Classification: Open

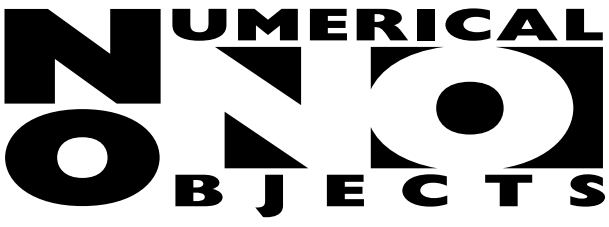
The Numerical Objects Report Series

Report 2000-01

Mixed Finite Elements in Diffpack

Hans Petter Langtangen
Kent-Andre Mardal

December 21, 2000



Numerical Objects Report 2000-01

Title

Mixed Finite Elements in Diffpack

Written by

Hans Petter Langtangen
Kent-Andre Mardal

Approved by

Not yet approved

Contact information

Phone: +47 22 06 73 00
Fax: +47 22 06 73 50
Email: info@nobjects.com
Web: http://www.nobjects.com

Contents

This report should be referenced as shown in the following BibTeX entry:

```
@techreport{NO2000-01,
  author = "Hans Petter Langtangen and Kent-Andre Mardal",
  title = "Mixed Finite Elements in Diffpack",
  institution = "Numerical Objects AS, Oslo, Norway",
  type = "The Numerical Objects Report Series",
  number = "\#{}2000-01",
  year = "December 21, 2000",
}
```

Mixed Finite Elements in Diffpack

Hans Petter Langtangen
Kent-Andre Mardal

Contents

1 Introduction

We shall in this report study two fundamental mathematical models in physics and engineering: the Stokes problem for incompressible viscous fluid flow and the Poisson equation for, e.g., inviscid fluid flow, heat conduction, porous media flow, and electrostatics. The Stokes problem cannot be discretized by a straightforward Galerkin method as this method turns out to be unstable in the sense of giving non-physical osciallations in the pressure. Mixed finite element methods, however, results in a stable solution. In this report we use the term mixed finite elements when different unknowns utilize different finite element basis functions. In Stokes problem one can think of using quadratic elements for the velocity and linear elements for the pressure. The Poisson equation, on the other hand, does not need mixed finite element methods for a stable solution. However, if the main interest regards the gradient of the solution of the Poisson equation, which is often the case, mixed finite element methods yield a more accurate computation of the gradient.

Two particular difficulties have prevented widespread application of mixed finite element methods: (i) construction of efficient solvers for the resulting linear systems and (ii) convenient flexible implementation of the methods on general unstructured grids. This report pays attention to these two difficulties and provides solution for both of them. Actually, the methods and

software from this report have applications far beyond the two model problems on which the exposition is focused. To the authors' knowledge, Diffpack is at the time of this writing the only software package that supports easy and flexible programming with mixed finite element methods on unstructured grids coupled with state-of-the-art iterative schemes, like multigrid, for optimal solution of the linear systems.

The Stokes problem can be formulated as follows:

$$-\mu\Delta\boldsymbol{v} + \nabla p = \boldsymbol{f} \text{ in } \Omega \quad \text{(equation of motion)}, \tag{1}$$

$$\nabla \cdot \boldsymbol{v} = 0 \text{ in } \Omega \quad \text{(mass conservation)}\,. \tag{2}$$

Here, $\boldsymbol{v}$ is the velocity of the fluid, p is the pressure in the fluid and $\boldsymbol{f}$ represents body forces. The Stokes equation is the stationary linearized form of the Navier-Stokes equations and describes the creeping (low Reynolds number) flow of an incompressible Newtonian fluid. The boundary conditions can be chosen as $\boldsymbol{v} = \boldsymbol{h}$ to simplify the theoretical descriptions in this report. A Neumann-type condition $\partial\boldsymbol{v}/\partial n$ is trivially included in the program examples (in complete analog to homogeneous Neumann conditions for, e.g., Poisson equations [14]).

The Poisson equation

$$-\nabla \cdot (\lambda\nabla p) = g \text{ in } \Omega, \tag{3}$$

$$p = k \text{ on } \partial\Omega, \tag{4}$$

appears in many physical contexts. One interpretation regards porous media flow, where (3) actually arises from a conservation equation combined with Newton's second law. The mass conservation equation reads $\nabla \cdot \boldsymbol{v} = g$, where $\boldsymbol{v}$ is some flux and g denotes the injection or production through wells. Newton's second law can be expressed by Darcy's law $\boldsymbol{v} = -\lambda\nabla p$. (This equation can be established as an average of the equation of motion in Stokes' problem over a large number of pores.) In porous media flow we are primarily interested in $\boldsymbol{v}$, which is usually computed by solving the Poisson equation and computing $\boldsymbol{v} = -\lambda\nabla p$ numerically. The numerical differentiation implies a loss of accuracy. The mixed formulation of the Poisson equation allows us to approximate the velocity $\boldsymbol{v}$ with at least as high accuracy as the pressure p. The appropriate formulation of the Poisson equation for application of mixed finite element methods is a system of PDEs:

$$\boldsymbol{v} + \lambda\nabla p = 0 \text{ in } \Omega \quad \text{(Darcy's law)}, \tag{5}$$

$$\nabla \cdot \boldsymbol{v} = g \text{ in } \Omega \quad \text{(mass conservation)}\,. \tag{6}$$

Appropriate boundary conditions on $\partial\Omega$ are either $\boldsymbol{v} \cdot \boldsymbol{n}$ prescribed (often zero) or p prescribed.

Although the Stokes problem and the Poisson equation have seemingly similar mathematical structure, they require different types of mixed finite element methods. Knowing how to solve these two classes of problems should provide sufficient information to solve a wide range of PDEs by mixed finite element methods.

The present report is organized as follows. In Section 2 we present the basic theory of mixed systems in an abstract setting. This abstract theory will introduce the Babuska-Brezzi condition, a condition the mixed elements should meet. In Section 3 we present finite element spaces appropriate for solving our model problems and software tools finite elements. Section 4 consider iterative methods for efficiently solving our model problems. We present the implementation of the simulators in Section 5.

2 Mixed Formulation

In this section we shall derive the finite element equations for our two model problems. First, we apply the weighted residual method [14, ch. 2.1] to the systems of PDEs and derive the resulting discrete equations. Thereafter we present continuous mixed variational formulations, which can be discretized by introducing appropriate finite-dimensional function spaces [14, ch. 2.10]. The discretization of our problems leads to the discrete version of the Babuska-Brezzi condition, which motivates the special finite elements presented in Section 3.

2.1 Weighted Residual Methods

The Stokes Problem. The starting point of a weighted residual formulation is the representation of the unknown scalar fields in terms of sums of finite element basis functions. In Stokes problem we need to use different basis functions for the velocity components and the pressure. Hence we may write

$$\boldsymbol{v} \approx \hat{\boldsymbol{v}} = \sum_{j=1}^{n_v} \boldsymbol{v}_j N_j, \tag{7}$$

$$p \approx \hat{p} = \sum_{j=1}^{n_p} p_j L_j \,. \tag{8}$$

Here, N_j are basis functions for the velocities and L_j are basis functions for the pressure. The nodal velocity vector v_j and the nodal pressure p_j are the unknown parameters to be computed. Notice that when $N_j \neq L_j$ the nodal points of the velocity and pressure fields do not necessarily coincide.

Inserting the approximations $\hat{\boldsymbol{v}}$ and $\hat{p}$ in the equation of motion and the equation of continuity results in a residual since neither $\hat{\boldsymbol{v}}$ nor $\hat{p}$ are in general exact solutions of the PDE system. The idea of the weighted residual method is to force the residuals to be zero in a weighted mean. Galerkin's method is a version of the weighted residual method where the weighting functions are the same as the basis functions N_i and L_i. Application of Galerkin's method in the present example consists of using N_i as weighting function for the equation of motion and L_i as weighting function for the equation of continuity. In this way we generate $dn_v + n_p$ equations for the $dn_v + n_p$ unknowns in d space dimensions. The second-order derivative term in the equation of motion is integrated by parts. So is often also the the pressure gradient term $\nabla\hat{p}$, resulting in the following weighted residual or discrete weak form:

$$\int_\Omega \left(\mu\nabla\hat{v}_r\cdot\nabla N_i - \frac{\partial N_i}{\partial x_r}\hat{p}\right)d\Omega = \int_\Omega f_r N_i d\Omega d\Omega, \quad r=1,\ldots,d \tag{9}$$

$$\int_\Omega L_i\nabla\cdot\hat{\boldsymbol{v}}d\Omega = 0\,. \tag{10}$$

The notation $\hat{v}_r$ means the r-th component of $\hat{\boldsymbol{v}}$, with a similar interpretation for the other quantities with subscript r.

In (9) we have integrated the ∇p term by parts. This is not necessary if differentiation of the L_i functions is straightforward, yielding a modified form of (9):

$$\int_\Omega \left(\mu\nabla\hat{v}_r\cdot\nabla N_i + N_i\frac{\partial\hat{p}}{\partial x_r}\right)d\Omega = \int_\Omega f_r N_i d\Omega\,. \tag{11}$$

Of reasons to be explained later, it might be advantagous to perturb the equation of continuity by $\epsilon\nabla^2 p$:

$$\nabla\cdot\boldsymbol{v} = \epsilon\nabla^2 p$$

Here ϵ is a regularization parameter than can be used to stabilize standard finite element discretizations of the Stokes problem. That is, with $\epsilon > 0$ one can avoid mixed finite elements. The extra term $\epsilon\nabla^2 p$ gives rise to a term

$$\int_\Omega \nabla L_i\cdot\nabla\hat{p}d\Omega - \int_{\partial\Omega} L_i\frac{\partial\hat{p}}{\partial n}d\Gamma\,.$$

Inserting the finite element expressions for $\hat{\boldsymbol{v}}$ and $\hat{p}$ in (9)–(10), perturbed with the stabilizing term $\epsilon\nabla^2\hat{p}$, gives the following linear system:

$$\sum_{j=1}^{n_v} A_{ij}v_r^j + \sum_{j=1}^{n_p} B_{ij}^{(r)}p_j = c_i^{(r)}, \quad i=1,\ldots,n_v,\ r=1,\ldots,d, \tag{12}$$

$$\sum_{r=1}^{d}\sum_{j=1}^{n_v} B_{ji}^{(r)}v_j^r - \sum_{j=1}^{n_p}\epsilon M_{ij}p_j = \epsilon\pi_i, \quad i=1,\ldots,n_p \tag{13}$$

where

$$\hat{v}_r = \sum_{j=1}^{n_v} v_j^r N_j, \tag{14}$$

$$\hat{p} = \sum_{j=1}^{n_p} p_j L_j, \tag{15}$$

$$A_{ij} = \int_\Omega \left(\sum_{k=1}^{d}\frac{\partial N_i}{\partial x_k}\frac{\partial N_j}{\partial x_k}\right) d\Omega, \tag{16}$$

$$B_{ij}^{(r)} = -\int_\Omega \frac{\partial N_i}{\partial x_r}L_j d\Omega, \tag{17}$$

$$M_{ij} = \int_\Omega \nabla L_i\cdot\nabla L_j d\Omega, \tag{18}$$

$$\pi_i = \int_{\partial\Omega} L_i\frac{\partial p}{\partial n}d\Gamma, \tag{19}$$

$$c_i^{(r)} = \int_\Omega f_r N_i d\Omega\,. \tag{20}$$

The Poisson Problem. The expansions (7)–(8) are natural candidates when formulating a weighted residual method for the Poisson equation expressed as a system of PDEs (5)–(6). A Galerkin approach consists in using N_i as weighting functions for (5) and L_i as weighting functions for (6). The pressure gradient term in (5) is integrated by parts, resulting in the following

system of discrete equations:

$$\int_\Omega \left(N_i \hat{\boldsymbol{v}} - \lambda \frac{\partial N_i}{\partial x_r} \hat{p} \right) d\Omega = 0, \quad r = 1, \ldots, d, \tag{21}$$

$$\int_\Omega L_i \nabla \cdot \hat{\boldsymbol{v}} d\Omega = \int_\Omega L_i \, g \, d\Omega \, . \tag{22}$$

fill in more, $\hat{\boldsymbol{v}} = \sum_{j=1}^{n_v} \nu_j \mathbf{N}_j$ etc etc.....

