Increasing the parallel efficiency of the Variational Multiscale method by means of deflation

Gertjan van Zwieten

2006

Master's thesis

MSc committee:

dr. ir. C. Vuik
dr. ir. M. B. van Gijzen
dr. ir. F. J. Lingen
dr. ir. W. T. van Horssen
prof. dr. ir. P. Wesseling

Preface

This thesis has been written for the degree of Master of Science in Applied Mathematics, faculty of Electrical Engineering, Mathematics and Computer Science of Delft University of Technology. The graduation has been done in the unit of Numerical Analysis, taking about nine months of work. The first three months have been used for literature study to get an overview of the main research topics, finalized with an intermediate report and presentation. This thesis is the result of the remaining six months of research.

Three different parties have been involved in this project. The subject of this thesis has been brought forth by research on Computational Fluid Dynamics at Aerospace Engineering, in the unit of Engineering Mechanics. This has been made into a graduation project at the faculty of EEMCS, in the unit of Numerical Analysis because of expertise in the relevant area. The actual work has been carried out as an internship at Habanera, the company that provides the Computational Mechanics unit with numerical software for its scientific computations.

Quite a number of people have hence been involved, and I would like to thank them for their help. My direct supervisors at Numerical Analysis Kees Vuik and Martin van Gijzen, for their valuable suggestions and feedback on my writings. Steven Hulshoff and Edwin Munts at Computational Mechanics, for talking me through their code, and for not looking too disappointed when I found out that I was not going to be able to improve it. And lastly, Erik Jan Lingen and Martijn Stroeven at Habanera, for thinking along with me, and for their company during the nine months that the project lasted. This time at Habanera has been a pleasant one.

Delft, July 2006

Gertjan van Zwieten

Contents

Introduction

The numerical simulation of fluid-structure interaction is a complicated problem, mainly because the dynamics are very sensitive to the unsteady behaviour of turbulent flows. Numerical simulation of these flows is theoretically possible; the governing Navier-Stokes equations have been known since the early 19th century. Practically, however, the direct solution is limited to either very small scale problems or low Reynolds number flows, as the discretized system will very rapidly outgrow the capabilities of current day computers. The Variational Multiscale (VMS) method has been suggested as a possible solution to this problem, solving a slightly different set of equations on a coarser mesh. Although this approach has been very successful, problems remain when the new system of equations is solved in parallel on a computer cluster.

Parallel computing is used for two reasons. First, by executing separate tasks simultaneously, the result can be obtained faster. With the current day processors running up against fundamental physical limits, further speed gains must be sought in exploiting parallelism. Second, by distributing the problem over multiple nodes, larger problems can be solved. Two processors will always be able to solve a larger problem than a single one, no matter how much memory each has separately. The disadvantage of all this is formed by the inevitable inter processor communication, which is usually so much slower than computation that it must be reduced to a minimum. This means in practice that lesser algorithms are preferred over higher quality algorithms because of their better parallel properties. The problem for VMS is that these lesser algorithms turn out to be inferior.

The goal of this report is to examine the possibilities of improving the performance of VMS in parallel computing environments. The aim is not to modify the VMS method itself, which will be considered a given fact. Instead, investigations will concentrate on the algorithms that are being used to solve the linear systems generated by this method — the solvers — that are responsible for a large part of the total computation time. Key focus will be to enhance these solvers by applying deflation, a numerical technique that has already been proved effective for various other types of problems. This report will try to determine if deflation can be effective in increasing the parallel efficiency of the VMS method as well.

The answer to this question has been divided in four chapters. Chapter 1 introduces the VMS method. This method is the source of the troublesome linear systems, and the underlying mathematics as well as the physical interpretation of the solution can

provide insight in the problem, and therefore, the solution. The deflation technique that is expected to accelerate the solution process of these systems will be introduced in Chapter 2, forming the conclusion of a general introduction in iterative solution methods. Because numerical experiments will be required, Chapter 3 describes the implementation of the deflation technique in the software framework that has been used for VMS as well. The results of these experiments will be presented and discussed in Chapter 4, at which point it should be clear whether deflation is indeed capable of alleviating the experienced parallelization problems for VMS.

Throughout the report a number of notational conventions will be used. Important terms that occur for the first time in the report are printed in **boldface**, signifying that they have been added to the index. The highlight will aid the location of terms looked up from the index. Other use of boldface is for vectors $\boldsymbol{v}$, lowercase, and matrices $\boldsymbol{A}$, uppercase. Vector spaces $\mathcal{V}$ are typeset in calligraphic font; the only exception being the set of reals, $\mathbb{R}$. The single other use for this blackboard font is in the projection operator $\mathbb{P}$ that will be introduced in Chapter 2.

Chapter 1

The VMS method

In the field of fluid dynamics, arguably the most important distinction made is that of laminar and turbulent flows. Laminar flows are characterized by a smooth and highly predictable nature, whereas turbulent flows are chaotic, exhibiting rapid and seemingly random variations in pressure and velocity. The distinctive difference between the two types of flow is clearly visible in Figure 1.1, which shows a comparison of both types of flow over a sphere. Unfortunately, most flows that are found in nature, be it stirred coffee or the airflow around an airplane, are of the turbulent type. Their chaotic nature makes numerical simulation extremely hard, due mostly to the enormous range of scales of motion characteristic for this type of flow. These simulations form the research field of **Computational Fluid Dynamics** (CFD).

This chapter will present the Variational Multiscale (VMS) method, which aims to accurately simulate turbulent flows at much lower costs than a direct numerical simulation. Section 1.1 presents the governing set of Navier-Stokes equations. Section 1.2 then gives an overview of Large Eddy Simulation, a method that is somewhat older but quite similar to VMS, which is introduced next in that section.

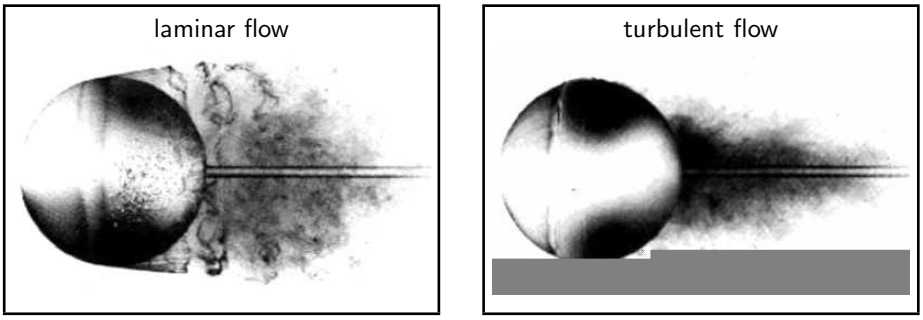

Figure 1.1. *Comparison of laminar and turbulent flow past a sphere in water, copied from Anderson [1], Figures 6.9 and 6.10. The laminar flow separates readily from the surface, leaving a large wake. The turbulent flow separates much further back on the surface.*

Section 1.3 defines the discrete representation of the flow, and shows how this flow can be computed using a discontinuous Galerkin method. Finally, Section 1.4 is the final step towards an actual implementation of this method by transforming the resulting non-linear system into a series of linearized equations.

Besides the notational conventions that apply throughout this report, equations in this chapter make extensive use of what is known as **Cartesian tensor notation**. In this notation, spatial and temporal differentiation are denoted as follows:

$$\phi_{,i} = \frac{\partial \phi}{\partial x_i}, \quad \phi_{,t} = \frac{\partial \phi}{\partial t}. \tag{1.1}$$

This convention is very common in CFD literature because of the clean and short notation it provides for the often large equations in this field. This is especially so when used in combination with the **Einstein summation convention**, in which Greek indices occurring twice within a term or product are interpreted as a summation over that index. For example, the divergence of a vector field $\boldsymbol{u}(\boldsymbol{x})$ in 3D space is notated compactly as $u_{\alpha,\alpha}$, which expands to $u_{1,1} + u_{2,2} + u_{3,3}$. Both Cartesian tensor notation and the Einstein summation convention are used exclusively in this chapter.

1.1 The Navier-Stokes equations

Turbulent flows consist of an enormous range of fluid motions. See for example Figure 1.2, the famous Leonardo da Vinci drawing which shows large circular motions around a sluice and much smaller whorls near the impact of the stream. Even the tiniest whorls, however, are about five orders of magnitude larger than the discrete molecules that the fluid is made of. The molecular nature of the fluid is therefore assumed to have only negligible effect on the macroscopic flow. Instead, the fluid is treated as a continuous medium, which makes it possible to speak in terms of speed and density at a distinct point in space and time. This important assumption is known in fluid mechanics as the **continuum hypothesis**.

The equations that describe the motion of a fluid are known as the **Navier-Stokes equations**, independently derived by G.G. Stokes and M. Navier in the early 1800's. In the incompressible, viscous case the system consists of five equations: the continuity equation, three momentum equations and the energy equation. These equations will be presented here with a brief outline of their origins; for more information on all underlying physics see for example Anderson's Fundamentals of aerodynamics [1].

1. Continuity equation

   An obvious condition that any fluid must obey is conservation of mass. In an arbitrary control volume in space, the net mass flow through the boundaries should equal the rate of mass production within. When the latter is assumed to be zero, this leads to the following relation between the flow density ρ and speed $\boldsymbol{u}$:

   $$\rho_{,t} + (\rho u_\alpha)_{,\alpha} = 0. \tag{1.2}$$

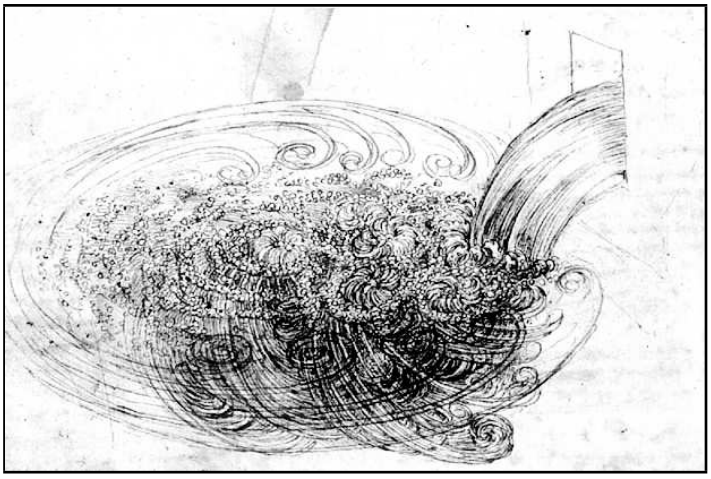

Figure 1.2. *Part of Leonardo da Vinci's* Studies of water passing obstacles and falling*, showing a flow field induced by a falling stream. The drawing shows many different scales of fluid motion.*

2. Momentum equations

   Another important law of nature is conservation of momentum, also known as Newton's second law: force equals rate of change of momentum. The forces that are exerted on a fluid fall in two categories. In terms of control volumes: body forces act on the entire volume (gravity etcetera, collectively $\boldsymbol{f}$) while surface forces act on its boundary (pressure p, shear stress τ_{ij}). The former is assumed to be known, the latter will be related to the flow field variables through constitutive relations. This gives three separate equations, one for each dimension i:

   $$(\rho u_i)_{,t} + (\rho u_i u_\alpha - \tau_{i\alpha})_{,\alpha} = \rho f_i. \tag{1.3}$$

   The following constitutive relations are most common:

   - $\tau_{ij} = -p\delta_{ij} + \mu(u_{i,j} + u_{j,i} - \frac{2}{3}\delta_{ij}u_{k,k})$
   - $p = \rho RT$,

   where μ is the dynamic viscosity of the fluid, T the temperature and R the specific gas constant that connects the fluid's pressure, density and temperature. The δ_{ij} in the shear stress expression is the Kronecker delta, which evaluates to one if i equals j, and zero otherwise.

3. Energy equation

   For incompressible flows, i.e. having constant density ρ, the above system of four equations is complete. For compressible flows, however, an additional equation is required for the extra unknown ρ. This is the energy equation, derived from the first law of thermodynamics which states that energy can neither be created nor destroyed, it can only change form. This means that the rate of change of the total (internal plus kinetic) energy e should balance the work exerted on the fluid and heat flux $\boldsymbol{q}$, yielding:

   $$(\rho e)_{,t} + (\rho u_\alpha e - \tau_{\alpha\beta}u_\beta + q_\alpha)_{,\alpha} = \rho f_\beta u_\beta, \tag{1.4}$$

with again two constitutive relations to close the system:

- $q_i = -\kappa T_{,i}$
- $e = \frac{1}{2} u_\alpha u_\alpha + c_v T.$

Here κ is the thermal conductivity and c_v the specific heat at constant volume, related to the specific heat at constant pressure c_p through $c_p - c_v = R$.

In vector notation, it can be seen that the above system of five equations can be written compactly as follows:

$$\boldsymbol{U}_{,t} + \boldsymbol{F}_{\alpha,\alpha} = \boldsymbol{S}, \tag{1.5}$$

where $\boldsymbol{U}(\boldsymbol{x}, t) \in \mathbb{R}^5$ contains the so-called **conservation variables**, $\boldsymbol{F}_\alpha(\boldsymbol{x}, t) \in \mathbb{R}^5$ is the **flux vector** in the α-th direction and $\boldsymbol{S}(\boldsymbol{x}, t) \in \mathbb{R}^5$ a **source vector**. Along with the constitutive relations, Equation 1.5 forms a closed, non-linear system that completely describes the flow of a fluid. It will be shown, however, that to create an efficient numerical method for solving this system is far from straightforward.

1.2 Resolving the large scales of motion

With the governing Navier-Stokes equations available as Equations 1.5, the obvious approach to simulating a fluid flow is to simply discretize the flow field and solve this equation numerically. This approach is known as **Direct Numerical Simulation** (DNS). Unfortunately, in practice, the smallest length scales are so small that the costs of such a direct simulation are extremely high. This makes DNS unfeasible for realistically sized problems. Various models — note: as opposed to simulations — have been developed to overcome this problem. Examples are Reynolds Averaged Navier-Stokes (RANS) and Large Eddy Simulation (LES). Compared to these, the Variational Multiscale (VMS) method that will be introduced in this chapter is relatively new.

An important idea behind these methods is that of the **energy cascade**, introduced by Richardson [14] in 1920. His idea was that turbulence is composed of eddies of different sizes — an eddy being a certain localized turbulent motion. Richardson observed that the large eddies are unstable and break up, transferring their energy to somewhat smaller eddies, which in turn break up into smaller eddies. This energy cascade continues until at very small scale the eddy motion is stable, and kinetic energy is dissipated through molecular viscosity. Richardson elegantly summarized his paper as follows:

Big whorls have little whorls,
Which feed on their velocity;
And little whorls have lesser whorls,
And so on to viscosity.

VMS and LES are based on similar ideas. Both methods solve the system of Navier-Stokes equations on a much coarser grid than required for a direct numerical simulation. Since this grid can not represent the smallest scales of fluid motion, the 'lesser whorls / and so on to viscosity', a so called **eddy-viscosity model** is used to compensate for this deficient resolution. Such a model aims to reconstruct the influence of the smallest, unresolved scales on the larger, resolved scales, based only on those resolved scales. The quality of the resulting simulation is determined largely by the quality of this model.

1.2.1 Large Eddy Simulation

Because LES is so closely related to VMS a short overview of the main ideas behind this method will be useful. Returning to Da Vinci's drawing, Figure 1.2, it can be seen that below a certain length scale the turbulent motions turn from anisotropic to isotropic, losing all information about the boundary imposed geometry of the large eddies. This divides the range of scales into two classes with markedly different properties. DNS resolves both of these, expending nearly all of its computational effort on the smallest scales of motion. LES resolves only the largest, exploiting the fact that the large eddies contain the bulk of the kinetic energy.

Pope [13] identifies four conceptual steps in LES:

1. A spatial filter is defined to decompose the velocity $\boldsymbol{u}$ into a filtered (resolved) component $\bar{\boldsymbol{u}}$ and a residual (unresolved) component $\hat{\boldsymbol{u}}$. The filtered velocity field $\bar{\boldsymbol{u}}$ represents the motion of the large eddies.
2. The filter is applied to the system of Navier-Stokes equations to derive a new system for the filtered velocity field $\bar{\boldsymbol{u}}$. Problems arise at the non-linear terms, since those can not be expressed exclusively in terms of the filtered velocity $\bar{\boldsymbol{u}}$. This is known as the **closure problem**.
3. Closure is obtained by replacing the velocity $\boldsymbol{u}$ in the non-linear terms with the filtered velocity $\bar{\boldsymbol{u}}$ and correcting this with a model term: the closure.
4. The new system is solved for $\bar{\boldsymbol{u}}$, which provides an approximation of the large scale motions in the turbulent flow.

Although LES does give quite good results for certain flow problems and in fact is widely used, the method has its drawbacks. For instance at step 2, the filter operation does not commute with spatial differentiation on non-uniform grids, which restricts the method to relatively simply geometries. Other problems arise at the walls, where the filter will either have to shrink or extend beyond the boundary. The major source of problems, however, is step 3. The closure term often fails to realistically model the influence of the neglected smallest eddies, in particular near the walls. Since this model applies to all large scale motions LES fails to converge to the true DNS solution, which in the worst case may lead to even qualitatively different results.

1.2.2 The Variational Multiscale method

The Variational Multiscale method, described a.o. by Hughes et al. [6], Collis [2] and Munts et al. [11], has been specifically designed to address some of the shortcomings of Large Eddy Simulation. Both methods use a similar approach; the system of Navier-Stokes equations is solved on a grid that is much too coarse to represent the small scales of fluid motion, and a model is used to compensate for the inability to simulate the smallest eddies. Instead of a single decomposition in resolved and unresolved scales, however, VMS goes one step further and subdivides the former into large and small resolved scales. This division is reflected by a corresponding division in function spaces:

$$\mathcal{V} = \bar{\mathcal{V}} \oplus \tilde{\mathcal{V}} \oplus \hat{\mathcal{V}}, \tag{1.6}$$

where $\mathcal{V}$ is the problem's solution space, containing space-time representations of the flow, and $\bar{\mathcal{V}}$, $\tilde{\mathcal{V}}$ and $\hat{\mathcal{V}}$ represent the large, small and unresolved scales, respectively. The first two are resolved and therefore finite dimensional. For brevity the resolved space is sometimes denoted as:

$$\bar{\tilde{\mathcal{V}}} \stackrel{\text{def}}{=} \bar{\mathcal{V}} \oplus \tilde{\mathcal{V}}. \tag{1.7}$$

The system of Navier-Stokes equations has been presented in strong form in Section 1.1 as Equation 1.5. Vectors $\boldsymbol{U}$ and $\boldsymbol{F}_\alpha$ can be expressed in any set of five flow field variables, joined in a solution vector $\boldsymbol{Y}(\boldsymbol{x}, t)$. This choice of variables defines the solution space $\mathcal{V}$. For practical reasons it was chosen to define the flow field by density, speed and temperature:

$$\boldsymbol{Y} = \{\rho, u_1, u_2, u_3, T\}^T \in \mathcal{V}. \tag{1.8}$$

Using the inner product $(\boldsymbol{f}, \boldsymbol{g}) = \int_Q \boldsymbol{f}^T \boldsymbol{g} \, \mathrm{d}Q$, taking the inner product with an arbitrary test function $\boldsymbol{W} \in \mathcal{V}$ transforms Equation 1.5 to its weak form:

$$\int_Q \boldsymbol{W}^T (\boldsymbol{U}_{,t} + \boldsymbol{F}_{\alpha,\alpha}) \, \mathrm{d}Q = \int_Q \boldsymbol{W}^T \boldsymbol{S} \, \mathrm{d}Q \quad \forall \, \boldsymbol{W} \in \mathcal{V}, \tag{1.9}$$

or, expressed in $\boldsymbol{Y}$:

$$B(\boldsymbol{W}, \boldsymbol{Y}) = (\boldsymbol{W}, \boldsymbol{S}) \quad \forall \, \boldsymbol{W} \in \mathcal{V}, \tag{1.10}$$

for some operator $B : \mathcal{V} \times \mathcal{V} \to \mathbb{R}$, that is linear in its first argument and non-linear in its second. The weak formulation is generally the first step in a solution procedure, such as a finite element method. Indeed, when the Navier-Stokes equations are reformulated in the VMS framework, this procedure will be continued in Section 1.3.2. The weak formulation is presented here already because it is an essential part of VMS as well, which uses the test functions $\boldsymbol{W}$ to separate the large, small and unresolved scales of motion. Using the decomposition defined in Equation 1.6, Equation 1.10 can be written as:

$$B(\bar{\boldsymbol{W}} + \tilde{\boldsymbol{W}} + \hat{\boldsymbol{W}}, \bar{\boldsymbol{Y}} + \tilde{\boldsymbol{Y}} + \hat{\boldsymbol{Y}}) = (\bar{\boldsymbol{W}} + \tilde{\boldsymbol{W}} + \hat{\boldsymbol{W}}, \boldsymbol{S}) \\ \forall \, \bar{\boldsymbol{W}} \in \bar{\mathcal{V}}, \tilde{\boldsymbol{W}} \in \tilde{\mathcal{V}}, \hat{\boldsymbol{W}} \in \hat{\mathcal{W}}. \tag{1.11}$$

Taking only one element $\bar{\boldsymbol{W}}$, $\tilde{\boldsymbol{W}}$ or $\hat{\boldsymbol{W}}$ non-zero at a time, the following system of three equations is obtained:

$$\begin{aligned} B(\bar{\boldsymbol{W}}, \bar{\boldsymbol{Y}} + \tilde{\boldsymbol{Y}} + \hat{\boldsymbol{Y}}) &= (\bar{\boldsymbol{W}}, \boldsymbol{S}) \quad \forall\, \bar{\boldsymbol{W}} \in \bar{\mathcal{V}} \\ B(\tilde{\boldsymbol{W}}, \bar{\boldsymbol{Y}} + \tilde{\boldsymbol{Y}} + \hat{\boldsymbol{Y}}) &= (\tilde{\boldsymbol{W}}, \boldsymbol{S}) \quad \forall\, \tilde{\boldsymbol{W}} \in \tilde{\mathcal{V}} \\ B(\hat{\boldsymbol{W}}, \bar{\boldsymbol{Y}} + \tilde{\boldsymbol{Y}} + \hat{\boldsymbol{Y}}) &= (\hat{\boldsymbol{W}}, \boldsymbol{S}) \quad \forall\, \hat{\boldsymbol{W}} \in \hat{\mathcal{V}}. \end{aligned} \tag{1.12}$$

Because all terms are linear in their first argument, summing the three equations transforms Equation 1.12 back to 1.11, which proves complete equivalence to the original set of Navier-Stokes equations. In other words, the above system is a reformulation of the original Equation 1.5 and as such still exact. No approximations have been made to this point.

Reviewing the above procedure, the original system has been reformulated in terms of unresolved, small and large scales by making a corresponding division in function spaces and writing the solution $\boldsymbol{Y}$ as a sum of its elements. The weak formulation acts as a kind of 'projection' of $\boldsymbol{Y}$ onto $\bar{\mathcal{V}}$, $\tilde{\mathcal{V}}$ and $\hat{\mathcal{V}}$. This is a striking difference with the LES method, because now the three scales are separated without the use of any filtering operation. As a consequence, the obtained formulation is still exact. Using test functions for the splitting eradicates all problems caused by the spatial filter (at the wall, on unstructured grids), without even introducing a substitute operation as these test functions will be part of the solution procedure anyway.

The exactness of Equation 1.12 will be lost when the unresolved scales $\hat{\boldsymbol{Y}}$ are ignored, which is necessary because these can not be represented on a coarse grid. The influence of these scales is present in both of the remaining large and small scale equations. However, according to Richardson's energy cascade introduced in the beginning of Section 1.2, eddies influence mostly nearby scales. Therefore, the influence of the unresolved scales on the largest scales can be assumed to be negligible, which means that the $\hat{\boldsymbol{Y}}$ term drops out of the first equation. The unresolved scales do dissipate energy from the small scales, therefore terms involving unresolved scales in the small scale equation will have to be modeled. Under these assumptions, the remaining non-linear system can be written as:

$$\begin{aligned} \boldsymbol{B}(\bar{\boldsymbol{W}}, \bar{\boldsymbol{Y}} + \tilde{\boldsymbol{Y}}) &= (\bar{\boldsymbol{W}}, \boldsymbol{S}) \quad \forall\, \bar{\boldsymbol{W}} \in \bar{\mathcal{V}} \\ \boldsymbol{B}(\tilde{\boldsymbol{W}}, \bar{\boldsymbol{Y}} + \tilde{\boldsymbol{Y}}) + \boldsymbol{M}(\tilde{\boldsymbol{W}}, \tilde{\boldsymbol{Y}}) &= (\tilde{\boldsymbol{W}}, \boldsymbol{S}) \quad \forall\, \tilde{\boldsymbol{W}} \in \tilde{\mathcal{V}}, \end{aligned} \tag{1.13}$$

where $\boldsymbol{M}$ is a model term that acts on the small scales only. Adding the large and small scale equation yields the final, non-linear system of equations:

$$\boldsymbol{B}(\bar{\tilde{\boldsymbol{W}}}, \bar{\tilde{\boldsymbol{Y}}}) + \boldsymbol{M}(\tilde{\boldsymbol{W}}, \tilde{\boldsymbol{Y}}) = (\bar{\tilde{\boldsymbol{W}}}, \boldsymbol{S}) \quad \forall\, \bar{\tilde{\boldsymbol{W}}} \in \bar{\tilde{\mathcal{V}}}. \tag{1.14}$$

Here $\bar{\tilde{\boldsymbol{Y}}}$ is the (finite dimensional) solution vector of the numerical scheme. In the following only finite dimensional approximations will be considered, and the distinction with the exact solution $\boldsymbol{Y}$ will be made no more. Therefore, from now on $\mathcal{V}$ will denote the solution space containing resolved scales, and its elements are $\boldsymbol{Y}$ and $\boldsymbol{W}$. The tilde does not change meaning; it will still denote the small components of the resolved scales. With these redefinitions, Equation 1.14 can be put back integral form as follows:

$$\int_Q \boldsymbol{W}^T(\boldsymbol{U}_{,t} + \boldsymbol{F}_{\alpha,\alpha})\, \mathrm{d}Q + \int_Q \tilde{\boldsymbol{W}}^T \tilde{\boldsymbol{F}}^m_{\alpha,\alpha}\, \mathrm{d}Q = \int_Q \boldsymbol{W}^T \boldsymbol{S}\, \mathrm{d}Q \quad \forall\, \boldsymbol{W} \in \mathcal{V}, \tag{1.15}$$

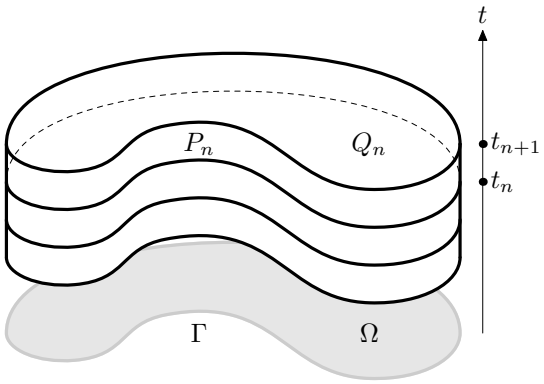

Figure 1.3. *Schematic overview of the division of a space-time domain Q into successive time-slabs Q_n. Note that in reality, the spatial domain Ω is three-dimensional.*

where $\tilde{\boldsymbol{F}}_\alpha^m$ is a model term. The tilde denotes its dependence on $\tilde{\boldsymbol{Y}}$. A final comparison with the exact Equation 1.9 gives a clear picture of the resulting VMS procedure: the unmodified Navier-Stokes equations are solved in weak form on a coarse grid, and an extra term $\tilde{\boldsymbol{F}}_\alpha$ models the effect of the eddies that can not be resolved on this grid. Because this model term is a function of the small resolved scales only, the solution space $\mathcal{V}$ should be constructed such that it allows an easy separation of scales.

1.3 Formulating a finite element method

Equation 1.15 derived in the previous section will be solved numerically using a finite element method. This means that a finite element discretization of the domain will need to be defined. Because $\boldsymbol{Y}$ represents a solution in both space and time, this domain Q is four-dimensional. Therefore solving Equation 1.15 on the entire domain at once is not feasible. In order to reduce the size of the final calculations the domain will first be divided into a series of so called **time-slabs** Q_n; disjoint, four-dimensional subdomains that are relatively 'thin' in the time dimension:

$$\begin{aligned} Q_n &= \Omega \times (t_n, t_{n+1}) \\ P_n &= \Gamma \times (t_n, t_{n+1}). \end{aligned} \tag{1.16}$$

Figure 1.3 gives a schematic overview of this setup; Ω denotes the spatial domain with boundary Γ and t_n and t_{n+1} are successive points in time. It will be shown that on these time-slabs the restrictions $\boldsymbol{Y}|_{Q_n}$ can be computed successively. This will not only reduce the size of the linear systems, it will also allow for a lot of flexibility in dealing with dynamically deforming domains. Recall that the VMS method has been designed for fluid-structure interaction problems. The flexibility is due mostly to the fact that successive time-slabs need only be 'weekly' connected, as will be explained in Section 1.3.2, after the solution space has been defined.

1.3.1 Construction of the solution space

A finite element discretization consists of two things. One, a set of finite elements. For a time-slab Q_n these are subdomains $Q_{(n,i)}$ such that $\cup_i Q_{(n,i)} = Q_n$ and $Q_{(n,i)} \cap Q_{(n,j)} = \varnothing$ if $i \neq j$. Two, a set of basis functions. Because multiple basis functions can have support on the same finite element an additional index a will be used for these. Hence $N_{(n,i,a)} : Q_{(n,i)} \rightarrow \mathbb{R}$ is basis function a for elements i of time-slab n. With these definitions, the solution space $\mathcal{V}$ can be defined via its restrictions on the finite elements:

$$\boldsymbol{Y} \in \mathcal{V}, \quad \boldsymbol{Y}\Big|_{Q_{(n,i)}} = \sum_{a \in \mathcal{A}} N_{(n,i,a)} \boldsymbol{y}_{(n,i,a)}, \tag{1.17}$$

where $\mathcal{A}$ is an index set for the basis functions and $\boldsymbol{y}_{(n,i,a)} \in \mathbb{R}^5$ are element vectors corresponding to time-slab n, element i, basis function a. The only restriction that is imposed on the element vectors is that the resulting function $\boldsymbol{Y}$ must be continuous within each time-slab. Continuity over successive time-slabs is not required, as shall be explained in Section 1.3.2.

The previous section pointed out that the function space $\mathcal{V}$ should allow easy separation of scales, because the model term in Equation 1.15 depends on the small scales $\tilde{\boldsymbol{Y}}$ only. Equation 1.17 suggests to choose the basis functions $N_{(n,i,a)}$ in such a way that each function can be identified with a certain scale. In that case the small-scale function space $\tilde{\mathcal{V}}$ can be defined as a sum over a subset of these basis functions:

$$\tilde{\boldsymbol{Y}} \in \tilde{\mathcal{V}}, \quad \tilde{\boldsymbol{Y}}\Big|_{Q_{(n,i)}} = \sum_{a \in \tilde{\mathcal{A}}} N_{(n,i,a)} \boldsymbol{y}_{(n,i,a)}, \tag{1.18}$$

where $\mathcal{A} \supset \tilde{\mathcal{A}}$ contains the indices of the basis functions that can be identified with small scales. Defined like this, $\tilde{\boldsymbol{Y}}$ and $\boldsymbol{Y}$ share the same element vector $\boldsymbol{y}_{(n,i,a)}$, which makes it very easy to relate one to the other in a numerical setting.

The construction of polynomial basis functions on a multi-dimensional domain is described in Chapter 3 of Karniadakis et al. [8]. Given a number of P polynomials $\psi_p : [-1, 1] \rightarrow \mathbb{R}$, the expansion on a four-dimensional domain is given by the tensor product:

$$\phi_{pqrs}(\xi_1, \xi_2, \xi_3, \xi_4) = \psi_p(\xi_1)\psi_q(\xi_2)\psi_r(\xi_3)\psi_s(\xi_4). \tag{1.19}$$

A set of P^4 basis functions can be created for each element $Q_{(n,i)}$ by mapping ξ_1 – ξ_4 onto the finite element domain. For these basis functions to correspond to a range of scales, the same must hold for the polynomial expansion set ψ_p from which they are derived. A hierarchical **modal p-type expansion** has this property, because its high order modes can be identified with small scales. This expansion is defined as follows:

$$\psi_p(\xi) = \begin{cases} \frac{1-\xi}{2} & p = 0 \\ \frac{1-\xi}{2}\frac{1+\xi}{2} P_{p-1}^{\alpha,\beta}(\xi) & 0 < p < P \\ \frac{1+\xi}{2} & p = P, \end{cases} \tag{1.20}$$

where $P_p^{\alpha,\beta}(\xi)$ represents Jacobi polynomials of order p, for example Legendre ($\alpha = \beta = 0$) or Chebychev ($\alpha = \beta = -\frac{1}{2}$) polynomials. The scale separation property

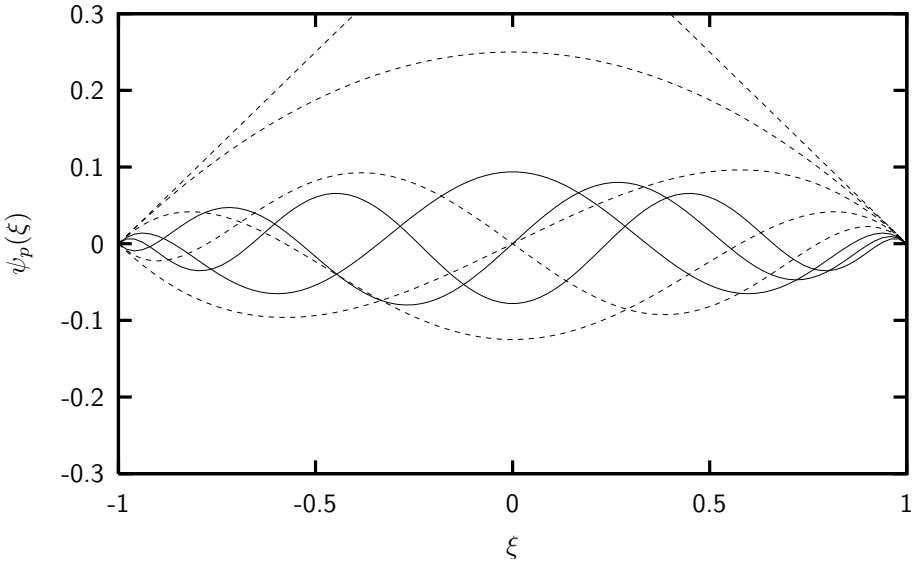

Figure 1.4. *Polynomial Legendre expansion, corresponding to $\alpha = \beta = 0$ and $P = 8$ in (1.20). The solid lines correspond to the highest order modes $p = 5, 6, 7$. In this graph it is clear that high order modes represent small length scales.*

is clearly visible in Figure 1.4, which shows a Legendre expansion with the highest orders highlighted. This property is propagated via Equation 1.19 to the basis functions $N_{(n,i,a)}$.

An attractive side effect of the modal expansion is that, according to Karniadakis et al. [8], its hierarchical nature "permits the use of a reduced number of expansion modes as compared with those in the full tensor space", which is shown in Figure 1.5. Most conveniently, the reduction of this so called **serendipity expansion** is especially significant in higher dimensions, like a four-dimensional space-time discretization. Consequently, this can lead to a significant reduction in size of the final system of equations.

