ← u_l
6:     end if
```

Now we are able to calculate the actual values, with the following algorithm 3.8. The representation (3.32) makes clear what kind of optimizations we have used. The array $\mathcal{M}_i(j)$ determines within each term the sign of the entry, this is done by the if-clause in line 12. The above defined variables p_1 and p_2 encode if it is the first, or the second term we are currently dealing with. This is implemented in line 10, which changes the sign of the entry $a_{p_k,j}$.

Lines 16-22 compute the $\hat{f}_l$-values implementing the last formulas of (3.31).

Parallel 3.27: Alike the parallelizations before, algorithm 3.7 is performed for every $i \in S_a$. All these iterations are independent from each other, which allows a parallel execution. ■

Algorithm 3.8 Setup of linear system to compute S_i^{e12} and S_i^{z123} for $p = 3$ – Part 2

```
7:    for k ← 1 to 2 do
8:        for all j with a_{p_k,j} ≠ 0 do
9:            col ← |M_i(j)|                                   ▷ col ∈ I_α ∪ I_β
10:           val ← a_{p_k,j} · (−1)^{k+1}                     ▷ 1st or 2nd term
11:           if M_i(j) ≠ 0 then                               ▷ j ∈ S_i^{e12} ∪ S_i^{z123}
12:               if M_i(j) > 0 then
13:                   f̂_{l,col} ← f̂_{l,col} + val              ▷ 1st position
14:               else
15:                   f̂_{l,col} ← f̂_{l,col} − val              ▷ 2nd position
16:                   f̂_l ← { f̂_l + val,      col ∈ I_α
                            { f̂_l + (5/9)val,  col ∈ I_β
17:               end if
18:           else if I_aez(j) == 2 then                       ▷ type-z-node of type s or t
19:               f̂_l ← f̂_l + (2/9)val
20:           else if j == i then
21:               f̂_l ← f̂_l + val
22:           end if
23:       end for
24:   end for
25: end for
```

By theorem 3.25, we know that the components of the solution vector λ of the linear system (3.23) can only attain a finite number of values. Particularly the first n_i components, which are $\alpha_1, \ldots, \alpha_{n_i}$ must be equal to $\frac{1}{3}$ or $\frac{2}{3}$. The following m_i components, which are $\beta_1, \ldots, \beta_{m_i}$ must be equal to $\frac{1}{9}$ or $\frac{4}{9}$.

$$\lambda = \Big(\underbrace{\tfrac{1}{3}/\tfrac{2}{3}, \ldots, \tfrac{1}{3}/\tfrac{2}{3}}_{\alpha_1,\ldots,\alpha_{n_i}}, \underbrace{\tfrac{1}{9}/\tfrac{4}{9}, \ldots, \tfrac{1}{9}/\tfrac{4}{9}}_{\beta_1,\ldots,\beta_{m_i}}\Big)^\top \in \mathbb{R}^{M_i}$$

Thus we get the following criteria to determine S_i^{e12} and S_i^{z123} for any fixed $i \in S_a$.

Criterion 3.28: For any fixed $i \in S_a$ let $\lambda^* = (\alpha_1^*, \ldots, \alpha_{n_i}^*, \beta_1^*, \ldots, \beta_{m_i}^*)^\top$ be the solution vector of equation (3.23) where the entries of the coefficient matrix and the components of the right hand side vector are defined by (3.31).

If $\alpha_l^* = \frac{2}{3}$ then $(j_l, k_l) \in S_i^{e12}$, otherwise $(k_l, j_l) \in S_i^{e12}$.

If $\beta_l^* = \frac{4}{9}$ then $(r_l, s_l, t_l, u_l) \in S_i^{z123}$, otherwise $(u_l, s_l, t_l, r_l) \in S_i^{z123}$. ■

Using criteria 3.17, 3.20, 3.21, 3.22, 3.24 and 3.28, we can get the $N_a \times N$ restriction operator P_h^H from A_h^3 to A_h^1.

Defining $P_H^h := (P_h^H)^\top$ and $A_h^1 := P_h^H A_h^3 P_H^h$ as in the biquadratic case, we give our algebraic multigrid algorithm 3.9 to solve the bicubic FEM equation (3.9).

Algorithm 3.9 Modified Algorithm for the bicubic Lagrangian FEM-Equation on squares

1: **Setup:**
2: Find the flag array I_{aez} by criteria 3.17 and 3.20
3: Find sets S_i^e, S_i^z, R_j^e, R_j^z, E_r^e and E_r^z by criterion 3.21
4: Find sets S_i^{e12} and S_i^{z123} by criteria 3.22, 3.24 and 3.28
5: Construct the restriction operator P_h^H by (3.13)
6: Construct the prolongation operator $P_H^h = \left(P_h^H\right)^\top$
7: Construct the coarse matrix $A_h^1 = P_H^h A_h^3 P_h^H$
8: **Presmoothing:** $u_h \leftarrow u_h + S(f - A_h^2 u_h)$, $j = 1, \ldots, m_1$
9: **Solving the coarse equation:** $e_{h,1} \leftarrow {A_h^1}^{-1} P_h^H (f - A_h^2 u_h)$
10: **Correcting:** $u_h \leftarrow u_h + P_H^h e_{h,1}$
11: **Postsmoothing:** $u_h \leftarrow u_h + S(f - A_h^2 u_h)$, $j = 1, \ldots, m_2$

3.3.3 Differences of the Algorithm for Problem $C2$ on a general mesh

In chapter 3.3.2 we discussed the algorithm to solve problem $C2$ on a mesh of squares. In this chapter we are going to generalize this result to a mesh of arbitrary quadrilateral elements. We will see that the differences are only of technical nature and that all essential ideas have already been derived. All results until including criterion 3.22 are the same. The only difference is, that we are not able to apply criterion 3.24 to identify nodes s and t in $(r, s, t, u) \in S_i^{z123}$ in order to simplify the minimization problem (3.20). Therefore we need to introduce two additional unknowns in (3.16) which correspond to nodes s and t with the previous coefficients $\frac{2}{9}$. This makes the minimization problem more technical and complex, but does not change the general approach.

We have to choose a more general representation for our linear function $\psi_i(x)$ than (3.16) by using four instead of two unknowns. For any fixed $i \in S_a$ we have

$$\begin{aligned}\psi_i(x) = \gamma_i(x) + \sum_{n=1}^{n_i} (\alpha_n \gamma_{j_n}(x) + (1-\alpha_n)\gamma_{k_n}(x)) + \ldots \\ \sum_{m=1}^{m_i} \left(\beta_m^1 \gamma_{r_m}(x) + \beta_m^2 \gamma_{s_m}(x) + \beta_m^3 \gamma_{t_m}(x) + (1 - \beta_m^1 - \beta_m^2 - \beta_m^3)\gamma_{u_m}(x)\right),\end{aligned} \tag{3.33}$$

where (j_n, k_n) is the n-th index in S_i^{ee}, (r_m, s_m, t_m, u_m) is the m-th index in S_i^{zz} and $\alpha_l, \beta_m^1, \beta_m^2$, $\beta_m^3 \in [0, 1]$ for $n = 1, \ldots, n_i$, $m = 1, \ldots, m_i$ are the unknown parameters.

We are interested in the algebraic properties of $\{\alpha_n\}_{n=1}^{n_i}$, $\{\beta_m^1\}_{m=1}^{m_i}$, $\{\beta_m^2\}_{m=1}^{m_i}$ and $\{\beta_m^3\}_{m=1}^{m_i}$.

As a first step we reorder (3.33) by its coefficients

$$\begin{aligned}\psi_i(x) &= \underbrace{\gamma_i(x) + \sum_{n=1}^{n_i} \gamma_{k_n} + \sum_{m=1}^{m_i} \gamma_{u_m}}_{=:\varphi(x)} + \sum_{n=1}^{n_i} \alpha_n \underbrace{(\gamma_{j_n} - \gamma_{k_n})}_{=:\tilde{\varphi}_n(x)} + \dots \\ &\quad \sum_{m=1}^{m_i} \beta_m^1 \underbrace{(\gamma_{r_m} - \gamma_{u_m})}_{=:\theta_m^1(x)} + \beta_m^2 \underbrace{(\gamma_{s_m} - \gamma_{u_m})}_{=:\theta_m^2(x)} + \beta_m^3 \underbrace{(\gamma_{t_m} - \gamma_{u_m})}_{=:\theta_m^3(x)} \\ &= \varphi(x) + \sum_{n=1}^{n_i} \alpha_n \tilde{\varphi}_n(x) + \sum_{m=1}^{m_i} \beta_m^1 \theta_m^1(x) + \beta_m^2 \theta_m^2(x) + \beta_m^3 \theta_m^3(x) \end{aligned} \tag{3.34}$$

Define $\lambda := (\alpha_1, \dots, \alpha_{n_i}, \beta_1^1, \dots, \beta_{m_i}^1, \beta_1^2, \dots, \beta_{m_i}^2, \beta_1^3, \dots, \beta_{m_i}^3)^\top$ and recap the functional of equation (3.18) given by

$$J(\lambda) = a(\psi_i, \psi_i),$$

where $a(\cdot, \cdot)$ is the bilinear form of our model problem (1.1). In a similar way as in the special case we define $M_i := n_i + 3m_i$ and the functions

$$\begin{aligned} \eta_j &:= \tilde{\varphi}_j && j = 1, \dots, n_i, \\ \eta_j &:= \theta_{j-n_i}^1 && j = n_i + 1, \dots, n_i + m_i, \\ \eta_j &:= \theta_{j-n_i-m_i}^2 && j = n_i + m_i + 1, \dots, n_i + 2m_i, \\ \eta_j &:= \theta_{j-n_i-2m_i}^3 && j = n_i + 2m_i + 1, \dots, M_i. \end{aligned}$$

This results in the following shorter notation for (3.34) which has the same structure as (3.19).

$$\psi_i(x) = \varphi(x) + \sum_{j=1}^{M_i} \lambda_j \eta_j(x) \tag{3.35}$$

Due to the same representation of (3.35) as in the previous chapter, the definition of the minimization problem (3.20) and the derivation (3.21) are identical. The only change is a different number M_i, which however does not influence these results. But we are able to give a more general result than theorem 3.25.

Theorem 3.29: There exists a unique solution vector

$$\lambda^* = (\alpha_1^*, \dots, \alpha_{n_i}^*, \beta_1^{1,*}, \dots, \beta_{m_i}^{1,*}, \beta_1^{2,*}, \dots, \beta_{m_i}^{2,*}, \beta_1^{3,*}, \dots, \beta_{m_i}^{3,*})^\top$$

of the minimization problem (3.20). Moreover, the solution

$$\begin{aligned} \psi_i^*(x) &= \gamma_i(x) + \sum_{n=1}^{n_i} (\alpha_n^* \gamma_{j_n}(x) + (1 - \alpha_n^*) \gamma_{k_n}(x)) + \dots \\ &\quad \sum_{m=1}^{m_i} \left(\beta_m^{1,*} \gamma_{r_m}(x) + \beta_m^{2,*} \gamma_{s_m}(x) + \beta_m^{3,*} \gamma_{t_m}(x) + (1 - \beta_m^{1,*} - \beta_m^{2,*} - \beta_m^{3,*}) \gamma_{u_m}(x) \right), \end{aligned} \tag{3.36}$$

must be the basisfunction $\psi_i(x)$ of the linear finite element related to the node x_i.

Proof: The proof is analogous to the proof of theorem 3.25. □

Theorem 3.29 implies that we have to solve a similar linear system to (3.23)

$$\hat{A}\lambda = \hat{F} \tag{3.37}$$

in order to determine $S_i^{e_{12}}$ and $S_i^{z_{123}}$.