2.2 Function Space Formulation

Both our problems can be viewed within the same abstract setting. Let $\boldsymbol{H}$ be a suitable Hilbert space for $\boldsymbol{v}$ and let P be the corresponding Hilbert space for p. Then both these model problems can be formulated as:
Find $(\boldsymbol{v}, p) \in \boldsymbol{H} \times P$ such that

$$\begin{array}{lll} a(\boldsymbol{v}, \boldsymbol{w}) + b(p, \boldsymbol{w}) & = & (\boldsymbol{f}, \boldsymbol{w}), \quad \forall \boldsymbol{w} \in \boldsymbol{H}, \\ b(q, \boldsymbol{v}) & = & (g, q), \quad \forall q \in P \, . \end{array} \tag{23}$$

The analysis of the mixed elements are related to this specific structure of the problem. More about variational formulations can be found in **??**. We will need the following spaces:

- $\boldsymbol{L}^2(\Omega) = \{ \, \boldsymbol{v} \mid \int_\Omega \boldsymbol{v}^2 d\Omega < \infty \}$
- $\boldsymbol{H}^1(\Omega) = \{ \, \boldsymbol{v} \mid \sum_{k \leq 1} \int_\Omega (D^k \boldsymbol{v})^2 d\Omega < \infty \}$, where $k = |\alpha|$ and

$$D^\alpha v_i = \frac{\partial^{|\alpha|} v_i}{\partial x_1^{\alpha_1} \cdots \partial x_d^{\alpha_d}}$$

- $\boldsymbol{H}(div; \Omega) = \{ \, \boldsymbol{v} \in \boldsymbol{L}^2(\Omega) \mid \nabla \cdot \boldsymbol{v} \in L^2(\Omega) \}$

Model Problem 1. Stokes problem has the following definition of the bilinear forms:

$$a(\boldsymbol{v}, \boldsymbol{w}) = \int_\Omega \mu \nabla \boldsymbol{v} \cdot \nabla \boldsymbol{w} d\Omega, \tag{24}$$

$$b(p, \boldsymbol{w}) = -\int_\Omega p \nabla \cdot \boldsymbol{w} d\Omega \, . \tag{25}$$

$$\tag{26}$$

The product $\nabla \boldsymbol{v} \cdot \nabla \boldsymbol{w}$ is the "scalar tensor product"

$$\sum_{i=1}^{d} \nabla v_i \cdot \nabla w_i = \sum_{i=1}^{d} \sum_{j=1}^{d} \frac{\partial v_i}{\partial x_j} \frac{\partial w_i}{\partial x_j} .$$

Suitable functions spaces are for this model problem are $\boldsymbol{H} = \boldsymbol{H}^1(\Omega)$ and $P = L^2(\Omega)$. The linear forms are:

$$(\boldsymbol{f}, \boldsymbol{w}) = \int_{\Omega} \boldsymbol{f} \cdot \boldsymbol{w} d\Omega + \int_{\partial\Omega_N} (\mu \boldsymbol{w} \cdot \frac{\partial \boldsymbol{v}}{\partial n} - p\boldsymbol{n}_r \cdot \boldsymbol{w}) ds, \tag{27}$$

$$(g, q) = 0, \tag{28}$$

where $(\frac{\partial \boldsymbol{v}}{\partial n} - pn_r)$ is known at the (parts of the) boundary with natural boundary condition.

Model Problem 2. The bilinear forms in the mixed formulation of the Poisson equation reads:

$$a(\boldsymbol{v}, \boldsymbol{w}) = \int_{\Omega} \lambda^{-1} \boldsymbol{v} \cdot \boldsymbol{w} d\Omega, \tag{29}$$

$$b(q, \boldsymbol{v}) = \int_{\Omega} \nabla q \boldsymbol{v} \, d\Omega = - \int_{\Omega} q \cdot \nabla \cdot \boldsymbol{v} d\Omega \, . \tag{30}$$

We have two possible choices of the operator in (30). The first alternative is $b(q, \boldsymbol{w}) = (\nabla q, \boldsymbol{w})$. We must then require that $p \in H^1(\Omega)$ and $\boldsymbol{v} \in \boldsymbol{L}^2(\Omega)$. Another formulation is $b(q, \boldsymbol{w}) = (q, \nabla \cdot \boldsymbol{w})$, where we have used Greens lemma. We must then require that $\boldsymbol{v} \in \boldsymbol{H}(div; \Omega)$ and $p \in L^2(\Omega)$. The parameter λ is related to the porosity of the medium. It is assumed to be positive, bounded above and below, but not necessarily symmetric. We note that the difference between $\boldsymbol{H}(div; \Omega)$ and $\boldsymbol{H}^1(\Omega)$ is important in the construction of preconditioner and finite elements, which we will see later in 3.2.2 and 4.4.

The linear forms are

$$(\boldsymbol{f}, \boldsymbol{w}) = - \int_{\partial\Omega_N} (p\boldsymbol{n}_r \cdot \boldsymbol{w}) ds, \tag{31}$$

$$(g, q) = - \int_{\Omega} g \cdot q \, d\Omega \, . \tag{32}$$

2.3 The Babuska-Brezzi Condition.

In the previous section we saw that the fields $\boldsymbol{v}$ and p had different regularity requirement. The discrete analog should reflect these differences and we therefore use mixed elements. However, not any combination of elements will work. In order to get satisfactory estimates the spaces must be conveniently chosen, such that we achieve both a stable and accurate solution. Stability and accuracy are in some sense conflicting and compromise must be made. In fact most usual elements will not work satisfactory. The discrete problem can be written:

$$\begin{array}{lcll} a(\boldsymbol{v}_h, \boldsymbol{w}_h) + b(p_h, \boldsymbol{w}_h) & = & (f, \boldsymbol{w}_h) & \forall \boldsymbol{w}_h \in \boldsymbol{H}_h, \\ b(q_h, \boldsymbol{v}_h) & = & (g, q_h), & \forall q_h \in P_h, \end{array} \tag{33}$$

where we have introduced the subscript h on discrete quantities according to the tradition in theoretical mixed finite element literature. (In the discrete equations to be implemented in a program it is, from a notational point of view, advantagous to avoid a superscript h, because there are other subscripts as well. Therefore we stick to the hat notation.) We have that $\boldsymbol{v}_h \equiv \hat{v}$ and $p_h \equiv \hat{p}$.
We shortly consider the matrix interpretation, which is needed for the implementation concepts in Section 3 and Section **??**. The matrix equation reads

$$\begin{bmatrix} \boldsymbol{A}_h & \boldsymbol{B}_h^T \\ \boldsymbol{B}_h & \boldsymbol{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{v}_h \\ \boldsymbol{p}_h \end{bmatrix} = \begin{bmatrix} \boldsymbol{f}_h \\ \boldsymbol{g}_h \end{bmatrix}, \tag{34}$$

Remark: The Locking Phenomena. In Stokes problem the approximate solution of the velocity should be divergence free. If we use linear continuous elements and have Dirichlet boundary conditions zero, we will in general end up with 0 as the only function satisfying this conditions. This is indeed a poor approximation. (This result depends on the geometry of the triangulation, but it will often hold.) This suggests that we should enrich the velocity space.

This phenomena should motivate that a careful discretization of our model problems is needed. We will now address the abstract properties that should be met by the mixed elements. The following conditions will ensure a well-posed discrete problem (i.e. there exists a unique solution depending continuously on the data).

Condition 1 *There exists a constant α (independent of h) such that $a(\cdot,\cdot)$ is V-elliptic on $\boldsymbol{H}_h$:*

$$a(\boldsymbol{v}_h, \boldsymbol{v}_h) \geq \alpha \|\boldsymbol{v}_h\|^2_{\boldsymbol{H}} \quad \forall \boldsymbol{v}_h \in \boldsymbol{H}_h \tag{35}$$

Condition 2 *The Babuska-Brezzi condition. There exists a constant β (independent of h) such that*

$$0 < \beta := \inf_{q_h \in P_h} \sup_{\boldsymbol{w}_h \in \boldsymbol{H}_h} \frac{|b(\boldsymbol{w}_h, q_h)|}{\|\boldsymbol{w}_h\|_{\boldsymbol{H}} \|q_h\|_L}, \tag{36}$$

The first condition is automatically verified for the Stokes problem. However, ithis is not the case for the mixed formulation of the Poisson equation, because Condition 1 will only hold for divergence free functions. This means that we must construct the finite elements divergence free to ensure this condition. Condition 2 is a compatibility condition between the elements used for the velocity and the pressure. The mixed elements in both our problems must be designed to meet this condition.

Regularization. We can avoid the Babuska-Brezzi condition (2) by perturbing the problem. We introduce a bilinear form $c(\cdot,\cdot)$, which is positive, symmetric and coercive on P_h. Then a regularized version of (33) is: *find* $(\boldsymbol{v}_h^\epsilon, p_h^\epsilon)$ *such that*

$$\begin{array}{rcll} a(\boldsymbol{v}_h^\epsilon, \boldsymbol{w}_h) + b(p_h^\epsilon, \boldsymbol{w}_h) & = & (\boldsymbol{f}, \boldsymbol{w}_h), & \forall \boldsymbol{w}_h \in \boldsymbol{H}_h, \\ b(q_h, \boldsymbol{v}_h^\epsilon) - \epsilon c(q_h, p_h^\epsilon) & = & (g, q_h), & \forall q_h \in P_h, \end{array} \tag{37}$$

where ϵ should be small. With this perturbation of the original problem, the solution is stable without needing to fulfill the Babuska-Brezzi condition. Or equivalently, the perturbation is a mean for avoiding mixed finite elements; standard finite element for $\boldsymbol{v}_h$ and p_h will work. This regularization technique also gives rise to a whole family of solution strategies, in terms of solving the problem (37) for smaller and smaller ϵ. These "Uzawa-type" methods are considered in [5, 10, 4, 11].

Remark: Regularization vs. Condition Number. A serious negative impact of the regularization is that it increases the condition number of the matrix significantly. As Conjugate Gradient-like methods tend to converge more slowly as the condition number increases, it might be challenging to design a good preconditioner and construct efficient iterative methods for the regularaized problem. Nevertheless, the structure of the linear system arising from the original problem discretized by mixed finite elements also poses challenging demands on iterative methods.

Regularization of the Mixed Poisson Equation. In the mixed formulation of the Poisson equation the elements must be designed to satisfy both Condition 1 and Condition 2. We can therefore introduce regularization to relax to the requirement of fullfilling the two conditions. Condition 1 can be regularized by adding:

$$\mu(\nabla \cdot \boldsymbol{w}, \nabla \cdot \boldsymbol{v}) = \mu(f, \nabla \cdot \boldsymbol{v}) \tag{38}$$

to 5. This might suggest that we can use Stokes elements and obtain a good approximation to the solution of this regularized equation. However this is not in general true. The strong $H^1(\Omega)$-continuity of the Stokes elements will in general give poor approximation, due to the aforementioned locking phenomenon. We can also regularize property 2. If we apply $\epsilon\nabla\cdot$ on the equation $\boldsymbol{v} + \lambda\nabla p = 0$, we get the equation $\epsilon\nabla \cdot \boldsymbol{v} + \epsilon\lambda\nabla^2 p = 0$. Adding this equation to equation (6) and dividing by $(1-\epsilon)$ gives the weak formulation

$$\begin{aligned} a(\boldsymbol{v}_h, \boldsymbol{w}_h) + b(p_h, \boldsymbol{w}_h) &= 0, \quad \forall \boldsymbol{w} \in \boldsymbol{H}_h, & (39)\\ b(q_h, \boldsymbol{v}_h) - \frac{\epsilon}{1-\epsilon} c(q_h, p_h) &= \frac{1}{1-\epsilon}(g, q_h), \ \forall q_h \in P_h, & (40) \end{aligned}$$

Notice that both these regularization techniques are discretizations of the exact problem for any $\epsilon \geq 0$ and not perturbations of the problem.

3 Some Basic Concepts of a Finite Element

This section presents the basics of a general finite element. We will focus on a definition which should be easy to use in a general implementation. This is needed later when we implement the finite elements appropriate to our model problems. For further reading about the finite element method, see [15, 12, 5, 4]. Detailed information on finite element programming in Diffpack can be found in [16, 13, 15].