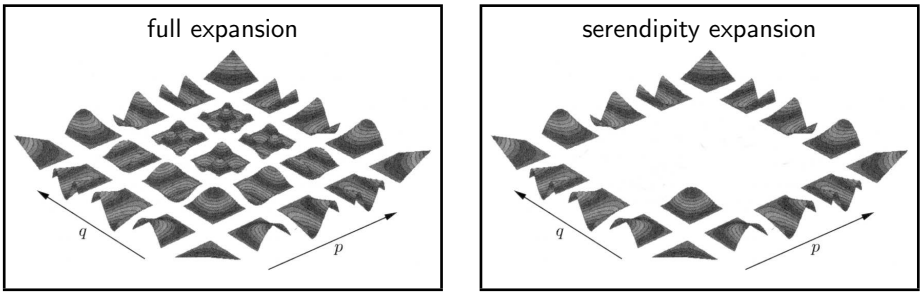

Figure 1.5. *Two-dimensional modal p-type expansion of order $P = 4$, copied from Karniadakis et al. [8], Chapter 3. Left the full expansion, right the reduced serendipity expansion in which only 17 of the total 25 elements are retained. In more dimensions, the reduction is even more significant.*

1.3.2 Discontinuous-Galerkin formulation

Now that the solution space $\mathcal{V}$ has been defined, the final system of equation can be obtained by substituting its elements $\boldsymbol{Y}$ and $\boldsymbol{W}$ into Equation 1.15, which is already in weak form due to the scale separating procedure applied in Section 1.2.2. Rewriting the left-hand terms using the divergence theorem yields:

$$\int_\Omega (\boldsymbol{W}^T|_{t_{n+1}^-}\boldsymbol{U}|_{t_{n+1}^-} - \boldsymbol{W}^T|_{t_n^+}\boldsymbol{U}|_{t_n^+})\ \mathrm{d}\Omega + \int_{P_n} (\boldsymbol{W}^T\boldsymbol{F}_\alpha + \tilde{\boldsymbol{W}}^T\tilde{\boldsymbol{F}}_\alpha^m)n_\alpha\ \mathrm{d}P$$
$$- \int_{Q_n} (\boldsymbol{W}_{,t}^T\boldsymbol{U} + \boldsymbol{W}_{,\alpha}^T\boldsymbol{F}_\alpha + \tilde{\boldsymbol{W}}_{,\alpha}\tilde{\boldsymbol{F}}_\alpha^m + \boldsymbol{W}^T\boldsymbol{S})\ \mathrm{d}Q = 0 \quad \forall\ \boldsymbol{W} \in \mathcal{V}, \quad (1.21)$$

where $\boldsymbol{n}$ in the second integral is the outward normal on P_n, $\boldsymbol{U}|_t$ is shorthand notation for $\boldsymbol{x} \to \boldsymbol{U}(\boldsymbol{x}, t)$ and t_n^+ and t_{n+1}^- are the initial and final time of time-slab n, respectively. Note that $\boldsymbol{U}$ and $\boldsymbol{F}_\alpha$ are functions of $\boldsymbol{Y}$, and $\tilde{\boldsymbol{F}}_\alpha^m$ of $\tilde{\boldsymbol{Y}}$. The spatial boundary conditions enter through the second boundary integral, and the initial condition $\boldsymbol{U}|_{t_n^+}$ in the first integral is formed from the result from the preceding time-slab, $\boldsymbol{U}|_{t_n^-}$. This leads to the so called **jump condition**:

$$\int_\Omega \boldsymbol{W}^T|_{t_n^+}\boldsymbol{U}|_{t_n^+}\ \mathrm{d}\Omega = \int_\Omega \boldsymbol{W}^T|_{t_n^+}\boldsymbol{U}|_{t_n^-}\ \mathrm{d}\Omega \quad \forall\ \boldsymbol{W} \in \mathcal{V}, \quad (1.22)$$

where t_n^+ denotes the initial time in time-slab n and t_n^- the end time of the preceding slab $n-1$. This weakly enforced initial condition allows the solution to be discontinuous over the time-slabs. For this reason, the resulting method is called a **time-discontinuous Galerkin** method.

The solution $\boldsymbol{Y}$ and test functions $\boldsymbol{W}$ are completely determined by their element vectors $\boldsymbol{y}_{(n,i,a)}$ and $\boldsymbol{w}_{(n,i,a)}$ through Equation 1.17. These element vectors are joined per time-slab in global vectors $\boldsymbol{y}_n$ and $\boldsymbol{w}_n$ of length N, from which they can be extracted by a sparse, boolean matrix $\boldsymbol{M}_{(n,i,a)} \in \mathbb{R}^{N\times 5}$, the **location operator**:

$$\begin{aligned} \boldsymbol{y}_{(n,i,a)} &= \boldsymbol{M}_{(n,i,a)}\boldsymbol{y}_n \\ \boldsymbol{w}_{(n,i,a)} &= \boldsymbol{M}_{(n,i,a)}\boldsymbol{w}_n. \end{aligned} \quad (1.23)$$

Location operators for adjacent elements may extract common values from $\boldsymbol{y}_n$ and $\boldsymbol{w}_n$ in order to satisfy the continuity condition. Note from Equation 1.20 that there are only two elements that have a non-zero value on the boundary: ψ_0 and ψ_P. This could have one believe that the continuity constraint affects only a small subset of the final collection of basis functions. In multiple dimensions this is not true. In fact most basis functions will be non-zero on at least one edge of the finite element domain. For instance in Equation 1.19, if $\psi_p \neq 0$ on the edge $\xi_1 = 1$, all combinations of ϕ_q, ϕ_r and ϕ_s will produce basis functions that have non-zero values on this edge.

Combining Equations 1.18 and 1.23, the test functions $\boldsymbol{W}$ can be expressed in terms of their global vectors $\boldsymbol{w}_n$:

$$\mathcal{V} \ni \boldsymbol{W}\Big|_{Q_{(n,i)}} = \left(\sum_{a\in\mathcal{A}} N_{(n,i,a)}\boldsymbol{M}_{(n,i,a)}\right)\boldsymbol{w}_n. \quad (1.24)$$

The small-scale functions $\tilde{\boldsymbol{W}}$ are identical, except that the polynomials are summed over the subset $\tilde{\mathcal{A}}$ only. These expressions for $\boldsymbol{W}$ and $\tilde{\boldsymbol{W}}$ can be substituted into the rewritten system of equations, Equation 1.21, but this requires the integrals to be split into subintegrals over the separate finite elements because the polynomials $N_{(n,i,a)}$ are defined per element $Q_{(n,i)}$ only. For brevity, these subintegrals will be denoted as vectors $\boldsymbol{v}_{(n,i,a)}$ and $\tilde{\boldsymbol{v}}_{(n,i,a)}$, the former dependent on $\boldsymbol{Y}$, the latter on $\tilde{\boldsymbol{Y}}$. When $\mathcal{I}_n$ denotes the index set of elements in time-slab n, Equation 1.21 transforms to:

$$\boldsymbol{w}_n^T \sum_{i \in \mathcal{I}_n} \left\{ \sum_{a \in \mathcal{A}} \boldsymbol{M}_{(n,i,a)}^T \boldsymbol{v}_{(n,i,a)} + \sum_{a \in \tilde{\mathcal{A}}} \boldsymbol{M}_{(n,i,a)}^T \tilde{\boldsymbol{v}}_{(n,i,a)} \right\} = 0 \quad \forall\ \boldsymbol{w}_n \in \mathbb{R}^N, \tag{1.25}$$

or, more compactly:

$$\boldsymbol{w}_n^T \boldsymbol{G}(\boldsymbol{y}_n, \boldsymbol{y}_{n-1}) = 0 \quad \forall\ \boldsymbol{w}_n \in \mathbb{R}^N, \tag{1.26}$$

where $\boldsymbol{G} : \mathbb{R}^N \times \mathbb{R}^N \rightarrow \mathbb{R}^N$ is a non-linear operator, composed of all separate element integrals $\boldsymbol{v}_{(n,i,a)}$ and $\tilde{\boldsymbol{v}}_{(n,i,a)}$ that are put in place by the location operators $\boldsymbol{M}_{(n,i,a)}$. Its dependence on the previous time-slab solution $\boldsymbol{y}_{n-1}$ is caused by the jump condition, Equation 1.22. With operator $\boldsymbol{G}$ defined, the requirement that $\boldsymbol{w}_n^T \boldsymbol{G}$ is zero for any vector $\boldsymbol{w}_n \in \mathbb{R}^N$ leads to the final, non-linear system of N equations:

$$\boldsymbol{G}(\boldsymbol{y}_n, \boldsymbol{y}_{n-1}) = \boldsymbol{0}. \tag{1.27}$$

Addendum

For sake of completeness, the complete expressions of the subintegrals $\boldsymbol{v}_{(n,i,a)}$ and $\tilde{\boldsymbol{v}}_{(n,i,a)}$ are presented below. They will not be used further in this report.

$$\begin{aligned}
\boldsymbol{v}_{(n,i,a)} = &\int_{\Omega_{(n,i)}} \left(N_{(n,i,a)}|_{t_{n+1}^-} \boldsymbol{U}|_{t_{n+1}^-} - N_{(n,i,a)}|_{t_n^+} \boldsymbol{U}|_{t_n^-} \right) \mathrm{d}\Omega \\
&+ \int_{P_{(n,i)}} N_{(n,i,a)} \boldsymbol{F}_\alpha n_\alpha \ \mathrm{d}P \\
&- \int_{Q_{(n,i)}} \left(N_{(n,i,a),t} \boldsymbol{U} + N_{(n,i,a),\alpha} \boldsymbol{F}_\alpha + N_{(n,i,a)} \boldsymbol{S} \right) \mathrm{d}Q,
\end{aligned}$$

$$\begin{aligned}
\tilde{\boldsymbol{v}}_{(n,i,a)} = &\int_{P_{(n,i)}} N_{(n,i,a)} \tilde{\boldsymbol{F}}_\alpha^m n_\alpha \ \mathrm{d}P \\
&- \int_{Q_{(n,i)}} N_{(n,i,a),\alpha} \tilde{\boldsymbol{F}}_\alpha^m \ \mathrm{d}Q.
\end{aligned}$$

1.4 Solving the non-linear system

Non-linear equations such as Equation 1.27 are usually solved using an iterative solution method. Such an iterative method starts with an initial guess $\boldsymbol{y}_n^0$ and uses a recurrence relation to construct a sequence of approximations $\boldsymbol{y}_n^k$ that converges to the exact solution $\boldsymbol{y}_n$ — assuming that this solution is unique. The iterations can be stopped when $\boldsymbol{y}_n^k$ is considered a good enough approximation of $\boldsymbol{y}_n$:

$$\|G(\boldsymbol{y}_n^k, \boldsymbol{y}_{n-1})\| < \epsilon, \tag{1.28}$$

where $\|\cdot\|$ is an appropriately chosen norm and ϵ is a measure of the desired precision. The solution of the preceding time-slab $\boldsymbol{y}_{n-1}$ is known. An obvious choice is to use this previous solution as a starting vector $\boldsymbol{y}_n^0$ for the new iterations as well. Note, however, that the jump condition from Section 1.3.2 does not require meshes on successive time-slabs to correspond in any way. Even if $\boldsymbol{y}_{n-1}$ has the same length as $\boldsymbol{y}_n$, it can represent a completely different flow on both time-slabs. This can be a point of concern for example for VMS implementations that deal with time-changing domains.

A widely used root-finding algorithm is **Newton's method**, which is based on a Taylor expansion. For a scalar function $f : \mathbb{R} \to \mathbb{R}$, the first two terms of this expansion read:

$$f(x+h) = f(x) + hf'(x) + \mathcal{O}(h^2), \tag{1.29}$$

where $\mathcal{O}$ is Landau's order symbol. If $x+h$ is a root of f, and x is an approximation of this root, the error h will be small. Under these assumptions, the above expression can be rewritten as:

$$h \approx -\frac{f(x)}{f'(x)}. \tag{1.30}$$

Here h is an approximation of the error, because the high order terms of the Taylor expansion have been neglected. Therefore, adding this error to an approximate solution will not result in an exact root of f. It will, however, result in a much improved approximation of this root. Each time that this procedure is repeated the error decreases, which justifies the neglection of high order terms even more. The resulting, rapidly converging algorithm is described by the following recurrence relation:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}. \tag{1.31}$$

This recurrence depends only on the most recent approximation x_k, which means that previous approximations do not have to be kept in memory. This **short recurrence property** becomes important when Newton's method is generalized to vector functions, where arguments can be really large. For such functions the derivative is replaced with a Jacobian matrix. Applied to Equation 1.27, Newton's method reads:

$$\boldsymbol{y}_n^{k+1} = \boldsymbol{y}_n^k - \left(\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{y}_n}(\boldsymbol{y}_n^k, \boldsymbol{y}_{n-1})\right)^{-1} \boldsymbol{G}(\boldsymbol{y}_n^k, \boldsymbol{y}_{n-1}), \tag{1.32}$$

where $\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{y}_n} \in \mathbb{R}^{N\times N}$ is the Jacobian matrix of $\boldsymbol{G}$. Theoretically, for every Newton iteration a new Jacobian matrix will need to be formed around the updated solution. This can be a lot of work. Fortunately, in practice the matrix can be reused for

multiple Newton iterations; the solution will simply converge a little less fast. For VMS the matrix is even reused for multiple successive time-slabs.

Whether the Jacobian matrix is updated or not, every Newton iteration will need to solve a linear system involving this matrix. For this, a large number of iterative solution methods are available. However, many of those demand special properties of the matrix, like symmetry, or they are known to work especially well or to break down under certain conditions. To find out which solvers apply to the system at hand, more information will be required for the Jacobian of $\boldsymbol{G}$. By Equations 1.25 and 1.26, the Jacobian matrix can be written as:

$$\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{y}_n} = \sum_{i \in \mathcal{I}_n} \Big\{ \sum_{a \in \mathcal{A}} \boldsymbol{M}^T_{(n,i,a)} \frac{\partial \boldsymbol{v}_{(n,i,a)}}{\partial \boldsymbol{y}_n} + \sum_{a \in \tilde{\mathcal{A}}} \boldsymbol{M}^T_{(n,i,a)} \frac{\partial \tilde{\boldsymbol{v}}_{(n,i,a)}}{\partial \boldsymbol{y}_n} \Big\}, \tag{1.33}$$

where $\frac{\partial \boldsymbol{v}_{(n,i,a)}}{\partial \boldsymbol{y}_n} \in \mathbb{R}^{5 \times N}$ and $\frac{\partial \tilde{\boldsymbol{v}}_{(n,i,a)}}{\partial \boldsymbol{y}_n} \in \mathbb{R}^{5 \times N}$ are the Jacobian matrices of $\boldsymbol{v}_{(n,i,a)}$ and $\tilde{\boldsymbol{v}}_{(n,i,a)}$, respectively. Since $\boldsymbol{v}_{(n,i,a)}$ and $\tilde{\boldsymbol{v}}_{(n,i,a)}$ consist of integrals over a single finite element i, only a subset of $\boldsymbol{y}_n$ — the element vectors $\boldsymbol{y}_{(n,i,b)}$ — produces non-zero derivatives. Hence, the Jacobian of $\boldsymbol{v}_{(n,i,a)}$ can be written as:

$$\frac{\partial \boldsymbol{v}_{(n,i,a)}}{\partial \boldsymbol{y}_n} = \sum_{b \in \mathcal{A}} \frac{\partial \boldsymbol{v}_{(n,i,a)}}{\partial \boldsymbol{y}_{(n,i,b)}} \boldsymbol{M}_{(n,i,b)}, \tag{1.34}$$

where $\frac{\partial \boldsymbol{v}_{(n,i,a)}}{\partial \boldsymbol{y}_{(n,i,b)}} \in \mathbb{R}^{5 \times 5}$ is a subset of the full Jacobian matrix, corresponding to the derivatives of $\boldsymbol{y}_{(n,i,b)}$. The location operator moves its five columns to their positions in the full Jacobian matrix. A similar expression holds for $\tilde{\boldsymbol{v}}_{(n,i,a)}$, in which b can even be restricted to $\tilde{\mathcal{A}}$ because $\tilde{\boldsymbol{v}}_{(n,i,a)}$ depends only on the small scales $\tilde{\boldsymbol{Y}}$. Substitution in Equation 1.33 yields a new expression for the Jacobian matrix:

$$\begin{aligned} \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{y}_n}(\boldsymbol{y}_n, \boldsymbol{y}_{n-1}) = \sum_{i \in \mathcal{I}_n} \Big\{ & \sum_{(a,b) \in \mathcal{A}^2} \boldsymbol{M}^T_{(n,i,a)} \frac{\partial \boldsymbol{v}_{(n,i,a)}}{\partial \boldsymbol{y}_{(n,i,b)}} \boldsymbol{M}_{(n,i,b)} \\ & + \sum_{(a,b) \in \tilde{\mathcal{A}}^2} \boldsymbol{M}^T_{(n,i,a)} \frac{\partial \tilde{\boldsymbol{v}}_{(n,i,a)}}{\partial \boldsymbol{y}_{(n,i,b)}} \boldsymbol{M}_{(n,i,b)} \Big\} \end{aligned} \tag{1.35}$$

This expression shows that the Jacobian has a symmetric sparsity pattern, because for every element (a, b) in $\mathcal{A}^2$ there is an element (b, a) as well. However, since $\frac{\partial \boldsymbol{v}_{(n,i,a)}}{\partial \boldsymbol{y}_{(n,i,b)}}$ and $\frac{\partial \boldsymbol{v}_{(n,i,b)}}{\partial \boldsymbol{y}_{(n,i,a)}}$ are no mutual transposes, symmetry for its contents can not be guaranteed. Indeed, the Jacobian matrix is not symmetric in general. This excludes a lot of solution methods from the list of candidates, such as the popular Conjugate Gradient method that demands the matrix to be symmetric and positive definite. Instead, the VMS method will require a solver that has general applicability.

Chapter 2

The deflation method

Finite dimensional linear systems occur in a many different fields of mathematics. They can for example represent differential equations, discretized through finite element, volume or difference methods. Or they can be linear least squares or optimal control problems. Regardless of the underlying problem, though, finite dimensional linear systems can always be put in the following form:

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}. \tag{2.1}$$

Finding a solution $\boldsymbol{x}$ is the terrain of linear solution methods — **solvers**, in short. In the course of time a wide range of different solvers has been developed. Initially, the solution algorithms were designed mainly with round-off errors and numerical robustness in mind. The recent widespread availability of relatively low-cost computing power, however, has boosted the desire to analyze the mathematical models much more accurately, leading to very large linear systems. This development has forced attention towards new bottlenecks such as memory usage, computation time and parallel efficiency. For these reasons, most current-day applications depend on iterative methods to solve their systems, in particular **Krylov subspace methods** which are considered the most powerful iterative methods currently available. One of the more recent developments is to accelerate these Krylov methods even further by a technique called **deflation**, subject of this chapter.

This chapter will give an overview of the most important linear solution methods, starting with direct methods in Section 2.1. The most straightforward of iterative methods, appropriately named basic iterative methods, are treated in Section 2.2, followed by the more advanced Krylov subspace methods in Section 2.3. Thereafter it will be a small step towards deflated Krylov methods, which — other than the name suggests — can actually be thought of as enriched Krylov methods. This is explained in Section 2.4, which concludes this chapter. The subsections will contain more detailed information that is not strictly required to follow the line of reasoning in this chapter.

2.1 Direct methods

The $n \times n$ system $\boldsymbol{Ax} = \boldsymbol{b}$ is short notation for the following system of n linear equations and n unknowns:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n. \end{aligned} \tag{2.2}$$

A straightforward way of solving this system is to start with the first equation and solve it for x_1 in terms of $x_2, \ldots, x_n$. Next, the second equation can be solved for x_2 in terms of $x_3, \ldots, x_n$, using the previously obtained expression for x_1. This can be repeated until at equation n the solution for x_n is obtained. At this point the direction reverses, and the remaining values are computed by subsequent substitution of the computed values into the generated expressions for x_{n-1}, x_{n-2} down to x_1. The thus defined two-stage algorithm is known as the **Gaussian elimination method**, which is the method of choice for small, dense systems.

An important observation in the above algorithm is that all operations are independent of the right hand side $\boldsymbol{b}$. It is therefore possible to compute the factors of subsequent additions once and reuse them for different right hand sides. It follows that these factors can be placed conveniently in a lower triangular matrix $\boldsymbol{L}$ and upper triangular matrix $\boldsymbol{U}$ such that $\boldsymbol{LU} = \boldsymbol{A}$, which transforms Equation 2.1 into a pair of triangular systems:

$$\boldsymbol{Ly} = \boldsymbol{b}, \quad \boldsymbol{Ux} = \boldsymbol{y}. \tag{2.3}$$

Solving the first and second equation corresponds to the forward and backward substitution phase of the Gaussian elimination method, respectively. Gaussian elimination and the thus defined **LU decomposition method** are hence mathematically equivalent.

For sparse matrices often large memory savings can be made by storing only the values and positions of the non-zero elements. A major problem with direct methods such as LU-decomposition is that they will change at least some of the zero-valued elements to non-zero, resulting in increased memory usage. This phenomenon is known as **matrix fill-in**. The amount of fill-in depends on the sparsity structure of the matrix; LU-decomposition fills at most the elements within the **matrix profile**, shown in Figure 2.1. This is markedly better than a full matrix inversion, during which the entire matrix will be filled. Moreover, fill-in can be reduced by strategic row and column permutations. However, it will still require extra memory, and for very large matrices this may be unacceptable. Iterative methods have no fill-in at all, and are therefore often a better choice in such situations.

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{13} & & & & a_{16} \\ & a_{22} & & & & \\ & & a_{33} & & & \\ a_{41} & & & a_{44} & & \\ & & & & a_{55} & \\ & & a_{63} & & & a_{66} \end{bmatrix}$$

Figure 2.1. *Profile structure of a sparse 6×6 matrix. Non-printed elements are zero. During LU-decomposition, only the zero elements within the gray bands can be filled in.*

2.1.1 Pivoting

An important point of concern for the Gaussian elimination or LU decomposition methods is the possibility of breakdown. Already in the first step, if a_{11} is zero, the first equation can not be solved in terms of x_1 and the algorithm breaks down. The solution to this problem is to reorder the equations such that this situation can not occur. If, for example, a_{21} and a_{12} are non-zero, the second equation can be solved for x_1 and the first for x_2 instead. This reordering of equations to prevent breakdown is called **partial pivoting**. Preventing breakdown through partial pivoting is always possible, but this will require extra work.

Preventing breakdown is not the only reason to apply pivoting. It is also useful for reducing the accumulation of round-off errors, which in certain situations can be substantial. Partial pivoting helps in this situation as well, and leads to a stable algorithm in most practical situations. Even better than partial pivoting, which uses row interchanges only, would be to use a combination of both row and column interchanges. The extra costs of such a **complete pivoting** algorithm, however, are so high that its use can not normally be justified. In practice, partial pivoting is often found to perform well enough.

An important thing to note about pivoting is that it will change the matrix profile, and therefore the amount of fill-in during LU-decomposition. For certain types of matrices this can lead to a substantial increase of memory. An example are banded matrices, that are efficiently stored using a fixed subset of diagonals. Pivoting destroys this structure, which means that a different type of storage must be used. In these situations, pivoting is to be avoided as much as possible.

2.2 Basic iterative methods

Direct methods are particularly well suited for solving small and dense systems. Many practical applications, however, deal with systems that are instead large and sparse. Efficient storage of these matrices allows problems to grow in size far beyond the point where a full matrix would reach a memory limit. Using a direct

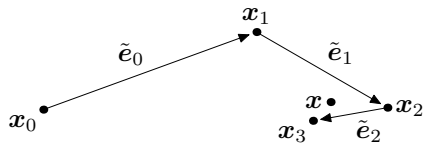

Figure 2.2. *Example convergence behaviour of a basic iterative method applied to a two-dimensional system. In each iteration i, the search vector $\tilde{\boldsymbol{e}}_i$ points more or less at the exact solution $\boldsymbol{x}$.*

method such as LU-decomposition, however, the sparse matrices will inevitably suffer a huge amount of fill-in of zero elements, thereby increasing the required memory for storage. The result is that the size of the problem will be limited by the memory required by the solver, rather than the matrix itself. Clearly, this situation is unsatisfactory.

A second problem that follows from the use of direct methods involves computation time. For a dense $n \times n$ matrix, the number of floating point operations required for performing a full LU-decomposition scales with n^3. For sparse matrices this will be somewhat better, depending on the level of sparsity, but the total workload and hence computation time will still increase rapidly with size. This work then results in a solution that is almost machine-accurate, which may be a lot more then required. In practice, linear systems are often inaccurate models of the truth, and do not have a really meaningful exact solution. An approximate solution is in general good enough, and should be a lot easier to obtain.

Iterative methods solve both of these problems. These methods are based solely on matrix-vector multiplication operation and do not require access to single elements of the matrix. This matrix can therefore be stored very efficiently. In fact, the matrix need not be stored at all, as its elements can be generated on the fly during the multiplication operation. Such a method is called **matrix-free**. Using only multiplications, each iteration produces a new approximate solution. The algorithm can therefore be stopped as soon as the desired accuracy is reached. Depending on the problem at hand and the iterative method used, the resulting computation time can be much lower than that of a direct method.

Of all iterative methods, the **basic iterative methods** are the simplest, yet least powerful. Their operation is based on the following observation. Suppose an approximate solution $\tilde{\boldsymbol{x}}$ is available for Equation 2.1, $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$. The **residual** vector at this point is defined as $\boldsymbol{r} = \boldsymbol{b} - \boldsymbol{A}\tilde{\boldsymbol{x}}$. To obtain the exact solution, the vector that should be added to $\tilde{\boldsymbol{x}}$ is the error $\boldsymbol{e} = \boldsymbol{A}^{-1}\boldsymbol{r}$. Clearly, an exact computation of this error is not feasible since this requires solving the system $\boldsymbol{A}\boldsymbol{e} = \boldsymbol{r}$, which is identical to Equation 2.1. However, if a non-singular, cheaply invertible approximation of matrix $\boldsymbol{A}$ is available, denoted $\boldsymbol{M}$, then this matrix can be used to compute an approximation $\tilde{\boldsymbol{e}}$ of the error. By repeatedly updating $\tilde{\boldsymbol{x}}$ with this approximate error, the algorithm is expected to converge to the exact solution $\boldsymbol{x}$.

Figure 2.2 shows a possible first three iterations of this algorithm. Starting with an initial guess $\boldsymbol{x}_0$, the first iterate $\boldsymbol{x}_1$ is formed by adding the approximate error $\tilde{\boldsymbol{e}}_0$. At this point a new approximate error $\tilde{\boldsymbol{e}}_1$ is calculated and added to $\boldsymbol{x}_1$, yielding

the second iterate $\boldsymbol{x}_2$. Continuing this process, the complete algorithm is described by the following few lines of pseudocode:

Algorithm 2.1

1. start: $\boldsymbol{x}_0$
2. for i in $0, 1, 2, \ldots$:
3. $\quad \boldsymbol{r}_i \leftarrow \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_i$
4. $\quad$ if $\|\boldsymbol{r}_i\|_2 <$ threshold:
5. $\quad\quad$ break
6. $\quad$ end if
7. $\quad \tilde{\boldsymbol{e}}_i \leftarrow \boldsymbol{M}^{-1}\boldsymbol{r}_i$
8. $\quad \boldsymbol{x}_{i+1} \leftarrow \boldsymbol{x}_i + \tilde{\boldsymbol{e}}_i$
9. end for

The crucial points in this algorithm are lines 1, 4 and 7. Line 1 defines the **starting vector** $\boldsymbol{x}_0$, which is used to supply the algorithm preliminary information about the solution. It depends on the problem that is solved if such information is available. In case of a time dependent problem, for instance, information from previous timesteps can often be used. In other situations the solution of a related problem can be a useful start, such as an analytic solution of a simplified problem, or a solution from a coarser grid. The possibility to use this preliminary information as a starting point of the iterative procedure is a great advantage of iterative methods over direct methods, that have no way of utilizing this information. When no sensible initial guess can be made, starting from zero is always possible.

Line 4 is the **termination criterion**, which determines when the iterative process can be stopped. Ideally, this is when a certain precision has been reached, i.e. when the error is small enough. However, since a direct measurement of this error is not possible, the termination criterion will have to rely on indirect measurements instead. The above algorithm terminates when the residual norm drops below a certain threshold. This threshold should be chosen with great care. If it is chosen too weak the solution will be meaningless, whereas if it is too severe the algorithm will become very costly and may not even converge. Sometimes the residual can be related to a physical quantity that gives information about the accuracy of the solution. In other situations, however, with no clear relation between error and residual, determining a suitable threshold is hard.

Lastly, line 7 computes the next **search vector** $\tilde{\boldsymbol{e}}_i$. The quality of this vector depends on the quality of the approximate solver $\boldsymbol{M}^{-1}$, usually called a **preconditioner** for matrix $\boldsymbol{A}$ because it lowers the **condition number**, the fraction of extreme eigenvalues, by clustering the eigenvalues of $\boldsymbol{M}^{-1}\boldsymbol{A}$ around one. In general, the more densely clustered, the faster the algorithm converges. Clearly, in the extreme case that $\boldsymbol{M} = \boldsymbol{A}$, all eigenvalues are exactly one and the algorithm converges in a single iteration. On the other extreme, when the eigenvalues of $\boldsymbol{M}^{-1}\boldsymbol{A}$ are widely scattered the algorithm will not converge at all. The following theorem states an exact condition for preconditioner $\boldsymbol{M}^{-1}$, illustrated by Figure 2.3, under which Algorithm 2.1 will yield a convergent process.

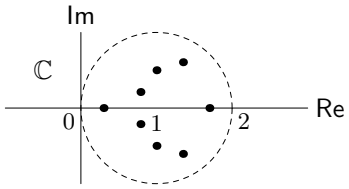

Figure 2.3. *Example eigenvalue distribution of a preconditioned matrix that yields a convergent basic iterative method, according to Theorem 2.1.*

Theorem 2.1. *The iterates $\boldsymbol{x}_i$ of Algorithm 2.1 converge to the exact solution of equation $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, for any starting vector $\boldsymbol{x}_0$, if all eigenvalues of the matrix $\boldsymbol{M}^{-1}\boldsymbol{A}$ are contained in the open unit sphere centered around one, i.e. if*

$$|1 - \lambda| < 1 \quad \forall\ \lambda \in \text{spectrum } \boldsymbol{M}^{-1}\boldsymbol{A}. \tag{2.4}$$

Proof. First the following is proved by induction:

$$\boldsymbol{e}_n = (\boldsymbol{I} - \boldsymbol{M}^{-1}\boldsymbol{A})^n \boldsymbol{e}_0, \tag{2.5}$$

where $\boldsymbol{e}_i = \boldsymbol{x} - \boldsymbol{x}_i$ is the exact error at iteration i. For $n = 0$ the statement is clearly true. Assuming that the statement is true for $n = i$, it follows that $\boldsymbol{x}_i = \boldsymbol{x} - (\boldsymbol{I} - \boldsymbol{M}^{-1}\boldsymbol{A})^n \boldsymbol{e}_0$. The next iterate $\boldsymbol{x}_{i+1}$ is generated by a single iteration of Algorithm 2.1:

3. $\boldsymbol{r}_i \leftarrow \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_i = \boldsymbol{A}(\boldsymbol{I} - \boldsymbol{M}^{-1}\boldsymbol{A})^i \boldsymbol{e}_0$
7. $\tilde{\boldsymbol{e}}_i \leftarrow \boldsymbol{M}^{-1}\boldsymbol{r}_i = \boldsymbol{M}^{-1}\boldsymbol{A}(\boldsymbol{I} - \boldsymbol{M}^{-1}\boldsymbol{A})^i \boldsymbol{e}_0$
8. $\boldsymbol{x}_{i+1} \leftarrow \boldsymbol{x}_i + \tilde{\boldsymbol{e}}_i = \boldsymbol{x} - (\boldsymbol{I} - \boldsymbol{M}^{-1}\boldsymbol{A})^{i+1} \boldsymbol{e}_0.$

This proves Equation 2.5 for $n = i + 1$, hence, by induction, for general $n \geq 0$. Using Lemma 7.3.1 from Golub et al. [5], it follows that the error converges to zero if all eigenvalues of $\boldsymbol{I} - \boldsymbol{M}^{-1}\boldsymbol{A}$ are smaller than one in absolute value. If λ is an eigenvalue of $\boldsymbol{M}^{-1}\boldsymbol{A}$ then $1 - \lambda$ is an eigenvalue of $\boldsymbol{I} - \boldsymbol{M}^{-1}\boldsymbol{A}$, therefore this proves Equation 2.4. □

Summarizing, basic iterative methods are controlled by three properties. The starting vector controls the precision of the first iteration, the termination criterion controls the precision of the last, and the preconditioner controls the number of iterations in between. The preconditioner is usually the identifying element that gives a basic iterative method its name. Many such iterative methods have been developed, but they have long since been surpassed by more advanced iterative methods such as Krylov subspace methods. In current day applications, the use of basic iterative methods as standalone solvers is negligible.

2.2.1 Preconditioners

A preconditioner $\boldsymbol{M}^{-1}$ is a matrix that approximates the inverse of a matrix $\boldsymbol{A}$, in the sense that the eigenvalues of $\boldsymbol{M}^{-1}\boldsymbol{A}$ are clustered around one. Like the matrix itself, the preconditioner is used only in the context of matrix-vector multiplications, so its individual elements are never accessed. In practice, if a matrix is stored at all, it will be the non-inverted matrix $\boldsymbol{M}$ since that will in general inherit the sparsity of $\boldsymbol{A}$. The multiplication operation then becomes a linear solution procedure. An important requirement for preconditioners is that this procedure can be performed at very low cost, because otherwise any positive effect on convergence will be offset by the increased computational costs.

The usual approach to preconditioners in basic iterative methods is to view them as part of an **operator splitting**. This means that $\boldsymbol{A}$ is written as the sum of two matrices $\boldsymbol{M}$ and $\boldsymbol{N}$, transforming the original system to $\boldsymbol{M}\boldsymbol{x} = \boldsymbol{b} - \boldsymbol{N}\boldsymbol{x}$. When the right hand side is formed from the most recent approximation of $\boldsymbol{x}$ this yields an iterative process that is equivalent to Algorithm 2.1. Theorem 2.1 shows that this process is convergent when the spectral radius of $\boldsymbol{M}^{-1}\boldsymbol{N}$ is strictly smaller than one. This forms the basis of many specific convergence results available for basic iterative methods, two of which are mentioned below.

When $\boldsymbol{M}$ consists only of the diagonal of $\boldsymbol{A}$, $\boldsymbol{M}^{-1}$ is called a **diagonal scaling** preconditioner. A basic iterative method based on this preconditioner is called **Gauss Jacobi iteration**. Clearly, this preconditioner is very sparse and cheap. Less obvious is the extend to which it approximates $\boldsymbol{A}^{-1}$. It is not hard to see, however, that the iterates generated by Algorithm 2.1 using this preconditioner are the following. Looking at System 2.2, the elements of the next iterate are obtained from solving the first equation for x_1, the second for x_2, and so on to x_n, using the remaining elements x_j from the previous iteration. It seems plausible that this can yield a convergent method. Theorem 4.5 by Saad [16] shows that this is the case for example when matrix $\boldsymbol{A}$ is strictly diagonal dominant.

