TECHNISCHE UNIVERSITÄT CHEMNITZ

Department of Mathematics

Research Group Numerical Mathematics (Partial Differential Equations)

Diploma Thesis

Implementation of a Geometric Multigrid Method for FEniCS and its Application

Felix Ospald

http://launchpad.net/fmg

Chemnitz, February 8, 2013

Advisor: Dr. rer. nat. Gerd Wachsmuth

Examiner: Prof. Dr. Roland Herzog,
Dr. rer. nat. Gerd Wachsmuth

Ospald, Felix
Implementation of a Geometric Multigrid Method for FEniCS and its Application
Diploma Thesis, Department of Mathematics
Technische Universität Chemnitz, März 2013

Abstract

The FEniCS project is a collection of free software components for the automated, efficient solution of partial differential equations (PDE). For a given variational problem, FEniCS takes care of the FEM discretization, assembly of the system of linear equations and the application of boundary conditions. For the solution of such a system, FEniCS provides a number of commonly used solvers and preconditioners, incorporated by 3rd party libraries. Here, multigrid solvers/preconditioners have become popular for the solution of elliptic PDE in two or more dimensions. Instead of solving the problem directly, a number of coarser problems is solved, using a hierarchy of problem discretizations. Restriction and prolongation operators are used to transfer coefficient vectors between two different discretization levels, which require a certain geometrical information about the problem. Algebraic multigrid methods (AMG) however, construct their hierarchy of operators directly from the system matrix, without any geometric interpretation. Because of their ease of use, this kind of black-box solvers have become popular. They are used in FEniCS, by simply providing the system matrix to a 3rd party library, like Hypre AMG. However, since all geometrical problem background is available for FEniCS, it seems to be natural to use a geometric multigrid method (GMG), which makes use of the geometrical problem background and does not to drop it.

In this thesis, we present a GMG implementation for FEniCS, called `FMG`. The implementation is used to solve/precondition a Poisson problem, a problem from linear elasticity and two Stokes flow problems. The performance of `FMG` is analyzed in an automated way and it is shown that the computation time can be reduced considerably in contrast to other solvers, when using `FMG`. Thereby, we also show that a GMG implementation can be realized in FEniCS, and is worth to be added to FEniCS.

Table of contents

List of figures iv

List of tables v

1 Introduction 1

2 Mathematical preliminaries 3

2.1 Basic definitions . . . 3
2.1.1 Vector spaces . . . 3
2.1.2 Normed spaces . . . 3
2.1.3 Bilinear forms . . . 4
2.1.4 Hilbert spaces . . . 5
2.2 Elliptic boundary value problems . . . 6
2.2.1 Introduction . . . 6
2.2.2 Existence and uniqueness of weak solutions . . . 7
2.3 The finite element method . . . 10
2.3.1 Galerkin and Ritz methods . . . 10
2.3.2 Construction of finite element spaces . . . 12
Basic definitions . . . 13
The local-to-global mapping . . . 14
The global function space . . . 14
Affine equivalence . . . 15
2.3.3 Finite element examples . . . 15
The Lagrange elements . . . 15
The Crouzeix-Raviart element . . . 18
2.3.4 Treatment of Dirichlet boundary conditions . . . 19
2.4 Solvers for large, sparse linear systems . . . 20
2.4.1 Splitting methods . . . 21
The Richardson method . . . 22
The Jacobi method . . . 23
The Gauss-Seidel method . . . 23
Successive over-relaxation (SOR) . . . 23
Symmetric successive over-relaxation (SSOR) . . . 24
2.4.2 Krylov subspace methods . . . 24

The conjugate gradient method . . . 25
The preconditioned conjugate gradient method . . . 27
The minimum residual method . . . 28
2.5 Geometric Multigrid . . . 29
2.5.1 Introduction . . . 29
2.5.2 The two-grid method . . . 35
2.5.3 The multi-grid method . . . 35
2.5.4 Finite element multigrid . . . 37
2.5.5 Canonical prolongation and restriction operators . . . 37
2.5.6 Galerkin coarse grid approximation . . . 40
2.5.7 Convergence of the V-cycle . . . 41
2.5.8 Multigrid preconditioned CG . . . 42

3 Implementation **45**
3.1 Overview . . . 45
3.1.1 Implementation tasks . . . 45
3.1.2 Design goals . . . 46
3.1.3 FEniCS and its components . . . 46
FEniCS . . . 46
DOLFIN . . . 47
PETSc . . . 58
Boost C++ libraries . . . 59
3.1.4 FMG Project structure . . . 60
3.2 Problem and mesh refinement - The fmg::Adaptor class . . . 61
3.3 Prolongation - The fmg::ProlongationAssembler class . . . 64
3.4 Smoothing/relaxation - The fmg::*SmoothingOperator classes . . . 68
3.5 Cycle patterns - The fmg::*CyclePattern classes . . . 69
3.6 Multigrid levels - The fmg::MultigridLevel class . . . 72
3.7 Multigrid problems - The fmg::MultigridProblem class . . . 72
3.8 Multigrid solver - The fmg::MultigridSolver class . . . 73
3.9 Multigrid preconditioning - The fmg::MultigridPreconditioner class . . . 79
3.10 Multigrid preconditioned Krylov solver - The fmg::MultigridPreconditionedKrylovSolver class . . . 79
3.11 Other classes . . . 80
3.11.1 The fmg::FMGTimer class . . . 80
3.11.2 The fmg::Table class . . . 81
3.11.3 The fmg::Utils class . . . 82
3.12 Problem testing - The fmg::Tests class . . . 82

4 Application and results **85**
4.1 Poisson problem . . . 85
4.1.1 Initialization timings . . . 86
4.1.2 Convergence behaviour . . . 87
4.1.3 Different smoothers . . . 92
4.1.4 Different cycle patterns . . . 92
4.1.5 Different coarse grid sizes . . . 94
4.1.6 Different coarse level solvers . . . 94
4.1.7 Execution timings . . . 96
4.1.8 Comparison to FEINS . . . 96
4.2 Linear elasticity . . . 99
4.2.1 Initialization timings . . . 101
4.2.2 Convergence behaviour . . . 103
4.2.3 Different smoothers . . . 108
4.2.4 Different cycle patterns . . . 108
4.2.5 Different coarse level solvers . . . 108
4.2.6 Execution timings . . . 110
4.3 Preconditioning of the Stokes problem . . . 111
4.3.1 Lid-driven cavity . . . 113
4.3.2 Pipe with obstacles . . . 118
4.4 Prolongation operator test . . . 123
4.5 A note on performance improvement . . . 125

5 Conclusions and outlook **127**

Bibliography **129**

Appendix **133**
A.1 FEINS modifications . . . 133

List of figures

2.1 Element domain and degree of freedom for Lagrange elements. 16
2.2 Affine transformation from reference element. 17
2.3 Example for a global basis function for 1st order Lagrange elements. . 17
2.4 Element domain and degrees of freedom for Crouzeix-Raviart elements. 19
2.5 Eigenvectors of Δ for $N = 39$, including the zero boundary values. . . 31
2.6 Eigenvalues of Δ for $N = 39$. 32
2.7 Error scaling factors s_i for $N = 39$ and different relaxation parameters. 32
2.8 Residual scaling factors $h^2\lambda_i s_i$ for $N = 39$ and different relaxation. . . 33
2.9 V- and W-cycle for 6 levels. 36
2.10 Nested iteration (F-cycle) for 6 levels. 36
2.11 Commutative diagram for the prolongation and restriction operators. . 38
2.12 Commutative diagram for the local prolongation operator p_T. 39

3.1 A schematic overview of the FEniCS components and their interplay. . 47
3.2 Solution of the Poisson problem. 52
3.3 Components of the PETSc library. 58
3.4 File and directory structure of the `FMG` project. 60
3.5 Class hierarchical view of the project. 61
3.6 Graph for of the `VCyclePattern` for 6 levels. 71
3.7 Schema of the `MultigridLevel` class. 72
3.8 Schema of the `MultigridProblem` class. 73

4.1 Plot of the residuals for the Poisson problem. 90
4.2 Plot of the solve time for the Poisson problem. 91
4.3 Different test cycle patterns. 93
4.4 Different coarse grid sizes and corresponding cycle patterns. 95
4.5 Plot of the solution for the FEINS test problem. 98
4.6 Domain Ω and initial mesh for the elasticity problem. 99
4.7 Solution of the elasticity problem after 0, 1, 2 and 3 refinements. . . . 102
4.8 Plot of the residuals for the elasticity problem. 106
4.9 Plot of the solve time for the elasticity problem. 107
4.10 Plot of the solve times for the ”lid-driven cavity“ problem. 116
4.11 Velocity plot with flow lines for the ”lid-driven cavity” problem. 117
4.12 Pressure plot for the ”lid-driven cavity“ problem. 117

4.13 Initial mesh for the ”pipe with obstacles“ problem. 118
4.14 Solve times for the ”pipe with obstacles“ problem. 121
4.15 Velocity plot with flow lines for the ”pipe with obstacles“ problem. . . 122
4.16 Pressure plot for the ”pipe with obstacles” problem. 122

List of tables

3.1 Example patterns used as argument for the `VectorCyclePattern` class. 71
3.2 Prarameters for the `MultigridSolver` class. 77
3.3 Prarameters for the `Tests` class. 83

4.1 Multigrid problem initialization timings for the Poisson problem. . . . 87
4.2 Detailed problem initialization timings for the Poisson problem. 88
4.3 Iteration count and solve time for the Poisson problem. 89
4.4 Iteration count and solve time of LU and CG for the Poisson problem. 89
4.5 Relative residuals for the Poisson problem. 90
4.6 Effect of different smoothers for the Poisson problem. 92
4.7 Effect of different cycle patterns for the Poisson problem. 93
4.8 Effect of different coarse level solver for the Poisson problem. 94
4.9 Effect of different coarse grid sizes for the Poisson problem. 95
4.10 Execution timings of the multigrid algorithm for the Poisson problem. 96
4.11 Obtained convergence results for a Poisson problem, using FEINS. . . 98
4.12 Multigrid problem initialization timings for the elasticity problem. . . 101
4.13 Detailed problem initialization timings for the elasticity problem. . . . 104
4.14 Iteration count and solve time for the elasticity problem. 105
4.15 Iteration count and solve time of LU and CG for the elasticity problem.105
4.16 Relative residuals for the elasticity problem. 106
4.17 Effect of different smoothers for the elasticity problem. 108
4.18 Effect of different cycle patterns for the elasticity problem. 109
4.19 Effect of different coarse level solver for the elasticity problem. 109
4.20 Execution timings for the elasticity problem. 110
4.21 System size and initialization time for the ”lid-driven cavity“ problem. 114
4.22 Performance of the ”lid-driven cavity“ problem, using `FMG`. 114
4.23 Performance of the ”lid-driven cavity“ problem, using LU. 115
4.24 Performance of the ”lid-driven cavity“ problem, using Hypre AMG. . . 115
4.25 Performance of the ”lid-driven cavity“ problem, using ML AMG. . . . 116

4.26 System size and initialization time for the ”pipe with obstacles“ problem. 119
4.27 Performance of the ”pipe with obstacles“ problem, using `FMG`. 119
4.28 Performance of the ”pipe with obstacles“ problem, using LU. 120
4.29 Performance of the ”pipe with obstacles“ problem, using Hypre AMG. 120
4.30 Performance of the ”pipe with obstacles“ problem, using ML AMG. . . 121
4.31 Results for the prolongation operator test. 124

List of algorithms

2.1 The conjugate gradient (CG) method. 26
2.2 The symmetric preconditioned conjugate gradient method. 27
2.3 The preconditioned conjugate gradient (PCG) method. 28
2.4 The two-grid method $TGM(x, b)$. 35
2.5 The multi-grid method $MGM(\ell, x, b)$. 36

1 Introduction

One very common task in engineering is to solve linear systems, arising from the discretization of *partial differential equations* (PDE). Not so long ago, the Richardson, Jacobi and Gauss-Seidel method ("relaxation"), where the only tools for solving such systems iteratively. However, these methods often had to be fitted to a particular problem and were not very effective for large systems. An first improvement was the *successive overrelaxation method* (SOR), developed in 1950 by Young [You50] and Frankel [Fra50] simultaneously. In 1952 Hestenes and Stiefel [HS52] developed the *conjugate gradient method* (CG), which also works for systems that are too large to be handled by direct methods. In 1961 Fedorenko [Fed61] formulated the first multigrid method and showed the convergence behaviour for Poisson's equation on a square. He "discovered" this method, when he had to solve a 2D hydrodynamics equations for doing weather forecast [Fed01]. For the numerical integration of these equations, a Poisson equation in a rectangular domain had to be solved for each time step. He tried to solve the problem on an 48x40 grid using an M-20 computer (4096 addresses of memory, 20000 flops, 50 kW power consumption, 170-200m^2 occupied area), such that the right hand side and solution vector used almost all memory of the computer. Apparently Fedorenko was not aware of the SOR method and other methods available at that time, instead he tried to solve the problem using a simple relaxation method. By trying to figure out, what the slow convergence caused, he observed that after one iteration the residual decreased fast and became smooth. After this first decrease the progress was desperately. At this point, he had the idea to switch to a coarser grid and solve for the error on this grid, which was justified by the smoothness of the right hand side (the residual). Because of the grid dependency, such a method is today known as *geometric multigrid* (GMG).

In 1972 to 1977 GMG was further improved by Brandt (adaptive multigrid, [Bra77]) et al. and several convergence proofs were done. In 1976 Hackbusch [Hac76] rediscovered the multigrid method independently. He generalized previous results and published software about his methods. After the publications of Hemker [Hem80], Trottenberg, Stüben et al. in 1980, the number of publications increased rapidly (cf. [Hac85; McC87; Wes92; Sha95; TS01; Bra03]).

Today, multigrid methods are still among the most efficient methods for solving such large linear systems. Solvers are available in numerous software libraries. However, many of these libraries implement *algebraic multigrid methods* (AMG), which in contrast to GMG construct their hierarchy of operators directly from the system matrix, without using any geometric information. Because of their ease of use and easy

integrability into any *problem solving environment* (PSE), these kind of black-box solvers have become very popular.

The FEniCS Project [LMW+12] is one of such PSE. It is a collection of free software components for the automated, efficient solution of PDE using the *finite element method* (FEM). For a given variational problem, FEniCS takes care of the FEM discretization, assembly of the system of linear equations and the application of boundary conditions. For the solution, FEniCS provides a number of commonly used solvers and preconditioners, incorporated by 3rd party libraries like PETSc [Bal+12b], Hypre's BoomerAMG [HY00] or Sandia's ML AMG [Gee+06]. However, since all geometrical problem background is available for FEniCS, it seems to be natural to use a geometric multigrid method (GMG), which makes use of the geometrical problem background and does not to drop it.

In this thesis, we present a GMG implementation for FEniCS, called `FMG` (not to be confused with full multigrid). The implementation is used to solve/precondition a Poisson problem, a problem from linear elasticity and two Stokes flow problems. The performance of `FMG` is analyzed in an automated way and it is shown that the computation time can be reduced considerably in contrast to other solvers, when using `FMG`. Thereby, we also show that a GMG implementation can be realized in FEniCS, and is worth to be added to FEniCS.

The outline of the diploma thesis is as follows:

- In chapter 2, we give the mathematical preliminaries for the discretization of elliptic boundary value problems using the FEM, and we will discuss basic iterative solvers, as well as CG and GMG for the solution of the arising linear systems of equations. Some basic convergence results are given for GMG.
- In chapter 3, the implementation and structure of `FMG` is described. Here, we give also an short introduction to FEniCS and the used software libraries.
- In chapter 4, the performance of `FMG` as solver and/or preconditioner for some basic problems is discussed.
- In chapter 5, the thesis is concluded and we give an outlook, regarding possible future work on `FMG`.

2 Mathematical preliminaries

2.1 Basic definitions

2.1.1 Vector spaces

Definition 2.1 (K-Vector space)**.** A *vector space* V over a field $(K, +, \cdot)$ is an Abelian group $(V, +)$, which additionally defines a so called *scalar multiplication* $\cdot\colon K \times V \to V$, which needs to satisfy

$$\text{(i)}\ \ \alpha \cdot (\beta \cdot v) = (\alpha \cdot \beta) \cdot v, \tag{2.1}$$
$$\text{(ii)}\ \ \alpha \cdot (u + v) = \alpha \cdot u + \alpha \cdot v, \tag{2.2}$$
$$\text{(iii)}\ \ (\alpha + \beta) \cdot v = \alpha \cdot v + \beta \cdot v, \tag{2.3}$$
$$\text{(iv)}\ \ 1 \cdot v = v, \quad \text{(where 1 denotes the identity element of } K) \tag{2.4}$$

for all $u, v \in V$ and α, $\beta \in K$.

Note: In this thesis K is usually $\mathbb{R}$ and the $\cdot$ is usually omitted. The operator $+$ is the usual addition of vectors.

Definition 2.2 (Inner product)**.** A *inner product* or *scalar product* $(\cdot, \cdot) : V \times V \to \mathbb{R}$ on a real valued vector space V is a symmetric positive definite bilinear form (see section 2.1.3).

2.1.2 Normed spaces

Definition 2.3 (Norm)**.** For a real or complex valued K-vector space V the *norm* is a mapping $\|.\| : V \to \mathbb{R}_+$, which satisfies

$$\text{(i)}\ \ \|u\| = 0 \ \Rightarrow\ u = 0, \quad \text{(definiteness)} \tag{2.5}$$
$$\text{(ii)}\ \ \|\alpha \cdot u\| = |\alpha| \cdot \|u\|, \quad \text{(positive homogeneity)} \tag{2.6}$$
$$\text{(iii)}\ \ \|u + v\| \leq \|u\| + \|v\|, \quad \text{(triangle inequality or subadditivity)} \tag{2.7}$$

for all $u, v \in V$ and $\alpha \in K$.

Definition 2.4 (Normed vector space)**.** A *normed vector space* is a pair $(V, \|.\|)$, where V is a vector space and $\|.\|$ is a norm on V.

Definition 2.5 (Banach space)**.** A *Banach space* is a normed vector space $(V, \|.\|)$, which is complete with respect to its norm.

Definition 2.6 (Induced vector norm)**.** For a real or complex valued vector space V equipped with a scalar product $(\cdot,\cdot)$, the *induced vector norm* $\|.\|$ is defined by

$$\|v\| := \sqrt{(v,v)}. \tag{2.8}$$

Definition 2.7 (Equivalence of norms)**.** Two norms $\|.\|_a$ and $\|.\|_b$ for a vector space V are said to be equivalent, if

$$\exists\, c_1, c_2 > 0 \quad \text{such that} \quad c_1\|v\|_b \leq \|v\|_a \leq c_2\|v\|_b, \quad \forall v \in V. \tag{2.9}$$

2.1.3 Bilinear forms

Definition 2.8 (Bilinear form)**.** A *bilinear form* on a K-vector space V is a mapping $a : V \times V \to K$, which is linear in each argument, i.e.

$$\begin{aligned} &\text{(i)} \quad a(u+v,w) = a(u,w) + a(v,w) && (2.10),\\ &\text{(ii)} \quad a(u,v+w) = a(u,v) + a(u,w) && (2.11),\\ &\text{(iii)} \quad a(\lambda u,v) = a(u,\lambda v) = \lambda a(u,v) && (2.12), \end{aligned}$$

for all $u, v \in V$ and α, $\beta \in K$.

Note: The above definition is usually used for real valued vector spaces. For complex valued vector spaces sesquilinear forms are used.

Definition 2.9 (Positive definiteness)**.** A bilinear form $a : V \times V \to \mathbb{R}$ on a real valued vector space V is said to be *positive definite*, if

$$a(v,v) \geq 0, \quad \forall v \in V \quad \text{and} \tag{2.13}$$

$$a(v,v) = 0 \Leftrightarrow v = 0. \tag{2.14}$$

Definition 2.10 (Symmetry)**.** A bilinear form $a : V \times V \to \mathbb{R}$ on a real valued vector space V is said to be *symmetric*, if

$$a(u,v) = a(v,u), \quad \forall u, v \in V. \tag{2.15}$$

Definition 2.11 (V-ellipticity)**.** A bilinear form $a : V \times V \to \mathbb{R}$ on a real valued vector space V with norm $\|\cdot\|$ is said to be *V-elliptic* or *coercive*, if

$$\exists\, \alpha > 0 \quad \text{such that} \quad a(v,v) \geq \alpha\|v\|^2 \quad \forall v \in V. \tag{2.16}$$

Lemma 2.12. A V-elliptic bilinear form $a(\cdot,\cdot)$ is positive definite.

Proof. From definition 2.11 it follows that $a(v,v) \geq 0\ \forall v \in V$. Let $a(v,v) = 0$ then $\exists\ \alpha > 0$ such that $0 \geq \alpha\|v\|^2\ \ \forall v \in V$. It follows that $\|v\|^2 = 0$ and by definition 2.3(i) $v = 0$. Let $v = 0$ then by definition 2.8(iii) it follows $a(v,v) = a(0v,v) = 0a(v,v) = 0$. □

2.1.4 Hilbert spaces

Definition 2.13 (Hilbert space)**.** A *Hilbert space* H is a real or complex valued vector space V equipped with a scalar product $(\cdot,\cdot)$, which is complete regarding the norm induced by the scalar product.

Note: The induced norm $\sqrt{(\cdot,\cdot)}$ for a Hilbert space H will be denoted by $\|.\|_H$.

Theorem 2.14 (Riesz representation theorem)**.** Let H be a Hilbert space with inner product $(\cdot,\cdot)$. Then for every $f \in H'$ there exists exactly one $y \in H$, such that

$$f(x) = (x, y), \quad \forall\, x \in H \quad \text{and} \tag{2.17}$$

$$\|f\|_{H'} = \|y\|_H. \tag{2.18}$$

Further for every $y \in H$ the mapping

$$x \mapsto (x, y) \tag{2.19}$$

is a continuous functional with operator norm $\|y\|_H$.

Proof. See [Alt99]. □

Theorem 2.15 (Hilbert projection theorem)**.** Let H be a Hilbert space and U a closed convex subset of H. Then for every $x \in H$ there exists a unique $y \in U$ with

$$\|x - y\|_H = \inf_{u \in U} \|x - u\|_H. \tag{2.20}$$

Proof. See [Alt99]. □

2.2 Elliptic boundary value problems

A *boundary value problem* is a differential equation, together with a set of additional constraints, called the *boundary conditions*. Such problems arise generally in all kind of physical phenomena. Assertions about the existence and uniqueness of the solution for such problems are essential. For simplicity we restrict ourselves to linear elliptic partial differential equations.

2.2.1 Introduction

In the following we assume that $\Omega \subset \mathbb{R}^d$ is a domain (i.e. a bounded and open set), with Lipschitz continuous boundary $\Gamma = \partial\Omega$. As model problem, we consider the Poisson equation with homogeneous Dirichlet boundary conditions, i.e. find u such that

$$-\Delta u = \tilde{f} \quad \text{in } \Omega, \tag{2.21}$$

$$u = 0 \quad \text{on } \Gamma. \tag{2.22}$$

For $\tilde{f} \in C(\Omega)$, a *classical solution* of this problem is a function $u \in C^2(\Omega) \cap C(\overline{\Omega})$, which satisfies the above equation and the boundary conditions pointwise. In general, the existence of solutions for such problems is hard to show. The introduction of *weak solutions* allows us to remove some of the smoothness requirements. This is done by multiplying eq. (2.21) with a smooth *test function* $v \in C_0^\infty(\Omega)$ and integrating over the domain Ω:

$$-\int_\Omega \Delta u v \,\mathrm{d}x = \int_\Omega \tilde{f} v \,\mathrm{d}x. \tag{2.23}$$

Integration by parts (Green's identities) gives further

$$-\int_\Omega \Delta u v \,\mathrm{d}x = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x - \int_\Gamma \frac{\partial u}{\partial n} v \,\mathrm{d}s, \tag{2.24}$$

where the last integral is zero, since $v = 0$ on Γ. Now, the preliminary weak formulation for eq. (2.21) reads as: find $u \in C^1(\Omega) \cap C(\overline{\Omega})$ such that

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x = \int_\Omega \tilde{f} v \,\mathrm{d}x, \quad \forall v \in C_0^\infty(\Omega). \tag{2.25}$$

However, since the space $C_0^\infty(\Omega)$ is not closed (e.g. there exists a sequence of smooth functions, which have a non smooth limit function), the mathematical discussion of the existence and uniqueness of solutions is difficult within this framework (without Banach spaces). Therefore, it seems to be natural to extend the test space and use the closure of $C_0^\infty(\Omega)$, with respect to a suitable norm. *Sobolev spaces* were originally designed for this purpose. They have the essential property, that $C_0^\infty(\Omega)$ is dense in

the Sobolev space $H_0^1(\Omega)$. Therefore, the *weak formulation* for eq. (2.21) reads as: find $u \in H_0^1(\Omega)$, such that

$$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x = \int_\Omega \tilde{f} v \, \mathrm{d}x, \quad \forall v \in H_0^1(\Omega). \tag{2.26}$$

Defining $V := H_0^1(\Omega)$, this may be written, more generally as

$$a(u, v) = f(v), \quad \forall v \in V, \tag{2.27}$$

with the bilinear form

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x, \tag{2.28}$$

and the linear form

$$f(v) = \int_\Omega \tilde{f} v \, \mathrm{d}x. \tag{2.29}$$

Every $u \in V$, which satisfies eq. (2.27) is called a *weak solution* to eq. (2.21).

2.2.2 Existence and uniqueness of weak solutions

This section is about the existence and uniqueness of finding a solution $u \in U$ to eq. (2.27), by considering the minimization problem

$$J(u) = \inf_{v \in U} J(v), \tag{2.30}$$

where U is a nonempty subset of a real valued vector space V with norm $\|.\|$, and the functional $J : V \to \mathbb{R}$ is defined by

$$J(v) = \frac{1}{2} a(v, v) - f(v), \tag{2.31}$$

where $a(\cdot, \cdot) : V \times V \to \mathbb{R}$ is a continuous, bilinear form and $f : V \to \mathbb{R}$ is a continuous linear form.

Theorem 2.16 (Existence and uniqueness)**.** Further assume that

(i) the space V is complete,
(ii) U is a closed convex subset of V,
(iii) the bilinear form $a(\cdot, \cdot)$ is symmetric and V-elliptic,

then there exists a unique solution for problem eq. (2.30).

Proof. By lemma 2.12 the V-ellipticity of the bilinear form implies the positive definiteness. Together with the symmetry, $a(\cdot,\cdot)$ defines an inner product (definition 2.2).

The induced norm $\|.\|_a = \sqrt{a(\cdot,\cdot)}$ is equivalent to the given norm $\|.\|$ on V, since from definition 2.11 it follows that $\exists\ c_1 > 0$, such that $a(v,v) \geq c_1\|v\|^2\ \forall v \in V$, and hence $\sqrt{c_1}\|v\| \leq \|v\|_a\ \forall v \in V$. On the other hand, it follows from the continuity of the bilinear form that $\exists\ c_2 > 0$, such that $\|v\|_a \leq c_2\|v\|\ \forall v \in V$. Therefore, V together with $\|.\|_a$ is also complete and hence a Hilbert space.

By theorem 2.14 there exists a $f_r \in V$ such that

$$f(v) = a(v, f_r), \quad \forall v \in V. \tag{2.32}$$

Using the symmetry of $a(\cdot,\cdot)$, the functional J may now be rewritten as

$$\begin{aligned} J(v) &= \frac{1}{2}a(v,v) - a(v,f_r) \\ &= \frac{1}{2}a(v, v-f_r) - \frac{1}{2}a(v,f_r) \\ &= \frac{1}{2}a(v-f_r, v-f_r) + \frac{1}{2}a(f_r, v-f_r) - \frac{1}{2}a(f_r,v) \\ &= \frac{1}{2}\|v-f_r\|_a^2 - \frac{1}{2}\|f_r\|_a^2. \end{aligned} \tag{2.33}$$

Since U is a closed convex subset of V it follows from theorem 2.15 that there exists a unique solution $u \in U$ with

$$J(u) = \inf_{v\in U} J(v) = \frac{1}{2}\inf_{v\in U}\|v-f_r\|_a^2 - \frac{1}{2}\|f_r\|_a^2 = \frac{1}{2}\|u-f_r\|_a^2 - \frac{1}{2}\|f_r\|_a^2. \tag{2.34}$$

□

If we confine ourselves to the case where $U = V$ and drop the assumption of the symmetry of the bilinear form, then existence and uniqueness can still be proven in the case of V is a Hilbert space. But no minimization problem, as eq. (2.30) is associated with this proof.

Theorem 2.17 (Lax-Milgram lemma)**.** Let V be a Hilbert space, $a(\cdot,\cdot) : V \times V \to \mathbb{R}$ be a continuous, V-elliptic bilinear form, and let $f : V \to \mathbb{R}$ be a continuous linear form. Then the variational problem: Find an $u \in V$ such that

$$a(u,v) = f(v), \quad \forall v \in V, \tag{2.35}$$

has a unique solution.

Proof. Since $a(u,v)$ is continuous for fixed $u \in V$ or fixed $v \in V$, there exists a constant M such that

$$|a(u,v)| \leq M\|u\|\|v\|, \quad \forall u, v \in V. \tag{2.36}$$

If we fix $u \in V$ then the linear form $v \in V \to a(u,v)$ is in V' and by theorem 2.14 there exists a unique $a_u \in V$ such that

$$a(u,v) = (v, a_u), \quad \forall v \in V. \tag{2.37}$$

Thus we have

$$\|(\cdot, a_u)\|_{V'} = \sup_{v \in V} \frac{|(v,a_u)|}{\|v\|} \leq \sup_{v \in V} \frac{M\|u\|\|v\|}{\|v\|} = M\|u\|. \tag{2.38}$$

This means the linear mapping $A : u \in V \mapsto (\cdot, a_u) \in V'$ is continuous, with

$$\|A\|_{\mathcal{L}(V,V')} \leq M. \tag{2.39}$$

Let $\mathcal{R} : V \to V'$ denote the (bijective) Riesz mapping (cf. theorem 2.14) and let $\rho = \mathcal{R}^{-1}$, then for $f \in V'$

$$f(v) = (v, \rho f), \quad \forall v \in V. \tag{2.40}$$

Therefore, solving

$$a(u,v) = (v, a_u) = (v, \rho A(u)) = A(u)(v) = (v, \rho f) = f(v), \quad \forall v \in V \tag{2.41}$$

is equivalent to solving

$$\rho A(u) = \rho f. \tag{2.42}$$

We will show that this equation has a unique solution, by showing that the affine mapping

$$m : v \in V \mapsto v - \gamma(\rho A(v) - \rho f) \in V \tag{2.43}$$

is a contraction for appropriate values of the parameter $\gamma > 0$ and applying the Banach fixed point theorem. Therefore, observe that

$$\|m(w) - m(u)\| = (w-u) - \gamma\rho A(w-u), \quad \forall u, v \in V, \tag{2.44}$$

and setting $v := (w-u)$

$$\begin{aligned} \|v - \gamma\rho A(v)\|^2 &= \|v\|^2 - 2\gamma(v, \rho A(v)) + \gamma^2\|\rho A(v)\|^2 \\ &\leq \|v\|^2 - 2\gamma(v, a_v) + \gamma^2\|a_v\|^2 \\ &\leq (1 - 2\gamma\alpha + \gamma^2 M^2)\|v\|^2, \end{aligned} \tag{2.45}$$

since V-ellipticity gives an $\alpha > 0$ such that

$$(v, a_v) = a(v, v) \geq \alpha \|v\|^2, \quad \forall v \in V. \tag{2.46}$$

It follows that eq. (2.43) is a contraction whenever $(1 - 2\gamma\alpha + \gamma^2 M^2) \in [0, 1)$ is satisfied. Which is true for all

$$\gamma \in \left(0, \frac{2\alpha}{M^2}\right) \setminus \left(\frac{\alpha}{M^2} \mp \frac{1}{M}\sqrt{\max\left(0, \frac{\alpha^2}{M^2} - 1\right)}\right). \tag{2.47}$$

Since always $\alpha \leq M$ this is equivalent to having $\gamma \in \left(0, \frac{2\alpha}{M^2}\right)$. □

Lemma 2.18. Under the assumptions of theorem 2.17 and in the case of a symmetric bilinear form, the solutions of eq. (2.35) and eq. (2.30) are equivalent.

Proof. See [Cia78]. □

2.3 The finite element method

A common approach to solve eq. (2.35) arising from PDEs, is the *finite element method* (FEM). The development of the FEM began in the middle 1950s, mainly for problems in structural analysis. Today, it is applied in almost all fields of engineering and research. To make problems solvable, the first step is to replace the infinite dimensional Hilbert space V by a finite dimensional subspace $V_h \subset V$, which is then known as the *Galerkin discretization* of the problem. The purpose of the FEM is then to define basis functions of the space V_h in a clever way. This is done by subdividing the domain Ω into a finite number of elements (e.g. triangles in 2d) and define local basis functions for each element, called the *nodal basis*. These local basis functions can then be patched together to form the global basis functions of V_h. Due to the limited support of the basis functions, the arising linear system of equations is sparse and can be solved efficiently, even for a large number of unknowns, using methods as discussed in section 2.4. In this section, we give a brief introduction of the FEM, based on [Cia78; BS02; Wel06] to clarify the notation, used in subsequent sections and chapters.

2.3.1 Galerkin and Ritz methods

We consider the linear variational problem: Find an $u \in V$ such that

$$a(u, v) = f(v), \quad \forall v \in V, \tag{2.48}$$

under the assumptions of the Lax-Milgram lemma (theorem 2.17). The idea of the *Galerkin method* for approximating the solution of such a problem is, to define a

finite N-dimensional subspace $V_h \subset V$ and solve the so called *discrete problem*: Find $u_h \in V_h$ such that

$$a(u_h, v_h) = f(v_h), \quad \forall v_h \in V_h \tag{2.49}$$

Since $V_h \subset V$, the bilinear form is also continuous and V-elliptic regarding V_h and theorem 2.17 is applicable. Therefore, there exists a unique solution u_h of eq. (2.49), which is called the *discrete solution*.

Note: In the case of a symmetric bilinear form, the discrete solution is also characterized by eq. (2.30) (with min over V_h). This alternate definition of the discrete solution is known as the *Ritz method*.

For the approximate solution u_h, it is important to know how it is related to the original solution u of eq. (2.48). Therefore, the two most important results are given in the following.

Lemma 2.19 (Galerkin orthogonality)**.** Let u be the solution of eq. (2.48) and u_h the solution of eq. (2.49). Then

$$a(u - u_h, v_h) = 0, \quad \forall v_h \in V_h. \tag{2.50}$$

Proof.

$$a(u - u_h, v_h) = a(u, v_h) - a(u_h, v_h) = f(v_h) - f(v_h) = 0. \tag{2.51}$$

□

Lemma 2.20 (Céa's Lemma)**.** Let u be the solution of eq. (2.48) and u_h the solution of eq. (2.49). Under the assumptions of the Lax-Milgram lemma (theorem 2.17) and given the constants $\gamma > 0$ and $\alpha > 0$, such that

$$|a(u, v)| \leq \gamma \|u\| \|v\|, \quad \forall u, v \in V \text{ (continuity)} \tag{2.52}$$

$$a(v, v) \geq \alpha \|v\|^2, \quad \forall v \in V \text{ (V-ellipticity)} \tag{2.53}$$

then

$$\|u - u_h\| \leq \frac{\gamma}{\alpha} \inf_{v \in V_h} \|u - v\|. \tag{2.54}$$

Proof. Using the V-ellipticity, Galerkin orthogonality and continuity of $a(\cdot, \cdot)$:

$$\alpha \|u - u_h\|^2 \leq a(u - u_h, u - u_h) = a(u - u_h, u - v) + a(u - u_h, v - u_h) \tag{2.55}$$

$$= a(u - u_h, u - v) \leq \gamma \|u - u_h\| \|u - v\|, \quad \forall v \in V_h, \tag{2.56}$$

therefore

$$\|u - u_h\| \leq \frac{\gamma}{\alpha} \|u - v\|, \quad \forall v \in V_h. \tag{2.57}$$

The result follows, by taking the infimum over all $v \in V_h$ on both sides. □

In order to get a linear system of equations for eq. (2.49), we make the following Ansatz for the unknown $u_h \in V_h$

$$u_h = \sum_{i=1}^{N} u_i \phi_i, \tag{2.58}$$

where $\{\phi_i\}_{i=1}^N$ denotes a basis for V_h. Inserting this Ansatz into eq. (2.49), gives the problem: Find $u_i \in \mathbb{R}^n$ such that

$$\sum_{i=1}^{N} u_i a(\phi_i, v_h) = f(v_h), \quad \forall v_h \in V_h \tag{2.59}$$

Since, we have the same basis for $v_h \in V_h$, this is equivalent to

$$\sum_{i=1}^{N} u_i a(\phi_i, \phi_j) = f(\phi_j), \quad \forall j = 1, ..., N, \tag{2.60}$$

which is a system of linear equations

$$Au = b, \tag{2.61}$$

with $A \in \mathbb{R}^{N \times N}$ and $u, b \in \mathbb{R}^N$ and

$$A_{i,j} = a(\phi_j, \phi_i), \tag{2.62}$$
$$b_i = f(\phi_i). \tag{2.63}$$

The process of building the matrix A and vector b is known as *system assembly*. This includes also the numerical evaluation of integrals, contained in the forms a and f.