3.1 A General Finite Element

Consider a spatially varying scalar field $f(x)$, where $x \in \Omega \subset \mathbb{R}^d$. The field $f(x)$ has the following approximate expansion $\hat{f}(x)$ in a finite element method:

$$f(x) \approx \hat{f}(x) = \sum_{j=1}^{n} \alpha_j N_j(x)\,. \tag{41}$$

If f also depends on time, we write

$$f(x,t) \approx \hat{f}(x,t) = \sum_{j=1}^{n} \alpha_j(t) N_j(x)\,. \tag{42}$$

We refer to $N_j(x)$ as the basis functions in global coordinates and α_j are the *coefficients* in the expansion, or the global degrees of freedom of the descrete scalar field. The choice of basis functions $N_j(x)$ is problem dependent, and we should choose them based on the regularity requirements of the differential equation. The basis functions are piecewise polynomials and we can therefore deduce the following (see [12]):

$$V_h \subset H^1(\Omega) \Leftrightarrow V_h \subset C^0(\Omega), \tag{43}$$

$$V_h \subset L^2(\Omega) \Leftrightarrow V_h \subset C^{-1}(\Omega), \tag{44}$$

where $C^0(\Omega)$ is the space of continuous functions and $C^{-1}(\Omega)$ is the space of discontinuous functions (with finite discontinuities).

Central in the finite element method is the partition of Ω into non-overlapping *subdomains* Ω_e, $e = 1, \ldots, E$, $\Omega = \Omega_1 \cup \cdots \cup \Omega_E$, $\Omega_i \cap \Omega_j = \emptyset$, $i \neq j$. In each subdomain Ω_e we define a set of basis functions with support only in Ω_e and the neighboring subdomains. The basis functions are associated with a number of local degrees of freedom. A subdomain, along with its basis functions and degrees of freedom, defines a *finite element*, and the finite elements throughout the domain define a finite element space. The finite element space has a global number of degrees of freedom. These degrees of freedom can be the values of the unknown functions at a set of points (nodes). From an implementational point of view, it is a great advantage of the finite element method that it can be evaluated locally and thereafter assembled to a global linear system. We therefore focus on a local consideration of the finite element method. Similar definitions of finite elements and more information can be found in [12, 4, 14].

For each Ω_e there is a parametric mapping M_e^{-1} from the physical subdomain to a reference subdomain:

$$\xi = M_e^{-1}(x), \quad x = M_e(\xi), \quad \xi \in \Omega_r \subset \mathbb{R}^d, \quad \boldsymbol{x} \in \Omega_e \subset \mathbb{R}^d\,. \tag{45}$$

The Ω_r domain is often called the *reference domain*. The mapping of this reference domain to the physical coordinate system is defined by a set of geometry functions $G_i(\xi)$, i.e., M_e is defined in terms of some G_i functions. Normally, we associate with term *reference element* the reference domain together with its degrees of freedom and basis functions.

One important class of elements is the *isoparametric elements*. The basis functions in these elements are exactly the geometry functions and the degrees of freedom are represented in the geometry nodes. This means that we get the physical elements (shape in of the element and the expressions for the basis function in physical coordinates) by applying the parametric mapping $M_e(\xi)$ on the reference elements. Non-isoparametric elements might also be defined in terms of a mapped reference element. However, such elements might also have other non-equivalent definitions.

Definition 1 *A reference element is characterized by a reference subdomain Ω_r with a set of n_g points $p_1, \ldots p_{n_g}$, referred to as geometry nodes, such that the mapping M_e from the reference domain Ω_r onto the corresponding physical domain Ω_e has the form*

$$x(\xi) = \sum_{j=1}^{n_g} G_j(\xi) x^{(j)}, \tag{46}$$

where $x^{(j)}$ are the coordinates in physical space corresponding to the geometry node p_j. Moreover, the geometry function $G_j(\xi)$ has the property $G_j(p_i) = \delta_{ij}$, where δ_{ij} is the Kronecker delta. The (transposed) Jacobi matrix element J_{ij} of the mapping M_e is then given by

$$J_{ij} = \sum_{k=1}^{n_g} \frac{\partial G_k(\xi)}{\partial \xi_i} x_j^{(k)},$$

where $x_j^{(k)}$ is the j-th component of $x^{(k)}$.

If the element is non-isoparametric we must also specify the basis functions. We define a number n_{bf} of basis functions $N_1^{ref}, \ldots N_{n_{bf}}^{ref}$ with associated nodes at points $q_1, \ldots q_{n_{bf}}$ in the reference domain. These may in principle be chosen arbitrary, designed for different purposes. A global basis function is then defined in terms of a mapping of the corresponding reference basis function. If the geometry mapping M_e is appropriate we get the following expression for the global basis function;

$$N_i^{glob}(x) = N_i^{ref}(M_e^{-1}(x)) = N_i^{ref}(\xi).$$

Definition 2 *The restriction of a finite element scalar field to a finite element in the reference domain is written as*

$$f(x)|_{\Omega_r} = \sum_{j=1}^{n_{bf}} \beta_j N_j(\xi), \quad \xi = (\xi_1, \ldots, \xi_d) \in \Omega_r \, .$$

where β_j are the local expansion coefficients, corresponding to the local degree of freedom. $N_j(\xi)$ are the local basis functions and n_{bf} is the number of local basis functions.

Finite element applications usually require the derivatives with respect to the physical (global) coordinates, $\partial N_i/\partial x_j$. The relation between derivatives with respect to local and global coordinates is given by

$$\frac{\partial N_i}{\partial \xi_j} = J_{ij} \frac{\partial N_i}{\partial x_j},$$

where J_{ik} is the transpose of the Jacobian of the mapping M_e.

3.2 Examples of Finite Element Spaces

3.2.1 Illustrating the Notation for Some Known Elements

To clarify the notation introduced n the previous section we apply it to some well-known finite elements.

The Linear Triangle. The popular linear 2D triangular element has

$$\Omega_r = \{(\xi_1, \xi_2) \mid 0 \leq \xi_1 \leq 1,\ \xi_2 \leq \xi_1\}$$

Moreover, $n_g = n_{bf} = n_b = 3$, $p_i = q_i$, $G_i = N_i$ and

$$\begin{aligned} N_1(\xi_1, \xi_2) &= 1 - \xi_1 - \xi_2 \\ N_2(\xi_1, \xi_2) &= \xi_1 \\ N_3(\xi_1, \xi_2) &= \xi_2 \end{aligned}$$

The geometry and basis function nodes are the corner points of the triangle; $p_1 = (0, 0)$, $p_2 = (1, 0)$ and $p_3 = (0, 1)$.

The 2D Piecewise Constant Element. The 2D piecewise constant element over quadrilaterals is defined by $\Omega_r = [-1, 1]^2$, $n_b = n_{bf} = 1$, $n_g = 4$, $N_1(\xi_1, \xi_2) = 1$, while G_i and p_i are identical to the expressions one has for standard bilinear elements. The location of the basis function node is arbitrary, but an obvious choice is $q_1 = (0, 0)$.

Traditional Mixed Elements for the Navier-Stokes Equations. When solving Navier-Stokes-type of equations for the pressure and velocity fields, mixed finite element methods are traditionally used. A possible choice is to let the geometry be described by eight or nine nodes in a quadrilateral reference element. The velocity components may then use basis functions that coincide with the geometry functions ($n_g = n_b = n_{bf}$, $q_i = p_i$ and $N_i = G_i$). The pressure field, on the other hand, is based on an element where the geometry is the same as for the velocity field, but where the basis functions are bilinear. For such an element, we see that the basis function nodes are identical to a subset of the geometry function nodes.

3.2.2 Elements Suited for Our Model Problems

As we stated in Section 2, our model problems require delicate combinations of finite element spaces, i.e they must satisfy the Babuska-Brezzi condition (2). Some appropriate mixed elements are presented here and we also discuss the implementation of these. More elements can be found in [11, 5]. We only consider conforming approximation.

Definition 3 *A finite element space V_h is a conforming approximation of V if it is a subspace of V. A non-conforming finite element space approximation of V is an outer approximation, that is, the finite element space is not a subspace of V.*

Details on conforming and nonconforming elements can be found in [4, 5]. Note that the mixed formulation of the Poisson equation is often referred to as a nonconforming approximation of the pressure. The usual formulation of the Poisson equation $-\nabla \cdot (\lambda \nabla p) = f$ require that $p \in H^1(\Omega)$, while in the mixed formulation the pressure usually is in $L^2(\Omega)$. However, we will seek a conforming method for the mixed formulation. That is $\boldsymbol{H}_h \subset \boldsymbol{H}$ and $P_h \subset P$.

Definition 4 *An isoparametric element is a finite element for which the basis functions and the geometry functions coincide. This implies that $p_i = q_i$ and $n_g = n_{bf} = n_b$. In other words, the mapping M_e has the same form as the finite element basis functions (hence the name isoparametric mapping).*

Often one finds conforming and non-conforming elements as the seemingly similar terms as isoparamteric and non-isoparameteric elements in the literature. However, an element frequently used for solving Navier-Stokes equations has geometry functions identical to a multi-quadratic element while the basis functions are the same as in the corresponding multi-linear element.

This element is conforming, but not isoparametric. Non-conforming elements will always be non-isoparametric in our terminology, but the converse is generally not true.

Mixed Finite Elements for the Poisson Equation. A popular choice of elements is the Raviart-Thomas elements of order 0 for the velocity approximation $\boldsymbol{v}_h$ and discontinuous piecewise constants for p_h. The Raviart-Thomas are designed to approximate $\boldsymbol{H}(div;\Omega)$. They are continuous only in the outward normal direction across the elements boundaries. A more detailed discussion of properties in $\boldsymbol{H}(div;\Omega)$ can be found in [**?**, 5]. The Raviart-Thomas elements of order 0 are on the form:

$$\boldsymbol{N}_i = \begin{pmatrix} a+dx \\ b+dy \\ c+dz \end{pmatrix} \text{ in 3D and } \boldsymbol{N}_i = \begin{pmatrix} a+dx \\ b+dy \end{pmatrix} \text{ in 2D}\,. \tag{47}$$

The constants a, b, c, d are determined such that

$$\boldsymbol{N}_i \cdot \boldsymbol{n}_j = \begin{cases} 1 & if\ i = j \\ 0 & if\ i \neq j\,. \end{cases} \tag{48}$$

where $\boldsymbol{N}_i$ are the basis functions at side i and $\boldsymbol{n}_j$ are the normal vectors at side j. This yields a global definition of the elements. We wish to define these elements in terms of a mapped reference element, because this is a very flexible strategy. We use the definition above in the reference element and map the reference element to the global element. However, the usual geometry mapping M_e in terms of the basis functions does *not* preserve the continuity in the normal direction and the mapped elements will therefore not be in $\boldsymbol{H}(div;\Omega)$. A slightly modified mapping does preserve the continuity in the normal direction, but unfortunately the mapping can not be defined componentwise. We remember the geometry mapping:

$$x = M_e(\xi)\,.$$

In order to make elements that are continuous in the normal direction out of mapped reference element, we must assume that the geometry mapping is affine. Affine mappings can be expressed by

$$x = M_e(\xi) = x_e + B_e\xi$$

for some matrix B_e and an a fixed point x_e in physical space. The mapped reference element can then be defined as:

$$\boldsymbol{N}(x) = \frac{1}{\det B_e} B_e \cdot \boldsymbol{N}^{ref}(M_e^{-1}(x)) \,. \tag{49}$$

The reference element is shown in figure 1 and we refer to [17, 5] for more material. As mentioned above the continuity across elements is only in the normal direction on the element boundary. This means that the interpolation operator is only well defined along the normal direction on the boundary of the element. In fact these elements are defined so that such an interpolation operator is well defined. Properties related to the mapping and the interpolation operator can be found in [18].

These mixed finite elements are known to give a stable approximation of the mixed formulation of the Poisson equation in the sense that:

$$\|\boldsymbol{v}_h\|_{\boldsymbol{H}(div;\Omega)} + \|p_h\|_{L^2(\Omega)} \leq C\|f\|_{\boldsymbol{L}^2(\Omega)} \,. \tag{50}$$

We also have the following approximation properties:

$$\|\boldsymbol{v} - \boldsymbol{v}_h\|_{L^2(\Omega)} \leq \|\boldsymbol{v} - \Pi\boldsymbol{v}\|_{L^2(\Omega)} \leq ch^k \|\boldsymbol{v}\|_{H^k(\Omega)} \; k = 1, 2, ..., r+1, \tag{51}$$

$$\|p - p_h\|_{L^2(\Omega)} \leq ch^k \|p\|_{H^k(\Omega)}, \quad k = 1, 2, ..., r+1, \tag{52}$$

$$\|p - p_h\|_{L^2(\Omega)} \leq c(h\|p\|_{H^1(\Omega)} + h^2\|p\|_{H^2(\Omega)}) \,. \tag{53}$$

More details on this can be found in [2, 17, 3].