The above procedure can be improved by using the most recently computed values for $x_1, \ldots, x_{j-1}$ when solving equation j. This corresponds to defining $\boldsymbol{M}$ as the lower triangle of $\boldsymbol{A}$. The resulting preconditioner can again be applied cheaply because a triangular system can be solved at low cost, even more so because this system inherits the sparsity of $\boldsymbol{A}$. An iterative method based on this preconditioner is called **Gauss Seidel iteration**. As expected, Gauss Seidel converges in general faster that Gauss Jacobi. Moreover, Theorem 10.1.2 by Golub et al. [5] shows that Gauss Seidel always yields a convergent process when applied to the very common class of symmetric, positive definite matrices.

A generalisation of Gauss Jacobi and Gauss Seidel iteration is **successive over-relaxation**, which can be seen as a linear combination of both: $\boldsymbol{M} = \frac{\omega-1}{\omega}\boldsymbol{M}_{GJ} + \boldsymbol{M}_{GS}$, for some constant $\omega \in \mathbb{R}$. The reason behind this is that Gauss Seidel is often found to underpredict the error. Successive over-relaxation extrapolates the computed error by a factor ω and is thus expected to yield faster convergence, if ω is suitably chosen. Being a generalization of Gauss Seidel, it is certain that it can always be at least as fast. However, finding the optimal value for ω is usually hard.

2.3 Krylov subspace methods

In the preceding text on preconditioners it was noted that the Gauss Seidel preconditioner often underestimates the size of the error, and should therefore improve from extrapolation. In general it will be true that search vectors can be improved by scaling. In Figure 2.2, for example, the iterates $\boldsymbol{x}_2$ and $\boldsymbol{x}_3$ would be much closer to the exact solution $\boldsymbol{x}$ if the search vectors $\tilde{\boldsymbol{e}}_1$ and $\tilde{\boldsymbol{e}}_2$ had been added scaled down, while $\boldsymbol{x}_1$ would have benefitted from scaling $\tilde{\boldsymbol{e}}_0$ up. The problem here is to find the optimal scaling factor, as this factor should minimize the distance to a solution that is not known at the time of scaling.

The solution to this problem is to use a different measure. In the above claim that $\tilde{\boldsymbol{e}}_1$ and $\tilde{\boldsymbol{e}}_2$ should have been scaled down to bring the iterates closer to $\boldsymbol{x}$ it was tacitly assumed that the distance is measured in the Euclidean norm, or 'with a ruler'. Other norms exist, however, that are equilly well suited for measuring distances and some of those are indeed capable of minimizing the distance to the unknown solution $\boldsymbol{x}$. Using such a norm it will be possible to compute the optimal scaling of an arbitrary set of search vectors. This scaling procedure forms the basis of the popular class of **Krylov subspace methods**.

A Krylov subspace method, like a basic iterative method, starts with an initial guess $\boldsymbol{x}_0$. A preconditioner is used to compute the approximate error $\tilde{\boldsymbol{e}}_0$, the first search vector. Prior to adding this vector to $\boldsymbol{x}_0$, however, it is scaled to move the result closest to $\boldsymbol{x}$, measured in a suitable norm. At this point $\boldsymbol{x}_1$ a new search vector $\tilde{\boldsymbol{e}}_1$ is computed, which is different and expectedly more accurate than its basic iterative counterpart. This time, instead of simply scaling the vector, the Krylov method chooses a combination of both search vectors that brings $\boldsymbol{x}_2$ closest to $\boldsymbol{x}$. See also Figure 2.4, the Krylov counterpart of Figure 2.2. For larger problems than this one, the updates that follow are constructed from the growing subspace of search vectors. This leads to the following algorithm:

Algorithm 2.2

1. start: $\boldsymbol{x}_0$
2. for $i = 0, 1, 2, \ldots$:
3. $\quad \boldsymbol{r}_i \leftarrow \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_i$
4. $\quad$ if $\|\boldsymbol{r}_i\| <$ threshold:
5. $\quad\quad$ break
6. $\quad$ end if
7. $\quad \tilde{\boldsymbol{e}}_i \leftarrow \boldsymbol{M}^{-1}\boldsymbol{r}_i$
8. $\quad \boldsymbol{x}_{i+1} \leftarrow \boldsymbol{x}_i + \mathbb{P}_{\mathrm{span}\{\tilde{\boldsymbol{e}}_0,\ldots,\tilde{\boldsymbol{e}}_i\}}(\boldsymbol{x} - \boldsymbol{x}_i)$
9. end for

Compared to the basic iterative method, Algorithm 2.1, the only thing that has changed is line 8. That is where the iterate $\boldsymbol{x}_i$ is updated by adding a linear combination of search vectors. The optimal update element can be viewed as a projection of the exact error onto the subspace spanned by the search vectors, such that it is closest measured in the chosen norm. This projection is defined as follows:

$$\mathbb{P}_{\mathcal{V}} : \mathcal{U} \to \mathcal{V}, \quad \boldsymbol{u} \to \underset{\boldsymbol{v}\in\mathcal{V}}{\mathrm{argmin}} \|\boldsymbol{v} - \boldsymbol{u}\|, \tag{2.6}$$

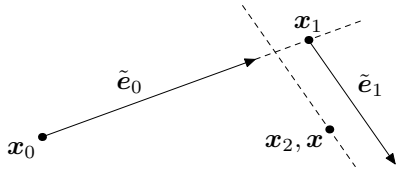

Figure 2.4. *Example convergence behaviour of a Krylov subspace method applied to a two-dimensional system. Compare with Figure 2.2. Again, the search vectors point more or less at the exact solution $\boldsymbol{x}$, but this time only directions are used. Since the search vectors $\tilde{\boldsymbol{e}}_0$ and $\tilde{\boldsymbol{e}}_1$ span $\mathbb{R}^2$, the method converges in two iterations.*

in which argmin return the element v that minimizes its argument. With $\mathbb{P}$ defined like this, line 8 shows that the difference vector is a linear combination of search vectors; $\boldsymbol{x}_{i+1} - \boldsymbol{x}_i \in \operatorname{span}\{\tilde{\boldsymbol{e}}_0, \ldots, \tilde{\boldsymbol{e}}_i\}$ for all i. When $\boldsymbol{x}_{i+1} - \boldsymbol{x}_0$ is written as a telescoping sum, $(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i) + \cdots + (\boldsymbol{x}_1 - \boldsymbol{x}_0)$, it is clear that $\boldsymbol{x}_{i+1} - \boldsymbol{x}_0 \in \operatorname{span}\{\tilde{\boldsymbol{e}}_0, \ldots, \tilde{\boldsymbol{e}}_i\}$ for all i. A straightforward shift argument, $\operatorname{argmin}_{\boldsymbol{v}\in\mathcal{V}} f(\boldsymbol{v}) = \boldsymbol{w} + \operatorname{argmin}_{\boldsymbol{v}\in\mathcal{V}} f(\boldsymbol{v} + \boldsymbol{w})$ for $\boldsymbol{w} \in \mathcal{V}$, now transforms line 8 into the following somewhat more usable expression:

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_0 + \mathbb{P}_{\operatorname{span}\{\tilde{\boldsymbol{e}}_0,\ldots,\tilde{\boldsymbol{e}}_i\}}(\boldsymbol{x} - \boldsymbol{x}_0). \tag{2.7}$$

Written in this form, it is clear that the iterate $\boldsymbol{x}_i$ is composed of the initial guess $\boldsymbol{x}_0$ plus an approximate error, composed of the first i search vectors such that it is closest to the exact initial error $\boldsymbol{x} - \boldsymbol{x}_0$ measured in the chosen norm. Surprisingly, the following theorem states that this norm in fact does not influence the subspace that is spanned by the search vectors, only the iterates that are formed from it. The theorem shows that regardless of the norm, the first i search vectors span a **Krylov subspace** of dimension i, which is defined as follows:

$$\mathcal{K}_i(\boldsymbol{A}, \boldsymbol{b}) \stackrel{\text{def}}{=} \operatorname{span}\{\boldsymbol{A}^{i-1}\boldsymbol{b}, \ldots, \boldsymbol{A}^2\boldsymbol{b}, \boldsymbol{A}\boldsymbol{b}, \boldsymbol{b}\}. \tag{2.8}$$

Theorem 2.2. *The subspace spanned by the first n search vectors in Algorithm 2.2 is independent of the norm used in projection $\mathbb{P}_{\mathcal{V}}$, and equals the Krylov subspace of dimension n corresponding to matrix $\boldsymbol{M}^{-1}\boldsymbol{A}$ and initial residual $\boldsymbol{M}^{-1}\boldsymbol{r}_0$:*

$$\operatorname{span}\{\tilde{\boldsymbol{e}}_0, \ldots, \tilde{\boldsymbol{e}}_{n-1}\} = \mathcal{K}_n(\boldsymbol{M}^{-1}\boldsymbol{A}, \boldsymbol{M}^{-1}\boldsymbol{r}_0). \tag{2.9}$$

Proof. The Krylov subspace of dimension n will be written compactly as $\mathcal{K}_n$. From Algorithm 2.2 it follows directly that $\tilde{\boldsymbol{e}}_0 = \boldsymbol{M}^{-1}\boldsymbol{r}_0$, so the theorem is true for $n = 1$. Left to prove is the relation $\mathcal{K}_n \oplus \operatorname{span}\{\tilde{\boldsymbol{e}}_n\} = \mathcal{K}_{n+1}$ for general n. By induction, assuming that this relation holds for all $n < i$, it suffices to prove the case $n = i$: $\mathcal{K}_i \oplus \operatorname{span}\{\boldsymbol{e}_i\} = \mathcal{K}_{i+1}$. For this two distinct possibilities are considered:

1. $\tilde{\boldsymbol{e}}_i \notin \mathcal{K}_i$

   Writing the residual of $\boldsymbol{x}_i$ as $\boldsymbol{r}_i = \boldsymbol{r}_0 + \boldsymbol{A}(\boldsymbol{x}_i - \boldsymbol{x}_0)$, line 7 of Algorithm 2.2 transforms to:

   $$\tilde{\boldsymbol{e}}_i = \boldsymbol{M}^{-1}\boldsymbol{r}_0 + \boldsymbol{M}^{-1}\boldsymbol{A}(\boldsymbol{x}_i - \boldsymbol{x}_0). \tag{2.10}$$

   Both terms are elements of $\mathcal{K}_{i+1}$, the latter due to Equation 2.7 which says that $\boldsymbol{x}_i - \boldsymbol{x}_0 \in \mathcal{K}_i$. Since the dimension of $\mathcal{K}_{i+1}$ is at most one higher than that of $\mathcal{K}_i$, adding $\tilde{\boldsymbol{e}}_i \notin \mathcal{K}_i$ must yield $\mathcal{K}_{i+1}$.

2. $\tilde{\boldsymbol{e}}_i \in \mathcal{K}_i$

   In this case the search vector does not enrich the search space, and it needs to be shown that in this case the Krylov subspace does not change either. For this, Equation 2.10 is put in the following form:

$$\tilde{\boldsymbol{e}}_i = \tilde{\boldsymbol{e}}_0 + \boldsymbol{M}^{-1}\boldsymbol{A}(\alpha_0 \cdot \tilde{\boldsymbol{e}}_0 + \cdots + \alpha_{i-1} \cdot \tilde{\boldsymbol{e}}_{i-1}), \tag{2.11}$$

   using the fact that $\boldsymbol{x}_i - \boldsymbol{x}_0 \in \mathcal{K}_i = \operatorname{span}\{\tilde{\boldsymbol{e}}_0, \ldots, \tilde{\boldsymbol{e}}_{i-1}\}$. By assumption, the left hand side $\tilde{\boldsymbol{e}}_i$ is in $\mathcal{K}_i$, as are all but the last terms on the right. It follows that $\alpha_{i-1}\boldsymbol{M}^{-1}\boldsymbol{A}\tilde{\boldsymbol{e}}_{i-1} \in \mathcal{K}_i$ as well.

   It will suffice to show that $\boldsymbol{M}^{-1}\boldsymbol{A}\tilde{\boldsymbol{e}}_{i-1} \in \mathcal{K}_i$ because then $\boldsymbol{M}^{-1}\boldsymbol{A}\mathcal{K}_i \subseteq \mathcal{K}_i$, hence $\mathcal{K}_{i+1} = \mathcal{K}_i$. If $\alpha_{i-1} \neq 0$, this follows directly from the obtained result. If $\alpha_{i-1} = 0$ then $\boldsymbol{x}_{i-1} = x_{i-2}$ due to optimality, hence $\tilde{\boldsymbol{e}}_{i-1} = \tilde{\boldsymbol{e}}_{i-2}$, which means that α_{i-1} and α_{i-2} can be interchanged. Continuing this process until the first non-zero coefficient if will always be possible to have $\alpha_{i-1} \neq 0$ in Equation 2.11, which proves the theorem. □

Theorem 2.2 opens a way to a wide range of different implementations, all being mathematically equivalent to Algorithm 2.2. The key to this is that the search space can now be formed without ever calculating the iterates $\boldsymbol{x}_i$ which involved the expensive projection operation. Instead of calculating approximate errors $\tilde{\boldsymbol{e}}_i$, Krylov subspace implementations in practice build their search space from **Krylov vectors**, that can be formed simply by repeatedly multiplying the previously calculated Krylov vector by $\boldsymbol{M}^{-1}\boldsymbol{A}$, starting from $\boldsymbol{M}^{-1}\boldsymbol{r}_0$. This process can be optimized in several ways, depending on specific properties of the system and the desired properties of the algorithm, such as memory efficiency. Some well known Krylov subspace methods will be discussed later in this section.

Because the Krylov subspace can never grow beyond the size of the problem, it is known in advance that it will reach the point of invariance — situation two in Theorem 2.2 — in at most n iterations for a system of n unknowns. Therefore, Krylov subspace methods can theoretically be considered finite methods. In practice this property is not really useful because the number of unknowns is often very large, which means that an enormous amount of work is possibly required to compute this exact solution. Moreover, more iterations will be required in practice due to rounding errors. The power of Krylov methods is not their ability to compute an exact solution, but to form a reasonable approximation in only a few iterations, depending mostly on the quality of the preconditioner. The optimality of the projection operator makes various convergence theorems possible, such as the following.

Theorem 2.3. *If the preconditioned matrix is diagonalizable, i.e. if there exist a matrix $\boldsymbol{X}$ and diagonal eigenvalue matrix $\boldsymbol{\Lambda}$ such that $\boldsymbol{M}^{-1}\boldsymbol{A} = \boldsymbol{X}\boldsymbol{\Lambda}\boldsymbol{X}^{-1}$, then the exact error at step i of Algorithm 2.2 is bounded by*

$$\|\boldsymbol{x} - \boldsymbol{x}_i\| \leq \epsilon^{(i)} \cdot \max_k \|\boldsymbol{e}_k\| \|\boldsymbol{X}\| \|\boldsymbol{X}^{-1}\| \|\boldsymbol{x} - \boldsymbol{x}_0\|, \tag{2.12}$$

where the $\boldsymbol{e}_k$ are basis vectors of the form $(0, \ldots, 0, 1, 0, \ldots, 0)$ and the vector norm is the same as used in projection $\mathbb{P}$. The matrix norm is induced as $\|\boldsymbol{X}\| = \max_{\|\boldsymbol{v}\|=1} \|\boldsymbol{X}\boldsymbol{v}\|$. Denoting $\mathcal{P}_i$ the space of polynomials of order less then i and σ

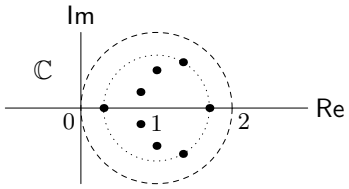

Figure 2.5. *Example eigenvalue distribution of a preconditioned matrix that yields a convergent Krylov subspace method, according to Theorem 2.3. The inner circle shows that all eigenvalues are contained in a disc of radius* 0.7 *and center* 1.0*, which provides an upper bound for the error of the successive approximations.*

the spectrum of the preconditioned matrix, $\epsilon^{(i)}$ *is defined as*

$$\epsilon^{(i)} = \min_{\substack{p \in \mathcal{P}_{i+1} \\ p(0)=1}} \max_{\lambda \in \sigma} |p(\lambda)|. \tag{2.13}$$

If, moreover, the eigenvalues σ *are contained in a disc with center* $C \in \mathbb{R}$ *and radius* $R < C$*, then the following upper bound holds:*

$$\epsilon^{(i)} \leq \left(\frac{R}{C} \right)^i. \tag{2.14}$$

Proof. Combining Equations 2.6, 2.7 and 2.9, the error at step i satisfies $\|\boldsymbol{x} - \boldsymbol{x}_i\| = \min_{\boldsymbol{y} \in \mathcal{K}_i} \|\boldsymbol{x} - \boldsymbol{x}_0 - \boldsymbol{y}\|$. By construction, the Krylov subspace $\mathcal{K}_i$ over which is minimized is bijective to the space of polynomials $\mathcal{P}_i$: for every Krylov element $\boldsymbol{y} \in \mathcal{K}_i$ there exists exactly one polynomial $p \in \mathcal{P}_i$ such that $\boldsymbol{y} = p(\boldsymbol{M}^{-1}\boldsymbol{A})\boldsymbol{M}^{-1}\boldsymbol{r}_0$. Using the diagonalization of $\boldsymbol{M}^{-1}\boldsymbol{A}$ this simplifies to $\boldsymbol{y} = \boldsymbol{X}p(\boldsymbol{\Lambda})\boldsymbol{\Lambda}\boldsymbol{X}^{-1}(\boldsymbol{x} - \boldsymbol{x}_0)$, in which $p(\boldsymbol{\Lambda})\boldsymbol{\Lambda}$ is a polynomial of increased order with a root at zero. With this, the error at step i of Algorithm 2.2 becomes

$$\|\boldsymbol{x} - \boldsymbol{x}_i\| = \min_{\substack{p \in \mathcal{P}_{i+1} \\ p(0)=1}} \|\boldsymbol{X}p(\boldsymbol{\Lambda})\boldsymbol{X}^{-1}(\boldsymbol{x} - \boldsymbol{x}_0)\|. \tag{2.15}$$

Because the matrix $p(\boldsymbol{\Lambda})$ is diagonal, its matrix norm is bounded from above by $\max_{\lambda \in \sigma} |p(\lambda)| \max_k \|\boldsymbol{e}_k\|$. Using this identity, repeated application of the induced matrix norm inequality $\|\boldsymbol{M}\boldsymbol{v}\| \leq \|\boldsymbol{M}\|\|\boldsymbol{v}\|$ results in the error bound Equation 2.12, with $\epsilon^{(i)}$ defined as in Equation 2.13. Equation 2.14 is a direct application of Zarantello's lemma, presented in Saad[16] as Lemma 6.4, Section 6.11.2. □

Compared with Theorem 2.1 that predicts convergence for basic iterative methods, Theorem 2.3 ensures the same and more. Equation 2.14 shows that for Krylov subspace methods, the speed of convergence depends on the amount of clustering of eigenvalues of the preconditioned matrix. In Figure 2.5 the spectrum shown before in Figure 2.3 is contained in a smaller disc, providing an exponentially decreasing upper bound for $\epsilon^{(i)}$. The size of this disc depends mostly on the quality of the preconditioner. A high quality preconditioner will be able to cluster the eigenvalues very tightly, leading to a very fast convergent process. Applying such a preconditioner, however, often incorporates a lot of work, increasing the computation time per iteration. An ideal preconditioner balances these two effects.

A last observation to be made from this theorem is that in Equation 2.15, when $\boldsymbol{x} - \boldsymbol{x}_0$ is built of a subset of eigenvectors (the columns of $\boldsymbol{X}$), the error $\boldsymbol{x} - \boldsymbol{x}_i$ is influenced only by the corresponding eigenvalues (the elements of $\boldsymbol{\Lambda}$). This means that the upper bound Equation 2.13 can be improved by restricting the maximum to this part of the spectrum. The same goes for the disc in Equation 2.14. Apparently, only those eigenvectors that yet have to be 'found' by the Krylov method for changing from $\boldsymbol{x}_0$ to $\boldsymbol{x}$ determine the speed of convergence. Tied to this subset of the spectrum is a possibly smaller **effective condition number** that governs convergence. During convergence, as more and more eigenvectors are found, the effective condition number will gradually decrease. This explains the super-linear speedup that is commonly observed in Krylov subspace methods, which is what makes them so very popular.

2.3.1 Preconditioners (continued)

To gain a better understanding of the role the preconditioner plays in Algorithm 2.2, it is useful to have a direct expression for the generated iterates. This is provided by Equation 2.7, using the result from Theorem 2.2. For a system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, with initial guess $\boldsymbol{x}_0$ and preconditioner $\boldsymbol{M}^{-1}$, the i-th iterate equals

$$\boldsymbol{x}_{i,\text{left}} = \boldsymbol{x}_0 + \mathbb{P}_{\mathcal{K}_i(\boldsymbol{M}^{-1}\boldsymbol{A}, \boldsymbol{M}^{-1}\boldsymbol{r}_0)}(\boldsymbol{x} - \boldsymbol{x}_0). \tag{2.16}$$

In the special case that both initial guess and preconditioner are absent, i.e. $\boldsymbol{x}_0 = \boldsymbol{0}$ and $\boldsymbol{M} = \boldsymbol{I}$, this simplifies to $\mathbb{P}_{\mathcal{K}_i(\boldsymbol{A},\boldsymbol{b})}\boldsymbol{x}$. From this it follows that completely equivalent to Equation 2.16 is $\boldsymbol{x}_{i,\text{left}} = \boldsymbol{x}_0 + \boldsymbol{y}_i$, with $\boldsymbol{y}_i$ the i-th iterate of Algorithm 2.2 applied to the system $\boldsymbol{M}^{-1}\boldsymbol{A}\boldsymbol{y} = \boldsymbol{M}^{-1}\boldsymbol{r}_0$, with no preconditioner and no initial guess. The equivalence of both representations shows that Krylov subspace methods need not know of preconditioners and initial guesses, as these can be supplied 'hidden' in the system. This modified system, left multiplied with the preconditioner, is called a **left preconditioned system**.

Equation 2.16 shows that the iterates of Algorithm 2.2 are formed in the affine subspace $\boldsymbol{x}_0 + \mathcal{K}_i(\boldsymbol{M}^{-1}\boldsymbol{A}, \boldsymbol{M}^{-1}\boldsymbol{r}_0)$. By construction of the Krylov subspace, this space is identical to $\boldsymbol{x}_0 + \boldsymbol{M}^{-1}\mathcal{K}_i(\boldsymbol{A}\boldsymbol{M}^{-1}, \boldsymbol{r}_0)$, which leads to the idea of creating iterates of the form

$$\boldsymbol{x}_{i,\text{right}} = \boldsymbol{x}_0 + \boldsymbol{M}^{-1}\mathbb{P}_{\mathcal{K}_i(\boldsymbol{A}\boldsymbol{M}^{-1}, \boldsymbol{r}_0)}\boldsymbol{M}(\boldsymbol{x} - \boldsymbol{x}_0). \tag{2.17}$$

This corresponds to solving $\boldsymbol{A}\boldsymbol{M}^{-1}\boldsymbol{y} = \boldsymbol{r}_0$ using the standard algorithm, and setting $\boldsymbol{x}_{i,\text{right}} = \boldsymbol{x}_0 + \boldsymbol{M}^{-1}\boldsymbol{y}_i$. The new systems is called a **right preconditioned system** for obvious reasons. Though these iterates are formed in the same subspace, they are different from Equation 2.16 because they are projected in a different way. This does not mean, however, that they are generally worse. Since the solution spaces are identical, the rate of convergence is approximately the same in both situations except when $\boldsymbol{M}$ is ill-conditioned, which could lead to substantial differences. In practice, an advantage of the right preconditioned system is that its residual can be directly related to that of the non-preconditioned system, whereas in the left preconditioned case this is obscured by the preconditioner.

Another reason for wanting to alter Algorithm 2.2 is to preserve symmetry of the system. A **symmetric and positive definite** (SPD) system has several nice properties that can be exploited by specialized Krylov implementations, and these should not be lost due to preconditioning. When the preconditioner is SPD as well, a Cholesky decomposition $\boldsymbol{M} = \boldsymbol{L}\boldsymbol{L}^T$ gives rise to a **two-sided preconditioned system** $\boldsymbol{L}^{-1}\boldsymbol{A}\boldsymbol{L}^{-T}\boldsymbol{y} = \boldsymbol{L}^{-1}\boldsymbol{r}_0$, post-processed with $\boldsymbol{x} = \boldsymbol{x}_0 + \boldsymbol{L}^{-T}\boldsymbol{y}$. A symmetric preconditioner will thus retain symmetry of the system, without actually changing the solution space. Again, the iterates do change:

$$\boldsymbol{x}_{i,\text{both}} = \boldsymbol{x}_0 + \boldsymbol{L}^{-T}\mathbb{P}_{\mathcal{K}_i(\boldsymbol{L}^{-1}\boldsymbol{A}\boldsymbol{L}^{-T},\boldsymbol{L}^{-1}\boldsymbol{r}_0)}\boldsymbol{L}^T(\boldsymbol{x} - \boldsymbol{x}_0). \tag{2.18}$$

Saad [16] shows in Section 9.2.1 that this two-sided preconditioned system is equivalent with a left preconditioned system solved with a slightly modified Krylov method. Therefore, in practice, the Cholesky decomposition needs not actually be performed. In the following situation, however, the decomposition is available for free so either variant can be used.

For Krylov subspace methods, the preconditioners introduced in Section 2.2 are often too weak. More sophisticated preconditioners are desirable. One example of a powerful preconditioner is incomplete LU or **ILU decomposition**. Like LU decomposition, presented in Section 2.1, the matrix is factored into a lower triangular matrix $\boldsymbol{L}$ and upper triangular matrix $\boldsymbol{U}$, with the difference that little or no fill-in is allowed. The idea is that the thus obtained, cheaply invertible matrix $\boldsymbol{L}\boldsymbol{U}$ still closely approximates the matrix $\boldsymbol{A}$, inheriting useful properties such as symmetry and sparsity. For a sparse matrix, skipping over its zero elements gravely reduces the work of factorization, as well as the memory required for storing the preconditioner.

2.3.2 Projection properties

Before moving on to some real Krylov subspace implementations, it is useful to gain a better understanding of the operator $\mathbb{P}_{\mathcal{V}}$ defined in Equation 2.6. In the preceding text, this operator has been called a **projection** without actually giving this a meaning. Formally, a projection operator $\mathbb{P}$ is a linear transformation that is idempotent, that is, $\mathbb{P}^2 = \mathbb{P}$. The idempotence property says that the result of a projection does not change from projecting it again, which is intuitively how a projection should behave. It is clear that this property holds for $\mathbb{P}_{\mathcal{V}}$.

Linearity is less trivial. To prove that under certain conditions $\mathbb{P}_{\mathcal{V}}$ is indeed a linear operator, an intermediate step will be required. The following theorem states that when a distance is minimized in a norm that is induced by an inner product, then the difference vector is perpendicular to the target subspace.

Theorem 2.4. *Let $\mathcal{U}$ and $\mathcal{V} \subseteq U$ be inner product spaces. For any two vectors $\boldsymbol{u} \in \mathcal{U}$ and $\boldsymbol{v} \in \mathcal{V}$, the following two statements are equivalent:*

1. $\|\boldsymbol{u} - \boldsymbol{v}\| \leq \|\boldsymbol{u} - \boldsymbol{w}\| \ \forall \ \boldsymbol{w} \in \mathcal{V}$
2. $(\boldsymbol{u} - \boldsymbol{v}, \boldsymbol{w}) = 0 \ \forall \ \boldsymbol{w} \in \mathcal{V}$

Proof. The implication $2 \Rightarrow 1$ is straightforward. From $\|\boldsymbol{u}-\boldsymbol{w}\|^2 = \|(\boldsymbol{u}-\boldsymbol{v}) + (\boldsymbol{v}-\boldsymbol{w})\|^2 = \|\boldsymbol{u}-\boldsymbol{v}\|^2 + 2(\boldsymbol{u}-\boldsymbol{v}, \boldsymbol{v}-\boldsymbol{w}) + \|\boldsymbol{v}-\boldsymbol{w}\|^2$, using the fact that $\boldsymbol{v}-\boldsymbol{w} \in \mathcal{V}$, it follows that $\|\boldsymbol{u}-\boldsymbol{w}\|^2 = \|\boldsymbol{u}-\boldsymbol{v}\|^2 + \|\boldsymbol{v}-\boldsymbol{w}\|^2 \geq \|\boldsymbol{u}-\boldsymbol{v}\|^2$ for all $\boldsymbol{w} \in \mathcal{V}$.

For the reverse implication, $1 \Rightarrow 2$, rewrite the above equality as $(\boldsymbol{u}-\boldsymbol{v}, \boldsymbol{v}-\boldsymbol{w}) = \frac{1}{2}(\|\boldsymbol{u}-\boldsymbol{w}\|^2 - \|\boldsymbol{u}-\boldsymbol{v}\|^2) - \frac{1}{2}\|\boldsymbol{v}-\boldsymbol{w}\|^2 \geq -\frac{1}{2}\|\boldsymbol{v}-\boldsymbol{w}\|^2$. Substituting for $\boldsymbol{w}$ the vectors $\boldsymbol{v} + \epsilon \cdot \boldsymbol{x}$ and $\boldsymbol{v} - \epsilon \cdot \boldsymbol{x}$, for some $\boldsymbol{x} \in \mathcal{V}$, yields the inequality $-\frac{\epsilon}{2}\|\boldsymbol{x}\|^2 \leq (\boldsymbol{u}-\boldsymbol{v}, \boldsymbol{x}) \leq \frac{\epsilon}{2}\|\boldsymbol{x}\|^2$ for all $\epsilon \neq 0$, hence $(\boldsymbol{u}-\boldsymbol{v}, \boldsymbol{x}) = 0$ for all $\boldsymbol{x} \in \mathcal{V}$. □

In Theorem 2.4, the vector $\boldsymbol{v} \in \mathcal{V}$ that satisfies the first statement is exactly $\mathbb{P}_{\mathcal{V}}\boldsymbol{u}$. When the norm is induced by an inner product, the equivalent second statement gives an important relation between a vector $\boldsymbol{u} \in \mathcal{U}$ and its projection. It follows that under this condition, the projection operator is indeed linear.

Theorem 2.5. *Let $\mathcal{U}$ and $\mathcal{V} \subseteq U$ be inner product spaces, and $\mathbb{P}_{\mathcal{V}}$ the projection defined in Equation 2.6. Then $\mathbb{P}_{\mathcal{V}}$ is a linear operator, and the following relation holds:*

$$(\mathbb{P}_{\mathcal{V}}\boldsymbol{u}, \boldsymbol{v}) = (\boldsymbol{u}, \boldsymbol{v}) \;\forall\; \boldsymbol{u} \in \mathcal{U}, \boldsymbol{v} \in \mathcal{V}. \tag{2.19}$$

Proof. The projection of a vector $\boldsymbol{u} \in \mathcal{U}$ is by definition the vector in $\mathcal{V}$ that is closest to $\boldsymbol{u}$ in the chosen norm, therefore $\|\mathbb{P}_{\mathcal{V}}\boldsymbol{u} - \boldsymbol{u}\| \leq \|\boldsymbol{v} - \boldsymbol{u}\|$ for all $\boldsymbol{v} \in \mathcal{V}$. By Theorem 2.4, this is equivalent to $(\mathbb{P}_{\mathcal{V}}\boldsymbol{u} - \boldsymbol{u}, \boldsymbol{v})$ for all $\boldsymbol{v} \in \mathcal{V}$, which proves Equation 2.19. Linearity now follows directly: $(\mathbb{P}_{\mathcal{V}}(a\boldsymbol{x} + b\boldsymbol{y}), \boldsymbol{v}) = (a\boldsymbol{x} + b\boldsymbol{y}, \boldsymbol{v}) = a(\boldsymbol{x}, \boldsymbol{v}) + b(\boldsymbol{y}, \boldsymbol{v}) = a(\mathbb{P}_{\mathcal{V}}\boldsymbol{x}, \boldsymbol{v}) + b(\mathbb{P}_{\mathcal{V}}\boldsymbol{y}, \boldsymbol{v}) = (a\mathbb{P}_{\mathcal{V}}\boldsymbol{x} + b\mathbb{P}_{\mathcal{V}}\boldsymbol{y}, \boldsymbol{v})$ for all $\boldsymbol{v} \in \mathcal{V}$, therefore, $\mathbb{P}_{\mathcal{V}}(a\boldsymbol{x} + b\boldsymbol{y}) = a\mathbb{P}_{\mathcal{V}}\boldsymbol{x} + b\mathbb{P}_{\mathcal{V}}\boldsymbol{y}$. □

When both $\mathcal{U}$ and $\mathcal{V}$ are finite dimensional vector spaces, the linearity of $\mathbb{P}_{\mathcal{V}}$ ensures that the operator can be presented as a matrix. Equation 2.19 provides a means of constructing this matrix for projection norms of the form $\|\boldsymbol{v}\| = \boldsymbol{v}^T\boldsymbol{X}\boldsymbol{v}$. The result is an explicit expression for the projection defined in Equation 2.6.

Theorem 2.6. *Let $\mathcal{U}$ and $\mathcal{V} \subseteq \mathcal{U}$ be real valued, finite dimensional inner product spaces. When the projection norm is induced by a SPD matrix $\boldsymbol{X}$, then there exist a matrix $\boldsymbol{V}$ such that the projection $\mathbb{P}_{\mathcal{V}}$ defined in Equation 2.6 equals*

$$\mathbb{P}_{\operatorname{col}\boldsymbol{V}} = \boldsymbol{V}(\boldsymbol{V}^T\boldsymbol{X}\boldsymbol{V})^{-1}\boldsymbol{V}^T\boldsymbol{X}. \tag{2.20}$$

Proof. The finite dimensionality of $\mathcal{V}$ means that it has a finite basis. When such a basis is formed by the columns of a matrix $\boldsymbol{V}$, each element $\boldsymbol{v} \in \mathcal{V}$ can be identified with a coordinate vector $\boldsymbol{v}' \in \mathbb{R}^{\dim\mathcal{V}}$ such that $\boldsymbol{v} = \boldsymbol{V}\boldsymbol{v}'$. Using the matrix-induced inner product $(\boldsymbol{v}_1, \boldsymbol{v}_2) = \boldsymbol{v}_1^T\boldsymbol{X}\boldsymbol{v}_2$, this transforms Equation 2.19 into the equivalent expression $\boldsymbol{V}^T\boldsymbol{X}\boldsymbol{V}(\mathbb{P}_{\mathcal{V}}\boldsymbol{u})' = \boldsymbol{V}^T\boldsymbol{X}\boldsymbol{u}$, for all $\boldsymbol{u} \in \mathcal{U}$. Solving this expression for $(\mathbb{P}_{\mathcal{V}}\boldsymbol{u})'$ yields Equation 2.20. □

2.3.3 Krylov methods for SPD matrices

All forework being done, it is now only a small step towards the actual Krylov subspace implementations that are used in practice. To recapitulate, Equation 2.7 showed that the iterates of a Krylov subspace method applied to a linear system $\boldsymbol{Ax} = \boldsymbol{b}$ can be formed by projecting the exact solution of the system onto the growing subspace of search vectors: $\boldsymbol{x}_i = \mathbb{P}_{\mathcal{V}}\boldsymbol{x}$; the initial guess and preconditioner can both be worked into the system beforehand and need henceforth not be available as separate components, as explained in Section 2.3.1. The search space $\mathcal{V}$ takes the form of a Krylov subspace, which by Theorem 2.2 is independent of the norm used in the projection. This norm, however, does have consequences for the implementation of the resulting method.