The structure of the coefficient matrix $\hat{A}$ and the the right hand side vector $\hat{F}$ is more complex than in the special case of squares. In the following we will use the index sets $I_\alpha := \{1, \dots, n_i\}$, $I_\beta^1 := \{n_i+1, \dots, n_i+m_i\}$, $I_\beta^2 := \{n_i+m_i+1, \dots, n_i+2m_i\}$ and $I_\beta^3 := \{n_i+2m_i+1, \dots, M_i\}$ as a shorthand. Furthermore we define the vectors $\tilde{\varphi} = (\tilde{\varphi}_1, \dots, \tilde{\varphi}_{n_i})^\top$, $\theta^1 = (\theta_1^1, \dots, \theta_{m_i}^1)^\top$, $\theta^2 = (\theta_1^2, \dots, \theta_{m_i}^2)^\top$ and $\theta^3 = (\theta_1^3, \dots, \theta_{m_i}^3)^\top$ which are used in the following matrix notation:

$$\begin{aligned}
a(\tilde{\varphi}, \tilde{\varphi}) &= (\hat{a}_{jm})_{j,m\in I_\alpha}\,, & & \\
a(\tilde{\varphi}, \theta^1) &= (\hat{a}_{jm})_{j\in I_\alpha, m\in I_\beta^1}\,, & & \\
a(\tilde{\varphi}, \theta^2) &= (\hat{a}_{jm})_{j\in I_\alpha, m\in I_\beta^2}\,, & & \\
a(\tilde{\varphi}, \theta^3) &= (\hat{a}_{jm})_{j\in I_\alpha, m\in I_\beta^3}\,, & a(\tilde{\varphi}, \varphi) &= (a(\tilde{\varphi}_j, \varphi))_{j\in\{1,\dots,n_i\}}\,, \\
a(\theta^1, \theta^1) &= (\hat{a}_{jm})_{j,m\in I_\beta^1}\,, & a(\theta^1, \varphi) &= \left(a(\theta_j^1, \varphi)\right)_{j\in\{1,\dots,m_i\}}\,, \\
a(\theta^1, \theta^2) &= (\hat{a}_{jm})_{j\in I_\beta^1, m\in I_\beta^2}\,, & a(\theta^2, \varphi) &= \left(a(\theta_j^2, \varphi)\right)_{j\in\{1,\dots,m_i\}}\,, \\
a(\theta^1, \theta^3) &= (\hat{a}_{jm})_{j\in I_\beta^1, m\in I_\beta^3}\,, & a(\theta^3, \varphi) &= \left(a(\theta_j^3, \varphi)\right)_{j\in\{1,\dots,m_i\}}\,, \\
a(\theta^2, \theta^2) &= (\hat{a}_{jm})_{j,m\in I_\beta^2}\,, & & \\
a(\theta^2, \theta^3) &= (\hat{a}_{jm})_{j\in I_\beta^2, m\in I_\beta^3}\,, & & \\
a(\theta^3, \theta^3) &= (\hat{a}_{jm})_{j,m\in I_\beta^3}\,. & &
\end{aligned}$$

By these definitions we see, that

$$\hat{A} = \begin{pmatrix} a(\tilde{\varphi}, \tilde{\varphi}) & a(\tilde{\varphi}, \theta^1) & a(\tilde{\varphi}, \theta^2) & a(\tilde{\varphi}, \theta^3) \\ & a(\theta^1, \theta^1) & a(\theta^1, \theta^2) & a(\theta^1, \theta^3) \\ & & a(\theta^2, \theta^2) & a(\theta^2, \theta^3) \\ & & & a(\theta^3, \theta^3) \end{pmatrix}, \qquad \hat{F} = \begin{pmatrix} -a(\tilde{\varphi}, \varphi) \\ -a(\theta^1, \varphi) \\ -a(\theta^2, \varphi) \\ -a(\theta^3, \varphi) \end{pmatrix}.$$

Note that the missing gentries in $\hat{A}$ can be obtained by symmetry. The entries of the coefficient matrix $\hat{A}$ and the entries of the right hand side vector $\hat{F}$ can be expressed as

$$\hat{a}_{\mathfrak{l}\mathfrak{m}} = a(\eta_{\mathfrak{l}}, \eta_{\mathfrak{m}}) = \begin{cases} a(\tilde{\varphi}_l, \tilde{\varphi}_m) = a_{j_l,j_m} - a_{j_l,k_m} - a_{k_l,j_m} + a_{k_l,k_m}, & \mathfrak{l},\mathfrak{m} \in I_\alpha, \\ a(\tilde{\varphi}_l, \theta^1_m) = a_{j_l,r_m} - a_{j_l,u_m} - a_{k_l,r_m} + a_{k_l,u_m}, & \mathfrak{l} \in I_\alpha, \mathfrak{m} \in I^1_\beta, \\ a(\tilde{\varphi}_l, \theta^2_m) = a_{j_l,s_m} - a_{j_l,u_m} - a_{k_l,s_m} + a_{k_l,u_m}, & \mathfrak{l} \in I_\alpha, \mathfrak{m} \in I^2_\beta, \\ a(\tilde{\varphi}_l, \theta^3_m) = a_{j_l,t_m} - a_{j_l,u_m} - a_{k_l,t_m} + a_{k_l,u_m}, & \mathfrak{l} \in I_\alpha, \mathfrak{m} \in I^3_\beta, \\ a(\theta^1_l, \theta^1_m) = a_{r_l,r_m} - a_{r_l,u_m} - a_{u_l,r_m} + a_{u_l,u_m}, & \mathfrak{l},\mathfrak{m} \in I^1_\beta, \\ a(\theta^1_l, \theta^2_m) = a_{r_l,s_m} - a_{r_l,u_m} - a_{u_l,s_m} + a_{u_l,u_m}, & \mathfrak{l} \in I^1_\beta, \mathfrak{m} \in I^2_\beta, \\ a(\theta^1_l, \theta^3_m) = a_{r_l,t_m} - a_{r_l,u_m} - a_{u_l,t_m} + a_{u_l,u_m}, & \mathfrak{l} \in I^1_\beta, \mathfrak{m} \in I^3_\beta, \\ a(\theta^2_l, \theta^2_m) = a_{s_l,s_m} - a_{s_l,u_m} - a_{u_l,s_m} + a_{u_l,u_m}, & \mathfrak{l},\mathfrak{m} \in I^2_\beta, \\ a(\theta^2_l, \theta^3_m) = a_{s_l,t_m} - a_{s_l,u_m} - a_{u_l,t_m} + a_{u_l,u_m}, & \mathfrak{l} \in I^2_\beta, \mathfrak{m} \in I^3_\beta, \\ a(\theta^3_l, \theta^3_m) = a_{t_l,t_m} - a_{t_l,u_m} - a_{u_l,t_m} + a_{u_l,u_m}, & \mathfrak{l},\mathfrak{m} \in I^3_\beta, \end{cases}$$

$l \in \{1, \dots, n_i\}$: (3.38)

$$\hat{f}_l = -a(\tilde{\varphi}_l, \varphi) = -\left[(a_{j_l,i} - a_{k_l,i}) + \sum_{n=1}^{n_i} (a_{j_l,k_n} - a_{k_l,k_n}) + \sum_{m=1}^{m_i} (a_{j_l,u_m} - a_{k_l,u_m}) \right],$$

$l \in \{1, \dots, m_i\}$:

$$\hat{f}_{l+n_i} = -a(\theta^1_l, \varphi) = -\left[(a_{r_l,i} - a_{u_l,i}) + \sum_{n=1}^{n_i} (a_{r_l,k_n} - a_{u_l,k_n}) + \sum_{m=1}^{m_i} (a_{r_l,u_m} - a_{u_l,u_m}) \right],$$

$$\hat{f}_{l+n_i+m_i} = -a(\theta^2_l, \varphi) = -\left[(a_{s_l,i} - a_{u_l,i}) + \sum_{n=1}^{n_i} (a_{s_l,k_n} - a_{u_l,k_n}) + \sum_{m=1}^{m_i} (a_{s_l,u_m} - a_{u_l,u_m}) \right],$$

$$\hat{f}_{l+n_i+2m_i} = -a(\theta^3_l, \varphi) = -\left[(a_{t_l,i} - a_{u_l,i}) + \sum_{n=1}^{n_i} (a_{t_l,k_n} - a_{u_l,k_n}) + \sum_{m=1}^{m_i} (a_{t_l,u_m} - a_{u_l,u_m}) \right],$$

where a_{nm}, $n, m = 1, \dots, N$ are the entries of the coefficient matrix A^3_h in (3.9). As in the biquadratic case we introduce a mapping between the indices $\mathfrak{l}, \mathfrak{m}$ and l, m which is given by

$$l = \begin{cases} \mathfrak{l} & \text{if } \mathfrak{l} \in I_\alpha, \\ \mathfrak{l} - n_i & \text{if } \mathfrak{l} \in I^1_\beta, \\ \mathfrak{l} - 2n_i & \text{if } \mathfrak{l} \in I^2_\beta, \\ \mathfrak{l} - 3n_i & \text{if } \mathfrak{l} \in I^3_\beta, \end{cases}$$

and analogous for m and $\mathfrak{m}$.

By theorem 3.29, the components of the solution vector λ of the linear system (3.37) can only be out of a finite number of values. The first n_i components, which are $\alpha_1, \dots, \alpha_{n_i}$ must be equal to $\frac{1}{3}$ or $\frac{2}{3}$. The following $3m_i$ components, which are $\beta^1_1, \dots, \beta^1_{m_i}$, $\beta^2_1, \dots, \beta^2_{m_i}$ and $\beta^3_1, \dots, \beta^3_{m_i}$ must be equal to $\frac{1}{9}$, $\frac{2}{9}$ or $\frac{4}{9}$.

$$\lambda = \Big(\underbrace{\tfrac{1}{3}/\tfrac{2}{3}, \dots, \tfrac{1}{3}/\tfrac{2}{3}}_{\alpha_1,\dots,\alpha_{n_i}}, \underbrace{\tfrac{1}{9}/\tfrac{2}{9}/\tfrac{4}{9}, \dots, \tfrac{1}{9}/\tfrac{2}{9}/\tfrac{4}{9}}_{\beta^1_1,\dots,\beta^1_{m_i}}, \underbrace{\tfrac{1}{9}/\tfrac{2}{9}/\tfrac{4}{9}, \dots, \tfrac{1}{9}/\tfrac{2}{9}/\tfrac{4}{9}}_{\beta^2_1,\dots,\beta^2_{m_i}}, \underbrace{\tfrac{1}{9}/\tfrac{2}{9}/\tfrac{4}{9}, \dots, \tfrac{1}{9}/\tfrac{2}{9}/\tfrac{4}{9}}_{\beta^2_1,\dots,\beta^3_{m_i}} \Big)^\top \in \mathbb{R}^{M_i}$$

Thus we get the following criterion to determine $S_i^{e_{12}}$ and $S_i^{z_{123}}$ for any fixed $i \in S_a$.

Criterion 3.30: For any fixed $i \in S_a$ let

$$\lambda^* = (\alpha_1^*, \ldots, \alpha_{n_i}^*, \beta_1^{1,*}, \ldots, \beta_{m_i}^{1,*}, \beta_1^{2,*}, \ldots, \beta_{m_i}^{2,*}, \beta_1^{3,*}, \ldots, \beta_{m_i}^{3,*})^\top$$

be the solution vector of equation (3.37) where the entries of the coefficient matrix and the components of the right hand side vector are defined by (3.38).

If $\alpha_l^* = \frac{2}{3}$ then $(j_l, k_l) \in S_i^{e_{12}}$, otherwise $(k_l, j_l) \in S_i^{e_{12}}$.

$$\text{If } \beta_l^{1,*} = \begin{cases} \frac{1}{9} & \text{then } (r_l, *, *, *) \in S_i^{z_{123}}, \\ \frac{2}{9} & \text{then } (*, r_l, *, *) \in S_i^{z_{123}} \text{ or } (*, *, r_l, *) \in S_i^{z_{123}}, \\ \frac{4}{9} & \text{then } (*, *, *, r_l) \in S_i^{z_{123}}. \end{cases}$$

$$\text{If } \beta_l^{2,*} = \begin{cases} \frac{1}{9} & \text{then } (s_l, *, *, *) \in S_i^{z_{123}}, \\ \frac{2}{9} & \text{then } (*, s_l, *, *) \in S_i^{z_{123}} \text{ or } (*, *, s_l, *) \in S_i^{z_{123}}, \\ \frac{4}{9} & \text{then } (*, *, *, s_l) \in S_i^{z_{123}}. \end{cases}$$

$$\text{If } \beta_l^{2,*} = \begin{cases} \frac{1}{9} & \text{then } (t_l, *, *, *) \in S_i^{z_{123}}, \\ \frac{2}{9} & \text{then } (*, t_l, *, *) \in S_i^{z_{123}} \text{ or } (*, *, t_l, *) \in S_i^{z_{123}}, \\ \frac{4}{9} & \text{then } (*, *, *, t_l) \in S_i^{z_{123}}. \end{cases}$$

The value u_l takes the last remaining space in the group of βs associated with index l. ∎

The final algebraic multigrid algorithm for a general quadrilateral mesh is constructed by the same first four criteria 3.17, 3.20, 3.21 and 3.22 as in the special case with a square mesh. Criterion 3.24 cannot be used in this case and criterion 3.28 is replaced by 3.30. With these results we can get the $N_a \times N$ restriction operator P_h^H from A_h^3 to A_h^1.