2.3.2 Construction of finite element spaces

Let $\Omega \subset \mathbb{R}^d$ be a domain, with Lipschitz-continuous boundary Γ. The construction of subspaces $V_h \subset V$ is characterized by the following aspects. Firstly the domain Ω is partitioned into a finite set of *cells*. All these cells together are called the *mesh* or *triangulation* of Ω. Typical shapes of the cells are intervals, triangles, quadrilaterals, tetrahedra or hexahedra. Secondly on each cell, we define local basis functions. These local basis functions are then patched together to form the global basis functions. Following [Cia78; BS02; LMW+12], we introduce some basic definitions.

Basic definitions

Definition 2.21 (Triangulation)**.** A set $\mathcal{T}_h$ is called *triangulation* of Ω, if

(i) $\overline{\Omega} = \bigcup_{T\in\mathcal{T}_h} T$.
(ii) All $T \in \mathcal{T}_h$ are closed and have a non-empty interior and Lipschitz-continuous boundary.
(iii) $\mathring{T}_1 \cap \mathring{T}_2 = \emptyset$ for any two distinct $T_1, T_2 \in \mathcal{T}_h$.

Definition 2.22 (Finite element)**.** A *finite element* in $\mathbb{R}^d$ is a triple $(T, \mathcal{P}, \mathcal{N})$ where

(i) T is a bounded closed subset of $\mathbb{R}^d$ ($d = 1, 2, 3, \dots$) with non empty interior and Lipschitz-continuous boundary (the *element domain*),
(ii) $\mathcal{P}$ is a n-dimensional function space of real valued functions on T (the space of *shape functions*),
(iii) $\mathcal{N} = \{\ell_1, \ell_2, \dots, \ell_n\}$ is a basis of the dual space $\mathcal{P}'$ (the set of *degrees of freedom* or *nodal variables*).

Definition 2.23 (Nodal basis)**.** Given a finite element $(T, \mathcal{P}, \mathcal{N})$, the basis $\{\phi_i\}_{i=1}^n$ of $\mathcal{P}$ dual to $\mathcal{N}$ is called the *nodal basis* of $\mathcal{P}$. This means

$$\ell_i(\phi_j) = \delta_{ij}, \quad \forall i, j = 1, \dots, n, \tag{2.64}$$

where δ_{ij} denotes the Kronecker delta function.

Definition 2.24 (Local interpolant)**.** Given a finite element $(T, \mathcal{P}, \mathcal{N})$ with nodal basis $\{\phi_i\}_{i=1}^n$ and $\mathcal{N} = \{\ell_i\}_{i=1}^n$. If v is a function on T, for which all $\ell_i(v)$ are well defined, then the *local interpolant* is defined by

$$\mathcal{I}_T v = \sum_{i=1}^n \ell_i(v)\phi_i. \tag{2.65}$$

Lemma 2.25. Given a finite element $(T, \mathcal{P}, \mathcal{N})$. Then for every $v \in \mathcal{P}$

$$\mathcal{I}_T v = v. \tag{2.66}$$

Proof.

$$v = \sum_{i=1}^n v_i\phi_i = \sum_{i=1}^n \sum_{j=1}^n \delta_{ij} v_i \phi_i = \sum_{i=1}^n \sum_{j=1}^n \ell_i(\phi_j) v_i \phi_i \tag{2.67}$$

$$= \sum_{i=1}^n \sum_{j=1}^n \ell_i(v_i\phi_j)\phi_i = \sum_{i=1}^n \ell_i\Big(\sum_{j=1}^n v_i\phi_j\Big)\phi_i = \sum_{i=1}^n \ell_i(v)\phi_i = \mathcal{I}_T v. \tag{2.68}$$

□

Note: The coefficients $\ell_i(v)$ are also called the *expansion coefficients of* v (with respect to the basis $\{\phi_i\}_{i=1}^n$).

The local-to-global mapping

Given a subdivision $\mathcal{T}_h$ of Ω we want to define the global function space $V_h = \operatorname{span}\{\phi_i\}_{i=1}^N$ on Ω from a given set $\{(T, \mathcal{P}_T, \mathcal{N}_T)\}_{T\in\mathcal{T}_h}$ of finite elements. Therefore, on each cell $T \in \mathcal{T}_h$ a *local-to-global mapping*

$$\iota_T : [1, n_T] \cap \mathbb{N} \to [1, N] \cap \mathbb{N} \tag{2.69}$$

is defined. This mapping specifies how the local degrees of freedom $\mathcal{N}_T = \{\ell_i^T\}_{i=1}^{n_T}$ of triangle T are mapped to the global degrees of freedom $\mathcal{N} = \{\ell_i\}_{i=1}^N$. That is, for any $v \in V_h$ the global degrees of freedom are defined by

$$\ell_{\iota_T(i)}(v) = \ell_i^T(v|_T), \quad i = 1, \dots, n_T, \tag{2.70}$$

and for all $T \in \mathcal{T}_h$. This means the global degrees of freedom are just associated with local degrees of freedom. The uniqueness of this definition is ensured in the following section.

The global function space

The *global function space* V_h is defined as the set of functions v over Ω, which satisfy:

(i) $v|_T \in \mathcal{P}_T,\ \forall T \in \mathcal{T}_h$ and (2.71)

(ii) for any $(T, T') \in \mathcal{T}_h \times \mathcal{T}_h$ and any $(i, i') \in ([1, n_T] \times [1, n_{T'}]) \cap (\mathbb{N} \times \mathbb{N})$ with $\iota_T(i) = \iota_{T'}(i')$ it holds that $\ell_i^T(v|_T) = \ell_{i'}^{T'}(v|_{T'})$. (2.72)

The first condition ensures that the restriction of any function $v \in V_h$ to the cell T is an element of the local function space $\mathcal{P}_T$. The latter says if any two local degrees of freedom map to the same global degree of freedom, then they have to evaluate to the same value for any function $v \in V_h$.

Under the assumption that the global degrees of freedom $\mathcal{N}$ are either point evaluations or directional derivatives at a node of the element, the basis functions of the global function space are derived from the basis functions of the finite element (cf. [Cia78] (2.3.28)). Let $\ell_i \in \mathcal{N}$ be a global degree of freedom with index i, let Λ_i be the set of elements $T \in \mathcal{T}$ for which the local-to-global mapping $\iota_T(j) = i$ for at least one index $j = 1, \dots, n_T$. For each $T_\lambda \in \Lambda_i$, let ϕ_λ denote the basis function of the finite element T_λ, associated with the restriction of ℓ_i to T_λ. Then the function $w_i \in V_h$, defined by

$$w_i = \begin{cases} \phi_\lambda & \text{on } T_\lambda \in \Lambda_i, \\ 0 & \text{else}, \end{cases} \tag{2.73}$$

is the basis function of the space V_h associated with the degree of freedom ℓ_i. Since equivalently global degrees of freedom are just associated with local degrees of freedom, we have also the desirable property

$$\ell_i(w_j) = \delta_{ij}, \quad \forall i, j = 1, \dots, N. \tag{2.74}$$

Lemma 2.25 applies analogously.

Affine equivalence

Definition 2.26. Let $(T, \mathcal{P}, \mathcal{N})$ and $(\hat{T}, \hat{\mathcal{P}}, \hat{\mathcal{N}})$ be finite elements and $F : T \to \hat{T}$ be an affine map (i.e. $F(x) = Ax + b$ with $A \in \mathbb{R}^{d\times d}$ nonsingular and $b \in \mathbb{R}^d$). Then both elements are said to be *affine equivalent* (under F), if

$$\text{(i)}\quad F(T) := \{F(x) : x \in T\} = \hat{T}, \tag{2.75}$$
$$\text{(ii)}\quad F^*(\hat{\mathcal{P}}) := \{\phi \circ F : \phi \in \hat{\mathcal{P}}\} = \mathcal{P} \text{ and} \tag{2.76}$$
$$\text{(iii)}\quad F_*(\mathcal{N}) := \{\ell(\cdot \circ F) : \ell \in \mathcal{N}\} = \hat{\mathcal{N}}. \tag{2.77}$$

When using affine equivalent elements (Lagrange elements for example), we can fix some *reference element* $(\hat{T}, \hat{\mathcal{P}}, \hat{\mathcal{N}})$ and specify all other elements $\{(T, \mathcal{P}_T, \mathcal{N}_T)\}_{T\in\mathcal{T}_h}$ by defining the affine mappings $\{F_T\}_{T\in\mathcal{T}_h}$ from the reference element. This has the advantage that the shape functions and degrees of freedom must be specified only once for the reference element.

2.3.3 Finite element examples

The Lagrange elements

As an example for an affine equivalent finite element, we consider the family of Lagrange elements, as shown in fig. 2.1.

Definition 2.27 (Lagrange element)**.** Let $\Omega \subset \mathbb{R}^d$ $(d = 1, 2, 3)$. The finite element $(T, \mathcal{P}, \mathcal{N})$ with

- T is the convex hull of $(x_i)_{i=1}^{n(q,d)}$, i.e. an interval for $d = 1$, a triangle for $d = 2$ or a tetrahedron for $d = 3$,
- $\mathcal{P}$ is the $n(q, d)$-dimensional function space of polynomials of order q on T,
- $\mathcal{N}$ is the set of point evaluations at x_i, i.e. $\ell_i(v) = v(x_i)$, $i = 1, ..., n(q, d)$,

is called the *Lagrange element* (CG_q).

Here $(x_i)_{i=1}^{n(q,d)}$ denotes an enumeration of points on the element T

$$(x_i)_{i=1}^{n(q,d)} = \begin{cases} \{i/q\}, & 0 \leqslant i \leqslant q, & \text{for } d = 1, \\ \{(i/q, j/q)\}, & 0 \leqslant i + j \leqslant q, & \text{for } d = 2, \\ \{(i/q, j/q, k/q)\}, & 0 \leqslant i + j + k \leqslant q, & \text{for } d = 3, \end{cases} \tag{2.78}$$

and

$$n(q, d) = \dim(\mathcal{P}) = \begin{cases} q + 1, & \text{for } d = 1, \\ \frac{1}{2}(q + 1)(q + 2), & \text{for } d = 2, \\ \frac{1}{6}(q + 1)(q + 2)(q + 3), & \text{for } d = 3. \end{cases} \tag{2.79}$$

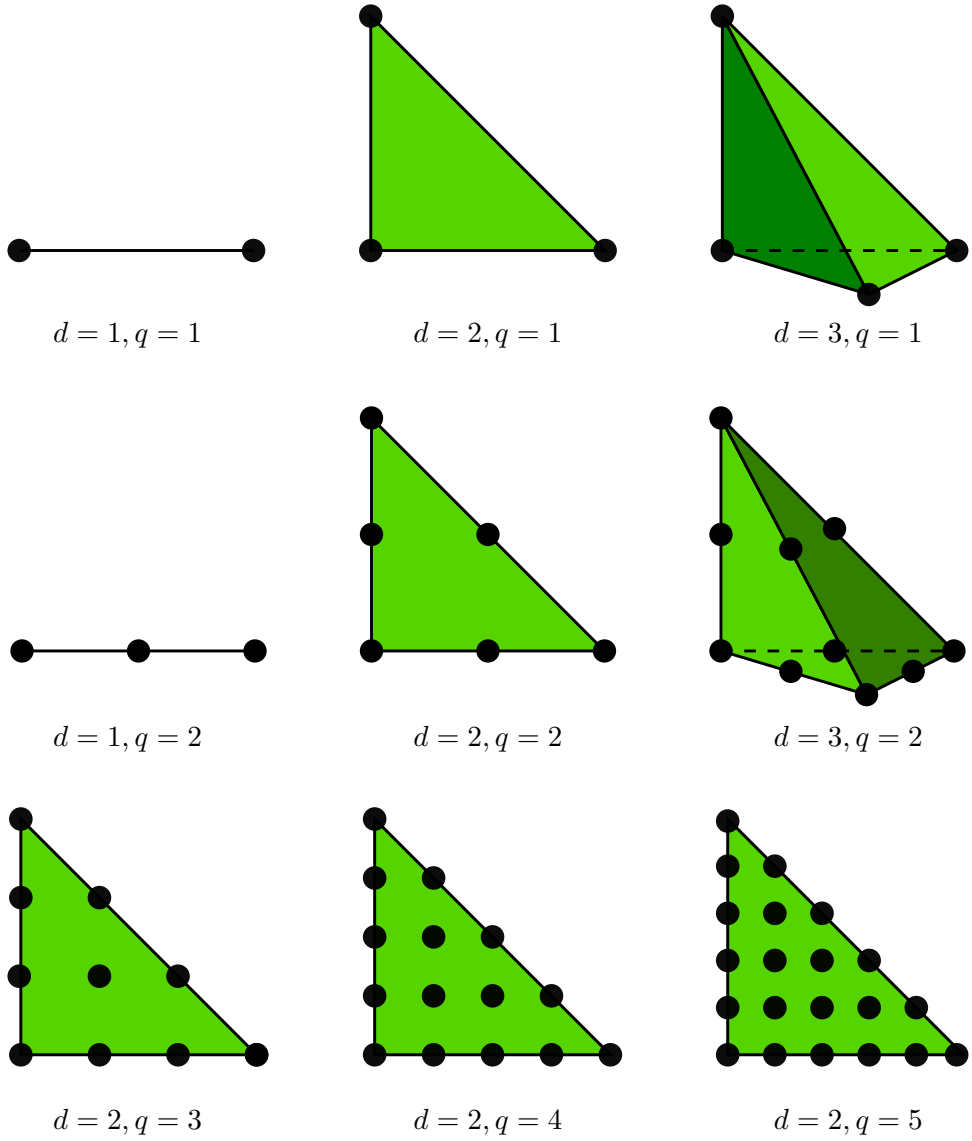


Figure 2.1: Element domain and degrees of freedom (location of point evaluation) for Lagrange elements in different dimensions d with a d-simplex element domain and of different polynomial order q (graphics by R.C. Kirby [LMW+12]).

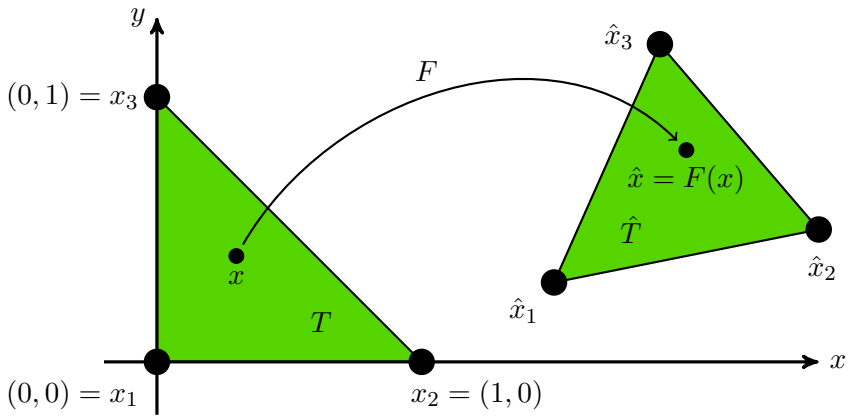


Figure 2.2: Affine transformation from reference element.

For $d = 1$ and $q = 1$ the nodal basis is given by

$$\phi_1(x) = 1 - x, \quad \phi_2(x) = x. \tag{2.80}$$

For $d = 2$ and $q = 1$ the nodal basis is given by

$$\phi_1(x) = 1 - x_1 - x_2, \quad \phi_2(x) = x_1, \quad \phi_3(x) = x_2. \tag{2.81}$$

For $d = 3$ and $q = 1$ the nodal basis is given by

$$\begin{array}{lllllll} \phi_1(x) & = & 1 - x_1 - x_2 - x_3, & \phi_2(x) & = & x_1, \\ \phi_3(x) & = & x_2, & \phi_4(x) & = & x_3. \end{array} \tag{2.82}$$

So far, this is the definition of the reference element $(T, \mathcal{P}, \mathcal{N})$. It can be transformed to an element $(\hat{T}, \hat{\mathcal{P}}, \hat{\mathcal{N}})$ of the same kind, but with a different element domain, by using affine transformations. As an example we consider the case $d = 2$ and $q = 1$,

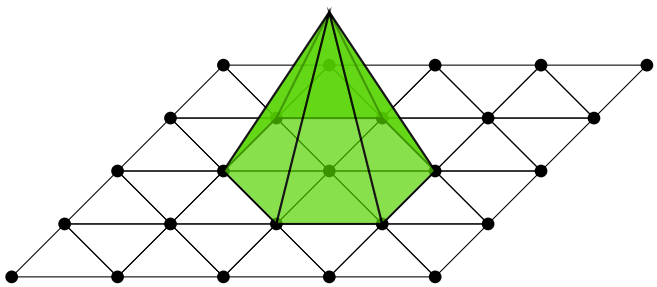


Figure 2.3: Example for a global basis function for 1st order Lagrange elements on a triangular mesh.

where the transformed element nodes are denoted by $\hat{x}_1, \hat{x}_2$ and $\hat{x}_3$ (see fig. 2.2). We make the Ansatz $F(x) = Ax + b$ with $A \in \mathbb{R}^{2\times 2}$ and $b \in \mathbb{R}^2$. The coefficients of A and b can be determined by the requirement $F(x_i) = \hat{x}_i$ for $i = 1, 2, 3$, which gives

$$F((0,0)^T) = b = \hat{x}_1, \tag{2.83}$$
$$F((1,0)^T) = (A_{11}, A_{21})^T + b = \hat{x}_2, \tag{2.84}$$
$$F((0,1)^T) = (A_{12}, A_{22})^T + b = \hat{x}_3. \tag{2.85}$$

So the affine transformation is described by

$$F(x) = \left(\hat{x}_2 - \hat{x}_1 \mid \hat{x}_3 - \hat{x}_1\right) x + \hat{x}_1, \tag{2.86}$$

where A is nonsingular, whenever the triangle $\hat{T}$ has a nonzero area.

An example for the resulting global basis functions, defined by eq. (2.73) using the transformed local basis functions $\hat{\phi} = \phi \circ F$, is given in fig. 2.3.

The Crouzeix-Raviart element

As an example of a $H^1(\Omega)$-nonconforming (i.e. $V_h \nsubseteq V$) finite element, we consider the Crouzeix-Raviart element, as shown in fig. 2.4.

Definition 2.28 (Crouzeix-Raviart element)**.** Let $\Omega \subset \mathbb{R}^d$ $(d = 2, 3)$. The finite element $(T, \mathcal{P}, \mathcal{N})$ with

- T is a triangle $(d = 2)$ or a tetrahedron $(d = 3)$,
- $\mathcal{P}$ is the $n(d)$-dimensional function space of linear polynomials on T,
- $\mathcal{N}$ is the set of point evaluations at x_i, i.e. $\ell_i(v) = v(x_i)$, $i = 1, ..., n(d)$,

is called the *Crouzeix-Raviart element* (CR).

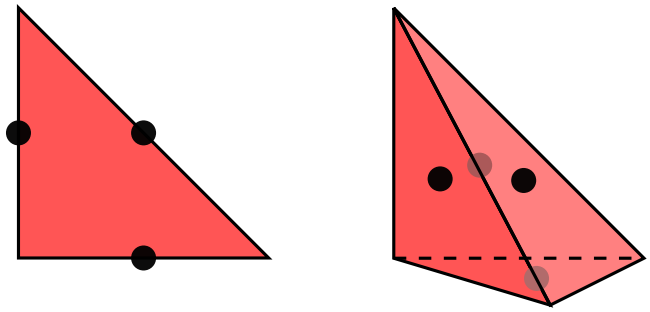
Here $(x_i)_{i=1}^{n(d)}$ are the barycenters of each facet of T, i.e.

$$(x_i)_{i=1}^{n(d)} = \begin{cases} \{\left(\frac{1}{2}, 0\right), \left(0, \frac{1}{2}\right), \left(\frac{1}{2}, \frac{1}{2}\right)\}, & \text{for } d = 2, \\ \{\left(0, \frac{1}{3}, \frac{1}{3}\right), \left(\frac{1}{3}, 0, \frac{1}{3}\right), \left(\frac{1}{3}, \frac{1}{3}, 0\right), \left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right)\}, & \text{for } d = 3, \end{cases} \tag{2.87}$$

and

$$n(d) = d + 1 \tag{2.88}$$

Like the Lagrange element, the CR element can also be transformed to an element of the same kind, but with a different element domain, using affine transformations.



$d = 2$ $d = 3$

Figure 2.4: Element domain and degrees of freedom for Crouzeix-Raviart elements in different dimensions d (graphics by R.C. Kirby [LMW+12]).

2.3.4 Treatment of Dirichlet boundary conditions

In the previous theory, the Dirichlet boundary conditions, were implicitly included in the spaces V and V_h. However, for performance reasons the Dirichlet boundary conditions are not imposed during the assembly of the linear system (cf. [Sch06] sec. 2.5.2.3). Instead they are given by some constraints $u_{m(i)} = g_i$, $i = 1, ..., N_D$, where N_D denotes the number of Dirichlet nodes, and $m : [1, N_D] \cap \mathbb{N} \mapsto [1, N] \cap \mathbb{N}$ is a map to the corresponding global degree of freedom index, and $g = (g_i)_{i=1}^{N_D}$ is the vector of Dirichlet boundary values. These constraints are then imposed on the linear system $Au = b$, after system assembly. This can be done, by reducing the system to a system

$$\tilde{A}\tilde{u} = \tilde{b} - \tilde{B}g, \tag{2.89}$$

where $\tilde{A} \in \mathbb{R}^{(N-N_D)\times(N-N_D)}$ is the matrix A with all columns and rows removed which are associated with a Dirichlet boundary condition, $\tilde{u} \in \mathbb{R}^{(N-N_D)}$ is the vector of unknowns with all rows removed which are associated with a Dirichlet boundary condition and $\tilde{B} \in \mathbb{R}^{(N-N_D)\times N_D}$ is the matrix of columns of A which are associated with a Dirichlet boundary condition.

Alternatively, we can keep the system size and do a partial Gaussian elimination instead

$$\hat{A}u = P(b - \hat{B}g), \tag{2.90}$$

where $\hat{A} \in \mathbb{R}^{N\times N}$, is the matrix A with all columns and rows set to δ_{ij}, which are associated with a Dirichlet boundary condition, $\hat{B} \in \mathbb{R}^{N\times N_D}$ is the matrix of columns of A which are associated with a Dirichlet boundary condition and $P(x)$ denotes the operation $x_{m(i)} = g_i$, $i = 1, ..., N_D$, i.e. setting all rows of x, which are associated with a Dirichlet boundary condition to the corresponding Dirichlet boundary value.

2.4 Solvers for large, sparse linear systems

Direct methods, like Gaussian elimination work well for dense matrices with full row rank, given the number of unknowns is not too large. However, at a certain number of unknowns, solve time, memory consumption and round-off errors become unacceptable. That's why for large, sparse systems, iterative methods are used. They have also the advantage, that the iteration can be stopped, whenever the accuracy of the solution is sufficient. In this section, we will give an short overview of different iterative methods for the solution of large, sparse linear systems, inspired by [Ree11]. In particular, we cover splitting methods and Krylov subspace methods. A separate section is devoted to the multigrid method and multigrid preconditioning in conjunction with the FEM. In all cases we are looking for the solution of a system of linear equations

$$Ax = b, \tag{2.91}$$

where $A \in \mathbb{R}^{N \times N}$ is a large, sparse, invertible and in general non-symmetric matrix, $b \in \mathbb{R}^N$ is the right hand side, and $x \in \mathbb{R}^N$ is the vector of unknowns. An iterative method starts with an initial guess x_0 for the solution, which is then improved in every step, until a certain accuracy of the solution is reached. One way to measure the accuracy of the solution, is the *solution error*, defined by

$$e_k := x - x_k. \tag{2.92}$$

However, since the exact solution x is not known apriori, eq. (2.92) is of little practical use. Therefore, one defines the *residual*

$$r_k := b - Ax_k. \tag{2.93}$$

The norm $||r_k||$, the *absolute residual*, is then used as convergence measure. However, the absolute residual is dependent on the scaling of eq. (2.91). To overcome this problem, one also uses the *relative residual* $||r_k||/||r_0||$.

Since $b = Ax$, we have the important relation

$$r_k = A(x - x_k) = Ae_k, \tag{2.94}$$

which is known as the *residual equation*.

Note: If A is ill-conditioned and/or non-normal (i.e. $A^T A \neq AA^T$), a small norm of the residual does not necessarily imply a small error. This needs to be taken into account, when using the residual as stopping criterion for iterative methods.

2.4.1 Splitting methods

Solving the residual equation (2.94) exactly, is as expensive as solving the original system (2.91). Therefore A is replaced by a non-singular *preconditioning* matrix M, which should be easy invertible and close to the matrix A, in the sense that $M^{-1}r_k$ is close to e_k. Inserting $M^{-1}r_k$ as e_k in eq. (2.92), suggests the following iteration step

$$x_{k+1} := x_k + M^{-1}r_k. \tag{2.95}$$

Inserting the residual eq. (2.93), gives

$$x_{k+1} = x_k + M^{-1}(b - Ax_k) = (I - M^{-1}A)x_k + M^{-1}b, \tag{2.96}$$

which can be rewritten as

$$Mx_{k+1} = Nx_k + b, \tag{2.97}$$

where $N = M - A$. $E := M^{-1}N$ is called the *iteration matrix*. Splitting the matrix A into $A = L + D + U$, where L is the strictly lower part, D is the diagonal and U is the strictly upper part of A, some basic *splitting methods* can be defined, depending on the choice of M:

$$M = I \qquad \text{(Richardson method)} \tag{2.98}$$
$$M = D \qquad \text{(Jacobi method)} \tag{2.99}$$
$$M = D + L \qquad \text{(Gauss-Seidel method)} \tag{2.100}$$

By introducing a *relaxation parameter* ω, the step may be accommodated to yield faster convergence ($\omega > 1$, *over-relaxation*) or to get convergence at all ($\omega < 1$, *under-relaxation*). The relaxed step can be defined using a modified preconditioning matrix

$$\tilde{M} := M + \frac{1-\omega}{\omega}\text{diag}(M), \tag{2.101}$$

which defines the relaxed versions of the splitting methods:

$$\tilde{M} = \omega^{-1}I \qquad \text{(relaxed Richardson method)} \tag{2.102}$$
$$\tilde{M} = \omega^{-1}D \qquad \text{(relaxed Jacobi method)} \tag{2.103}$$
$$\tilde{M} = \omega^{-1}D + L \qquad \text{(SOR method)} \tag{2.104}$$

Splitting methods usually converge linearly to x, i.e. there is a $0 < c < 1$ such that

$$\|x_{k+1} - x\| \leq c\|x_k - x\|, \quad k = 0, 1, ... \tag{2.105}$$

Or using eq. (2.92)

$$\|e_{k+1}\| \leq c\|e_k\|, \quad k = 0, 1, ... \tag{2.106}$$

Inserting eq. (2.92) into eq. (2.96) gives the relation

$$e_{k+1} = M^{-1}Ne_k = Ee_k. \tag{2.107}$$

Therefore, E is also called the *error operator*. Splitting methods are convergent, if we have for the spectral radius of the error operator

$$\rho(E) := \max_{1 \leq i \leq N} |\lambda_i(E)| < 1, \tag{2.108}$$

where $\lambda_i(E)$, denotes the i-th eigenvalue of E.

For most practical purposes, these methods are too slow to be used as iterative solver, but because of their simplicity, they are used as preconditioners. As we will see later, splitting methods are also used as smoother in multigrid methods.

Note: In the following subsections we, will slightly change the notation from lower indices x_k to upper indices x^k, in order to access individual vector components by x_j^k.

The Richardson method

For $M = I$, we get the most simple splitting method, the Richardson method:

$$x^{k+1} = (I - A)x^k + b, \tag{2.109}$$

which may also be written component-wise, as

$$x_i^{k+1} = x_i^k + b_i - \sum_{j=1}^{N} a_{ij}x_j^k, \quad i = 1, ..., N. \tag{2.110}$$

The relaxed version is given by

$$x^{k+1} = (I - \omega A)x^k + \omega b. \tag{2.111}$$

For $A > 0$, the optimal relaxation parameter is given by

$$\omega_{opt} = \frac{2}{\lambda_{max}(A) + \lambda_{min}(A)} \tag{2.112}$$

where $\lambda_{max}(A)$ and $\lambda_{min}(A)$ are the maximum and minimum eigenvalues of A respectively. The spectral radius for the iteration matrix $E = I - \omega_{opt}A$, is given by

$$\rho(E) = \frac{\kappa(A) - 1}{\kappa(A) + 1}, \tag{2.113}$$

where $\kappa(A)$ is the condition number of A.

The Jacobi method

For $M = D$, we get the Jacobi method:

$$x^{k+1} = -D^{-1}(L+U)x^k + D^{-1}b, \tag{2.114}$$

which may also be written component-wise, as

$$x_i^{k+1} = \frac{1}{a_{ii}}\left(b_i - \sum_{j\neq i} a_{ij}x_j^k\right), \quad i = 1, ..., N \tag{2.115}$$

The relaxed version is given by

$$x^{k+1} = -((\omega-1)I + \omega D^{-1}(L+U))x^k + \omega D^{-1}b. \tag{2.116}$$

Due to eq. (2.108), $\omega \in (0, 1)$ is required for convergence.

The Gauss-Seidel method

For $M = D + L$, we get the (forward) Gauss-Seidel method:

$$x^{k+1} = -(D+L)^{-1}Ux^k + (D+L)^{-1}b, \tag{2.117}$$

which may also be written component-wise, as

$$x_i^{k+1} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i+1}^{N} a_{ij}x_j^k\right), \quad k = 1, ..., N \tag{2.118}$$

Equivalently for $M = D + U$, we get the backward Gauss-Seidel method

$$x^{k+1} = -(D+U)^{-1}Lx^k + (D+U)^{-1}b. \tag{2.119}$$

Successive over-relaxation (SOR)

Relaxation of the Jacobi method ($M = \omega^{-1}D + L$) gives the (forward) SOR method:

$$x^{k+1} = -(D+\omega L)^{-1}(\omega U + (1-\omega)D)x^k + \omega(D+\omega L)^{-1}b, \tag{2.120}$$

which may also be written component-wise, as

$$x_i^{k+1} = (1-\omega)x^k + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i+1}^{N} a_{ij}x_j^k\right), \quad k = 1, ..., N. \tag{2.121}$$

Equivalently for $M = D + U$, we get the backward SOR method

$$x^{k+1} = -(D + \omega U)^{-1}(\omega L + (1 - \omega)D)x^k + \omega(D + \omega U)^{-1}b. \tag{2.122}$$

For the spectral radius of the iteration matrix, the following property holds

$$\rho(E) \geq |1 - \omega|, \tag{2.123}$$

together with $\rho(E) < 1$, this gives the requirement $\omega \in (0, 2)$.

Symmetric successive over-relaxation (SSOR)

Using one forward SOR, followed by one backward SOR, defines the symmetric SOR:

$$x^{k+1/2} = x^k + (\omega^{-1}D + L)^{-1}r^k \tag{2.124}$$

$$x^{k+1} = x^{k+1/2} + (\omega^{-1}D + U)^{-1}(b - Ax^{k+1/2}). \tag{2.125}$$

Inserting the first equation into the second equation gives

$$x^{k+1} = x^k + \omega(2 - \omega)(D + \omega U)^{-1}D(D + \omega L)^{-1}r^k. \tag{2.126}$$

Therefore, the preconditioning matrix M is given by:

$$M = (\omega(2 - \omega))^{-1}(D + \omega L)D^{-1}(D + \omega U) \tag{2.127}$$

Using $\omega = 1$ we get also the matrix for the symmetric Gauss-Seidel method:

$$M = (D + L)D^{-1}(D + U) \tag{2.128}$$

As for the SOR method, $\omega \in (0, 2)$ is necessary for convergence.

2.4.2 Krylov subspace methods

Using eq. (2.95), the first few iterates (beginning with $x_0 = 0$) of the Richardson method ($M = I$) can be written as

$$x_1 = r_0 \tag{2.129}$$

$$x_2 = (I - A)r_0 + r_0 = (2I - A)r_0 \tag{2.130}$$

$$x_3 = (I - A)^2 r_0 + (I - A)r_0 + r_0 = (3I - 3A + A^2)r_0 \tag{2.131}$$

$$x_{k+1} = (I - A)x_k + r_0 \tag{2.132}$$

We see that

$$x_{k+1} = \sum_{i=0}^{k}(I - A)^i r_0 = P_k(A)r_0, \quad k = 0, 1, ... \tag{2.133}$$

where P_k is a matrix polynomial of degree k with $P_0(A) = I$. Thus we have

$$x_{k+1} \in \text{span}\{r_0, Ar_0, \ldots, A^k r_0\} =: \mathcal{K}^{k+1}(A, r_0). \tag{2.134}$$

$\mathcal{K}^{k+1}(A, r_0)$ is called the *Krylov subspace* of order $k+1$, generated by A and r_0. In general Krylov subspace methods approximate the solution x_{k+1} by minimizing the residual over the subspace

$$x_0 + \mathcal{K}^{k+1}(A, r_0), \tag{2.135}$$

under certain constraints. Because these methods form a basis of $\mathbb{R}^N$, they converge at most N steps, if no round-off errors are encountered.

The conjugate gradient method

For symmetric and positive definite $A \in \mathbb{R}^{N\times N}$, the *conjugate gradient (CG) method* is most prominent Krylov subspace method for solving sparse systems of linear equations. We will give here only a demonstrative explanation of this method, a detailed explanation can be found in [HS52].

The idea of CG is to search for the solution x, by beginning with a step in the direction $p_0 = r_0$ (i.e. one iterate of the of the Richardson method) and all subsequent steps p_k ($k = 1, 2, ...$) are chosen, such that they are A-orthogonal (*conjugate*) to all previous search directions, i.e.

$$p_i^T A p_j = 0 \quad \forall i \neq j. \tag{2.136}$$

The subsequent search direction p_{k+1} is then computed, by orthogonalizing the residual r_{k+1} with respect to the previous search direction p_k:

$$p_{k+1} = r_{k+1} + \beta p_k, \tag{2.137}$$

where

$$\beta_k = \frac{r_{k+1}^T A p_k}{p_k^T A p_k}, \tag{2.138}$$

such that $p_{k+1}^T A p_k = 0$. It is possible to show that this is the same β_k as

$$\beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}, \tag{2.139}$$

and that this is sufficient to have A-orthogonality for all other search directions too. The step length α_k is chosen such that the function

$$f(x) = \frac{1}{2} x^T A x - x^T b \tag{2.140}$$

is minimized along the search direction p_k

$$\alpha_k = \arg\min_{\alpha \in \mathbb{R}} f(x_k + \alpha p_k), \tag{2.141}$$

which has the unique solution

$$\alpha_k = \frac{p_k^T r_k}{p_k^T A p_k}. \tag{2.142}$$

The next residual is updated using the relation

$$r_{k+1} = b - Ax_{k+1} = b - A(x_k + \alpha p_k) = r_k - \alpha A p_k. \tag{2.143}$$

Together this gives alg. 2.1. The convergence of this method is roughly described by

$$\|e_k\|_A \leq 2 \left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^k \|e_0\|_A, \tag{2.144}$$

where $\kappa(A)$ denotes the condition number of A. Although CG converges in at most N iterations, this is usually still too much for large systems. However, it is known that if A has k distinct eigenvalues, then CG finishes in k iterations (assuming no round-off errors). Also if the eigenvalues of A are closely grouped together in k groups, then CG converges very quickly (finishes almost in k iterations).