For a general matrix $\boldsymbol{X}$-norm, the projection operator is made explicit in Equation 2.20. Unfortunately, not any matrix $\boldsymbol{X}$ yields a valid Krylov method. In the beginning of Section 2.3 already it was noted that the norm should allow measurement of the unknown solution, that is, it must be possible to evaluate $\mathbb{P}_{\mathcal{V}}\boldsymbol{x}$. Equation 2.20 shows that one norm that has this property is the matrix $\boldsymbol{A}$-norm. Substituting the right hand side for $\boldsymbol{Ax}$, this leads to the following iterates:

$$\boldsymbol{x}_i = \boldsymbol{V}_i(\boldsymbol{V}_i^T\boldsymbol{A}\boldsymbol{V}_i)^{-1}\boldsymbol{V}_i^T\boldsymbol{b}, \tag{2.21}$$

in which matrix $\boldsymbol{V}_i$ spans the Krylov subspace $\mathcal{K}_i(\boldsymbol{A}, \boldsymbol{b})$ of dimension i. This norm forms the basis of the widely used **Conjugate Gradient** method, popular for its low resource usage due to a number of special optimizations. Unfortunately, in order to yield a valid norm matrix $\boldsymbol{A}$ must be symmetric and positive definite, which restricts this method to this special subset of problems.

The key to an efficient Krylov subspace implementation is to construct a Krylov basis that in some way simplifies the evaluation of Equation 2.21. CG uses a process known as **Arnoldi's method** to construct a set of orthonormal basis vectors $\boldsymbol{v}_i$. By definition, Equation 2.8, $\mathcal{K}_1$ is spanned only by right-hand side $\boldsymbol{b}$ which means that the first basis vector must be of the form $\boldsymbol{v}_0 = \beta^{-1}\boldsymbol{b}$. The coefficient β is chosen such that $\boldsymbol{v}_0$ has unit length. Next, since $\boldsymbol{v}_0$ and $\boldsymbol{A}\boldsymbol{v}_0$ span $\mathcal{K}_2$, so do $\boldsymbol{v}_0$ and any linear combination of $\boldsymbol{v}_0$ and $\boldsymbol{A}\boldsymbol{v}_0$. In particular it is possible to construct a basis vector $\boldsymbol{v}_1$ of unit length that is orthogonal to $\boldsymbol{v}_0$. Repeating this process i times yields a set of basis vectors satisfying

$$\boldsymbol{A}\boldsymbol{v}_{i-1} = \boldsymbol{v}_0 h_{0,i-1} + \cdots + \boldsymbol{v}_{i-1}h_{i-1,i-1} + \boldsymbol{v}_i h_{i,i-1}, \quad \boldsymbol{v}_i^T\boldsymbol{v}_j = \delta_{ij}. \tag{2.22}$$

Matrix $\boldsymbol{V}_i$ in Equation 2.21 is formed of the first i basis vectors. When $\boldsymbol{H}_i$ denotes the $i \times i$ matrix formed of coefficients h, the above relation can be written as

$$\boldsymbol{A}\boldsymbol{V}_i = \boldsymbol{V}_i\boldsymbol{H}_i + \boldsymbol{v}_i(0, \ldots, 0, h_{i,i-1}), \quad \boldsymbol{V}_i^T\boldsymbol{V}_i = \boldsymbol{I}. \tag{2.23}$$

Because $h_{i,j} = 0$ if $i-j > 1$, $\boldsymbol{H}_i$ is a **Hessenberg matrix**. From Equation 2.23 now follows the important equality $\boldsymbol{H}_i = \boldsymbol{V}_i^T\boldsymbol{A}\boldsymbol{V}_i$, which shows that if $\boldsymbol{A}$ is symmetric, so is $\boldsymbol{H}_i$ and hence $h_{i,j} = 0$ if $j - i > 1$. This has two important consequences. First of all, Equation 2.22 shows that Arnoldi's method requires only the two most

recent basis vectors in computing a new vector that is orthonormal to all others, although in practice the orthogonality will slowly degenerate due to rounding errors. Second, $\boldsymbol{V}_i^T \boldsymbol{A} \boldsymbol{V}_i$ is tridiagonal and can be inverted very efficiently. Writing the right hand side vector as $\beta \boldsymbol{v}_0 = \beta \boldsymbol{V}_i \boldsymbol{e}_1$, where $\boldsymbol{e}_1$ is the first unit vector of $\mathbb{R}^i$, Equation 2.21 simplifies to

$$\boldsymbol{x}_i = \beta \boldsymbol{V}_i \boldsymbol{H}_i^{-1} \boldsymbol{e}_1. \tag{2.24}$$

This expression corresponds to Equation 6.65 of Saad [16], Section 6.7, in which it is shown to lead to a highly efficient algorithm that updates $\boldsymbol{x}_i$ each iteration, requiring only a single vector in memory, regardless of how long the iterative process proceeds. This **short recurrence property**, together with the optimality property imposed by the projection behind Equation 2.21, is what makes the Conjugate Gradient method so popular. It would be very attractive to have the same two properties united in a Krylov method for general matrices as well. Unfortunately, it was shown by Faber and Manteuffel [3] that such is not possible.

2.3.4 Krylov methods for general matrices

For general matrices, the $\boldsymbol{A}$ matrix on itself does not yield a valid norm; a zero norm no longer implies zero length, which renders the projection useless. An alternative norm that is valid for any matrix $\boldsymbol{A}$ and still capable of measuring the unknown solution vector $\boldsymbol{x}$ is the matrix norm based on the product $\boldsymbol{A}^T \boldsymbol{A}$. Using this norm, the Krylov iterates take the following form:

$$\boldsymbol{x}_i = \boldsymbol{V}_i (\boldsymbol{V}_i^T \boldsymbol{A}^T \boldsymbol{A} \boldsymbol{V}_i)^{-1} \boldsymbol{V}_i^T \boldsymbol{A}^T \boldsymbol{b}. \tag{2.25}$$

By definition of the matrix norm, $\|\boldsymbol{x}_i - \boldsymbol{x}\|_{\boldsymbol{A}^T \boldsymbol{A}} = \|\boldsymbol{r}_i\|_2$, which means that when the projection minimizes the distance to the exact solution in $\boldsymbol{A}^T \boldsymbol{A}$ norm, it actually minimizes the residual vector's Euclidean length. Whether this is better or worse than the projection used in CG is in general hard to say. Both projections bring down the error by minimizing related, though different quantities; it will depend largely on the quality of the preconditioner which of the two is related most closely. Recall that the preconditioner has been worked into the system matrix $\boldsymbol{A}$. A very high quality preconditioner will make this matrix approach identity, which means that the residual will likewise approach the error. In practice this quality of preconditioning is not feasible, and the CG measured quantity can well be closer.

A popular Krylov method based on the $\boldsymbol{A}^T \boldsymbol{A}$ norm is **GMRES**, short for Generalized Minimum Residual method and named after the above interpretation of the projection operator. Like the CG method, GMRES relies on Arnoldi's method to construct an orthonormal Krylov basis. The identity $\boldsymbol{H}_i = \boldsymbol{V}_i^T \boldsymbol{A} \boldsymbol{V}_i$ still holds but it is of little use because the inverted matrix in Equation 2.25 is now $\boldsymbol{V}_i^T \boldsymbol{A}^T \boldsymbol{A} \boldsymbol{V}_i$. In order to efficiently invert this matrix, GMRES uses a series of **Givens rotations** $\boldsymbol{\Omega}_i$ to put $\boldsymbol{H}_i$ in triangular form, thus transforming Equation 2.23 to

$$\boldsymbol{A} \boldsymbol{V}_i = \boldsymbol{V}_{i+1} \boldsymbol{\Omega}_1^T \cdots \boldsymbol{\Omega}_i^T \begin{pmatrix} \boldsymbol{U}_i \\ \boldsymbol{0}^T \end{pmatrix}, \tag{2.26}$$

where $\boldsymbol{U}_i$ is a triangular matrix. Details of these transformations are for example in Section 6.5.3 of Saad [16]. Because Givens rotations are unitary, it follows that $\boldsymbol{V}_i^T \boldsymbol{A}^T \boldsymbol{A} \boldsymbol{V}_i = \boldsymbol{U}_i^T \boldsymbol{U}_i$. Substitution in Equation 2.25 yields the following expression for the iterates of GMRES:

$$\boldsymbol{x}_i = \beta \boldsymbol{V}_i \boldsymbol{U}_i^{-1} \boldsymbol{\Omega}_i \cdots \boldsymbol{\Omega}_1 \boldsymbol{e}_1. \tag{2.27}$$

The rotations can be applied progressively and will therefore add only a fixed amount of work per iteration. The diagonal system can be solved very cheaply as well, so this method is indeed very efficient in terms of work. Note, however, that since all previously calculated Krylov vectors are required to evaluate this expression, memory usage does increase with the number of iterations. This was of coarse known in advance because short recurrences and optimality are mutually exclusive in the general case. A possible way of reducing memory usage is to restart the iterative procedure after a fixed number of iterations, using the last iterate as the new starting point. As the algorithm will need to rebuild a Krylov subspace from scratch, however, super-linear convergence will be lost.

Somewhat better than restarting the iterative procedure would be to truncate the Krylov subspace, such that a fixed number of the most recently computed Krylov vectors will be kept in memory. Variants of GMRES that allow for truncation do exist, but this requires some serious modifications to the above approach. A Krylov method for which truncation comes naturally is the **Generalized Conjugate Residuals** method, which is based on the same $\boldsymbol{A}^T \boldsymbol{A}$-norm as GMRES. This method stays very close to the 'theoretic' Algorithm 2.2, forming a Krylov subspace from actual residuals. Therefore, no matter how the subspace is modified during the iterative process, GCR will always head in the direction of the exact solution.

Like CG and GMRES, GCR has its own way of evaluating Equation 2.25. The idea is to form a set of basis vectors for which $\boldsymbol{V}_i^T \boldsymbol{A}^T \boldsymbol{A} \boldsymbol{V}_i$ equals identity, so as to eliminate the matrix inverse. This means that the Krylov vectors must now be made $\boldsymbol{A}^T \boldsymbol{A}$-orthonormal, $(\boldsymbol{A}\boldsymbol{v}_i, \boldsymbol{A}\boldsymbol{v}_j) = \delta_{ij}$. Similar to Arnoldi's method, this is done by decomposing each newly calculated residual in a Krylov component and an $\boldsymbol{A}^T \boldsymbol{A}$-orthogonal component, the latter forming the new Krylov vector $\boldsymbol{v}_i$. To prevent a dramatic accumulation of matrix-vector multiplications in this process, a second set of vectors $\boldsymbol{w}_i = \boldsymbol{A}\boldsymbol{v}_i$ is stored. With that, Equation 2.21 simplifies to

$$\boldsymbol{x}_i = \boldsymbol{V}_i \boldsymbol{W}_i^T \boldsymbol{b} = \boldsymbol{v}_1 (\boldsymbol{w}_1, \boldsymbol{b})_2 + \cdots + \boldsymbol{v}_{i-1} (\boldsymbol{w}_{i-1}, \boldsymbol{b})_2. \tag{2.28}$$

Note that $(\boldsymbol{w}_i, \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_j) = (\boldsymbol{w}_i, \boldsymbol{b})$ for all $j < i$ due to orthogonality. GCR can therefore be implemented progressively as $\boldsymbol{x}_i = \boldsymbol{x}_{i-1} + \boldsymbol{v}_i (\boldsymbol{w}_i, \boldsymbol{r}_{i-1})$. In this form truncation of the Krylov subspace is allowed, limiting the memory that is required by the algorithm. The downside is that each dimension now requires two vectors in memory: $\boldsymbol{v}_i$ and $\boldsymbol{w}_i$. Moreover, when truncation is not used, the theoretically equivalent GMRES method proves to be more robust in certain situations. Therefore, in many applications, the latter is to be preferred.

2.4 Deflated Krylov methods

The Krylov subspace introduced in Section 2.3, Equation 2.8, has a fairly rigid structure. The only means of modification are by using a preconditioner and an initial guess, which corresponds to solving $(\boldsymbol{M}^{-1}\boldsymbol{A})\boldsymbol{y} = \boldsymbol{M}^{-1}\boldsymbol{r}_0$ instead of the unmodified system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$. In many applications these two are enough to get a fast convergent method, meaning that the low-dimensional Krylov subspaces already contain a close approximation to the exact solution. The quality of either, however, can be hampered by specific properties of the system, or by restrictions imposed by the computational setting, often a parallel cluster. In those situations, it would be useful to have an additional means of using additional, preliminary information about the system at hand.

This is the main idea behind **deflation**, a numerical technique for accelerating existing Krylov subspace methods by augmenting the subspaces with additional basis vectors, called **deflation vectors**. The nature of these vectors can depend entirely on the problem that is solved. Let a fixed set of deflation vectors span a linear subspace $\mathcal{Z}$. The following modification of Algorithm 2.2 directly augments the set of search vectors with these deflation vectors, thereby expanding the search space and thus improving the quality of the next iterate.

Algorithm 2.3

1. start: $\boldsymbol{x}_0$
2. $\boldsymbol{x}_0 \leftarrow \boldsymbol{x}_0 + \mathbb{P}_{\mathcal{Z}}(\boldsymbol{x} - \boldsymbol{x}_0)$
3. for $i = 0, 1, 2, \ldots$:
4. $\quad \boldsymbol{r}_i \leftarrow \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_i$
5. $\quad$ if $\|\boldsymbol{r}_i\| <$ threshold:
6. $\quad\quad$ break
7. $\quad$ end if
8. $\quad \tilde{\boldsymbol{e}}_i \leftarrow \boldsymbol{M}^{-1}\boldsymbol{r}_i$
9. $\quad \boldsymbol{x}_{i+1} \leftarrow \boldsymbol{x}_i + \mathbb{P}_{\mathcal{Z} \oplus \operatorname{span}\{\tilde{\boldsymbol{e}}_0, \ldots, \tilde{\boldsymbol{e}}_i\}}(\boldsymbol{x} - \boldsymbol{x}_i)$
10. end for

Compared to Algorithm 2.2, line 2 is new and line 9 has been changed. The added line enhances the initial guess with information from the deflation subspace, and the modification makes this information available to all iterates that follow. Note that the entire search space changes because of these modifications, as the search vectors $\tilde{\boldsymbol{e}}_i$ are based on these iterates $\boldsymbol{x}_i$. Therefore, the deflation subspace offers more than just a head start, it can continuously force the search vectors in more fruitful directions. Considerations for choosing a good deflation subspace will be treated later in this section.

Algorithm 2.3 is once again of theoretic value only. To get to practical implementations, it must be transformed back to the Krylov Algorithm 2.2 for which several efficient implementations exist. The following theorem will be invaluable to this transformation.

Theorem 2.7. *Let $\mathcal{Z}$ and $\mathcal{V}$ be orthogonal linear spaces with respect to a certain inner product, i.e. $(\boldsymbol{z}, \boldsymbol{v}) = 0$ for all $\boldsymbol{z} \in \mathcal{Z}$, $\boldsymbol{v} \in \mathcal{V}$. When $\mathbb{P}$ is the projection defined in Equation 2.6, based on the induced norm, the following identity holds:*

$$\mathbb{P}_{\mathcal{Z}\oplus\mathcal{V}} = \mathbb{P}_{\mathcal{Z}} + \mathbb{P}_{\mathcal{V}} \tag{2.29}$$

Proof. By linearity of the inner product, $((\mathbb{P}_{\mathcal{Z}}+\mathbb{P}_{\mathcal{V}})\boldsymbol{u}, \boldsymbol{z}+\boldsymbol{v})$ expands to $(\mathbb{P}_{\mathcal{Z}}\boldsymbol{u}, \boldsymbol{z})+(\mathbb{P}_{\mathcal{V}}\boldsymbol{u}, \boldsymbol{v})+(\mathbb{P}_{\mathcal{Z}}\boldsymbol{u}, \boldsymbol{v})+(\mathbb{P}_{\mathcal{V}}\boldsymbol{u}, \boldsymbol{z})$, in which the latter two terms are zero due to the orthogonality of $\mathcal{Z}$ and $\mathcal{V}$. The remaining two terms transform through Theorem 2.5 to $(\boldsymbol{u}, \boldsymbol{z})+(\boldsymbol{u}, \boldsymbol{v}) = (\boldsymbol{u}, \boldsymbol{z}+\boldsymbol{v}) = (\mathbb{P}_{\mathcal{Z}\oplus\mathcal{V}}\boldsymbol{u}, \boldsymbol{z}+\boldsymbol{v})$ for all $\boldsymbol{z}+\boldsymbol{v} \in \mathcal{Z}\oplus\mathcal{V}$ and $\boldsymbol{u} \in \mathcal{U}$, which proves Equation 2.29. □

Theorem 2.7 can not be used directly to transform line 9 because the search vectors will in general not be orthogonal to the deflation subspace. It is easy to see, however, that for general subspaces $\mathcal{Z}$ and $\mathcal{V}$, the direct sums $\mathcal{Z}\oplus\mathcal{V}$ and $\mathcal{Z}\oplus(\boldsymbol{I}-\mathbb{P}_{\mathcal{Z}})\mathcal{V}$ are equal: $\boldsymbol{z}+\boldsymbol{v} = (\boldsymbol{z}+\mathbb{P}_{\mathcal{Z}}\boldsymbol{v})+(\boldsymbol{I}-\mathbb{P}_{\mathcal{Z}})\boldsymbol{v}$ and $\boldsymbol{z}+(\boldsymbol{I}-\mathbb{P}_{\mathcal{Z}})\boldsymbol{v} = (\boldsymbol{z}-\mathbb{P}_{\mathcal{Z}}\boldsymbol{v})+\boldsymbol{v}$ prove mutual inclusion. Theorem 2.5 shows that $(\boldsymbol{z}, \boldsymbol{v}-\mathbb{P}_{\mathcal{Z}}\boldsymbol{v}) = 0$, hence the subspaces in the latter direct sum are indeed orthogonal. Theorem 2.7 now shows that the projection on line 9 can be split into two independent projections as follows:

$$\boldsymbol{x}_{i+1} \leftarrow \boldsymbol{x}_i + \mathbb{P}_{\mathcal{Z}}(\boldsymbol{x}-\boldsymbol{x}_i) + \mathbb{P}_{(\boldsymbol{I}-\mathbb{P}_{\mathcal{Z}})\,\mathrm{span}\{\tilde{\boldsymbol{e}}_0,\ldots,\tilde{\boldsymbol{e}}_i\}}(\boldsymbol{x}-\boldsymbol{x}_i). \tag{2.30}$$

Not quite equal to Algorithm 2.2 yet, but it is moving in the right direction. The next observation is that for both $i = 0$ (line 2) and $i \geq 1$ (Equation 2.30) the projection of the iterate onto the deflation subspace equals that of the exact solution, $\mathbb{P}_{\mathcal{Z}}\boldsymbol{x}_i = \mathbb{P}_{\mathcal{Z}}\boldsymbol{x}$. This follows from the linearity of $\mathbb{P}_{\mathcal{Z}}$ and the projection property $\mathbb{P}_{\mathcal{Z}}^2 = \mathbb{P}_{\mathcal{Z}}$; in Equation 2.30, the second projection vanishes due to orthogonality. Consequently, the first projection in Equation 2.30 is always zero and drops out. Defining a new set of search vectors $\tilde{\boldsymbol{e}}_i'$, lines 8 and 9 can now be modified such that Algorithm 2.3 assumes the form of a standard Krylov method, i.e. that of Algorithm 2.2:

$$\begin{aligned} \tilde{\boldsymbol{e}}_i' &\leftarrow (\boldsymbol{I}-\mathbb{P}_{\mathcal{Z}})\boldsymbol{M}^{-1}\boldsymbol{r}_i \\ \boldsymbol{x}_{i+1} &\leftarrow \boldsymbol{x}_i + \mathbb{P}_{\mathrm{span}\{\tilde{\boldsymbol{e}}_0',\ldots,\tilde{\boldsymbol{e}}_i'\}}. \end{aligned} \tag{2.31}$$

Thus put into Krylov form, all results obtained in Section 2.3 become valid again, including efficient implementations such as CG and GMRES. It would seem that, in the end, deflation is not essentially different from any other preconditioner. There is, however, one essential difference: the new preconditioner $(\boldsymbol{I}-\mathbb{P}_{\mathcal{Z}})\boldsymbol{M}^{-1}$ is singular. Its null space is spanned by the columns of $\boldsymbol{MZ}$. This means that the Krylov subspace can reach only part of the total solution space — to be precise, the orthogonal complement of the deflation subspace. The deflation subspace component is determined independently. Deflation can thus be seen as a means to narrow the search space, constantly forcing the Krylov method away from possibly problematic regions. The iterates take the following form:

$$\boldsymbol{x}_i = \boldsymbol{x}_0 + \mathbb{P}_{\mathcal{Z}}(\boldsymbol{x}-\boldsymbol{x}_0) + \boldsymbol{y}_i, \quad \boldsymbol{y}_i \perp \mathcal{Z} \tag{2.32}$$

Of course this 'explanation' of improved convergence due to deflation is merely heuristic. More solid grounds will be built on insights provided by eigenvalue deflation, a subject to be treated later in this section. Before discussing these specific

examples of deflation, however, there is one final step to be taken in order to arive at the 'standard' formulation, that is commonly found in articles on this subject such as by Vuik et al. [19] and Frank and Vuik [4]. This step is similar to the transition from a left to a right preconditioned system in Section 2.3.1, which transformed Equation 2.16 to 2.17 by shifting the preconditioner through the Krylov subspace. In exactly the same way, the subspace in which $\boldsymbol{y}_i$ in Equation 2.32 is formed can be expressed in three different ways:

$$\begin{aligned}\boldsymbol{y}_i &\in \mathcal{K}_i\left[(\boldsymbol{I}-\mathbb{P}_{\mathcal{Z}})\mathcal{M}^{-1}\boldsymbol{A}, (\boldsymbol{I}-\mathbb{P}_{\mathcal{Z}})\mathcal{M}^{-1}\boldsymbol{A}(\boldsymbol{I}-\mathbb{P}_{\mathcal{Z}})(\boldsymbol{x}-\boldsymbol{x}_0)\right] &(2.33)\\ &= (\boldsymbol{I}-\mathbb{P}_{\mathcal{Z}})\mathcal{K}_i\left[\mathcal{M}^{-1}\boldsymbol{A}(\boldsymbol{I}-\mathbb{P}_{\mathcal{Z}}), \mathcal{M}^{-1}\boldsymbol{A}(\boldsymbol{I}-\mathbb{P}_{\mathcal{Z}})(\boldsymbol{x}-\boldsymbol{x}_0)\right] &(2.34)\\ &= (\boldsymbol{I}-\mathbb{P}_{\mathcal{Z}})\mathcal{M}^{-1}\mathcal{K}_i\left[\boldsymbol{A}(\boldsymbol{I}-\mathbb{P}_{\mathcal{Z}})\mathcal{M}^{-1}, \boldsymbol{A}(\boldsymbol{I}-\mathbb{P}_{\mathcal{Z}})(\boldsymbol{x}-\boldsymbol{x}_0)\right]. &(2.35)\end{aligned}$$

Although three times the exact same search space, the Krylov subspace inside differs which means that the connected Krylov methods yield different iterates due to a different projection. The first of those methods is completely equivalent to Algorithm 2.3. Its iterates are formed from

$$\begin{aligned}\boldsymbol{x}_i &= \boldsymbol{x}_0 + \mathbb{P}_{\mathcal{Z}}(\boldsymbol{x}-\boldsymbol{x}_0) + \boldsymbol{y}_i,\\ &\left[(\boldsymbol{I}-\mathbb{P}_{\mathcal{Z}})\boldsymbol{M}^{-1}\right]\boldsymbol{A}\boldsymbol{y} = \left[(\boldsymbol{I}-\mathbb{P}_{\mathcal{Z}})\boldsymbol{M}^{-1}\right]\boldsymbol{A}(\boldsymbol{I}-\mathbb{P}_{\mathcal{Z}})(\boldsymbol{x}-\boldsymbol{x}_0), \quad (2.36)\end{aligned}$$

in which the $\boldsymbol{y}_i$ result from a Krylov subspace method applied to the left-preconditioned system. Due to the use of the projection operator, this system is singular. This is not necessarily a problem. A well known result from Kaasschieter [7] is that Krylov subspace methods have no problem solving singular systems, as long as the system is **consistent**, meaning that the right hand side lies in the column space of the system matrix. Since the same preconditioner is used on both sides of the equation, this condition is met and the Krylov method will generate a valid solution for this system.

There does appear to be a problem, though, because this solution is not fully determined. It can contain an arbitrary component from its the null space, formed of the columns of $\boldsymbol{A}^{-1}\boldsymbol{M}\boldsymbol{Z}$. As the original, non-linear system is not singular and therefore does have a uniquely defined solution, arbitrary iterates are useless. The reason that this is not a problem is that Krylov subspace iterates are in fact not arbitrary, but fully determined projections onto the growing Krylov subspace. By construction, these subspace elements are all of the form $(\boldsymbol{I}-\mathbb{P}_{\mathcal{Z}})\boldsymbol{v}$, which means the Krylov method will always select the single solution that is orthogonal to the deflation subspace.

It follows that the solution method defined in Equation 2.36 produces fully determined, valid iterates. However, its high dependence on the structure of the Krylov subspace will prove to be very restrictive. A method based on the reformulated search space, Equation 2.34, will allow for much more flexibility. These iterates are formed as

$$\begin{aligned}\boldsymbol{x}_{i,\text{left}} &= \mathbb{P}_{\mathcal{Z}}\boldsymbol{x} + (\boldsymbol{I}-\mathbb{P}_{\mathcal{Z}})(\boldsymbol{x}_0 + \boldsymbol{y}_{i,\text{left}}),\\ &\left[\boldsymbol{M}^{-1}(\boldsymbol{I}-\boldsymbol{A}\mathbb{P}_{\mathcal{Z}}\boldsymbol{A}^{-1})\right]\boldsymbol{A}\boldsymbol{y} = \left[\boldsymbol{M}^{-1}(\boldsymbol{I}-\boldsymbol{A}\mathbb{P}_{\mathcal{Z}}\boldsymbol{A}^{-1})\right]\boldsymbol{r}_0, \quad (2.37)\end{aligned}$$

where the $\boldsymbol{y}_{i,\text{left}}$ result from a Krylov method applied to the new preconditioned system. Note that the $\boldsymbol{A}^{-1}$ in the new preconditioner will cancel against the trailing

$\boldsymbol{A}$ in $\mathbb{P}_{\mathcal{Z}}$, which is there as explained in Sections 2.3.3 and 2.3.4. The preconditioner is still singular, with a null space equal to the deflation subspace $\mathcal{Z}$. The system therefore has solutions of the form $\boldsymbol{y} = \boldsymbol{x} - \boldsymbol{x}_0 + \boldsymbol{z}$, with $\boldsymbol{z}$ an arbitrary deflation subspace component. The main difference with the previous method is that the iterates $\boldsymbol{x}_{i,\text{left}}$ are unique regardless of the Krylov subspace, because the arbitrary component $\boldsymbol{z}$ is annihilated separately by the projection $\boldsymbol{I} - \mathbb{P}_{\mathcal{Z}}$.

By loosing dependence of the Krylov subspace, the solution method has become very tolerant to modifications. For instance, it is now possible to use different projection norms in the deflation and Krylov projections, something that would immediately destroy convergence of Equation 2.36. In fact, the new method can only exist by virtue of this new possibility. Note that the reformulation of the search space involves shifting $\boldsymbol{I} - \mathbb{P}_{\mathcal{Z}}$ out of the Krylov subspace. Section 2.3.1 noted that such can lead to substantially different iterates in case the shifted matrix is ill-conditioned. A singular matrix can be considered an extreme case, and indeed performing the Krylov projection in either the $\boldsymbol{A}$ or $\boldsymbol{A}^T\boldsymbol{A}$ norm does not result in a convergent method. This is fixed by projecting in a seminorm based on the deflated matrix $\boldsymbol{A} - \boldsymbol{A}\mathbb{P}_{\mathcal{Z}}$, thus measuring orthogonal to the deflation subspace only. In practice this discrepancy is of no concern, as actual Krylov methods project in the preconditioned matrix norm anyway.

Equation 2.37 shows deflation the way it is usually found in literature: a left-preconditioned system with a solution that needs post-processing to make it unique. Less commonly found is the right-preconditioned variant, based on the second reformulation of the search space, Equation 2.35. Its iterates are

$$\begin{aligned}\boldsymbol{x}_{i,\text{right}} = \boldsymbol{x}_0 + \mathbb{P}_{\mathcal{Z}}(\boldsymbol{x} - \boldsymbol{x}_0) + \left[(\boldsymbol{I} - \mathbb{P}_{\mathcal{Z}})\boldsymbol{M}^{-1}\right]\boldsymbol{y}_{i,\text{right}},\\ \boldsymbol{A}\left[(\boldsymbol{I} - \mathbb{P}_{\mathcal{Z}})\boldsymbol{M}^{-1}\right]\boldsymbol{y} = \boldsymbol{b} - \boldsymbol{A}(\boldsymbol{x}_0 + \mathbb{P}_{\mathcal{Z}}(\boldsymbol{x} - \boldsymbol{x}_0)),\end{aligned} \quad (2.38)$$

where $\boldsymbol{y}_{i,\text{right}}$ are Krylov iterates from the right preconditioned, singular system. The null space is spanned by the columns of $\boldsymbol{MZ}$, and it can be seen that an arbitrary component from this space is again annihilated in the formation of $\boldsymbol{x}_{i,\text{right}}$. What is different from left preconditioned deflation is that annihilation is now caused by the preconditioner instead of a separate post-processor. Compare Equation 2.38 to the standard right preconditioned system introduced in Section 2.3.1. It seems that, using $\boldsymbol{x}_0 + \mathbb{P}_{\mathcal{Z}}(\boldsymbol{x} - \boldsymbol{x}_0)$ as initial guess and $(\boldsymbol{I} - \mathbb{P}_{\mathcal{Z}})\boldsymbol{M}^{-1}$ as preconditioner, deflation simplifies to a normal, right preconditioned Krylov method that does not require any extras like a post-processing step. This step is not lost, however; it has simply changed into a preprocessing step to form the new initial condition.

In practice, the preconditioned Krylov variants of deflation coexists with algorithms that stay more close to Algorithm 2.3. Morgan [9] for example has formulated a modified GMRES algorithm that is capable of dealing with arbitrary deflation subspaces. Such algorithms are called **augmented Krylov methods**, which sounds as though they are the exact opposite of 'deflated' Krylov methods while in fact they are very similar. From the above it is clear that the two classes relate in similar ways as left and right preconditioned iterative methods do: they merely measure in different spaces. Therefore both classes are expected to show the same convergence behaviour for a given deflation subspace. The only question left is which subspaces will actually improve convergence, and why.

2.4.1 Krylov deflation

An interesting deflation subspace that springs to mind is a Krylov subspace of fixed dimension n: $\mathcal{Z} = \mathcal{K}_n(\boldsymbol{MA}, \boldsymbol{Mr}_0)$. This subspace is spanned by the first n search vectors of the non-deflated Algorithm 2.2. Equation 2.7 now shows that line 2 assignes $\boldsymbol{x}_0$ the n-th non-deflated Krylov iterate, from which point onward both algorithms construct exactly the same search vectors. As always, actual iterates can differ due to modifications in the implementation of both algorithms, such as those underlying Equations 2.37 and 2.38, but the search spaces are identical. For this deflation subspace, apparently deflation is no more than a head start of n Krylov iterations, which is of no practical use as it will take an additional n iterations to form the deflation subspace. Still, this does lead to two ideas about how deflation can be put to use.

The first has to do with restarting. Recall from Section 2.3 that most Krylov subspace methods require all Krylov vectors to be kept in memory, inevitably leading to memory problems when convergence takes many iterations. A common solution is to restart the Krylov method after a fixed number of iterations, using the last computed approximation as initial guess. The obvious problem to this approach is that all information about the Krylov subspace is lost, with dramatic consequences for convergence. The above example shows that feeding this information through deflation fixes this problem and leaves convergence intact after a restart, only it doesn't resolve the memory problem as the Krylov vectors are basically just renamed to deflation vectors. However, when the information present in the Krylov subspace can somehow be compressed into a smaller deflation subspace, deflation could be really useful in the context of restarted Krylov methods.

Another idea about deflation is brought by further inspection of the Krylov deflated process. What happens is that the unmodified Krylov subspace rapidly catches up with the deflation subspace, embodying the deflation vectors in only a few iterations. Once this point is reached, the deflated method continuous as though it were a normal Krylov method, leaving only a fixed amount of iterations gain at best. In this respect, the first few Krylov vectors form the worst deflation subspace thinkable. In order to be more than just a flying start, deflation should be based on vectors that come into the Krylov subspace only after many iterations, long after a suitable approximate solution is found. That way deflation will continuously accelerate convergence by injecting external information into the algorithm, as well as provide a leap start.

However different these two ideas may seem, it will be shown that in both situations eigenvectors are the key to a good deflation subspace. In practice, the first idea underlies most augmented Krylov implementations, while the second is the main idea behind most deflated ones. This can be explained partially from the fact that for SPD matrices, restarts are not normally required due to the short recurrence property. The deflation preconditioner retains this property and can thus still be used to speed up convergence through enrichment. For general Krylov methods this restriction does not apply, and for those directly adding vectors to the Krylov subspace can be a little more straightforward than preconditioning.

2.4.2 Eigenvalue deflation

The following special case will give an explanation of improved convergence in deflated Krylov methods. Consider a symmetric and positive definite system $\boldsymbol{Ax} = \boldsymbol{b}$, solved using a deflated Krylov method based on Equation 2.37 with no preconditioner $\boldsymbol{M}$. An SPD matrix induces a norm that can be used in the projection $\mathbb{P}_{\mathcal{Z}}$ onto the deflation subspace. When this subspace is spanned by the columns of a matrix $\boldsymbol{Z}$, Equation 2.20 shows that the deflated system takes the following form:

$$\boldsymbol{PA} = \boldsymbol{A} - \boldsymbol{AZ}(\boldsymbol{Z}^T\boldsymbol{AZ})^{-1}\boldsymbol{Z}^T\boldsymbol{A}. \tag{2.39}$$

Note that the deflated system is still symmetric, which means that highly optimized Krylov methods such as the Conjugate Gradient method are still applicable. Moreover, for a carefully chosen deflation subspace, deflation will retain not only symmetry but even a large part of the spectrum of $\boldsymbol{A}$. To see this, let the eigenvectors of $\boldsymbol{A}$ be denoted $\boldsymbol{v}_0, \boldsymbol{v}_1, \ldots$, with corresponding eigenvalues $\lambda_0, \lambda_1, \ldots$ such that $\boldsymbol{A}\boldsymbol{v}_i = \lambda_i\boldsymbol{v}_i$. A well known SPD matrix property is that its eigenvectors can be chosen such that they form an orthogonal set. Due to this property, when $\boldsymbol{Z}$ is composed of a subset of eigenvectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n$, the following holds:

$$(\boldsymbol{PA})\boldsymbol{v}_i = \begin{cases} \boldsymbol{0} & i \leq n \\ \lambda_i\boldsymbol{v}_i & i > n \end{cases} \tag{2.40}$$

This shows that deflation can be used to selectively 'project' certain eigenvalues of $\boldsymbol{A}$ to zero, while leaving the others in place. Recall from Section 2.3 that Krylov convergence is governed by the effective condition number, which is tied to the set of eigenvectors that are to be found by the Krylov method. Since the deflation vectors are known from the start, the corresponding zero eigenvalues do not influence convergence. With the remaining part of the original spectrum left in place, the effective condition number of the thus deflated system is at most as high as the real condition number. When extreme eigenvalues are projected out, it will be smaller.