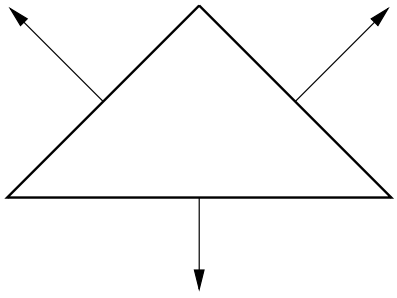


Figure 1: Sketch of the 2D Raviart-Thomas element for the mixed formulation of the Poisson equation.

This combination of finite element spaces are often used in simulators approximating the mixed Poisson equation. However, we can easily see that these elements are not conforming elements for the Stokes problem. In Stokes problem we seek a subspace of $\boldsymbol{H}^1(\Omega)$ and this means that the

space must consist of continuous (C^0) functions. The Raviart-Thomas elements are continuous only in the normal direction, which means that they are not in $\boldsymbol{H}^1(\Omega)$. The next example is a conforming stable approximation of Stokes problem.

Mixed Finite Elements for the Stokes Problem. The velocity elements must consist of continuous piecewise polynomials as we seek a finite dimensional subspace of $\boldsymbol{H}^1(\Omega)$. The pressure finite elements only need to be in $L^2(\Omega)$ and can therefore be discontinuous. The simplest choice is then linear velocity elements and constant pressure elements. From the locking phenomena in elasticity theory we know that first order polynomials do not approximate divergence free functions very well. The only divergence free solution will in general be $\{0\}$. However, if we assume that we have a rectangular grid where the sides of the elements are parallel to the axis, we can choose continuous biquadratic velocity and continuous linear pressure (more regularity then strictly needed of the pressure). This combination does satisfy the Babuska-Brezzi condition (2) and we end up with the following approximation result (see [9]).

$$\|\boldsymbol{v} - \boldsymbol{v}_h\|_{H^1(\Omega)} + \|p - p_h\|_{\boldsymbol{L}^2(\Omega)} \leq Ch(\|\boldsymbol{v}\|_{H^1(\Omega)} + \|p\|_{\boldsymbol{L}^2(\Omega)}) \,. \tag{54}$$

These elements are implemented in Diffpack as `ElmB9n2D` (continuous biquadratic polynomials) and `ElmB9gn4bn2D` (continuous linear pressure). This poor approximation $\mathcal{O}(h)$ is related to the coarseness of the pressure elements.

A better approximation can be obtained by using the so-called Mini elements, with continuous linear elements for pressure and velocity. This combination does not satisfy the Babuska-Brezzi condition, and we must hence enrich the velocity space. This can be done with the so-called "bubble" functions defined as follows. Let the triangle be numbered such that e_i are the edges at the opposite side of the vertex v_i. Let λ_i be a linear function such that $\lambda_i(v_i) = 1$, while $\lambda_i(x_i) = 0 \ \forall x_i$ on e_i. The function $\lambda_1\lambda_2\lambda_3$ is the so called bubble function. The bubble function is zero on the edge of the element and has therefore support only in one element. If we enrich the velocity space with the bubble functions then the mixed finite elements satisfy the Babuska-Brezzi condition (2). The degrees of freedom are the nodes at the vertices and the center of the triangle. The element is available in Diffpack as `ElmT3gn4bn2D` and `ElmT4gn5bn3D`.

We have the following error estimate (see [11]) for the Mini element:

$$\|\boldsymbol{v} - \boldsymbol{v}_h\|_{\boldsymbol{H}^1(\Omega)} + \|p - p_h\|_{L^2(\Omega)} \leq C_1 h(\|\boldsymbol{v}\|_{\boldsymbol{H}^2(\Omega)} + \|p\|_{H^1(\Omega)}), \tag{55}$$

if $\boldsymbol{v} \in \boldsymbol{H}^2(\Omega) \cap \boldsymbol{H}_0^1(\Omega)$ and $p \in \boldsymbol{H}^1(\Omega) \cap L_0^2(\Omega)$ and Ω is a bounded plane polygon with regular triangulation. If Ω is convex we also have the following estimate

$$\|\boldsymbol{v} - \boldsymbol{v}_h\|_{\boldsymbol{L}^2(\Omega)} \leq C_2 h^2 (\|\boldsymbol{v}\|_{\boldsymbol{H}^2(\Omega)} + \|p\|_{H^1(\Omega)}) \,. \tag{56}$$

We note that the approximation results rely on the linear polynomials, whereas the bubble functions give stability.

3.3 Diffpack Implementation.

The definition of the reference element is provided by a subclass of `ElmDef`. This definition is based on parameters (variables) in the base class `ElmDef` as well as virtual functions defined in `ElmDef`. For example, n_b, n_{bf} and n_g are integers in class `ElmDef` having the names `nne_basis`, `nbf` and `nne_geomt`, respectively. The virtual function `geomtFunc` defines the geometry functions $G_i(\xi)$, while the virtual function `basisFunc` defines the basis functions $N_i(\xi)$. The derivatives $\partial N_i/\partial \xi_j$ and $\partial G_i/\partial \xi_j$ are provided by the virtual functions `dLocBasisFunc` and `dLocGeomtFunc`, respectively. Notice that the derivatives refer to the reference coordinates ξ. It is the derivatives with respect to the physical coordinates that are of interest. Class `BasisFuncAtPt` is designed for evaluating and storing the geometry and basis functions, their global derivatives and the (transposed) Jacobi matrix determinant, at a particular ξ point. This class relies on information from `ElmDef` and the global coordinates of the geometry nodes of the element.

Class `FiniteElement` is designed to offer the programmer easy access to the global element: the basis functions, their global derivatives, and the Jacobi determinant, evaluated at a point ξ in the reference domain Ω_r (corresponding to a physical point $\boldsymbol{x}$ in Ω_e). Class `FiniteElement` is naturally based on a layered design where it gains its information from `BasisFuncGrid`, `ElmDef`, `BasisFuncAtPt` and `GridFE` objects. In case of mixed finite elements, one needs a `FiniteElement` object for each type of basis functions. This is provided by the class `MxFiniteElement`, which contains an array of pointers (handles) to `FiniteElement` objects and much of the same interface as class `FiniteElement`. Scalar finite element fields are represented by `FieldFE` and rely on information from `FiniteElement`, `BasisFuncGrid` and `GridFE` objects. `FieldFE` objects are designed to represent the solution field and can be evaluated and differentiated at arbitrary points.

The initial version of Diffpack was written for isoparametric elements. Hence, most of the virtual functions in the `ElmDef` have a default implemen-

tation in class `ElmDef` for isoparametric elements (if the function will depend on the element shape, the default version assumes a box shape).

The Raviart-Thomas element for the mixed formulation of the Poisson equation is built using some subclasses of `ElmDef`. The 2D version of this element has the velocity components defined by `ElmT3gn3bn2Du` and `ElmT3gn3bn2Dv`. The corresponding pressure element is implemented as `ElmT3gn1bn2D`. As the velocity elements are discontinuous in the nodal points, they require special interpolation and projection operators (or from the historical point of view, this element were made to fulfill certain interpolations properties). Class `MxProj` offers the projections operators.

The information on the elements, in particular the geometry nodes, their connectivity and boundary indicators [13, 7], is represented by class `GridFE` in Diffpack. Class `BasisFuncGrid` contains the basis function nodes, their connectivity, the boundary indicators at basis function nodes etc. In addition, class `BasisFuncGrid` has a `GridFE` object (or rather a pointer or handle to such an object) such that a `BasisFuncGrid` object has complete information of all the geometry and basis function nodes and their relevant associated data.

When working with isoparametric elements, class `BasisFuncGrid` simply uses the `GridFE` object for looking up information on basis function nodes and degrees of freedom in scalar fields. For a user it is then only necessary to create a `GridFE` object. When other classes are initialized with a `GridFE` object, and they need information on the basis functions (class `FieldFE` is an example), it is assumed that the elements are isoparametric and a `BasisFuncGrid` object is easily constructed internally in these classes. None of the terms related to the distinction between basis and geometry functions need to be familiar to the user in the isoparametric case. In particular, class `BasisFuncGrid` is not apparent at all, see for example [13].

If non-isoparametric elements are applied, the user must allocate and initialize a `BasisFuncGrid` for each scalar field. In this way, the extra complexity associated with the details of non-isoparametric elements is only visible when it is really needed. One should notice that this design goal is readily achieved due to our usage of abstract data types and object-oriented programming.

The `GridFE` and `BasisFuncGrid` classes organize the *global* topology of the geometry and basis function information. The (local) definition of the basis functions are provided by an `ElmDef` subclass.

4 Solution Strategies

Having performed the mixed finite element discretization, we end up as usual with a system of linear algebraic equations. The book-keeping of element degrees of freedom and linear system degrees of freedom is non-trivial in mixed methods and is a major subject of the present section. The nature of the degrees of freedom numbering in the linear system depends on the type of solution methods that one intends to use. It is therefore natural to divide the discussion into a two parts; one for direct methods and one for iterative methods.

4.1 Direct Methods

When applying direct methods like Gaussian elimination for solving large systems of linear equations, it is advantageous to keep the bandwidth of the coefficient matrix as small as possible. The bandwidth depends on the nodal numbering and the ordering of the unknowns. Some examples will illustrate this point, but first we define two basic concepts: the *special* and the *general* numbering.

The special numbering. It is assumed that isoparametric elements are used and that there are m unknowns per node and n nodes. In other words, one needs to use the same element type for all the unknown fields that enter a system of partial differential equations. Problems where this is a suitable approach involve the Navier equations of elasticity, the incompressible Navier-Stokes equations treated by a penalty function approach, and simultaneous (implicit) solution of pressures and concentrations (saturations) in multi-phase porous media flow. The geometry and basis function nodes coincide, and it is natural to number the degrees of freedom of the linear system in this sequence:

$$u_1^{(1)}, u_1^{(2)}, \ldots, u_1^{(m)}, u_2^{(1)}, \ldots, u_n^{(1)}, \ldots, u_n^{(m)}, \tag{57}$$

where $u_i^{(j)}$ is the degree of freedom of scalar field no. j at node i. Given a global node i and a scalar field number j, the global degree of freedom number is $m(i-1)+j$. The `DegFreeFE` class in Diffpack takes care of computing the global degree of freedom number according to this formula.

At the element level, the structure of the special numbering is the same as at the global level. That is, if i is a local node, j is the field number, the associated *local* degree of freedom of the merged fields is $m(i-1)+j$. This information is very important since the local numbering is fundamental when

setting up the elemental matrix and vector contributions in the `integrands` functions in Diffpack. The book [14] contains some relevant examples in chapters 5.1, 6.3, and 7.2.

As an example on the special numbering, consider the 2×2 grid of bilinear

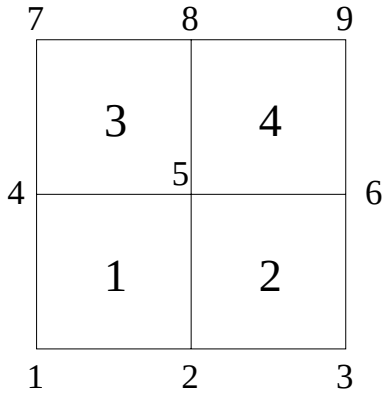


Figure 2: Sketch of a 2×2 grid, with bilinear elements, and the corresponding numbering of elements and nodes.

elements in Figure 2. If there is only one scalar field, the degrees of freedom numbering in the linear system naturally follows the nodal numbering. A measure of the associated matrix bandwidth could be the largest difference between two degree freedom numbers in the same element. Here this is 4 (e.g. $5 - 1 = 4$ in the first element). With two unknown scalar field, $u^{(1)}$ and $u^{(2)}$, there are two ways of structuring the degrees of freedom in the linear system. The suggestion above results in

$$u_1^{(1)}, u_1^{(2)}, u_2^{(1)}, u_2^{(2)}, \ldots, u_n^{(1)}, u_n^{(2)} \tag{58}$$

and depicted in Figure 3. Instead of numbering the local degrees of freedom at each node consecutively, we could first number all the degrees of freedom of scalar field one and then number the degrees of freedom of scalar field two:

$$u_1^{(1)}, u_2^{(1)}, \ldots, u_n^{(1)}, u_1^{(2)}, \ldots, u_n^{(2)} \tag{59}$$

The numbering on a 2×2 grid is displayed in Figure 4. The impact of the two numberings on the bandwidth of the coefficient matrix should be clear: 9 in the first case and 13 in the second case. On a grid with $q \times q$ elements the corresponding figures read $2q + 5$ and $q^2 + 3q + 3$!