Algorithm 2.1: The conjugate gradient (CG) method. CG(A, x_0, b)

$r_0 := b - Ax_0$
$p_0 := r_0$
$k := 0$
while not converged **do**
    $\alpha := \frac{p_k^T r_k}{p_k^T A p_k}$
    $x_{k+1} := x_k + \alpha p_k$
    $r_{k+1} := r_k - \alpha A p_k$
    $\beta := \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
    $p_{k+1} := r_{k+1} + \beta p_k$
    $k := k + 1$
end
return x_k

The preconditioned conjugate gradient method

For the reasons of the last section, CG is usually used in conjunction with a symmetric preconditioner $M \in \mathbb{R}^{N \times N}$, which concentrates the eigenvalues of A. That is, instead of solving $Ax = b$, we solve now the system

$$M^{-1}Ax = M^{-1}b, \tag{2.145}$$

which is known as *left preconditioning*. However, since $M^{-1}A$ is usually not symmetric anymore the CG method alg. 2.1 is not applicable. Instead *symmetric preconditioning* must be used:

$$H^{-T}AH^{-1}(Hx) = H^{-T}b, \tag{2.146}$$

where H comes from a Cholesky-decomposition of M such that $M = H^T H$. However, the expensive computation of H can be avoided, as shown in the following. Inserting $H^{-T}AH^{-1}$ as A, Hx as x and $H^{-T}b$ as b into alg. 2.1, inserting some identity matrices and doing some line multiplications, the PCG algorithm can be written as:

Algorithm 2.2: Symmetric preconditioned conjugate gradient method.
PCG(H, A, x_0, b)

$H^T r_0 := (b - Ax_0)$
$H^{-1}p_0 := H^{-1}H^{-T}H^T r_0$
$k := 0$
while not converged **do**
  $\alpha := \frac{p_k^T H^{-T} H^T r_k}{p_k^T H^{-T} A H^{-1} p_k}$
  $x_{k+1} := x_k + \alpha H^{-1} p_k$
  $H^T r_{k+1} := H^T r_k - \alpha A H^{-1} p_k$
  $\beta := \frac{r_{k+1}^T H H^{-1} H^{-T} H^T r_{k+1}}{r_k^T H H^{-1} H^{-T} H^T r_k}$
  $H^{-1}p_{k+1} := H^{-1}H^{-T}H^T r_{k+1} + \beta H^{-1} p_k$
  $k := k + 1$
end
return x_k

Substituting all $H^T r_k$ to $\tilde{r}_k$, $H^{-1}p_k$ to $\tilde{p}_k$ and $H^{-T}H^T$ to M^{-1}, we get the following algorithm, which is now only dependent on M:

Algorithm 2.3: The preconditioned conjugate gradient (PCG) method.
PCG(M, A, x_0, b)

$\tilde{r}_0 := (b - Ax_0)$
$\tilde{p}_0 := M^{-1}\tilde{r}_0$
$k := 0$
while not converged **do**
  $\alpha := \frac{\tilde{p}_k^T \tilde{r}_k}{\tilde{p}_k^T A \tilde{p}_k}$
  $x_{k+1} := x_k + \alpha \tilde{p}_k$
  $\tilde{r}_{k+1} := \tilde{r}_k - \alpha A \tilde{p}_k$
  $\beta := \frac{\tilde{r}_{k+1}^T M^{-1} \tilde{r}_{k+1}}{\tilde{r}_k^T M^{-1} \tilde{r}_k}$
  $\tilde{p}_{k+1} := M^{-1}\tilde{r}_{k+1} + \beta \tilde{p}_k$
  $k := k + 1$
end
return x_k

Alg. 2.3 is known as the *preconditioned conjugate gradient* (PCG) method, which may be further optimized by avoiding the multiple occurrence of $A\tilde{p}_k$ and $M^{-1}\tilde{r}_{k+1}$. A more beautiful and general derivation of the PCG method, which completely avoids the temporary use of Cholesky factors is given in [Gue+11].

Since

$$\|Hx\|_{H^{-T}AH^{-1}} = x^T H^T H^{-T} A H^{-1} H x = \|x\|_A, \tag{2.147}$$

we get for the convergence estimate eq. (2.144)

$$\|e_k\|_A \leq 2 \left(\frac{\sqrt{\kappa(M^{-1}A)} - 1}{\sqrt{\kappa(M^{-1}A)} + 1} \right)^k \|e_0\|_A. \tag{2.148}$$

Simple preconditioners for PCG, are the relaxed Jacobi (diagonal) preconditioner eq. (2.103) and the SSOR preconditioner eq. (2.127).

The minimum residual method

The *minimum residual method* (MINRES) [PS75] is a Krylov subspace method for symmetric systems, which not necessarily have to be positive definite in contrast to CG. The iterates x_k are generated by minimizing the residual over the sequence of growing Krylov spaces

$$x_{k+1} := \operatorname{argmin}_{x \in \mathcal{K}^{k+1}(A, r_0)} \|b - Ax\|_{\mathbb{R}^N}. \tag{2.149}$$

In contrast to CG, the convergence is naturally estimated in terms of the Euclidean norm of the residual

$$\|r_k\|_{\mathbb{R}^N} \leq 2 \left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^k \|r_0\|_{\mathbb{R}^N}. \tag{2.150}$$

Equivalently to CG, there is also a preconditioned version of MINRES (cf. [Gue+11]).

2.5 Geometric Multigrid

2.5.1 Introduction

Multigrid solvers are considered to be among the fastest solvers for problems coming from a finite element discretization, with a large number of unknowns. In the case of elliptic partial differential equations they have an optimal or near-optimal efficiency of $\mathcal{O}(N)$, i.e. the computational work is only proportional to the number of unknowns. Multigrid methods are based on the fact that for the low and high frequency components of the solution, different solution approaches are reasonable. In this introduction we will motivate this with a simple example. As previously, we would like to solve a linear system of equations

$$Ax = b, \tag{2.151}$$

where $A \in \mathbb{R}^{N \times N}$ is a large, sparse and in general non-symmetric matrix, $b \in \mathbb{R}^N$ is the right hand side, and $x \in \mathbb{R}^N$ is the vector of unknowns.

In a first step, we will explain the smoothing effect of classical iterative methods (like splitting methods), known as the *smoothing principle* or *smoothing property*. Therefore, we consider one step of such an iterative method, which may be written as (cf. eq. (2.97))

$$Mx_{k+1} = Nx_k + b. \tag{2.152}$$

We know that for the error $e_k := x - x_k$, we have the property (cf. eq. (2.107))

$$e_{k+1} = M^{-1}Ne_k = Ee_k, \tag{2.153}$$

where $E := M^{-1}N$. For some basic methods, we have for example:

$$E = I - \omega A \qquad \text{(relaxed Richardson method)} \tag{2.154}$$

$$E = I - \omega D^{-1} A \qquad \text{(relaxed Jacobi method)} \tag{2.155}$$

$$E = I - (D + \omega L)^{-1}(\omega A + 2(1 - \omega)D) \qquad \text{(SOR method)} \tag{2.156}$$

using a splitting of the matrix $A = L + D + U$, where L is the strictly lower part, D is the diagonal and U is the strictly upper part of A. Assuming A has a full set of orthonormal eigenvectors q_i and eigenvalues λ_i with

$$Aq_i = \lambda_i q_i, \quad i = 1, ..., n \tag{2.157}$$

we may express the error e_k as

$$e_k = \sum_{i=1}^{n} \alpha_i q_i, \tag{2.158}$$

for suitable coefficients α_i. Assuming $E = I - \omega D^{-1} A$ (relaxed Jacobi method), it follows for the error after one iteration

$$e_{k+1} = Ee_k = \sum_{i=1}^{n} \alpha_i E q_i = \sum_{i=1}^{n} \alpha_i (I - \omega D^{-1} \lambda_i) q_i = e^k - \omega D^{-1} \sum_{i=1}^{n} \alpha_i \lambda_i q_i. \tag{2.159}$$

Since the eigenvectors are orthonormal, we have $\alpha_i = e_k^T q_i$ and may write

$$e_{k+1} = e_k - \omega D^{-1} \sum_{i=1}^{n} \lambda_i (e_k^T q_i) q_i. \tag{2.160}$$

This can be interpreted in the following way: The inner product $e_k^T q_i$ measures the signal content of e_k with respect to q_i, which is very similar to a Fourier coefficient of e_k. Overall the sum can be interpreted as spectral scaling/multiplication of e_k with the spectrum of A. If we deal with elliptic partial differential equations, the interpretation is begged by the fact, that the operator A is the discretization of a differential operator. The eigenvectors of such operators are usually the harmonics of some kind of oscillation. For example the matrix

$$\Delta = \frac{1}{h^2} \begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & -1 \\ & & & -1 & 2 \end{pmatrix} \in \mathbb{R}^{N \times N} \tag{2.161}$$

is the finite difference discretization for the homogeneous Poisson problem on the unit interval, using $N + 2$ equally spaced points (the zero boundary points are removed from the matrix). The orthonormal eigenvectors of Δ, shown in fig. 2.5, are given by [Hac85; Som03]

$$q_i = \sqrt{h} \begin{pmatrix} \sin(1h\, i\pi) \\ \sin(2h\, i\pi) \\ \vdots \\ \sin(Nh\, i\pi) \end{pmatrix}, \tag{2.162}$$

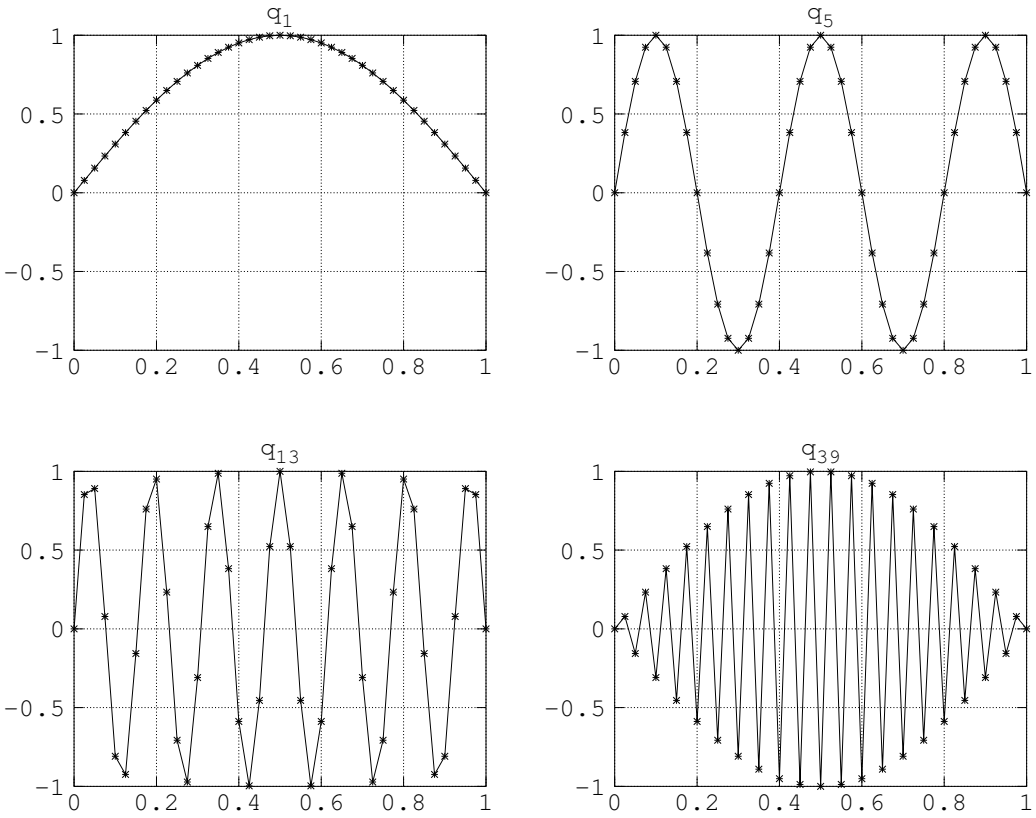


Figure 2.5: Eigenvectors of Δ for $N = 39$, including the zero boundary values.

where $h = \frac{1}{N+1}$. The corresponding eigenvalues, shown in fig. 2.6, are

$$\lambda_i = \frac{4}{h^2} \sin^2 \left(\frac{i\pi h}{2} \right). \tag{2.163}$$

In this special case the term $e_k^T q_i$ is the i-th frequency coefficient of the discrete sine transformation of e_k. For a 2d discretization of the same problem (see [Bra03]), the resulting eigenvalues and eigenvectors have a similar structure.
Rewriting eq. (2.159) as

$$e_{k+1} = \sum_{i=1}^{n} (I - \omega D^{-1} \lambda_i) \alpha_i q_i = \sum_{i=1}^{n} s_i \alpha_i q_i, \tag{2.164}$$

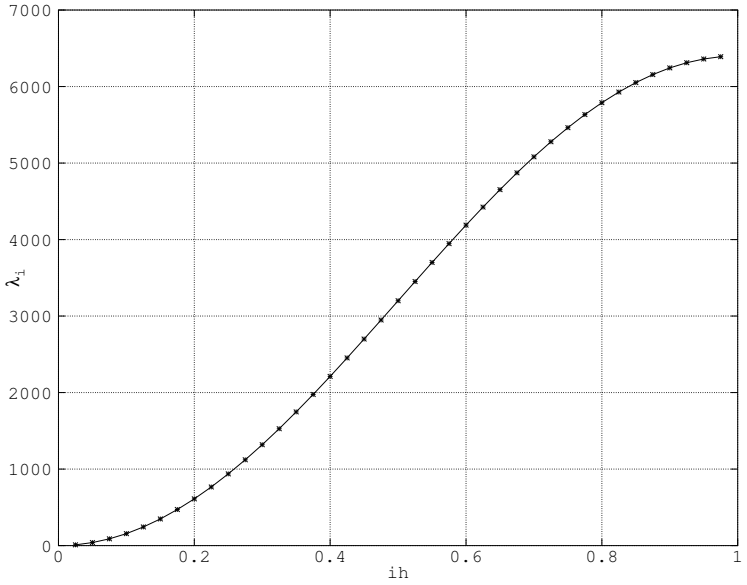


Figure 2.6: Eigenvalues of Δ for $N = 39$.

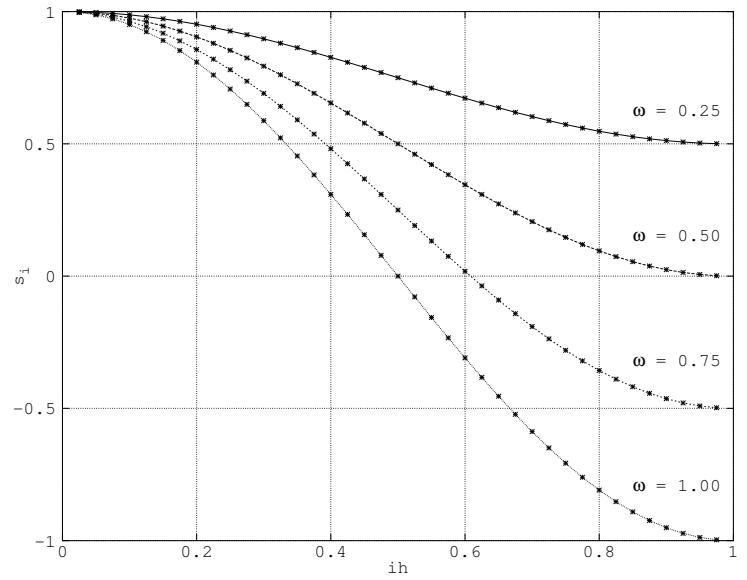


Figure 2.7: Error scaling factors s_i for $N = 39$ and different relaxation parameters ω.

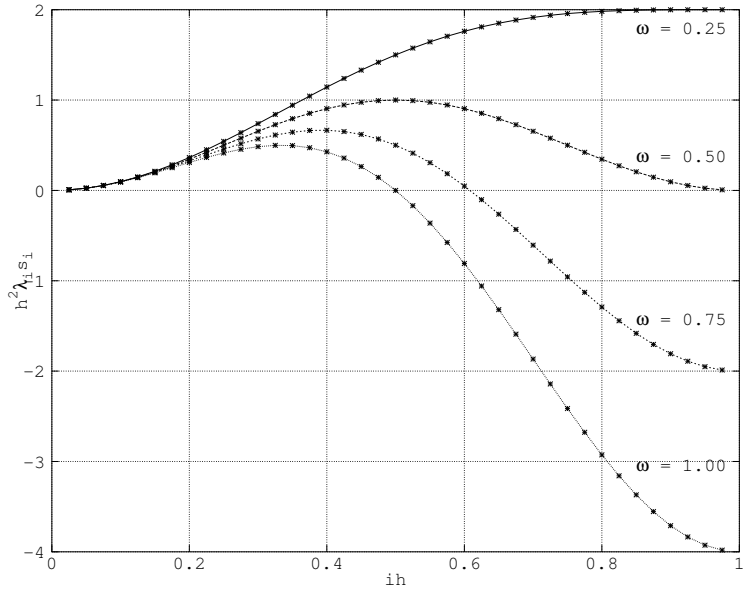


Figure 2.8: Residual scaling factors $h^2\lambda_i s_i$ for $N = 39$ and different relaxation parameters ω.

with $s_i = I - \omega D^{-1}\lambda_i$ and since $D^{-1} = \frac{h^2}{2}I$ for our example matrix $A = \Delta$, we get

$$s_i = 1 - 2\omega \sin^2\left(\frac{i\pi h}{2}\right). \tag{2.165}$$

We see that the Jacobi method just scales the expansion coefficients α_i with s_i. This scaling depends on ω, as shown in fig. 2.7. For $\omega = 0.5$ high frequency errors are damped/smoothed, while low frequency errors remain. However, since e_k is not known, we can't compute e_{k+1}. But the resulting x_{k+1} is known and can be used to compute the residual. This residual is also smoothed, because

$$r_{k+1} = Ae_{k+1} = \sum_{i=1}^{n} s_i\alpha_i Aq_i = \sum_{i=1}^{n} \lambda_i s_i \alpha_i q_i. \tag{2.166}$$

This effectively smooths also low frequency parts of the residual, as seen in fig. 2.8. Due to the fact that high frequency residuals are damped, the residual can be well approximated on a coarser grid (e.g. of spacing $h/2$). Therefore we introduce the *restriction operator* R, which maps the residual from the fine grid to a coarse grid, e.g.

$$R : \mathbb{R}^N \mapsto \mathbb{R}^{N/2}. \tag{2.167}$$

For our example we could choose [Hac85]

$$R = \frac{1}{4}\begin{pmatrix} 2 & 1 & & & & & & & \\ & 1 & 2 & 1 & & & & & \\ & & & 1 & 2 & 1 & & & \\ & & & & & \ddots & & & \\ & & & & & 1 & 2 & 1 & \\ & & & & & & & 1 & 2 \end{pmatrix} \in \mathbb{R}^{N/2 \times N}. \tag{2.168}$$

On the coarser grid it is much cheaper to solve (e.g. using a direct method) the residual equation $A_c e_c = R r_{k+1}$ for the error on the coarse grid e_c, given the coarse problem matrix A_c (e.g. $\Delta \in \mathbb{R}^{N/2 \times N/2}$ from eq. (2.161)). Once we have e_c on the coarse grid, we can go back to the fine grid using the *prolongation operator*

$$P : \mathbb{R}^{N/2} \mapsto \mathbb{R}^{N}. \tag{2.169}$$

For our example one could choose [Hac85]

$$P = \frac{1}{2}\begin{pmatrix} 2 & & & & & & \\ 1 & 1 & & & & & \\ & 2 & & & & & \\ & 1 & 1 & & & & \\ & & 2 & & & & \\ & & 1 & \ddots & 1 & & \\ & & & & 2 & & \\ & & & & 1 & 1 & \\ & & & & & 2 & \end{pmatrix} = 2R^T \in \mathbb{R}^{N \times N/2}. \tag{2.170}$$

This gives an approximation $\tilde{e}_{k+1}$ of the error e_{k+1}, we were looking for:

$$\tilde{e}_{k+1} = P e_c = P A_c^{-1} R r_{k+1}. \tag{2.171}$$

Using this approximation, we can compute an approximate solution $\tilde{x}$ of x

$$\tilde{x} = x_{k+1} + \tilde{e}_{k+1}. \tag{2.172}$$

The approximate solution is then used as starting value for the next iterate. This iteration scheme is know as the *two-grid method*. If we do not solve the linear equation on the coarse grid using a direct method, but using again a two grid approach, the method can be extended recursively to a certain depth level ℓ. This is then considered as a *multigrid method*. However, this was only a demonstration to get an idea how multigrid works. In practice, algorithms, as discussed in the next sections are used.

2.5.2 The two-grid method

Following the introduction, we get an algorithm for one iteration similar to alg. 2.4. Here $\mathcal{S}_1^{\nu_1}(x,b)$ and $\mathcal{S}_1^{\nu_2}(x,b)$ denote the so called *pre-smoother* and *post-smoother* on level 1, applied ν_1 and ν_2 times to the vector x, for the right hand side b. Basic smoothers among others are the Jacobi, Gauss-Seidel, SOR and SSOR methods, as discussed in section 2.4.1. The second line restricts the *defect* (residual) to the coarser level, where r_1 denotes a restriction operator from level 1 to level 0, and A_1 denotes the system matrix on level 1. In line three the residual equation is solved. In line four the computed error is prolongated back to level 1, where the correction step is performed. After the correction step the post-smoother is applied. When used as solver, the *two-grid method* (TGM) is repeated until a convergence criterion is met. As a preconditioner, the TGM is usually applied only a few times starting with $x = 0$.

Algorithm 2.4: The two-grid method $TGM(x,b)$.

1 $x := \mathcal{S}_1^{\nu_1}(x,b)$ `// pre-smoothing`
2 $d := r_1(b - A_1 x)$ `// defect computation + restriction to coarser grid`
3 $e := A_0^{-1} d$ `// solve coarse problem`
4 $x := x + p_0 e$ `// prolognation + correction step`
5 $x := \mathcal{S}_1^{\nu_2}(x,b)$ `// post-smoothing`
6 **return** x

Alg. 2.4 may also be written in one equation as

$$x_{k+1} = \mathcal{S}_1^{\nu_2}(\mathcal{S}_1^{\nu_1}(x_k,b) + p_0 A_0^{-1} r_1(b - A_1 \mathcal{S}_1^{\nu_1}(x_k,b)),b) \tag{2.173}$$

2.5.3 The multi-grid method

Instead of solving the coarse problem $A_0 e = d$ in alg. 2.4 (line 3) directly, we approximate the solution by adding one more coarse level and doing $\gamma_{\ell-1}$ iterations of the two-grid method. Performing this approach recursively for ℓ levels, gives the *multi-grid (multigrid) method* (MGM), shown in alg. 2.5. Now the operators A_ℓ, r_ℓ, and p_ℓ are given for all levels (except r_0 and p_ℓ). Similarly $\mathcal{S}_\ell^{\nu_2}$ and $\mathcal{S}_\ell^{\nu_1}$ are given for the levels $1,...,\ell$. Depending on γ_ℓ, different types of cycles are possible, as seen in fig. 2.9. For example we get for $\gamma_\ell = 1$ the so called *V-cycle* and for $\gamma_\ell = 1 + \delta_{\ell,3}$ the shown *W-cycle*. However, the first V-cycles of an MGM iteration are very expensive, since they do not contribute very much to the solution. Therefore, the iteration is usually started with an F-cycle on the coarsest level as seen in fig. 2.10 (the first five restrictions from the fine level to the coarse level are just for the initialization and can be omitted, if the right hand side of the coarse problem is known), which performs some cheaper V-cycles over an increasing number of levels. This procedure is called *nested iteration* or *full multigrid*.

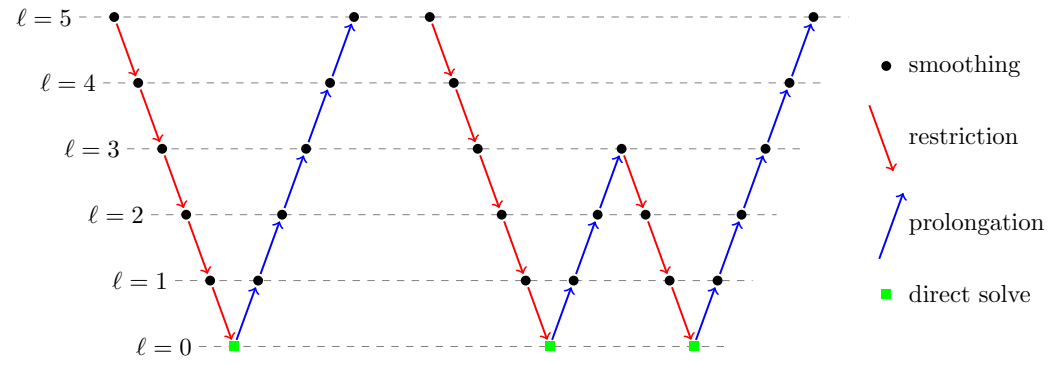


Figure 2.9: V- and W-cycle for 6 levels.

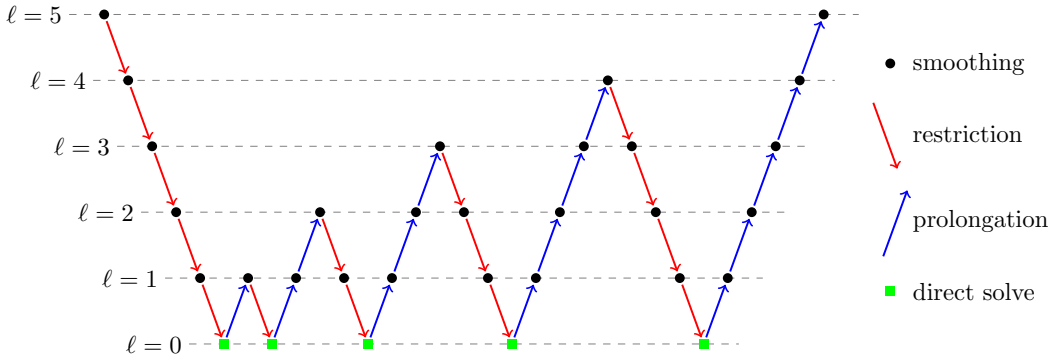


Figure 2.10: Nested iteration (F-cycle) for 6 levels.

Algorithm 2.5: The multi-grid method $MGM(\ell, x, b)$.

if $\ell = 0$ **then**
  return $A_0^{-1} b$ `// direct solve on coarse grid`
end
$x := \mathcal{S}_\ell^{\nu_1}(x, b)$ `// pre-smoothing`
$d := r_\ell(b - A_\ell x)$ `// defect computation + restriction to coarser grid`
$e := 0$ `// start value for coarse grid iteration`
for $i := 1$ **to** $\gamma_{\ell-1}$ **do** `// coarse grid iteration`
  $e := \mathrm{MGM}(\ell - 1, e, d)$ `// solve coarse problem`
end
$x := x + p_{\ell-1} e$ `// prolongation + correction step`
$x := \mathcal{S}_\ell^{\nu_2}(x, b)$ `// post-smoothing`
return x

2.5.4 Finite element multigrid

We introduced the multigrid method for general symmetric $N \times N$ matrices. For the construction of prolongation and restriction operators and convergence analysis, it is necessary to introduce some more formalism and confine ourselves to the FEM discretization of an elliptic variational problem: find the solution $u \in V$ such that

$$a(u, v) = f(v), \quad \forall v \in V, \tag{2.174}$$

where V denotes an discrete finite element function space, associated with a triangulation $\mathcal{T}$. Further we assume $\mathcal{T}$ was obtained by a sequence of regular subdivisions of an initial triangulation $\mathcal{T}_0$, i.e.

$$\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_L := \mathcal{T}, \tag{2.175}$$

and we assume that the associated finite element function spaces are also nested, i.e.

$$V_0 \subset V_1 \subset \cdots \subset V_L := V, \tag{2.176}$$

and the matrices A_ℓ denote the discretization of $a(\cdot, \cdot)$ for the space $V_\ell \subset V$ (cf. section 2.3). In Alg. 2.5 it remains to clarify how to construct the restriction and prolongation operators r_ℓ and p_ℓ, which is discussed in the next section.

2.5.5 Canonical prolongation and restriction operators

Under the setting of section 2.5.4, we choose two nested spaces $V_{\ell-1} \subset V_\ell$ of dimension N and $\hat{N}$, which we denote by $V \subset \hat{V}$ and let $\mathcal{T} \subset \hat{\mathcal{T}}$ be the associated triangulations. Let $U = \mathbb{R}^N$ and $\hat{U} = \mathbb{R}^{\hat{N}}$. The goal of this section is the explicit construction of the prolongation operator $p : U \mapsto \hat{U}$, which maps the coefficients of an element in V to the coefficients of an element in $\hat{V}$, and the construction of the restriction operator $r : \hat{U} \mapsto U$, which maps the coefficients of an element in $\hat{V}'$ to the coefficients of an element in V'. We will start with the construction of the prolongation operator.

For every $v \in V$ there exists a bijection $P : U \mapsto V$, which maps the associated vector of coefficients $u \in U$ to V. Equally for every $\hat{v} \in \hat{V}$ there exists a bijection $\hat{P} : \hat{U} \mapsto \hat{V}$, which maps the associated vector of coefficients $\hat{u} \in \hat{U}$ to $\hat{V}$ (cf. lemma 2.25). Since $V \subset \hat{V}$, a canonical choice for p is given in terms of the canonical injection $i : V \hookrightarrow \hat{V}$ by $p = \hat{P}^{-1} i P$ (cf. left side of fig. 2.11).

Now, every $v \in V$ and every $\hat{v} \in \hat{V}$ may be represented as (cf. section 2.3.2)

$$v = \sum_{i=1}^{N} \ell_i(v)\phi_i \quad \text{and} \quad \hat{v} = \sum_{i=1}^{\hat{N}} \hat{\ell}_i(\hat{v})\hat{\phi}_i, \tag{2.177}$$

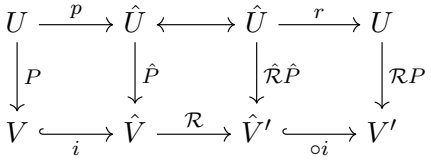


Figure 2.11: Commutative diagram for the prolongation and restriction operators p and r.

where $\{\phi_i\}_{i=1}^{N}$ and $\{\hat{\phi}_i\}_{i=1}^{\hat{N}}$ are the global basis of V and $\hat{V}$ and $\mathcal{N} = \{\ell_1, \dots, \ell_N\}$ and $\hat{\mathcal{N}} = \{\hat{\ell}_1, \dots, \hat{\ell}_{\hat{N}}\}$ are the associated global degrees of freedom, respectively. Further we have also for the global basis (cf. section 2.3.2)

$$\hat{\ell}_i(\hat{\phi}_j) = \delta_{ij}, \quad \forall i, j = 1, \dots, \hat{N}. \tag{2.178}$$

By definition 2.29, we have for every $u \in U_T$

$$\hat{P}pu = iPu. \tag{2.179}$$

The operators $\hat{P}$ and P are defined by eq. (2.177), which gives

$$\sum_{i=1}^{\hat{N}} (pu)_i \hat{\phi}_i = \sum_{i=1}^{N} u_i \phi_i. \tag{2.180}$$

Applying $\hat{\ell}_j$ for $j = 1, \dots, N$ to both sides gives

$$\sum_{i=1}^{\hat{N}} (pu)_i \underbrace{\hat{\ell}_j(\hat{\phi}_i)}_{\delta_{ij}} = \sum_{i=1}^{N} u_i \hat{\ell}_j(\phi_i). \tag{2.181}$$

And finally

$$(pu)_j = \hat{u}_j = \sum_{i=1}^{N} u_i \hat{\ell}_j(\phi_i), \tag{2.182}$$

which can be written as matrix

$$p_{ij} = \hat{\ell}_i(\phi_j), \quad i = 1, \dots, \hat{N},\ j = 1, \dots, N. \tag{2.183}$$

For the construction of the restriction operator r let $\hat{f} \in \hat{V}'$, $b \in U$ and $\hat{b} \in \hat{U}$, with $\hat{b}_i = \hat{f}(\hat{\phi}_i)$, where $\{\hat{\phi}_i\}_{i=1}^{\hat{N}}$ denotes the global basis of $\hat{V}$. We define the restriction by

$$(r\hat{b})_j = b_j := \hat{f}(i\phi_j), \quad j = 1, \dots, N, \tag{2.184}$$

with the canonical injection $i : V \hookrightarrow \hat{V}$. Further we have

$$\hat{f}(i\phi_j) = \hat{f}\left(\sum_{i=1}^{\hat{N}} \hat{\ell}_i(\phi_j)\hat{\phi}_i\right) = \sum_{i=1}^{\hat{N}} \hat{\ell}_i(\phi_j)\hat{f}(\hat{\phi}_i) = \sum_{i=1}^{\hat{N}} \hat{\ell}_i(\phi_j)\hat{b}_i, \tag{2.185}$$

and

$$(r\hat{b})_j = \sum_{i=1}^{\hat{N}} r_{ji}\hat{b}_i, \tag{2.186}$$

which gives the identity

$$r_{ji} = \hat{\ell}_i(\phi_j) = p_{ij}. \tag{2.187}$$

Thus, the following definition makes sense:

Definition 2.29 (Prolongation and restriction operators [Hac85])**.** The canonical choice for the *prolongation operator* $p : U \mapsto \hat{U}$ is $p = \hat{P}^{-1} i P$. The canonical choice for the *restriction operator* $r : \hat{U} \mapsto U$ is $r = p^T$.

Let $\mathcal{R} : \hat{V} \mapsto V'$ and $\hat{\mathcal{R}} : \hat{V} \mapsto \hat{V}'$ denote the Riesz mappings, then an equivalent definition of the restriction operator is $r\hat{u} = (\mathcal{R}P)^{-1}((\hat{\mathcal{R}}\hat{P}\hat{u}) \circ i)$ for $\hat{u} \in \hat{U}$, which is shown in the right part of fig. 2.11.

Further it is also possible to construct the prolongation operator element wise, which is often more convenient to use. Therefore, we restrict ourselves to an element $\hat{T} \in \hat{\mathcal{T}}$ and $T \in \mathcal{T}$, such that $\hat{T} \subset T$ and we consider the local function spaces $\mathcal{P} = V|_{T \cap \hat{T}} \subset V|_T$ and $\hat{\mathcal{P}} = \hat{V}|_{\hat{T}}$ of dimension n and $\hat{n}$. Equivalently, we denote by $U_T = \mathbb{R}^n$ and $\hat{U}_T = \mathbb{R}^{\hat{n}}$ the local coefficient spaces and by $P_T : U_T \mapsto \mathcal{P}$ and $\hat{P}_T : \hat{U}_T \mapsto \hat{\mathcal{P}}$ the mappings from the coefficient to the local function spaces. Further we have the injection $i_T : \mathcal{P} \hookrightarrow \hat{\mathcal{P}}$ so that we can define the *local prolongation operator* as (cf. fig. 2.12)

$$p_T := \hat{P}_T^{-1} i_T P_T. \tag{2.188}$$

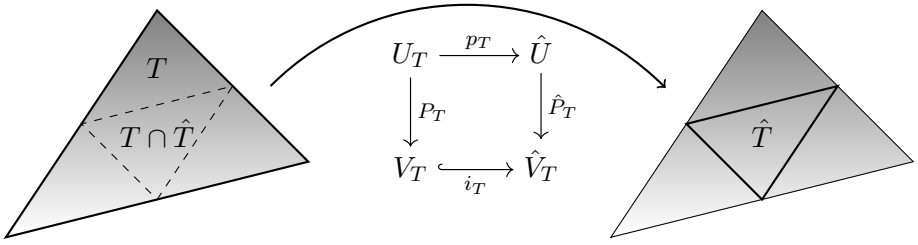


Figure 2.12: Commutative diagram for the local prolongation operator p_T.