This example illustrates the explanation of improved convergence in deflated Krylov methods. Instead of shifting the eigenvalues of the system together, like a normal preconditioner does, the effective condition number is lowered by selectively projecting the smallest eigenvalues to zero. This can be thought of as 'deflating' the spectrum of the matrix, which at last explains the name of this method. Ideally, the remaining part of the spectrum is left unharmed, but in practice this is not really feasible as determining eigenvectors is a very expensive procedure. Practical deflated methods therefore rely on various techniques to cheaply generate a set of approximate eigenvectors. Although this will slightly change the spectrum of the deflated matrix, the effective condition number should still decrease.

Returning to the first of the two deflation approaches suggested in the previous section, it is now clear how the information in a Krylov subspace should be 'compressed' into a set of deflation vectors. The convergence speed that is wished to be preserved after a restart is due to a decreased effective condition number, meaning a few extreme eigenvectors are closely approximated by the Krylov subspace. These approximate eigenvectors span a subspace of a much lower dimension than the complete Krylov space, while still containing most of the information that keeps

convergence up to speed. What is more, these approximate eigenvectors, known as **Ritz vectors**, can be cheaply obtained from the Arnoldi method underlying for example GMRES. Results of restarted GMRES based on this type of deflation are presented a.o. by Morgan [9], who tested it with his own modified algorithm.

The other idea was to speed up convergence by explicitly adding a set of vectors that are not so easily found by the Krylov method. These vectors are now understood to be eigenvectors, corresponding to the lower part of the spectrum. This time there is no way of obtaining these vectors other than through expert knowledge of the system. For example Vuik et al. [19] manually composed a set of approximate eigenvectors based purely on physical grounds, which they called **physical deflation** vectors. Another option is to base deflation on insights on the matrix itself, in which case it is sometimes called **algebraic deflation**. This type of deflation, described a.o. by Frank and Vuik [4], is most often found in domain decomposition contexts where it can be seen as a means of global communication between subdomains.

2.4.3 Subdomain deflation

This chapter started with the observation that current day research often involves increasingly large systems of equations. As technology progresses this growth will continue. The techniques discussed so far aimed at reducing both work and memory usage, so as to stretch the limits of what can be solved within reasonable time. No matter how efficient these methods may be, however, limits will always continue to exist. The final important step is therefore to implement the techniques discussed thus far in such a way that a group of separate computational nodes can work in joint collaboration. Such a **parallel computing** environment can be thought of as a single system with a large amount of memory and computing power, with one major bottleneck caused by inter node communication. This communication should be the main concern when developing for a parallel cluster.

Methods for solving linear systems in parallel, at least those representing physical problems, often build on **domain decomposition** techniques. This means that the physical domain is subdivided into a number of subdomains, that an equal number of computational nodes can claim as their part of the problem. The approximate solutions are thus spread over the parallel cluster, and during the iterative process relevant parts are communicated with other nodes in the cluster. In particular, operations requiring this communication are inner products and matrix-vector products. Due to the underlying physics, communication data is usually limited to the interface regions between adjacent subdomains. This is an important difference with direct solution methods, which require heavy communication due to their inherent global nature. The impact of this on performance makes direct methods less suitable for parallelization.

Recall that the popular ILU preconditioner is a modified version of full LU decompomposition. As such, ILU exhibits the same kind of communication problems as direct methods when used in parallel computing environments. For this reason, in most practical domain decomposition methods preconditioners are applied on subdomain level only. These are then combined into one global preconditioner through

block variants of the Gauss Jacobi or Gauss Seidel preconditioners introduced in Section 2.2.1, although in this setting they are usually named **additive Schwarz** and **multiplicative Schwarz**, respectively. Using ILU for the (approximate) block inversions, the resulting preconditioner is sometimes referred to as **block ILU**. Other popular preconditioners for domain decomposition methods are based on the Schur complement method, which takes a quite different approach. Both types of methods are covered extensively in Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations, by Smith et al. [17].

Being approximate solvers, preconditioners improve convergence by smoothing out the error in subsequent approximations. As such they can be viewed as a means of global communication within the iterative method, added to the local, physical connections manifested in the non-zero entries of the original matrix. When no preconditioner is used, propagation of information from one side of the domain to the other through these local physical connections can require many iterations. Since block variants operate subdomain local, inter subdomain communication in block preconditioned methods falls back on these slow, local connections, manifested in slowly decaying, though rapidly smoothed subdomain errors. For this reason parallel methods always experience a drop in convergence speed, but the total computation time can still go down due to the division of work. For increasingly large clusters, however, the situation worsens to the point where further parallelization can not be justified.

Deflation can help improve this situation. When a set of deflation vectors is constructed that are zero everywhere on the domain, except for a single subdomain on which they are constant, the resulting deflation subspace is the set of solutions that are constant along subdomains. Since the deflation subspace is the null space of the deflated system, subdomain wide shifts in the solution go unnoticed. The absolute size of the subdomain error is ignored by the deflated system, which means that inter subdomain communication is taken over by the deflation operator. Thus complementing the intra subdomain communication brought by the block-preconditioner with a means of inter subdomain communication, **subdomain deflation** is expected to reduce the ill effects of a domain decomposition.

The preceding subsection gave an explanation of improved convergence due to deflation in terms of eigenvectors. Indeed, the subdomain constant deflation vectors can be thought to span a subspace similar to that of the smallest few eigenvectors of a block preconditioned system. Since the remaining eigenvalues are shifted to one by the block-preconditioners, projecting these smallest few to zero reduces the effective condition number. This shows the great difference with another widely used acceleration technique, coarse grid correction, which uses a similar subspace to shift the smallest eigenvalues to one. Nabben and Vuik [12] proved with Theorem 2.6 that for identical subspaces, the effective condition number of a deflated SPD system is always at most as high as that of the coarse grid corrected system.

The eigenvector explanation suggests that if a richer deflation subspace manages to approximate the unwanted eigenvectors more closely, this will lead to an even better convergence speed up. One possible way of enrichment is adding higher order deflation vectors, such as proposed by Verkaik [18] in his master thesis. Note, however, that a larger subspace will evenly enlarge the small system that is solved

for each application of the deflation operator, countering the decrease of iterations by increased work per iteration. As with most things in numerics, constructing a good deflation subspace is a matter of optimization, which may require new trial and error investigations every time that one is confronted with a new type of system.

Chapter 3

Implementing deflation

The VMS method introduced in Chapter 1 is implemented by a research group at the faculty of Aerospace Engineering of Delft, University of Technology. They have taken a flexible approach, making not only the problem itself but also many aspects of the solution procedure configurable at runtime. This includes solver type, preconditioners, number of subdomains, accuracy, etcetera, which makes the program very suitable for numerical experiments. The flexibility is due mainly to the Jem/Jive programming toolkits developed by Habanera. These toolkits allow for rapid development of efficient numerical software in the C++ programming language. Strictly speaking the VMS code is written on top of MPF, a toolkit developed at Aerospace Engineering that specifically targets fluid-structure interaction problems. Since MPF in turn builds on Jem/Jive, however, this distinction is irrelevant to this report.

Experiments with the existing implementation have shown that the VMS method performs quite well when executed as a single process. However, this performance deteriorates quickly when the problem is distributed over an increasingly large number of computational nodes and solved in parallel. In other words, it scales very poorly. These experiments were performed using an additive Schwarz preconditioner in combination with local approximate solvers, typically an ILU preconditioner. Section 2.4.3 showed that this situation can possibly improve from using a deflation technique, which takes the form of a new solver. Due to the modular design enforced by Jem/Jive, the current program should be able to use the new solver without change, once it becomes available.

This chapter describes the implementation of a deflation solver in Jem/Jive. General knowledge of the C++ programming language will be assumed, including object oriented concepts such as classes and inheritance. Knowledge of function templates — although heavily used in Jem/Jive — is not required for reading this chapter. The first two sections are a general introduction to Jem/Jive, with focus on parts that are relevant to solver development. Section 3.3 describes the implementation of the deflation solver in detail. The complete source code of the deflation solver has been included as Appendix A.

3.1 Jem

The first and foremost choice to be made when a new software project is started is which programming language is going to be used. In particular, one should choose between high and low level languages. High level languages, such as Python and Scheme, abstract away error prone tasks like memory management, which increases the maintainability and also portability of code at the cost of efficiency. Well written low level code will in general execute faster because these tasks can be optimized for specific applications, which is the main reason that low level languages are still used in areas where speed is of prime importance. Other considerations that can force the use of a certain language are company policy, existing code base and programming expertise. It will therefore be very useful to have some high-level functionality without having to switch language.

Jem is a toolkit for the relatively low-level C++ programming language that aims to be exactly this. It introduces typical high-level features such as garbage collection, complex data types and, most of all, portability of code while keeping the possibility of low-level optimization. This means that the non time-crucial parts — in general the large majority — of code can be written fast and cleanly because focus can lie on essentials instead of accessories, like memory management. Typical candidates for optimization are numerical algorithms, for which all potential power of C++ is still available. This section is a reference of the main Jem objects and their intended use. For ease of reference these objects are indexed under 'Jem'.

3.1.1 System interface

Ideally, C++ source code should compile to a valid program, unchanged, on all computer architectures that have a C++ compiler available. The compiler translates the source into a sequence of low level, architecture specific instructions that form the same intended program on any of these. The reason that most code still needs changing in order to run on different platforms is mostly not architecture but operating system related. Problems occur where low-level interaction is required. For instance things like user input and output require different system calls in Windows and Linux, with obvious implications on code. This difference is usually hidden by libraries such as STL, which have different implementations on supported platforms. From the programmer's perspective these provide a consistent way of performing certain tasks and using these he is able to write code that runs on all platforms that have this library.

Jem is a similar abstraction layer, that provides consistent access to various aspects of the operating system through several classes, in particular the **System** class. This includes environment variables, system properties, stack traces, as well as user interaction through input-output operations and logging facilities. By accessing the system strictly through Jem, the source code will be completely operating system independent because all dependence has been moved into Jem, which has different implementations on the supported platforms. Currently these are Linux and Mac OS X. When in a later stage Jem is made to support an other operating system

as well, Windows for example, this support will automatically transfer to all code written on top of it.

One of the things that are accessible through the System class is printing text to the standard output. The standard C++ way of doing this is by using the cout stream object, which is capable of printing a wide range of data types through an overloaded shift operator. Jem uses a similar approach, except that it provides five stream objects instead of one. In increasing order of priority these are debug, info, out, warn and err. These stream objects can all be configured to behave differently, for example to hide data sent to debug or to print a prefix text indicating priority, process id, time, etc. These settings can be read from a configuration file a runtime, which makes it possible to alter the program's output without having to recompile.

The following example code shows 'hello world' through Jem's info stream.

```
1 using namespace jem;
2
3 int main( int argc, char** argv )
4 {
5   System::info() << "hello world\n";
6 }
```

Another useful member of the System class is the traceback function. When this function is called at some point during execution of the program, it will display the sequence of function calls that led to that particular point in code. The exception handler by default calls this function when an exception occurs, thus giving valuable information about what exactly went wrong. Clearly this is very useful in debugging newly written code. Moreover, users of a finished software product will be able to provide the same information after they have come across an undiscovered bug, because the exception handler can remain active after a program has been released.

3.1.2 Properties

The standard data types provided by C++ include character strings, number types such as integers and floats, and arrays of these. High level languages often provide more complex data types such as dynamic lists and databases. Commonly used databases are organized in key-value pairs; in Python these are called dictionaries, in Jem they are **Properties** but these are essentially the same thing. The keys are unique strings that are tied to an object that can be of any type, possibly a new (nested) Properties object. These values can be acquired through a call to either the get or find member functions, the difference being that the former raises an exception when the requested key is not found while the latter silently returns **NIL**, Jem's general null object. Nested Properties can be accessed quickly by joining multiple keys with dots and using this string in either get or find.

The key-value structure suggests a text representation of the form "{key1=value1; key2=value2; ...}", in which the values are formatted according to their own text

representations. For example string objects will be quoted, arrays are enclosed in square brackets and nested Properties objects are formatted as above, i.e. enclosed in curly braces. When a Properties object is printed via one of the stream objects from the System class, it will be formatted like this. The other way round, starting with a text representation, it is possible to build the represented Properties object. Jem comes with a string parser that does exactly this. Its main use is to parse strings read from text files, which gives the user the possibility to define properties at run time. Such a **properties file** can for instance be:

```
num    = 1.25;              // float
solver =                    // nested propery
{
  type    = "GMRES";        // string
  restart = 100;            // integer
};
```

Note that the superfluous curly braces at the highest level are left out. What remains is very similar to standard C syntax. Text starting with a double foreslash are comments, ignored by the parser, and whitespace is not significant which allows the kind of formatting used in the above example. The parser will translate this string to a Properties object with two keys, "num" and "solver", the latter corresponding to a nested property object with again two keys. Using either the find or get methods, the nested property values can be accessed directly as "solver.type" and "solver.restart" which is shorthand notation for the alternative of two consecutive calls. This convention can be used in the **property file** as well, for example writing 'solver.type = "GMRES"' instead of the above scoped notation. Lastly, it is possible to refer to already defined keys, for example adding 'x = solver.restart' would be identical to 'x = 100' in the above example.

3.1.3 Garbage collection

When software projects reach a certain size, memory management becomes increasingly hard and tedious. Dynamically allocated blocks of memory can be used in many different places, and great care should be taken to ensure that this memory is released exactly at the moment that it is no longer used in any of these places. Releasing it too early will mean that data that is still in use can be overwritten, while releasing it too late or not at all will result in diminished performance due to reduced available memory, a situation usually referred to as **memory leaks**. High level programming languages often simplify this task by counting the number of references to each object. When this **reference count** drops to zero, the object is automatically removed — a process known as **garbage collection**. Jem implements a garbage collection scheme of its own that should likewise simplify memory management in C++.

In Jem, not all C++ objects can be used in combination with this scheme, only those that are derived from the **Collectable** base class. Fortunately, most of Jem's objects are. Collectable instances must be made through the **newInstance** function, which

returns a **Ref** instance that behaves much like a pointer to the created object. This means for example that member items are accessible through the arrow operator. For every new reference to that object its reference count is incremented, and it is decremented when the reference is lost. This happens automatically when the local reference variable looses its scope. The object itself is removed only when the reference count has become zero, i.e. directly after the last reference to the object is lost. This way memory management is taken completely out of the hands of the programmer, who can instead focus on main aspects of the program instead of implementation details. In places where performance is crucial, low level management can still be used.

3.1.4 Numerical tools

Jem mainly targets numerical software. A minimum requirement for that is a multi-dimensional data type that can represent vectors and matrices. The standard C++ arrays are quite limited in that they can only be used to store and retrieve elements. Jem's **Array** class is a much more advanced data type that supports arbitrary slicing and resizing and that has many operators defined on it such as scalar multiplication and block assignment. Moreover, this Array object is automatically garbage collected, which is extremely useful for a data type that is usually created and destroyed and passed around so often that it is very hard to keep track of. For reasons of implementation, however, Arrays have their own garbage collection system and are therefore not instantiated by a Ref object.

The Array class is a template class with two template variables that define its type and dimension. For example, Array<int,2> is a two-dimensional array of integers, of a shape determined by its constructor arguments. A 100-element integer vector is created as Array<int,1> vec(100), which claims the memory but leaves the vector uninitialized. The single scalar assignment vec = 0 makes all of its elements zero. Likewise, a 100×100 array of uninitialized doubles is created as Array<double,2> A(100,100).

Instead of dimensions it is also possible to call the constructor with another Array instance, which is then copied to a new location in memory. This is very different from slicing, which creates a new Array instance that points at the same block of memory. An example of this is A(SliceAll,0) which returns the 100-element vector that points at the first column of a matrix A. More complicated slices are allowed as well, such as A(Slice(0,50,10),Slice(0,50,10)) that returns a 5×5 matrix formed of the elements of rows and columns 0, 10, 20, 30 and 40 — the three arguments of the **Slice** object are first index, last index (exclusive) and stride, respectively. Slices provide a flexible way of selecting parts of a matrix without having to worry about underlying memory access. It is important to realize, however, that a slicing operation results in shared memory and that modifications in one object can lead to changes in another.

Besides the basic operations defined on Matrix objects, Jem provides a set of linear algebra tools in the **numeric** namespace. These include a crossProduct and dotProduct function to compute the outer and inner product of two vectors; matmul for

standard matrix-vector, vector-matrix or matrix-matrix multiplications, depending on the type of arguments; and the LUSolver class that contains a set of methods to factor, solve and invert linear systems through LU decomposition. The numeric namespace also contains a SparseMatrix object that stores only the positions and values of nonzero elements. For sparse matrices, which have only a small fraction of nonzero elements, this is much more memory efficient than full matrix storage. An inherent consequence of this storage mode, however, is that SparseMatrix objects are not as full-featured as their full counterparts. Slicing for instance is not supported because the submatrices can no longer use the same memory. However, the basic assignment operators still apply, as do some of the numeric tools like matmul.

3.1.5 Parallel computing

It is often necessary to solve a certain numerical problem in parallel on a cluster instead of on a single processor. Not in the first place to speed up the solution process by having multiple processors working together on the same problem, but because of sheer data size. The matrices that represent the high level of detail required by many current day applications are so large that they simply do not fit in memory completely. A parallel application will therefore need to distribute the matrix over the available nodes and adapt the numerical algorithm to handle this situation.

Obviously, the parallelized algorithm will at certain points need to exchange matrix or vector elements between the various nodes. To make full use of available communication hardware requires full knowledge of all platforms that the software should possibly run on, turned into specialized code. Fortunately, this task is taken out of the hands of the programmer, as this specialized code is usually available in the form of a **Message Passing Interface** library. MPI is the de facto standard for writing parallel software, and most computer clusters provide their own implementation of it, optimized for its specific hardware. The library abstractly defines a wide range of operations such as send, receive, broadcast and gather. Complete information can be found on various websites such as [10].

Strictly speaking, it is not necessary to access MPI through Jem. As MPI is portable, the same holds for applications that use it natively. Nonetheless, Jem makes its functions available through its own **mp::Context** class for two reasons. First is convenience, because it takes some administrative tasks out of the hand of the programmer, who this way does not need to know of underlying libraries. The other reason is that Jem provides its own alternative parallel processing mechanism, based on posix threads. Threads generally have much less overhead than MPI processes, but they are limited to single or multi-core shared memory machines. Using the mp::Context class, the preferred mechanism can be chosen at runtime.

3.1.6 Event handler

In order to improve maintainability of code, Jem encourages a modular software design in which tasks are strictly separated. Ideally these tasks operate without any idea of each other's existence. However, since modifications to shared objects will have global effect, there needs to be a way notifying other tasks of certain events. For example, in a particle collision simulation, one possible division of tasks is 1. generating and deleting particles, 2. calculating gravity forces and 3. calculating collision forces. These tasks can operate completely independently, except that when the first task generates a new particle the other two should be notified of this as they will need to include it in their calculations.

Jem provides an event handler that allows functions to be connected to derived **util::Event** objects. In the above example, the first task can emit a 'new particle' event, that the other two may have connected to functions that make the necessary changes. By this construction none of the three tasks need to know of the others' existence. They only know of an abstract event that can occur, or that they can make to occur. Multiple events can be connected to the same function, and vice versa, which makes the event system very flexible. Note the difference with completely event-driven applications, such as most graphical applications, where events are mainly external. In Jem they are raised by the software itself, and after handling of all connections the normal execution flow is restored.

3.2 Jive

Jem provides a set of very useful, though still quite low level tools to facilitate software development in general. The garbage collector, for instance, will serve its purpose just as well in game development as it will in numerical software, but both situations will still require a set of (completely different) collectable objects in order to put it to use. Because numerical software is the main target, such a set of high level numerical tools has been implemented on top of Jem, forming **Jive**. Figure 3.1 shows that Jem forms an abstraction layer between Jive's components and the operating system, assuring portability of code. The various components include mesh generators, finite element tools, numerical solvers, etcetera. The complete range of tools is too wide to be covered completely in this section. Therefore, only those that are relevant to this report will be discussed here.

Like Jem, Jive is a toolkit that can be used to create numerical software, but that is no software on its own. The user is responsible for putting the various components together in such a way that they compile to a program that does what is intended. This does require some basic programming skills, but it implies a great flexibility in the range of problems that can be handled. The Jive components are designed with extensibility in mind, meaning that they should easily be made to cooperate with non-Jive, problem specific code. The resulting program will be tuned and optimized for a single, specific task, possibly one that none of the closed numerical package on the market are able to perform.

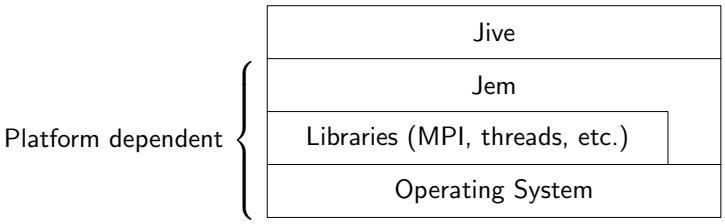

Figure 3.1. *Schematic overview of the Jem/Jive framework. Jem provides a cross-platform interface to the Operating System and certain platform-specific libraries. Through this interface, Jive maintains portability to all supported platforms.*

Most realistic problems will require some amount of problem specific code. More standard problems can be solved by simply putting together of existing components. An example of this is the following complete Jive application, which is built exclusively of standard components. This section uses this example code to introduce some of the most important Jive components — which will be indexed under 'Jive' — meanwhile revealing what exactly this program does, and how. It will be shown that in fact the code below is not enough to answer this question, because many fundamental decisions are made at runtime.

```
1  using namespace jem;
2  using namespace jive;
3
4  Ref<app::Module> mainModule()
5  {
6    fem::declareMBuilders();
7    model::declareModels();
8    femodel::declareModels();
9
10   Ref<app::ChainModule> mainChain = newInstance<app::ChainModule>();
11
12   mainChain->pushBack( newInstance<mesh::MeshgenModule>()   );
13   mainChain->pushBack( newInstance<fem::MPInputModule>()    );
14   mainChain->pushBack( newInstance<fem::PartitionModule>()  );
15   mainChain->pushBack( newInstance<fem::InputModule>()      );
16   mainChain->pushBack( newInstance<app::ControlModule>()    );
17   mainChain->pushBack( newInstance<fem::ShapeModule>()      );
18   mainChain->pushBack( newInstance<fem::InitModule>()       );
19   mainChain->pushBack( newInstance<app::InfoModule>()       );
20   mainChain->pushBack( newInstance<implict::LinsolveModule>() );
21   mainChain->pushBack( newInstance<app::OutputModule>()     );
22
23   return mainChain;
24 }
25
26 int main( int argc, char** argv )
27 {
28   return app::Application::pexec( argc, argv, & mainModule );
29 }
```

3.2.1 Modules

Section 3.1.6 explained how Jem's event handler makes it possible to split a program into components that can operate virtually unaware of each other. Jive strongly builds on this idea by organizing its components in **modules**, which are independent objects that perform certain specific tasks. The only connection between these objects is a globally shared Properties object, named **globdat**. This globdat is where each module expects to find certain specific items, input for its own task, and where it makes its results available for other modules to use. For example, a module can expect to find a matrix and a right hand side vector in globdat, and add to this the solution vector that it computes for this system.

The module system is driven by the pexec function, called by the example's main function in line 35. This function performs some basic initialization, after which it hands over operation to the module specified in its third argument. Technically, the argument is a function that generates a module object, which amounts to the same thing. Module objects are derived from the **Module** base class, which in turn is a Collectable. This means that instances must be made through newInstance, returning Ref objects. Note again that Jive is built firmly on top of the Jem foundation. The Ref objects behave as pointers through which the member functions of the referred Module instance can be accessed. These are:

- init
- run
- shutdown

Part of the pexec's initialization is parsing the command line, which should at least specify a properties file to configure the application's further operation. A properties file was defined in the previous section to be the string representation of a Properties object. This object is created by pexec and handed over to the module's init function, together with the globdat object, to be used in its own initialization. Then the run function is called with the globdat argument, repeatedly, until it returns a status value indicating that the program can shut down. This happens after the module's shutdown is called, which allows it to clean things up, close files, etcetera.

Clearly, a single module does not really constitute a modular design. The mainModule function defined in the example in line 21 therefore returns a ChainModule object, which forwards the calls to init, run and shutdown to a chain of modules that all have a specific task. The mainChain contains modules that mesh a domain, partition it for parallel processing, define a finite element solution space and that solve a linear system, to name a few. The forwarded function calls are made in the same order as modules are added to the chain, i.e. starting with the MeshgenModule and ending with the OutputModule. During run phase the chain is closed, meaning that the first module follows the last, until one of the modules returns a stopping status, initiating shutdown.

3.2.2 Models

Most of Jive's modules are configurable through the properties specified on the command line, which are made available to each individual module's init function. The module bases its own operation on the properties that are relevant to its task. This can go a lot further then simply defining constants and switching options on and off. The example's module chain, for instance, suggests that it performs some kind of finite element simulation, but the nature of this simulation is not clear. The reason is that certain identifying actions are performed by **models**, which are defined at runtime. Models can be viewed as user pluggable functions. Internally, they are **model::Model** objects that implement one main function:

- takeAction

The takeAction function expects three arguments: a string specifying the action to be performed, a properties object used for both input and results, and the globdat object. The actions should be as general as 'build a matrix', 'build a preconditioner', 'provide a set of boundary conditions', etcetera. Many different models can implement actions like these based on completely different grounds. Since the LinsolveModule relies on these actions to build its linear system, this shows that the actual type of problem that is solved is completely model dependent. For example, the following model entry in the properties file will simulate a transport (diffusion, convection) problem with no source term.

```
1  model =
2  {
3    type   = "MP";
4    model =
5    {
6      type    = "Matrix";
7      matrix =
8      {
9        type       = "FEM";
10       symmetric = true;
11     };
12     model =
13     {
14       type       = "Transport";
15       elements   = "all";
16        precision  = 1;
17       mobility   = 1.0;
18     };
19   };
20 };
```

The configured model is of type "MP", which stands for multi-processing. This model takes care of various parallelization issues, but it has no idea of the type of simulation that is performed. For that it has a nested model to which it forwards the actions that it can not perform itself. This "Matrix" model, in turn, has a nested "Transport" model that actually defines the type of problem that is solved. Simply specifying this model at top level will not do because it does not provide all the required actions. This shows that the models must fulfill certain requirements imposed by the modules, which are fixed at compile time. Within these bounds, however, models can be chosen freely.

The model system assures a lot of flexibility even after a Jive application has been compiled. Compared to modules, which are fixed at compile time, models are much more dynamic. The idea is that modules determine the application's global operation only and depend on models to fill in the details. This way, a Jive application can handle a wide range of (mildly) similar problems without requiring recompilation. Besides making experimenting a lot easier by lifting the burden of recompiling, the resulting standalone application is also clearly a lot more useful than one that can only solve a single distinct problem.

3.2.3 Discretization

Jive provides a set of tools that facilitate the discretization of partial differential equations, using for instance finite elements. Some of these tools will be used by models, because these define the system that will be solved. The FEM model in the above properties excerpt, for example, will interpret the 'build a matrix' action as to discretize the equation defined by its nested model, in this case a "Transport" model which holds the Laplace equation. Information such as size and shape of the domain, boundary conditions and finite element type are all obtained from globdat, where it was made available by other modules or models previously executed in the chain.

The resulting system will most likely be solved using an iterative method. This means that the composed matrix is used only in matrix-vector multiplications; individual matrix elements need not be accessible. For this situation, Jive provides an **algebra::AbstractMatrix** and a **solver::Preconditioner** base class with a matmul member that performs this operation. Derived classes are completely free in the way that this operation is implemented. It is for instance possible to work matrix-free and form matrix elements on the fly during the multiplication, without storing anything. Other derived classes will have some form of underlying data structure. Because iterative solution methods will use only the members of the AbstractMatrix base class, models are free in choosing a suitable matrix type for their problem, or defining a completely new one.

A finite element method forms a solution as a linear combination of basis functions, each of which amounts to one or more degrees of freedom, or **DOF**s, in the final linear system. These are the elements of the final solution vector. To keep track of which DOF corresponds to which basis function, Jive provides a **DofSpace**, a bidirectional mapping between DOFs and basis functions that is essential in the

interpretation of the solution of the linear system. The DofSpace forms the bridge between a solution vector and the function that it represents. Actually, it can represent multiple functions, referred to as types. A DofSpace therefore maps a DOF to a unique string-integer pair, where the string denotes the DOF type ("temperature", "v_x") and the integer is an enumeration of basis functions within this type.

Usually, not all degrees of freedom are really free. Some will be subject to certain constraints, for instance imposed by boundary conditions. Dirichlet boundary conditions will simply enforce a value, whereas other types such as Neumann will add linear constraints to the final system. Combined, the constrained solution will have the following form:

$$\boldsymbol{x} = \bar{\boldsymbol{x}} + \boldsymbol{C}\hat{\boldsymbol{x}}, \tag{3.1}$$

with $\bar{\boldsymbol{x}}$ known and $\boldsymbol{C}$ holding all linear constraints. The remaining vector $\hat{\boldsymbol{x}}$ is computed from the smaller, nonsingular system that follows from substituting Equation 3.1 into the original system $\boldsymbol{Ax} = \boldsymbol{b}$ and left multiplying with $\boldsymbol{C}^T$:

$$(\boldsymbol{C}^T\boldsymbol{AC})\hat{\boldsymbol{x}} = \boldsymbol{C}^T(\boldsymbol{b} - \boldsymbol{A}\bar{\boldsymbol{x}}). \tag{3.2}$$

Constraints are usually kept separate from the formation of $\boldsymbol{A}$. They are applied in a later stage, allowing the investigation of different constraints in the same system without having to recompose the matrix. Jive therefore provides a **solver::ConstrainedMatrix** wrapper class, which matmul function performs the $\boldsymbol{C}^T\boldsymbol{AC}$ multiplication without actually forming this matrix. In addition to this, it provides a pre-processor initRhs that transforms a right hand side vector $\boldsymbol{b}$ to that of Equation 3.2, and a post-processor getLhs to transform the solution of this system to the real solution via Equation 3.1. These should be called before and after the linear solution procedure, respectively, after which a solution of the original system is formed that satisfies all the imposed constraints.

3.2.4 Parallel computing

In parallel computations, vectors are usually distributed over computational nodes. To multiply this distributed vector with a matrix, this matrix needs to be distributed over the nodes as well, in such a way that corresponding matrix and vector elements are hosted on the same node. This is schematically shown in Figure 3.2 for the case of three computational nodes. To perform the multiplication, the relevant parts of the vector will need to be present on all the nodes, which forces a communication step. By distributing the vector elements in such a way that they correspond to coherent subsections of the physical domain, however, the sparsity structure of the matrix will in general be such that only a few 'neighbouring' elements will need to be exchanged.

Parallel programs are often written in such a way that exactly the same code runs on all the nodes, using a unique **process id** to distinguish the various cooperating processes at runtime. This id is made available through Jem's mp::Context object,

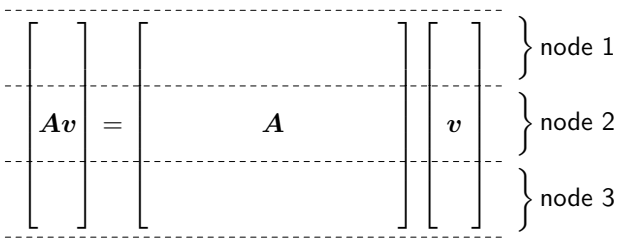

Figure 3.2. *Schematic overview of the distribution of matrix and vectors over multiple computational nodes in a parallel computing environment. The matrix-vector multiplication can be performed per node using only a subset of matrix rows, but some communication will be required to acquire off-node vector elements.*

which provides access to low level communication — communication is a typical case where distinction based on process id is required because it is something that has to be coordinated with other processes. The above described parallel multiplication will for instance require exchanging Array elements with neighbouring nodes. After that, the procedure is the same on all processes: the assembled vector is multiplied with the local submatrix, yielding a new local Array object that is part of the global result. Therefore, apart from the communication step, processes need not even know that they are doing only part of the work.

This idea is the basis of Jive's **algebra::MPMatrixObj** class, which is a matrix wrapper like the solver::ConstrainedMatrix class. Its matmul operation performs the necessary communication and forwards the assembled vector to the wrapped matrix' matmul function. The MPMatrixObj class hides all parallel aspects inside the matmul function, a member of the AbstractMatrix base class. This means that algorithms that depend only on matrix-vector multiplications — most notably, iterative solution methods — are automatically parallel when applied to a matrix of this type. Of coarse, some preparations must be made beforehand, such as distributing the vector over the nodes. The MPMatrixObj object will be the result of these preparations.

It is true that iterative solution methods use matrices only in the context of matrix-vector multiplications. There are, however, other operations that require communication as well in a parallel computing environment. These are operations that involve vectors only, such as inner products. After computing an inner product on each computational node, the local results need to be gathered, summed and redistributed to have the global inner product available on each node. To keep this distinction out of code as well, Jive provides **VectorSpace** and **mp::VectorExchanger** classes, with multi-process counterparts, through which many of these vector operations can be performed.

3.2.5 Linear Solvers

Returning to the example code at the beginning of this section, a global picture of the program's operation can now be formed. Most of it is already given away by the module names. For instance the MeshgenModule, first in the chain, generates a mesh on a user defined domain and makes it available in globdat. The three modules from the app namespace handle input and output to the user. The five fem namespace modules in the chain jointly define a problem, based on the mesh found in globdat, and store their results there as well. This problem is then solved by the **implict::LinsolveModule** next in the chain. Because of its relevance to this report, this module will be discussed in more detail.

Instead of taking objects directly from globdat, the LinsolveModule fulfils most of its dependencies via model takeActions. The matrix, preconditioner and constraints are acquired this way during the init phase. During run phase a right hand side vector is acquired, as well as an initial left hand side vector, which is to be replaced with a sufficiently accurate solution of the system. This is where the LinsolveModule returns its main result, obtained from either a direct or an iterative solution procedure. This procedure is determined at runtime via the 'linsolve.solver' properties, as are all other solver settings. For example, the following properties excerpt defines a restarted GMRES procedure that targets a residual over right hand side ratio of 0.001, using an ILU0 preconditioner if no other is specified by a model.

```
1 linsolve.solver =
2 {
3   type          = "GMRES";
4    restartIter  = 200;
5   precon.type   = "ILU0";
6    precision    = 1e-3;
7 };
```

Other solver types provided by Jive include iterative solvers such as CG and GCR, as well as some direct solvers. A solver in Jive is a separate object, derived from the **solver::Solver** base class. This object is created during init phase, based amongst others on the matrix, preconditioner and constraints that are provided by the various models. These can optionally be updated at later times, which makes it possible to solve a different system in each module chain run. This is useful for instance in time simulations. In order to deal with these late changes the solver can use Jem's event framework to connect to certain events, for instance to alter internal data structures upon matrix changes, if such is necessary.

Upon creation, the solver object is of the right type and contains the relevant information about the system, but other than that is still unconfigured. For configuration, the solver::Solver class defines two member functions:

- configure
- getConfig

The configure function receives a single Properties argument that corresponds to the solver entry in the properties file, 'linsolve.solver' in the above example. Solver objects that are nested, or otherwise used differently, will be configured elsewhere but this is hidden from the object itself. The solver just receives a Properties object and uses this to configure itself, using default values for non-specified keys as much as possible. To present a complete overview of which settings are really used, including default values, and which are ignored, getConfig is called next with an empty Properties object that is to be filled with these actual settings. It can therefore be seen as the exact opposite of the configure function, forming properties based on configuration instead of configuring based on properties.