The general numbering. The general numbering is based on the following strategy. At the element level, a field-by-field numbering like (59) is used

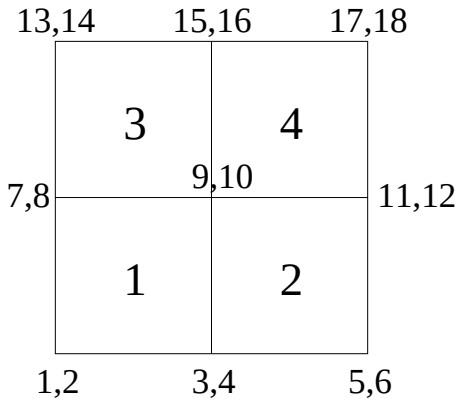


Figure 3: The special numbering of degrees of freedom in the linear system arising from two unknown scalar fields over the grid in Figure 2.

for the element degrees of freedom,

$$u_1^{(1)}, u_2^{(1)}, \ldots, u_n^{(1)}, u_1^{(2)}, \ldots, u_n^{(2)}, \ldots, u_1^{(m)}, \ldots, u_n^{(m)}$$

where now n is the number of nodes in the element and m is the number of unknown scalar fields. More generally, if we have n_j degrees of freedom for field no. j, we order the unknowns like this:

$$u_1^{(1)}, u_2^{(1)}, \ldots, u_{n_1}^{(1)}, u_1^{(2)}, \ldots, u_{n_2}^{(2)}, \ldots, u_1^{(m)}, \ldots, u_{n_m}^{(m)} .$$

The corresponding *global* numbering is constructed from a simple (and general) algorithm that yields a reasonable small bandwidth: For each element we run through each local degree of freedom and increase the corresponding global number by one, provided the local degree of freedom has not been given a global number in a previously visited element. Such numbering of the degrees of freedom applied to a single scalar field is exemplified in Figure 5. In element 1 we go through the local nodes 1-4 and assign corresponding global degree of freedom numbers 1-4. In element 2, the first local degree of freedom was already treated in element 1. The next local degree of freedom (local node no. 2) has not been treated before and can therefore be given the global degree of freedom number 5. The reader is encouraged to continue the algorithm and understand how the rest of the degree of freedom numbers arise.

Let us extend the general number example on a 2×2 grid to the case where we have two scalar fields as unknowns. In the first element we then run through the local degrees of freedom 1-4 of the first scalar field and generate corresponding global numbers 1-4. Then we run through the four degrees

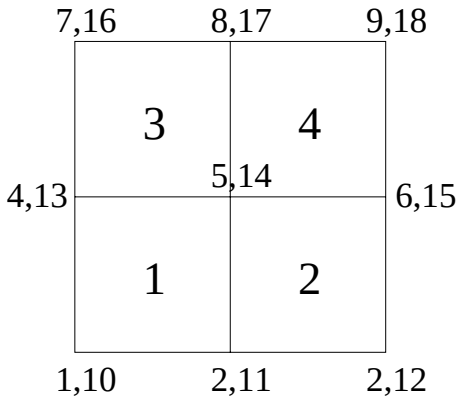


Figure 4: A possible numbering of degrees of freedom in the linear system arising from two unknown scalar fields over the grid in Figure 2.

of freedom of the second scalar field and assign them to the global numbers 5-8. Proceeding with element two, the two nodes on the left have already been treated in element 2 so the second degree of freedom of scalar field no. 1 is assigned the global number 9, while the fourth degree of freedom of scalar field no. 1 corresponds to the global number 10. The second scalar field contributes with two new degrees of freedom, 11 and 12. Also in this case the reader should understand the rest of the global degree of freedom numbers. Figure 6 shows the numbers.

Finally, we consider an example involving different number of degrees of freedom in different fields. Again we focus on the grid in Figure 2, but now we have two scalar fields with bilinear elements and one scalar field with piecewise constant elements. The location of the degree of freedom for the constant value in an element could be the centroid. (This example could correspond to bilinear elements for a vector field and piecewise constant elements for a scalar field.) Figure 7 shows the complete numbering.

As one can see, the general numbering requires quite some book-keeping. This is performed by the `DegFreeFE` object. However, the programmer must explicitly deal with the *local* degrees of freedom numbering when setting up the element matrix and vector, but this is quite simple, as one applies either the special numbering or the field-by-field numbering on the element level. The complicated details arise when going from the element to the global degrees of freedom numbering, but class `DegFreeFE` hides the book-keeping from the programmer. Some program examples appear later and illustrates the usage of various Diffpack tools.

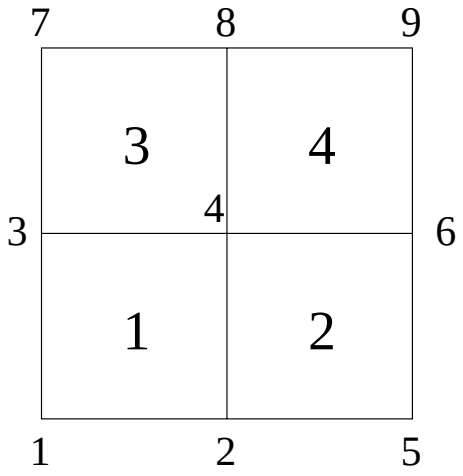


Figure 5: The general numbering applied to a single scalar field over the grid in Figure 2.

4.2 Iterative Methods

Our model problems in this report are fundamental problems in engineering and science and they are needed to be solved in real-life situations, that is very large scale simulations. In such situations the algorithm for solving the matrix equation is of vital importance. When solving a sparse matrix equation with a direct solver like Gaussian elimination the sparsity pattern of the matrix will be destroyed by fill-in values. This leads to unacceptable memory requirement. Banded Gaussian elimination is also a very slow algorithm which typically needs $\mathcal{O}(n^{7/3})$ operations (with a suitable numbering the bandwidth in 3D is $n^{2/3}$). This leads us to the study of iterative methods that need less memory and computing power. One family of methods that is particularly interesting is the preconditioned Krylov solvers, which we will focus on in this section.

Other iterative methods that have been suggested are the "Uzawa-type" methods. We refer to [10] for a comprehensive treatment of these algorithms. They are iterative methods with an outer and inner iteration. The problem with these methods is that it might be necessary to iterate the inner iteration until it converges within computer precision, which makes the process expensive. There are also certain input parameters that need to be chosen. The Uzawa method and the generalized Augmented Lagrangian method are also somewhat Stokes specific. We will therefore consider the more general block preconditioned Krylov solvers. However, the "Uzawa-type" methods can easily be implemented within the setup presented.

There has been developed a rich set of Krylov solvers for different pur-

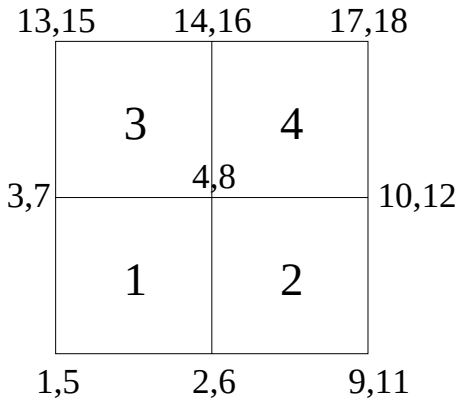


Figure 6: The general numbering applied to two scalar fields over the grid in Figure 2.

poses. The reader is referred to [6] for a general discussion of Krylov solvers and solvers in general. We only make some general remarks on these methods in the context of our model problems. Both our model problems are indefinite and symmetric problems, i.e. having both positive and negative eigenvalues, and we must use a method that can handle such situations[1]. Diffpack already has appropriate solvers for symmetric and indefinite systems: `Symmlq` and `MinRes` (see [8]). The basic operations in the Krylov solvers are matrix-vector product, vector addition and inner products, which all are of $\mathcal{O}(n)$ operations (as we have very sparse matrices). The memory requirement is $\mathcal{O}(n)$ entries of memory, since we only need to store the matrix and some vectors. (Non-symmetrical solvers need to orthogonalize the search vectors, and this operation is of order $number\ of\ iterations \times n$. However, efficient preconditioning can bound the number of iterations.)

Our ultimate goal is to use preconditioned Krylov solvers that will solve the problem in a given number of iterations, independent of the grid size h. The following are typical demands for an efficient iterative solving algorithm:

- The number of operations in each iteration should be proportional to the number of unknowns.

- The number of iterations to reach convergence should be bounded independently of the mesh.

[1] Recall that e.g. the Conjugate Gradient method requires a positive definite coefficient matrix, which appears for the standard finite element formulation of the Poisson equation and many other problems.

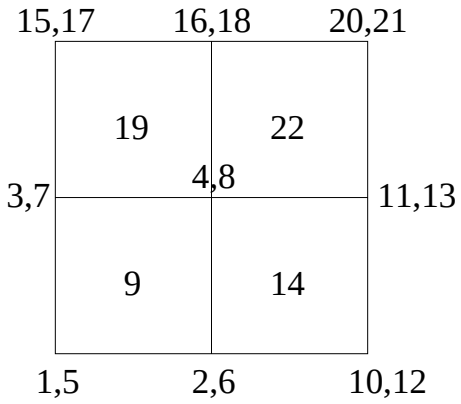


Figure 7: The general numbering applied to two scalar fields with 9 nodes and one scalar field with 4 nodes over the grid in Figure 2.

- The memory requirement should be proportional with the number of unknowns.

Our two discrete model problems can be written as mixed systems.

$$\mathcal{A}_h \left[\begin{array}{c} \boldsymbol{v}_h \\ \boldsymbol{p}_h \end{array} \right] = \left[\begin{array}{cc} \boldsymbol{A}_h & \boldsymbol{B}_h^T \\ \boldsymbol{B}_h & \boldsymbol{0} \end{array} \right] \left[\begin{array}{c} \boldsymbol{v}_h \\ \boldsymbol{p}_h \end{array} \right] = \left[\begin{array}{c} f_h \\ g_h \end{array} \right] . \tag{60}$$

The convergence of an appropriate Krylov solver can be estimated in terms of the condition number of $\mathcal{A}_h$, $\kappa(\mathcal{A}_h)$. The condition number of a general matrix C is $\kappa(C) = \sigma_m/\sigma_1$, where σ_1 is the smallest singular value and σ_m is the largest singular value. We have assumed that $C \in R^{n,m}$, $m \geq n$ and $\text{rank}(C) = n$. The matrix $\mathcal{A}_\langle$ and the blocks $\boldsymbol{A}_h$ and $\boldsymbol{B}_h$ have these properties. The condition number $\kappa(\mathcal{A}_h)$ is governed by the three parameters $\kappa(\boldsymbol{A}_h)$, $\kappa(\boldsymbol{B}_h)$ and the relative scaling between $\boldsymbol{A}_h$ and $\boldsymbol{B}_h$, $\rho(\boldsymbol{A}_h, \boldsymbol{B}_h)$, defined by the ratio between the smallest singular value of $\boldsymbol{B}_h$ and the largest of A_h (see [19]). However, differential operators are unbounded and the condition number increase with finer grid. In order to reduce the condition number we consider the following system:

$$\mathcal{B}_h \mathcal{A}_h \left[\begin{array}{c} \boldsymbol{v}_h \\ \boldsymbol{p}_h \end{array} \right] = \mathcal{B}_h \left[\begin{array}{c} f_h \\ g_h \end{array} \right] , \tag{61}$$

where $\mathcal{B}_h$ should be a well designed operator.
The above mentioned requirement can then be stated as properties of the so-called preconditioner $\mathcal{B}_h$.

1. $\mathcal{B}_h$ should be evaluated in $\mathcal{O}(n)$ operations.
2. $\mathcal{B}_h$ should be spectrally equivalent with ${\mathcal{A}_h}^{-1}$.
3. the memory requirement of $\mathcal{B}_h$ should be proportional with the number of unknowns.