We define $P_H^h := (P_h^H)^\top$ and $A_h^1 := P_h^H A_h^3 P_H^h$. The changes to algorithm 3.9 in order to solve (3.9) are only in the setup phase, the solution phase stays the same.

Algorithm 3.10 Setup Phase of Modified Algorithm for the bicubic Lagrangian FEM-Equation on a general quadrilateral mesh

1: **Setup:**
2: Find the flag array I_{aez} by criteria 3.17 and 3.20
3: Find sets S_i^e, S_i^z, R_j^e, R_j^z, E_r^e and E_r^z by criterion 3.21
4: Find sets $S_i^{e_{12}}$ and $S_i^{z_{123}}$ by criteria 3.22 and 3.30
5: Construct the restriction operator P_h^H by (3.13)
6: Construct the prolongation operator $P_H^h = \left(P_h^H\right)^\top$
7: Construct the coarse matrix $A_h^1 = P_H^h A_h^3 P_h^H$

Chapter 4

Implementations

In the following chapter we introduce the basic behavior of the implemented codes and give an overview of the code structure. A detailed documentation can be found on the enclosed CD at the head of each subprogram, where the input/output-variables and the behavior are described.

The established theory and the resulting algorithms have been implemented in Fortran 95 [1,8,25,26] in the software package `maiprogs` which has been developed by Matthias Maischak. It provides implementations of standard data types like sparse matrices and vectors and functions for their management, fast and efficient algorithms which are often used in numerical mathematics and a big variety of FEM and BEM (boundary element method) procedures which allow personal customizations on an abstract level. For that reason the script language `BCL` is used, in which a problem and the solution strategy is defined, while all technical aspects, like computations and data management is done in the background. The computed results can be accessed from the `BCL`-level and used for further computations. For more details we refer to the following documents: The concept of the software package is described in [20], key aspects of the technical implementations can be found in [21] and applications with sample scripts are listed in [22].

The classical multigrid method has already been implemented in the software package in the subroutine `fmg` in the module `mlfbas`, and therefore the classical smoothers like Gauss-Seidel, Jacobi or SOR-iterations are already available. In our AMG methods which are used in the numerical experiments of the following chapter 5, these preimplemented routines have been used. All other parts have been written independently and as an extension to `maiprogs` and can be found on the enclosed CD.

The codes are grouped in five different modules which are described in table 4.1. In order to start the solving process the module `amg` is used. The core modules are `coarsening` and `amgip` as they implement all necessary algorithms for the setup phase of AMG, whereas the modules `wutil` and `cdef` contain administrative functions and data type definitions.

The setup and solution process is started by calling the subprogram `amg22` which is contained in the module `amg`. All other functions and programs are called automatically by specifying the desired parameters. This subprogram is the most important one as well as the only one that should be called directly by the user. The most important parameters and their values are described below. More details about the other values can be found on the enclosed CD.

Module	Description
`amg`	Routines for starting the solution process and solving a linear system (arising from higher-order FEM discretization) with AMG
`coarsening`	Coarsening procedures
`amgip`	Routines which initialize the restriction (interpolation) operator
`wutil`	Utilities for the AMG programs, e.g. bubble sort, error messages, ...
`cdef`	AMG coarsening data type definitions

Table 4.1: Module Structure

The solution process is visualized in figure 4.1 (cf. algorithm 2.2). First `amg22` is called and starts the subprogram `iniamg22` which implements the setup phase. After that the subprogram `amg22` starts the solution phase which generally can be done in two different ways. Either a classical V- or W-cycle is applied by calling the subprogram `fmg` which is already part of `maiprogs` or a special cycle type, a $V_0(m_0)$-cycle is started (see figure 5.1), which has been implemented by our codes. The decision depends on the value of the variable `muf`, if it is greater zero a $V_0(m_0)$-cycle with $m_0 =$ `muf` is performed, otherwise a V- or W-cycle is executed.

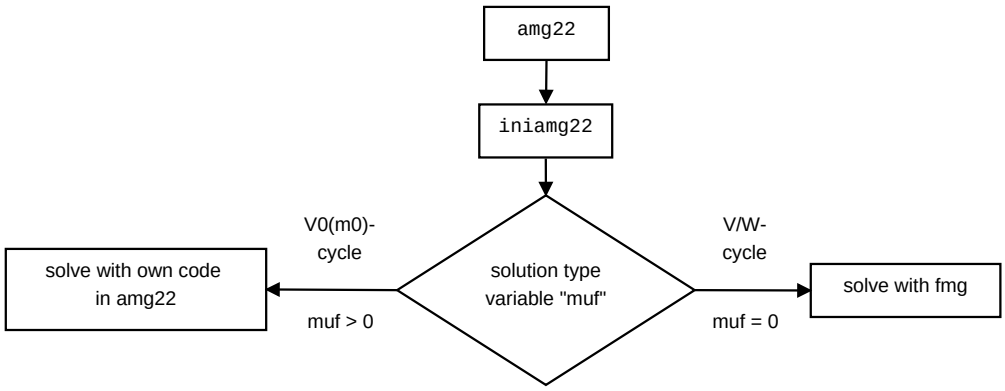


Figure 4.1: Implemented AMG solution process

Figure 4.2 represents the intern solution process in a flow chart (cf. algorithm 2.1). The arrows indicate the order in which the individual routines are called. The choice of the coarsening/prolongation strategy depends on the variables `cstr/pstr`. The entire setup phase is performed by the subprogram `coarsenamg22` which is in the center of this figure. First the system matrix which is stored in a sparse CSR-format is sorted row by row with respect to the column offsets. In the notation of section 1.2 this means, that the values in AA between $IA(i)$ and $IA(i+1)-1$ are sorted with respect to the corresponding entries in JA for every $i = 1, \ldots, N$. This step can be performed in nearly linear time as the sets are normally nearly sorted, which is why bubble sort had been chosen. In particular many operations on sets have been optimized by assuming a sorted ordering (cf. section 1.6.1). After this step `coarsenamg22` generates the coarser levels by calling a coarsening and prolongation routine iteratively until a stopping criterion is reached. This is either reaching

- a maximum number of levels (variable `mnum`) or
- less or equal 40 unknowns in the current level or

- a density of the current system matrix over 60% or
- no change in the coarser system.

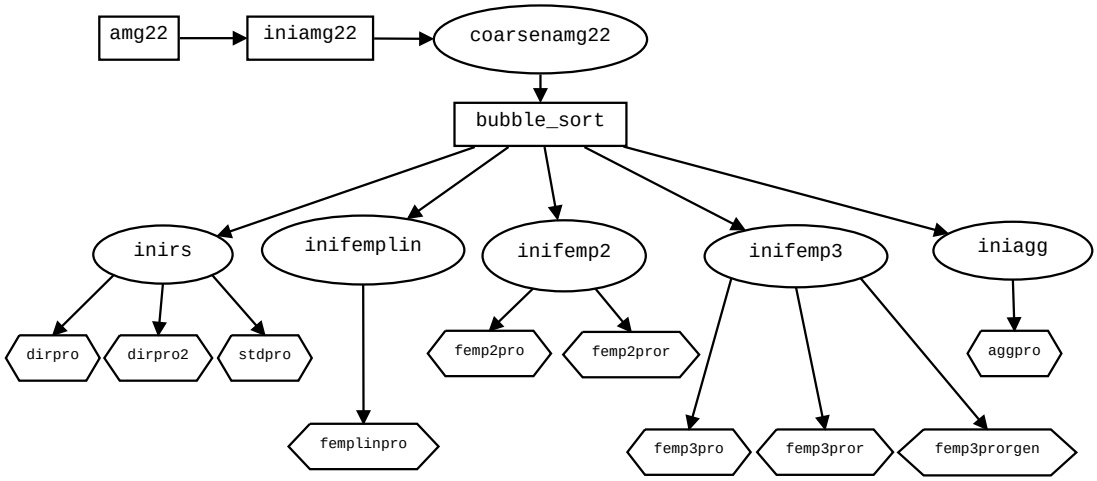


Figure 4.2: Implemented AMG Setup Phase

All subprograms contained in the module `coarsen` are shown in circles, the subroutines of `amgip` are depicted by hexagons and the functions in rectangles are either in `amg` or `wutil`. The table 4.2 gives an overview of the available coarsening functions and which value of the `cstr` variable chooses this option.

Subprogram	Description	`cstr`	Details	Section
`inirs`	Classical RS-Coarsening	0/1	normal/weaker form	2.4.2
`inifemplin`	Hierarchical Basis, arb. p	11/21	triangles/rectangles	5.3
`inifemp2`	Higher-Order FEM for $p = 2$	12/22	triangles/rectangles	3.2
`inifemp3`	Higher-Order FEM for $p = 3$	13/23	triangles/rectangles	3.3
`iniagg`	Smoothed Aggregation	-2	—	2.4.1

Table 4.2: Coarsening options and their subprograms

Analogously, table 4.3 describes the available restriction/prolongation functions and which values of the `pstr` variable choose these options.

In the case of a hierarchical basis an arbitrary degree p for the FEM-basis can be chosen, but it has to be specified in the variable `p`. Experiments in section 5.3 have been undertaken up to degree 8 and show the applicability and efficiency of the theory even for higher degrees than $p = 3$.

At the end we provide a list of all algorithms, which were developed in the previous chapters, including the details of where they can be found. A detailed description of the behavior is included before every subprogram on the enclosed CD.

Table 4.4 states the implementations of all mentioned algorithms, and table 4.5 completes the list of all theoretically discussed topics which were not mentioned explicitly as algorithms, but still resulted in implementations.

Subprogram	Description	pstr	Details	Section
dirpro	Direct Interpolation	1	no difference for +/−	[45, p. 222]
dirpro2	Direct Interpolation	2	pos./neg. connections	2.5.2
stdpro	Standard Interpolation	3	fastest of the 3	2.5.3
femplinpro	Hierarchical Basis, arb. p	11/21	triangles/rectangles	5.3
femp2pro	HO FEM for $p = 2$	12	triangles	[31, pp. 355]
femp2pror	HO FEM for $p = 2$	22	rectangles	3.2
femp3pro	HO FEM for $p = 3$	13	triangles	[31, pp. 361]
femp3pror	HO FEM for $p = 3$	231	squares	3.3
femp3prorgen	HO FEM for $p = 3$	23	rectangles	3.3.3
aggpro	Smoothed Aggregation	−1	—	2.5.1

Table 4.3: Restriction/Prolongation options and their subprograms

Alg.	Description	Module	Subprogram
2.1	AMG Setup Phase	amg	iniamg22
2.2	AMG Solve Phase	amg	amg22
2.3	One step of a general iterative solution method used in the smoothing process	amg	smooth
2.4	Aggregation	coarsening	aggregate
2.5	RS Coarsening, First pass	coarsening	rs1pass
2.6	RS Coarsening, Second pass	coarsening	rscolor
2.7	Construction of tentative prolongator	amgip	aggpro
2.8	Setup of filtered matrix A_h^F	amgip	createah
2.9	Matrix-Matrix product $A_h^F Y_{k+1}^k$ to setup restriction operator R_k	amgip	smoothpro
3.1	Two-Level Algorithm for the biquadratic Lagrangian FEM-Equation (theoretical)	—	—
3.2	Generate R_j^{bc} and E_k^{bc} for $p = 2$	coarsening	setnbp2
3.3	Generate $S_i^{b_1}$ for $p = 2$ for triangular and rectangular case, respectively	amgip	femp2pro/femp2pror
3.4	Final AMG algorithm for the biquadratic Lagrangian FEM-Equation	coarsening	inifemp2 and femp2pro/femp2pror
3.5	Generate S_i^{ee} for $p = 3$	amgip	femp3pro
3.6	Generate S_i^{zz} for $p = 3$		
3.7/3.8	Setup of linear system to compute $S_i^{e_{12}}$ and $S_i^{z_{123}}$ for $p = 3$		
3.9	Final AMG algorithm for the bicubic Lagrangian FEM-Equation on squares	coarsening	inifemp3 and femp3pror
3.10	Final AMG algorithm for the bicubic Lagrangian FEM-Equation on a general mesh	coarsening	inifemp3 and femp3prorgen

Table 4.4: Implementations of developed algorithms

Subprogram	Module	Called by	Description	Theory
`amgcomp`	`amg`	`amg22`	Computes grid and operator complexity	Table 2.1
`inistrongcon`	`coarsening`	`inirs`	Initializes strong connections by generating sets S_i and $S_i^\top$ for $i = 1, \ldots, N$	Def. 2.7
`inistrongcoup`	`coarsening`	`iniagg`	Initializes strongly coupled neighborhoods $N_i^k(\theta)$	Def. 2.4
`setnbp3`	`coarsening`	`inifemp3`	Creates R_j^e, R_j^z, E_r^e and E_r^z for $j \in S_e$ and $r \in S_z$	Crit. 3.21
`validneigh`	`coarsening`	`setnbp3`	Checks general assumptions for triangulations; used on unstructured grid for croissant-like domain in section 5.5	Ass. 3.4

Table 4.5: Other used subprograms with theoretical connections

Chapter 5

Numerical Experiments

In the previous chapters we developed the theory for several algebraic multigrid methods. We will apply it to several model problems and present the results in the following chapter. The implementation was done in Fortran 95 [1,8,25,26] and is based on the software package `maiprogs` by Matthias Maischak [20,22]. For more details see chapter 4.