Once the solver object is created and properly configured, it can be put to use for the actual solving. This is done by specifying a right hand side vector for the configured system, using one of the following functions:

- solve
- improve

Besides the right hand side vector both functions also require a left hand side vector that they can replace with the generated solution of the system upon return; the right hand side remains unchanged. The only difference between the two functions is that improve uses the left hand side as an initial guess, whereas solve ignores its initial contents and starts searching simply from zero. During iterations the left hand side is repeatedly improved in place. This is stopped as soon as the residual over right hand side ratio drops between a user specified threshold, the 'precision' property in the above example.

To keep track of the various available solver objects, instances are created via a so called solver factory. This requires two additional functions that can be called prior to making the instance, typically organized as static members of the solver class.

- declare
- makeNew

The first function, declare, registers the second function, makeNew, under a certain name at the solver factory. When the LinsolveModule requests, for instance, a "GMRES" solver object, the factory looks it up in its database and calls the corresponding makeNew function. This function will call the solver's actual constructor, often after extracting the required data from the globdat. The factory approach is used extensively in Jive because it provides a flexible way of adding new components without having to alter code. Other objects that are managed this way include models and preconditioners.

3.3 Deflation

The aim is to make the deflation technique, introduced in Chapter 2, part of Jive. This requires a number of early design decisions. For instance, Section 2.4 defines three slightly different variants of deflation from which one should be chosen; Equation 2.36 that is equivalent to Algorithm 2.3, Equation 2.37 that uses a left preconditioner and Equation 2.38 that uses a right preconditioner. As shown in Equations 2.33 to 2.35, iterates are formed in exactly the same spaces so convergence speed should be roughly the same for all three options. This and other decisions will therefore be based mostly on ease of implementation and the level of integration with existing Jive components. As a general principle: the more code that can be reused, the better.

'Normal' preconditioners are in general applicable on both the left and right hand side of the matrix. For deflation this is different, because Section 2.4 showed that the operator changes shape depending on the side that it is used. Because all Jive solvers use right preconditioning this forces the use of the right preconditioned variant as otherwise all solvers will need to be recreated in a specialized deflation version. A very bad example of code reuse, and dramatic for the maintainability of Jive as a whole. The only benefit of a standalone deflation solver, resistance to changes in other components like the side of preconditioning, can not turn the scales. This means that deflation in Jive will be based on the right preconditioned deflation variant, Equation 2.38, repeated here for convenience:

$$\boldsymbol{x}_i = \boldsymbol{x}_0 + \boldsymbol{P}\boldsymbol{y}_i, \quad \boldsymbol{A}\boldsymbol{P}\boldsymbol{y} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0, \tag{3.3}$$

where $\boldsymbol{P} = (\boldsymbol{I} - \mathbb{P}_{\mathcal{Z}})\boldsymbol{M}^{-1}$ is the right preconditioner and $\boldsymbol{x}_0 = \tilde{\boldsymbol{x}}_0 + \mathbb{P}_{\mathbb{Z}}(\boldsymbol{x} - \tilde{\boldsymbol{x}}_0)$ the initial guess. In these, $\mathcal{Z}$ is the deflation subspace, $\mathbb{P}_{\mathcal{Z}}$ the projection onto this subspace, $\boldsymbol{M}^{-1}$ a standard preconditioner (such as block ILU) and $\tilde{\boldsymbol{x}}_0$ the initial guess that is provided to the solver — possibly zero.

The real initial guess $\boldsymbol{x}_0$ that is used by the iterative solver is enhanced by the projected error. This is the 'pre-processing step' mentioned in Section 2.4. If not for this single step, deflation would take the form of a normal preconditioner, albeit forcedly used on the right hand side. Unfortunately, this step is a crucial part of deflation without which the final solution lacks its deflation subspace component. This leaves no other option than to make deflation a solver, rather than a preconditioner, that simply starts a second 'nested' Jive solver as soon as the necessary preparations have been made. This nested solver will use the deflation preconditioner that is supplied internally by the solver.

Appendix A contains the complete source code for a deflation solver based on this principle. The preceding introduction to the Jem/Jive toolkits should be sufficient to understand the main ideas. This section discusses the design of this solver and the closely connected preconditioner, with references to the relevant parts in code. The discussion has been divided into four parts, corresponding to the four configurable components of the solver: the deflation subspace $\mathcal{Z}$, the projection $\mathbb{P}_{\mathcal{Z}}$, the preconditioner $\boldsymbol{M}^{-1}$ and the nested iterative solver. Ahead of explanation, a typical configuration may look as follows:

```
1  solver = {
2    type = "Deflation";
3    subspace = {
4      type = "Subdomain";
5      dofs = [ "u", "v" ];
6      linear = true;
7    };
8    projection = {
9      type = "Standard";
10   };
11   precon = {
12     type = "ILU0";
13   };
14   solver = {
15     type = "GMRES";
16     precision = 1e-3;
17   };
18 };
```

3.3.1 Subspace

Chapter 2 showed that the class of Krylov subspace methods is named after the search space that these methods build to form a solution. Deflation was shown to be a means of augmenting this space with a set of predefined vectors, spanning what is called the deflation subspace $\mathcal{Z}$. The first iteration is formed completely in this deflation subspace; it is the initial vector x_0 in Equation 3.3. Subsequent iterations approach the true solution in a Krylov subspace that is built orthogonal to the deflation subspace — orthogonal with respect to the norm used in projection $\mathbb{P}_{\mathcal{Z}}$. The orthogonality ensures that the deflation subspace component needs only be determined once and does not require adjustment during the iterative process.

The deflation vectors are free to choose; Section 2.4 discussed some general ideas about deflation vectors that can be expected to improve convergence. A common element is that deflation vectors are closely connected to the physics underlying the system of equations. They can for example represent a constant temperature on part of the physical domain. By manually making this link, deflation can be used to bring expert knowledge from physics into the numerical algorithm. In this respect it is convenient to make the connection to simulated variables ('DOF types' in Jive) already in code, resulting in the 'subspace.dofs' property that can be used to select types or groups of types for which deflation vectors should be created.

The initSubspace function (line 522) is in charge of forming the deflation subspace. For each configured DOF type it builds two arrays: one with the vector indices that correspond to that DOF, and one with the corresponding element id's. These arrays are then handed over to a specialized vector builder that forms the actual deflation vectors, selected by the 'subspace.type' property. The selected vector builder can have its own additional properties for further configuration, such as 'subspace.linear' for the "Subdomain" type selected in the above example. After some additional

checking of the generated vectors, initSubspace adds them to the subspace matrix and moves on to the next configured DOF.

Currently, the number of vector builders to choose from is limited; no more than three (quite general) subspace types have been hard coded in the solver. The idea is to add the possibility for user defined functions in future versions of the solver, that can be tailor made for various kinds of problems. The vector builder approach has been taken with this in mind, and should be quite easily extendible to user defined functions by using a factory object similar to that used for solvers and preconditioners. Until that time, however, only the following types of deflation can be selected.

Type: Empty

The vector builder for this subspace type is the getEmptySubspace function, line 592. Since all it does is return an empty matrix, this leaves a deflation subspace of dimension zero which effectively disables deflation. As such, the "Empty" subspace is a convenient way of temporarily switching of deflation, falling back on the nested solver and standard preconditioner as though these were configured directly. Especially in testing this is often useful for quick comparison.

Type: Subdomain

The "Subdomain" subspace is the first real deflation type, linked to the getSubdomainSubspace vector builder at line 603. The generated subspace corresponds to that proposed in Section 2.4.3: when the physical domain has been decomposed into several subdomains, usually prior to a parallel computation, the generated vectors represent solutions (for the DOF type under consideration) that are constant valued on a single subdomain and zero on all the others. The subspace spanned by this set of vectors corresponds to the set of piecewise constant solutions over the various subdomains. Section 2.4.3 explained that convergence should improve depending on how closely these solutions manage to approximate the eigenvectors that belong to the smallest few eigenvalues of the system.

A problem arises when the various subdomains have a certain amount of overlap. Simply creating a constant vector for each complete subdomain can not be right as then the constant solutions are no longer part of the spanned subspace. There are two different ways of fixing this, both implemented and controllable through the 'subspace.blend' property. When false, each DOF in the overlap regions is being uniquely assigned to a single subdomain, thus fixing the problem by effectively removing overlap. The other option (blend is true) is to divide each DOF in the overlap region by the amount of subdomains sharing it. This way the deflation vectors will again add up to a constant solution, ensuring that these are still part of the spanned subspace. The latter situation is shown in the top row of Figure 3.3.

Section 2.4.3 also mentioned the possibility of higher order subdomain vectors in ad-

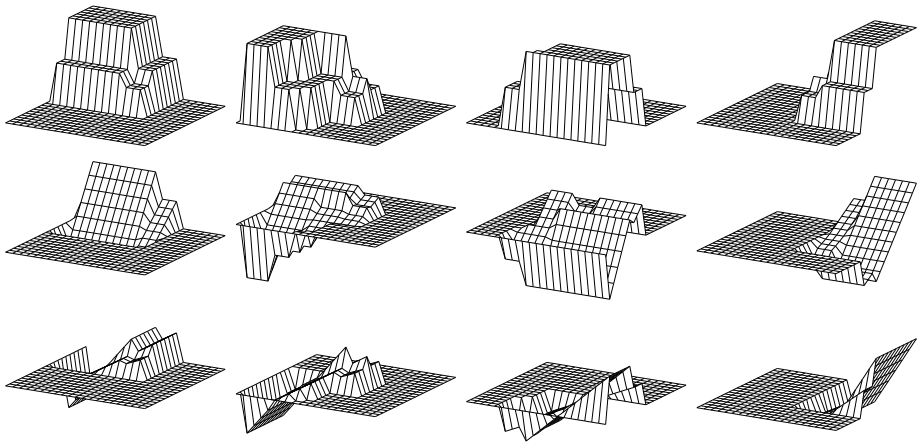

Figure 3.3. *Deflation vectors generated by the "Subdomain" vector builder with subspace.blend set to true, using a decomposition in four subdomains. The top row shows the piecewise constant vectors, corresponding to subspace.linear set to false; the lower two are added when subspace.linear is set to true.*

dition to the constant vectors. These vectors should enrich the deflation subspace, bringing it closer to the targeted eigenvectors and thus further increasing convergence speed. An important complication compared to the constant subdomain case is that to build these higher order vectors, each DOF needs to be connected to a physical coordinate. This is why the initSubspace function supplies an array with vector id's in addition to the vector indices, for which coordinates can be looked up. The "Subdomain" vector builder implements only first order vectors, activated by the 'subspace.linear' property. When set to true, an additional set of subdomain vectors is added for each physical dimension; two for the situation shown in Figure 3.3 — the last two rows show the resulting vectors.

Type: Basis

The "Basis" subspace is the other deflation type currently provided by the solver, that is linked to the getBasisSubspace vector builder at line 662. This vector builder takes a quite different approach than the Subdomain type. Instead of using an existing decomposition in subdomains, it performs its own decomposition by computing the bounding box of the domain and dividing it into rectangular cells. The advantage of this type of decomposition is that it makes it possible to define a deflation subspace that is inherently continuous; the "Subdomain" type subspace clearly is not. Since there is no reason to assume that eigenvectors are discontinuous on subdomain boundaries, this space (the linear variant) is quite a bit too rich — a waste.

The deflation vectors formed by this vector builder are similar to the basis functions used in finite element methods. When the physical domain has dimension n and the rectangular cells have size $\delta_1 \times \cdots \times \delta_n$, a deflation vector is formed for each

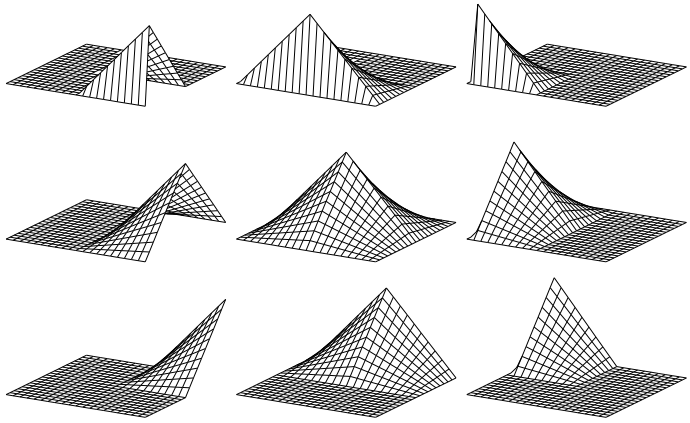

Figure 3.4. *Deflation vectors generated by the "Basis" vector builder with 'subdomain.nodes' set to 3, given the same square domain as in Figure 3.3.*

grid point $\boldsymbol{p}$ based on the mapping $\mathbb{R}^n \to [0,1]$:

$$f_{\boldsymbol{p}}(\boldsymbol{x}) = \begin{cases} \left(1 - \frac{|x_1 - p_1|}{\delta_1}\right) \dots \left(1 - \frac{|x_n - p_n|}{\delta_n}\right) & \boldsymbol{x} - \boldsymbol{p} \in [-\delta_1, \delta_1] \times \dots \times [-\delta_n, \delta_n] \\ 0 & \text{else} \end{cases} \tag{3.4}$$

This function is zero in all cells that do not contain $\boldsymbol{p}$, one only in $\boldsymbol{p}$, and continuous throughout the domain $\mathbb{R}^n$. On a square two dimensional domain, divided into 2×2 cells, this results in the nine basis functions shown in Figure 3.4.

Adding up to one, these functions again span a function space that contains at least the constant solutions. The fact that all functions in this space are continuous makes it seemingly more numerically efficient than the "Subdomain" type. At the same time, however, a large potential for parallel efficiency is lost. The property that "Subdomain" type vectors have local support on a single process makes it possible to implement the formation and inversion of the small projection matrix in a parallel efficient manner. This benefit is largely lost when deflation vectors have support that overlaps multiple subdomains, as is the case here. This is why deflation vectors are preferred to have local support on single subdomains, even though the current solver has no way of exploiting this yet. For testing purposes, though, it will be interesting to see how the "Basis" type deflation subspace behaves in comparison to the more common "Subdomain" type.

3.3.2 Projection

When the deflation subspace $\mathcal{Z}$ has been defined, the next step in implementing deflation is the projection $\mathbb{P}_{\mathcal{Z}}$, defined in Equation 2.6. This projection maps a general vector to the closest vector in $\mathcal{Z}$, measured in a certain norm. Sections 2.3.3 and 2.3.4 explained that possible norms are the matrix $\boldsymbol{A}$ and $\boldsymbol{A}^T\boldsymbol{A}$-norms, because these make it possible to project the unknown solution of the problem. Using this and Theorem 2.6, which gives an explicit expression for this projection, the projection

onto a subspace spanned by the columns of $\boldsymbol{Z}$ can be implemented as follows:

$$\mathbb{P}_{\text{col}\,\boldsymbol{Z}} = \boldsymbol{Z}(\boldsymbol{Y}^T\boldsymbol{A}\boldsymbol{Z})^{-1}\boldsymbol{Y}^T\boldsymbol{A}, \tag{3.5}$$

where $\boldsymbol{Y} = \boldsymbol{Z}$ corresponds to a projection in $\boldsymbol{A}$-norm ("Standard" deflation) and $\boldsymbol{Y} = \boldsymbol{A}\boldsymbol{Z}$ corresponds to a projection in $\boldsymbol{A}^T\boldsymbol{A}$-norm ("GMRES" deflation). Both variants are supported in the current solver, using a construction similar to the vector builders from the previous section. The 'projection.type' property selects either getStandardProjection on line 754 or getGMRESProjection on line 762, that simply yield the corresponding set of vectors $\boldsymbol{Y}$. The real initializing work is done by the initProjection function on line 718.

To efficiently perform the projection operation, Equation 3.5, a number of different approaches can be followed. To name:

1. Store $(\boldsymbol{Y}^T\boldsymbol{A}\boldsymbol{Z})^{-1}$ and $\boldsymbol{Y}$

   The matrix $\boldsymbol{Y}^T\boldsymbol{A}\boldsymbol{Z}$ is in general very small, with dimensions equal to the number of deflation vectors. Because the projection $\mathbb{P}_{\mathcal{Z}}$ is part of the preconditioner $\boldsymbol{P}$ in Equation 3.3, and will therefore need to be performed in each iteration of the solution method, this is one of the rare occasions where a direct matrix inversion will actually pay off. When this inverted matrix is available on all of the computational nodes, performing the projection is a matter of four matrix vector multiplications, two of which requiring communication.

   The "GMRES" projection requires additional storage of the vectors $\boldsymbol{Y} = \boldsymbol{A}\boldsymbol{Z}$ because the transposed matrix $\boldsymbol{A}^T$ is not directly available — left multiplication is not defined. This necessity turns out to be quite convenient, because it hides the difference between the two variants of deflation. During initialization the $\boldsymbol{Y}$ vectors can be defined to be either the same block of memory as the deflation vectors $\boldsymbol{Z}$, or a new block of memory containing $\boldsymbol{A}\boldsymbol{Z}$. During iterations the distinction between the two variants is lost.

2. Store $(\boldsymbol{Y}^T\boldsymbol{A}\boldsymbol{Z})^{-1}\boldsymbol{Y}^T\boldsymbol{A}$

   When an additional set of vectors needs to be kept in memory anyway, such as being the case with "GMRES" deflation, storing $\boldsymbol{Y}$ is not the best thing to do. The four matrix vector multiplications that are required for one projection can be reduced to two by performing these during initialization and storing the result. This will simplify the projection operation to a transformation from a large to a small vector (first multiplication) back to a large vector (second multiplication by $\boldsymbol{Z}$). What is more, the small matrix does not need to be stored separately, even freeing some memory for the case of "GMRES" deflation.

   There are, however, two problems. First, the absence of a left multiplication operation restricts this technique to symmetric matrices, for which left and right multiplication are identical. Because symmetric matrices will generally be used in combination with "Standard" deflation, option 1 is to be preferred. Second, incorporating the multiplication by $\boldsymbol{A}$ makes it impossible to project the unknown solution $\boldsymbol{x}$, which can otherwise be performed by substituting the right hand side for $\boldsymbol{A}\boldsymbol{x}$. Although this is necessary only once during initialization and can therefore be computed as an intermediate step, this

does mean that the set of vectors needs to be recreated for each new right hand side.

3. Store $(\boldsymbol{Y}^T\boldsymbol{A}\boldsymbol{Z})^{-1}\boldsymbol{Y}^T$

   Both these problems are solved by computing the small matrix times $\boldsymbol{Y}$ product during initialization and leaving the multiplication by $\boldsymbol{A}$ for during the actual projection. This projection will then require three matrix vector multiplications, two of which requiring communication: multiplication by $\boldsymbol{A}$ and multiplication by $(\boldsymbol{Y}^T\boldsymbol{A}\boldsymbol{Z})^{-1}\boldsymbol{Y}^T$, stored as a set of column vectors just like the deflation subspace $\boldsymbol{Z}$. The last multiplication by $\boldsymbol{Z}$ does not require communication. Compared to option 1 a single communication-free multiplication will be lost, at the cost of additional memory for "Standard" deflation but actually freeing some memory for "GMRES" deflation. And, as $\boldsymbol{A}$ is multiplied during projection, projecting the unknown solution is still possible.

Option 3 is chosen over 1 mainly for reasons of simplicity; the additional required memory (equal to that of the deflation subspace) is not expected to be a serious problem, and the reduced work per iteration is a welcome side effect. The set of vectors is computed by the initProjection function and stored in the projection matrix. Currently the small matrix is inverted on all computational nodes simultaneously, not using any form of cooperation. Optimizations are clearly possible here, although they do depend heavily on the structure of the deflation subspace. However, because this inversion needs to be performed only once for every new matrix, the true benefits are questionable.

3.3.3 Preconditioner

Nearing completion, the deflation preconditioner $\boldsymbol{P} = (\boldsymbol{I} - \mathbb{P}_{\mathcal{Z}})\boldsymbol{M}^{-1}$ lacks only a standard preconditioner $\boldsymbol{M}^{-1}$. This preconditioner is defined by the 'precon.type' property. The properties shown on the beginning of this section for example define $\boldsymbol{M}^{-1}$ to be a block ILU preconditioner. Additional 'precon' properties are possible for specific configuration of the selected type. Normally, the selected preconditioner is created by the solver's makeNew function and handed over to the solver's constructor. Here, due to the nested solver and the added deflation projection, things are slightly different.

The preconditioner object is formed in makeNew, line 1012, as usual. Here, instead of being handed over to the solver (either the DeflationSolver or the nested solver) it is used in the construction of a DeflationPrecon object, which is derived from a jive::solver::Preconditioner. Its matmul function (line 822) subsequently performs the multiplications $\boldsymbol{M}^{-1}$ and $\boldsymbol{I} - \mathbb{P}_{\mathcal{Z}}$, effectively making this object matrix $\boldsymbol{P}$. This object is then handed over to the nested solver, which knows only about its Preconditioner nature and can therefore use it only in this limited context. To the DeflationSolver, several more functions are available.

3.3.4 Solver

The DeflationSolver object is created at the end of makeNew, line 1012, after all other objects have been created. It requires only two objects: the DeflationPrecon object for the additional functionality it offers, and the nested solver that already received this object during creation. The nested solver type is determined by the 'solver.type' property, which can be "CG", "GMRES" or any of the iterative solvers offered by Jive. Again the type is optionally accompanied by additional solver specific settings, such as a 'solver.precision' to set a stop criterion. Any specified 'solver.precon' will be ignored because the deflation preconditioner is supplied internally.

Upon calling solve or improve, line 898, the DeflationSolver must start solving Equation 3.3, involving the deflated system $\boldsymbol{AP} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$. The difference between solve and improve is that the former does not use its left hand side vector as an initial guess, and improve does. For deflation, however, the initial guess must always contain the projected solution $\mathbb{P}_{\mathcal{Z}}\boldsymbol{x}$, so in both cases the nested solver is handed an initial vector to improve. This initial vector $\boldsymbol{x}_0$ is computed by the DeflationPrecon's 'improve' function, line 797. When the DeflationSolver is called as 'solve', this function is simply handed a zero vector.

Recall from Section 3.2.3 that any constraints imposed on the solution are not normally worked into the system matrix. Instead, given a set of linear constraints $\boldsymbol{x} = \bar{\boldsymbol{x}} + \boldsymbol{C}\tilde{\boldsymbol{x}}$, the first thing a solver does is transform the system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ into the constrained system $(\boldsymbol{C}^T\boldsymbol{A}\boldsymbol{C})\tilde{\boldsymbol{x}} = \boldsymbol{C}^T(\boldsymbol{b} - \boldsymbol{A}\bar{\boldsymbol{x}})$. This means that the initial projection should be based on this right hand side, rather than $\boldsymbol{b}$. The DeflationPrecon's improve function computes this right hand side using the matrix' 'getRhs' function and uses this to compute the deflation subspace projection. This deflation subspace was created such that the result is automatically in 'tilde form', i'e' not containing any of the constraints. Upon return this is put back in normal form by 'getLhs', which makes it a valid first approximation of the solution.

After improving the initial guess, the nested solver's improve function is called. This immediately undoes the latest transformation by calling initLhs, which removes the Dirichlet boundary conditions $\bar{\boldsymbol{x}}$. This may seem like a waste, but it has a clear programming advantage over staying in 'tilde form' after improving the initial guess and preventing the nested solver from transforming the system — which is possible. This would mean that pre- and post processing of the system become the responsibility of the DeflationSolver, which currently does not need to know anything of constraints, matrices, or anything. This is the sole terrain of the DeflationPrecon object; the DeflationSolver merely controls. Also in terms of code reuse this would be bad, and most of all, the added work is negligible.

The nested solver will repeatedly improve its left hand side until the configured precision is attained. This precision is measured by the residual over right hand side ratio; iterations stop as soon as this ratio drops under the threshold value configured through the 'solver.precision' property. Deflation does not influence this process; since no post processing of the generated solution is performed after the nested solver returns, the solver has full view of the solution it generates and therefore

does not require changes in its stop criterion. Note that for left preconditioned deflation, which does require post processing of the generated solution, this would be different.

Chapter 4

Numerical results

At this point, the Variational Multi Scale method that was introduced in Chapter 1 is a method for which the benefits of deflation are very much an open question. The nature of the deflation method is such that it is very difficult to judge beforehand if the method will yield great improvements in convergence. Even the question which deflation subspace will be best suited is far from trivial. The main guideline is that eigenvectors should be approximated, but this can be done in many ways. One approach is to use information from underlying physics, but this is not always possible. Another approach is to approximate eigenvectors in a general function space, such as that of piecewise constant solutions. Notwithstanding the various underlying ideas, actually predicting the performance gains is quite something different. For this, the only reliable source of information is numerical experiments.

Instead of immediately running experiments on VMS, however, it is important to test if the current implementation is functioning properly in situations where deflation is already known to work, or at least where its behaviour is known. This to reduce the chance of programming errors and other possible mistakes. Fortunately, many deflation experiments have been performed and documented in articles on this subject; these can all be used for comparison. A second reason for not turning immediately to VMS is that a simpler problem will allow for easy visualisation of the solution and intermediate steps, which will provide practical insight in the various mechanisms at work. The four-dimensional VMS time slabs are somewhat more complicated to visualise.

This Chapter will present the results obtained with deflation on two distinct classes of problems. Section 4.1 starts with the two dimensional Laplace equation, a well known problem that has a clear physical interpretation and for which many external results are available for reference. This problem will be used both for checking the solver implementation and for testing some new features such as the "GMRES" projection for general matrices. Finally, Section 4.2 will apply deflation to the VMS problem.

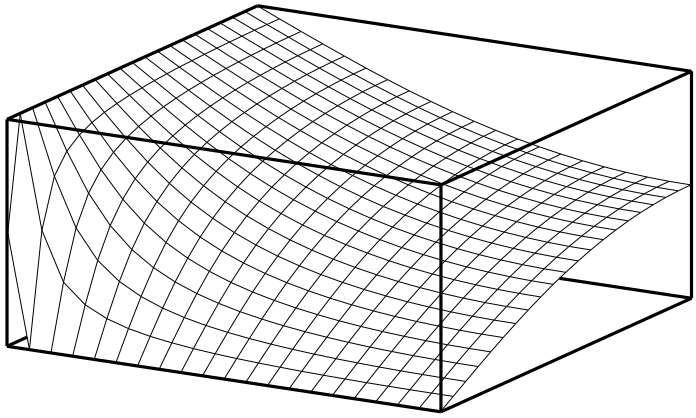

Figure 4.1. *Exact solution of Equation 4.1 discretized on a* 21×21 *grid, representing the steady-state temperature distribution of a square plate that is heated on one side and cooled on another.*

4.1 Laplace test problem

One problem for which deflation is known to yield good results is solving the Laplace equation. Many articles on deflation have used this problem in convergence tests, for example Saad et al. [15], Nabben and Vuik [4] and Verkaik [18]. This means that a large amount of both qualitative and quantitative information is available for this problem, which makes it ideal for initial testing of the created solver. The results of these tests will be compared to those found by the named authors to see if they match. This will prove that the solver is functioning properly on this single problem, which increases the chance that it will do so on other problems as well. It is impossible to be completely sure in software.

The test problem that will be used throughout this section is the two dimensional Laplace equation on a square domain, with Dirichlet boundary conditions on two adjacent sides and Neumann boundary conditions on the remaining two:

$$u : [0,1] \times [0,1] \rightarrow [-1,1] \quad \begin{cases} u_{xx} + u_{yy} = 0 & 0 < x < 1, \quad 0 < y < 1 \\ u = -1 & 0 < x \leq 1, \quad y = 0 \\ u_x = 0 & x = 1, \quad 0 \leq y \leq 1 \\ u_y = 0 & 0 \leq x \leq 1, \quad y = 1 \\ u = 1 & x = 0, \quad 0 < y \leq 1 \end{cases} \tag{4.1}$$

Physically, the solution to this problem can be interpreted as the temperature distribution of a square plate that is held at a constant high temperature on one side and at a constant low temperature on another, adjacent side, after a steady state has been formed. This distribution is shown in Figure 4.1.

The results in this section are obtained using the same Jive application that was used as an example in Section 3.2. As explained in Section 3.2.2, this application can be made to simulate a wide variety of problems because of to the model chain that is defined at run time. This includes the test problem defined in Equation 4.1. The

relevant module chain has already been shown in Section 3.2.2, where a "Transport" model was nested inside a "FEM" matrix model. The transport model defines the transport equation, of which Laplace is a special case; the FEM model uses this in its finite element procedure to build the coefficient matrix. The result is as though the following nine-point stencil was used:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}. \tag{4.2}$$

The system is solved using non-restarted GMRES, not CG as would perhaps be expected for a symmetric problem such as Laplace. Obviously, in real life, CG would be preferred for its low memory usage due to the short recurrence property. In this testing phase, however, memory is not an issue and the convergence of both methods is the same; when loss of orthogonality due to numerical inaccuracies is ignored, the Krylov subspaces that are built by both algorithms are exactly equal. The only difference is in the projection of the iterates, and as the convergence plots will use the residual as a measure for the solution accuracy, it seems best to find an optimal Krylov element based on this criterion. More practically, in GMRES the residuals are automatically available whereas CG offers only approximations and would therefore require modification.

4.1.1 Deflation versus no deflation

An important application of deflation is in parallel computing environments, where certain restrictions hamper the creation of 'classic' preconditioners. The problem is that data is distributed over a set of computational nodes, which means that heavy communication will be required for global operations such as computing an (approximate) matrix inverse. High latency and limited bandwidth make that this kind of operation is no longer feasible, which discards a large set of preconditioners that are commonly used in non-parallel algorithms. They are often replaced with block variants of the existing preconditioners, such as block ILU. As these blocks can be built in parallel, requiring no communication, these block preconditioners are highly suited for parallel computations.

The gains in parallelism, however, come at a cost. The inherent locality of block preconditioners makes that they are ill capable of changing the lower part of the eigenvalue spectrum, which belongs to the slowly varying, global eigenvectors. As a result the smallest eigenvalues will stay more or less in place, thus hampering convergence according to Theorem 2.3. Figure 4.2 compares convergence for four different preconditioners, three of which block ILU0 and the last one diagonal scaling. At 16 blocks already block ILU0 is only slightly better than diagonal scaling in terms of iterations, and accounting for the extra work per iteration the latter is to be preferred. Diagonal scaling can in fact be considered the limiting case of block ILU, when no overlap is used; the fact that block preconditioners tend to this limit illustrates the problem.

Deflation is meant to solve this problem by improving the existing preconditioners within the restrictions imposed by the parallel environment. Section 2.4 explained

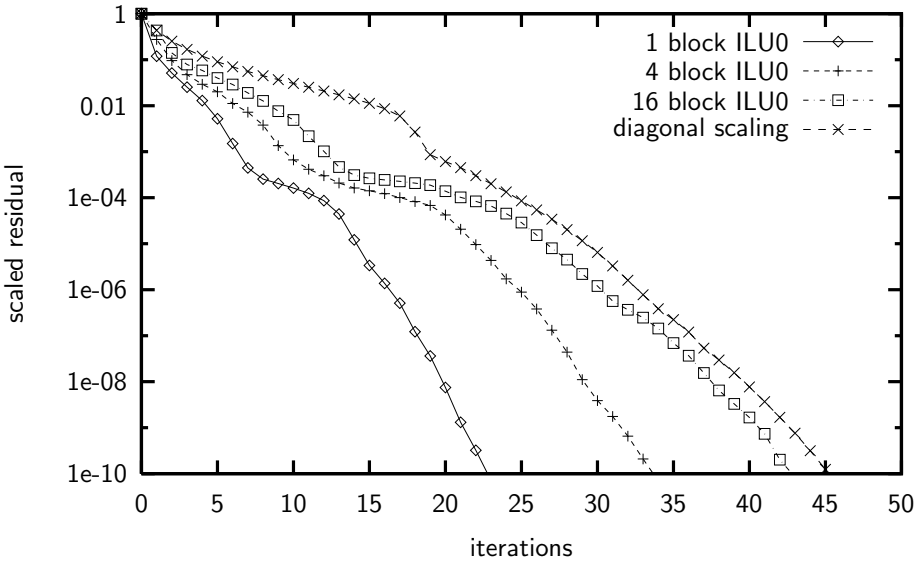

Figure 4.2. *GMRES convergence of the test problem on a coarse* 21×21 *grid, comparing four different preconditioners: 1, 4 and 16 block ILU0 and diagonal scaling. The plot shows that block ILU0 deteriorates quickly as the number of blocks increases, to the point where convergence is almost identical to that of simple diagonal scaling.*

that a central element of this method is a projection onto a predefined deflation subspace, that minimizes the distance in a convenient norm. Typically used is the matrix $\boldsymbol{A}$-norm which is also the basis of the CG method. This, however, is only a valid norm for matrices that are symmetric and positive definite (SPD). For this reason the solver offers the alternative of minimizing in $\boldsymbol{A}^T\boldsymbol{A}$, which yields a valid norm for any matrix. Because the Laplace matrix is SPD, however, initial tests will be based on the former projection, labeled "Standard". This is the same projection that has been used in all reference articles.

The deflation subspace can currently be selected from two configurable presets, the "Subdomain" and the "Basis" subspace, introduced in Section 3.3.1. When a deflation subspace has been selected, the deflated Krylov method starts with projecting the exact solution onto this subspace. For the test problem this solution was shown in Figure 4.1, and the resulting projections on the example subspaces in Figures 3.3 and 3.4 are shown in Figure 4.3; the corresponding residuals shown on the right. The projections form the start vector of the deflated Krylov process that will be started next — that is to say, except for the unlikely situation that the exact solution was part of the deflation subspace and no further improving will be required.