An alternative statement of the second property means is that, roughly speaking, $\mathcal{B}_h\mathcal{A}_h$ is close to the identity operator (which imples fast convergence of an iterative method).
How should this preconditioner be constructed? Such systems have been studied in [19] and we follow their conclusion. The best preconditioner is of course ${\mathcal{A}_h}^{-1}$, which is too costly to be used as preconditioner. Our mixed system is indefinite, but we see that if we use the best preconditioner possible, ${\mathcal{A}_h}^{-1}$, we get a positive definite system. Hence a natural question to ask, is whether the preconditioned system should be indefinite or definite. We seek an "approximation of the inverse" and may not know exactly whether the preconditioned system will be indefinite or definite. Hence we can have eigenvalues near zero, causing a breakdown of the the basic iterative method. The cure is to make a preconditioner such that the preconditioned system is a saddle point problem. Guided by [19] we consider block preconditioners on the form $\mathcal{B}_h = diag(\boldsymbol{M}, \boldsymbol{L})$ such that (61) has the following form

$$\mathcal{B}_h\mathcal{A}_h \begin{bmatrix} \boldsymbol{v}_h \\ \boldsymbol{p}_h \end{bmatrix} = \begin{bmatrix} \boldsymbol{M}\boldsymbol{A}_h & \boldsymbol{M}\boldsymbol{B}_h^T \\ \boldsymbol{L}\boldsymbol{B}_h & \boldsymbol{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{v}_h \\ \boldsymbol{p}_h \end{bmatrix} . \tag{62}$$

A good preconditioned system will then be:

$$\mathcal{B}_h\mathcal{A}_h \begin{bmatrix} \boldsymbol{v}_h \\ \boldsymbol{p}_h \end{bmatrix} = \begin{bmatrix} \boldsymbol{I} & \boldsymbol{Q}^T \\ \boldsymbol{Q} & \boldsymbol{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{v}_h \\ \boldsymbol{p}_h \end{bmatrix}, \text{ where } \boldsymbol{Q}\boldsymbol{Q}^T = \boldsymbol{I} . \tag{63}$$

By a "good" preconditioner we mean in this context a system that has eigenvalues sufficiently away from zero (here the eigenvalues are $1, 1/2 \pm 1/2\sqrt{5}$) to avoid breakdown of iterative schemes. To construct a similar system we must have block operators such that $\boldsymbol{M}\boldsymbol{A}_h \sim \boldsymbol{I}$ and $\boldsymbol{L}\boldsymbol{B}_h\boldsymbol{M}\boldsymbol{B}_h^T \sim \boldsymbol{I}$. If we are able to construct blocks with these properties we will bound the spectrum of eigenvalues above and away from zero, independent of the mesh size.

4.3 The Linear System Structure of Model Problem 1

The Stokes problem fits elegantly in this setup. We remember the discrete coefficient operator from the Stokes problem:

$$\begin{bmatrix} \boldsymbol{A}_h & \boldsymbol{B}_h^T \\ \boldsymbol{B}_h & \boldsymbol{0} \end{bmatrix} = \begin{bmatrix} -\Delta_h & \nabla_h \\ \nabla_h^T & \boldsymbol{0} \end{bmatrix}. \tag{64}$$

(Notice that the discrete divergence operator is the matrix transpose of the discrete gradient operator, which is apparent from the formulas for $\boldsymbol{B}$.) We obviously choose $\boldsymbol{M}$ as a preconditioner for $\boldsymbol{A}_h^{-1}$ and $\boldsymbol{L}$ remains to be chosen. However, $\boldsymbol{B}\boldsymbol{M}\boldsymbol{B}^T \sim I$ so we choose $\boldsymbol{L} = \boldsymbol{I}$. We are now able to construct an optimal preconditioner for the Stokes problem, if we can construct a good preconditioner for $\boldsymbol{A}_h$. $\boldsymbol{A}_h$ is an elliptic operator and it is well known that multigrid combined with e.g. SSOR, efficiently preconditions elliptic operators (see [20]). We present some numerical experiments of this preconditioner in Section **??**, similar experiments are shown in [19].

4.4 The Linear System Structure of Model Problem 2

We saw in the previous section how the Stokes problem and multigrid fit together and gave an optimal preconditioner. However, the same strategy does not apply directly to the mixed formulation of the Poisson equation. The coefficient operator is:

$$\mathcal{A}_h = \begin{bmatrix} \boldsymbol{I} & \nabla_h \\ \nabla_h^T & \boldsymbol{0} \end{bmatrix}. \tag{65}$$

We remember from Section 2.2 that we have two formulations. The first formulation where the gradient is applied on the pressure, takes a finite dimensional subspace of $\boldsymbol{L}^2(\Omega) \times H^1(\Omega)$ onto its dual space in $\boldsymbol{L}^2(\Omega) \times H^{-1}(\Omega)$. The natural preconditioner is then on the form:

$$\mathcal{B}_h = \begin{bmatrix} \boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{0} & \widehat{\Delta_h^{-1}}, \end{bmatrix} \tag{66}$$

and it is well conditioned by the same arguments as in the previous section. However, we are primarily interested in using discontinuous elements, and Δ_h applied on these pressure elements is not well defined. Guided by [2] we use another strategy. Let X be a shorthand notation for the space $\boldsymbol{H}(div;\Omega) \times L^2(\Omega)$ with X^* as the dual space. If we denote the corresponding finite element spaces X_h and X_h^*, then $\|\mathcal{A}_h\|_{\mathcal{L}(X_h, X_h^*)}$ and $\|\mathcal{A}_h^{-1}\|_{\mathcal{L}(X_h^*, X_h)}$ are bounded independently of h. Constructing $\mathcal{B}_h : X_h^* \to X_h$ such that $\|\mathcal{B}_h\|_{\mathcal{L}(X_h^*, X_h)}$ and $\|\mathcal{B}_h^{-1}\|_{\mathcal{L}(X_h, X_h^*)}$ are bounded uniformly in h we can bound

the eigenvalues of $\mathcal{B}_h\mathcal{A}_h : X_h \to X_h$ such that a Krylov solver is efficient (see [2, 1]). We therefore choose a preconditioner on the form

$$\begin{bmatrix} \Theta & \mathbf{0} \\ \mathbf{0} & \boldsymbol{I} \end{bmatrix}, \tag{67}$$

where Θ is the preconditioner of the $\boldsymbol{H}(div;\Omega)$-inner product,

$$(\boldsymbol{v},\boldsymbol{w})_{\boldsymbol{H}(div;\Omega)} = (\boldsymbol{v},\boldsymbol{w})_{\boldsymbol{L}^2(\Omega)} + (\nabla\cdot\boldsymbol{v},\nabla\cdot\boldsymbol{w})_{L^2(\Omega)}\,.$$

Hence the preconditioner for this model problem is based on another operator than the blocks in the mixed system. This operator must be made in addition to the mixed system, which leads to a greater memory requirement in the computation process. The preconditioner can be made without reference to the detailed structure of $\mathcal{A}$. It is only dependent on the norm in X. To find an efficient preconditioner for the $\mathbf{H}(div)$ inner product is unfortunately not as easy as for elliptic operators. To see this we consider the differences between the $\boldsymbol{H}(div;\Omega)$ inner product and the $\boldsymbol{H}^1(\Omega)$ inner product. The $\boldsymbol{H}(div;\Omega)$ inner product is (apart from surface integral terms) the weak form of the differential operator:

$$\boldsymbol{I} - \text{grad div}, \tag{68}$$

which should not be confused with the $I -$ div grad operator ($\nabla\nabla\cdot$ versus $\nabla\cdot\nabla$). The following identity explains the differences between standard elliptic operators and the operator in (68).

$$\Delta = \text{grad div} - \text{curl curl}\,. \tag{69}$$

See [11] for more about this identity and properties related to the divergence and curl operators.

Since the divergence of a curl field and the curl of a gradient field both vanish, the operator behaves completely differently applied to these different fields. The $I -$ grad div applied to a curl field coincides with the identity. On the other hand the $\boldsymbol{I} -$ grad div operator applied to a gradient field coincides with the $\boldsymbol{I} - \Delta$ and is therefore elliptic[2] on the gradient field. It is a degenerated elliptic operator. The multigrid algorithm needs a smoother that efficiently smoothes all high frequent errors. However, a highly oscillating curl field is associated with the eigenvalue 1 and therefore the classical smoothers

[2]The operator typically arises from an equation with left-hand side $u - \Delta u$.

like SSOR do not work at all. Efficient preconditioners of the $\mathbf{H}(div)$ inner product have been studied in [2, 3]. This preconditioner has been proven to be optimal in the finite element space presented in example 3.2.2. However, similar results should not be expected in any finite element space. The smoother in this case is made by domain decomposition on small subdomains.

4.4.1 Block-Matrices and Numbering Strategies

Diffpack has a general numbering strategy capable of numbering several different fields simultaneously. However, a straightforward use of this numbering (as was explained in Section 4.1) results in a merged matrix where the block structure of the matrix is lost. Iterative methods for the solution of the linear system will normally exploit the block structure of the system. The block structure is particularly important when constructing efficient preconditioners. One could argue that application of mixed finite element methods normally demands to resulting linear system to be available on a block-structured form.

Diffpack has several tools for assembling and solving linear system on block-structured form. Each block represents the coupling of an unknown scalar field and a component of a PDE. When solving the Stokes problem, there are blocks reflecting, for example, the unknown pressure field p in the second component of the equation of motion. The coupling of unknown fields and their degree of freedom numbers in the linear system is handled by a `DegFreeFE` object, and in case of block-structured systems, we use a `DegFreeFE` object for each block matrix. The numbering within a block matrix is taken as the special (i.e. natural) numbering, cf. Section 4.1.

Here is a list of the tools used to facilitate the handling of block-structured systems.

- A `DegFreeFE` object represents the numbering of an (i, j) block (the numbering in block (i, j) is symmetric, i.e., equal to the numbering in block (j, i), such that we need, e.g., six distinct `DegFreeFE` objects for a 2D Stokes problem with 3×3 block matrices),

- Class `MxElmMatVec` holds a matrix of `ElmMatVec` objects, where each `ElmMatVec` object contains a block matrix and vector at the element level.

- Class `SystemCollector` administers the computation of the element block matrices and vectors, as well as the modification of the matrices and vectors due to essential boundary conditions. The class contains

handles to an `MxElmMatVec` object, all the `DegFreeFE` objects, and integrand functors [14] for each block matrix and vector (at the element level). The assembly of element contributions in `LinEqAdmFE::assemble` relies on a `SystemCollector` object.

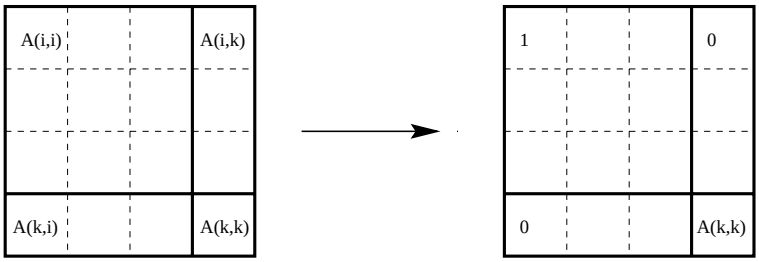


Figure 8: Enforcing boundary condition subject to node i.

Linear systems in Diffpack supports block structures. That is, the classes `LinEqAdm`, `LinEqSolver` and `LinEqSystem` and their subclasses always work on `LinEqMatrix` and `LinEqVector`. `LinEqMatrix` is a `MatSimplest` (usually 1×1) of pointers (handles) to `Matrix` objects. The `LinEqVector` has a similar structure. This means that the Krylov solvers can be used on the mixed system. Indefinite mixed systems are difficult to precondition efficiently if one does not utilize the block structure of the matrix. `LinEqSystemBlockPrec` is developed to extend Diffpack to handle block preconditioners of various forms. `LinEqSystemBlockPrec` is a subclass of
`LinEqSystemPrec` and has the same interface plus some more functions associated with the block preconditioner. We refer to the man page for the full interface. The use of `LinEqSystemBlockPrec` is exemplified in the source code in Section 6.

5 Programming with Mixed Finite Elements in a Simulator

Programming with mixed finite elements is closely related to programming with isoparametric elements. The structure of the class that represents the simulator is the same in the two cases. The basic differences are

- Class `FEM` is replace by its subclass `MxFEM`.
- Class `FiniteElement` is replaced by `MxFiniteElement`, which contains a set of `FiniteElement` objects, each corresponding to the element type of a field involved in the partial differential equations being solved.