Following the same procedure as for the global prolongation operator, we get

$$(p_T)_{ij} = \hat{\ell}_i(\phi_j), \quad i = 1, \dots, \hat{n},\ j = 1, \dots, n, \tag{2.189}$$

where $\{\phi_i\}_{i=1}^n$ is the nodal basis of $\mathcal{P}$ and $\{\hat{\ell}_1, \dots, \hat{\ell}_{\hat{n}}\}$ are the degrees of freedom of $\hat{T}$. Using the local-to-global mappings (cf. section 2.3.2)

$$\iota_T : [1, n] \cap \mathbb{N} \mapsto [1, N] \cap \mathbb{N} \text{ and} \tag{2.190}$$

$$\iota_{\hat{T}} : [1, \hat{n}] \cap \mathbb{N} \mapsto [1, \hat{N}] \cap \mathbb{N}, \tag{2.191}$$

we may assemble our global prolongation operator as

$$(p)_{\iota_{\hat{T}}(i), \iota_{T(\hat{T})}(j)} = (p_{T(\hat{T})})_{ij}, \quad \forall\, \hat{T} \in \hat{\mathcal{T}},\ i = 1, \dots, \hat{n},\ j = 1, \dots, n, \tag{2.192}$$

where we assume that all untouched entries of p are set to zero and $T(\hat{T})$ means that T is chosen such that $\hat{T} \subset T$ (which is unique). The same result can be achieved just by inserting the definitions of the global basis and global degree of freedom into eq. (2.183).

2.5.6 Galerkin coarse grid approximation

In the multigrid algorithm 2.5, we assume that A_ℓ and b_ℓ are given on all levels (b_ℓ at least on the finest level). This could be done by assembly of the matrices on each level, known as *discretization coarse grid approximation* (DCA) [ZW91].

However given the prolongation $p_{\ell-1}$ and restriction r_ℓ, $A_{\ell-1}$ may be defined in terms of A_ℓ in a purely algebraic way [Hac85]

$$A_{\ell-1} := r_\ell A_\ell p_{\ell-1}, \tag{2.193}$$

which is called the *Galerkin coarse grid approximation* (GCA) of $A_{\ell-1}$. Using the relation $r_\ell = p_{\ell-1}^T$ we get equivalently

$$A_{\ell-1} = p_{\ell-1}^T A_\ell p_{\ell-1}. \tag{2.194}$$

The GCA is motivated by the fact that A_ℓ is the matrix version of an operator mapping from V_ℓ to V_ℓ'. Since we need an operator acting on $V_{\ell-1}$, we use the injection i ($p_{\ell-1}$) to the right. The result of $A_{\ell-1}$ should be an element of $V_{\ell-1}'$, thus we use the restriction $\circ i$ (r_ℓ or $p_{\ell-1}^T$) to the left.

Let's give some advantages/disadvantages of this approach:

- Only assembly of the finest A_ℓ is necessary to compute all other $A_{\ell-1}, \dots, A_0$.
- If A_ℓ is symmetric positive definite, and if $r_\ell = p_{\ell-1}^T$, then $A_{\ell-1}$ is also symmetric positive definite.

- The computation may be faster than assembly, but estimating the sparsity may be difficult.
- It is (in terms of the multigrid algorithm) a natural way of defining $A_{\ell-1}$.
- $A_{\ell-1}$ may not be the same as the assembled version (e.g. quadrature of form coefficients is more accurate on the finer mesh).

The GCA may be helpful to check if a computed prolongation operator is valid, by verifying

$$||r_\ell A_\ell p_{\ell-1} - A_{\ell-1}||_\infty < \epsilon, \tag{2.195}$$

where A_ℓ and $A_{\ell-1}$ were assembled in the conventional way and ϵ is a small number (depending on the machine precision).

2.5.7 Convergence of the V-cycle

In this section we restate the result from [Bra03], that alg. 2.5 is a contraction, with a contraction number independent of the number of levels and smoothing iterations, given a number of assumptions.

Theorem 2.30. (Convergence of the V-cycle [Bra03]) Assuming an equal number of Jacobi pre- and post-smoothing iterations $v_1 = v_2 = v$ with a suitable relaxation factor $\omega \geq \rho(A_\ell)$ and provided that,

1. the boundary value problem is $H^1(\Omega)$- or $H_0^1(\Omega)$-elliptic,
2. the boundary value problem is $H^2(\Omega)$ regular,
3. the spaces V_ℓ are nested and associated with conforming finite elements with uniform triangulations and
4. the nodal basis is used,

then alg. 2.5 is a contraction with

$$\|x - MGM(\ell, x_0, b)\|_{A_\ell} \leq \left(\frac{c}{c+2v}\right)^{1/2} \|x - x_0\|_{A_\ell}, \tag{2.196}$$

where c is a constant independent of ℓ and v.

Proof. See [Bra03, theorem 3.5] □

Proofs for multigrid methods are in general very technical and different approaches like *local Fourier analysis* or pure algebraic methods are possible. This theorem can also be proven under less regularity assumption, however the proof will then become even more technical.

2.5.8 Multigrid preconditioned CG

The preconditioned conjugate gradient method requires a symmetric and positive definite preconditioner. In this section we will shortly give the conditions for which the multigrid preconditioner is symmetric (cf. [Tat93] for the positive definiteness). The two-grid preconditioner applied to a right hand side vector b, is given by (cf. eq. (2.173)):

$$Mb = \mathcal{S}_1^{v_2}(\mathcal{S}_1^{v_1}(0,b) + p_0 A_0^{-1} r_1(b - A_1 \mathcal{S}_1^{v_1}(0,b)), b). \tag{2.197}$$

Assuming that the pre- and post smoothing methods are of the form

$$\mathcal{S}_1^{v_1}(x,b) = H_1 x + B_1 b \quad \text{and} \quad \mathcal{S}_1^{v_2}(x,b) = H_2 x + B_2 b \tag{2.198}$$

and using $r_0 = p_0^T$ the preconditioner may be generally written as

$$M = H_2(B_1 + \tilde{A}_1^{-1}(I - A_1 B_1)) + B_2. \tag{2.199}$$

where $\tilde{A}_1^{-1} = p_0 A_0^{-1} p_0^T$. From section 2.4.1, we know that the splitting methods (with $v = 1$) have the form $H_i = I - B_i A_1$, which gives

$$\begin{aligned} M &= (I - B_2 A_1)(B_1 + \tilde{A}_1^{-1}(I - A_1 B_1)) + B_2 \\ &= B_1 + B_2 - B_2 A_1 B_1 + (I - B_2 A_1)\tilde{A}_1^{-1}(I - A_1 B_1) \\ &= B_1 + B_2 - B_2 A_1 B_1 + \tilde{A}_1^{-1} + B_2 A_1 \tilde{A}_1^{-1}(B_1^T A_1)^T \\ &\quad - \left(B_2 A_1 \tilde{A}_1^{-1} + (B_1^T A_1 \tilde{A}_1^{-1})^T\right). \end{aligned} \tag{2.200}$$

As we see, M becomes symmetric, if $B_2 = B_1^T$. Now suppose we apply the smoother $v_i > 1$ times, then the smoother may be written as

$$\mathcal{S}_1^{v_i}(x,b) = \underbrace{H_i^{v_i}}_{\tilde{H}_i} x + \underbrace{\sum_{k=0}^{v_i-1} H^k B_i}_{\tilde{B}_i} b. \tag{2.201}$$

Now we would like to show, that $\tilde{H}_i = I - \tilde{B}_i A_1$, given that $H_i = I - B_i A_1$, i.e.

$$(I - B_i A_1)^{v_i} = I - \sum_{k=0}^{v_i-1} (I - B_i A_1)^k B_i A_1. \tag{2.202}$$

This is equivalent to showing the identity

$$X \sum_{k=0}^{v_i-1} (I - X)^k = I - (I - X)^{v_i} \tag{2.203}$$

with $X = B_i A_1$. Using the substitution $Y = I - X$ this is equivalent to showing

$$(I - Y) \sum_{k=0}^{v_i - 1} Y^k = I - Y^{v_i}, \tag{2.204}$$

which is obviously true. This shows that, given $B_2 = B_1^T$ and $H_i = I - B_i A_1$ (and symmetric A_ℓ), the two-grid preconditioner is symmetric, independently of the number of smoothing steps. By induction, the multigrid preconditioner (alg. 2.5 with $x = 0$) is then also symmetric, given that the γ_ℓ are constant for all levels.

3 Implementation

In this chapter we discuss all aspects of the implementation of a geometric multigrid method in FEniCS. In the first section we give an overview about the different tasks, which are required for the implementation as well as the design goals we want to achieve. As part of the overview we give also a short introduction to FEniCS and some of its components that were used. Finally we give an overview of the project structure and the implemented classes, which are discussed in detail in the remaining sections of this chapter.

3.1 Overview

3.1.1 Implementation tasks

The following list should give a brief overview of the implementation tasks, which must be performed.

1. Given a discrete variational problem on a mesh, we must be able to refine the mesh and describe the problem on the finer mesh. This means also that we are aware of the nested discrete function spaces $V_0 \subset V_1 \subset \ldots$, which are used to describe the solution/test space on each level.
2. Given two nested discrete function spaces $V_\ell \subset V_{\ell+1}$ we need a function to compute the linear prolongation p, in order to map coefficient vectors from V_ℓ to $V_{\ell+1}$. We also must be able to apply the restriction operator $r = p^T$, in order to map coefficient vectors from $V_{\ell+1}$ back to V_ℓ.
3. Different relaxation (smoothing) methods should be applicable. We should also provide basic methods such as relaxed Jacobi and SOR smoothing.
4. We must be able to describe the flow of the multigrid algorithm, i.e. we need a general way to describe V, W, F and other types of cycles.
5. We need to manage the working vectors u, d, e as well as pre- and post-smoothers and eventually a direct solver for each refinement level. And we need to manage all these levels in a convenient way.
6. The actual multigrid solver algorithm needs to be implemented.
7. Make the multigrid solver also applicable as preconditioner for Krylov subspace methods.
8. Have some convenient way to test the solver for a given problem.

3.1.2 Design goals

Beside the necessary implementation tasks, as listed above, there are some general design goals and concepts, which are good practice and were applied throughout the implementation:

- Use object oriented design.
- Use existing design patterns and interfaces.
- Be intuitively usable.
- Be easy configurable and extendable.
- Be maintainable, but also have good solver performance.
- Be compatible to and seamlessly integrated into FEniCS as much as possible.
- Provide a good documentation.

3.1.3 FEniCS and its components

In this section we give a brief introduction to the finite element toolbox FEniCS and some of its central components.

FEniCS

The FEniCS Project is designed as an umbrella project for a collection of interoperable components. The core components are (cited from [Wik]):

- UFL (Unified Form Language), a domain-specific language embedded in Python for specifying finite element discretizations of differential equations in terms of finite element variational forms;
- FIAT (Finite element Automatic Tabulator), a Python module for generation of arbitrary order finite element basis functions on simplices;
- FFC (FEniCS Form Compiler), a compiler for finite element variational forms taking UFL code as input and generating UFC output;
- UFC (Unified Form-assembly Code), a C++ interface consisting of low-level functions for evaluating and assembling finite element variational forms;
- Instant, a Python module for inlining C and C++ code in Python;
- DOLFIN, a C++/Python library providing data structures and algorithms for finite element meshes, automated finite element assembly, and numerical linear algebra.

This component interplay is visualized in fig. 3.1.

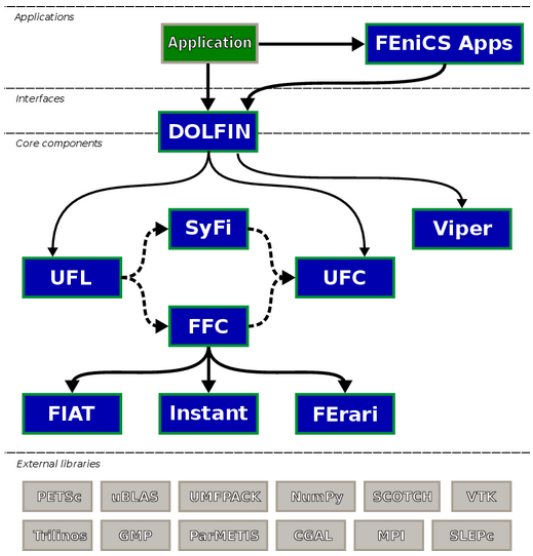


Figure 3.1: A schematic overview of the FEniCS components and their interplay (graphics by Anders Logg [LMW+12]).

DOLFIN

DOLFIN functions as the main problem solving environment and user interface. Its functionality integrates the other FEniCS components and handles communication with external libraries such as PETSc, Trilinos, MTL4, and uBLAS for numerical linear algebra, ParMETIS and SCOTCH for mesh partitioning, and MPI and OpenMP for distributed computing [Wik].

In order to have a rough understanding how to solve a PDE with DOLFIN, we will solve Poisson's equation on the unit square with mixed boundary conditions:

$$-\Delta u = f \quad \text{in } \Omega, \tag{3.1}$$
$$u = 0 \quad \text{on } \Gamma_D, \tag{3.2}$$
$$\frac{\partial u}{\partial n} = g \quad \text{on } \Gamma_N, \tag{3.3}$$

and where

$$\Omega = (0,1) \times (0,1), \tag{3.4}$$
$$\Gamma_D = \{(x,y) \in \partial\Omega : x = 0 \vee x = 1\}, \tag{3.5}$$
$$\Gamma_N = \partial\Omega \setminus \Gamma_D, \tag{3.6}$$
$$f(x,y) = 10\exp(-((x-0.5)^2 + (y-0.5)^2)/0.02), \tag{3.7}$$
$$g(x,y) = \sin(5x). \tag{3.8}$$

The weak formulation for this problem is: find $u \in H_D(\Omega)$ such that

$$a(u, v) = L(v) \quad \forall v \in H_D^1(\Omega), \tag{3.9}$$

with $H_D^1(\Omega) = \{v \in H^1(\Omega) : \text{trace}(v) = 0 \text{ on } \Gamma_D\}$ and with the bilinear and linear form

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x, \tag{3.10}$$

$$L(v) = \int_\Omega f v \, \mathrm{d}x + \int_{\Gamma_N} g v \, \mathrm{d}s. \tag{3.11}$$

The first necessary step will be to translate this variational formulation to UFL (see [Aln+12] for a good reference). This means creating a file `Poisson.ufl` with the following contents:

```
1  cell = triangle
2  element = FiniteElement("Lagrange", cell, 1)
3
4  u = TrialFunction(element)
5  v = TestFunction(element)
6  f = Coefficient(element)
7  g = Coefficient(element)
8
9  a = inner(grad(u), grad(v))*dx
10 L = f*v*dx + g*v*ds
```

Precisely the code has the following meaning:

- Line 1: Define the cell shape (i.e. the pieces that are used for the triangulation of Ω). Currently DOLFIN supports only intervals (1d), triangles (2d) and tetrahedra (3d), while UFL also supports quadrilaterals and hexahedra.
- Line 2: Define the finite element space. Here we use scalar Lagrange elements (continuous piecewise polynomial functions) of first order.
- Line 4: Define unknown u to be solved for. This called the trial function.
- Line 5: Define the test functions v.
- Line 6-7: Define f and g to be coefficients (a normal function that is used within the forms).
- Line 9: Define the bilinear form a. This is automatically identified as bilinear form, since it depends on a `TrialFunction` and a `TestFunction`.
- Line 10: Define the linear form L. This is automatically identified as linear form, since it depends only on a `TrialFunction`.

Now the UFL code must be translated to C++ using FFC. This is done by executing the following command line (see [Log+12] for more information):

```
ffc -f split -l dolfin -O -f eliminate_zeros \
    -f precompute_basis_const Poisson.ufl
```

The parameters have the following meaning:

- `-f split`: Generates separate header and cpp files. This reduces project recompile time, since without this flag all code resides in the header.
- `-l dolfin`: Tells FFC to add some DOLFIN specific bindings.
- `-O`: Generate optimized code with a lower operation count.
- `-f eliminate_zeros`: Tables containing basis function values will be compressed such that they only contain non zero values.
- `-f precompute_basis_const`: Generates code that pre-computes terms, which are constant in the loops involving basis indices.

After completion there will be two new files `Poisson.h` and `Poisson.cpp`, which contain several classes representing everything which was described in the `Poisson.ufl` previously. More precisely after inclusion of the `Poisson.h` header we find within the namespace `Poisson` the following classes:

- `CoefficientSpace_f` and `CoefficientSpace_g`,
- `Form_a_FunctionSpace_0` and `Form_a_FunctionSpace_1`,
- `Form_L_FunctionSpace_0`,
- `Form_a` and `Form_L`.

Further we have a number of alias names (typedefs) for these classes:

- `Form_L_FunctionSpace_1` for `CoefficientSpace_f`,
- `Form_L_FunctionSpace_2` for `CoefficientSpace_g`,
- `BilinearForm` and `JacobianForm` for `Form_a`,
- `LinearForm` and `ResidualForm` for `Form_L`,
- `Form_a::TestSpace` for `Form_a_FunctionSpace_0`,
- `Form_a::TrialSpace` for `Form_a_FunctionSpace_1`,
- `Form_L::TestSpace` for `Form_L_FunctionSpace_0`,
- `Form_L::CoefficientSpace_f` for `CoefficientSpace_f`,
- `Form_L::CoefficientSpace_g` for `CoefficientSpace_g`,
- `FunctionSpace` for `Form_a::TestSpace`.

All `*Form*` classes are derived from the DOLFIN class `dolfin::Form` and all `*Space*` classes are derived from `dolfin::FunctionSpace`, which is explained in more detail later. For now let us go back to the solution of the Poisson problem.

In the next step these classes are utilized in order to assemble the bilinear form a to a matrix and the linear form L to a vector, apply boundary conditions and solve the resulting linear system. To do so, we create a file `main.cpp` with the following contents:

```
#include <dolfin.h>
#include "Poisson.h"

using namespace dolfin;

// Source term (right-hand side)
class Source : public Expression {
  void eval(Array<double>& values, const Array<double>& x) const  {
    double dx = x[0] - 0.5;
    double dy = x[1] - 0.5;
    values[0] = 10*exp(-(dx*dx + dy*dy) / 0.02);
  }
};

// Normal derivative (Neumann boundary condition)
class dUdN : public Expression {
  void eval(Array<double>& values, const Array<double>& x) const {
    values[0] = sin(5*x[0]);
  }
};

// Sub domain for Dirichlet boundary condition
class DirichletBoundary : public SubDomain {
  bool inside(const Array<double>& x, bool on_boundary) const {
    return x[0] < DOLFIN_EPS or x[0] > 1.0 - DOLFIN_EPS;
  }
};

int main(int argc, char* argv[])
{
  // Create mesh and function space
  UnitSquare mesh(32, 32);
  Poisson::FunctionSpace V(mesh);

  // Define boundary condition
  Constant u0(0.0);
  DirichletBoundary boundary;
  DirichletBC bc(V, u0, boundary);

  // Define variational forms
  Poisson::BilinearForm a(V, V);
  Poisson::LinearForm L(V);
  Source f;
  dUdN g;
  L.f = f;
```

```
46    L.g = g;
47
48    // Compute solution
49    Function u(V);
50    solve(a == L, u, bc);
51
52    // Plot solution
53    plot(u);
54    interactive();
55
56    return 0;
57  }
```

The code is more or less self explaining, but let us briefly comment it:

- Line 1-2: Include the DOLFIN headers and the previously generated `Poisson.h`.
- Line 7-13: Define an `Expression` class for evaluating the right hand side f.
- Line 16-20: Define an `Expression` class for evaluating the right hand side g.
- Line 23-27: Define a `SubDomain` class which marks Γ_D.
- Line 32: Create a uniform 32x32 triangle mesh over the unit square Ω.
- Line 33: Create a discrete function space for the mesh. As previously noted, `Poisson::FunctionSpace` is an alias for `Poisson::Form_a::TestSpace`.
- Line 36-38: Define a `DirichletBC` object, which describes the homogeneous Dirichlet boundary conditions on Γ_D.
- Line 41-46: Create instances of the linear and bilinear forms and assign the coefficients to it. Note that coefficients are objects derived from `dolfin::GenericFunction` like `dolfin::Expression` and `dolfin::Function`.
- Line 49-50: Create solution function `u` (which is represented by the coefficient vector `u.vector()`) and solve the problem (the solution is stored to `u`).
- Line 53-54: Plot the solution (see fig. 3.2) and switch to an interactive view.

Line 50, `solve(a == L, u, bc)` is the part where most of the work is done. That is why we want to describe what is happening there in more detail. Note that we modified some of the shown original DOLFIN source code for brevity in all following listings. First of all, the `a == L` is an overloaded operator expression, defined in `dolfin/fem/Form.cpp`:

```
1  Equation Form::operator==(const Form& rhs) const
2  {
3    Equation equation(reference_to_no_delete_pointer(*this),
4          reference_to_no_delete_pointer(rhs));
5    return equation;
6  }
```

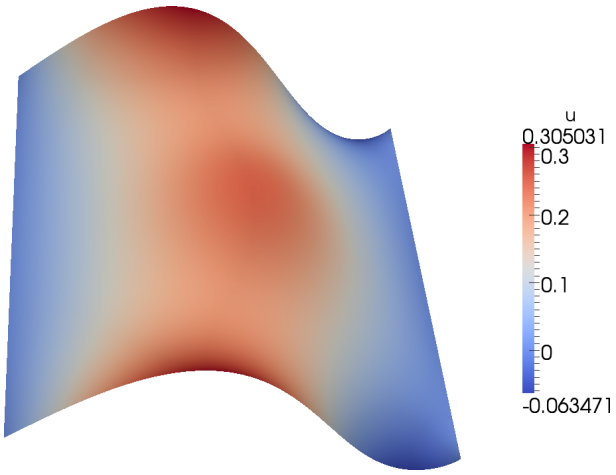


Figure 3.2: Solution of the Poisson problem.

which creates an `Equation` object that simply holds `a` and `L`. After that, a `solve` function within `dolfin/fem/solve.cpp` is called, which just adds the single boundary condition to a list and calls a more common `solve` function:

```
1 void dolfin::solve(const Equation& equation, Function& u,
2                    const BoundaryCondition& bc, Parameters params)
3 {
4   // Create list containing single boundary condition
5   std::vector<const BoundaryCondition*> bcs;
6   bcs.push_back(&bc);
7
8   solve(equation, u, bcs, params);   // Call common solve function
9 }
```

where the last parameter defaults to `dolfin::empty_parameters()`. As you can see, this calls just another function in the same file:

```
1  void dolfin::solve(const Equation& equation, Function& u,
2    std::vector<const BoundaryCondition*> bcs, Parameters params)
3  {
4    if (equation.is_linear()) {   // Solve linear problem
5      LinearVariationalProblem problem(*equation.lhs(), *equation.rhs(),
           u, bcs);
6      LinearVariationalSolver solver(problem);
7      solver.parameters.update(params);
8      solver.solve();
9    }
10   else {   // Solve nonlinear problem
11     ...
12   }
13 }
```

In our case the equation is linear and so the function uses the `LinearVariationalSolver` within `dolfin/fem/LinearVariationalSolver.cpp`. Again the `LinearVariationalProblem` is just an object to hold the left and right hand side, as well as the boundary conditions `bcs` and the function `u` for storing the solution:

```
LinearVariationalSolver::
LinearVariationalSolver(LinearVariationalProblem& problem)
  : problem(reference_to_no_delete_pointer(problem))
{
  // Set parameters
  parameters = default_parameters();
}
```

Where the `default_parameters()` are:

```
static Parameters default_parameters()
{
  Parameters p("linear_variational_solver");

  p.add("linear_solver", "default");
  p.add("preconditioner", "default");
  p.add("symmetric", false);
  ...
  return p;
}
```

The `solve()` method then basically creates a matrix and a vector in order to assemble the linear system and solves the system by using a linear solver.

```
void LinearVariationalSolver::solve()
{
  // Get parameters
  std::string solver_type   = parameters["linear_solver"];
  const std::string pc_type = parameters["preconditioner"];
  const bool symmetric      = parameters["symmetric"];

  // Get problem data
  dolfin_assert(problem);
  boost::shared_ptr<const Form> a(problem->bilinear_form());
  boost::shared_ptr<const Form> L(problem->linear_form());
  boost::shared_ptr<Function> u(problem->solution());
  std::vector<boost::shared_ptr<const BoundaryCondition> > bcs(
    problem->bcs());

  // Create matrix and vector
  dolfin_assert(u->vector());
  boost::scoped_ptr<GenericMatrix> A(u->vector()->factory().
      create_matrix());
  boost::scoped_ptr<GenericVector> b(u->vector()->factory().
      create_vector());
```

```
20
21    // Different assembly depending on whether or not the system is
          symmetric
22    if (!symmetric)
23    {
24      // Assemble linear system
25      assemble(*A, *a);
26      assemble(*b, *L);
27
28      // Apply boundary conditions
29      for (uint i = 0; i < bcs.size(); i++) {
30        bcs[i]->apply(*A, *b);
31      }
32    }
33
34    // Solve linear system
35    LinearSolver solver(solver_type, pc_type);
36    solver.solve(*A, *u->vector(), *b);
37  }
```

We investigate this further by having a look at `dolfin::assemble` within `fem/assemble.cpp`:

```
1  void dolfin::assemble(GenericTensor& A, const Form& a,
2    bool reset_sparsity, bool add_values, bool finalize_tensor)
3  {
4    Assembler::assemble(A, a, 0, 0, 0, reset_sparsity, add_values,
         finalize_tensor);
5  }
```

which calls `Assembler::assemble` within `fem/Assembler.cpp`:

```
1  void Assembler::assemble(GenericTensor& A, const Form& a,
2    const MeshFunction<uint>* cell_domains,
3    const MeshFunction<uint>* exterior_facet_domains,
4    const MeshFunction<uint>* interior_facet_domains,
5    bool reset_sparsity, bool add_values, bool finalize_tensor)
6  {
7    // Get cell domains
8    if (!cell_domains || cell_domains->size() == 0)
9    {
10     cell_domains = a.cell_domains_shared_ptr().get();
11     if (!cell_domains)
12       cell_domains = a.mesh().domains().cell_domains(a.mesh()).get();
13   }
14
15   // Get interior and exterior facet domains
16   ...
17
18   // Create data structure for local assembly data
```

```
19    UFC ufc(a);
20
21    // Gather off-process coefficients
22    const std::vector<boost::shared_ptr<const GenericFunction> >
23      coefficients = a.coefficients();
24    for (uint i = 0; i < coefficients.size(); ++i)
25      coefficients[i]->gather();
26
27    // Initialize global tensor
28    AssemblerTools::init_global_tensor(A, a, reset_sparsity, add_values)
          ;
29
30    // Assemble over cells
31    assemble_cells(A, a, ufc, cell_domains, 0);
32
33    // Assemble over exterior facets
34    assemble_exterior_facets(A, a, ufc, exterior_facet_domains, 0);
35
36    // Assemble over interior facets
37    assemble_interior_facets(A, a, ufc, interior_facet_domains, 0);
38
39    // Finalize assembly of global tensor
40    if (finalize_tensor)
41      A.apply("add");
42  }
```

Here the most important functions are:

- `assemble_cells`: Refers to the assembly of the expressions within the form, which are integrals over Ω (`dx` in UFL).
- `assemble_exterior_facets`: Refers to the assembly of the expressions within the form, which are integrals over $\partial\Omega$ (`ds` in UFL).
- `assemble_interior_facets`: Refers to the assembly of the expressions within the form, which are integrals over the facets of each cell excluding the facets, which are part of the boundary (`dS` in UFL). This is not used very often.

For brevity, we will have a closer look at `assemble_cells` only:

```
1   void Assembler::assemble_cells(GenericTensor& A, const Form& a,
2     UFC& ufc, const MeshFunction<uint>* domains, std::vector<double>*
          values)
3   {
4     // Extract mesh
5     const Mesh& mesh = a.mesh();
6
7     // Form rank
8     const uint form_rank = ufc.form.rank();
9
```

```
10    // Collect pointers to dof maps
11    std::vector<const GenericDofMap*> dofmaps;
12    for (uint i = 0; i < form_rank; ++i)
13      dofmaps.push_back(a.function_space(i)->dofmap().get());
14
15    // Vector to hold dof map for a cell
16    std::vector<const std::vector<uint>* > dofs(form_rank);
17
18    // Cell integral
19    dolfin_assert(ufc.cell_integrals.size() > 0);
20    ufc::cell_integral* integral = ufc.cell_integrals[0].get();
21
22    // Assemble over cells
23    for (CellIterator cell(mesh); !cell.end(); ++cell)
24    {
25      // Get integral for sub domain (if any)
26      if (domains && domains->size() > 0) {
27        ...
28      }
29
30      // Update to current cell
31      ufc.update(*cell);
32
33      // Get local-to-global dof maps for cell
34      for (uint i = 0; i < form_rank; ++i)
35        dofs[i] = &(dofmaps[i]->cell_dofs(cell->index()));
36
37      // Tabulate cell tensor
38      integral->tabulate_tensor(&ufc.A[0], ufc.w(), ufc.cell);
39
40      // Add entries to global tensor. Either store values cell-by-cell
41      // (currently only available for functionals)
42      if (values && ufc.form.rank() == 0)
43        (*values)[cell->index()] = ufc.A[0];
44      else
45        A.add(&ufc.A[0], dofs);
46    }
47  }
```

This function basically iterates over all cells of the mesh and

1. gets the local-to-global mapping of degrees of freedom for the cell and
2. computes the local contribution to the form from the integral over the cell, by calling `integral->tabulate_tensor`. The local contributions are stored to the temporary array `ufc.A`, which are then added to the tensor `A` at the corresponding global dof positions.

The specific implementation of the integration routines, which are called with `integral->tabulate_tensor`, as well as the `dofmaps` are contained in the FFC

generated `Poisson.cpp` for each form. `w` is an two-dimensional array of the (possible vector valued) expansion coefficients of the function to be integrated restricted to the cell, which is updated every loop with a call to `ufc.update(*cell)`.

For example the cell integration for the linear form of our example reads as:

```
void poisson_cell_integral_1_0::tabulate_tensor(
  double* A, const double * const * w, const ufc::cell& c) const
{
    // Extract vertex coordinates
    const double * const * x = c.coordinates;

    // Compute Jacobian of affine map from reference cell
    const double J_00 = x[1][0] - x[0][0];
    const double J_01 = x[2][0] - x[0][0];
    const double J_10 = x[1][1] - x[0][1];
    const double J_11 = x[2][1] - x[0][1];

    // Compute determinant of Jacobian
    double detJ = J_00*J_11 - J_01*J_10;

    // Compute inverse of Jacobian

    // Set scale factor
    const double det = std::abs(detJ);

    // Compute geometry tensor
    const double G0_0 = det*w[0][0]*(1.0);
    const double G0_1 = det*w[0][1]*(1.0);
    const double G0_2 = det*w[0][2]*(1.0);

    // Compute element tensor
    A[0] = 0.0833333333*G0_0 + 0.0416666667*G0_1 + 0.0416666667*G0_2;
    A[1] = 0.0416666667*G0_0 + 0.0833333333*G0_1 + 0.0416666667*G0_2;
    A[2] = 0.0416666667*G0_0 + 0.0416666667*G0_1 + 0.0833333333*G0_2;
}
```

In the notion `poisson_cell_integral_1_0`, the `1` denotes the form 1 (form 0 is the bilinear form within the UFL file), the `0` denotes the first cell integral of the form. If we would have another `dx` over a different subdomain within the linear form, there would be another `poisson_cell_integral_1_1`. Since the coefficient f for the cell integral within L is scalar valued, `w` is here like a vector. And as we use Lagrange elements, `w` just contains the values of f at the three vertices of the triangular cell. In this case, FFC used the Gauss-Legendre-Jacobi quadrature rule [Rat+08], which gives the weightings as in the last three lines, after transforming the cell to the reference cell.

After the form assembly, the solution is computed using a linear solver, available within the PETSc library, which is explained in the next section. FEniCS also makes

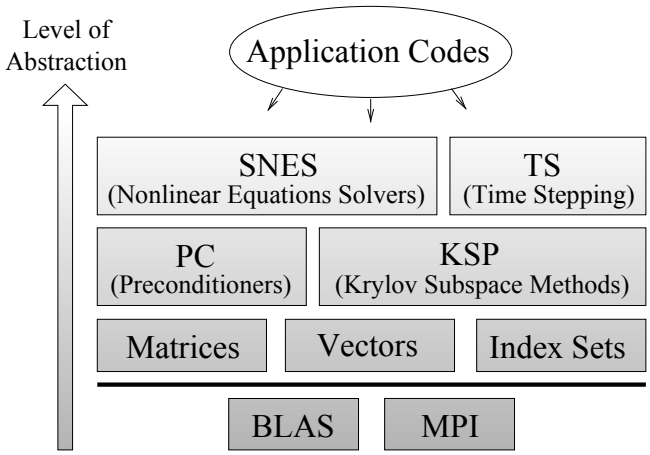


Figure 3.3: Components of the Portable, Extensible Toolkit for Scientific Computation (PETSc) library (graphics by [Bal+12a]).

heavily use of the shared pointer concept using the boost C++ library, which is explained thereafter.

PETSc

The Portable, Extensible Toolkit for Scientific Computation (PETSc) [Bal+12b] is a collection of data structures and routines for the scalable (parallel) solution of scientific problems, modeled by partial differential equations. It employs the Message Passing Interface (MPI) standard for all message-passing communication. Figure 3.3 shows the various components of PETSc, which includes (cited from [Bal+12a])

- index sets (IS), for indexing into vectors, renumbering, etc,
- vectors (Vec),
- matrices (Mat) (generally sparse),
- managing interactions between mesh data structures, vectors and matrices (DM),
- over fifteen Krylov subspace methods (KSP),
- dozens of preconditioners, including multigrid, block solvers, and sparse direct solvers (PC),
- nonlinear solvers (SNES) and
- timesteppers for solving time-dependent (nonlinear) PDEs, including support for differential algebraic equations (TS).

PETSc is also employed within FEniCS as default linear algebra back-end. Since we had to use some special matrix operations, that would be otherwise painfully slow,

and because none of the other back-ends offers as many possibilities as PETSc, we confined ourselves to use PETSc within our multigrid solver and preconditioner.

At this point it should also be noticed, that PETSc has support for multigrid preconditioning [Pet], but only in a sense of a skeleton, where still all prolongation, restriction and coarser grid matrices need to be created manually. Since this is quite more inflexible and more difficult to debug, we did not follow such an approach. Also we can not expect much performance improvement from such an integrated approach, since the most expensive operations (like matrix-vector multiplication) are carried out in the same way.

Boost C++ libraries

The Boost C++ library [Boo] is a free library, consisting of multiple portable sub-libraries. The sub-libraries fulfil various purposes, like

- processing of strings and text,
- lists, sets, hash tables and other containers,
- sorting algorithms, iterators,
- memory management,
- multithreading and network support and
- image processing.

Since C++ does not have a garbage collection mechanism, developers often face the problem of memory leakage in their programs. To avoid this, FEniCS uses the `boost::shared_ptr<T>` class, which stores a pointer to a dynamically allocated object. Whenever a copy of such an object is made, an internal shared reference counter is increased. Whenever such an object is destroyed, the reference counter is decreased by one. If the shared reference counter reaches zero, then the dynamically allocated object is deleted. As an example, the following piece of code does not create any memory leaks or segmentation faults:

```
1 {
2   boost::shared_ptr<int> x;
3   {
4     boost::shared_ptr<int> y(new int);
5     x = y;
6   }
7   *x = 3;
8 }
```

As a guideline to avoid memory leaks, pointers of type `T*` are simply replaced by `boost::shared_ptr<T>`, and every allocation using the `new` keyword should be kept as a shared pointer.