Table 4.1 shows convergence results obtained with "Subdomain" type deflation, comparing various variants including the upper two from Figure 4.3. The table lists the number of iterations required for reaching various levels of accuracy, using non-restarted GMRES and three different block ILU0 preconditioners, ranging up to 16 blocks. Judged by these numbers it seems that the blended variants — in which

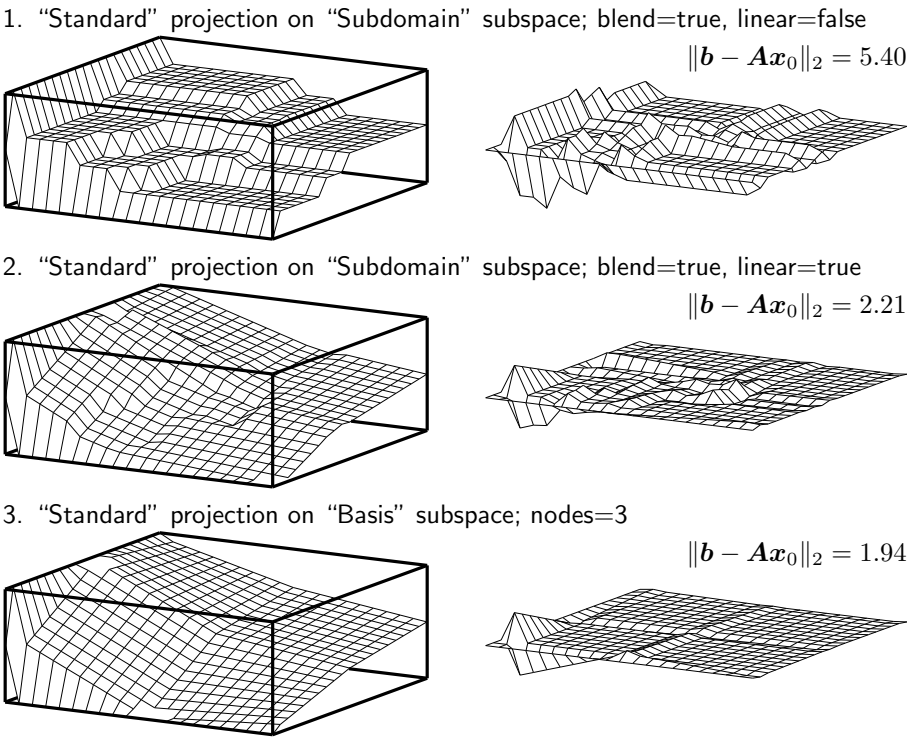

Figure 4.3. *Visualization of the projected solutions (left) and the corresponding residuals (right) for the three deflation subspaces that were shown in Section 3.3, Figures 3.3 and 3.4, with dimensions 4, 12 and 9, respectively. Comparing the first and second row, the figure shows that linear subdomain vectors bring the projected solution much closer to the exact solution shown in Figure 4.1. This is reflected by the smaller residual on the right. The "Basis" subspace comes slightly closer, and does so with greater efficiency as its dimension is down by three.*

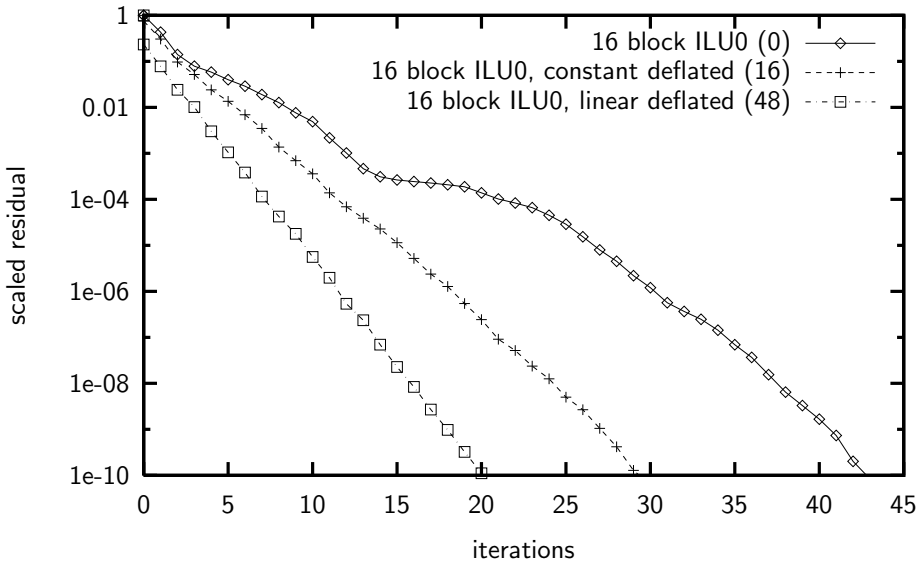

Figure 4.4. *GMRES convergence of the test problem on a* 21×21 *grid, comparing two variants of "Subdomain" deflation with standard 16 block ILU0 preconditioning. The displayed variants are marked with an asterisk in Table 4.1 below.*

	no deflation			"Subdomain" deflation											
				linear=false						linear=true					
				blend=false			blend=true			blend=false			blend=true		
	(0, 0, 0*)			(1, 4, 16)			(1, 4, 16*)			(3, 12, 48)			(3, 12, 48*)		
10^{-1}	2	2	3	2	2	3	2	2	2	1	2	1	1	1	1
10^{-2}	5	7	9	5	6	7	5	6	6	3	4	4	3	3	4
10^{-3}	7	10	13	7	11	10	7	10	9	6	7	6	6	6	6
10^{-4}	12	18	22	11	15	14	11	14	12	7	10	8	7	9	8
10^{-5}	15	22	27	14	18	18	14	17	16	10	12	10	10	11	10
10^{-6}	17	25	31	16	21	21	16	20	19	12	14	12	12	14	12
10^{-7}	19	28	35	18	24	24	18	22	21	14	16	14	14	16	14
10^{-8}	20	30	38	19	27	26	19	25	25	16	18	17	16	18	16
10^{-9}	22	32	41	21	29	30	21	28	28	17	21	19	17	20	18
10^{-10}	23	34	43	22	31	33	22	30	30	19	23	21	19	22	21

Table 4.1. *Number of iterations required to reach various levels of accuracy (the first column) measured as the residual over right hand side ratio. The first block of three columns show the number of iterations required by 1, 4 and 16 block ILU0, respectively, as shown in Figure 4.2. The next four blocks compare the same three preconditioners combined with various variants of "Subdomain" deflation, the subspace dimensions shown in braces.*

deflation vectors overlap in shared regions — have a small but steady advantage over the strictly disjoint variants, a single three iterations difference being the worse case. Henceforth concentrating on blended subspaces, Figure 4.4 graphically displays some of the data in Table 4.1, comparing convergence of constant and linear deflation in a 16 subdomain situation against non-deflated convergence.

Figure 4.4 shows that both variants of deflation yield almost perfect linear convergence, compared to the non deflated process which suffers from periods of relatively slow progress such as in iterations 13 to 20. These periods are typically caused by tightly clustered eigenvalues that the Krylov process has trouble finding. It seems that the deflation preconditioner has been able to favourably change these parts of the spectrum. The same observation can be made in Figure 7.1 of Saad et al. [16], which shows the results of a similar experiment except that the deflation subspace is formed of up to three eigenvectors. Note that this is the ideal situation; Section 2.4 explained that this subspace effectively deletes the smallest few eigenvalues of the spectrum. Although less apparent due to the relatively low dimension of the deflation subspace, the figure shows the same linear convergence and absence of slowly convergent periods as found here, especially in the first 30 iterations.

The "Subdomain" type subspaces are expected to approximate the same eigenvectors that were constructed explicitly by Saad et al. [16], while using less work and maintaining sparsity — although the latter is not exploited by the current solver. The linear deflation vectors were proposed by Verkaik [18] as an enrichment in order to better achieve this goal. Figure 5.10 in this thesis shows some convergence results for Laplace, using both CG and GCR. Note that the convergence of the two is qualitatively the same, which agrees with the assumption that convergence is governed by the common Krylov subspace. This allows for comparison with Figure 4.4 as well. Although the smaller grid and the 25 subdomains make the effects of deflation relatively larger, qualitative the same linear convergence is observed throughout the process for both constant (CD) and linear (CLD) deflation, the latter converging faster in terms of iterations.

Note that in practice not iterations but time is of essence. In order to choose either for or against deflation, or between different deflation subspaces, startup costs and work per iteration will need to be taken into account. The linear subspace of Figure 4.4, for example, is thrice as large as the constant subspace, which means that the work for each projection will triple in work, increasing the work per iteration. Moreover, the 'small' matrix that is inverted at the start of the iterative process will triple in size as well. In extremity this matrix can grow to the size of the original matrix, in which case deflation becomes a direct solution method; the required number of iterations will be zero. Depending largely on the quality of implementation, an optimum will lie somewhere between the two extremes where the two effects are in balance, iteration decrease versus work increase, yielding fastest convergence in time.

In general a comparison based on iterations between deflation subspaces of equal dimension is reasonably fair. Except for subspace specific optimizations, both initializing work and work per iteration should be equal. This is interesting to keep in mind when examining the other deflation subspace offered by the solver, the "Basis" type. Table 4.2 again shows the required number of iterations for reaching various

levels of accuracy, using non restarted GMRES and the same three preconditioners as in Table 4.1. Figure 4.5 graphically represents an interesting subset of this data. Comparing this with Figure 4.4, it is immediately apparent that of the two subspaces of equal dimension, constant and basis-4 deflation, the latter converges much faster. In fact, it converges about as fast as linear deflation, using a subspace one third in size.

These findings correspond well with the original idea behind this "Basis" subspace type, which can be put as 'efficiency'. Because the deflation vectors represent the continuous functions defined in Equation 3.4, they span a subspace consisting of piecewise linear, continuous functions. Note that continuity refers here to the underlying functions; this property does not transfer over to the discrete domain. In the same way, "Subdomain" deflation vectors represent discontinuous functions with support only on certain subsections of the physical domain. Apart from piecewise linear, continuous functions, this subspace contains functions with discontinuities on the subdomains. As there is no reason to assume that the eigenvectors that are approximated will be discontinuous exactly on these boundaries, this subspace is actually too rich.

Although from an algorithmic point of view the "Basis" vectors seem superior over the "Subdomain" vectors, there are two practical reasons for using the latter. The first is that the sparsity of these vectors is ideal for parallel exploitation, because each processor can simply ignore all but the few vectors that have support on its subdomain. Besides reducing work, this also allows for highly efficient storage. Moreover, the sparsity of the deflation vectors can be used in the inversion of the small matrix which increases efficiency during initialization as well. Similar optimizations will be possible for "Basis" vectors, if only the subdomains correspond exactly with the rectangular blocks used in the construction of this subspace. The fact that these are not inherently the same hinders the optimization of this deflation type.

The second reason for using "Subdomain" deflation is that the subspace dimension follows the number of subdomains, making the method more or less insensitive to parallelism. This can be seen in Table 4.1, comparing the required iterations for one, four and sixteen subdomains. Block ILU0 deteriorates as the number of subdomains increases, but as this is compensated by the richer deflation subspace, the iterations remain more or less constant. Again for "Basis" deflation a similar result can be obtained by manually refining the grid on which these vectors are formed. Indeed, comparing one node on one block, two nodes on four blocks and three nodes on sixteen blocks, iterations are once again constant. The rigid structure of this grid, however, makes it much harder to closely follow the number of subdomains.

The claimed insensitivity to scaling is valid only within limits. Constant deflation on a single domain requires less iterations than on four or sixteen subdomains. At this point the ILU0 preconditioner deteriorates more quickly than deflation can compensate it. The reverse is true on the other side of the spectrum. Linear deflation can be seen to yield faster convergence on sixteen subdomains than on four. Apparently from some point onward deflation improves faster than block ILU0 worsens. This was to be expected, because it was noted before that an extremely rich deflation subspace turns the Krylov method into a direct method, in which

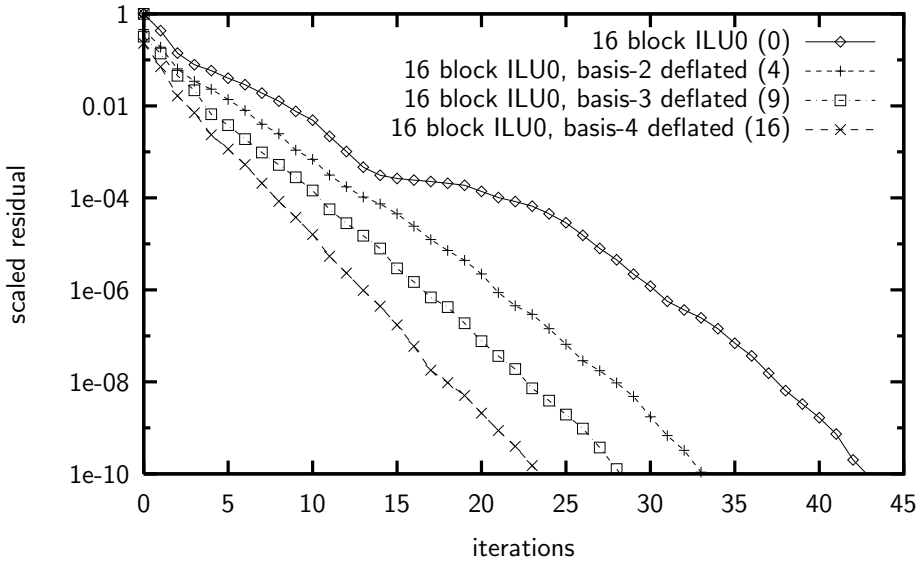

Figure 4.5. *GMRES convergence of the test problem on a* 21×21 *grid, comparing three variants of "Basis" deflation with standard 16 block ILU0 preconditioning. The displayed variants are marked with an asterisk in Table 4.2 below.*

	no deflation			"Basis" deflation											
				nodes=1			nodes=2			nodes=3			nodes=4		
	(0, 0, 0*)			(1, 1, 1)			(4, 4, 4*)			(9, 9, 9*)			(16, 16, 16*)		
10^{-1}	2	2	3	2	2	3	1	2	2	1	1	2	1	1	1
10^{-2}	5	7	9	5	7	9	3	5	6	2	3	4	2	2	3
10^{-3}	7	10	13	7	10	13	6	8	10	4	6	7	3	4	6
10^{-4}	12	18	22	11	13	17	7	11	14	5	9	11	4	7	8
10^{-5}	15	22	27	14	17	21	10	14	18	7	11	14	6	9	11
10^{-6}	17	25	31	16	23	29	12	17	21	9	13	17	7	11	13
10^{-7}	19	28	35	18	26	32	13	20	25	10	16	20	8	13	16
10^{-8}	20	30	38	19	28	35	15	23	28	12	19	23	9	15	18
10^{-9}	22	32	41	21	30	38	17	25	31	13	20	26	10	18	21
10^{-10}	23	34	43	22	33	41	19	28	34	15	23	29	12	19	24

Table 4.2. *Number of iterations required to reach various levels of accuracy (the first column) measured as the residual over right hand side ratio. The first block of three columns show the number of iterations required by 1, 4 and 16 block ILU0, respectively, as shown in Figure 4.2. The next four blocks compare the same three preconditioners combined with various variants of "Basis" deflation, the subspace dimensions shown in braces.*

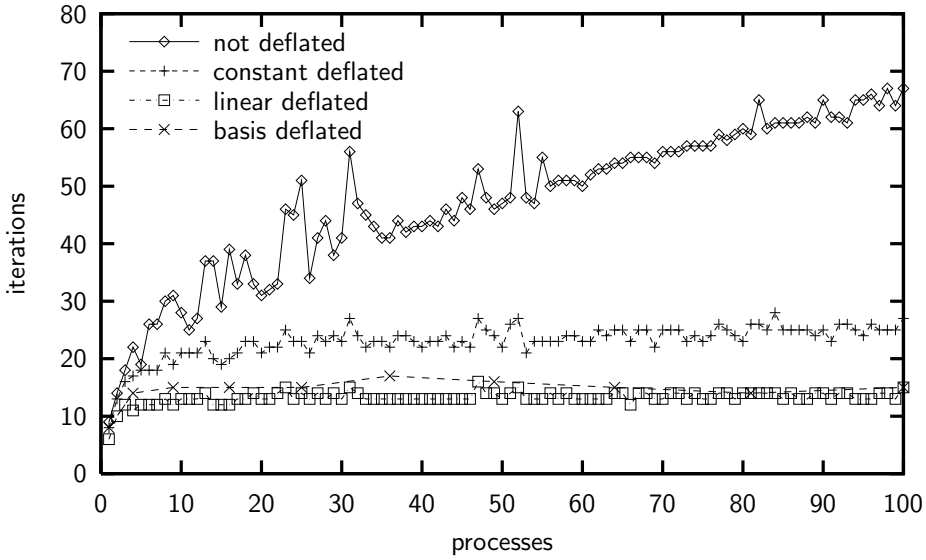

Figure 4.6. *Number of iterations required to reach a precision of 10^{-5}, solving the test problem Equation 4.1 on increasingly dense grids using various deflation variants. The grid density is connected to the number of processes such that all problems maintain a fixed ratio of 100 degrees of freedom per process. By fixing the dimension of the deflation subspace to the number of processes, all deflated methods become insensitive to parallel scaling.*

case the preconditioner has become irrelevant. The same is happening in "Basis" deflation because based on subspace dimension, the four nodes subspace should have been the last one to compare.

Because of this effect, true parallel scaling can only be investigated when the size of the problem is scaled along with the number of subdomains. This has been done in Figure 4.6, which shows the number of iterations required to reach a fixed precision for a wide range of subdomains, keeping the degrees of freedom per subdomain fixed at one hundred. For a normal, non deflated Krylov method this number increases, albeit sub linearly. For all tested variants of deflation it is practically constant from a number of subdomains onward; about twenty for constant deflation and about five for linear and basis deflation. The number of basis deflation vectors have been kept equal to the number of subdomains, which explains why data points exists only for square numbers. These few points, however, coincide very well with linear deflation, which once again confirms that the discontinuities in linear deflation are completely superfluous.

Figure 4.6 corresponds very well with Figure 6.1 of Frank and Vuik [4], which shows the results of a similar experiment. Here, too, the number of iterations if found to be (bounded) independent of the number of subdomains. With this, all reference articles have confirmed that the current solver behaves as expected in well known situations. It will be assumed henceforth that the deflation method has been implemented correctly, which means that investigations can continue in directions for which no reference data is yet available.

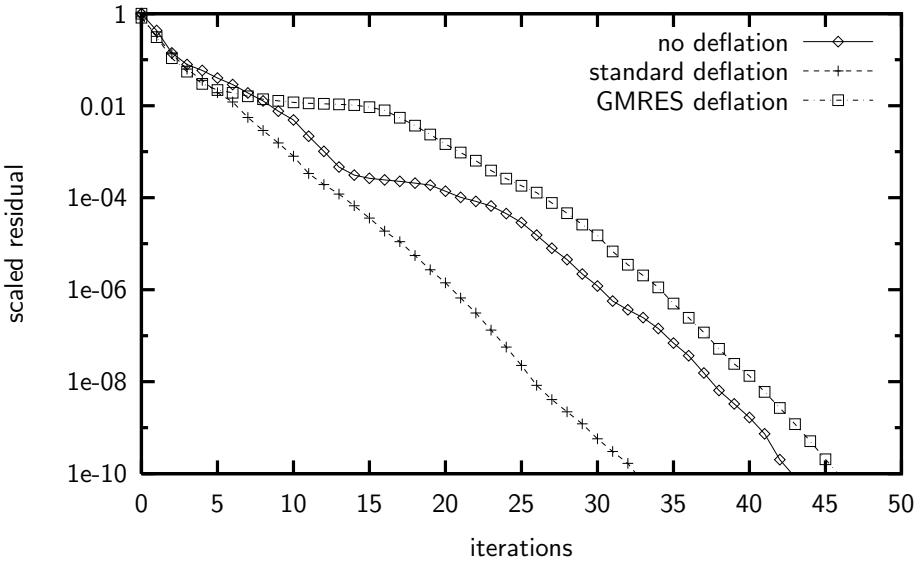

Figure 4.7. *GMRES convergence of the test problem om a* 21×21 *grid, comparing "Standard" and "GMRES" projected constant subdomain deflation, combined with 16 block ILU0 preconditioning. The "GMRES" deflated method performs very badly compared to the "Standard" deflated method, which is unexpected.*

4.1.2 Standard versus GMRES deflation

The results from the previous section are all based on "Standard" projection, which means that the deflation projection minimizes the distance to the deflation subspace measured in matrix $\boldsymbol{A}$-norm. For the Laplace test problem this is mathematically correct, as the matrix for this problem is SPD and therefore yields a valid norm. This is not true for general matrices; the resulting 'norm' can well be zero for non-zero vectors, or even negative. This problem can be solved by projecting in $\boldsymbol{A}^T\boldsymbol{A}$ instead, which is SPD for any matrix. This is exactly the same difference that underlies the CG and GMRES methods, of which only the latter is applicable to general matrices. By analogy, deflation based on $\boldsymbol{A}^T\boldsymbol{A}$ projection has been labeled "GMRES" deflation.

Although there is no mathematical need to use "GMRES" deflation for Laplace, being SPD, this method is still expected to yield similar convergence improvements as found before with "Standard" deflation. The only difference is the projection norm; from an algorithmic point of view, either should work. Figure 4.7, however, shows something quite different. Not only does the "GMRES" deflated method converge much slower than the equivalent "Standard" deflated method, convergence has even gotten worse than that of the original, non deflated method. From the point of view that deflation should at the very worst not contribute anything, deterioration is completely unsatisfactory.

In search of what exactly causes this bad convergence, the eigenvector spectra of the two deflated matrices have been compared. The deflation subspace used in

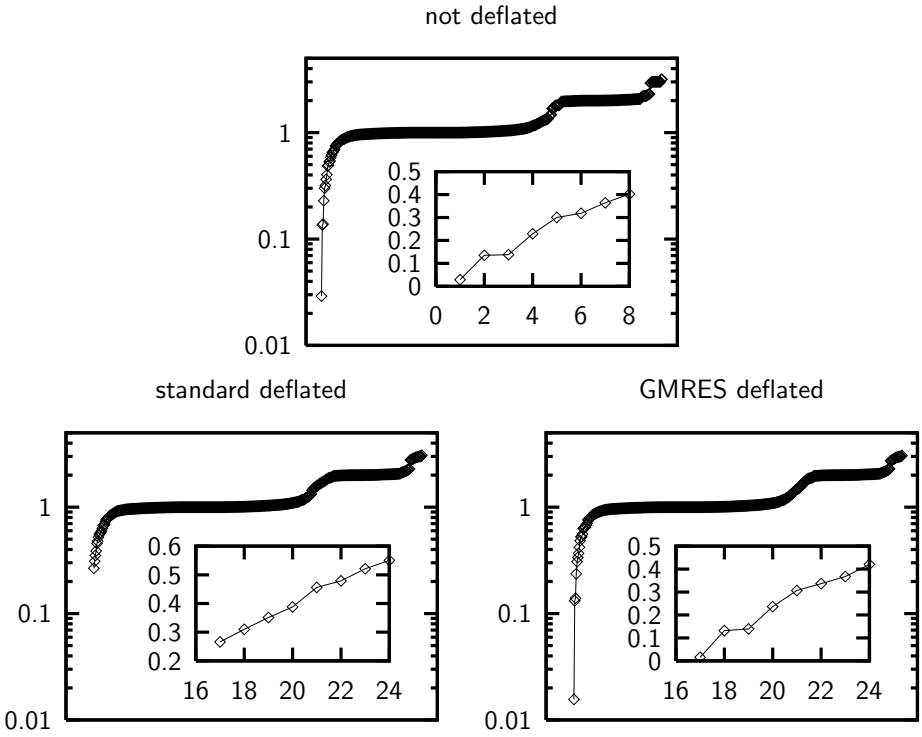

Figure 4.8. *Eigenvalue spectra of the preconditioned and deflated matrices used in Figure 4.7. The insets show the lower range of the spectrum, counting from 17 for both deflated matrices due to a 16 dimensional null space. Standard deflation is shown to be quite effective in removing the lower part of the spectrum, whereas GMRES deflation does not make any substantial changes.*

Figure 4.7 has dimension sixteen, which means that both deflated matrices should have a null space of equal dimension. One possible explanation of bad convergence could be that due to rounding errors the "GMRES" deflated matrix is not actually singular, instead has a few small but significant eigenvalues that actually hinder convergence instead of improve it. Such would be caused by an ill conditioned 'small matrix' $(\boldsymbol{Z}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{Z})$ inside the projection operator, which as a result would no longer qualify to be a projection at all. This possibility of a numerical cause is suggested by the well known property that for Laplace the product $\boldsymbol{A}^T\boldsymbol{A}$ is highly ill conditioned compared to the matrix $\boldsymbol{A}$. This has been confirmed for the problem used in Figure 4.7, in which case a factor 356 difference was found.

The spectra of the non deflated and both deflated matrices are shown in Figure 4.8. For both deflated matrices the sixteen smallest eigenvalues have been skipped; they all proved to be of order 10^{-16}, which discards the theory of a numerical problem. Indeed, the condition number of the small matrix inside "GMRES" projection was found to be only about twice as large as "Standard" projection, not by far enough to cause problems in the inversion. The remaining part of the spectrum, however, suggests that the nature of the problem is mathematical. While "Standard" deflation has effectively lowered the condition number by discarding the smallest eigenvalues

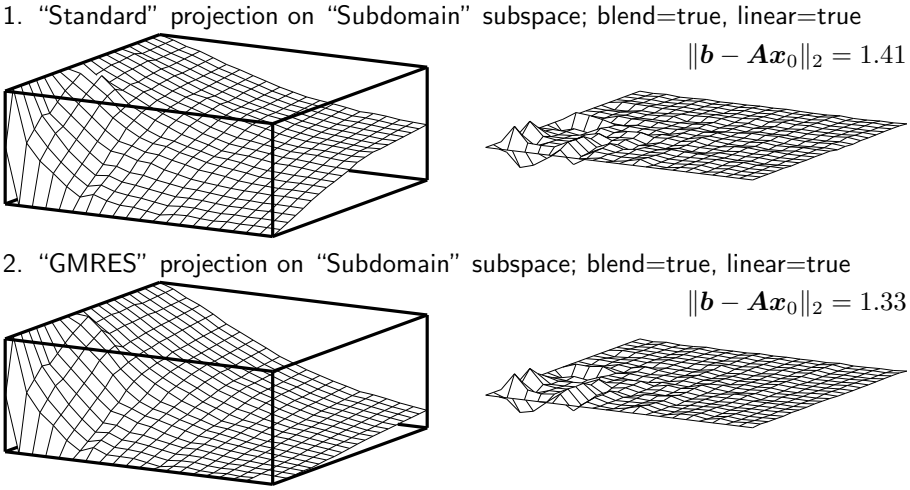

Figure 4.9. *Visualization of the projected solutions (left) and the corresponding residual (right) for twice the same 48 dimensional linear deflation subspace. The first row shows the result of "Standard" projection, the second that of "GMRES" projection; the former clearly has a smaller error, the latter a smaller residual.*

and leaving the rest of the spectrum more or less in place, "GMRES" deflation has shifted the remaining eigenvalues back to the position of the old extremes where they keep the condition number of the matrix at its old high value.

Although the observed difference in convergence behaviour can now be related to the different condition number of the two deflated matrices, an explanation of this difference has not yet been found. The ideas underlying both methods are the same, the only difference is the norm that measures the distance to the exact solution. To get insight in the qualitative differences of these two norms, Figure 4.9 compares the two in a projection of the exact solution onto a sixteen subdomain linear deflation subspace, similar to Figure 4.3. Immediately the qualitative difference shows: the point where the two Neumann boundary conditions meet is much too low for the "GMRES" projection, compared to the exact solution in Figure 4.1, whereas for "Standard" projection it is exactly where it should be: halfway. Although the former has a lower residual — by construction, the lowest possible — the latter clearly has the smallest error.

Figure 4.9 shows very clearly that minimum error and minimum residual do not coincide. This makes the projection in $\boldsymbol{A}^T\boldsymbol{A}$ norm very ineffective — for this problem at least — which seems to be a likely cause for the slow convergence observed in Figure 4.7. After all, Algorithm 2.3 showed that in each iteration the projected solution is used as the point from which the new search vector is constructed. When this projection is unnecessarily far off, the new search vector will lack accuracy as well and so the whole iterative process will take much longer. It is possible that the deflation subspace projection looses its accelerating effect and reduces to a merely improved initial guess; the spectrum of the deflated matrix will hence be the same, which indeed seems to be the case in Figure 4.8.

The question rises why the above problem does not show in GMRES, which is based on the same projection norm, when compared against CG. Two answers can be named. First, Theorem 2.2 states that the built Krylov subspace is independent of which projection norm is used. The projection therefore does not affect convergence, its only influence is in the creation of the iterates during convergence. Looking purely at the residual, such as is done in all convergence plots in this section, no problem shows. The method convergences fast, and the iterates formed by GMRES have slightly smaller residuals than those formed by CG. Figure 4.9 suggests that the latter will have a smaller error, but in practice there will be no way of checking this.

The second answer to why the GMRES method does not suffer from the same problem as "GMRES" deflation is that the $\boldsymbol{A}^T\boldsymbol{A}$ norm in Section 2.3.4 involved the preconditioned matrix. Therefore, instead of minimizing the residual, the projection actually minimizes the approximate error. Although the quality of the approximation depends on the quality of the preconditioner, the result can expected to be better than that shown in Figure 4.9. Quick tests have shown that deflation based on this norm does indeed converge faster, but the difference seems to be moderate. A reason for this can be that preconditioners — block preconditioners especially — in general leave the slowly varying eigenvectors of the matrix virtually unchanged, which suggests that it will not much effect the projection onto a subspace of a similarly slowly varying nature.

The disappointing results with "GMRES" deflation are a setback, because the matrix resulting from the VMS method introduced in Chapter 1 is asymmetric. This means that, theoretically at least, "Standard" deflation does no longer apply. The question now is if "Standard" deflation will still be able to aid convergence, despite this theory, and if "GMRES" deflation will keep failing.

4.2 The Variational Multiscale method

The previous section showed that the created deflation solver is capable of accelerating convergence for the discretized Laplace equation. It showed that the deflation subspaces can even be chosen such that in an increasingly parallel environment this acceleration balances the failing quality of the preconditioner, which makes the solution method insensitive to scaling. These findings correspond closely to various reference articles on this subject, which led to conclude that the deflation method has been implemented correctly. This means that attention can finally return to the reason that deflation was implemented in the first place: the Variational Multiscale (VMS) method.

The VMS method was introduced in Chapter 1. To recapitulate, this method allows for accurate solving of the system of non-linear Navier-Stokes equations by making effective use of a high order finite element basis. This method has been successfully implemented and has proved to be superior in many cases to competing methods — as far as flow accuracy is concerned. A serious problem, however, is that the linearized systems resulting from this method are very ill conditioned, and therefore demand a high quality preconditioner in order to get reasonable convergence. In

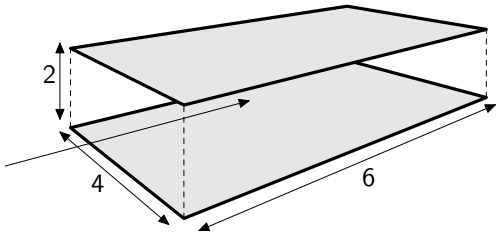

Figure 4.10. *Spatial domain of the VMS planer channel flow test problem. Periodic boundary conditions apply in streamwise (distance 6) and spanwise (distance 4) direction, thus simulating an infinite domain. No-slip and constant temperature are imposed on the walls (distance 2).*

parallel computing environments especially, due to the various restrictions that apply, this high quality preconditioner is not feasible which means that convergence in these situations is slow. Deflation has been suggested as a possible means of improving this situation.

The test problem that will be used for convergence experiments throughout this section is the **planar channel flow** problem; a widely used test case for turbulent flow research which assumes that flow is passing between two infinitely large parallel planes, driven by a constant pressure gradient. Numerically, the infinity is simulated by imposing periodic boundary conditions in streamwise and spanwise direction, over a region that is large enough to ensure that the turbulence is sufficiently decorrelated in both directions. The result is a rectangular domain of dimensions shown in Figure 4.10. In wall normal direction a no-slip condition is imposed, and the temperature at the walls is held constant. The total simulation involves five flow field variables: three velocity components, temperature, and density.

The planar channel flow problem will be solved using VMS, which means that a high order finite element basis needs to be defined on the domain shown in Figure 4.10. For this the three spatial dimensions are divided in eight cells, forming a total of 512 elements. A stretched mesh is used in wall-normal direction to increase resolution in the vicinity of the wall. On these elements a second order modal p-type expansion basis is formed; $P = 2$ in Equation 1.20, corresponding to two first order and one second order mode in each direction which brings the total number of basis function per element to $3^3 = 27$. For separation of scales this is the absolute minimum, but it is fine for numerical testing. After reduction due to continuity constraints and serendipity expansion, the total degrees of freedom of the resulting system is found to be 19712.

Note that the thus defined finite element basis is purely spatial. Planar channel flow is turbulent, hence non-stationary, which means that a time dependent simulation is required. Section 1.3 showed how the VMS method uses a fourth dimension for simulation in time, forming so called time-slabs. All elements can be though to be extruded by a certain amount in time. No separation of scales is required for this dimension, hence no high order expansion is required. A first order expansion suffices, which still doubles the number of basis functions per element. Since the jump condition enforces a value on half of these, however, the total degrees of

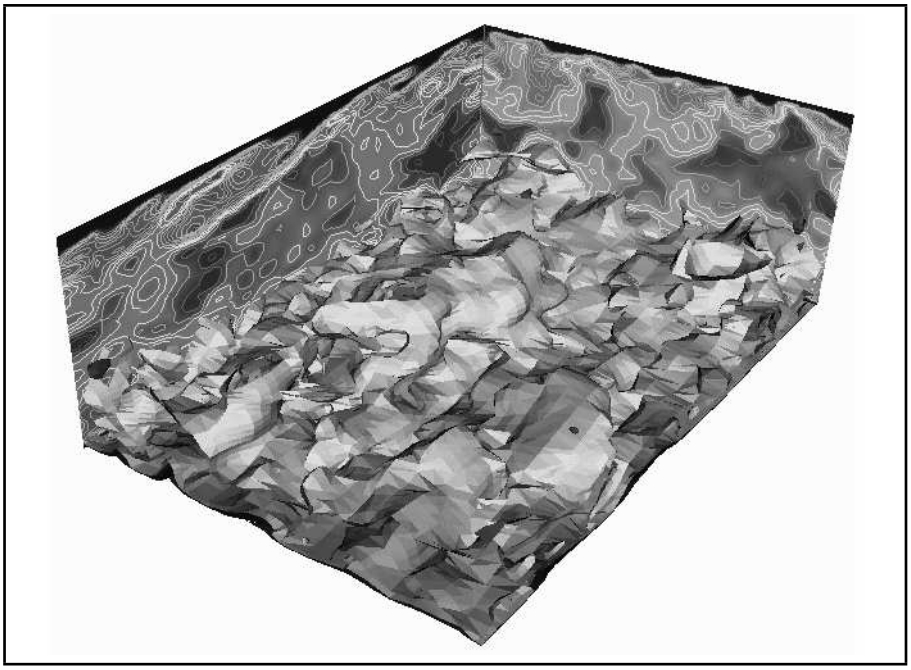

Figure 4.11. *Visualization of a three dimensional velocity field computed via VMS, showing only the streamwise component; velocity contours are shown both on the walls and as three dimensional surfaces within the domain. Copied from the Engineering Mechanics website.*

freedom does not change.

The planar channel flow problem does not have an exact solution in the same sense that the Laplace problem from the previous section had one. The simulated turbulent flow will depend heavily on the initial condition and small (numerical) perturbations, which means that quantitatively, the computed flow field at a certain point in time is not really meaningful. Instead, in practice the computed solution will be used to gather statistical data about the turbulent flow and the influence it has on its surroundings. Qualitatively, an image of the flow field can be useful, but being three dimensional it is complicated to visualize. A possible way is shown in Figure 4.11, which shows the result of a similar, though more accurate VMS simulation of planar channel flow. For deflation, however, it was not deemed useful to create similar images; analysis will be purely residual based.

4.2.1 Deflation results

To confirm that the current VMS method does indeed suffer from bad parallelization, Figure 4.12 shows the convergence of the non-deflated method decomposed in one, four and sixteen subdomains. The three stacked figures are three subsequent Newton iterations for the solution of one time slab. Recall that the VMS method

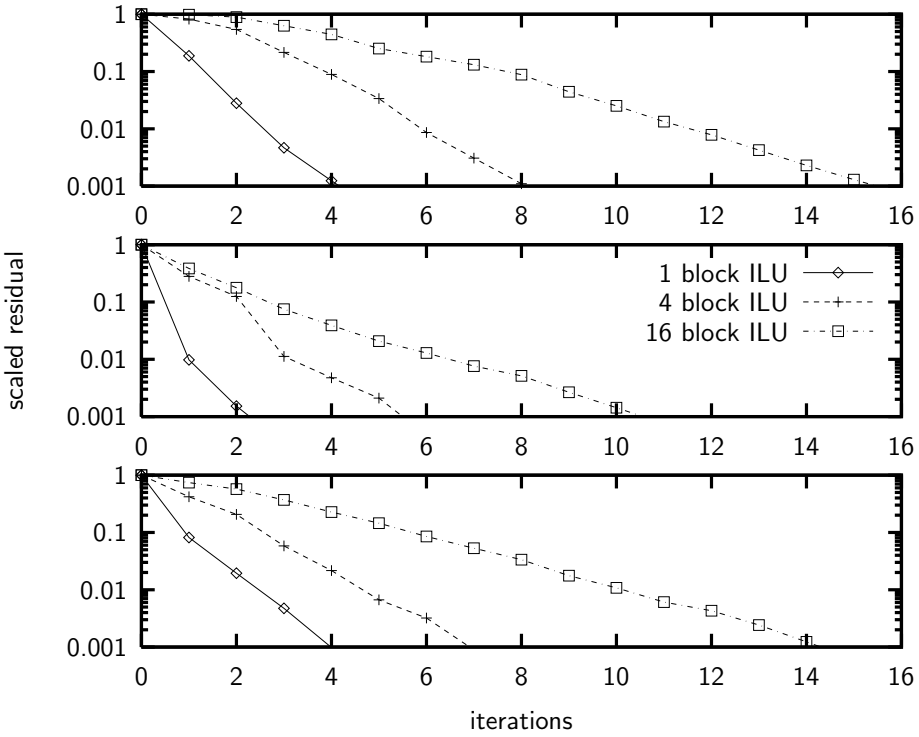

Figure 4.12. *GMRES convergence of the planar channel flow test problem, comparing 1, 4 and 16 block ILU preconditioning. The three stacked plots show convergence of the three consequent Newton iterations that are required for computing a single time slab solution. The plot shows that block ILU deteriorates quickly as the number of blocks increases.*

results in a system of non-linear equations, which are solved using Newton linearization. During these iterations the Jacobi matrix is not recreated; the performance gain can not outweigh the considerable amount of work required for forming this matrix. After three Newton iterations, each solved with precision 10^{-3}, the solution is considered accurate enough and the simulation proceeds to the next time slab. Even then, the Jacobi matrix is not recreated.