5.1 Hardware

The developed programs were written and benchmarked on two independent platforms.

1. A personal notebook equipped with an Intel Core i7-2630QM (2.00 GHz) CPU and 8 GB RAM memory on Ubuntu Linux 12.04 LTS (64 bit) was used. The processor has four physical cores and is able to run up to eight threads via OpenMP.
2. A server equipped with two Intel Xeon E5630 (2.53 GHz) CPUs and 24 GB RAM memory on OpenSUSE 11.4 (64 bit) was used. Each processor has four physical cores and is able to run up to eight threads via OpenMP.

All following experiments, except the last one in section 5.5, were carried out on the server system.

5.2 Performance of New AMG Algorithm

We consider the model problem (1.1) on the domain $\Omega = [-1,1]^2 \setminus [0,1]^2$, called the *L-shape*. In particular, we are going to deal with a *discontinuous coefficient problem*, with

$$f(x_1,x_2) = 2\pi^2 \sin(\pi x_1)\sin(\pi x_2),$$
$$a(x_1,x_2) = \begin{cases} c, & x_2 > 0, \\ 1, & x_2 \leq 0. \end{cases}$$

The AMG algorithm used for this experiment is a parallel variant of [37] and introduced in chapters 2.4.1 and 2.5.1. For simplicity of presentation, we will call this method Algorithm SA.

The used algorithm in [31] is called Algorithm V and is based on an energy minimizing approach (developed in [23], see also [32,42]). If we want to point out similarities and differences to our Algorithm SA we will mention it as Algorithm V. In the following investigations the details of this algorithm are not crucial, and it could be replaced by any AMG algorithm. We are mainly interested in how any of these algorithms can by improved by the techniques developed in chapter 3. The new algorithm will be called new (higher-order) AMG method or Algorithm HO, where these two terms will be used synonymously.

The investigated cases are the bilinear ($p = 1$), the biquadratic ($p = 2$) and the bicubic ($p = 3$) one. The domain Ω was discretized on a uniform mesh of quadrilaterals and we always use the SOR iteration with relaxation factor $\omega = 4/3$ as a pre- and postsmoothing operator with the number of $m_1 = 3$ pre- and $m_2 = 3$ postsmoothing steps. As iteration control precision $\|x_k - x_{k-1}\|/\|x_k\| < 10^{-8}$ we measure the relative change in the solution vector, where x_k is the solution vector after the k-th iteration.

The performance of Algorithm SA is measured in terms of the number of $V-$cycle iterations and CPU wall times in seconds. Our experiments in tables 5.1 and 5.2 show, that the choice of the discontinuous coefficient nearly does not influence the performance of Algorithm SA. A similar experiment in [42, p. 1644] leads to the same result: AMG methods in general do not depend on the size of the jump. Additionally we see that Algorithm SA performs best on bilinear Lagrangian finite elements. This robust and efficient properties hold as well if Algorithm SA is replaced by a different method and also if triangular elements are chosen instead of rectangular ones (see [31, pp. 352]).

With increasing order of the finite element functions, the stiffness matrices become denser, which is the reason why coarsening techniques become more complicated and it is more difficult to control the coarse grid degrees of freedom. However, Algorithm SA possesses good properties for FEM functions of higher degrees, as its CPU times and iteration numbers are far lower than the ones of Algorithm V. This is still true when applying Algorithm SA to a uniform triangulation and results in similar results which are omitted for brevity.

The above discussion motivates the idea of the new AMG algorithms: Use the bilinear finite element space as the first coarser level, then apply a classical AMG method on the bilinear finite element space. By this method the first coarser matrix is the stiffness matrix which corresponds to the bilinear finite element functions. This leads to a very aggressive first coarsening step, as all functions of degree $p \geq 2$ are interpolated by functions of degree $p = 1$.

From tables 5.3 and 5.4 we see, that our new AMG algorithm is more efficient than the classical AMG Algorithm SA. It is on average 2 times faster than the classical Algorithm SA which is applied directly to the linear system.

Experiments have shown, that the performance of Algorithm SA can be further increased by a specially chosen AMG cycle. The first coarsening step is always the most expensive one, which motivates the idea of performing it less often than the others. This is achieved by repeating the V-cycles of levels $1 - M$ m_0 times, where 0 is the finest and M the coarsest level (see Fig. 5.1). We call this type $V_0(m_0)$-cycle. The tables 5.5 and 5.6 show the results of the tables 5.3 and 5.4 with $m_0 = 4$.

If one multiplies the iteration numbers of the tables 5.5 and 5.6 by $m_0 = 4$ the numbers are roughly of the same magnitude as the iteration numbers of performed V-cycles. Hence, loosely speaking, one $V_0(m_0)$-cycle consists of approximately m_0 V-cycles. But the CPU times

Elements	$p=1$ CPU times	$p=1$ Iterations	$p=2$ CPU times	$p=2$ Iterations	$p=3$ CPU times	$p=3$ Iterations
16×16	0.00	6	0.01	15	0.04	22
32×32	0.01	11	0.05	19	0.17	27
64×64	0.03	12	0.18	23	0.70	33
128×128	0.13	19	0.74	29	3.34	41
256×256	0.43	18	3.25	33	13.29	41
512×512	2.02	22	15.16	42	59.70	48
1024×1024	8.99	28	70.96	51	277.93	58

Table 5.1: Performance in CPU times [s] and V-cycles of Algorithm SA with $c = 1$

Elements	$p=1$ CPU times	$p=1$ Iterations	$p=2$ CPU times	$p=2$ Iterations	$p=3$ CPU times	$p=3$ Iterations
16×16	0.00	6	0.01	15	0.04	23
32×32	0.01	11	0.05	19	0.19	31
64×64	0.03	12	0.18	24	0.67	31
128×128	0.13	21	0.75	31	3.81	48
256×256	0.43	18	3.24	33	13.72	43
512×512	1.94	22	14.92	41	64.93	53
1024×1024	9.75	32	71.00	51	274.14	57

Table 5.2: Performance in CPU times [s] and V-cycles of Algorithm SA with $c = 1000$

Elements	DOF	$c=1$ CPU times	$c=1$ Iterations	$c=1000$ CPU times	$c=1000$ Iterations
16×16	705	0.01	6	0.01	6
32×32	2945	0.03	10	0.03	10
64×64	12033	0.13	12	0.13	12
128×128	48641	0.60	19	0.64	21
256×256	195585	2.32	18	2.26	17
512×512	784385	10.62	22	10.61	22
1024×1024	3141633	49.69	28	50.08	28

Table 5.3: Performance in CPU times [s] and V-cycles of the new AMG method for $p = 2$ (see algorithm 3.4)

		$c = 1$		$c = 1000$	
Elements	DOF	CPU times	Iterations	CPU times	Iterations
16×16	1633	0.02	7	0.02	6
32×32	6721	0.10	11	0.10	11
64×64	27265	0.39	12	0.39	12
128×128	109825	2.09	19	2.14	21
256×256	440833	7.80	18	7.58	17
512×512	1766401	35.24	22	35.31	22
1024×1024	7071745	164.39	28	164.58	28

Table 5.4: Performance in CPU times [s] and V-cycles of the new AMG method for $p = 3$ (see algorithm 3.10)

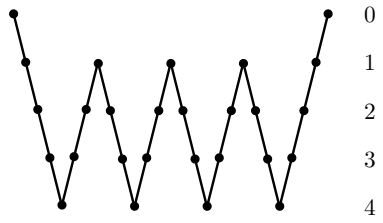


Figure 5.1: $V_0(m_0)$-cycle for $m_0 = 4$ and $M = 4$

are far lower than in the tables above. This variant is on average $2 - 3$ times faster than the original Algorithm SA.

Figures 5.2 and 5.3 combine the last results for $c = 1$. The graphs for $c = 1000$ are very similar, as the numbers are nearly the same. In figure 5.2 the x-axis only is scaled logarithmically whereas in figure 5.3 this applies for both axes in order to visualize the entire range of values. The abbreviation SA denotes Algorithm SA, HO stands for Algorithm HO and V4-cylce is short for $V_0(4)$-cycle.

The left plot in figure 5.2 presents the number of iterations if a V-cycle is used whereas the right plot deals with the $V_0(4)$-cycle variant. Additionally to the tables above, the right plot contains the results for Algorithm SA combined with a $V_0(4)$-cycle. Both figures show clearly

		$c = 1$		$c = 1000$	
Elements	DOF	CPU times	Iterations	CPU times	Iterations
16×16	705	0.01	6	0.01	6
32×32	2945	0.03	5	0.03	5
64×64	12033	0.09	4	0.09	4
128×128	48641	0.42	6	0.45	7
256×256	195585	1.61	6	1.49	5
512×512	784385	6.94	7	6.91	7
1024×1024	3141633	29.71	8	33.09	9

Table 5.5: Performance in CPU times [s] and $V_0(4)$-cycles of the new AMG method for $p = 2$

		$c = 1$		$c = 1000$	
Elements	DOF	CPU times	Iterations	CPU times	Iterations
16×16	1633	0.02	6	0.02	6
32×32	6721	0.08	5	0.08	5
64×64	27265	0.30	5	0.30	5
128×128	109825	1.31	6	1.36	7
256×256	440833	5.09	6	4.84	5
512×512	1766401	22.16	7	22.28	7
1024×1024	7071745	91.62	8	95.91	9

Table 5.6: Performance in CPU times [s] and $V_0(4)$-cycles of the new AMG method for $p = 3$

that the new Algorithm HO suffices with far less iterations than the classical Algorithm SA. Furthermore the left plot is a very good presentation of how the new method works: The graphs of Algorithm SA for $p = 1$ and the new Algorithm HO for $p = 2$ and $p = 3$ have nearly the same shape, but are shifted along the axis of the degrees of freedom. This can be best explained by the basis behavior of Algorithm HO, if we interpret it as a two-level method. First the method coarsens onto the bilinear finite element space (which shifts the graph HO, $p = 2$ or $p = 3$ to the right) and then applies Algorithm SA on this space (which corresponds to graph SA, $p = 1$).

The right plot shows, that V_0-cylces are of no big advantage in the early stage of Algorithm SA, as they correspond to 4 V-cycles. It is interesting to discover, that despite increasing degrees of freedom, the amount of V_0-cycles is nearly constant. We can draw the conclusion that the most efficiency can be obtained by combining Algorithm HO with V_0-cycles.

The comparison of the CPU times in figure 5.3 shows a similar picture. If we fix $p = 2$ or $p = 3$ and compare the corresponding graphs we see, that Algorithm SA using V-cycles is always the slowest and Algorithm HO using V_0-cycles is always the fastest method. The magnitude of the times for Algorithm HO using V-cycles and Algorithm SA using V_0-cycles are the same, but they are always between the previously mentioned methods. This implies that an optimization of the cycle-type or the algorithm brings the same boost in performance. Hence we come to the same conclusion, that the combination of both ideas leads to the most efficient method.

At the end of this section we want to investigate the $V_0(m_0)$-cycle for different choices of m_0. For this purpose we compare in the tables 5.7 and 5.8 solution times without setup times for the smoothed aggregation algorithm (SA) using a V-cycle with our new AMG method for increasing values m_0 using a $V_0(m_0)$-cycle, where a hyphen denotes a classical V-cycle. As a model problem we choose the above L-shape with $c = 1$ and $p = 3$. The tables show the results for a triangular and quadrilateral mesh, respectively and iteration control precision equal to 10^{-8} as before.