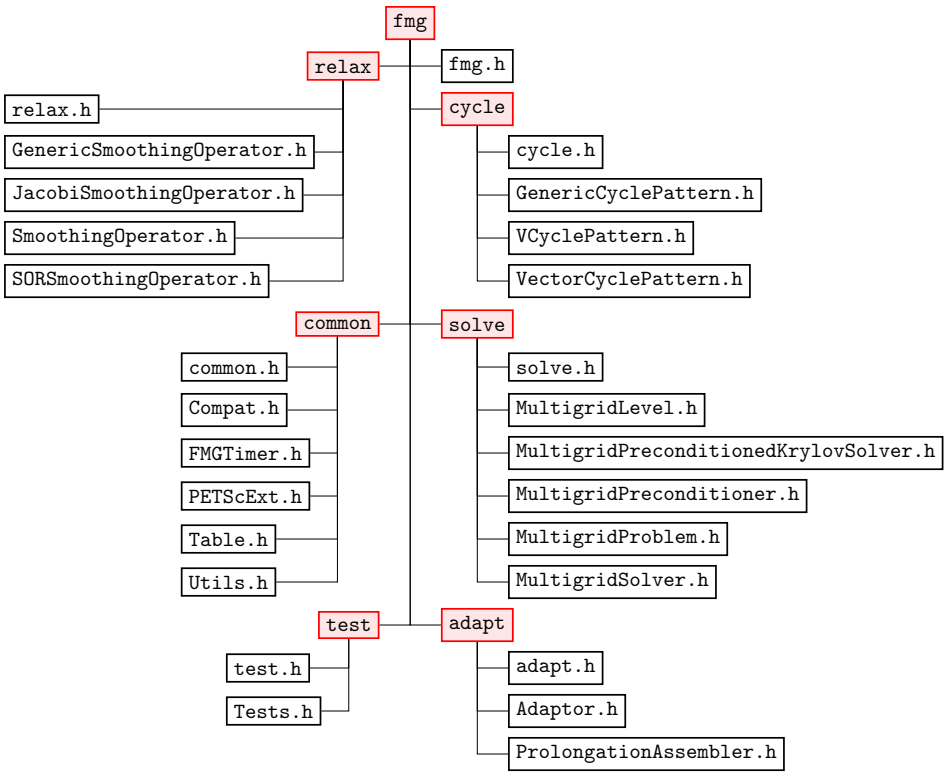


Figure 3.4: File and directory structure of the `FMG` project. Note: only the header files are displayed, the `cpp` files are named accordingly.

3.1.4 FMG Project structure

Figure 3.4 shows the file and directory structure of the project. The project was split up into six modules (directories) to organize the files.

- `adapt`: Classes for mesh refinement and creation of the prolongation operator.
- `common`: Classes that have a common purpose (are used by other classes).
- `cycle`: Classes for different types of cycles.
- `relax`: Classes for relaxation/smoothing.
- `solve`: Classes for multigrid solving, preconditioning and problem hierarchy.
- `test`: Classes for testing the multigrid (preconditioned) solver.

Each of these directories contains a header `"directory name.h"`, which can be used to include all files of that directory. The headers of the whole project may be included with the `fmg.h` file.

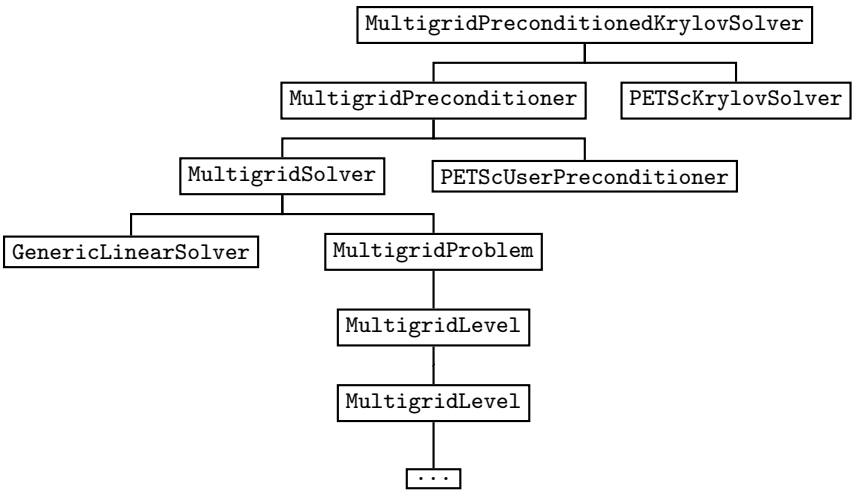


Figure 3.5: Class hierarchical view of the project.

Another view than in terms of files is given in fig. 3.5, which shows the hierarchical dependency of classes from each other. The implementation of each of these classes will be described in the next sections.

3.2 Problem and mesh refinement - The fmg::Adaptor class

One of the first things when implementing a multigrid algorithm concerns the mesh refinement or mesh coarsening. Along with the refined mesh also the `FunctionSpaces` (and all depending objects) of the problem, as well as the boundary conditions must be refined in order to solve the problem on the refined mesh. Let us consider listing 3.1.3 as basis for our examples. Instead of writing `solve(a == L, u, bc)` one could also write

```
1 LinearVariationalProblem problem(a, L, u, bc);
2 LinearVariationalSolver solver(problem);
3 solver.solve();
```

If we would like to solve the `problem` on a finer mesh, DOLFIN offers various methods for mesh and problem refinement within `adaptivity/adapt.h`. Here is an excerpt of this header:

```
1 // Refine mesh uniformly
2 const Mesh& adapt(const Mesh& mesh);
3
4 // Refine mesh based on cell markers
5 const Mesh& adapt(const Mesh& mesh,
```

```
6       const MeshFunction<bool>& cell_markers);
7
8  // Refine function space based on refined mesh
9  const FunctionSpace& adapt(const FunctionSpace& space,
10      boost::shared_ptr<const Mesh> adapted_mesh);
11
12 // Refine GenericFunction based on refined mesh
13 const GenericFunction& adapt(const GenericFunction& function,
14      boost::shared_ptr<const Mesh> adapted_mesh);
15
16 // Refine Dirichlet bc based on refined mesh
17 const DirichletBC& adapt(const DirichletBC& bc,
18      boost::shared_ptr<const Mesh> adapted_mesh,
19      const FunctionSpace& S);
20
21 // Refine linear variational problem based on mesh
22 const LinearVariationalProblem& adapt(
23      const LinearVariationalProblem& problem,
24      boost::shared_ptr<const Mesh> adapted_mesh);
```

Each of these methods return an instance to the refined object. However if the object (`Mesh` for example) was already previously refined, then the previously refined object is returned. This is because each of these objects derive from the `Hierarchical<T>` class, which allows each of these objects to have a parent and a child object of the same type. Calling `adapt(x, mesh)` first checks if `x` has an child by calling `x.has_child()`, if this is the case this child `x.child()` is returned. If there is no child set, the object is refined. The refined object is set as child for the original object and the original object is set as parent for the refined object.

When refining a `LinearVariationalProblem` then all forms and their coefficients and function spaces as well the boundary conditions and the solution are refined in the same manner. To come back to the example, this is how to solve the problem on a refined grid and plot the solution:

```
1 LinearVariationalProblem problem(a, L, u, bc);
2 adapt(mesh);
3 adapt(problem, mesh.child_shared_ptr());
4 LinearVariationalSolver solver(problem.child());
5 solver.solve();
6 plot(u.child());
```

The `fmg::Adaptor` class provides basically the same functions for refining a `Mesh`, `LinearVariationalProblem` and `NonlinearVariationalProblem`. However some modifications were done in order to improve the performance of the refinement:

1. **Not all mesh entities of the refined mesh are initialized**: In the original DOLFIN source code, all mesh entities (vertices, edges, faces, etc.) of the

refined mesh are initialized, which were already initialized in the original mesh. We observed that this is too much. The `fmg::Adaptor` method only initializes the mesh entities, which are necessary to run the code. This saves considerable time.

2. **The solution function is not interpolated to the new mesh**: By default the DOLFIN `adapt` methods interpolate the solution to the new mesh. This is usually not necessary, since the solution does not contain anything meaningful at this stage. However if the user wishes to do so, the `adapt` methods within `fmg::Adaptor` provide an optional argument to enable interpolation.

3. **Discrete form coefficients (of type `Function`) are not interpolated to the new mesh**: By default the DOLFIN `adapt` methods interpolate form coefficients to the new mesh. This is disabled by default within the `fmg::Adaptor` methods, since the interpolated coefficients will be of poor quality on the fine mesh. So it is generally better to set these coefficients explicitly on the finer meshes. Note that this affects only coefficients of type `Function`. `Expression`s and other `GenericFunction`s are always valid on the new mesh. But the user is also free here to provide an optional argument to enable interpolation.

Now let us have a look at the `fmg::Adaptor` class, which may be used in the same way as in the example above:

```
1  class Adaptor
2  {
3  public:
4    /// Refine mesh uniformly.
5    static const dolfin::Mesh& adapt(const dolfin::Mesh& mesh);
6
7    /// Refine mesh based on cell markers.
8    static const dolfin::Mesh& adapt(const dolfin::Mesh& mesh,
9        const dolfin::MeshFunction<bool>& cell_markers);
10
11   /// Refine linear variational problem based on mesh.
12   static const dolfin::LinearVariationalProblem& adapt(
13       const dolfin::LinearVariationalProblem& problem,
14       const dolfin::Mesh& adapted_mesh,
15       bool interpolate_coef = false,
16       bool interpolate_solution = false);
17   ...
18 }
```

3.3 Prolongation - The fmg::ProlongationAssembler class

After we have a refined mesh and function space, we can implement the prolongation operator, as discussed in section 2.5.5 theoretically, i.e. computing

$$(p)_{\iota_{\hat{T}}(i),\iota_{T(\hat{T})}(j)} = (p_{T(\hat{T})})_{ij}, \quad \forall\, \hat{T} \in \hat{\mathcal{T}},\ i = 1,\dots,\hat{n},\ j = 1,\dots,n, \tag{3.12}$$

and where

$$(p_T)_{ij} = \hat{\ell}_i(\phi_j), \quad i = 1,\dots,\hat{n},\ j = 1,\dots,n, \tag{3.13}$$

and given the local-to-global dof mappings

$$\iota_T : [1, n_T] \mapsto [1, n] \text{ and} \tag{3.14}$$
$$\iota_{\hat{T}} : [1, n_{\hat{T}}] \mapsto [1, \hat{n}]. \tag{3.15}$$

Further we assume that all untouched entries of p are set to zero and $T(\hat{T})$ means that T is chosen such that $\hat{T} \subset T$ (which is unique).

A simplified (and hence less efficient) version of the implementation is given in the following listings. Note that we omitted the `std`, `dolfin` and `boost` namespaces in the code for simplicity.

```
1  shared_ptr<GenericMatrix> ProlongationAssembler::assemble_matrix(
2    const FunctionSpace& space,
3    const FunctionSpace& refined_space)
4  {
5    const Mesh& mesh = *(space.mesh());
6    const Mesh& rmesh = *(refined_space.mesh());
7    vector<shared_ptr<const FunctionSpace> > subspaces;
8    vector<shared_ptr<const FunctionSpace> > rsubspaces;
9    CellIterator cellit(mesh);
10   Cell* cell = &(*cellit);
11   UFCCell ufc_rcell(rmesh);
12   UFCCell ufc_cell(*cell);
13   basis_func basis;
14   shared_ptr<GenericMatrix> P;
15   shared_ptr<Mat> PMat(new Mat);
16
17   // create matrix
18   MatCreateSeqAIJ(PETSC_COMM_SELF, refined_space.dim(),
19     space.dim(), 50, NULL, PMat.get());
20   P.reset(new PETScMatrix(PMat));
21
22   // get list of scalar subspaces
23   Utils::extract_subspaces_recursive(
24     reference_to_no_delete_pointer(space), subspaces);
25   Utils::extract_subspaces_recursive(
```

```
26      reference_to_no_delete_pointer(refined_space), rsubspaces);
27
28    // iterate over cells of refined mesh
29    for (CellIterator rcell(rmesh); !rcell.end(); ++rcell)
30    {
31      ufc_rcell.update(*rcell);
32
33      // find cell which contains rcell
34      int icell = mesh.intersected_cell(rcell->midpoint());
35      cell = &(cellit[icell]);
36      ufc_cell.update(*cell);
37
38      // compute prolongation for each subspace
39      for (size_t isub = 0; isub < subspaces.size(); isub++)
40      {
41        const FunctionSpace& subspace = *(subspaces[isub]);
42        const FunctionSpace& rsubspace = *(rsubspaces[isub]);
43        const FiniteElement& element = *(subspace.element());
44        const FiniteElement& relement = *(rsubspace.element());
45        const GenericDofMap& dofmap = *(subspace.dofmap());
46        const GenericDofMap& rdofmap = *(rsubspace.dofmap());
47        const vector<uint>& dofs = dofmap.cell_dofs(cell->index());
48        const vector<uint>& rdofs = rdofmap.cell_dofs(rcell->index());
49
50        // iterate over all dof coordinates
51        for (size_t irdof = 0; irdof < rdofs.size(); irdof++)
52        {
53          // iterate over dofs of cell
54          for (size_t idof = 0; idof < dofs.size(); idof++)
55          {
56            // evaluate dof
57            basis.cell = &ufc_cell;
58            basis.element = &element;
59            basis.ibasis = idof;
60            value = relement.evaluate_dof(irdof, basis, ufc_rcell);
61
62            // set matrix entry
63            P->setitem(make_pair(rdofs[irdof], dofs[idof]), value);
64          }
65        }
66      }
67    }
68
69    return P;
70  }
```

And the `basis_func` type is defined by:

```
1  class basis_func : public ufc::function
2  {
3  public:
4    const FiniteElement* element;
5    const ufc::cell* cell;
6    uint ibasis;
7
8    inline void evaluate(double* values, const double* x,
9      const ufc::cell& c) const
10   {
11     element->evaluate_basis(ibasis, values, x, *cell);
12   }
13 };
```

Let us discuss the code of `assemble_matrix` in more detail:

- Line 1-3: First of all, the static method `assemble_matrix` of `ProlongationAssembler` has two arguments. The first one is the function space `space`, which corresponds to the space $V \subset \hat{V}$, the second one is the function space `refined_space`, which corresponds to the space $\hat{V}$.
- Line 18-20: Create a sparse matrix P which corresponds to the matrix eq. (2.192).
- Line 23-23: In the case V and $\hat{V}$ are vector valued (or generally tensor valued) spaces or product spaces, each of the spaces are recursively decomposed into a set of scalar valued subspaces V_i in the sense that $V = \bigoplus_i V_i e_i$, where e_i is the i-th unit vector. This is done because the treatment of scalar spaces is much more simple than compound spaces.
- Line 29: The `for`-loop accounts for the $\forall\, \hat{T} \in \hat{\mathcal{T}}$.
- Line 31: This is just necessary to update cell entities and coordinates of the refined UFC cell.
- Line 34-36: Here we determine the cell $T(\hat{T})$, by finding the cell of T, which intersects the midpoint of the `rcell`.
- Line 39: Iterate over every subspace.
- Line 47-48: Get the dof maps. The `dofs` vector corresponds to the mapping ι_T and the `rdofs` vector corresponds to the mapping $\iota_{\hat{T}}$
- Line 51: This accounts for the $i = 1, \ldots, n_{\hat{T}}$ in eq. (2.192).
- Line 54: This accounts for the $j = 1, \ldots, n_T$ in eq. (2.192).
- Line 57-60: These lines are for the evaluation $\hat{\ell}_i(\phi_j)$.
- Line 63: This line does the assignment $(p)_{\iota_{\hat{T}}(i), \iota_{T(\hat{T})}(j)} = \hat{\ell}_i(\phi_j)$.

However the actual implementations incorporates some more settings, which are shown in the following listing:

```
static shared_ptr<GenericMatrix> assemble_matrix(
    const FunctionSpace& space,
    const FunctionSpace& refined_space,
    const GenericLinearAlgebraFactory& factory,
    AssemblyMethod asm_method = AssemblyMethods::cellwise,
    DirichletProjectionMethod dbc_method =
        DirichletProjectionMethods::keep,
    BoostFlag boost_flags = BoostFlags::space_grouping,
    const vector<int>* global_to_reduced_dofs = NULL,
    const vector<uint>* reduced_to_global_dofs = NULL,
    const vector<int>* refined_global_to_reduced_dofs = NULL,
    const vector<uint>* refined_reduced_to_global_dofs = NULL);
```

The parameters have the following meaning:

- `space`: The space $V \subset \hat{V}$.
- `refined_space`: The refined space $\hat{V}$.
- `factory`: The `LinearAlgebraFactory` for creating the matrix P.
- `asm_method`: This parameter must be chosen according to the way the mesh was refined. This also affects performance of the assembly process. Can be one of:
  - `AssemblyMethods::uniform`: Assumes a uniform mesh refinement (i.e. every cell is split into `n` cells and `refined_mesh.cell(n*i+j)` is contained in `mesh.cell(i)` for `0 <= j < n` and all `i`)
  - `AssemblyMethods::cellwise`: Allows adaptive mesh refinement (e.g. using `cell_markers`), i.e. not every cell of the mesh is refined or cells are refined differently.
  - `AssemblyMethods::dofwise`: Not implemented (since computationally too expensive). An implementation should determine the dof coordinates by using `tabulate_dof_coordinates` and determine the corresponding `mesh.cell` for each dof separately.
- `dbc_method`: This parameter specifies the treatment of Dirichlet boundary conditions. Can be one of:
  - `DirichletProjectionMethods::remove`: Removes rows/columns which are set to -1 in the mappings `refined_global_to_reduced_dofs`/ `global_to_reduced_dofs`.
  - `DirichletProjectionMethods::zero`: Zeroes rows/columns which are set to -1 in the mappings `refined_global_to_reduced_dofs`/ `global_to_reduced_dofs`.
  - `DirichletProjectionMethods::keep`: Does not change the original prolongation operator. In this case, the mapping parameters are not used.

- `boost_flags`: These flags should affect the performance of the assembly process only. But if you encounter problems with the prolongation operator you should try to disable this feature. Can be a bit-wise combination of the following flags:
  - `BoostFlags::none`: No flags are set.
  - `BoostFlags::space_grouping`: Subspaces with the same signature will be grouped together (components of a vector space for example) and dofs will be evaluated on the first space only.
  - `BoostFlags::hashing`: Builds a hash map for the dof evaluation, which stores only a representative for a class of cell evaluations. This method requires, that the meshes consists only of a small set of cells, possessing translational equivalence (i.e $T_1 = T_2 + t$).
- `global_to_reduced_dofs`, `reduced_to_global_dofs`, `refined_global_to_reduced_dofs` and `refined_reduced_to_global_dofs`: These maps must be specified in the case `dbc_method` is not `DirichletProjectionMethods::keep`.

3.4 Smoothing/relaxation - The fmg::*SmoothingOperator classes

The relaxation methods, as described in section 2.4.1, are all implemented using the common interface `GenericSmoothingOperator`, which just allows the application of the relaxation method to a vector `x`

```
1 class GenericSmoothingOperator
2 {
3 public:
4   virtual void apply(GenericVector& x, const GenericVector& b,
5     bool& x_assumed_zero, uint reps) const = 0;
6 };
```

where `b` specifies the right hand side of the linear system. The Boolean flag `x_assumed_zero`, specifies if `x` should be assumed to be zero (i.e. the result will be computed as if `x=0`), and must be passed as reference to this function. The result of the relaxation is stored back to `x` and if `x` was changed (to non-zero) `x_assumed_zero` is set to `false` (which is generally done whenever the number of repetition `reps` is greater zero).

The actual implementations are accessible by the `SmoothingOperator` class, which is constructed using the matrix `A` of the linear system and specifying the desired relaxation method and an optional relaxation parameter `omega`.

```
class SmoothingOperator : public GenericSmoothingOperator
{
public:
  SmoothingOperator(const boost::shared_ptr<const GenericMatrix>& A,
    const std::string& method, double omega = 1.0);
  ...
};
```

The `method` string may be one of:

- `"jacobi"`: Jacobi relaxation sweep, as in eq. (2.114)
- `"fsor"`: forward SOR sweep, as in eq. (2.120)
- `"bsor"`: backward SOR sweep, as in eq. (2.122)
- `"ssor"`: symmetric SOR sweep, as in eq. (2.126)

The `SmoothingOperator` class acts only as a proxy to the actual implementations. The Jacobi relaxation is implemented within the `JacobiSmoothingOperator` class and the three SOR methods are implemented within the `SORSmoothingOperator`. While the implementation of the Jacobi relaxation was straight forward, the implementation of the `SORSmoothingOperator` relies (for performance reasons) on the `MatSOR` function of PETSc. Due to some shortcomings in this function (up to PETSc 3.3-p5), the pointwise SOR did not work for one repetition (but for greater one), hence we had to make a nasty (but functional) workaround, which is described in the `SORSmoothingOperator.cpp` in more detail.

3.5 Cycle patterns - The fmg::*CyclePattern classes

Instead of doing recursive calls within our multigrid algorithm, our implementation uses a directed cyclic graph like approach. That is restriction, prolongation and direct solving is done by cycling through a graph. For each node of the graph one of these actions is specified. Therefore, the `GenericCyclePattern` interface was specified:

```
class GenericCyclePattern
{
public:
  typedef struct { enum e { restrict, solve, prolongate, pr_transition
      }; } Actions;
  typedef Actions::e Action;

  virtual void reset() = 0;
  virtual void next() = 0;
  virtual Action action() const = 0;
  virtual uint action_index() const = 0;
  virtual uint cycle_index() const = 0;
  virtual uint level_index() const = 0;
};
```

Let us give some comments on each of the functions:

- `reset()`: Go to the starting node of the graph. Set the cycle index to zero.
- `next()`: Go to the next node of the graph.
- `action()`: Return the action of the current node for the transition to the next node.
- `action_index()`: Returns an zero based index to the current action. This index is zeroed at the starting node and is incremented with each call to `next()`.
- `cycle_index()`: Returns the zero based number of cycles performed (i.e. the number of times we went through the starting node).
- `level_index()`: Returns the current level ℓ of the node within the multigrid algorithm.

The `Actions` returned by `action()` have the following meaning:

- `restrict`: Smooth, compute defect and restrict defect to the coarser grid.
- `solve`: Direct solve on the current grid.
- `prolongate`: Prolongate the solution to the finer level, correct the error and post smooth on the finer level.
- `pr_transition`: This action describes the transition between a `prolongate` and a `solve` action, e.g. it does the transition from the current V-cycle to the next V-cycle.

For the support of V-, W-cycles and other cycles the `VectorCyclePattern` has been implemented. This class creates a graph, by specifying the number of iterations γ_ℓ on each level ℓ, as in algorithm 2.5.

```
1 class VectorCyclePattern : public GenericCyclePattern
2 {
3 public:
4   VectorCyclePattern(const vector<uint>& cycles_per_level);
5   VectorCyclePattern(const string& pattern, uint number_of_levels);
6   ...
7 };
```

Alternatively the cycle pattern may be created from a string of concatenated numbers of cycles per level, separated by any non-numeric characters, except for the following characters, which have a special meaning:

- `'-'` : Finishes the current pattern, concatenates the pattern to the previous and starts a new pattern.
- `'*'` : The current pattern is padded with 1-cycles such that the pattern has `number_of_levels` levels and do the same as in `'-'`.
- `'V'` : Substitute for `'*'`.

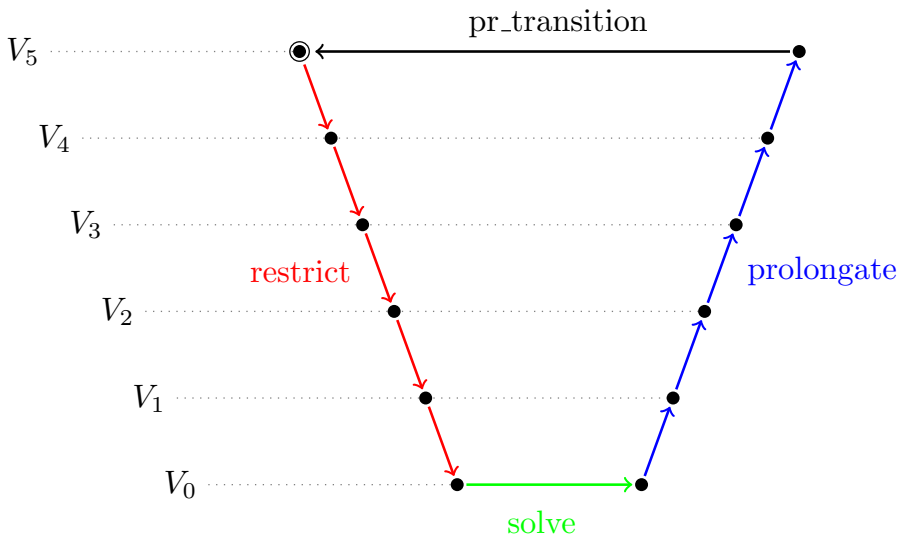


Figure 3.6: Graph for of the `VCyclePattern` for 6 levels.

For example the pattern ”2,1”, ”2*”, ”1*1*”, ”1 1-1*”, ”1e1-1x1”, ”VV”, ”**” and `number_of_levels=3` gives a 3-grid VV-cycle. As an further example the graph for the V-cycle is visualized in fig. 3.6, which may be created using `VectorCyclePattern("V", 6)` or the shortcut `VCyclePattern(6)`.

Some examples for the initialization, using the vector `cycles_per_level`, are given in table 3.1, where `<x1,x2,...>` denotes a `std::vector<uint>`, containing the integers `x1`, `x2`, etc.

Pattern	**Description**
<1>	direct solve only
<1,1>	2-grid V-cycle
<1,1,1>	3-grid V-cycle
<2,1,1>	3-grid VV-cycle
<1,2,1>	3-grid W-cycle
<2,2,1>	3-grid WW-cycle
<<1,1,1>, <1,2,1>>	3-grid VW-cycle
<<1,2,1,1>, <1,1,1,1>>	4-grid wV-cycle (w: 3-grid W-cycle)

Table 3.1: Example patterns used as argument for the `VectorCyclePattern` class.

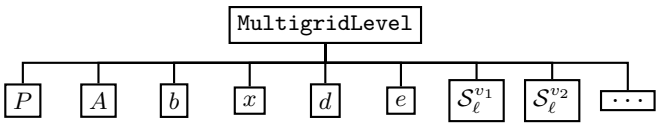


Figure 3.7: Schema of the `MultigridLevel` class.

3.6 Multigrid levels - The fmg::MultigridLevel class

The purpose of the `MultigridLevel` class is to manage all level (grid) dependent data and objects, used within the multigrid algorithm 2.5. This includes the matrices and vectors A, b for the linear system, information about boundary conditions, temporary vectors e, d, the solution x, the prolongation operator P, smoothers and direct solvers. Further it provides references to the coarser and finer level (kind of double linked list) and a number of convenient methods:

- `pre_smooth(reps)`, `post_smooth(reps)`:
  Apply pre-/post-smoother `reps` times to `x`,
- `calc_defect()`: Computes `d = Ax - b`,
- `restrict_to_coarser_level()`: Restricts defect to coarser level:
  `coarser_level->b = coarser_level->P`^{T}`d`,
- `prolongate_to_finer_level()`: Prolongate solution to finer level:
  `finer_level->e = Px`,
- `coarse_solve()`: Computes `x =` A^{-1}`b`,
- `correct_error()`: Computes `x = x - e`,

as well as a number of initialization methods for each of those objects.

3.7 Multigrid problems - The fmg::MultigridProblem class

The `MultigridProblem` class manages a list of `MultigridLevels` and a number of properties, which are shared by all levels. Further a `MultigridProblem` may be constructed from a `dolfin::LinearVariationalProblem` or from a `dolfin::NonlinearVariationalProblem`, which is used as coarse level problem. Additional refinement levels may be added simply by calling `adapt()` on the multigrid problem. Here are some of the methods:

- `MultigridProblem(LinearVariationalProblem& problem,`
  `uint nrefine = 0, bool symmetric = true, bool uniform = true,`
  `bool interpolate_coef = false, bool interpolate_solution = false)`

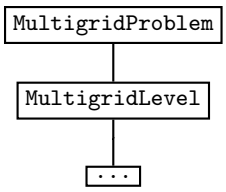


Figure 3.8: Schema of the `MultigridProblem` class.

- `MultigridProblem(NonlinearVariationalProblem& problem, uint nrefine = 0, bool symmetric = false, bool uniform = true, bool interpolate_coef = false, bool interpolate_solution = false)`
- `adapt()`, `adapt(cell_markers)`, `adapt(refined_mesh)`
- `is_linear()`, `is_symmetric()`, `is_uniform()`
- `finest_level()`, `level(i)`, `num_levels(i)`
- `solution()`

If `nrefine` is greater zero, the problem will be refined uniformly `nrefine` times. For non-symmetric problems the `symmetric` flag must be set to `false`. If the problem was refined non-uniformly in advance, you have to set `uniform` to `false`. The `interpolate_coef` and `interpolate_solution` parameters describe the interpolation behaviour of the refined form coefficients and solution variables (see `fmg::Adaptor` class). For more information we also refer to the `FMG` programmers-reference.

3.8 Multigrid solver - The fmg::MultigridSolver class

This class implements an easy to use multigrid solver for DOLFIN, using the algorithm as in 2.5. It is derived from `dolfin::GenericLinearSolver` and makes use of all previously described classes. However for performance reasons it is mostly recommended to use the `MultigridPreconditionedKrylovSolver`, as described in the next section, than using this solver directly. Three constructors are available for the `fmg::MultigridSolver` class:

```
1 MultigridSolver(MultigridProblem& problem);
2 MultigridSolver(dolfin::LinearVariationalProblem& problem,
3   uint nrefine = 0, bool symmetric = true, bool uniform = true,
4   bool interpolate_coef = false, bool interpolate_solution = false);
5 MultigridSolver(dolfin::NonlinearVariationalProblem& problem,
6   uint nrefine = 0, bool symmetric = false, bool uniform = true,
7   bool interpolate_coef = false, bool interpolate_solution = false);
```

which takes either a `MultigridProblem` as argument, or takes the arguments, which are required to implicitly construct a `MultigridProblem` from a `LinearVariationalProblem` or a `NonlinearVariationalProblem`.

The most essential part of the class is the `solve` method, performing the recursive multigrid algorithm, using the cycle pattern scheme.

```
uint MultigridSolver::solve(dolfin::GenericVector& xr,
  const dolfin::GenericVector& br, bool x_assumed_zero)
{
  MultigridLevel& flevel = finest_level();
  boost::shared_ptr<GenericCyclePattern> cycle_pattern;
  bool loop = true;

  flevel.x = &xr;
  flevel.b = &br;
  flevel.x_assumed_zero = x_assumed_zero;

  init_cylce_pattern(cycle_pattern);

  before_solve();

  do {
    uint ilevel = cycle_pattern->level_index();
    MultigridLevel& level = _problem->level(ilevel);

    before_cycle_action(level, *cycle_pattern);

    switch (cycle_pattern->action()) {
      case GenericCyclePattern::Actions::restrict:
        pre_smooth_level(level);
        calc_defect_level(level);
        if (ilevel == 0) loop = loop &&
          !test_residual_break_condition(level, *cycle_pattern);
        if (loop) restrict_to_coarser_level(level);
        break;
      case GenericCyclePattern::Actions::prolongate:
        prolongate_to_finer_level(level);
        correct_error_level(*(level.finer_level));
        post_smooth_level(*(level.finer_level));
        break;
      case GenericCyclePattern::Actions::solve:
        coarse_solve_level(level);
        if (ilevel == 0) loop = false;
        break;
      case GenericCyclePattern::Actions::pr_transition:
        pr_transition(level);
        break;
    }

    next_cycle_action(level, *cycle_pattern);
```

```
45
46      loop = loop && !test_loop_break_condition(*cycle_pattern);
47    }
48    while (loop);
49
50    after_solve();
51
52    return cycle_pattern->cycle_index();
53  }
```

Let us give some comments on the code:

- Line 4: Get the finest gird level.
- Line 8-10: Set the right hand side and solution of the fines gird level to the given arguments.
- Line 12: Initialize the cycle pattern (e.g. create a V-cycle pattern), depending on the configuration of the solver.
- Line 14: Call `before_solve` method, which may be re-implemented by the user to perform some user specific tasks. Currently this method does nothing.
- Line 17-18: Get the current multigrid level of the cycle pattern.
- Line 20: Call `before_cycle_action` method, which may be re-implemented by the user to perform some user specific tasks. Currently this method does nothing.
- Line 22: Decide which action to perform at the current node of the cycle graph, as described in section 3.5.
- Line 24: Calls `level->pre_smooth()`. Similarly the other `x_level(level)` methods just call `level->x()`.
- Line 26-27: Test break condition, using residuals on the finest level.
- Line 37: In the case we do a direct solve on the finest level, we break the loop.
- Line 40: Call `pr_transition` method, which may be re-implemented by the user to perform some user specific tasks. Currently this method does nothing.
- Line 44: Call `next_cycle_action` method, which may be re-implemented by the user to perform some user specific tasks. Currently this method just calls `cycle_pattern->next()`.
- Line 46: Test break condition, e.g. checking if maximum iteration count is exceeded.
- Line 50: Call `after_solve` method, which may be re-implemented by the user to perform some user specific tasks. Currently this method does nothing.
- Line 52: Return number of cycles (iterations).

The following listing gives an example for using the multigrid solver for a `LinearVariationalProblem`, which will be refined three times uniformly.

```
LinearVariationalProblem problem(a, L, u, bcs);
fmg::MultigridSolver mg_solver(problem, 3);

mg_solver.parameters["pre_smoother"] = "jacobi";
mg_solver.parameters["pre_smoother_relax"] = 2.0/3.0;
mg_solver.parameters["post_smoother"] = "jacobi";
mg_solver.parameters["post_smoother_relax"] = 2.0/3.0;
mg_solver.parameters["relative_tolerance"] = 1e-5;
mg_solver.parameters["absolute_tolerance"] = 1e-14;
mg_solver.solve();

plot(mg_solver.solution());
```

Here we also configure the solver to use a Jacobi smoother with $\omega = 2/3$ relaxation and different tolerances. Further configuration parameters are given in table 3.2 and table 3.2.

Similarly the solver may be constructed from an `fmg::MultigridProblem`:

```
LinearVariationalProblem problem(a, L, u, bcs);
fmg::MultigridSolver mg_problem(problem, 3);
fmg::MultigridSolver mg_solver(mg_problem);
```

or equivalently

```
LinearVariationalProblem problem(a, L, u, bcs);
fmg::MultigridSolver mg_problem(problem);
for (int i = 0; i < 3; i++)
  mg_problem.adapt();
fmg::MultigridSolver mg_solver(mg_problem);
```

In order to be able to change parameters after compilation, it is good practice to parse them form command line (see DOFLIN `Parameters` class), after setting some default parameters.

```
int main(int argc, char* argv[])
{
  ...
  mg_solver.parameters["relative_tolerance"] = 1e-5;
  mg_solver.parameters.parse(argc, argv);
  mg_solver.solve();
  ...
}
```

The program may then be invoked with parameters to alter the default settings:

```
./main --relative_tolerance 1e-6
```

Parameter	Default	Description
`pre_smoother`	”fsor”	Specifies the pre-smoother for each level. See `SmoothingOperator` for a list of available smoothers.
`pre_smoother_relax`	1.0	Specifies the pre-smoother relaxation factor for each level.
`pre_smoother_reps`	1	Specifies the number of pre-smoother repetitions for each level.
`post_smoother`	”bsor”	Specifies the post-smoother for each level. See `SmoothingOperator` for a list of available smoothers.
`post_smoother_relax`	1.0	Specifies the post-smoother relaxation factor for each level.
`post_smoother_reps`	1	Specifies the number of post-smoother repetitions for each level.
`cycle_pattern`	”V”	Specifies the cycle pattern. See `VectorCyclePattern` for valid pattern values.
`coarse_solver_type`	”lu”	Specifies the type of the coarse solver. Valid values are ”lu” and ”krylov”.
`lu_solver_method`	”petsc”	Specifies the name of the coarse solver when `coarse_solver_type` is set to ”lu”.
`lu_solver_parameters`	Parameters	Specifies the parameters of the coarse solver when `coarse_solver_type` is set to ”lu”. See DOLFIN manual for valid parameters.
`krylov_solver_method`	”cg”	Specifies the name of the Krylov solver when `coarse_solver_type` is set to ”krylov”. For non-symmetric problems the default value is ”gmres”.
`krylov_solver_preconditioner`	”none”	Specifies the name of the Krylov solver preconditioner when `coarse_solver_type` is set to ”krylov”.
`lu_krylov_parameters`	Parameters	Specifies the parameters of the Krylov solver when `coarse_solver_type` is set to ”krylov”. See DOLFIN manual for valid parameters.
`prolongation_assembly_method`	”uniform”	See `ProlongationAssembler`. For non-uniformly refined problems the default value is ”cellwise”.
`prolongation_assembly_eps`	1e-13	See `ProlongationAssembler`
`prolongation_assembly_space_grouping`	`true`	See `ProlongationAssembler`
`prolongation_assembly_hashing`	`false`	See `ProlongationAssembler`
`level_offset`	0	Specifies which level is considered as the finest level for which the problem is solved.