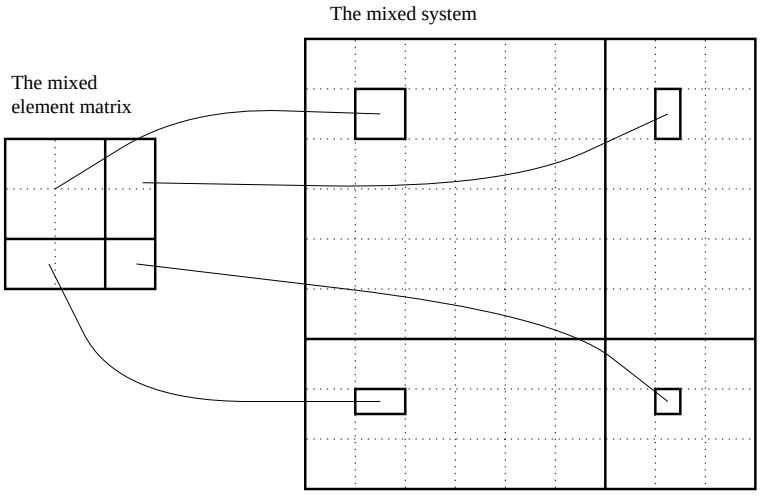


Figure 9: The assemble process of the mixed system.

- `integrands` is replaced by

  ```
  void integrandsMx (ElmMatVec& elmat, MxFiniteElement& mfe)
  ```

- Similarly, `integrands4side` is replaced by `integrands4sideMx`, `makeSystem` is replaced by `makeSystemMx` (with a slightly different argument list).
- The programmer must explicitly declare a `BasisFuncGrid` or

  ```
  VecSimplest(BasisFuncGrid)
  ```

  for each field type that corresponds to a primary unknown.
- All the `BasisFuncGrid` objects must be explicitly created and initialized by the programmer (but the initialization is trivial, just call a function with the element type for each field). The creation of `DegFreeFE`, `FieldFE` and `FieldsFE` objects must be based on a `BasisFuncGrid` object as input rather than on a straight `GridFE` object.

6 Some Code Examples

We will in this section build simulators appropriate for mixed systems, step by step. We begin with a simple demonstration program for the numbering schemes in Section [**?**]. Then show the usage of `BasisFuncGrid` and special vs. general numbering in a `Poisson1`-like (cf. [14]) solver. The next step concerns a simple version of a Stokes-problem solver before we end up with a sophisticated block/multigrid solver for the Stokes problem.

6.1 A Demo Program for Specialized Versus General Numbering

Section 4.1 presents a series of examples, Figures 2–7, on various degree of freedom numbering strategies. Here we shall make a simple demo program that produces these numberings. The program is found in

```
$NOR/doc/mixed/src/numbering/numbering.cpp
```

Assume we have a finite element grid stored in a `GridFE` object `grid`. Making a `DegFreeFE` object directly from the grid, with one unknown per node,

```
DegFreeFE dof (grid, 1);
```

results in a standard numbering of the unknowns, which coincides with the nodal numbering (cf. Figure 2). If we have two fields over the grid, i.e., two unknowns per node, we just alter the second parameter to the `DegFreeFE` constructor:

```
DegFreeFE dof (grid, 2);
```

The resulting degree of freedom numbering correspond to the special numbering (cf. Figure 3).

The general numbering requires a slightly different initialization procedure of the `DegFreeFE` object. Now we must explicitly define a `BasisFuncGrid` for the basis functions over the grid and tie a finite element field to this `BasisFuncGrid` object:

```
BasisFuncGrid f_grid (grid);
FieldFE f (f_grid, "f");
```

Having the field `f` we can initialize the `DegFreeFE` object:

```
DegFreeFE dof (f);
```

The corresponding numbering of the degrees of freedom is shown in Figure 5. Notice that sending a field, instead of a grid, as argument to a `DegFreeFE` constructor implies the general numbering of the degrees of freedom.

With two scalar fields, having the same number of degrees of freedom, the fields must be attached to a `FieldsFE` collector prior to initializing the degree of freedom handler:

```
BasisFuncGrid u_grid (grid);
FieldFE u (u_grid, "u");
FieldFE v (u_grid, "v");
FieldsFE collection (2, "coll");
collection.attach (u, 1);
collection.attach (v, 2);

DegFreeFE dof (collection);
```

The degree of freedom numbering is depicted in Figure 6.

Our final example concerns two fields with isoparametric elements, like `u` and `v` above, plus one field having (possibly) non-isoparametric elements. In other words, we address the general case with several fields and different elements in the different fields. The set-up is the same:

```
BasisFuncGrid u_grid (grid);
FieldFE u (u_grid, "u");
FieldFE v (u_grid, "v");

BasisFuncGrid p_grid (grid);
p_grid.setElmType (p_elm);   // can change to non-isoparametric elm
FieldFE p (p_grid, "p");

FieldsFE collection (3, "coll");
collection.attach (u, 1);
collection.attach (v, 2);
collection.attach (p, 3);

DegFreeFE dof (collection);
```

Figure 7 displays the associated degree of freedom numbering.

6.2 A Standard Solver with Some New Utilities

Consider the `Poisson1` class from [14, ch. 3.1]. Now we change the test problem a bit such that we have a numerical solution that coincides with the exact solution regardless of the number of elements. The test problem for this purpose reads $\nabla^2 u = 0$ in $[0,1]^d$, with $\partial u/\partial n = 0$ on the boundary, except at $x_1 = 0$ where $u = 0$ and at $x_1 = 1$ where $u = 1$. The exact solution is then given as $u(x_1, \ldots, x_d) = x_1$. This should be reproduced by any grid.

The test problem is implemented in class `Laplace1` whose directory is found in `$NOR/doc/mixed/src`. This class is a slight edit of `Poisson1` where we basically have thrown away the flux computations, fixed the domain, and changed the analytical solution and the `fillEssBC` function. Of course, the `define` and `scan` functions are significantly changed.

Our first task is to extend class `Laplace1` to allow for a general node numbering, just for the purpose of explaining how one introduces `BasisFuncGrid` objects in a solver and how the `DegFreeFE` object must be initialized. The following modifications to class `Laplace1` must be performed:

- inclusion of a `BasisFuncGrid` object in the class:

  ```
  Handle(BasisFuncGrid) bgrid;
  ```

- definition of a menu item for the type of numbering

- creation of the unknown field from the `BasisFuncGrid` object[3]:

```
bgrid.rebind (new BasisFuncGrid (*grid));
u.rebind (new FieldFE (*bgrid, "u"));
```

- creation of the `DegFreeFE` object in two ways, depending on whether the user has chosen the special or general numbering:

```
if (numbering == "special")
  dof.rebind (new DegFreeFE (*grid, 1));
else
  dof.rebind (new DegFreeFE (*u));
```

- of reasons to be explained later, the `fillEssBC` function needs to be rewritten

These extra statements are conveniently placed in a subclass of `Laplace1`:

```
class Laplace1GN : public Laplace1
{
public:
  Handle(BasisFuncGrid) bgrid;
  Laplace1GN () : Lanplace1() {}
 ~Laplace1GN () {}
  virtual void define (MenuSystem& menu, int level = MAIN);
  virtual void scan ();
  virtual void fillEssBC ();
};
```

The source code is placed in a subdirectory `Laplace1GN` of `Laplace1`[4].

The redefinition of `fillEssBC` is perhaps not obvious. If we use the familiar constructions for setting boundary conditions, e.g.,

```
if (grid->boNode (i, 1))     // i=geometry node, bo.ind.=1
  dof->fillEssBC (i, 1.0);
```

a wrong solution is computed in the case of a general numbering. The reason is that using the node counter `i` in `dof->fillEssBC(i,1.0)` *implies that the numbering of the unknowns in the linear system coincides with the numbering of the geometry nodes.* Instead of the node number `i` we should use the *global degree of freedom number* corresponding to this node[5]. This number is in general computed by

[3] The field has a `BasisFuncGrid` which contains a `GridFE` object, making the access to all grid information complete from a field object.

[4] We have run `AddMakeSrc ..` to tell the makefile for `Laplace1GN` that the source depends on files in the parent directory.

[5] The numbering of the geometry nodes and the linear system degrees of freedom with a general numbering differ even when there is only one unknown scalar field.

```
idof = dof->fields2dof (i, 1);
```

Moreover, we should for the case of generality in the non-isoparametric case ask `BasisFuncGrid` if a basis function node is subject to the boundary condition:

```
if (bgrid->essBoNode (i, 1))
  dof->fillEssBC (idof, 1.0);
```

With isoparametric elements only this can be simplified to

```
if (grid->boNode (i, 1))
  dof->fillEssBC (idof, 1.0);
```

i.e. asking the geometry nodes for essential boundary conditions. (If there is only one unknown per node, we have `idof=i`, of course.)

The next natural extension of class `Laplace1GN` is to allow for the mixed versions of `FEM` and `FiniteElement`. This extension has no numerical effect, but provides a gentle step for the reader towards a full mixed finite element code. The neccesary steps are listed in Section 5 and implemented in class `Laplace1Mx`. This class is essentially a merge of classes `Laplace1` and `Laplace1GN`, where `FEM` and `FiniteElement` are replaced by `MxFEM` and `MxFiniteElement`. In addition, we need a `FieldsFE` collection of all fields that enter as unknowns in the linear system.

The new parts of the code contain the construction of a `Handle(FieldsFE) collection` object, which is used in many of the mixed finite element utilities:

```
collection.rebind (new FieldsFE (1 /*no of fields*/, "collection"));
collection->attach (*u, 1);
dof.rebind (new DegFreeFE (*collection));
```

The `collection` object is needed when making the linear system:

```
makeSystemMx (*dof, *lineq, *collection, 1);
```

The final argument `1` represents the field number in `collection` whose grid should be used for the numerical integration. (Usually, this should be the field that has the `BasisFuncGrid` with the highest order elements).

The `integrands` routine is replaced by `integrandsMx` and works with an array of `FiniteElement` objects, one entry for each unknown field in the `collector` collection of fields. Here things are as simple as possible, i.e., only one unknown field.

```
void Laplace1:: integrandsMx (ElmMatVec& elmat, const MxFiniteElement& mfe)
{
  int i,j,q;
  if (mfe.size() != 1)
    errorFP("Elliptic7:: integrandsMx", "Wrong size of MxFiniteElement");

  const FiniteElement& fe = mfe(1);
  const real detJxW = mfe.detJxW();
  ...
}
```

The rest of the routine is identical to `Laplace1::integrands`.

6.3 A Simulator for the Stokes Problem

The linear system arising from the Stokes problem can be written on the form (12)–(13). Recall that the ϵ parameter represents the regularization parameter explained in Section 2.3 and that $\epsilon = 0$ in the original Stokes problem.

In our first implementation of Stokes' problem we shall assemble a single merged linear system, suitable for direct methods. The linear system is then not on the form (12)–(13), because the degrees of freedom and equations are merged according to the general numbering, but at the element level, the discrete equations are on the form (12)–(13) with n_v being the number of velocity component nodes and v_p the number of pressure nodes in an element.

The simulator is realized as class `Stokes`, found. Its basic contents are, as usual, handles to a `GridFE`, `LinEqAdmFE`, `DegFreeFE`, `FieldFE`, and `SaveSimRes` objects. In addition we need some new data structures for mixed methods: `BasicFuncGrid` objects for specifying basis function nodes in mixed elements and `FieldsFE` objects for collecting unknown fields for book-keeping of the global set of degrees of freedom in the `DegFreeFE` object. Of pedagogical reasons, especially when we explan block preconditioning of the discrete Stokes system, we introduce explicit 2D code. That is, instead of working with a velocity vector field `v` and its components `v(i)`, we work with scalar components `v_x` and `v_y`. This reduces the amount of indices and details in the code, at the cost of more programming to make the solver work in 3D.