The solution times show clearly that the $V_0(m_0)$-cycle is of advantage for this kind of solution strategies. Taking into account that the coarsening process from a bicubic to a bilinear finite element space decreases the degrees of freedom immensely, the restriction and prolongation between these two levels are by far the most expensive operations of the entire cycle. By applying a $V_0(m_0)$-cycle, the linear system of level 1 is solved more accurately which results in possibly more total iterations, but as the iterations from level 1 to M are very fast, because

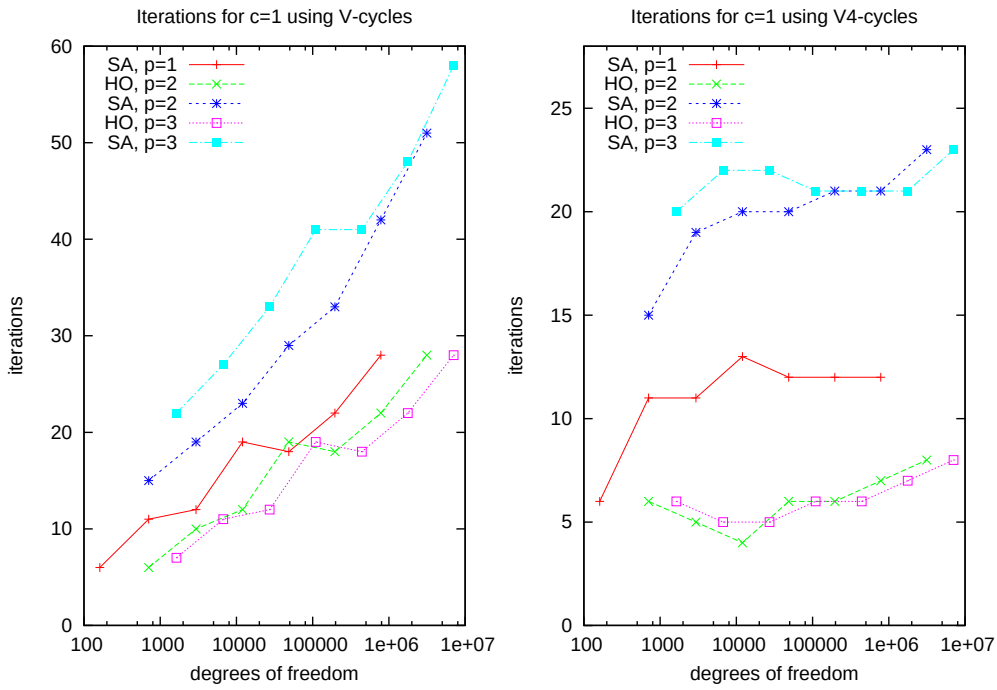


Figure 5.2: Iterations for $c = 1$

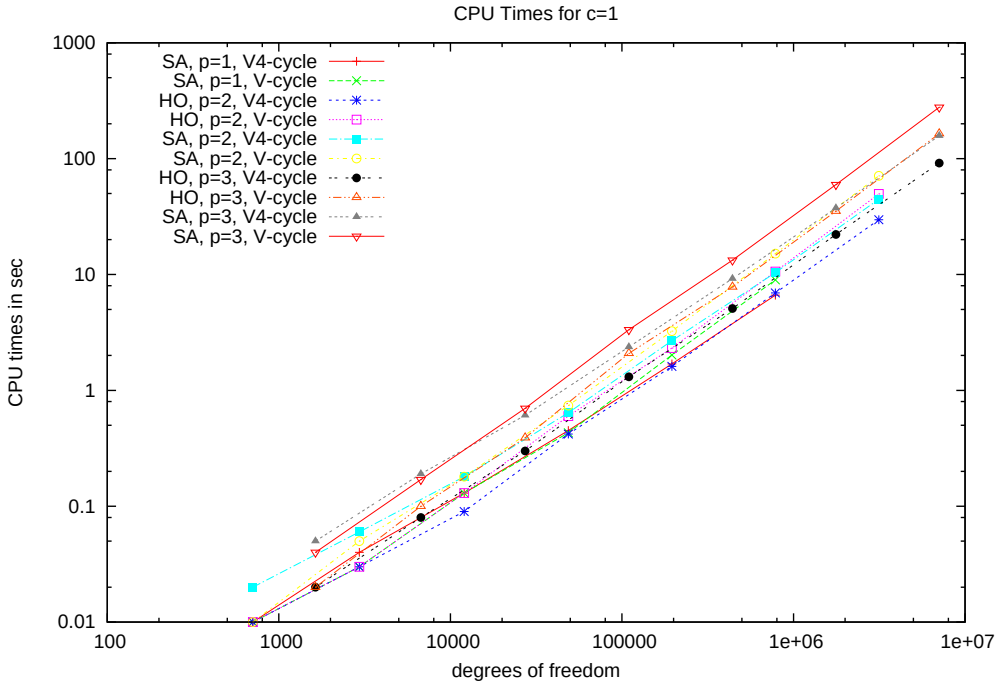


Figure 5.3: CPU Times in seconds for $c = 1$

they deal with a far lower number of unknowns, this leads to remarkably lower solution times. The tables show additionally that this effect can be achieved on quadrilateral as well as on triangular meshes. Moreover, we see a decrease of the solution times even for bigger values than $m_0 = 4$. On some point more cycles on the coarser levels are not of advantage anymore, but this effect is only visible for the coarser meshes. The times on the finest mesh decrease in every step. Thus, the optimal value of m_0 depends on the number of unknowns and the complexity of the problem, but is not known in advance. In consequence of this fact the value $m_0 = 4$ was chosen for the computations above, as it leads to very good solution times in all investigated cases.

Algorithm	m_0	128×128	256×256	512×512	1024×1024
SA	—	3.33	16.50	72.81	310.86
HO	—	0.83	3.90	22.72	90.68
HO	2	0.49	2.38	12.92	53.26
HO	3	0.40	1.79	10.59	40.61
HO	4	0.36	1.63	9.01	34.75
HO	5	0.38	1.44	7.17	33.78
HO	6	0.40	1.52	7.55	30.22
HO	7	0.43	1.61	6.65	26.54

Table 5.7: Comparing solution times of $V_0(m_0)$ for $p = 3$ and $c = 1$ on triangular mesh

Algorithm	m_0	128×128	256×256	512×512	1024×1024
SA	—	3.37	12.59	59.19	292.67
HO	—	1.56	5.65	27.75	142.20
HO	2	0.88	3.34	16.11	81.20
HO	3	0.65	2.54	11.30	64.04
HO	4	0.58	2.27	10.35	49.45
HO	5	0.51	1.99	9.27	45.28
HO	6	0.54	1.60	8.09	40.28
HO	7	0.56	1.67	6.75	34.91

Table 5.8: Comparing solution times of $V_0(m_0)$ for $p = 3$ and $c = 1$ on quadrilateral mesh

5.3 A Bound for the Polynomial Degree p – A Hierarchical Basis

The methods developed in the previous chapters are restricted to a Lagrangian finite element basis. We consider now a straightforward adaption to the case of a hierarchical basis and we will use this case to analyze the behavior of the main idea, the interpolation of grid points corresponding to basis functions of degree $p \geq 2$ by grid points corresponding to bilinear basis function ($p = 1$), on degrees $p \geq 4$. For more information about a hierarchical basis we refer to [21].

Our new AMG algorithm basically consists of two levels. First, we have to identify grid points $i \in S_a$ among all grid points S associated with bilinear basis functions $\psi_i(x)$. Second, the

remaining points have to be sorted according to their geometry in relation to the nodes in S_a. This process involves the distinguishing between grid points on edges and on elements, followed by the possible (re)ordering, i.e. solving a minimization problem like (3.20). Finally, the first coarse matrix, which is the one of the bilinear finite element space, is constructed and the intergrid transfer operators are initialized. This ends level 1, the setup of the bilinear finite element space. In level 2 a classical AMG algorithm, like Algorithm SA, is applied to solve the problem.

The main effort in phase 1 is the construction of the bilinear basis functions $\psi_i(x)$ in (3.6) and (3.13). This step is necessary, due to the fact that $\psi_i(x)$ is neither a member of $\{\phi_i\}_{i\in S}$ nor of $\{\gamma_i\}_{i\in S}$, the families of basis functions for Lagrangian finite elements in the cases $p = 2$ or $p = 3$, respectively. But if we use a hierarchical basis, the bilinear basis functions will be part of the family of basis functions for any degree $p \geq 1$. Hence, the only necessary step in level 1 is the identification of grid points $i \in S_a$ which is achieved by a simple counting argument.

We have always distinguished three types of nodes:

1. Points on a vertex
2. Points on an edge
3. Points in an element

The first ones, are the ones we are looking for, represented in the set S_a. To find them we analyze their interaction with other points in the stiffness matrix A_h^p. By compact support arguments it holds, that rows in A_h^p associated with points of type 2 or 3 always have less entries than rows corresponding to points of type 1 (compare criteria 3.6 and 3.17). These entries are called neighbors of the node or basis function i.

In the construction of basis functions in two dimensions, the x- and y-axis of the reference triangle or rectangle is split into $p+1$ distinct nodes each. The interpolation points are defined as the Cartesian product of these nodes. A function of degree p is completely determined by its values at these $\frac{(p+1)(p+2)}{2}$ nodes on a triangle or $(p+1)^2$ nodes on a rectangle. After transforming and combining these reference basis functions to global basis functions, we obtain three distinct types given in table 5.9. For a more detailed description and derivation see chapter 1.4.

Nodal Type	Associated with	Name	Nr. Elements in Support
1	Vertex	Nodal Function	≥ 3
2	Edge	Edge Function	2
3	Element	Bubble Function	1

Table 5.9: Basis Function Types

This allows us to uniquely identify type-a nodes, by computing the maximum number of possible neighbors of type 2, as type-a nodes have more neighbors than that. Note, that because of the restricted support, points of type 3 will always have less neighbors than points of type 2. Thus, we have derived the general result of criteria 3.6 and 3.17.

Criterion 5.1: For any $p \geq 1$ and any index $i \in S$, if there are more than $(p+1)^2$ nonzero entries on a mesh of triangles or more than $(2p+1)(p+1)$ on a mesh of quadrilaterals in the

row i of the matrix A_h^p, then the node x_i related to the index i is a type-a node. ∎

With these results we are going to solve the following specification of problem (1.1) on the Square $\Omega = [-1,1]^2$ with

$$f(x_1, x_2) = -2(1 - x_1^2) - 2(1 - x_2^2),$$
$$a(x_1, x_2) \equiv 1.$$

The domain Ω is discretized on a uniform mesh of triangles and quadrilaterals and we always use the point Gauss-Seidel iterations as a pre- and postsmoothing operator with the number of $m_1 = 3$ pre- and $m_2 = 3$ postsmoothing steps. The iteration control precision is chosen bigger than before as $\|x_k - x_{k-1}\|/\|x_k\| < 10^{-6}$. Iterations are measured in classical V-cycles of the respective algorithm and are aborted after a maximum of 1000.

The tables arise from a uniform discretization of the square $[-1,1]^2$ in 32 or 64 parts in each dimension. All tables clearly illustrate the high efficiency of the classical AMG Algorithm SA on a stiffness matrix associated with a bilinear finite element space. Comparing Algorithm SA for $p = 1$ with the cases for $p \geq 2$ in the tables 5.10, 5.11, 5.12 and 5.13 shows, that $10-20$ times less iterations than in the higher dimensional cases are necessary and that the solution times are much lower. On the one hand this is due to a smaller finite element space, which implies less degrees of freedom in the linear system, but on the other hand is the difference to the other cases in no relation to the increase in degrees of freedom.

All tables reflect the increase of complexity in the stiffness matrix for higher degrees p. Algorithm SA has difficulties to handle these more complex structures, as the iteration numbers are increasing dramatically for $p \geq 2$. They nearly stay on a constant level for a quadrilateral mesh, whereas they increase again substantially on triangular meshes for $p \geq 6$. The experiments with an iteration count of 1000 have been aborted, and are therefore of no value for our interpretation.

Algorithm HO shows some robust features for even higher degrees than $p = 3$. It is interesting to note, that the number of iterations is nearly constant for two consecutive values of p. This could be explained by a similar structure in the corresponding stiffness matrix. In contrast to Algorithm SA, the iterations do not remain on a constant level for increasing p. This can be explained by the need to interpolate more and more fine grid points from the same amount of coarse grid points on the first level. The tables 5.14 and 5.15 give details of the coarse levels for the quadrilateral cases. The first step of coarsening for every higher-order problem, is done onto the bilinear space. Hence, the levels $1 - M$ (coarsest level: M) are identical for all values of p. Only the finest level is different and shows the increasing degrees of freedom.

Moreover, the algorithm seems to be more efficient on quadrilateral meshes, than on triangular ones. This is due to the current implementation of a hierarchical basis on triangles, which is not optimized in the sense, that it is an energy minimizing basis, which in contrast is done for the hierarchical basis on quadrilaterals.

Despite a very aggressive coarsening, the new AMG method still seems to be more efficient than the classical method as the solution times in all cases are $2 - 10$ times smaller. The setup times of Algorithm HO are just stated for completeness, as knowledge of a hierarchical basis simplifies this phase dramatically.