Table 3.2: Prarameters for the `MultigridSolver` class.

Parameter	Default	Description
`dirichlet_projection_method`	"identify"	If set to "remove", Dirichlet boundary conditions will be removed from the linear system. If set to "identify", the linear system is modified such that Dirichlet boundary conditions are identifies with the right hand side.
`dirichlet_dof_identification_method`	"topological"	Method of Dirichlet dof identification. Valid values are: "topological", "pointwise", "geometric". Refer to the DOLFIN documentation for more information.
`tolerance_checking`	`true`	If set to `true` the `absolute_tolerance` and `relative_tolerance` values are used as stopping criterion.
`absolute_tolerance`	1e-15	The absolute tolerance used as stopping criterion. If `tolerance_checking` is enabled the solver will stop if the current defect in the `defect_norm` norm is less than `absolute_tolerance`.
`relative_tolerance`	1e-6	The relative tolerance used as stopping criterion. If `tolerance_checking` is enabled the solver will stop if the current defect in the `defect_norm` norm is less than `relative_tolerance` times the initial defect.
`maximum_iterations`	10000	The maximum number of iterations used as failure criterion. If `error_on_nonconvergence` is enabled the solver will throw an exception if the number of iterations exceeds this value. If `error_on_nonconvergence` is not enabled the solver will return without an error.
`monitor_convergence`	`false`	Print residuals for each iteration.
`error_on_nonconvergence`	`true`	See `maximum_iterations` parameter.
`galerkin_lhs_assembly`	`true`	If set to `true` the lhs of the problems on the coarser levels are not assembled but computed from the lhs of the finest level.
`galerkin_rhs_assembly`	`true`	If set to `true` the rhs of the problems on the coarser levels are not assembled but computed from the rhs of the finest level.
`coarse_levels_rhs_assembly`	`false`	If set to `true` the rhs of the problems on the coarser levels are assembled. Usually the rhs on the coarser levels are not needed.
`keep_A0`	`false`	If set to `true` the `A0` matrix will be kept for each level.
`keep_P0`	`false`	If set to `true` the `P0` matrix will be kept for each level.

Table 3.2: Prarameters for the `MultigridSolver` class (continued).

3.9 Multigrid preconditioning - The fmg::MultigridPreconditioner class

The `fmg::MultigridPreconditioner` class is just a simple implementation of the `dolfin::PETScUserPreconditioner`, which uses a `fmg::MultigridSolver` as an approximate solver. Therefore, the constructor of the preconditioner takes a multigrid solver object, which is adjusted to work as a preconditioner. That is, convergence checking is disabled and the maximum iteration count is set to one:

```
MultigridPreconditioner(MultigridSolver& mg_solver)
{
  mg_solver.parameters["error_on_nonconvergence"] = false;
  mg_solver.parameters["tolerance_checking"] = false;
  mg_solver.parameters["maximum_iterations"] = 1;
  _mg_solver = reference_to_no_delete_pointer(mg_solver);
}
```

The preconditioner may be used for a PETSc Krylov solver in the following way:

```
MultigridPreconditioner prec(mg_solver);
PETScKrylovSolver solver("cg", prec);
```

The preconditioner is applied within the PETSc Krylov subspace method for a rhs vector `b`, by calling the `solve` method of the multigrid solver with zero initial `x` (i.e. `x_assumed_zero` set to `true`):

```
void MultigridPreconditioner::solve(dolfin::PETScVector& x,
  const dolfin::PETScVector& b)
{
  _mg_solver->solve(x, b, true);
}
```

3.10 Multigrid preconditioned Krylov solver - The fmg::MultigridPreconditionedKrylovSolver class

The `fmg::MultigridPreconditionedKrylovSolver` class derives from `PETScKrylovSolver` and is just a shortcut for doing constructions like:

```
MultigridSolver mg_solver(problem);
MultigridPreconditioner prec(mg_solver);
PETScKrylovSolver solver("cg", prec);
```

There are numerous constructors available, which allow either the construction from a given multigrid solver, a given multigrid problem or a given variational problem:

```
MultigridPreconditionedKrylovSolver(MultigridSolver& mg_solver,
  const std::string& method = "default");

MultigridPreconditionedKrylovSolver(MultigridProblem& problem,
  const dolfin::Parameters& mg_params = dolfin::empty_parameters,
  const std::string& method = "default");

MultigridPreconditionedKrylovSolver(LinearVariationalProblem& problem,
  uint nrefine = 0, bool symmetric = true, bool uniform = true,
  bool interpolate_coef = false, bool interpolate_solution = false,
  const dolfin::Parameters& mg_params = dolfin::empty_parameters,
  const std::string& method = "default");
```

Here the `method` string can be any valid Krylov method as for `PETScKrylovSolver`. The `"default"` method is `"cg"` for symmetric problems and `"gmres"` for non-symmetric problems.

3.11 Other classes

3.11.1 The fmg::FMGTimer class

This class is an extension of the `dolfin::Timer` class and is used for time measurements within the `FMG` classes. The following example measures the time needed to execute the code within the curly braces:

```
{
  FMGTimer t1("task1");

  // do something here
}

std::cout << FMGTimer::get_sum("task1") << std::endl;
```

If the code is executed several times, then `get_sum(task)` returns the total time, spent for that task. All completed `FMGTimer` tasks are also listed within `dolfin::list_timings()`, where the task name is prefixed with the string `"fmg:"`. Overall the class provides the following methods:

- `start()`, `stop()`,
- `reset_all()`, `reset(task)`, `have_task(task)`,
- `get_reps(task)`, `get_sum(task)`, `get_min(task)`, `get_max(task)`, `get_mean(task)`.

3.11.2 The fmg::Table class

This class is used for creating simple formatted tables, intended for console output, as well as for subsequent processing within other programs. As an example consider the following listing:

```
Table tab;

tab.add_int_column("r", "Rank");
tab.add_text_column("name", "Name", 6);
tab.add_float_column("time", "Time");
tab.finish_header();

tab["r"] << 1;
tab["name"] << "Tom";
tab["time"] << 10.2;
tab.finish_row();

tab["r"] << 2;
tab["name"] << "Jerry";
tab["time"] << 11.6;
tab.finish_row();
```

which will produce the following output:

```
Rank | Name   | Time
-----+--------+-----
1    | Tom    | 10.2
2    | Jerry  | 11.6
```

The `fmg::Table` also supports the output in other formats, which may be specified with the constructor `Table(TableFormat format)`.

- `TableFormats::console`: The default format, as shown in the example.
- `TableFormats::csv`: Uses comma separated values (CSV) format.
- `TableFormats::tab`: Separates the columns by a tab-stop.
- `TableFormats::latex`: Format as LaTeXtable.
- `TableFormats::rest`: Format as `reStructuredText` [Res] table.

3.11.3 The fmg::Utils class

This class contains some utility functions, for printing vectors and matrices, as well as a set of buffers

```
/// Returns a temporary vector of size >= n.
static std::vector<double>& temp(uint n);

/// Returns a zero vector of size >= n.
static const std::vector<double>& zero(uint n);

/// Returns the vector of integers 0 to size >= n.
static const std::vector<uint>& ints(uint n);
```

It also contains a global debug flag `fmg::Utils::DEBUG`, which may be used to enable debug mode for `FMG`.

3.12 Problem testing - The fmg::Tests class

This class provides easy to use methods for testing the `FMG` solver against a given problem. Typical usage is:

```
// ... create coarse bilinearform a, linearform L, solution u and
    boundary conditions bcs

LinearVariationalProblem problem(a, L, u, bcs);

// create tests instance
fmg::Tests tests(problem);

// set some parameters (optional)
tests.parameters("fmg_solver")["pre_smoother_relax"] = 0.6;
tests.parameters("fmg_solver")["post_smoother_relax"] = 0.6;
tests.parameters("fmg_solver")["pre_smoother"] = "jacobi";
tests.parameters("fmg_solver")["post_smoother"] = "jacobi";
tests.parameters["test_solver"] = "cg,cg+fmg";
tests.parameters["num_refinements"] = 3;
tests.parameters.parse(argc, argv);

// run the tests on problem
tests.run();
```

which tests a multigrid preconditioned CG solver with 3 refinement levels of the problem against unpreconditioned CG. Further the default smoother is changed to a relaxed Jacobi smoother. More configuration parameters can be found in table 3.3 and table 3.3. As an example, this class was also used to produce the tables, presented in chapter 4.

Parameter	Default	Description
`num_refinements`	1	Specifies the number of refinements for the problem.
`primary_solver`	`"cg"` for symmetric problems, `"gmres"` else	Specifies the primary Krylov solver for the problem. This should be set to "gmres" for non-symmetric problems for example.
`test_solver`	`""`	Specifies a (comma) separated list of solver(+preconditioner) pairs that should be tested for the problem, where "*" is replaced by the `primary_solver`. The `primary_solver+fmg` are added automatically to the end of this list. "fmg" may be used as direct solver and as preconditioner for any Krylov solver available within `dolfin::PETScKrylovSolver::methods()`. Valid direct solvers are given by `dolfin::PETScLUSolver::methods()`.
`test_coarse_solver`	`""`	Specifies a (comma) separated list of solver(+preconditioner) pairs that should be tested as coarse level solver for the problem, where "*" is replaced by the `primary_solver`. Valid solvers are given by `dolfin::PETScKrylovSolver::methods()` and `dolfin::PETScLUSolver::methods()`.
`test_smoothers`	`""`	Specifies a (comma) separated list of pre-smoother(@relaxation)(+post-smoother(@relaxation)) pairs that should be tested for the problem.
`test_cycle_patterns`	`""`	Specifies a (comma) separated list of cycle patterns that should be tested for the problem. See `VectorCyclePattern` for valid pattern values.
`max_smoothing_reps`	1	Specifies the maximum number of smoothing repetitions that should be tested.
`min_coarse_depth`	1	Specifies the minimum number of coarsenings, when testing different cycle depths.

Table 3.3: Prarameters for the `Tests` class.

Parameter	Default	Description
`write_solutions`	`false`	If set to `true` files of the solution for each refinement level are written to filenames given by `solution_file_format`.
`solution_file_format`	`"solution_%d.pvd"`	File format for `write_solutions` the `"%d"` is replaced by the mesh refinement level starting from zero.
`residual_count`	-1	Specifies the number of residuals to be listed in the residual table. If set to -1 the number will be equal to the number of iterations required for convergence.
`integration_time`	0.0	Specifies the minimum runtime of tests in order to improve accuracy by averaging the runtime.
`table_format`	`"console"`	Specifies the table output format for the results. Valid values are `"console"`, `"csv"`, `"tab"`, `"latex"` and `"rest"`.
`show_help`	`true`	If set to `true`, for each table additional help text is displayed.
`fmg_solver`	Parameters	Specifies the default parameters for the multigrid solver.

Table 3.3: Prarameters for the `Tests` class (continued).

4 Application and results

In this chapter we will present some numerical results, obtained with the implemented multigrid solver/preconditioner. All tests were performed in the following system environment:

- AMD Athlon 64 X2 5000+ 2.6GHz dual core processor (only one core used), 512KB cache, 8GB DDR2 RAM 800MHz
- Ubuntu 3.5.0-18-generic x86_64
- FEniCS 1.0.0+ (DOLFIN revision around 7013)
- PETSc 3.3p3

Further we used the following parameters for all tests:

- relevant FFC parameters: `-l dolfin -f split -O -f eliminate_zeros -f precompute_basis_const`
- relevant gcc parameters: `-g -O2 -fstack-protector --param=ssp-buffer-size=4 -frounding-math -fopenmp`
- convergence check: 2-norm of preconditioned residual less than 10^{-15} or 2-norm of preconditioned residual divided by 2-norm of rhs less than 10^{-6} (FEniCS default tolerances)

4.1 Poisson problem

As first example we will consider the Poisson problem again, as in section 3.1.3. This was: find $u \in H_D(\Omega) = \{v \in H^1(\Omega) : \text{trace}(v) = 0 \text{ on } \Gamma_D\}$ such that

$$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x = \int_\Omega f v \, \mathrm{d}x + \int_{\Gamma_N} g v \, \mathrm{d}s \quad \forall v \in H_D(\Omega), \tag{4.1}$$

where we had

$$\Omega = (0,1) \times (0,1) \tag{4.2}$$
$$\Gamma_D = \{(x,y) \in \partial\Omega : x = 0 \vee x = 1\} \tag{4.3}$$
$$\Gamma_N = \partial\Omega \setminus \Gamma_D \tag{4.4}$$
$$f(x,y) = 10\exp(-((x-0.5)^2 + (y-0.5)^2)/0.02) \tag{4.5}$$
$$g(x,y) = \sin(5x). \tag{4.6}$$

The UFL code and the `main.cpp` is the same as in section 3.1.3, except from using the `fmg::Tests` class instead of `dolfin::solve` this time:

```
int main(int argc, char* argv[])
{
  UnitSquare mesh(7, 7);
  Poisson::FunctionSpace V(mesh);
  ...

  LinearVariationalProblem problem(a, L, u, bc);

  fmg::Tests tests(problem);
  tests.parameters.parse(argc, argv);
  tests.run();

  return 0;
}
```

The source is located in `demo/poisson-unitsquare` of the `FMG` installation. We invoked the program using the following command line (`test.sh` script):

```
./main --num_refinements 8 --test_solver cg+hypre_amg,fmg,petsc,cg \
       --test_coarse_solver petsc,cg,cg+hypre_amg,mumps \
       --max_smoothing_reps 2 --table_format latex \
       --test_cycle_patterns V,2/V,1/2/V,1/1/2/V,1/1/1/2/V,1/1/1/1/2/V
           ,1/1/1/1/1/2/V,1/1/1/1/1/1/2/V \
       --residual_count 10 --integration_time 2 --min_coarse_depth 2 \
       --test_smoother jacobi@0.66,fsor+bsor,ssor | tee results.txt
```

After completion the `results.txt` file contains the results of the test run (additionally they are dumped to screen). Each of the results and the corresponding parameters will be discussed in detail on the following pages.

4.1.1 Initialization timings

The multigrid solver needs a certain time for the initialization of all grid levels. Therefore, we will first compare the the time needed for the initialization of the multigrid problem to initializing the problem directly on the finest level. Table 4.1 shows for each component of the initialization phase the additional time, required for the initialization of the multigrid problem. For the current problem the initialization time for 8 refinement levels is about 48% (27.5s) higher than initializing the problem directly on a `UnitSquare(1792,1792)`. If we do no refinements, we would actually expect to have no additional initialization time. However the instantiation of the `MultigridSolver`, `MultigridLevel` and related classes takes also a small amount of time, but as the system size increases this additional time plays a minor role in contrast to the system assembly time for example. That's why the relative `t_loss` tends to a constant value as the number of refinements increases. Further table 4.2 shows the detailed build-up for the 27.5s at 8 refinement levels.

ref.	dofs	t_direct	t_mg	t_loss	t_loss %
0	64	0.00196	0.00274	0.000775	39.5
1	225	0.00372	0.00722	0.00350	94.1
2	841	0.0117	0.0209	0.00924	79.3
3	3 249	0.0491	0.0764	0.0273	55.6
4	12 769	0.207	0.308	0.101	49.0
5	50 625	0.853	1.26	0.402	47.1
6	201 601	3.52	5.15	1.64	46.6
7	804 609	14.4	21.2	6.73	46.6
8	3 214 849	57.3	84.8	27.5	48.0

Table 4.1: Multigrid problem initialization timings compared to a direct problem initialization on the finest mesh.

4.1.2 Convergence behaviour

Table 4.3 and 4.4 shows the number of iterations and solve time versus the number of mesh refinements for all methods listed in the `-test_solver` parameter. As we see all solvers in table 4.3 show a constant iteration count (independently of the mesh size), which is characteristic for the multigrid methods. All listed `FMG` solver timings don't include the time, required for initialization of the `FMG` solver. You have to add the `t_loss` from the previous table to the timings for all solvers using `FMG` to have the total time, that is required when doing only a single solve. In this case, we would have to add 27.5s to the 4.81s of the `cg+fmg` method, which makes about 32.3s, which means for this problem the `cg+hypre_amg` combination (27.2s) is superior to the other methods. However we also see that the `cg+fmg` method is superior, when considering only the solve time. This could be advantageous, if the solver is used several times and not all components of the solver must be reinitialized (e.g. nonlinear problems). In table 4.4 we also listed a direct solver (PETSc LU) and an unpreconditioned CG as comparison. The relative residuals for 10 iterations (defined by the `residual_count` parameter) of the `cg+fmg` method are shown in table 4.5, measured in the norms:

- `l2`: 2-norm (ℓ_2) of the residual $\|r\|_2$
- `linf`: max-norm (ℓ_∞) of the residual $\|r\|_{\max}$
- `natural`: C^{-1} norm of the residual $\|r\|_{C^{-1}} = \sqrt{r^T C^{-1} r}$
- `prec_l2`: 2-norm of the preconditioned residual $\|C^{-1}r\|_2$
- `prec_linf`: max-norm of the preconditioned residual $\|C^{-1}r\|_{\max}$

Here C^{-1} denotes the multigrid preconditioner. The corresponding plot of the residuals is given in fig. 4.1.

name	**description**	**time**	**loss %**
t_mesh	direct problem mesh creation	1.46	
t_mesh_mg	multigrid adaptive mesh creation	3.82	
t_loss_mesh	loss by mesh creation	2.36	8.58
t_init	direct problem initialization (without t_mesh)	14.2	
t_init_mg	multigrid adaptive problem initialization (without t_mesh_mg)	18.7	
t_loss_init	loss by problem initialization	4.49	16.3
t_asm_bc	direct problem initialization of boundary values	28.9	
t_asm_bc_mg	multigrid problem initialization of boundary values	38.8	
t_loss_asm_bc	loss by initialization of boundary values	9.88	35.9
t_asm_lhs	direct problem assembly of lhs	7.51	
t_asm_lhs_mg	multigrid problem assembly of lhs	9.96	
t_loss_asm_lhs	loss by assembly of lhs	2.45	8.90
t_asm_rhs	direct problem assembly of rhs	5.31	
t_asm_rhs_mg	multigrid problem assembly of rhs	7.09	
t_loss_asm_rhs	loss by assembly of rhs	1.78	6.49
t_loss_pro	loss by prolongation operator creation	5.52	20.1
t_loss_sr	loss by system reduction	0.923	3.36
t_loss_other	other losses	0.0924	0.336
t_direct	total direct problem initialization (“ready to solve“)	57.3	
t_mg	total multigrid problem initialization (“ready to solve“)	84.8	
t_loss	total losses	27.5	100.

Table 4.2: Detailed problem initialization timings for the Poisson problem.

refinements	dofs	cg+hypre_amg iterations	cg+hypre_amg time	fmg iterations	fmg time	cg+fmg iter	cg+fmg time
0	64	3	0.000479	1	0.000246	1	0.000446
1	225	4	0.00151	9	0.00166	5	0.00150
2	841	4	0.00488	10	0.00347	5	0.00261
3	3 249	4	0.0191	10	0.00891	5	0.00649
4	12 769	4	0.0873	10	0.0298	5	0.0217
5	50 625	4	0.389	10	0.111	5	0.0797
6	201 601	4	1.55	10	0.423	5	0.307
7	804 609	4	6.32	10	1.69	5	1.20
8	3 214 849	4	27.2	10	7.10	5	4.81

Table 4.3: Iteration count and solve time for the Poisson problem.

refinements	dofs	petsc time	cg iter	cg time
0	64	0.000189	23	0.000122
1	225	0.000581	44	0.000405
2	841	0.00297	87	0.00318
3	3 249	0.0194	172	0.0190
4	12 769	0.145	343	0.235
5	50 625	1.18	685	2.18
6	201 601	9.17	1 368	17.8
7	804 609	-	2 736	141.
8	3 214 849	-	5 471	1130

Table 4.4: Iteration count and solve time for the PETSc LU solver and unpreconditioned CG.

iter	l2	linf	natural	prec_l2	prec_linf
0	1.00	1.00	1.00	1.00	1.00
1	0.154	0.255	0.0698	0.0431	0.0575
2	0.0179	0.0250	0.00536	0.00273	0.00361
3	0.00232	0.00323	0.000438	0.000168	0.000304
4	0.000308	0.000219	3.04e-05	8.26e-06	1.19e-05
5	3.18e-05	3.33e-05	1.98e-06	4.03e-07	7.14e-07
6	3.28e-06	3.11e-06	1.71e-07	3.86e-08	6.84e-08
7	2.32e-07	4.34e-07	1.58e-08	3.09e-09	6.28e-09
8	1.63e-08	3.69e-08	1.18e-09	2.14e-10	4.80e-10
9	1.40e-09	3.53e-09	8.37e-11	2.32e-11	3.45e-11
10	1.24e-10	3.30e-10	6.38e-12	2.43e-12	3.33e-12

Table 4.5: Relative residuals in different norms for the `cg+fmg` method.

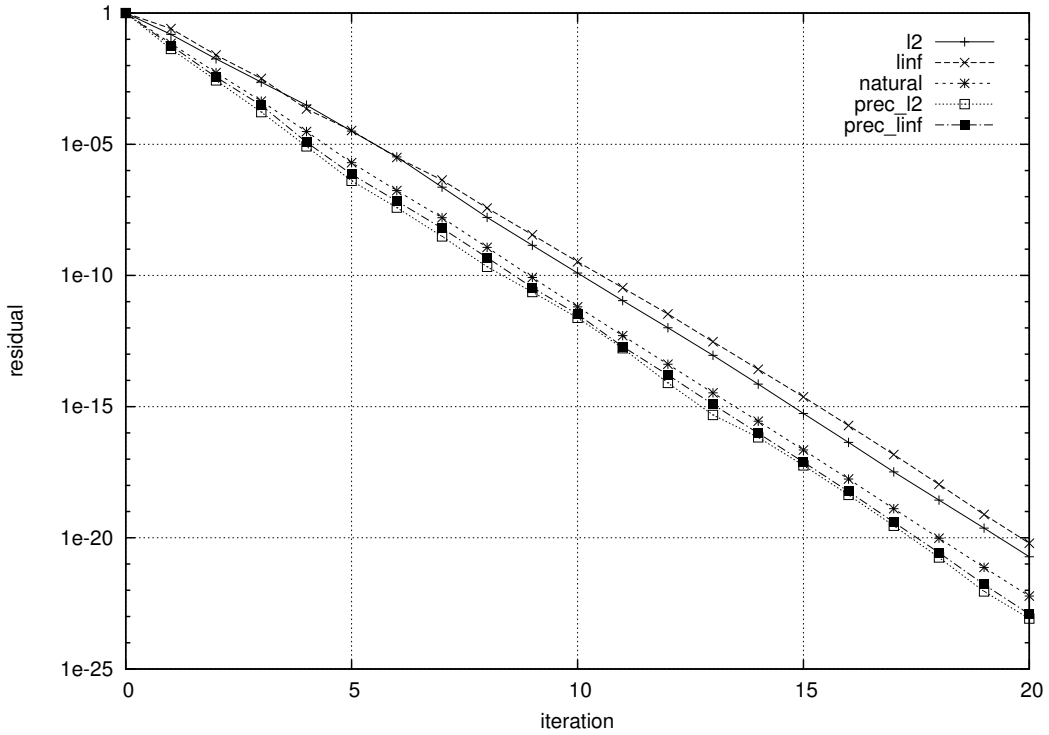


Figure 4.1: Plot of the residuals (table 4.5) in different norms for the `cg+fmg` method.

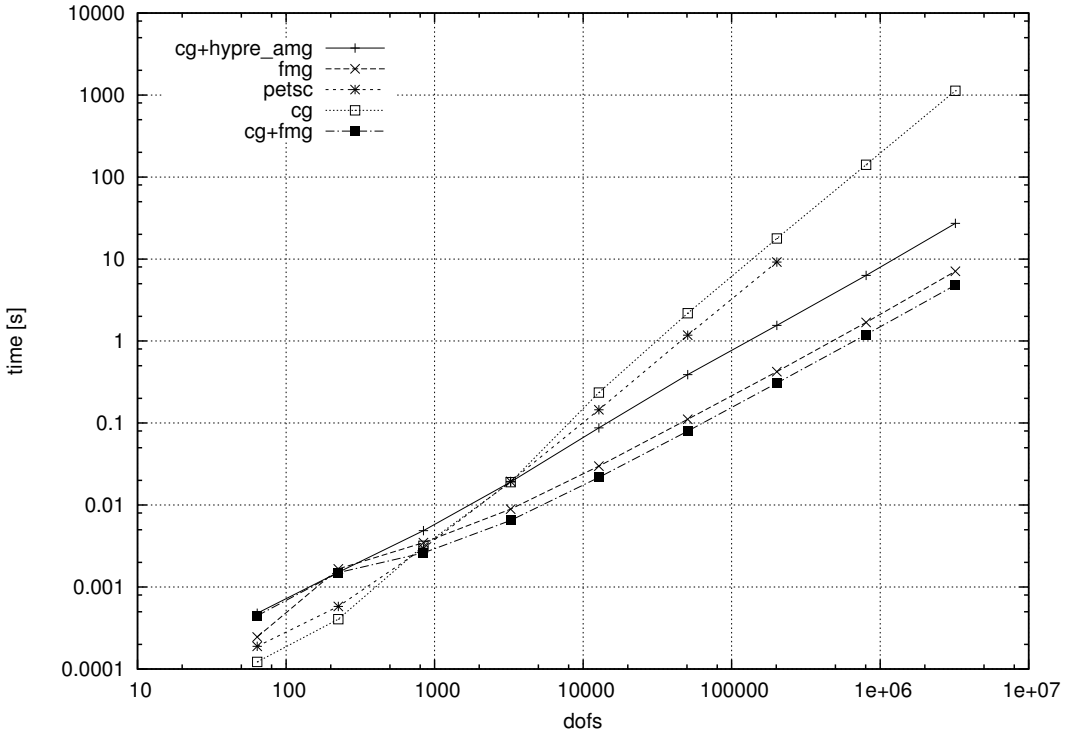


Figure 4.2: Plot of the solve time (table 4.4) for different methods.

A plot of the solve times for each solver is shown in fig. 4.2. As we can see the slopes of the lines are approximately 1 for the multigrid methods and 3/2 for the other methods. This may be expressed in terms of an equation as

$$\frac{\log(t_2) - \log(t_1)}{\log(n_2) - \log(n_1)} = c. \tag{4.7}$$

In terms of an ordinary differential equation, the solution may be described as $\log(t(n)) = c\log(n) + d$, which gives the following equation for the time:

$$t(n) = \exp(c\log(n) + d) = \exp(d)n^c. \tag{4.8}$$

This is equivalent to having a computational complexity of $\mathcal{O}(n^c)$, which is in accordance with the theory (for the 2d Possion problem), predicting a complexity of $\mathcal{O}(n)$ for multigrid methods and $\mathcal{O}(n^{3/2})$ for CG and sparse LU ([TS01], table 1.1).

smoother	reps	fmg iter	fmg time	cg+fmg iter	cg+fmg time
jacobi@0.66	1	17	11.7	7	5.76
jacobi@0.66	2	10	12.0	6	8.04
fsor+bsor	1	10	6.69	5	4.81
fsor+bsor	2	5	9.34	4	9.18
ssor	1	7	7.54	4	5.57
ssor	2	4	13.5	3	12.5

Table 4.6: Effect of different smoothers.

4.1.3 Different smoothers

Table 4.6 shows the number of iterations and solve time for different smoother methods (as listed in the `test_smoothers` parameter up to `max_smoothing_reps` repetitions) for the `cg+fmg` solver on the finest level. The `fsor+bsor` (foward-backward SOR) smoother with one repetition shows the default behaviour as in table 4.3. We also observe that this smoother works best for this problem.

4.1.4 Different cycle patterns

Table 4.7 shows the number of iterations and solve time for different cycle patterns (as listed in the `test_cycle_patterns` parameter) for the `cg+fmg` solver on the finest level. The meaning of the different cycle patterns is shown in fig. 4.3. In this case, we may say that the cycle pattern `1/1/2/V` works best for the `cg+fmg` solver, however if changing the convergence tolerances this may change. For the `fmg` solver we observe, that the `2/V` cylce halves the number of iterations, which makes sense, since the `2/V`-cycle is equivalent to doing 2 iterations on a single `V`-cycle.

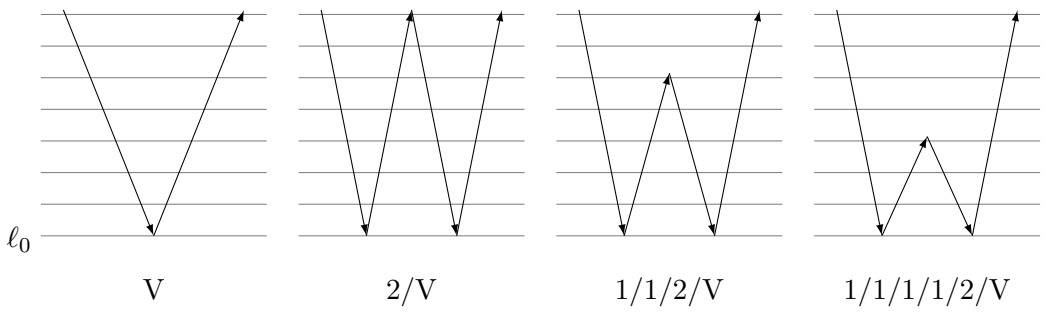


Figure 4.3: Different test cycle patterns.

pattern	**fmg iter**	**fmg time**	**cg+fmg iter**	**cg+fmg time**
V	10	6.80	5	5.13
2/V	5	6.74	4	7.07
1/2/V	8	6.59	4	4.71
1/1/2/V	9	6.33	4	4.12
1/1/1/2/V	9	6.06	5	4.83
1/1/1/1/2/V	9	5.98	5	4.76
1/1/1/1/1/2/V	10	6.61	5	4.74
1/1/1/1/1/1/2/V	10	6.59	5	4.74

Table 4.7: Effect of different cycle patterns.

coarse solver	fmg iter	fmg time	cg+fmg iter	cg+fmg time
petsc	10	6.75	5	4.80
cg	10	6.80	5	4.77
cg+hypre_amg	10	6.74	5	4.82
mumps	10	7.14	5	5.04

Table 4.8: Effect of different coarse level solver.

4.1.5 Different coarse grid sizes

Table 4.9 shows the number of iterations and solve time for different cycle depths (the number of coarsenings relative to the finest level, see fig. 4.4) for the `cg+fmg` solver on the finest level. The minimum number of these "virtual" coarsenings is given by the `min_coarse_depth` parameter. Note that a zero value of `min_coarse_depth` means that the you directly solve on the finest mesh. Because for a direct solver, like `petsc`, this requires too much memory, which is the reason why the first two depths were omitted. As we see, using very coarse grids in combination with more refinements does not change the solve time. In this case, it would be sufficient to use a 57x57 grid (3249 dofs) with only 5 refinements to have the same solve time.

4.1.6 Different coarse level solvers

Table 4.8 shows the number of iterations and solve time for different solver for the coarse level (as listed in the `test_coarse_solver` parameter) for the `cg+fmg` solver on the finest level. As the coarse level problem consists only of 64 unknowns, the influence to the overall time is rather minor.

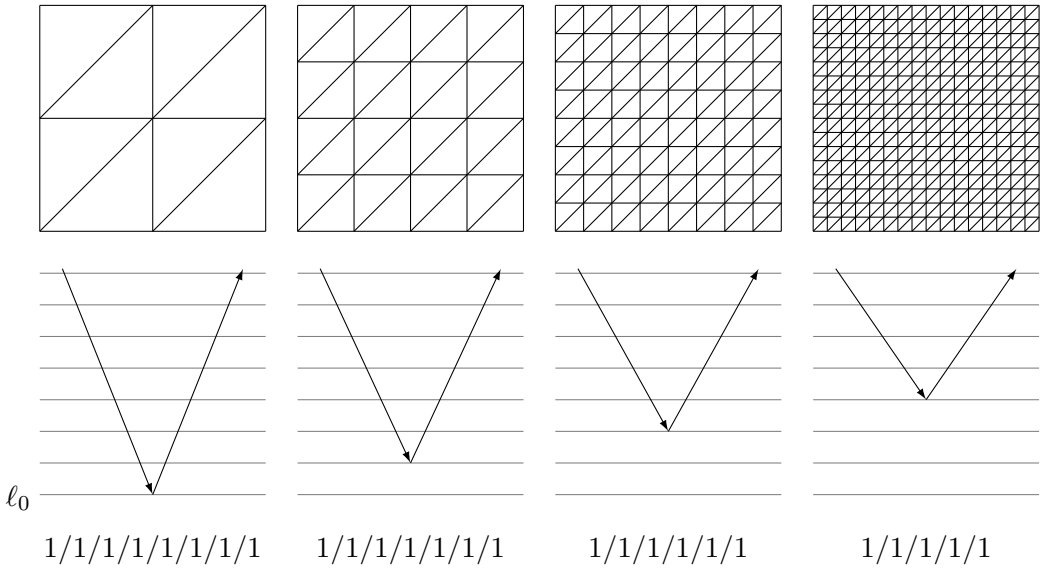


Figure 4.4: Different coarse grid sizes and corresponding cycle patterns.

depth	pattern	coarse dofs	fmg iter	fmg time	cg+fmg iter	cg+fmg time
2	1/1/1	201 601	9	15.8	4	13.7
3	1/1/1/1	50 625	9	7.21	4	5.16
4	1/1/1/1/1	12 769	9	6.15	5	4.88
5	1/1/1/1/1/1	3 249	10	6.61	5	4.76
6	1/1/1/1/1/1/1	841	10	6.58	5	4.74
7	1/1/1/1/1/1/1/1	225	10	6.59	5	4.77
8	1/1/1/1/1/1/1/1/1	64	10	6.61	5	4.75

Table 4.9: Effect of different coarse grid sizes.

operation	reps	max time	total	time/rep	time/rep %
calc defect	56	0.123	1.12	0.0199	25.5
pre smooth	56	0.139	1.15	0.0206	26.4
post smooth	56	0.136	1.22	0.0218	27.9
restrict	56	0.0397	0.371	0.00663	8.49
prolongate	56	0.0400	0.365	0.00651	8.34
correct error	56	0.0161	0.143	0.00256	3.28
coarse solve	7	9.11e-05	0.000426	6.08e-05	0.0779
test break	7	9.06e-06	5.39e-05	7.70e-06	0.00986

Table 4.10: Execution timings for 7 cycles (4.4s) of the multigrid algorithm.

4.1.7 Execution timings

Table 4.10 shows the execution time of different operations for 7 cycles (4.4s) of the multigrid algorithm (`fmg` solver) on the finest level, using a nonzero initial guess. As we see the defect calculation, pre- and post smoothing are the most expensive operations, as they all involve vector multiplication with the system matrix. Restriction and prolongation operators also involve matrix vector multiplication, but usually the prolongation matrix is much sparser than the system matrix. The error correction step only involves vector addition. As we already mentioned earlier, the solve time of the coarse solver does not have much influence on the overall solve time for this example.