An outline of the class `Stokes` is listed next.

```
class Stokes : public MxFEM
{
protected:
  Handle(GridFE)        grid;
  Handle(BasisFuncGrid) v_x_grid, v_y_grid, p_grid;
  Handle(FieldFE)       v_x, v_y, p; // primary unknowns
  Handle(FieldsFE)      coll;    // (v_x, v_y, p)
  Handle(FieldsFE)      v;       // (v_x, v_y)
```

```
  Handle(DegFreeFE)     dof;
  Vec(real)             linsol;
  Handle(LinEqAdmFE)    lineq;
  Handle(SaveSimRes)    database;
  real epsilon;                  // regularization parameter

  bool p0_given,p1_given;   // turns on/off ess. b.c. for p
  bool dpdx_by_parts;       // indicates type of weak form

  // data structures for computing errors in test problems
  bool p0_given,p1_given;   // turns on/off ess. b.c. for p
  bool dpdx_by_parts;
  int  Vx, Vy, P;           // used for nice indexing

  // used to partition scan into manageable parts:
  void scanGrid();
  virtual void initFieldsEtc();

  // standard functions:
  virtual void fillEssBC ();

  virtual void calcElmMatVecMx
    (int e, ElmMatVec& elmat, MxFiniteElement& mfe);

  virtual void integrandsMx
    (ElmMatVec& elmat, const MxFiniteElement& mfe);

  virtual void integrands4sideMx  // surface integrals in weak form
    (int side, int boind,
     ElmMatVec& elmat, const MxFiniteElement& mfe);

public:
  Stokes ();
 ~Stokes ();

  virtual void adm (MenuSystem& menu);
  virtual void define (MenuSystem& menu, int level = MAIN);
  virtual void scan ();

  virtual void solveProblem ();
  virtual void resultReport ();
};
```

The `coll` field contains all the unknown scalar fields and is required in some mixed finite element routines. The `v` field is simply the velocity field as a *vector field* suitable for dumping to a simres database and used later for visualization purposes.

We allow for setting p boundary conditions and for integrating the ∇p term in the equations by parts or not.

For test purposes we specify $\boldsymbol{v}$ as $\boldsymbol{v} = (1-y^2, 0)$ and $p = 2x - 1$ (channel flow). The exact solution is used to specify essential boundary conditions on the whole boundary. The error should, of course, be zero regardless of the element size.

The first non-trivial part of the code in class `Stokes` concerns initialization of the data structures. First we compute the grid, then the basis function grids, followed by the fields, then the degree of freedom handler, and finially the linear system. The element used in each basis function grid can be set on the menu. A typical initialization of a basis function grid object is then like

```
v_x_grid.rebind (new BasisFuncGrid (*grid));
String v_x_elm_tp = menu.get ("v_x element");
v_x_grid->setElmType (v_x_grid->getElmType(), v_x_elm_tp);
```

with similar code for `v_y_grid` and `p_grid`. The field objects for the velocity components are created from the basis function grids:

```
v_x.rebind (new FieldFE (*v_x_grid, "v_x"));
v_y.rebind (new FieldFE (*v_y_grid, "v_y"));
p.rebind (new FieldFE (*p_grid, "p"));
```

The next step is to create a collector of all unknown scalar fields. For nice indexing of the unknown scalar fields, we introduce literal indices:

```
coll.rebind (new FieldsFE (3, "collector"));
Vx = 1;  Vy = 2;  P = 3;
coll->attach (*v_x, Vx);
coll->attach (*v_y, Vy);
coll->attach (*p, P);
```

A central function is `fillEssBC`. Provided that the velocity components have the same set of basis function nodes, the following code segment sets assigns the essential boundary conditions (by calling up functors for the exact $\boldsymbol{v}$ at the boundary):

```
const int v_nno = v_x_grid->getNoNodes();
for (i = 1; i <= v_nno; i++)
  {
    // is v_x node i subject to any indicator? if so,
    // set the ess. boundary condition according to the
    // analytical sol.

    if (v_x_grid->essBoNode (i)) {
      idof = dof->fields2dof(i, 1);
      v_x_grid->getCoor(x,i);  // or x=v_x_grid->getCoor(i)
      dof->fillEssBC (idof, v_x_anal->valuePt(x));
    }

    // same for v_y
    if (v_y_grid->essBoNode (i)) {
      idof = dof->fields2dof(i, 2);
      v_y_grid->getCoor(x,i);
      dof->fillEssBC (idof, v_y_anal->valuePt(x));
    }
  }
```

The statements should be obvious from the previous material on the `Laplace1GN` and `Laplace1Mx` solvers.

The heart of any Diffpack finite element simulator is the `integrands` routine. In the present case this routine is more complicated than in scalar PDE problems. The discrete equations (12)–(13) interpreted at the element level has the suitable form for direct implementation in an `integrands` routine as we have already stated in Section 4.1 that the numbering of equations and degrees of freedom at the element level is always of the field-by-field form. That is, the element equations consists of n_v equations from the x components of the equation of motion, then n_v equations from the y component (these two correspond to $r = 1$ and $r = 2$ in (12)). The last n_p equations stem from the continuity equation (13). The unknowns consists of the n_v nodal values of `v_x`, followed by n_v nodal values of `v_y` followed by n_p nodal values of `p`.

The element equations can be partitioned like this:

$$\begin{pmatrix} A & 0 & B^{(1)} \\ 0 & A & B^{(2)} \\ (B^{(1)})^T & (B^{(2)})^T & \epsilon M \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ p \end{pmatrix} = \begin{pmatrix} c^{(1)} \\ c^{(2)} \\ \epsilon\pi \end{pmatrix}$$

with $\boldsymbol{A} = \{A_{ij}\}$, $B^{(i)} = \{B^{(i)}_{ij}\}$, $(B^{(i)})^T = \{B^{(i)}_{ji}\}$, $M = \{M_{ij}\}$, $c^{(i)} = \{c^{(i)}_i\}$, and $\pi = \{\pi_i\}$.

In `integrands` it is convenient to generate four block matrices: the upper left matrix

$$\begin{pmatrix} A & 0 \\ 0 & A \end{pmatrix},$$

corresponding to the Laplace term $\nabla^2\boldsymbol{v}$, the upper right matrix

$$\begin{pmatrix} B^{(1)} \\ B^{(2)} \end{pmatrix},$$

corresponding to the pressure term ∇p, the lower left matrix

$$\begin{pmatrix} (B^{(1)})^T & (B^{(2)})^T \end{pmatrix},$$

corresponding to the divergence term $\nabla\cdot\boldsymbol{v}$, and finally the lower right matrix

$$\begin{pmatrix} \epsilon M \end{pmatrix},$$

corresponding to the regularization term $\epsilon\nabla^p$ in the continuity equation.

With the above formulas we are ready to present the gory details of the `integrands` function:

```
void Stokes:: integrandsMx (ElmMatVec& elmat, const MxFiniteElement& mfe)
{
  const int nsd   = mfe.getNoSpaceDim();
  const int nvxbf = mfe(Vx).getNoBasisFunc();
  const int nvbf  = nvxbf*nsd;
  const int npbf  = mfe(P).getNoBasisFunc();
  const real detJxW = mfe.detJxW();
  real  nabla;
  int   i,j,k,d,s;

  // upper left block matrix, the term -Laplace(u)*mfe(U).N(i)

  for (d = 1; d <= nsd; d++)
    for (i = 1; i <= nvxbf; i++) {
      for (j = 1; j <= nvxbf; j++) {
        nabla = 0;
        for (k = 1; k <= nsd; k++)
          nabla += mfe(d).dN(i,k)*mfe(d).dN(j,k);
        elmat.A((d-1)*nvxbf+i,(d-1)*nvxbf+j) += nabla*detJxW;
      }
      elmat.b((d-1)*nvxbf +i) += 0;
    }

  // upper right block matrix, the term (dp/dx)*mfe(U).N(i)
  for (s = 1; s <= nsd; s++)
    for (i = 1; i <= nvxbf; i++)
      for (j = 1; j <= npbf; j++) {
        if  (dpdx_by_parts)
          elmat.A((s-1)*nvxbf+i,nvbf+j)
            += mfe(s).dN(i,s)*mfe(P).N(j)*detJxW;
        else
          elmat.A((s-1)*nvxbf+i,nvbf+j)
            +=-mfe(s).N(i)*mfe(P).dN(j,s)*detJxW;
      }

  // lower left block matrix, the term du/dx*mfe(P).N(i),
  // eq.no. nvbf+i
  for ( d=1 ; d<= nsd; d++)
    for (i = 1; i <=  npbf; i++)
      for (j = 1; j <= nvxbf; j++)
        elmat.A(nvbf+i,(d-1)*nvxbf+j)+= mfe(P).N(i)*mfe(d).dN(j,d)*detJxW;


  // lower right block matrix,
  // the term -epsilon*Laplace(p)*mfe(P).N(i)

  for (i = 1; i <= npbf; i++) {
    for (j = 1; j <= npbf; j++) {
      nabla = 0;
      for (k = 1; k <= nsd; k++)
        nabla += mfe(P).dN(i,k)*mfe(P).dN(j,k);
      elmat.A(nvbf+i,nvbf+j) += epsilon*nabla*detJxW;
    }
    elmat.b(nvbf+i) += 0;
  }
}
```

Notice that `mfe(Vx)` or `mfe(Vy)` holds the N_i functions and their derivatives, whereas `mfe(P)` holds the L_i functions.

The program administration takes place in `solveProblem`:

```
void Stokes:: solveProblem ()
{
  fillEssBC ();
  makeSystemMx (*dof, *lineq, *coll, 1);
  linsol.fill(0.0);
  lineq->solve();
  dof->vec2field (linsol, *coll);

  database->dump(*v);
  database->dump(*p);
}
```

The contents of `solveProblem` follows the set-up from the examples in [14], except that we call `makeSystemMx` and use the `coll` collection of fields both in `makeSystemMx` and when storing the solution of the linear system (`linsol`) back in the fields (`dof->vec2field`). The complete source code is found in

```
$NOR/doc/mixed/src/Stokes
```

6.4 Block-Structured Preconditioners

Upcoming material on block-structured preconditioners utilizing, e.g., multigrid.

References

[1] D. N. Arnold, R. S. Falk, and R. Winther. Preconditioning discrete approximation of the Reissner Mindlin plate problem. *Matematical Modeling and Numerical Analysis*, 1996.

[2] D. N. Arnold, R. S. Falk, and R. Winther. Preconditioning in H(div) and applications. *Math. Comp. 66*, 1997.

[3] D. N. Arnold, R. S. Falk, and R. Winther. Multigrid in H(div) and H(curl). *Preprint Mittag Leffler*, 1997/98.

[4] S. C. Brenner and L. R. Scott. *The Mathematical Theory of Finite Element Methods.* Springer-Verlag, 1994.

[5] F. Brezzi and M. Fortin. *Mixed and Hybrid Finite Element Methods.* Springer-Verlag, 1991.

[6] A. M. Bruaset. *A Survey of Preconditioned Iterative Methods*. Addison-Wesley Pitman, 1995.

[7] A. M. Bruaset and H. P. Langtangen. A comprehensive set of tools for solving partial differential equations; Diffpack. In M. Dæhlen and A. Tveito, editors, *Mathematical Models and Software Tools in Industrial Mathematics*, pages 61–90. Birkhäuser, 1997.

[8] A. M. Bruaset and H. P. Langtangen. Object-oriented design of preconditioned iterative methods in Diffpack. *Transactions on Mathematical Software*, 23:50–80, 1997.

[9] R. Wait et al. *The mathematical basis of finite element methods*. Clarendon Press, 1986.

[10] M. Fortin and R. Glowinski. *Augmented Lagrangian methods: Applications to the numerical solution of boundary value problems*. Elsevier Science Publisher B.V, 1982.

[11] V. Girault and P. A. Raviart. *Finite Element Methods for Navier-Stokes Equations*. Springer-Verlag, 1986.

[12] C. Johnson. *Numerical solution of partial differential equations by the finite element method*. Studentlitteratur, 1987.

[13] H. P. Langtangen. Getting started with finite element programming in Diffpack. World Wide Web document: Diffpack v1.4 Report Series, SINTEF & University of Oslo, 1996.
See URL *http://www.math.uio.no/~hpl/dpreports.html*.

[14] H. P. Langtangen. *Computational Partial Differential Equations – Numerical Methods and Diffpack Programming*. Lecture Notes in Computational Science and Engineering. Springer-Verlag, 1999.

[15] H. P. Langtangen. *Computational Partial Differential Equations - Numerical Methods and Diffpack Programming*. Springer-Verlag, 1999 (to appear).

[16] H. P. Langtangen. Details of finite element programming in Diffpack. The Numerical Objects Report Series #1997:9, Numerical Objects AS, Oslo, Norway, October 6, 1997. See *ftp://ftp.nobjects.com/pub/doc/NO97-09.ps.gz*.

[17] J. C. Nedelec. Mixed finite elements in R^3. *Numerische Mathematik*, 1980.

[18] P. A. Raviart and J. M. Thomas. A mixed finite element method for 2-order elliptic problems. *Matematical Aspects of Finite Element Methods*, 1977.

[19] T. Rusten and R. Winther. A preconditioned iterative method for saddlepoint problems. *SIAM J. Matrix Anal.*, 1992.

[20] J. Xu. Iterative methods for space decomosision and subspace correction. *SIAM Review*, 1992.