	Algorithm HO			Algorithm SA		
p	Iterations	Setup Times	Solve Times	Iterations	Setup Times	Solve Times
1	9	0.01	0.01	9	0.00	0.00
2	11	0.01	0.03	83	0.04	0.20
3	11	0.01	0.06	84	0.24	0.50
4	25	0.02	0.29	86	0.57	1.13
5	25	0.03	0.58	86	1.49	2.34
6	47	0.05	1.98	89	3.85	4.77
7	47	0.09	3.41	90	8.09	7.77
8	77	0.15	8.79	99	12.82	13.43

Table 5.10: Performance on hierarchical basis for uniform rec. mesh 32×32 on $[-1,1]^2$

	Algorithm HO			Algorithm SA		
p	Iterations	Setup Times	Solve Times	Iterations	Setup Times	Solve Times
1	16	0.02	0.03	16	0.02	0.02
2	16	0.03	0.11	287	0.17	1.88
3	16	0.04	0.33	292	1.06	6.38
4	24	0.07	1.12	295	2.49	15.71
5	24	0.13	2.25	295	6.68	32.40
6	44	0.22	7.36	296	17.45	60.37
7	44	0.36	12.47	296	36.89	107.83
8	70	0.60	31.61	298	58.52	168.00

Table 5.11: Performance on hierarchical basis for uniform rec. mesh 64×64 on $[-1,1]^2$

All in all, the new AMG algorithm proves to be also very efficient for higher degrees than $p = 3$. The question arises, how expensive the setup phase for these cases might be, as the investigated structures are definitively more complicated than before. It is the question of future investigations, if the advantage of a faster solution phase compensates for the costs of a more complex setup phase.

At the end of this chapter, we want to compare parameters of the coarse grid hierarchies of Algorithm SA with the one's of other classical AMG methods, shown in table 5.16. In particular the weaker Ruge-Stüben (RS) algorithm in combination with standard interpolation, the classical RS algorithm with standard interpolation and the classical RS algorithm with direct interpolation are analyzed. As characteristic for the coarse grid hierarchy, we consider the grid complexity (Grid C.) and the operator complexity (Op. C.) (see section 2.1).

The experiments are performed on the square $[-1,1]^2$ with a uniform discretization of 16×16 squares. We chose a smaller scale, as the presented algorithms are slower in the execution than Algorithm SA. This is due to a less aggressive coarsening and denser coarse grid matrices. This results in a more complex matrix chain product of the type RAP, which consumes here more than 50% of the required setup time. If this problem could be solved, the algorithms are more competetive to Algorithm SA than with the current implemenation. Most parts have already been parallelized, but as mentioned before, is the matrix chain product the main problem. A possible solution of a parallel matrix chain product using MPI is presented in [6]

	Algorithm HO			Algorithm SA		
p	Iterations	Setup Times	Solve Times	Iterations	Setup Times	Solve Times
1	11	0.01	0.01	11	0.00	0.01
2	10	0.01	0.02	139	0.02	0.24
3	11	0.01	0.05	136	0.11	0.58
4	13	0.01	0.11	132	0.39	1.22
5	53	0.02	0.83	138	0.60	2.29
6	261	0.04	7.32	651	0.96	18.92
7	433	0.07	19.76	1000	2.17	47.60
8	247	0.09	17.54	1000	4.54	76.64

Table 5.12: Performance on hierarchical basis for uniform tri. mesh 32×32 on $[-1,1]^2$

	Algorithm HO			Algorithm SA		
p	Iterations	Setup Times	Solve Times	Iterations	Setup Times	Solve Times
1	13	0.02	0.02	13	0.02	0.02
2	12	0.03	0.07	480	0.10	2.53
3	12	0.04	0.16	471	0.46	6.57
4	13	0.06	0.44	455	1.66	16.63
5	45	0.09	2.91	479	2.51	32.38
6	171	0.15	19.27	713	4.07	87.93
7	117	0.23	21.21	1000	9.21	186.45
8	176	0.35	48.85	1000	19.41	292.30

Table 5.13: Performance on hierarchical basis for uniform tri. mesh 64×64 on $[-1,1]^2$

p	Level	Number of rows	Number of nonzeros	Density (% full)	Average entries per row
8	3	65025	6385729	0.002	98.20
7	3	49729	3948169	0.002	79.39
6	3	36481	2283121	0.002	62.58
5	3	25281	1207801	0.002	47.78
4	3	16129	564001	0.002	34.97
3	3	9025	218089	0.003	24.16
2	3	3969	61009	0.004	15.37
1	2	961	8281	0.009	8.62
1	1	121	961	0.066	7.94
1	0	16	116	0.453	7.25

Table 5.14: Coarsening for hierarchical basis on 32×32 quadrilateral mesh of $[0,1]^2$

p	Level	Number of rows	Number of nonzeros	Density (% full)	Average entries per row
8	4	261121	25877569	0.000	99.10
7	4	199809	16024009	0.000	80.20
6	4	146689	9284209	0.000	63.29
5	4	101761	4923961	0.000	48.39
4	4	65025	2307361	0.001	35.48
3	4	36481	896809	0.001	24.58
2	4	16129	253009	0.001	15.69
1	3	3969	34969	0.002	8.81
1	2	441	3797	0.020	8.61
1	1	49	421	0.175	8.59
1	0	9	61	0.753	6.78

Table 5.15: Coarsening for hierarchical basis on 64×64 quadrilateral mesh of $[0,1]^2$

where an extension for matrix tripple products of the form RAP is suggested. This problem presents an interesting basis for future research.

The table shows, that Algorithm SA possess a very good coarsening strategy. This implies in general the requirement of more V-cycles than the other methods, but saves time in the setup phase. In contrary are the classical methods much more expensive in terms of computation as well as memory consumption. But these methods converge in far less V-cycles than Algorithm SA. Therefore one has to decide if fast convergence or low computation time is more preferable. But unless the matrix chain product is implemented in a reasonable way, big problems should be solved with Algorithm SA. On the other hand if a similar problem has to be solved for varying right hand side, it is maybe worth considering to use a faster converging algorithm.

	Algorithm SA		WRS & Std. IP		CRS & Std. IP		CRS & Direct IP	
p	Grid C.	Op. C.	Grid C.	Op. C.	Grid C.	Op. C.	Grid C.	Op. C.
1	1.11	1.10	1.11	1.10	1.26	1.51	1.28	1.56
2	1.26	1.14	1.09	1.09	1.44	3.04	1.71	6.13
3	1.11	1.04	1.10	1.14	1.38	3.27	1.81	8.49
4	1.06	1.02	1.11	1.15	1.57	3.58	1.97	9.26
5	1.04	1.01	1.13	1.17	1.51	2.80	2.08	8.74
6	1.03	1.00	1.15	1.19	1.55	2.59	2.00	6.47

Table 5.16: Grid and Operator Complexity depending on p on hierarchical basis for uniform quadrilateral Mesh 16×16 on $[-1,1]^2$

5.4 Parallel vs. Serial

In the following we compare the advantage of our parallel implementations over serial ones. We consider the same problem from the previous experiment, but focus on the setup times only as we have tried to optimize this step by parallel variations of serial algorithms. All

performed experiments show, that the matrix chain product of the form RAP which is used in the generation of the coarse system matrix, represents the slowest part of the coarsening process. In our codes the preimplemented routine from `maiprogs` has been used, which is in the current version a serial algorithm. This process consumes up to 80% of the total setup time and hence gives rise for future research. Therefore the next step of optimization should be an attempt to parallelize this part.

Due to the above reasons, the CPU times of the setup phase are analyzed without all matrix chain products used in the construction of coarser matrices. The following figures compare the efficiency of the smoothed aggregation algorithm (SA) and the new higher-order AMG method (HO) by limiting the total number of parallel executed threads. Figures 5.4 and 5.5 compare the algorithms for different numbers of degrees of freedom, in particular the square is split in a uniform mesh of 128×128, 256×256, 512×512 or 1024×1024 elements. The times were measured for 1, 2, 4, 8 and 16 threads on the Xeon server consisting of two CPUs with 4 cores each and allowing a maximum of 16 parallel threads.

Both figures show a similar pattern. The smoothed aggregation algorithm is slower than the new AMG method for $p = 2$ as well as for $p = 3$. For 2 threads both routines are slower for $p = 2$, than executed only with one thread. This can be explained by the additional overhead in parallel computation and the fact, that the times are all very small as this part of the setup phase is already quite fast. In contrast for $p = 3$ the new AMG method improves its performance steadily all the time which can be explained by a more complex structure of the system matrix. But on the long term an increasing number of threads leads in both methods and in both cases to an increasing performance.

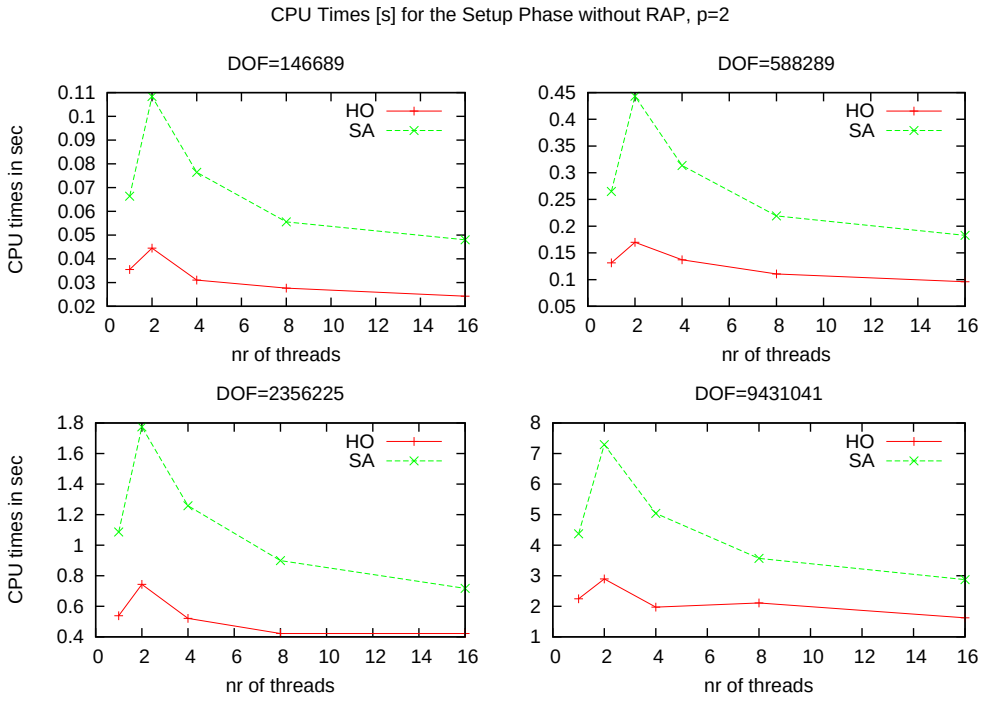


Figure 5.4: Setup times without matrix chain product, $p = 2$, quadrilaterals, square

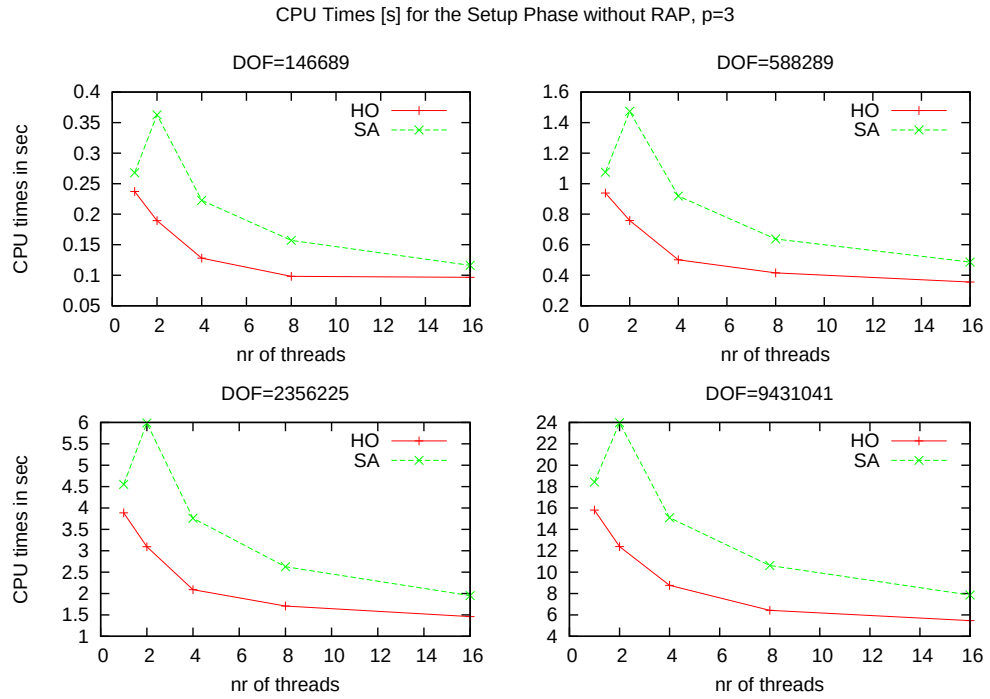


Figure 5.5: Setup times without matrix chain product, $p = 3$, quadrilaterals, square

At the end of this experiment we want to demonstrate the bottle neck, the matrix chain product RAP, in numbers. This time is nearly independent of the used method and the number of threads, as it is a serial implementation. Table 5.17 shows the approximately consumed time for each degree of freedom. Comparing these numbers with the graphs it is evident that this part is the slowest one of the coarsening process and possesses the most optimization potential.