4.1.8 Comparison to FEINS

FEINS [Schb] is a finite element solver, developed during the PhD thesis of Rene Schneider [Scha] at the University of Leeds, and was written entirely in C. It originated as a finite element incompressible Navier-Stokes solver (FEINS), but evolved over time to solve also other problems. As it also includes an example for solving Poisson's equation on a unit square, using a V-cycle multigrid preconditioned CG, we'd like to give here the results for the computation of the Poisson example as comparison. However, it should be noted that FEINS uses different approaches re-

garding the storage of matrices and hence is less memory exhaustive than FEniCS. Also the problem we tested, has different boundary conditions than the problem we considered in section 3.1.3, but this should not have a significant influence on the computation time. The tests were performed using the SVN snapshot revision 2579 of FEINS, with minor modifications to the `src/test_asem.c` file and the configuration file `meshfiles/poisson/square_cmess_tests.f1m`, to adjust the tolerances and stopping criterion, used throughout this chapter (see appendix A.1 for details). FEINS was compiled, using the following command within the `build` directory:

```
cmake ../src/ -DUSE_OPENMP=OFF -DCMAKE_BUILD_TYPE=Release
```

The problem being solved is: find $u \in H_D(\Omega) = \{v \in H^1(\Omega) : \text{trace}(v) = u_0 \text{ on } \partial\Omega\}$ on the unit square $\Omega = (0,1)^2$ such that

$$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x = 0 \quad \forall v \in H_D(\Omega), \tag{4.9}$$

where the boundary values are given by

$$u_0(x,y) = \begin{cases} 1 - 4(y-0.5)^2 & \text{for } x = 1 \\ 0 & \text{else.} \end{cases} \tag{4.10}$$

The program was executed by

```
./build/test_assem meshfiles/poisson/square_cmess_tests.f1m
```

and returned the results as given in fig. 4.5 and table 4.11. Here `iterations` and `time_cg` denote the number of required iterations and the solve time of the PCG respectively, `time_asm` denotes the time required to assemble the system matrix on the current level.

From the previous results we can say that the computation time per dof and iteration for the `cg+fmg` solver is in the range of $4.8s/(3214849 * 5) \approx 3.0 \cdot 10^{-7}s$, while for FEINS we have a computation time per dof and iteration in the range of $15.7s/(4198401 * 7) \approx 5.3 \cdot 10^{-7}s$. This difference may be explained due to the more efficient linear algebra implementation of PETSc. For the mesh refinement, matrix assembly and multigrid initialization on all levels, we have for `cg+fmg` a time per dof of $84.8s/3214849 \approx 2.6 \cdot 10^{-5}s$ (includes everything, for all levels, except solve time). For FEINS, an equivalent value would be to take the total time, subtract all the solve times and divide by the number of dofs, which makes $(109.4s - 15.74s - 3.916s - 0.945s - ...)/4198401 \approx 2.1 \cdot 10^{-5}s$.

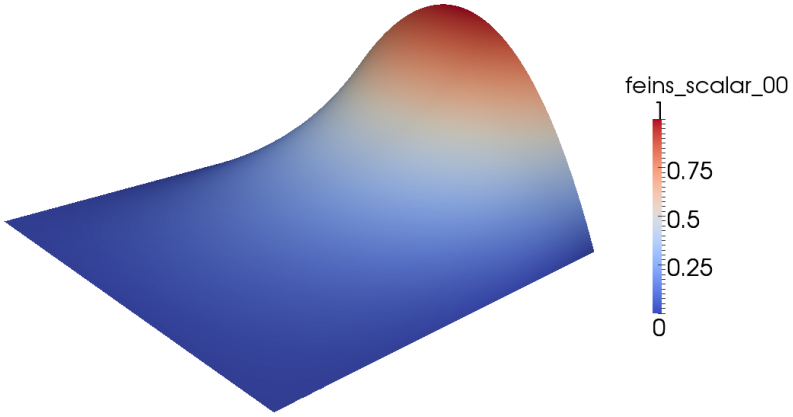


Figure 4.5: Plot of the solution for problem 4.9.

nrefine	iterations	|res|	vx_nr	u_0	time_asm	time_cg	time_total
0	1	0.00e-00	9	0.250000000	0.0001590	0.00001907	0.0001862
1	6	2.29e-07	25	0.218750091	0.0002539	0.00004005	0.0006001
2	8	3.98e-07	81	0.208869381	0.0009091	0.0001330	0.001761
3	7	1.71e-06	289	0.206217083	0.01026	0.0004351	0.01269
4	7	2.39e-06	1 089	0.205541121	0.04645	0.01395	0.07387
5	7	4.32e-06	4 225	0.205371316	0.1357	0.02255	0.2359
6	7	7.88e-06	16 641	0.205328807	0.2944	0.04865	0.5968
7	7	1.59e-05	66 049	0.205318177	1.004	0.2238	1.889
8	7	3.33e-05	263 169	0.205315521	3.929	0.9450	6.908
9	7	6.97e-05	1 050 625	0.205314858	16.02	3.916	27.40
10	7	1.44e-04	4 198 401	0.205314693	64.17	15.74	109.4

Table 4.11: Obtained convergence results for problem 4.9, using FEINS.

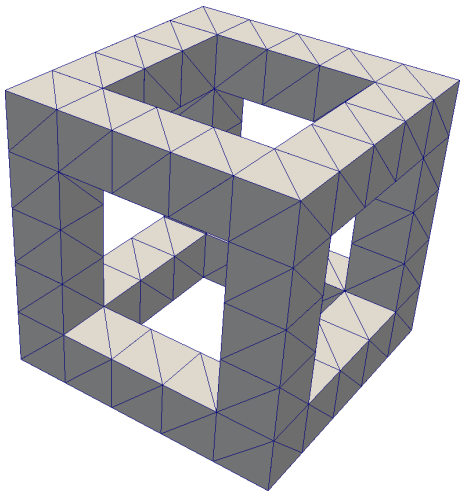


Figure 4.6: Domain Ω and initial mesh for the elasticity problem.

4.2 Linear elasticity

In this section we discuss the results for the following problem: find the displacement field $u : \Omega \mapsto \mathbb{R}^3$ such that

$$\operatorname{div}(\sigma(u)) + f = 0 \quad \text{in } \Omega, \tag{4.11}$$

$$\sigma(u) \cdot n = 0 \quad \text{on } \partial\Omega, \tag{4.12}$$

$$u = 0 \quad \text{on } \Gamma_D, \tag{4.13}$$

where $\sigma : \Omega \mapsto \mathbb{R}^3$ is the symmetric Cauchy stress tensor, $f : \Omega \mapsto \mathbb{R}^3$ is a volume force, $g : \Omega \mapsto \mathbb{R}^3$ is a prescribed boundary traction and $u_0 : \Omega \mapsto \mathbb{R}^3$ is a prescribed boundary displacement. The domain and Dirichlet boundary for our example (see. fig. 4.6) are given by

$$\Omega = a^3 \setminus (a \times b^2 \cup b \times a \times b \cup b^2 \times a), \tag{4.14}$$

$$\text{with } a = (0, 1) \text{ and } b = (1/6, 5/6), \tag{4.15}$$

$$\Gamma_D = \{(x, y, z) \in \partial\Omega : x = 0\}, \tag{4.16}$$

and the volume force is

$$f(x, y, z) = \vec{e_z} \begin{cases} -10 \exp(2z + y) & : x \geq 5/6 \\ 0 & : \text{else.} \end{cases} \tag{4.17}$$

Further we use the constitutive model of linearized elasticity for an isotropic, homogeneous material

$$\sigma(u) = 2\mu\epsilon(u) + \lambda \operatorname{tr}(\epsilon(u))\, I, \tag{4.18}$$

where

$$\epsilon(u) = \frac{1}{2}\left(\mathrm{grad}(u) + \mathrm{grad}(u)^T\right) \tag{4.19}$$

is the strain tensor and λ and μ are the Lamé parameters, which are related to the Young's modulus E and Poisson's ratio ν of the material by

$$\mu = E\frac{1}{2(1+\nu)} \quad \text{and} \quad \lambda = E\frac{\nu}{(1+\nu)(1-2\nu)}. \tag{4.20}$$

In our example we use $E = 210000$ (without units as before) and $\nu = 0.3$.

The weak formulation of the problem is: find $u \in H_D(\Omega) = \{v \in (H^1(\Omega))^2 : \mathrm{trace}(v) = 0 \text{ on } \Gamma_D\}$ such that

$$\int_\Omega \sigma(u) : \epsilon(v)\,\mathrm{d}x = \int_\Omega f v\,\mathrm{d}x \quad \forall v \in H_D(\Omega). \tag{4.21}$$

The associated UFL code is given by:

```
cell    = tetrahedron
element = VectorElement("Lagrange", cell, 2)

u = TrialFunction(element)
v = TestFunction(element)
f = Coefficient(element)

mu    = Constant(cell)
lmbda = Constant(cell)

def epsilon(v):
  return 0.5*(grad(v) + grad(v).T)

def sigma(v):
  return 2.0*mu*epsilon(v) + lmbda*tr(epsilon(v))*Identity(v.cell().d)

a = inner(sigma(u), epsilon(v))*dx
L = inner(f, v)*dx
```

Note that we typed `lmbda` instead of `lambda`, which is a Python keyword for anonymous functions. This time we used $\mathcal{P}_2$ instead of $\mathcal{P}_1$ elements.

The source is located in `demo/elasticity-open-cube` of the `FMG` installation.

We invoked the program using the following command line (`test.sh` script):

```
./main --num_refinements 2 --test_solver cg+hypre_amg,fmg,petsc,cg
       --test_coarse_solver petsc,cg,cg+hypre_amg,mumps
       --integration_time 2.0 --table_format latex
       --test_cycle_patterns V,2/V,1/2/V,1/1/2/V
       --max_smoothing_reps 2 --min_coarse_depth 1
       --test_smoother fsor+bsor,ssor --residual_count 10 | tee
          results.txt
```

For the 3rd refinement we changed the parameters `--num_refinements 3 --test_solver fmg --min_coarse_depth 2`, since the other solvers are too memory exhaustive in this case. A plot of the solution after each refinement is given in fig. 4.7.

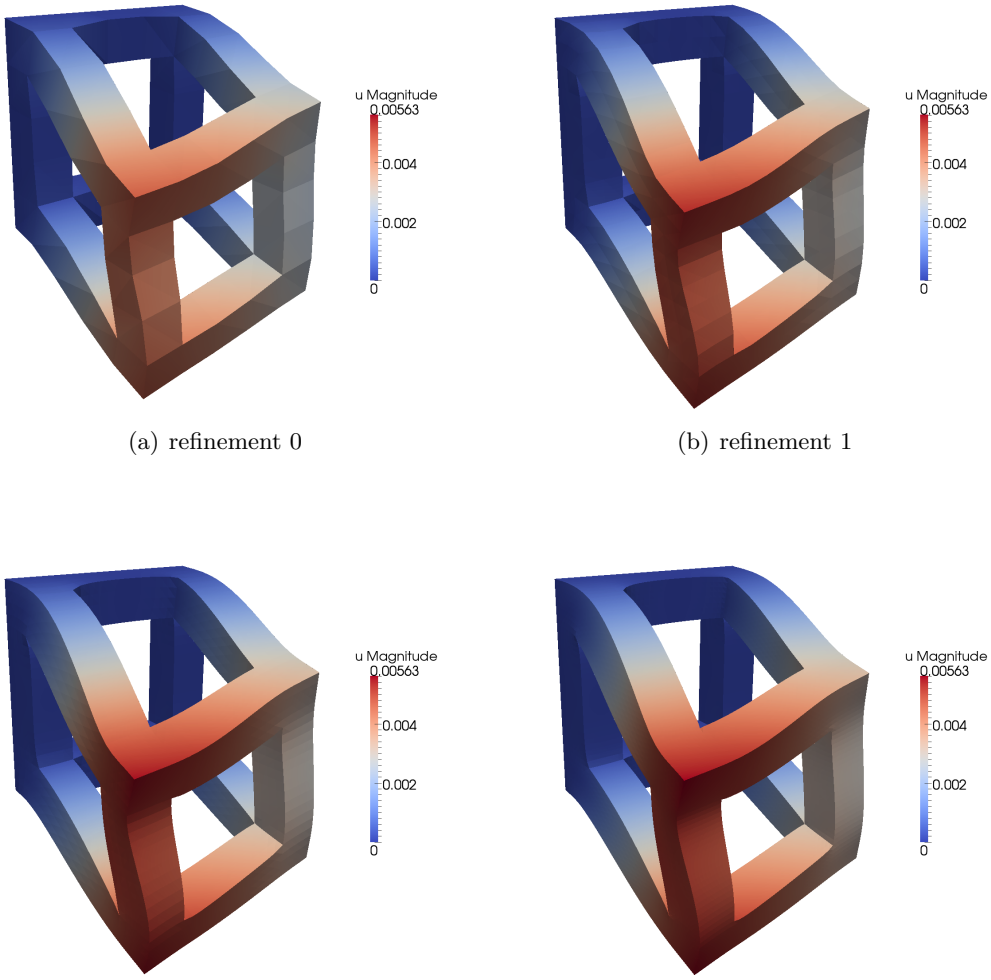
After completion the `results.txt` file contains the results of the test run. Each of the results and the corresponding parameters will be discussed in detail on the following pages.

4.2.1 Initialization timings

The multigrid solver needs a certain time for the initialization of all grid levels. Therefore, we will first compare the the time needed for the initialization of the multigrid problem to initializing the problem directly on the finest level. Table 4.12 shows for each component of the initialization phase the additional time, required for the initialization of the multigrid problem. For the current problem the initialization time for 3 refinement levels is about 20% (8.95s) higher than initializing the problem directly on the fine mesh. If we do no refinements, we would actually expect to have no additional initialization time. However the instantiation of the `MultigridSolver`, `MultigridLevel` and related classes takes also a small amount of time, but as the system size increases this additional time plays a minor role in contrast to the system assembly time for example. That's why the relative `t_loss` tends to a constant value

ref.	dofs	t_direct	t_mg	t_loss	t_loss %
0	2 268	0.0439	0.0470	0.00313	7.14
1	12 900	0.673	0.767	0.0943	14.0
2	84 564	5.43	6.51	1.08	19.9
3	606 900	44.3	53.3	8.95	20.2

Table 4.12: Multigrid problem initialization timings compared to a direct problem initialization on the finest mesh.



(a) refinement 0

(b) refinement 1

(c) refinement 2

(d) refinement 3

Figure 4.7: Solution of the elasticity problem after 0, 1, 2 and 3 refinements.

as the number of refinements increases. Further table 4.13 shows the detailed build-up for the 8.95s at 3 refinement levels.

4.2.2 Convergence behaviour

Table 4.14 and 4.15 shows the number of iterations and solve time versus the number of mesh refinements for all methods listed in the `-test_solver` parameter. As we see in table 4.14, this time only the `cg+fmg` solver shows a constant iteration count (independently of the mesh size). Also note again that all listed `FMG` solver timings don't include the time, required for initialization of the `FMG` solver. You have to add the `t_loss` from the previous table to the timings for all solvers using `FMG` to have the total time, that is required when doing only a single solve. In this case, we would have to add 8.95s to the 4.81s of the `cg+fmg` method, which makes about 13.8s. This means for this problem the `cg+fmg` combination is superior to the other methods. In table 4.15 we also listed a direct solver (PETSc LU) and an unpreconditioned CG as comparison. It should also be noted, that no other solver+preconditioner combination, available within FEniCS 1.0, works essential better than a pure `cg` [Osp12]. This means, using the `fmg` preconditioner is a great improvement.

The relative residuals for 10 iterations (defined by the `residual_count` parameter) of the `cg+fmg` method are shown in table 4.16, measured in the following norms:

- `l2`: 2-norm (ℓ_2) of the residual $\|r\|_2$
- `linf`: max-norm (ℓ_∞) of the residual $\|r\|_{\max}$
- `natural`: C^{-1} norm of the residual $\|r\|_{C^{-1}} = \sqrt{r^T C^{-1} r}$
- `prec_l2`: 2-norm of the preconditioned residual $\|C^{-1}r\|_2$
- `prec_linf`: max-norm of the preconditioned residual $\|C^{-1}r\|_{\max}$

Here C^{-1} denotes the multigrid preconditioner. The corresponding plot of the residuals is given in fig. 4.8.

name	**description**	**time**	**loss %**
t_mesh	direct problem mesh creation	0.0286	
t_mesh_mg	multigrid adaptive mesh creation	0.0658	
t_loss_mesh	loss by mesh creation	0.0372	0.416
t_init	direct problem initialization (without t_mesh)	19.3	
t_init_mg	multigrid adaptive problem initialization (without t_mesh_mg)	22.0	
t_loss_init	loss by problem initialization	2.72	30.4
t_asm_bc	direct problem initialization of boundary values	1.22	
t_asm_bc_mg	multigrid problem initialization of boundary values	1.35	
t_loss_asm_bc	loss by initialization of boundary values	0.134	1.50
t_asm_lhs	direct problem assembly of lhs	23.4	
t_asm_lhs_mg	multigrid problem assembly of lhs	26.7	
t_loss_asm_lhs	loss by assembly of lhs	3.24	36.2
t_asm_rhs	direct problem assembly of rhs	0.324	
t_asm_rhs_mg	multigrid problem assembly of rhs	0.370	
t_loss_asm_rhs	loss by assembly of rhs	0.0467	0.522
t_loss_pro	loss by prolongation operator creation	1.88	21.1
t_loss_sr	loss by system reduction	0.847	9.47
t_loss_other	other losses	0.0396	0.443
t_direct	total direct problem initialization (“ready to solve“)	44.3	
t_mg	total multigrid problem initialization (“ready to solve“)	53.3	
t_loss	total losses	8.95	100.

Table 4.13: Detailed problem initialization timings for 3 refinement levels.

refinements	dofs	cg+hypre_amg iterations	cg+hypre_amg time	fmg iterations	fmg time	cg+fmg iter	cg+fmg time
0	2 268	65	0.809	1	0.0499	1	0.0569
1	12 900	113	18.7	48	0.767	9	0.208
2	84 564	218	397.	55	5.23	9	1.14
3	606 900	-	-	63	45.5	9	8.34

Table 4.14: Iteration count and solve time for multigrid methods applied to the elasticity problem.

refinements	dofs	petsc time	cg iter	cg time
0	2 268	0.0499	626	0.225
1	12 900	1.51	1 384	4.17
2	84 564	83.3	3 032	64.8
3	606 900	-	-	-

Table 4.15: Iteration count and solve time for the PETSc LU solver and unpreconditioned CG.

iter	l2	linf	natural	prec_l2	prec_linf
0	1.00	1.00	1.00	1.00	1.00
1	21.7	45.8	0.165	0.0105	0.0114
2	11.4	39.4	0.0502	0.00256	0.00312
3	4.80	14.1	0.0171	0.00101	0.000930
4	1.85	7.35	0.00583	0.000272	0.000351
5	0.701	2.32	0.00208	0.000105	0.000117
6	0.262	0.729	0.000734	3.32e-05	3.75e-05
7	0.0941	0.256	0.000250	1.11e-05	1.25e-05
8	0.0335	0.0866	8.55e-05	3.17e-06	4.01e-06
9	0.0121	0.0456	2.99e-05	9.43e-07	1.18e-06
10	0.00458	0.0267	1.09e-05	3.19e-07	3.79e-07

Table 4.16: Relative residuals in different norms for the `cg+fmg` method.

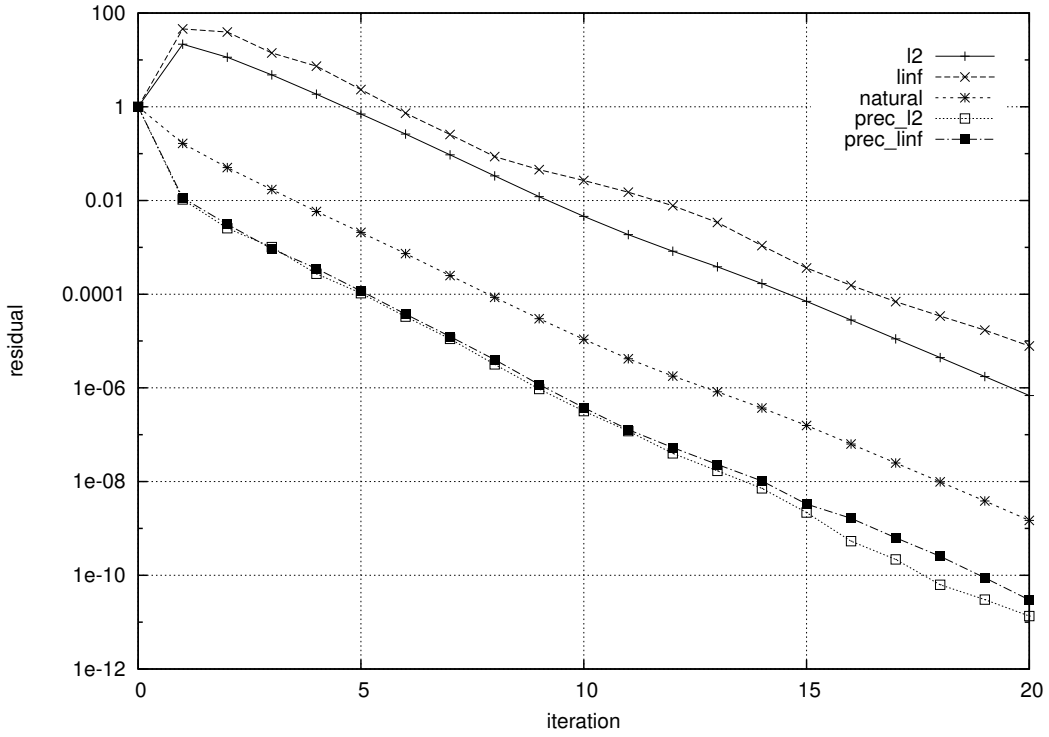


Figure 4.8: Plot of the residuals (table 4.16) in different norms for the `cg+fmg` method.

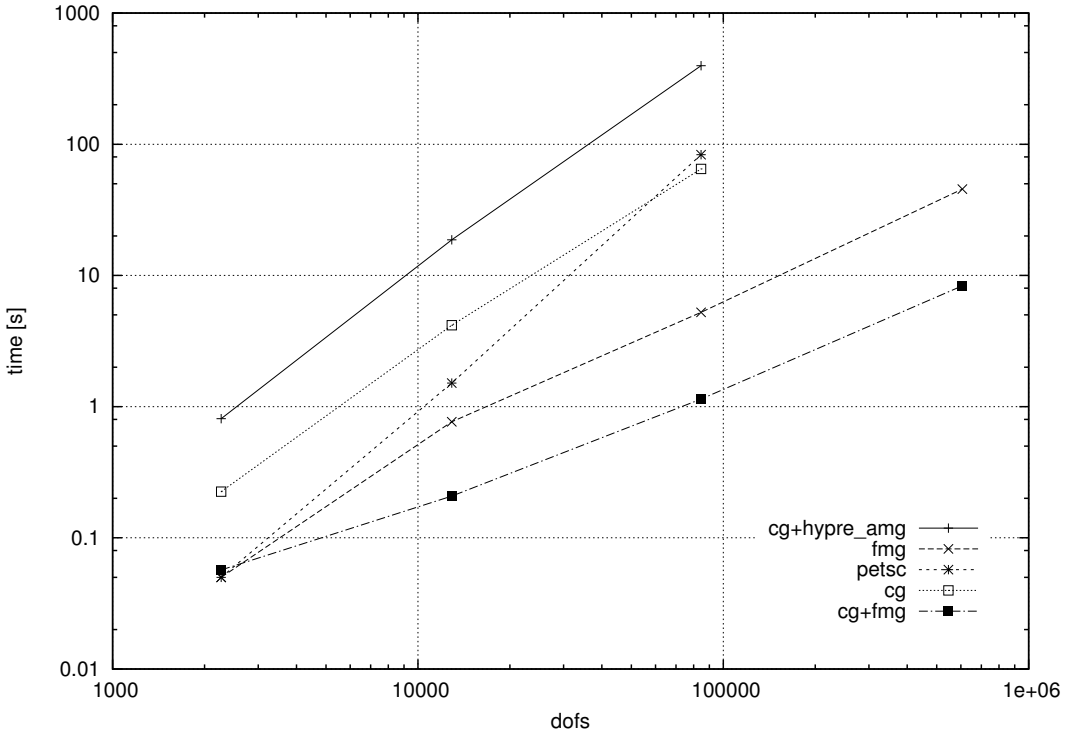


Figure 4.9: Plot of the solve time (table 4.15) for different methods.

A plot of the solve times for each solver is shown in fig. 4.9. As we can see the slopes of the lines are approximately 1 for `fmg` and `cg+fmg`, 3/2 for `cg` and `cg+hypre_amg` and 2 for `petsc`. This may be expressed in terms of an equation as

$$\frac{\log(t_2) - \log(t_1)}{\log(n_2) - \log(n_1)} = c. \tag{4.22}$$

In terms of an ordinary differential equation w.r.t. the number of dofs n, the solution may be described as $\log(t(n)) = c\log(n) + d$, where c and d are constants. This gives the following equation for the solve time:

$$t(n) = \exp(c\log(n) + d) = \exp(d)n^c. \tag{4.23}$$

This is equivalent to having a computational complexity of $\mathcal{O}(n^c)$. In turn we have a complexity of $\mathcal{O}(n)$ for `fmg` and `cg+fmg`, $\mathcal{O}(n^{3/2})$ for `cg` and `cg+hypre_amg` and $\mathcal{O}(n^2)$ for `petsc`.

smoother	reps	fmg iter	fmg time	cg+fmg iter	cg+fmg time
fsor+bsor	1	63	45.5	9	8.48
fsor+bsor	2	17	39.0	6	16.4
ssor	1	33	40.6	8	12.2
ssor	2	14	65.9	5	27.6

Table 4.17: Effect of different smoothers.

4.2.3 Different smoothers

Table 4.17 shows the number of iterations and solve time for different smoother methods (as listed in the `test_smoothers` parameter up to `max_smoothing_reps` repetitions) for the `cg+fmg` solver on the finest level. The `fsor+bsor` (foward-backward SOR) smoother with one repetition shows the default behaviour as in table 4.14. We also observe that this smoother works best for this problem.

4.2.4 Different cycle patterns

Table 4.18 shows the number of iterations and solve time for different cycle patterns (as listed in the `test_cycle_patterns` parameter) for the `cg+fmg` solver on the finest level. The meaning of the different cycle patterns is shown in fig. 4.3. In this case, we may say that a `V`-cycle works best for the `cg+fmg` solver, however if changing the convergence tolerances this may change. For the `fmg` solver we observe, that the `2/V` cylce approximately halves the number of iterations, which makes sense, since the `2/V`-cycle is equivalent to doing 2 iterations on a single `V`-cycle.

4.2.5 Different coarse level solvers

Table 4.19 shows the number of iterations and solve time for different solver for the coarse level (as listed in the `test_coarse_solver` parameter) for the `cg+fmg` solver on the finest level. As we see, using a direct solver like `petsc` as coarse solver for this problem is preferable to an iterative solver like `cg`.

pattern	fmg iter	fmg time	cg+fmg iter	cg+fmg time
V	63	48.9	9	8.93
2/V	31	49.1	6	11.6
1/2/V	54	47.3	8	9.06
1/1/2/V	58	45.7	9	9.06

Table 4.18: Effect of different cycle patterns.

coarse solver	fmg iter	fmg time	cg+fmg iter	cg+fmg time
petsc	63	49.5	9	8.95
cg	63	62.1	9	11.2
cg+hypre_amg	63	90.5	9	16.6
mumps	63	48.9	9	9.02

Table 4.19: Effect of different coarse level solver.

operation	reps	max time	total	time/rep	time/rep %
calc defect	9	0.166	0.551	0.0612	23.9
pre smooth	9	0.217	0.694	0.0772	30.2
post smooth	9	0.270	0.927	0.103	40.3
restrict	9	0.0162	0.0556	0.00618	2.42
prolongate	9	0.0154	0.0527	0.00585	2.29
correct error	9	0.00310	0.0104	0.00116	0.452
coarse solve	3	0.00107	0.00318	0.00106	0.414
test break	3	8.11e-06	2.19e-05	7.31e-06	0.00286

Table 4.20: Execution timings for 3 cycles (2.3s) of the multigrid algorithm.

4.2.6 Execution timings

Table 4.20 shows the execution time of different operations for 3 cycles (2.3s) of the multigrid algorithm (`fmg` solver) on the finest level, using a nonzero initial guess. As we see the defect calculation, pre- and post smoothing are the most expensive operations, as they all involve vector multiplication with the system matrix. Restriction and prolongation operators also involve matrix vector multiplication, but usually the prolongation matrix is much sparser than the system matrix. The error correction step only involves vector addition. Using `petsc` as coarse solver, the solve time does not have much influence on the overall solve time for this example. However, using `cg` as coarse solver will affect the preconditioner performance negatively.

4.3 Preconditioning of the Stokes problem

The Stokes problem describes a steady state, incompressible, laminar flow. Let the domain $\Omega \subset \mathbb{R}^d$ $(1 \leq d \leq 3)$, the *Stokes problem* is to find a velocity field $u : \Omega \mapsto \mathbb{R}^d$ and a pressure field $p : \Omega \mapsto \mathbb{R}$, such that

$$-\mu \Delta u + \nabla p = f \quad \text{in } \Omega, \tag{4.24}$$

$$\nabla \cdot u = 0 \quad \text{in } \Omega, \tag{4.25}$$

$$u = g \quad \text{on } \Gamma_D, \tag{4.26}$$

where μ is the dynamic viscosity and $g : \Omega \mapsto \mathbb{R}^d$ is a given velocity field. The corresponding variational form is: find $(u, p) \in H_D(\Omega) \times \Pi(\Omega)$, such that

$$\begin{aligned} \mu \int_\Omega \nabla u : \nabla v \, \mathrm{d}x - \int_\Omega \nabla \cdot u q \, \mathrm{d}x - \int_\Omega \nabla \cdot v p \, \mathrm{d}x \\ = \int_\Omega f v \, \mathrm{d}x, \quad \forall (v, q) \in H_D(\Omega) \times \Pi(\Omega), \end{aligned} \tag{4.27}$$

where

$$H_D(\Omega) = \{v \in (H^1(\Omega))^2 : \mathrm{trace}(v) = g \text{ on } \Gamma_D\} \text{ and} \tag{4.28}$$

$$\Pi(\Omega) = \left\{ q \in L^2(\Omega) : \int_\Omega q \, \mathrm{d}x = 0 \right\}. \tag{4.29}$$

The corresponding linear system may be written as (just to see the structure)

$$(\mathcal{A}(u, p), (v, q)) = (\mathcal{L}, (v, q)), \quad \forall (v, q) \in H_D(\Omega) \times \Pi(\Omega), \tag{4.30}$$

with

$$\mathcal{A} = \begin{pmatrix} A & B^* \\ B & 0 \end{pmatrix} \quad \text{and} \quad \mathcal{L} = \begin{pmatrix} C \\ 0 \end{pmatrix}, \tag{4.31}$$

where

$$(Au, v) = \mu \int_\Omega \nabla u : \nabla v \, \mathrm{d}x, \tag{4.32}$$

$$(Bu, q) = -\int_\Omega \nabla \cdot u q \, \mathrm{d}x, \tag{4.33}$$

$$Cv = \int_\Omega f v \, \mathrm{d}x. \tag{4.34}$$

For this kind of problem preconditioning is essential. Therefore the system is solved, using MINRES with an approximation of the following block-preconditioner

$$\mathcal{B} = \begin{pmatrix} K & 0 \\ 0 & M \end{pmatrix}, \tag{4.35}$$

where

$$(Ku, v) = \mu \int_\Omega \nabla u : \nabla v \, \mathrm{d}x, \tag{4.36}$$

$$(Mp, q) = \int_\Omega pq \, \mathrm{d}x. \tag{4.37}$$

More details can be found in [MH12].

For simplicity, we assume $\mu = 1$ in the following. We will solve two different problems in $d = 2$, each discretized using Taylor-Hood elements (P_2 for $H_D(\Omega)$ and P_1 for $\Pi(\Omega)$). The respective UFL code for $\mathcal{A}$, $\mathcal{L}$, K and M is given by:

```
P2 = VectorElement("Lagrange", triangle, 2)
P1 = FiniteElement("Lagrange", triangle, 1)
TH = P2 * P1

(u, p) = TrialFunctions(TH)
(v, q) = TestFunctions(TH)
f = Coefficient(P2)

a = (inner(grad(u), grad(v)) - div(v)*p - div(u)*q)*dx
L = dot(f, v)*dx
```

```
P2 = VectorElement("Lagrange", triangle, 2)

u = TrialFunction(P2)
v = TestFunction(P2)
z = Constant(triangle)

a = inner(grad(u), grad(v))*dx
L = dot(as_vector([z,z]),v)*dx
```

```
P1 = FiniteElement("Lagrange", triangle, 1)

p = TrialFunction(P1)
q = TestFunction(P1)
z = Constant(triangle)

a = p*q*dx
L = z*q*dx
```

Note: The linear forms L within the UFL for the preconditioners are zero and only required to construct the multigrid preconditioner.

For both problems, we test the performance of the preconditioned MINRES, using different approaches for the application of the block-preconditioner:

- apply an approximate K^{-1}, using the `FMG` preconditioner,
- using the Hypre AMG or Sandia ML AMG preconditioner,
- using an exact LU-factorization.

The M^{-1} block is approximated by using a Jacobi preconditioner (inverse of diagonal of M) for all cases. For all computations we used the default solver and preconditioner settings. The demo programs don't use the `fmg::Tests` class, but they can be invoked using a similar syntax:

```
./main --num_refinements 7 --prec X
```

where `X` is either `fmg`, `hypre_amg`, `ml_amg` or `lu`.

4.3.1 Lid-driven cavity

The source of this demo is located in `demo/stokes-lid-driven-cavity` of the `FMG` installation. For this problem we use:

$$\Omega = (0,1) \times (0,1), \quad \Gamma_D = \partial\Omega, \tag{4.38}$$

$$g(x,y) = \begin{cases} (1,0)^T & \text{for } y = 1 \\ (0,0)^T & \text{else.} \end{cases} \tag{4.39}$$

We start on a `UnitSquare(2,2)` and the mesh is refined 7 times uniformly. For each refinement, the corresponding system and preconditioner sizes are shown in table 4.21. The initialization time is the total time to assemble all matrices and to initialize the `FMG` preconditioner. However, since the current implementation of this demo can be improved further, and the initialization time is relatively small, compared to the observed solve time, we can neglect the `FMG` initialization time (which is also only 20% of this time) for the comparison. The multigrid preconditioners are applied only on the K-block, using the default settings (i.e. 1 V-cycle and Gauss-Seidel smoothing for `FMG`). For the M-block a Jacobi (diagonal) preconditioner was used. Table 4.22 shows the results, when using the `FMG` preconditioner. The total time required by the preconditioners is shown, as well as the number of MINRES iterations and the total solve time. Table 4.23 shows the results, when using an LU-factorization as preconditioner (i.e. the K-block is inverted exactly). This also shows the best possible iteration numbers. Table 4.24 and table 4.25 shows the results, using the Hypre AMG and Sandia's ML AMG preconditioner. Here we see a similar behaviour to `FMG`, but with an increased solve time. Summarized in fig. 4.10, `FMG` seems to be superior method for this problem, regarding solve time. However, it is possible to to do various adjustments for each of the multigrid preconditioners, to eventually get better results. Figure 4.11 and fig. 4.12, show the solution of this problem.

nrefine	dofs	K dofs	M dofs	init time
0	59	50	9	0.0610
1	187	162	25	0.00899
2	659	578	81	0.0184
3	2 467	2 178	289	0.0508
4	9 539	8 450	1 089	0.188
5	37 507	33 282	4 225	0.671
6	148 739	132 098	16 641	2.67
7	592 387	526 338	66 049	10.9

Table 4.21: System and preconditioner size, initialization time for the "lid-driven cavity" problem.

nrefine	dofs	prec K time	prec M time	minres iter	minres time (total)
0	59	0.00144	0.000149	22	0.00251
1	187	0.0118	0.000201	46	0.0140
2	659	0.0398	0.000604	62	0.0531
3	2 467	0.129	0.00103	73	0.182
4	9 539	0.416	0.00168	80	0.561
5	37 507	1.78	0.00395	87	2.51
6	148 739	7.59	0.0129	92	10.7
7	592 387	32.5	0.0539	97	45.8

Table 4.22: Performance of the "lid-driven cavity" problem, using the `FMG` preconditioner.

nrefine	dofs	prec K time	prec M time	minres iter	minres time (total)
0	59	0.00138	0.000103	22	0.00243
1	187	0.00269	0.000156	37	0.00463
2	659	0.0118	0.000320	49	0.0182
3	2 467	0.0619	0.000837	56	0.0887
4	9 539	0.312	0.00125	59	0.421
5	37 507	1.72	0.00323	63	2.25
6	148 739	9.87	0.0110	67	12.1
7	592 387	64.1	0.0404	69	73.5

Table 4.23: Performance of the "lid-driven cavity" problem, using an LU-factorization for the preconditioner.

nrefine	dofs	prec K time	prec M time	minres iter	minres time (total)
0	59	0.00149	0.000121	22	0.00273
1	187	0.00608	0.000186	38	0.00844
2	659	0.0356	0.000522	51	0.0450
3	2 467	0.154	0.000924	58	0.183
4	9 539	0.914	0.00146	65	1.03
5	37 507	4.62	0.00346	69	5.20
6	148 739	20.6	0.0103	72	23.0
7	592 387	87.9	0.0426	76	98.2

Table 4.24: Performance of the "lid-driven cavity" problem, using the Hypre AMG preconditioner.

nrefine	dofs	prec K time	prec M time	minres iter	minres time (total)
0	59	0.00900	0.000141	24	0.0103
1	187	0.0147	0.000236	42	0.0173
2	659	0.0502	0.000836	60	0.0621
3	2 467	0.187	0.00106	74	0.224
4	9 539	0.830	0.00183	86	0.990
5	37 507	4.57	0.00616	110	5.51
6	148 739	22.7	0.0202	129	27.1
7	592 387	110.	0.0945	154	131.

Table 4.25: Performance of the "lid-driven cavity" problem, using Sandia's ML AMG preconditioner.

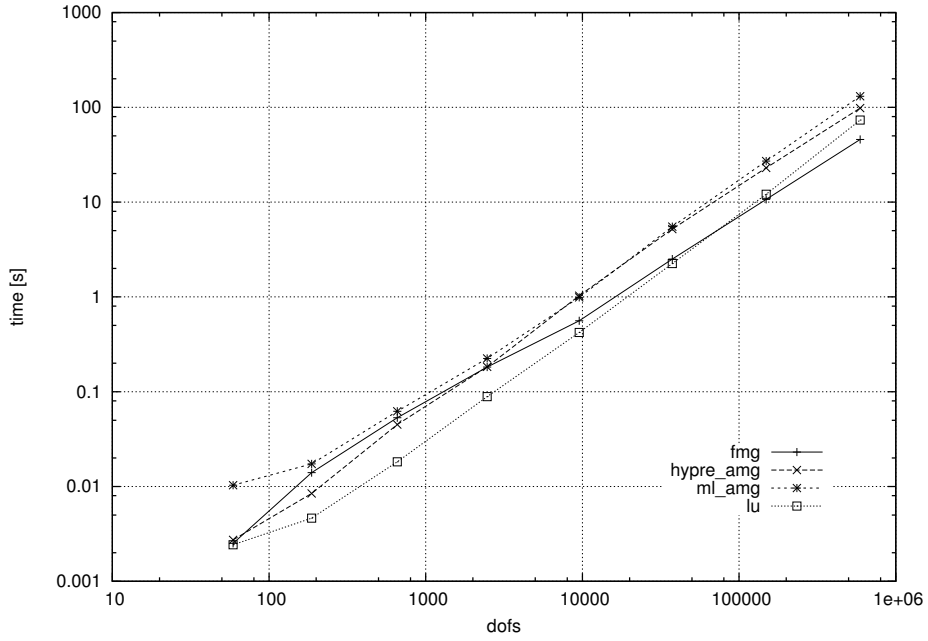


Figure 4.10: Solve times for the "lid-driven cavity" problem, using different preconditioners for the M-block.

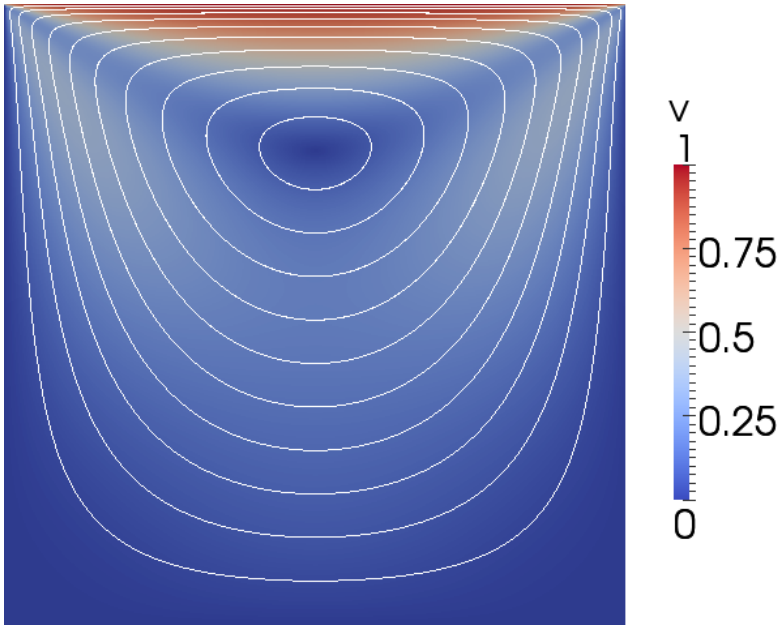


Figure 4.11: Velocity magnitude plot with flow lines for the ”lid-driven cavity” problem.

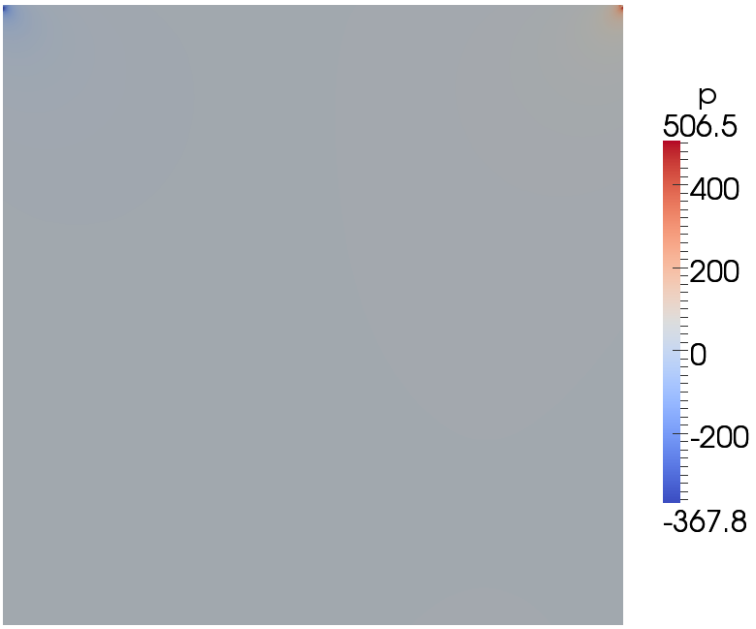


Figure 4.12: Pressure plot for the ”lid-driven cavity“ problem.

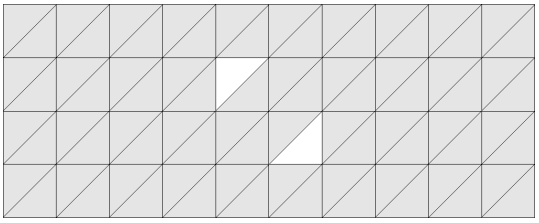


Figure 4.13: Initial mesh for the "pipe with obstacles" problem.

4.3.2 Pipe with obstacles

The source of this demo is located in `demo/stokes-taylor-hood` of the `FMG` installation. For this problem we use:

$$\Omega = ((0,1) \times (0.3, 0.7)) \setminus (T_1 \cup T_2), \tag{4.40}$$

$$\Gamma_D = \partial\Omega \setminus (\{1\} \times (0.3, 0.7)), \tag{4.41}$$

$$g(x,y) = \begin{cases} (\cos^2(\pi(y-0.5)/0.4), 0)^T & \text{for } x = 0 \\ (0,0)^T & \text{else.} \end{cases} \tag{4.42}$$

where T_1 and T_2 denote the two cutout triangles, as shown in fig. 4.13

For each refinement, the corresponding system and preconditioner sizes are shown in table 4.26. The initialization time is the total time to assemble all matrices and to initialize the `FMG` preconditioner. However, since the current implementation of this demo can be improved further, and the initialization time is relatively small, compared to the observed solve time, we can neglect the `FMG` initialization time (which is also only 20% of this time) for the comparison. Table 4.27 shows the results, when using the `FMG` preconditioner. The total time required by the preconditioners is shown, as well as the number of MINRES iterations and the total solve time. In contrast to the "lid-driven cavity" problem, we see that the number of MINRES iterations become relatively constant after a few refinements. Table 4.28 shows the results, when using an LU-factorization as preconditioner (i.e. the K-block is inverted exactly). This also shows the best possible iteration numbers. Table 4.29 and table 4.30 shows the results, using the Hypre AMG and Sandia's ML AMG preconditioner. Here we see a similar behaviour to `FMG`, but with an increased solve time. Summarized in fig. 4.14, `FMG` seems to be superior method for this problem, regarding solve time. However, it is possible to to do various adjustments for each of the multigrid preconditioners, to eventually get better results. Figure 4.15 and fig. 4.16, show the solution of this problem.

nrefine	dofs	K dofs	M dofs	init time
0	433	378	55	0.0685
1	1 571	1 382	189	0.0289
2	5 953	5 262	691	0.107
3	23 141	20 510	2 631	0.410
4	91 213	80 958	10 255	1.65
5	362 141	321 662	40 479	6.54
6	1 443 133	1 282 302	160 831	26.5

Table 4.26: System and preconditioner size, initialization time for the "pipe with obstacles" problem.

nrefine	dofs	prec K time	prec M time	minres iter	minres time (total)
0	433	0.00628	0.000182	40	0.00925
1	1 571	0.0475	0.000632	60	0.0651
2	5 953	0.196	0.00112	65	0.264
3	23 141	0.799	0.00233	66	1.11
4	91 213	3.27	0.00584	66	4.60
5	362 141	13.6	0.0232	68	19.1
6	1 443 133	54.4	0.0877	68	76.1

Table 4.27: Performance of the "pipe with obstacles" problem, using the `FMG` preconditioner.

nrefine	dofs	prec K time	prec M time	minres iter	minres time (total)
0	433	0.00612	0.000183	40	0.00953
1	1 571	0.0318	0.000620	46	0.0559
2	5 953	0.154	0.000900	50	0.211
3	23 141	0.816	0.00191	50	1.08
4	91 213	4.66	0.00605	52	5.81
5	362 141	24.8	0.0210	50	29.1
6	1 443 133	161.	0.0698	50	178.

Table 4.28: Performance of the ”pipe with obstacles“ problem, using an LU-factorization for the preconditioner.

nrefine	dofs	prec K time	prec M time	minres iter	minres time (total)
0	433	0.0123	0.000234	41	0.0160
1	1 571	0.0674	0.000644	49	0.0829
2	5 953	0.403	0.00103	52	0.460
3	23 141	2.09	0.00186	54	2.35
4	91 213	9.73	0.00526	56	10.9
5	362 141	39.5	0.0199	56	44.0
6	1 443 133	152.	0.0669	54	170.

Table 4.29: Performance of the ”pipe with obstacles“ problem, using the Hypre AMG preconditioner.

nrefine	dofs	prec K time	prec M time	minres iter	minres time (total)
0	433	0.0295	0.000369	44	0.0345
1	1 571	0.124	0.000720	55	0.145
2	5 953	0.443	0.00113	64	0.518
3	23 141	1.81	0.00265	75	2.17
4	91 213	9.13	0.00850	86	10.9
5	362 141	41.3	0.0349	97	49.1
6	1 443 133	195.	0.143	114	232.

Table 4.30: Performance of the "pipe with obstacles" problem, using Sandia's ML AMG preconditioner.

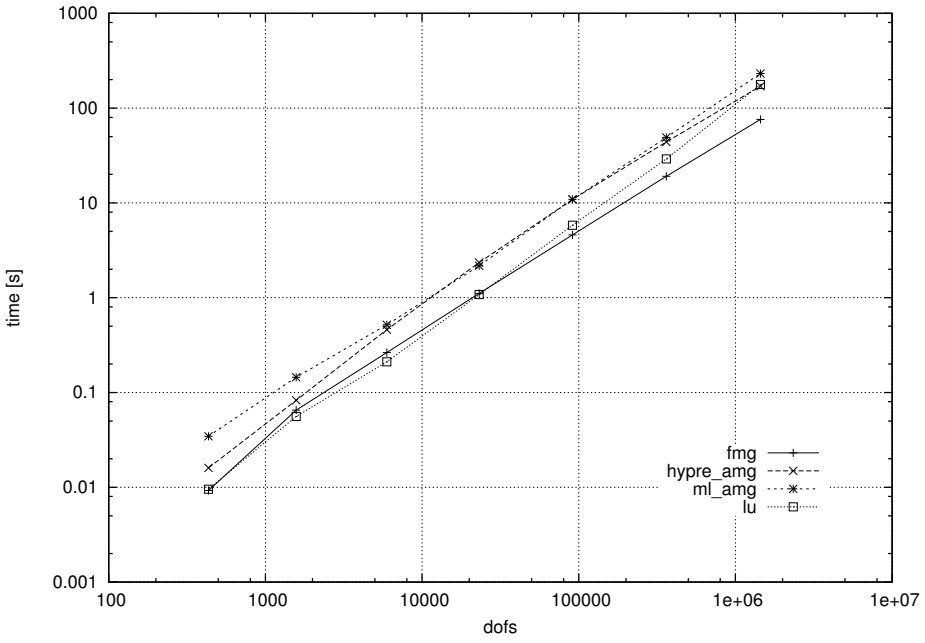


Figure 4.14: Solve times for the "pipe with obstacles" problem, using different preconditioners for the K-block.

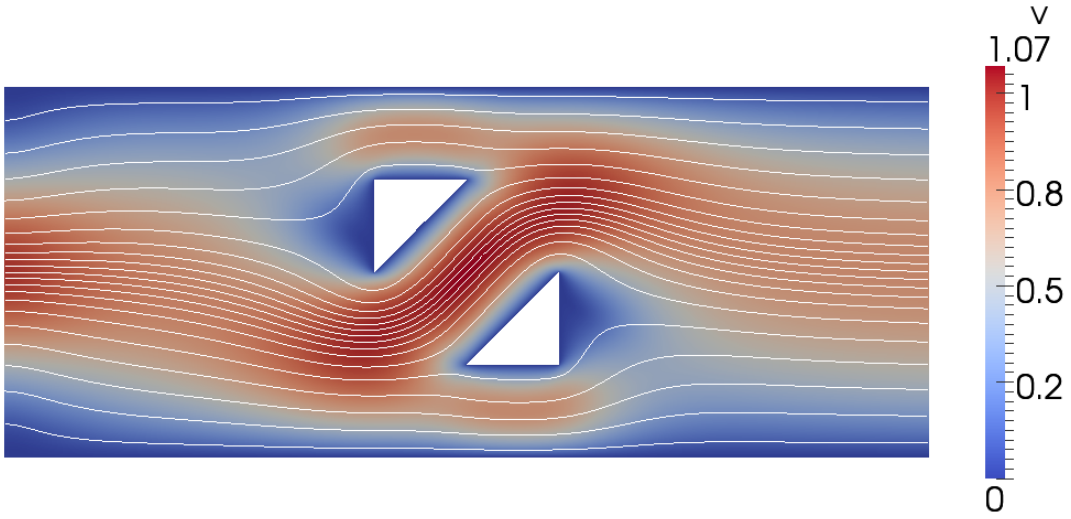


Figure 4.15: Velocity magnitude plot with flow lines for the ”pipe with obstacles“ problem.

Figure 4.16: Pressure plot for the ”pipe with obstacles” problem.

4.4 Prolongation operator test

In order to check the correctness of the prolongation operator, we performed a sequence of tests, based on the Galerkin approximation, as described in section 2.5.6.

Therefore, we use a two grid scheme $V_1 \subset V_2$ and compute the matrix

$$\Delta A = p^T A_2 p - A_1, \tag{4.43}$$

where p is the prolongation operator from the coefficient space of V_1 to the coefficient space of V_2, and A_1 and A_2 are mass matrices assembled for V_1 and V_2 respectively, using the following UFL code:

```
V = FiniteElement(family, cell, d)
u = TrialFunction(V)
v = TestFunction(V)
a = inner(u, v)*dx
```

Here `family` is one of

- `"CR"`: Crouzeix-Raviart,
- `"CG"`: Lagrange,
- `"DG"`: Discontinuous Lagrange,
- `"N1curl"`: Nédélec 1st kind H(curl),
- `"N2curl"`: Nédélec 2nd kind H(curl),
- `"N1div"` (`"RT"`): Nédélec 1st kind H(div) (Raviart-Thomas),
- `"N2div"` (`"BDM"`): Nédélec 2nd kind H(div) (Brezzi-Douglas-Marini),

`cell` is one of `"interval"`, `"triangle"` or `"tetrahedron"` and the polynomial degree ($\mathcal{P}$-order) `d` is 1,2 or 3. For creating the function space V_1, we used as mesh

- `UnitInterval(2)` for `cell = "interval"` (dim = 1),
- `UnitCircle(2)` for `cell = "triangle"` (dim = 2),
- `UnitSphere(2)` for `cell = "tetrahedron"` (dim = 3),

and the mesh for V_2 is the uniform refinement of the mesh of V_1.

The corresponding UFL files we generated using a script, which is located in `demo/elements` of the `FMG` installation. The `main.cpp` includes the header files, generated by FFC, and computes $\|\Delta A\|_\infty$ for each of the `(family, cell, d)` combinations. The result of the computation is given in table 4.31 (`"interval"` cells omitted). As we see the deviation $\|\Delta A\|_\infty$ is very small (round-off error) for all elements, except for the Crouzeix-Raviart element, which does not satisfy $V_1 \subset V_2$. We also observe, that the round-off error increases with polynomial degree and complexity of the finite element families. This error may be slightly reduced using a smaller truncation value for the `fmg::ProlongationAssembler::assemble_matrix` method (default `eps = 1e-13`) or raising the default `-f precision=15` parameter of the FFC compiler to 17 digits.

Element	**dim**	$\mathcal{P}$**-order**	$\|\Delta A\|_\infty$
Crouzeix-Raviart	2	1	0.0441942
Crouzeix-Raviart	3	1	0.0777855
Lagrange	2	1	6.41848e-16
Lagrange	2	2	2.31802e-16
Lagrange	2	3	7.07767e-16
Lagrange	3	1	1.12757e-15
Lagrange	3	2	2.90133e-16
Lagrange	3	3	4.10096e-16
Discontinuous Lagrange	2	0	0.0
Discontinuous Lagrange	2	1	9.36751e-17
Discontinuous Lagrange	2	2	1.1395e-16
Discontinuous Lagrange	2	3	6.94974e-16
Discontinuous Lagrange	3	0	2.08167e-17
Discontinuous Lagrange	3	1	1.75207e-16
Discontinuous Lagrange	3	2	7.91468e-17
Discontinuous Lagrange	3	3	2.52152e-16
Nédélec 1st kind H(curl)	2	1	2.81719e-15
Nédélec 1st kind H(curl)	2	2	1.82583e-12
Nédélec 1st kind H(curl)	2	3	7.45132e-11
Nédélec 1st kind H(curl)	3	1	4.23966e-15
Nédélec 1st kind H(curl)	3	2	1.87092e-14
Nédélec 1st kind H(curl)	3	3	4.37431e-11
Nédélec 2nd kind H(curl)	2	1	9.39526e-15
Nédélec 2nd kind H(curl)	2	2	3.52821e-11
Nédélec 2nd kind H(curl)	2	3	1.96281e-10
Nédélec 2nd kind H(curl)	3	1	1.55865e-14
Nédélec 2nd kind H(curl)	3	2	4.89958e-11
Nédélec 2nd kind H(curl)	3	3	2.1819e-10
Nédélec 1st kind H(div) (Raviart-Thomas)	2	1	2.1233e-15
Nédélec 1st kind H(div) (Raviart-Thomas)	2	2	1.46237e-14
Nédélec 1st kind H(div) (Raviart-Thomas)	2	3	2.23207e-14
Nédélec 1st kind H(div) (Raviart-Thomas)	3	1	6.54392e-15
Nédélec 1st kind H(div) (Raviart-Thomas)	3	2	1.32377e-14
Nédélec 1st kind H(div) (Raviart-Thomas)	3	3	6.1767e-14
Nédélec 2nd kind H(div) (Brezzi-Douglas-Marini)	2	1	7.61891e-15
Nédélec 2nd kind H(div) (Brezzi-Douglas-Marini)	2	2	3.52681e-11
Nédélec 2nd kind H(div) (Brezzi-Douglas-Marini)	2	3	2.60247e-10
Nédélec 2nd kind H(div) (Brezzi-Douglas-Marini)	3	1	5.15421e-14
Nédélec 2nd kind H(div) (Brezzi-Douglas-Marini)	3	2	2.5719e-10
Nédélec 2nd kind H(div) (Brezzi-Douglas-Marini)	3	3	3.41726e-09

Table 4.31: Results for the prolongation operator test.

4.5 A note on performance improvement

During tests, it was discovered, that disabling the FEniCS parameter `reorder_dofs` (`reorder_dofs_serial` in FEniCS 1.0.0+), generally reduces the overall runtime, given the number of solver iterations is not too high. In particular disabling dof reordering did

- increase the solve time about 25%, when using `cg+fmg` as solver (probably because of a better inode blocking by PETSc),
- decrease the solve time about 3%, when using `petsc` as solver,
- decrease the initialization time about 20-40%,
- decrease the (peak) memory usage.

It depends on the solver and number of iterations, required for the solution of the problem, if disabling dof reordering pays off. When using the `cg+fmg` solver, disabling dof reordering payed off for all considered problems in this chapter, regarding the total runtime.

5 Conclusions and outlook

We successfully implemented a geometric multigrid solver/preconditioner for FEniCS. The source and documentation for the project was published under the GNU GPL v3 license at `http://launchpad.net/fmg` and is freely available for everyone. The project was designed under modern object oriented patterns (as FEniCS) and is easy usable and extendable. Further the `fmg::Tests` class provides an easy way to test the performance of the solver, for a user supplied problem under different solver settings.

We applied the `FMG` solver, to solve a Poisson problem, a problem from linear elasticity and two Stokes flow problems. For the problems, the computation time was considerable reduced or at least equal in contrast to other solvers available within FEniCS. However the initialization time for the multigrid levels, has still a considerable weight. But at least this is also a proof of concept, that a geometric multigrid implementation can be realized in FEniCS, and is worth to be added to FEniCS.

During the work on the `FMG` solver, an earlier development release of DOLFIN was also functional with `FMG`. However the development of the next DOLFIN release seems to make such large progress, that the current `FMG` version does not compile anymore with the current DOLFIN development release, since some structural changes in class inheritance and some interface definitions. However we also see that these changes absolutely point to the right direction (for example the introduction of `GenericLinearOperator`s as generalization of matrices), and hope that these changes will also be beneficial for this project.

In this last paragraph let us outline, which work could be done in the future:

- **reduce the solver initialization time**: From table 4.1, we saw that we currently have an excess of 50% for the multigrid initialization for the 2d Poisson problem. From table 4.12, we saw an excess of 20% for the initialization for the 3d elasticity problem. For both problems the initialization time of the prolongation operators were only about 20% of this excess. This is already good, but there might be ways to improve this, like a more integrated concept of mesh refinement and prolongation operator assembly. Further analysis is required to find out, why the Poisson problem behaves so much worse regarding the initialization time.
- **support of non-nested function spaces**: Nested function spaces are actually not required for the definition of a prolongation operator. However for nested spaces, the prolongation can be chosen to be the canonical injection,

which greatly simplifies everything and seems to work for all element families, supported by FEniCS, except for the Crouzeix-Raviart element.

- **independent linear algebra back-end**: Currently `FMG` totally relies on PETSc as linear algebra back-end. It would be nice to support other back-ends as well. However, PETSc seems to be the preferable choice regarding performance and the availability of different relaxation methods.
- **more relaxation methods**: Like a SOR preconditioned Chebyshev iteration (used as default smoother in PETSc PCMG).
- **F-cycle pattern**: Implementation of a F-cycle pattern, using the `fmg::GenericCyclePattern` class and adding support for full multigrid.
- **additive multigrid**: Besides the standard multigrid (multiplicative multigrid), there is also an additive multigrid method, which has an equal work count and is preferably used for preconditioning [BHW]. This method could be integrated into `FMG` and compared to the standard multigrid.
- **add Python support**: Of course it would be nice to use `FMG` with the Python part of DOLFIN.
- **parallel computing**: Currently `FMG` does not work in parallel. Parallelization could be added after the next FEniCS release, as it will include support for mesh partitioning.
- **implementation/adaptations for FEniCS 1.0.0+ release**: The present code does only work with FEniCS 1.0.0, changes are necessary to work with the next FEniCS release.

Bibliography

[Aln+12] M.S. Alnaes et al. "Unified Form Language: A domain-specific language for weak formulations of partial differential equations". In: *arXiv preprint arXiv:1211.4047* (2012) (cit. on p. 48).

[Alt99] Hans Wilhelm Alt. *Linear functional analysis. An application oriented introduction*. Springer-Verlag, 1999 (cit. on p. 5).

[Bal+12a] Satish Balay et al. *PETSc Users Manual*. Tech. rep. ANL-95/11 - Revision 3.3. Argonne National Laboratory, 2012 (cit. on p. 58).

[Bal+12b] Satish Balay et al. *PETSc Web page*. http://www.mcs.anl.gov/petsc. 2012 (cit. on pp. 2, 58).

[BHW] Peter Bastian, Wolfgang Hackbusch, and Gabriel Wittum. *Additive and Multiplicative Multi-Grid - a Comparison* (cit. on p. 128).

[Boo] *Boost C++ Libraries*. http://www.boost.org (cit. on p. 59).

[Bra03] Dietrich Braess. *Finite Elemente : Theorie, schnelle Löser und Anwendungen in der Elastizitätstheorie*. Springer-Lehrbuch. Berlin: Springer, 2003. ISBN: 3-540-00122-0 (cit. on pp. 1, 31, 41).

[Bra77] Achi Brandt. "Multi-Level Adaptive Solutions to Boundary-Value Problems". In: *Mathematics of Computation* 31.138 (Apr. 1977), pp. 333–390. ISSN: 00255718. DOI: 10.2307/2006422. URL: http://dx.doi.org/10.2307/2006422 (cit. on p. 1).

[BS02] S.C. Brenner and L.R. Scott. *The Mathematical Theory of Finite Element Methods*. Springer-Verlag, 2002 (cit. on pp. 10, 12).

[Cia78] Philippe G. Ciarlet. *The finite element method for elliptic problems / Philippe G. Ciarlet*. English. North-Holland Pub. Co. ; sole distributors for the U.S.A. and Canada, Elsevier North-Holland, Amsterdam ; New York : New York : 1978, xvii, 530 p. : ISBN: 0444860169 0444850287 (cit. on pp. 10, 12, 14).

[Fed01] R.P. Fedorenko. *On the history of the Multigrid method creation (translated by M.A. Botchev)*. http://wwwhome.math.utwente.nl/~botchevma/fedorenko/index.php. University of Manchester, 2001 (cit. on p. 1).

[Fed61] R.P. Fedorenko. *A relaxation method for solving elliptic difference equations*. http://mi.mathnet.ru/zvmmf8014. 1961 (cit. on p. 1).

[Fra50] Stanley P. Frankel. *Convergence rates of iterative treatments of partial differential equations*. http://www.ams.org/journals/mcom/1950-04-030/S0025-5718-1950-0046149-3/. 1950 (cit. on p. 1).

[Gee+06] M.W. Gee et al. *ML 5.0 Smoothed Aggregation User's Guide*. Tech. rep. SAND2006-2649. Sandia National Laboratories, 2006 (cit. on p. 2).

[Gue+11] A. Guennel et al. *Convergence analysis for Krylov subspace methods in Hilbert space*. http://www.tu-chemnitz.de/mathematik/part_dgl/publications. 2011 (cit. on pp. 28 sq.).

[Hac76] Wolfgang Hackbusch. *Ein iteratives Verfahren zur schnellen Auflösung elliptischer Randwertprobleme*. Math. Inst., Univ., 1976. URL: http://books.google.de/books?id=bAO9PgAACAAJ (cit. on p. 1).

[Hac85] Wolfgang Hackbusch. *Multigrid Methods and Applications*. Springer-Verlag Berlin, 1985 (cit. on pp. 1, 30, 34, 39 sq.).

[Hem80] P. W. Hemker. "On the structure of an adaptive multi-level algorithm". In: (1980) (cit. on p. 1).

[HS52] M.R. Hestenes and E. Stiefel. *Methods of conjugate gradients for solving linear systems*. http://nvl.nist.gov/pub/nistpubs/jres/049/6/V49.N06.A08.pdf. 1952 (cit. on pp. 1, 25).

[HY00] Van Emden Henson and Ulrike Meier Yang. "BoomerAMG: a Parallel Algebraic Multigrid Solver and Preconditioner". In: *Applied Numerical Mathematics* 41 (2000), pp. 155–177 (cit. on p. 2).

[LMW+12] Anders Logg, Kent-Andre Mardal, Garth N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012. ISBN: 978-3-642-23098-1. DOI: 10.1007/978-3-642-23099-8 (cit. on pp. 2, 12, 16, 19, 47).

[Log+12] A. Logg et al. "FFC: the FEniCS form compiler". In: *Automated Solution of Differential Equations by the Finite Element Method* (2012), pp. 227–238 (cit. on p. 48).

[McC87] Stephen F. McCormick. *Multigrid Methods*. SIAM, 1987 (cit. on p. 1).

[MH12] Kent-Andre Mardal and Joachim Berdal Haga. "Block preconditioning of systems of PDEs". In: *Automated Solution of Differential Equations by the Finite Element Method*. Ed. by K.-A. Mardal A. Logg and G. Wells. Vol. 84. Lecture Notes in Computational Science and Engineering, Springer, 2012. Chap. 35, pp. 643–654. ISBN: 978-3-642-23098-1 (cit. on p. 112).

[Osp12] Felix Ospald. *Studienarbeit über Multigrid-Verfahren*. TU Chemnitz, 2012 (cit. on p. 103).

[Pet] *PETSc PCMG - multigrid preconditioning*. http://www.mcs.anl.gov/petsc/petsc-current/docs/manualpages/PC/PCMG.html (cit. on p. 59).

[PS75] C. C. Paige and M. A. Saunders. "Solution of Sparse Indefinite Systems of Linear Equations". In: *SIAM Journal on Numerical Analysis* 12.4 (1975), pp. 617–629. ISSN: 00361429. DOI: 10.1137/0712047. URL: http://dx.doi.org/10.1137/0712047 (cit. on p. 28).

[Rat+08] H. T. Rathod et al. "Gauss Legendre-Gauss Jacobi quadrature rules over a tetrahedral region." In: *Applied Mathematics and Computation* 190.1 (Sept. 9, 2008), pp. 186–194 (cit. on p. 57).

[Ree11] Glyn Owen Rees. *Efficient "black-box" multigrid solvers for convection-dominated problems*. https://www.escholar.manchester.ac.uk/uk-ac-man-scw:131408. University of Manchester, 2011 (cit. on p. 20).

[Res] *reStructuredText*. http://docutils.sourceforge.net/rst.html (cit. on p. 81).

[Scha] René Schneider. http://www-user.tu-chemnitz.de/~rens/ (cit. on p. 96).

[Schb] René Schneider. *Finite Element Incompressible Navier-Stokes solver (FEINS)*. http://www-user.tu-chemnitz.de/~rens/software/feins/ (cit. on p. 96).

[Sch06] René Schneider. *Applications of the Discrete Adjoint Method in Computational Fluid Dynamics*. http://www.comp.leeds.ac.uk/research/pubs/theses/schneider.pdf. 2006 (cit. on p. 19).

[Sha95] V.V. Shaidurov. *Multigrid Methods for Finite Elements*. 1995 (cit. on p. 1).

[Som03] Andreas Sommer. *Einführung in Mehrgitterverfahren*. http://www.math.uni-trier.de/~schulz/numerik2-material/MG_sommer.pdf. Uni Trier, 2003 (cit. on p. 30).

[Tat93] Osamu Tatebe. *The Multigrid Preconditioned Conjugate Gradient Method*. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.38.7855&rep=rep1&type=pdf. 1993 (cit. on p. 42).

[TS01] Ulrich Trottenberg and Anton Schuller. *Multigrid*. Orlando, FL, USA: Academic Press, Inc., 2001. ISBN: 0-12-701070-X (cit. on pp. 1, 91).

[Wel06] G. Wells. "The Finite Element Method: An Introduction". In: *CT5123 Lecture Notes, Delft University of Technology* (2006) (cit. on p. 10).

[Wes92] P. Wesseling. *An Introduction to Multigrid Methods*. Chichester: John Wiley & Sons, 1992 (cit. on p. 1).

[Wik] Wikipedia. *FEniCS Project*. http://en.wikipedia.org/wiki/FEniCS_Project. [accessed 06-12-2012] (cit. on pp. 46 sq.).

[You50] David Young. *Iterative methods for solving partial difference equations of elliptic type*. http://www.stat.uchicago.edu/~lekheng/courses/324/young.pdf. Harvard University, 1950 (cit. on p. 1).

[ZW91] S. Zeng and P. Wesseling. *Galerkin Coarse Grid Approximation for the Incompressible Navier-Stokes Equations in General Coordinates*. 1991 (cit. on p. 40).

Appendix

A.1 FEINS modifications

Patch for `src/test_assem.c`:

```
--- test_assem.c.org    2011-03-31 21:46:14.000000000 +0200
+++ test_assem.c        2013-01-12 15:39:39.910431000 +0100
@@ -70,7 +70,7 @@
   struct timezone tz;
   int sec, musec;
   double t0, ti, t01;
-
+  double ts0;

   if (argc>1)
     {
@@ -162,8 +162,11 @@
         projector1_no_precon, &Ks[level], &rhs, &P );
         FUNCTION_FAILURE_HANDLE( err, PCG, main); /* */

+      TIMEGET;
+      ts0=ti;
+
       strcpy(solver,"PCG_MG");
-      err=PCG( 10000, 2, atol, rtol, 1, &x, &resi, &iter, sparse_mul_mat_vec,
+      err=PCG( 10000, 3, atol, rtol, 1, &x, &resi, &iter, sparse_mul_mat_vec,
               gen_proj_MG_tx, &Ks[level], &rhs, &mg );
       FUNCTION_FAILURE_HANDLE( err, PCG, main); /* */

@@ -195,8 +198,8 @@
       TIMEGET;

       printf("%s: %4d iterations, |res|=%8.2e, vx_nr= %9"dFIDX",  "
-            "u_0=%16.9e, time_level=%10.3e, time_total=%10.3e\n",
-            solver, iter, resi, msh1.vx_nr, pointerror, ti-t01, ti-t0); /* */
+            "u_0=%16.9e, time_asm=%10.3e, time_cg=%10.3e, time_total=%10.3e\n",
+            solver, iter, resi, msh1.vx_nr, pointerror, ts0-t01, ti-ts0, ti-t0); /* */

       /* printf("solution:\n");
         for (i=0; i<x.len; i++)
@@ -212,6 +215,7 @@
        15,"visual/poisson" );
        FUNCTION_FAILURE_HANDLE( err, mesh_write_solution_vtk_t1, main);/* */

+      if (0==1)
       {
        /* output for CMESS */
        char basename[100], filename[105];
```

Configuration file `meshfiles/poisson/square_cmess_tests.f1m`:

```
<solver_settings>
  <refine_ini      0 >
  <refine_type     1 >
  <refine_steps    10 >
  <adap_mark       0 >
  <adap_mark_par   0.6 >
  <solver          1       >
  <solver_atol     1.0e-15 >
  <solver_ini_rtol 1.0e-6  >
  <solver_ref_rtol 1.0e-6  >
  <write_mesh      0 >
  <write_ssegs     0 >
</solver_settings>
```

Statutory declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Chemnitz, February 8, 2013

Felix Ospald