		Total time RAP	
DOF	Elements	$p = 2$	$p = 3$
146689	128×128	0.28	1.05
588289	256×256	1.15	4.27
2356225	512×512	4.67	17.25
9431041	1024×1024	18.78	69.42

Table 5.17: Time consumption of matrix chain product RAP in setup phase

5.5 Delaunay Triangulation

All above experiments have been carried out on structured meshes. In this chapter we will show, that the algorithms work on unstructured meshes and more complicated domains as well. For this purpose we will use the theory of Delaunay mesh generation introduced in section 1.5 on a “croissant” like domain shown in figure 5.6. As a model problem we choose

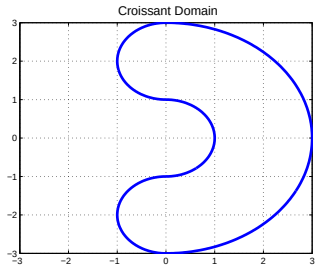


Figure 5.6: Croissant-like domain

$-\Delta u = 1$ in Ω and homogeneous Dirichlet boundary conditions. Remark, that the following computations were performed on the personal notebook and not on the server system.

The mesh was generated in Matlab using the built-in function `delaunay` in the self-programmed function `kipferldel.m`. This function distributes a certain number of random points on the boundary and in the interior of the domain which are then connected via the `delaunay` function. At the end the mesh is checked for edges which do not fully lie in the interior of the domain and triangles of such edges are deleted. The only exception are triangles which intersect with the concave part of the domain in the vicinity of the origin. These are only deleted if at least one edge intersects with the x-axis or at least one of the starting points of at least one edge does not lie in a small vicinity of the Dirichlet boundary. This special treatment prevents spikes of the mesh in this area around the origin. The idea is to get a "proper" mesh for our "wildly" shaped domain. For more details see the script files in the folder `Matlab Meshes` on the enclosed CD. The mesh generation is started with the script file `genKipferlDel.m` and two created triangulations are shown in figure 5.8. Figure 5.8a consists of 1928 elements which corresponds to 3737 or 8596 unknowns for $p = 2$ or $p = 3$, respectively, and figure 5.8b contains 7945 elements which corresponds to 15639 or 35728 unknowns for $p = 2$ or $p = 3$, respectively. These two meshes amongst others have been used to compute the experiments below. Figure 5.9 shows the solutions for these discretizations.

Note, that the above method of generating an unstructured mesh is a very naive one, but for the sake of presenting our developed theory on non-uniform meshes, it is as good as any other method as we are only interested in the comparison of our method with the classical one.

As smoother we apply SOR iterations with relaxation factor $\omega = 4/3$ with a number of $m_1 = 1$ pre- and $m_2 = 1$ postsmoothing steps. This choice led to the best results in the following experiments. In the same manner as in the second experiment we choose the iteration control precision $\|x_k - x_{k-1}\|/\|x_k\| < 10^{-6}$. Iterations are measured in the number of V-cycles.

We compare our developed AMG algorithms using the smoothed aggregation approach (SA) and our new algorithm for higher-order finite elements (HO) with a classical CG method combined with diagonal preconditioning (CG) which is available in `maiprogs` [20].

The tables 5.18 and 5.19 show the superiority of our new AMG method over the AMG algorithm SA and the classical CG method. For an easier interpretation the data of these two tables is plotted in figure 5.7. Algorithm HO is nearly twice as fast as algorithm SA but even ten times faster than the CG method. This is due to a higher number of iterations in the latter method. Both AMG methods are more efficient on the unstructured mesh, since they

are able to use the structural information of the stiffness matrix much better. But comparing the results of the unstructured mesh to the structured one of the tables 5.3 and 5.4 which have a similar number of unknowns, we still see an increase in iterations and CPU times.

	HO		SA		CG	
DOF	Iterations	CPU times	Iterations	CPU times	Iterations	CPU times
3737	37	0.19	24	0.04	208	0.04
15639	26	0.15	51	0.28	386	0.34
64389	42	0.78	56	1.03	725	2.73
259854	39	2.67	76	5.09	1143	15.84
1043027	57	14.22	101	26.04	2190	123.59

Table 5.18: Performance on croissant-like domain for $p = 2$

	HO		SA		CG	
DOF	Iterations	CPU times	Iterations	CPU times	Iterations	CPU times
8596	52	0.16	122	0.41	346	0.17
35728	108	1.04	126	1.34	1602	3.82
144898	60	2.31	135	5.60	1323	12.32
585199	62	10.16	266	36.12	2072	74.54
2348982	83	50.77	206	114.66	3992	555.29

Table 5.19: Performance on croissant-like domain for $p = 3$

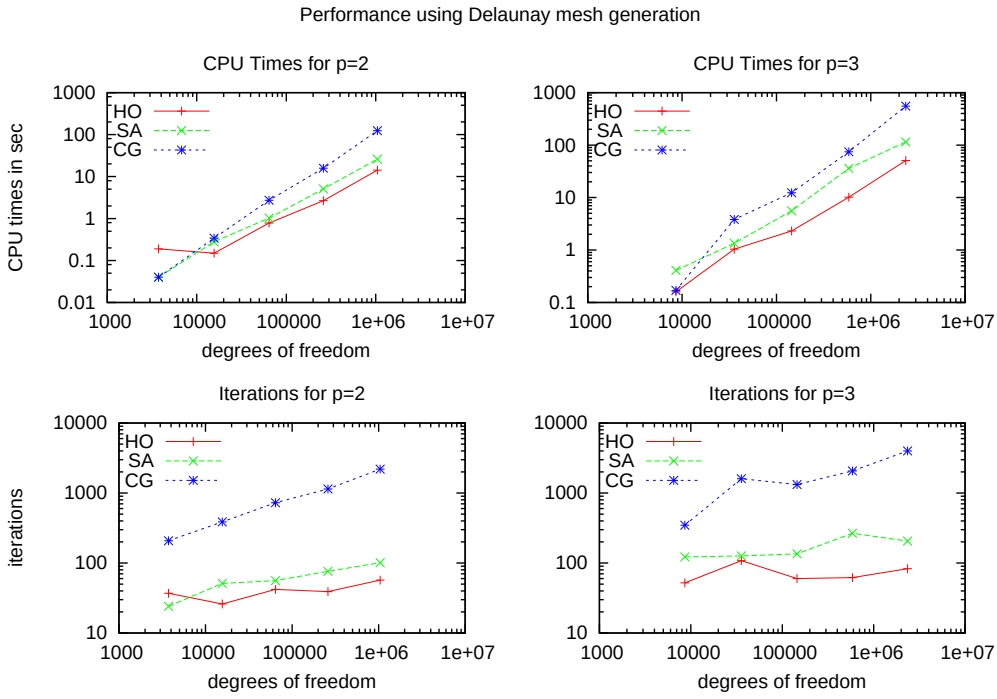
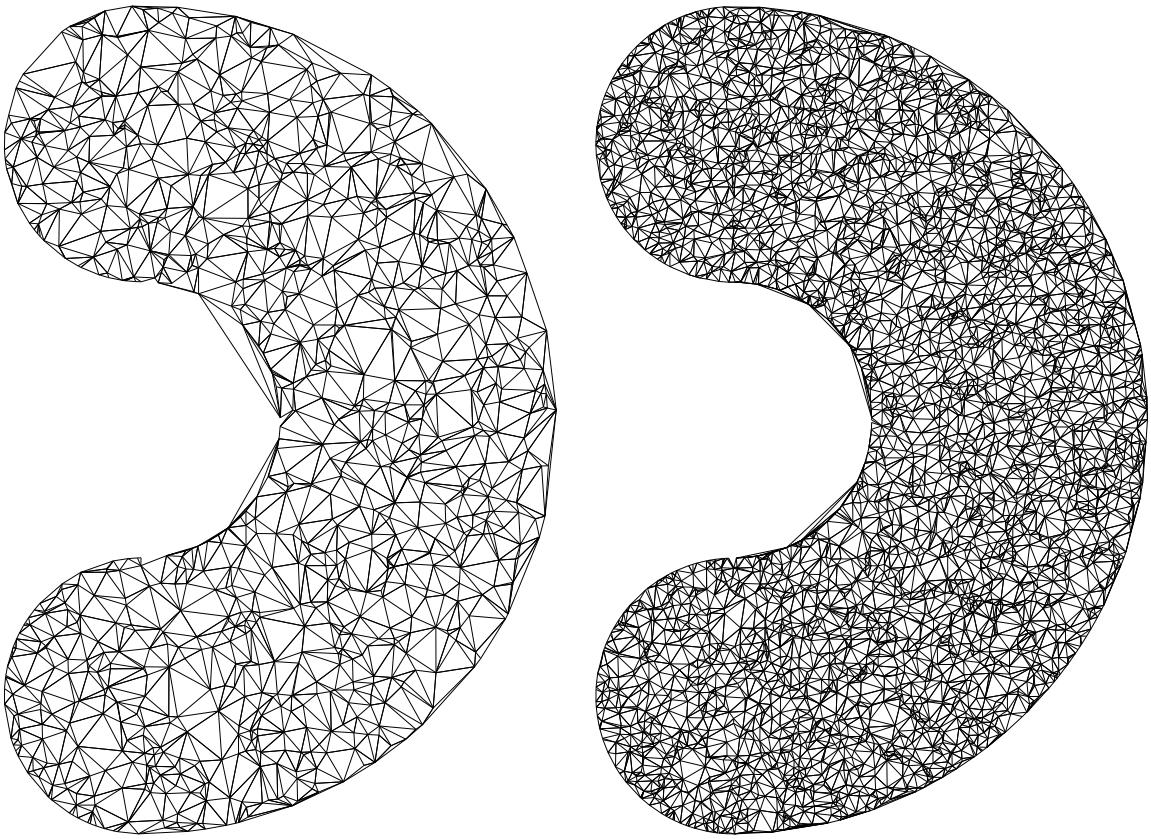
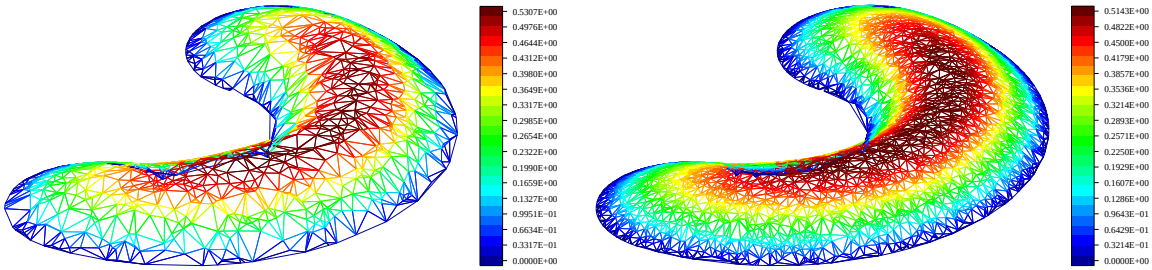


Figure 5.7: Performance on croissant-like domain and unstructured mesh



(a) elements=1928 (b) elements=7945

Figure 5.8: Delaunay triangulations of croissant-like domain



(a) elements=1928 (b) elements=7945

Figure 5.9: Solutions of croissant-like domain

Conclusion and Further Work

In the present work, we gave an introduction into algebraic multigrid (AMG) and motivated its applicability over the concept of algebraic smoothness which lies at the center of this approach. The method of smoothed aggregation by Vaněk, Mandel and Brezina [34] was presented and adapted for parallel execution on a multi-core CPU by the use of several independent threads. Moreover the classical method of Ruge-Stüben was briefly presented. The main chapter of this work was dedicated to the development of a hybrid AMG method using additional input in order to utilize hidden geometrical information in the linear system. In particular we retrieved the (bi-)linear finite element subspace and its associated stiffness matrix from a given stiffness matrix arising from discretization by higher-order Lagrangian finite elements in two dimensions. The retrieved stiffness matrix of (bi-)linear elements was solved by a classical algebraic multigrid method together with a standard smoother, such as the Gauss-Seidel or SOR method. The current work was developed for discretizations with quadrilaterals and continues the work of Shu et al. [31] who introduced the algorithms for triangulations. Furthermore we proved the correctness of our new AMG method for a certain class of elliptic partial differential equations and a discretization on quadrilaterals using polynomials of degree 2 and 3.

The developed method is less algebraic than classical AMG, as it needs the a-priori knowledge of the type of the used finite element space (i.e. (bi-)quadratic, (bi-)cubic, ...) and of the type of used basis functions. But all other geometrical information stays unknown.

In several numerical experiments we have proven, that our new AMG method is significantly more efficient than classical AMG methods which are applied directly to the high-order system. A speedup between 2 and 4 times was measured in all experiments when compared to the classical AMG methods. Moreover, we introduced the $V_0(m_0)$-cycle which proved to be very efficient in combination with our solution strategy, as it reduces the most expensive operations during the solution process. Furthermore, we also tested our main idea with degrees up to $p = 8$ on a hierarchical basis. It has been shown, that it is still very efficient, despite a very aggressive first coarsening step. The efficiency of AMG on problems discretized by (bi-)linear finite elements, which represents the first coarsening step, outweighs the loss of precision.

Due to the fact that all developed algorithms have been developed with special focus on their parallelization potential, we also compared our parallel implementations with respect to different numbers of simultaneously executed threads. With an increasing amount of complexity, which is represented by more degrees of freedom or rising degrees of basis functions, the advantage of a parallel setup became more and more noticeable. Therein an increase from 1 to 2 threads even led to longer computing times in our new method in the case of

$p = 2$. In contrast an increase of threads in the case of $p = 3$ nearly always led to a faster execution. Our parallel variant of the smoothed aggregation approach yielded similar results, but with the difference that the choice of 2 threads always gave the slowest computations. We can conclude, that the profit of a parallel implementation is the higher, the bigger the underlying complexity is but there exists a minimum number of threads which is necessary to compensate the additional overhead of parallel algorithms.

Besides we demonstrated the efficiency of our new algorithm on unstructured meshes created by Delaunay triangulations and a more complicated domain. It was also compared with a CG method where it confirmed its efficiency yet again.

During the development we discovered some issues which would require a separate investigation. The bottle neck of the entire algorithm is definitively the matrix chain product of the type RAP which appears during the generation of the coarse linear systems. In more complex problems it can consume more than 80% of the total setup time. This problem is not specific to our problem class but is inherently coupled with many AMG methods. A parallelization of this operation could boost the computing times significantly and would form a very interesting topic for further research. A good starting point could be found in [6] as it mentions a possible extension of the therein developed algorithm to matrix tripple products of such specific form.

All performed parallelizations were focused on multi-core CPUs and performed via OpenMP. As a next step an integration into cluster systems via MPI could be considered in order to utilize the positive effects of parallel computing even more.

The smoothed aggregation approach in the used configuration possesses a very aggressive coarsening which leads to low memory consumption and low complexities in the coarser levels. This comes with the disadvantage of slower convergence in terms of more cycles. Other methods like the ones from Ruge-Stüben use less aggressive coarsening, which leads to longer setup times but faster convergence. The parallel potential of these methods could be analyzed to utilize the advantages of lower solution times.

Another issue which is part of any (algebraic) multigrid method is the used smoother. The current implementation in `maiprogs` is a serial one, but a parallelization of this part could also result in a significant speed up.

Finally, our main algorithms for discretizations of Lagrangian finite elements only consider biquadratic and bicubic ones in two dimensions. Experiments on a hierarchical basis showed the potential of extensions of the developed theory to elements of even higher order or in three spacial dimensions. But as we saw in the extension from degree 2 to degree 3, these enhancements require a more careful analysis of the underlying geometry, which, however, does not necessarily result in much more expensive algorithms.

List of Algorithms

List of Figures

List of Tables

Bibliography

[1] J.C. Adams, W.S. Brainerd, R.A. Hendrickson, R.E. Maine, J.T. Martin, and B.T. Smith. *The Fortran 2003 Handbook: The Complete Syntax, Features and Procedures*. Springer, 2008.

[2] James H. Bramble. *Multigrid methods*, volume 294 of *Pitman Research Notes in Mathematics Series*. Longman Scientific & Technical, Harlow, 1993.

[3] Achi Brandt. Multi-level adaptive solutions to boundary-value problems. *Mathematics of Computation*, 31(138):pp. 333–390, 1977.

[4] Achi Brandt. Multiscale scientific computation: review 2001. In *Multiscale and multiresolution methods*, volume 20 of *Lect. Notes Comput. Sci. Eng.*, pages 3–95. Springer, Berlin, 2002.

[5] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A multigrid tutorial*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, second edition, 2000.

[6] A. Buluç and J.R. Gilbert. Highly parallel sparse matrix-matrix multiplication. 2010.

[7] John Burkardt. The finite element basis for simplices in arbitrary dimensions. Technical report, Florida State University, 2011.

[8] S.J. Chapman. *Fortran 95/2003 for Scientists & Engineers*. McGraw-Hill, 2007.

[9] Bhaskar DasGupta. *Applied Mathematical Methods*. Pearson Education, Delhi, second edition, 2006.

[10] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition, April 2008.

[11] R. P. Fedorenko. A relaxation method of solution of elliptic difference equations. *Ž. Vyčisl. Mat. i Mat. Fiz.*, 1:922–927, 1961.

[12] Joseph E. Flaherty. Finite element analysis. Course Notes, Rensselaer Polytechnic Institute, `http://www.cs.rpi.edu/~flaherje/feaframe.html`, 2005. Online; accessed 04/08/2012.

[13] Wolfgang Hackbusch. Ein iteratives verfahren zur schnellen auflösung elliptischer randwertprobleme. Technical report, Mathematisches Institut, Universität zu Köln, 1976.

[14] Wolfgang Hackbusch. *Multigrid methods and applications*, volume 4 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, 1985.

[15] Van Emden Henson and Ulrike Meier Yang. BoomerAMG: a parallel algebraic multigrid solver and preconditioner. *Appl. Numer. Math.*, 41(1):155–177, 2002. Developments and trends in iterative methods for large systems of equations—in memoriam Rüdiger Weiss (Lausanne, 2000).

[16] J. J. Heys, T. A. Manteuffel, S. F. McCormick, and L. N. Olson. Algebraic multigrid for higher-order finite elements. *J. Comput. Phys.*, 204(2):520–532, 2005.

[17] Yun-qing Huang, Shi Shu, and Xi-jun Yu. Preconditioning higher order finite element systems by algebraic multigrid method of linear elements. *J. Comput. Math.*, 24(5):657–664, 2006.

[18] Claes Johnson. *Numerical solution of partial differential equations by the finite element method*. Dover Publications Inc., Mineola, NY, 2009. Reprint of the 1987 edition.

[19] Rolf Klein. *Algorithmische Geometrie : Grundlagen, Methoden, Anwendungen (eXamen.press)*. Springer, March 2005.

[20] Matthias Maischak. Manual of the software package maiprogs. Technical report, Institut für Angewandte Mathematik, Universität Hannover, 2001.

[21] Matthias Maischak. Technical manual of the program system maiprogs. Technical report, Institut für Angewandte Mathematik, Universität Hannover, 2006.

[22] Matthias Maischak. B.O.N.E. - Book of numerical experiments. Preprint, Institut für Angewandte Mathematik, Universität Hannover, SISCM, Brunel University, `http://people.brunel.ac.uk/mastmmm/bone.pdf`, June 2012. Online; accessed 27/08/2012.

[23] J. Mandel, M. Brezina, and P. Vaněk. Energy optimization of algebraic multigrid bases. *Computing*, 62(3):205–228, 1999.

[24] Stephen F. McCormick, editor. *Multigrid methods*, volume 3 of *Frontiers in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1987.

[25] M. Metcalf, J. Reid, and M. Cohen. *Modern Fortran Explained*. Numerical Mathematics and Scientific Computation. OUP Oxford, 2011.

[26] M. Metcalf, J.K. Reid, and M. Cohen. *Fortran 95/2003 Explained*, volume 416. Oxford University Press New York, 2004.

[27] J. W. Ruge and K. Stüben. Algebraic multigrid. In *Multigrid methods*, volume 3 of *Frontiers Appl. Math.*, pages 73–130. SIAM, Philadelphia, PA, 1987.

[28] Yousef Saad. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, second edition, 2003.

[29] M.S. Shephard, S. Dey, and J.E. Flaherty. A straightforward structure to construct shape functions for variable p-order meshes. *Computer Methods in Applied Mechanics and Engineering*, 147(3):209–233, 1997.

[30] Jonathan Richard Shewchuk. *Delaunay Refinement Mesh Generation*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1997. Available as Technical Report CMU-CS-97-137.

[31] S. Shu, D. Sun, and J. Xu. An algebraic multigrid method for higher-order finite element discretizations. *Computing*, 77(4):347–377, 2006.

[32] Shi Shu, Jinchao Xu, Yingxiong Xiao, and Ludmil Zikatanov. Algebraic multigrid method on lattice block materials. In *Recent progress in computational and applied PDEs (Zhangjiajie, 2001)*, pages 289–307. Kluwer/Plenum, New York, 2002.

[33] U. Trottenberg, C. W. Oosterlee, and A. Schüller. *Multigrid*. Academic Press Inc., San Diego, CA, 2001. With contributions by A. Brandt, P. Oswald and K. Stüben.

[34] P. Vaněk, J. Mandel, and M. Brezina. Algebraic multigrid on unstructured meshes. Technical report, 1994.

[35] P. Vaněk, J. Mandel, and M. Brezina. Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. *Computing*, 56(3):179–196, 1996. International GAMM-Workshop on Multi-level Methods (Meisdorf, 1994).

[36] P. Vaněk, J. Mandel, and M. Brezina. Energy optimization of algebraic multigrid bases. *Computing*, 62(3):205–228, 1999.

[37] P. Vaněk, J. Mandel, and M. Brezina. Convergence of algebraic multigrid based on smoothed aggregation. *Numer. Math.*, 88(3):559–579, 2001.

[38] E. B. Vinberg. *A course in algebra*, volume 56 of *Graduate Studies in Mathematics*. American Mathematical Society, Providence, RI, 2003. Translated from the 2001 Russian original by Alexander Retakh.

[39] C. Wagner and G. Wittum. Filtering decompositions with respect to adaptive test vectors. *Multigrid Method V, Lect. Notes Comput. Sci. Eng*, 3:320–334, 1996.

[40] Christian Wagner. Tangential frequency filtering decompositions for unsymmetric matrices. *Numerische Mathematik*, 78(1):143–163, 1997.

[41] Christian Wagner. Introduction to algebraic multigrid. Lecture Notes, University of Heidelberg, `http://www.mgnet.org/mgnet/papers/Wagner/amgV11.ps.gz`, 1998. Online; accessed 12/05/2012.

[42] W. L. Wan, Tony F. Chan, and Barry Smith. An energy-minimizing interpolation for robust multigrid methods. *SIAM J. Sci. Comput.*, 21(4):1632–1649, 1999/00.

[43] Junxian Wang, Shi Shu, and Liuqiang Zhong. Efficient parallel preconditioners for high-order finite element discretizations of $H(grad)$ and $H(\mathbf{curl})$ problems. In *Domain decomposition methods in science and engineering XIX*, volume 78 of *Lect. Notes Comput. Sci. Eng.*, pages 325–332. Springer, Heidelberg, 2011.

[44] Pieter Wesseling. *An introduction to multigrid methods*. Pure and Applied Mathematics (New York). John Wiley & Sons Ltd., Chichester, 1992.

[45] Ulrike Meier Yang. Parallel algebraic multigrid methods—high performance preconditioners. In *Numerical solution of partial differential equations on parallel computers*, volume 51 of *Lect. Notes Comput. Sci. Eng.*, pages 209–236. Springer, Berlin, 2006.

[46] O. C. Zienkiewicz, R. L. Taylor, and J.Z. Zhu. *The Finite Element Method: Its Basis and Fundamentals, Sixth Edition*. Butterworth-Heinemann, 2005.

Index