Figure 4.12 illustrates the experienced parallelization problems. Even more than observed with Laplace, the convergence speed drops drastically due to deterioration of the block ILU preconditioner. With sixteen subdomains, almost four times as many iterations are required compared to one domain, for all three Newton iterations. This, however, is still markedly better than a diagonal scaling preconditioner. An attempt to solve the system using this preconditioner had to be aborted when the first Newton iteration had not converged in 2000 iterations. This illustrates how ill conditioned the Jacobi matrix really is, and why it is so sensitive to the quality of the preconditioner.

In a first attempt to improve this situation, deflation has been tried against the worst case shown in Figure 4.12; that of sixteen subdomains. A constant subdomain

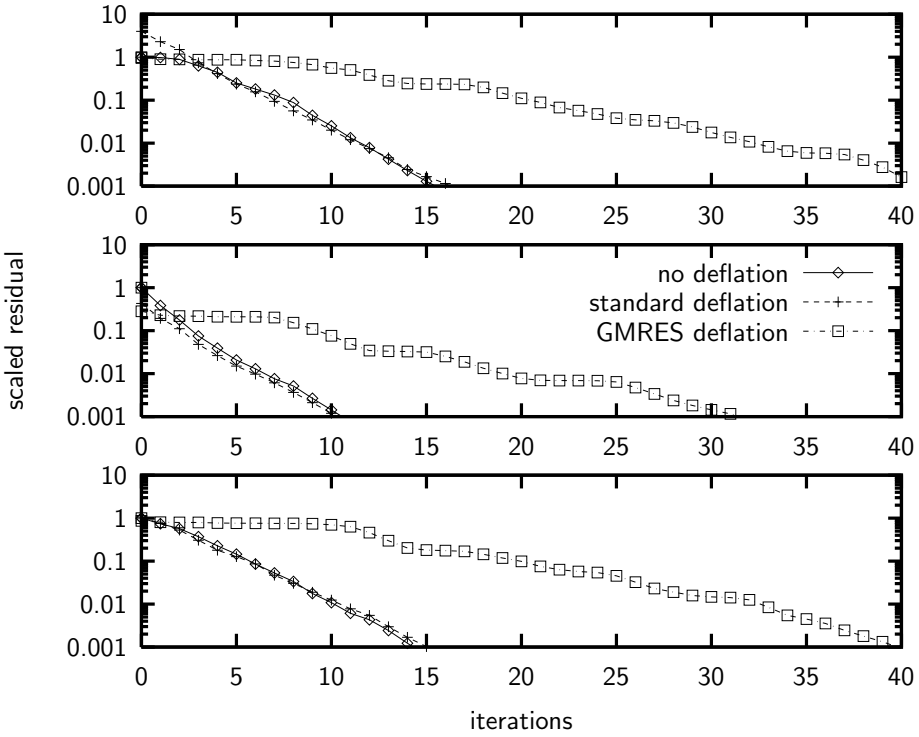

Figure 4.13. *GMRES convergence of the planer channel flow test problem, comparing "Standard" and "GMRES" projected constant subdomain deflation, combined with 16 block ILU preconditioning.*

subspace has been formed using only the linear basis functions in the expansion. Because separate deflation vectors are formed for all five computed flow field variables, the constant subdomain subspace has dimension $5 \cdot 16 = 80$. It would seem that a subspace of this size should be able to contribute something to the convergence of the iterative method. Unfortunately, Figure 4.13 shows otherwise. The figure compares convergence for both the "Standard" and "GMRES" variants of deflation, of which only the latter is theoretically sound due to the asymmetry of the Jacobi matrix — but neither shows improvement. On the contrary, "GMRES" deflation completely destroys convergence, even more than it did with the Laplace problem. "Standard" deflation does not help either, but at least the original convergence is maintained.

In an attempt to gain some insight in what is causing the observed behavior, the three subsequent Newton iterations shown in Figure 4.13 will be examined more closely.

1. In the first Newton iteration, "Standard" deflation appears to have a bad start. After 'improving' the initial condition (which is zero) with the projected solution, the scaled residual becomes nearly four times as large. However, after three Krylov iterations the deflated method has caught up with original convergence due to instant linear convergence. "GMRES" deflation maintains

the residual, but it does so for the first eight iterations long, after which improvement very reluctantly sets in.

2. The second Newton iteration is different from the first in that both "Standard" and "GMRES" deflation are able to bring down the initial residual. Only the former is able to maintain this gain throughout the iterative process, however small it may be. The latter looses it in two iterations, and again it will wait for another six iterations before convergence finally starts.

3. The third and last Newton iteration, neither of the two methods change the initial residual. In fact "Standard" deflation does not seem to affect anything at all as convergence is identical to that of the non-deflated method. "GMRES" deflation, once again, starts with a stationary period of ten iterations.

Because "GMRES" deflation explicitly minimizes the residual of the projected vectors, each newly formed iterate must have a residual that is as most as large as the previous one. Figure 4.13 shows that this condition is indeed satisfied. Unfortunately, the figure shows that the residual is very often exactly equal to the previous one, or only slightly smaller. The reason for this is not clear. The previous section suggested that the bad convergence of "GMRES" deflation may be caused by a weak connection between the residual and the actual error. In the first Newton iteration, although the "Standard" projection leads to a higher residual, the actual error may still be close to original one. The further convergence behaviour seems to support this. The other way round, the "GMRES" projection has only slightly changed the residual, but it may well have increased the error, with apparent consequences. Note that this is mere speculation; further investigations are clearly required. As things stand, however, "GMRES" deflation is not usable and will not be considered any further.

Although "Standard" deflation has lost its mathematical grounds since the asymmetric Jacobi matrix does not induce a valid norm, the results obtained with constant deflation still look interesting. Granted, the examined subspace does not contribute anything, but an even richer deflation subspace may be a able to make a difference. Table 4.14 shows convergence results for one, four and sixteen subdomains combined with different deflation subspaces, similar to Tables 4.1 and 4.2 in the previous section. Results for the last "Subdomain" variant are missing due to singularity of the small matrix that is part of the projection operator. The reason for this singularity has not been investigated because it does not seem to be related to the experienced convergence problems. Based on experience with Laplace, the "Basis" deflation types will provide comparable information.

The largest deflation subspace listed in Table 4.14 has dimension 320. Yet even this subspace does not lead to changed convergence, in either direction; two iterations difference is the largest difference, occurring once, and disfavouring deflation. Figure 4.3, which displays the results for sixteen subdomains, leads to the same conclusion. It must therefore be concluded that none of the deflation variants that are provided by the current solver are capable of aiding VMS convergence. Not even slightly.

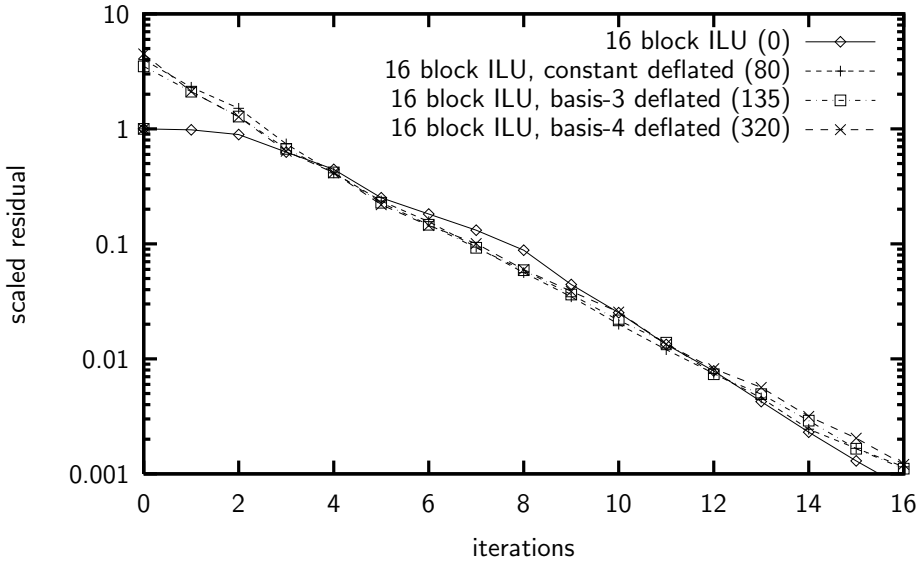

Figure 4.14. *GMRES convergence of the planar channel flow test problem, comparing three variants of deflation with 16 block ILU preconditioning. The displayed variants are marked with an asterisk in Table 4.14 below.*

	no deflation			"Subdomain" deflation blend=true linear=false (5, 20, 80*)			"Subdomain" deflation blend=true linear=true (20, 80,320)			"Basis" deflation nodes=3 (135,135,135*)			"Basis" deflation nodes=4 (320,320,320*)		
	(0, 0, 0*)														
10^{-1}	2	4	8	2	4	7	2	5	-	2	5	7	2	5	8
10^{-2}	3	6	12	3	6	12	3	7	-	3	7	12	3	7	12
10^{-3}	5	9	16	5	9	17	5	9	-	5	10	17	5	10	17
10^{-1}	1	3	3	1	2	3	1	2	-	1	2	2	1	2	2
10^{-2}	1	4	7	1	4	6	1	4	-	1	4	6	1	4	6
10^{-3}	3	6	11	3	6	11	3	6	-	3	6	11	3	6	11
10^{-1}	1	3	6	1	3	6	1	3	-	1	3	6	1	3	6
10^{-2}	3	5	11	3	6	11	3	6	-	3	6	11	3	6	11
10^{-3}	4	7	15	5	8	16	5	8	-	5	8	16	4	9	16

Table 4.3. *Number of iterations required to reach various levels of accuracy (the first column) measured as the residual over right hand side ratio. The three sets of rows show three subsequent Newton iterations. The first of three columns show the number of iterations required by 1, 4 and 16 block ILU preconditioners, respectively. The next four blocks compare the same three preconditioners combined with various variants of deflation, the subspace dimension shown in braces.*

Conclusions

Based on the wide range of numerical experiments that have been performed, the question whether deflation can increase the parallel efficiency of the Variational Multiscale (VMS) method will have to be answered negatively. None of the examined deflation subspaces have led to improved convergence of the linearized systems. However, in want of a clear explanation of these findings, no definitive judgement can be made. Deflation has been shown to be a close interplay between the deflation subspace and the projection norm, both of which can be varied with reasonable freedom. There may well be a combination of two that does aid VMS convergence, but this has to be a subspace other than those considered in this report, or a different projection, or both.

In order to construct a better deflation subspace or projection norm, it will first need to be clear why all current attempts have failed. Unfortunately, a definitive explanation has not been found. Both components were based on seemingly sound theory; the subspaces were constructed such that slowly varying eigenvectors can be approximated, the projection such that this approximation is accurate. Experiments on the Laplace equation, however, did suggest a high sensitivity to the latter. For this problem it has been shown that minimizing in $\boldsymbol{A}$-norm led to a much smaller error than minimizing in $\boldsymbol{A}^T\boldsymbol{A}$-norm, that would be applicable to general matrices. This lesser accuracy is likely to cause slower convergence, but whether this can explain the very large differences that have been found is questionable.

For the VMS method, the explanation of failed accuracy can be supported for both examined projections, for different reasons. The "Standard" projection is based on the $\boldsymbol{A}$-norm, which in the case of a VMS Jacobian matrix does not qualify to be a norm at all because it is not SPD. Failing the positivity property, the distance between a vector and its projection can be zero without being equal. This can not be good for accuracy. The "GMRES" projection based on the $\boldsymbol{A}^T\boldsymbol{A}$-norm was introduced for exactly this reason, because this norm is valid for general matrices. However, the Laplace results have shown that this norm does not constitute an accurate projection either, mainly because the minimum error and minimum residual do not coincide. This may still be the case for VMS; judging by the convergence data, it may be worse.

These above explanations are all mere speculation. To leave this speculative stage and to get a clear idea of what would make a fitting deflation preconditioner for the VMS method, further investigations will be required. In particular, the following

two questions require an answer:

1. Why does "GMRES" deflation perform as bad as it does, in general?

Although the suggested explanation of an inaccurate projection is plausible, it has one potential flaw: "GMRES" deflation has been used before, and with success. Deflation has been introduced in Section 2.4 as a natural extension of the Krylov subspace methods, starting from a predefined subspace instead of an empty one. Morgan [9] has designed a modified 'augmented' GMRES algorithm that stays very close to this idea, using a single projection for the entire search space. The deflation method used in this report, on the other hand, is based on a splitting of the the projection operator into a deflation projection and a Krylov projection, ow which the former has been changed into a preconditioner. It has been argued that the resulting methods relate in the same way as left and right preconditioned methods do, and that convergence behaviour of the two should be equal.

Unfortunately, augmented GMRES has not been tested on Laplace, which makes direct comparison impossible. Instead it has been applied to a number of specifically designed problems. Furthermore, the augmented GMRES method has not been used with constant or linear deflation subspaces, but with Ritz vectors that are obtained from previous GMRES iterations before restarting. Both the test problems and the Ritz vectors can be used in the current Jem/Jive solver, however, and it will be very interesting to see if this way the results found by Morgan can be reproduced. In that case, this suggests that the "GMRES" projection performs well in combination with a Ritz subspace, which would be really interesting. If on the other hand the results can not be reproduced, this would mean that the reasoning behind the equivalence of the two is flawed.

It will also be interesting to reverse the experiment, and try to reproduce the Laplace results from Section 4.1 using the augmented GMRES method. Although this method has been used solely in combination with Ritz vectors, to aid convergence after restarting, there is no reason why it can not be used with other subspaces as well. Again the results are expected to be equal — this is the reason why focus has been completely on preconditioned deflation. However, if for some reason they are not equal, augmented GMRES will be interesting for further investigation with VMS because of its applicability for general matrices.

2. Why do all variants of deflation fail when used in VMS?

An important question here is if the problem is caused by the deflation subspace, the projection operator, or both. To answer this question, a useful experiment will be to compute the exact eigenvectors for the lower part of the spectrum, and use these to form a deflation subspace. If convergence improves then the subspace is at least part of the problem; if it does not, the projection is to blame. The quality of the projection can be tested by comparing it against a projection in Euclidean norm, that is, directly minimizing the error. Note that this projection will necessarily involve solving the complete system. If the resulting error is much smaller, this indicates an accuracy problem which means that the projection should be improved.

Starting with the projection norm, an obvious attempt of improvement that has

already been mentioned in the discussion of the Laplace results is to use the preconditioned matrix in "GMRES" projection. Instead of minimizing the residual, the projection will then minimize the approximate error. Depending on the quality of the preconditioner this can be expected to bring some improvement. Another possibility worth examining is to use the residuals of the five simulated quantities separately (three speed components, temperature, and density) and define a projection that somehow minimizes these. This will have to be within the restriction that the unknown solution can be projected by making use of the right hand side.

The deflation subspace can fortunately be constructed more freely. When the exact eigenvectors prove to be useful, it may be worthwhile to visualize these vectors and try to extract qualitative information. Exact computation of the eigenvectors is not feasible, but it may be possible to cheaply construct a set of vectors that span approximately the same subspace. A similar approach has been followed by Vuik et al. [19], who have constructed a deflation subspace based on an inferred relation between eigenvectors and earth layers. Similar relations may be possible for VMS.

Another way of staying close to the real eigenvectors has already been suggested in question one. After a number of GMRES iterations, Ritz vectors can be formed that approximate precisely those eigenvectors that are interesting for deflation. This may not seem very interesting at first as these vectors will be available only after the solution has been found; restarting should not be necessary. Note, however, that the Jacobian matrix will not be changed very often, not even for different time-slabs. This means that when a set of approximate eigenvectors has been computed, they can be used for many iterations that follow until a new Jacobian matrix is formed — and one GMRES procedure later the new set of Ritz vectors will be available. Of all the suggestions made in this section, this Ritz subspace is probably the most promising idea.

Appendix A

Deflation source code

A.1 DeflationSolver.h

```
1  #ifndef JIVE_SOLVER_DEFLATIONSOLVER_H
2  #define JIVE_SOLVER_DEFLATIONSOLVER_H
3
4  #include <jive/solver/Preconditioner.h>
5  #include <jive/solver/IterativeSolver.h>
6
7  JIVE_BEGIN_PACKAGE( solver )
8
9  //  DeflationPrecon class
```

Matrix that behaves as the preconditioner $\boldsymbol{P} = (\boldsymbol{I} - \mathbb{P}_{\mathcal{Z}})\boldsymbol{M}^{-1}$ in Equation 3.3, plus additional functions for preparing the right-hand side. Important private variables:

- *precon_: nested preconditioner $\boldsymbol{M}^{-1}$*
- *conmat_: constrained matrix $\boldsymbol{A}$*
- *coords_: normalized coordinates of all items*
- *subspace_: deflation vectors $\boldsymbol{Z}$*
- *projection_: projection vectors $(\boldsymbol{Y}^T A \boldsymbol{Z})^{-1}\boldsymbol{Y}^T$ (see Section 3.3.2)*

The class does not provide the usual makeNew / declare pair because the preconditioner is not suitable for standalone use; it will be used exclusively by the DeflationSolver. The preconditioner will hence not be registered at a factory.

```
10
11 class DeflationPrecon : public Preconditioner
12 {
13  public:
14
15   JEM_DECLARE_CLASS( DeflationPrecon, Preconditioner );
16
17   typedef jem::mp::Context              Context;
```

```
18  typedef jive::mp::VectorExchanger  VectorExchanger;
19  typedef jive::util::DofSpace       DofSpace;
20
21                                     DeflationPrecon
22
23    ( const String&                    name,
24      Ref<AbstractMatrix>              matrix,
25      Ref<Constraints>                 cons,
26      Ref<VectorSpace>                 vspace,
27      Ref<Preconditioner>              precon );
28
29  virtual Shape                      shape ()                const;
30
31  virtual void                       configure
32
33    ( const Properties&                props );
34
35  virtual void                       getConfig
36
37    ( const Properties&                props )                const;
38
39  double                             getResidual
40
41    ( const Vector&                    lhs,
42      const Vector&                    rhs   )                const;
43
44  void                               improve
45
46    ( const Vector&                    lhs,
47      const Vector&                    rhs   )                const;
48
49  virtual void                       matmul
50
51    ( const Vector&                    lhs,
52      const Vector&                    rhs   )                const;
53
54  virtual void                       init  ();
55
56 protected:
57
58  void                               initSubspace_ ();
59
60  Matrix (DeflationPrecon::*getSubspace_)
61
62    ( const IntVector&                 idofs,
63      const IntVector&                 iitems )               const;
64
65  Matrix                             getEmptySubspace_
66
67    ( const IntVector&                 idofs,
68      const IntVector&                 iitems )               const;
69
70  Matrix                             getSubdomainSubspace_
71
```

```
72      ( const IntVector&          idofs ,
73        const IntVector&          iitems )          const;
74
75    Matrix                      getBasisSubspace_
76
77      ( const IntVector&          idofs ,
78        const IntVector&          iitems )          const;
79
80    void                        initProjection_   ();
81
82    Matrix (DeflationPrecon ::* getProjection_ )
83
84      ( )                                           const;
85
86    Matrix                      getStandardProjection_
87
88      ( )                                           const;
89
90    Matrix                      getGMRESProjection_
91
92      ( )                                           const;
93
94    void                        newValuesHandler_ ();
95
96    void                        newStructHandler_ ();
97
98    virtual                     ~DeflationPrecon  ();
99
100   private:
101
102   Ref<Preconditioner>         precon_;
103   Ref<ConstrainedMatrix>      conmat_;
104   Ref<VectorSpace>            vspace_;
105   Ref<VectorExchanger>        vex_;
106   Ref<DofSpace>               dofs_;
107   Ref<Context>                context_;
108   Matrix                      coords_;
109
110   Vector                      smallvec_;
111   Vector                      largevec_;
112
113   Matrix                      subspace_;
114   bool                        subspaceReady_;
115   String                      subspaceType_;
116   StringVector                subspaceDofs_;
117   struct
118   {
119     bool                        linear;
120     bool                        blend;
121     int                         cells;
122   }                           subspaceProps_;
123
124   Matrix                      projection_;
125   bool                        projectionReady_;
```

```
126   String                    projectionType_;
127 };
128
129 //  DeflationSolver class
```

Solver that uses a nested solver to solve a deflated system, as in Equation 3.3. Simply forwards all function calls to the nested solver except for solve and improve, which first call the deflation preconditioner's improve function. Importent private variables:

- *precon_: deflation preconditioner*
- *solver_: nested solver*

The solver has been designed to have no knowledge of the linear system it solves.

```
130
131 class DeflationSolver : public IterativeSolver
132 {
133  public:
134
135   JEM_DECLARE_CLASS( DeflationSolver, IterativeSolver );
136
137   typedef jem::io :: FileWriter  FileWriter;
138
139   static const char*        TYPE_NAME;
140   static const char*        SOLVER_PROP;
141   static const char*        PRECON_PROP;
142
143                             DeflationSolver
144
145     ( const String&           name,
146       Ref<Solver>             solver,
147       Ref<DeflationPrecon>    precon );
148
149   virtual void              configure
150
151     ( const Properties&       props );
152
153   virtual void              getConfig
154
155     ( const Properties&       props )          const;
156
157   virtual void              solve
158
159     ( const Vector&           lhs,
160       const Vector&           rhs );
161
162   virtual void              improve
163
164     ( const Vector&           lhs,
165       const Vector&           rhs );
166
167   virtual void              setMaxIterCount
168
```

```
169       ( int                       n );
170
171     virtual int                   getMaxIterCount   () const;
172
173     virtual void                  setMode
174
175       ( int                       mode );
176
177     virtual int                   getMode           () const;
178
179     virtual void                  setPrecision
180
181       ( double                    eps );
182
183     virtual double                getPrecision      () const;
184
185     static Ref<Solver>            makeNew
186
187       ( const String&             name,
188         const Properties&         conf,
189         const Properties&         props,
190         const Properties&         params,
191         const Properties&         globdat );
192
193     static void                   declare           ();
194
195   protected:
196
197     virtual                      ~DeflationSolver   ();
198
199     void                          nextIterHandler_
200
201       ( int                       iiter ,
202         double                    err );
203
204     void                          restartHandler_
205
206       ( int                       iiter ,
207         double                    err );
208
209   private:
210
211     Ref<DeflationPrecon>          precon_;
212     Ref< IterativeSolver >        solver_ ;
213
214     Ref<FileWriter>               plotFile_ ;
215     String                        plotFilename_;
216     double                        rscale_ ;
217   };
218
219   JIVE_END_PACKAGE( solver )
220
221   #endif
```

A.2 DeflationSolver.cpp

```
222 #include <jem/base/assert.h>
223 #include <jem/base/System.h>
224 #include <jem/base/ClassTemplate.h>
225 #include <jem/base/Array.h>
226 #include <jem/base/Float.h>
227 #include <jem/base/IllegalArgumentException.h>
228 #include <jem/io/PrintWriter.h>
229 #include <jem/io/FileWriter.h>
230 #include <jem/util/Event.h>
231 #include <jem/util/Properties.h>
232 #include <jem/mp/Context.h>
233 #include <jem/mp/UniContext.h>
234 #include <jem/mp/Buffer.h>
235 #include <jem/numeric/algebra/LUSolver.h>
236 #include <jem/numeric/algebra/matmul.h>
237 #include <jem/numeric/algebra/utilities.h>
238
239 #include <jive/fem/NodeSet.h>
240 #include <jive/util/ utilities .h>
241 #include <jive/util/Constraints.h>
242 #include <jive/util/DofSpace.h>
243 #include <jive/util/PointSet.h>
244 #include <jive/mp/VectorExchanger.h>
245 #include <jive/mp/XBorderSet.h>
246 #include <jive/algebra/VectorSpace.h>
247 #include <jive/algebra/MPMatrixObject.h>
248 #include <jive/solver/ConstrainedMatrix.h>
249 #include <jive/solver/SolverFactory.h>
250 #include <jive/solver/SolverParams.h>
251 #include <jive/solver/GenericConstrainer.h>
252 #include <jive/solver/ DeflationSolver .h>
253
254 JEM_DEFINE_CLASS( jive::solver::DeflationSolver );
255 JEM_DEFINE_CLASS( jive::solver::DeflationPrecon );
256
257 JIVE_BEGIN_PACKAGE( solver )
258
259 //  DeflationPrecon constructor
```

Initializes conmat_, dofs_, vspace_, precon_, vex_, context_, coords_, subspaceReady_ and projectionReady_. The latter two are set to false to signify that subspace_ and projection_ are uninitialized, respectively. Also, connects two events from the nested preconditioner to local handers.

```
260
261 DeflationPrecon :: DeflationPrecon
262
263   ( const String&           name,
264     Ref<AbstractMatrix>     matrix,
265     Ref<Constraints>        cons,
266     Ref<VectorSpace>        vspace,
267     Ref<Preconditioner>     precon ) :
268
```

```
269     Super( name )
270
271   {
272     using jem::mp::UniContext;
273     using jem::mp::RecvBuffer;
274     using jem::mp::SendBuffer;
275     using jem::mp::MIN;
276     using jem::mp::MAX;
277
278     using  jive :: algebra :: MPMatrixObj;
279     using  jive :: mp::newXBorderSet;
280     using  jive :: mp::SEND_RECV_BORDERS;
281     using  jive :: util :: ItemSet;
282     using  jive :: util :: PointSet;
283     using  jive :: util :: GroupSet;
284
285     jem::System::info () << myName_
286       << " : Initializing deflation preconditioner ...\n";
287
288     JEM_PRECHECK( matrix != jem::NIL );
289     JEM_PRECHECK( cons != jem::NIL );
290     JEM_PRECHECK( vspace != jem::NIL );
291     JEM_PRECHECK( precon != jem::NIL );
292
293     conmat_ = jem::newInstance<ConstrainedMatrix>( matrix, cons );
294     dofs_   = cons->getDofSpace();
295     vspace_ = vspace;
296     precon_ = precon;
297
298     Ref<MPMatrixObj> mpmatrix = jem::dynamicCast<MPMatrixObj>( matrix );
299     if ( mpmatrix != jem::NIL )
300     {
301       vex_ = mpmatrix->getExchanger();
302       context_ = vex_->getMPContext();
303       jem::System::info () << " * multi process: " << context_->size() << '\n';
304     }
305     else
306     {
307       jem::System::info () << " * single process\n";
308       context_ = jem::newInstance<UniContext>();
309       vex_ = jem::newInstance<VectorExchanger>(
310         context_,
311         dofs_,
312         newXBorderSet( SEND_RECV_BORDERS, dofs_->getItems() )
313       );
314     }
315
316     Ref<ItemSet> itemset = dofs_->getItems();
317     JEM_PRECHECK( itemset != jem::NIL );
318
319     Matrix trueCoords;
320     Ref<PointSet> pointset = jem::dynamicCast<PointSet>( itemset );
321     if ( pointset  != jem::NIL )
322     {
```

```
323       jem::System::info () << " * dofs are connected to nodes\n";
324
325       trueCoords.ref( pointset->toMatrix() );
326     }
327     else
328     {
329       jem::System::info () << " * dofs are connected to elements\n";
330
331       Ref<GroupSet> groupset = jem::dynamicCast<GroupSet>( itemset );
332       JEM_PRECHECK( groupset != jem::NIL );
333
334       itemset  = groupset->getGroupedItems();
335       pointset = jem::dynamicCast<PointSet>( itemset );
336       JEM_PRECHECK( pointset != jem::NIL );
337
338       Vector coord( pointset->rank() );
339       IntVector ipoints( groupset->maxGroupSize() );
340       trueCoords.resize( pointset->rank(), groupset->size() );
341       for ( int iitem = 0; iitem < groupset->size(); iitem++ )
342       {
343         int count = groupset->getGroupMembers( ipoints, iitem );
344         trueCoords( ALL, iitem ) = 0.0;
345         for ( int i = 0; i < count; i++ )
346         {
347           pointset->getPointCoords( coord, ipoints[ i ] );
348           trueCoords( ALL, iitem ) += coord / double( count );
349         }
350       }
351     }
352     JEM_PRECHECK( trueCoords.size( 1 ) == dofs_->itemCount() );
353
354     coords_.resize( trueCoords.shape() );
355     int dimCount = 0;
356
357     jem::System::info () << " * bounding box:";
358     for ( int idim = pointset->rank() - 1; idim >= 0; idim-- )
359     {
360       double xmin = jem::min( trueCoords( idim, ALL ) );
361       double xmax = jem::max( trueCoords( idim, ALL ) );
362
363       context_->allreduce( RecvBuffer( &xmin, 1 ), SendBuffer( &xmin, 1 ), MIN );
364       context_->allreduce( RecvBuffer( &xmax, 1 ), SendBuffer( &xmax, 1 ), MAX );
365
366       jem::System::info () << String::format( " %.1f:%.1f", xmin, xmax );
367       if ( ! jem::Float::isTiny( xmax - xmin ) )
368       {
369         coords_( dimCount, ALL ) = (trueCoords( idim, ALL )-xmin) / (xmax-xmin);
370         dimCount++;
371       }
372     }
373     jem::System::info () << '\n';
374     if ( dimCount < trueCoords.size( 0 ) )
375     {
376       jem::System::info () << " * discarded " << trueCoords.size( 0 ) - dimCount
```

```
377         << " empty dimensions, leaving " << dimCount << '\n';
378       coords_.reshape( dimCount, coords_.size( 1 ) );
379     }
380
381     subspaceReady_   = false;
382     projectionReady_ = false;
383
384     connect( precon_->newValuesEvent, this, &Self::newValuesHandler_ );
385     connect( precon_->newStructEvent, this, &Self::newStructHandler_  );
386
387     jem::System::info() << myName_
388       << " : Initialization complete.\n";
389   }
390
391   DeflationPrecon::~DeflationPrecon()
392
393   {}
394
395   //  newValuesHandler, newStructHandler
```

Event handlers, connected to corresponding events from the precon_ object. The handlers take no action other than re-emitting the event from this DeflationPrecon object, forming a bridge between the (private) nested preconditioner and the nested solver.

```
396
397   void DeflationPrecon::newValuesHandler_()
398
399   {
400     newValuesEvent.emit( *this );
401   }
402
403   void DeflationPrecon::newStructHandler_()
404
405   {
406     newStructEvent.emit( *this );
407   }
408
409   //  configure
```

Configures the deflation preconditioner based on two defation-specific run time properties:

- *subspace.type: "Empty" / "Subdomain" / "Basis"*
- *projection.type: "Standard" / "GMRES"*

At configuration time conmat_ will not be ready for use, which means necessary preparations can not be made. Instead the addresses of relevant functions are stored in getSubspace_ and getProjection_, postponing final preparations until first use.

```
410
411   void DeflationPrecon::configure ( const Properties& props )
412
413   {
414     Super::      configure( props );
```

```
415     precon_->configure( props );
416
417     Properties myProps = props.findProps( myName_ );
418
419     subspaceType_ = "Empty";
420     myProps.find( subspaceType_, "subspace.type" );
421     if ( subspaceType_ == "Empty" )
422     {
423       getSubspace_ = &getEmptySubspace_;
424     }
425     else if ( subspaceType_ == "Subdomain" )
426     {
427       getSubspace_ = &getSubdomainSubspace_;
428       subspaceProps_.linear = false;
429       myProps.find( subspaceProps_.linear, "subspace.linear" );
430       subspaceProps_.blend = false;
431       myProps.find( subspaceProps_.blend, "subspace.blend" );
432     }
433     else if ( subspaceType_ == "Basis" )
434     {
435       getSubspace_ = &getBasisSubspace_;
436       subspaceProps_.nodes = 2;
437       myProps.find( subspaceProps_.nodes, "subspace.nodes" );
438     }
439     else
440     {
441       throw jem::IllegalArgumentException( JEM_FUNC,
442         "invalid subspace type: " + subspaceType_ );
443     }
444
445     subspaceDofs_.ref( dofs_->getTypeNames() );
446     myProps.find( subspaceDofs_, "subspace.dofs" );
447
448     projectionType_ = "Standard";
449     myProps.find( projectionType_, "projection.type" );
450     if ( projectionType_ == "Standard" )
451     {
452       getProjection_ = &getStandardProjection_;
453     }
454     else if ( projectionType_ == "GMRES" )
455     {
456       getProjection_ = &getGMRESProjection_;
457     }
458     else
459     {
460       throw jem::IllegalArgumentException( JEM_FUNC,
461         "invalid projection type: " + projectionType_ );
462     }
463   }
464
465   //   getConfig

       Creates props based on configuration.

466
```

```
467 void DeflationPrecon::getConfig ( const Properties& props ) const
468
469 {
470   Super::     getConfig( props );
471   precon_->getConfig( props );
472
473   Properties myProps = props.makeProps( myName_ );
474
475   myProps.set( "subspace.type", subspaceType_ );
476   myProps.set( "subspace.dofs", subspaceDofs_ );
477   if ( subspaceType_ == "Subdomain" )
478   {
479     myProps.set( "subspace.linear", subspaceProps_.linear );
480     myProps.set( "subspace.blend", subspaceProps_.blend );
481   }
482   else if ( subspaceType_ == "Basis" )
483   {
484     myProps.set( "subspace.nodes", subspaceProps_.nodes );
485   }
486   myProps.set( "projection.type", projectionType_ );
487 }
488
489 //  shape
```

Returns the rows and columns of the preconditioner matrix as a two-element array.

```
490
491 DeflationPrecon::Shape DeflationPrecon::shape () const
492
493 {
494   return precon_->shape();
495 }
496
497 //  init
```

Initializes the preconditioner. Calls either initSubspace_ and initProjection_, initProjection_ only, or nothing, depending on subspaceReady_ and projectionReady_. After init both subspace_ and projection_ are initialized.

```
498
499 void DeflationPrecon:: init ()
500
501 {
502   if ( ! subspaceReady_ || ! projectionReady_ )
503   {
504     conmat_->update();
505
506     if ( ! subspaceReady_ )
507     {
508       initSubspace_();
509     }
510     if ( subspace_.size( 1 ) > 0 )
511     {
512       initProjection_ ();
513     }
```

```
514       largevec_ . resize ( subspace_. size ( 0 ) );
515       smallvec_ . resize ( subspace_. size ( 1 ) );
516
517       subspaceReady_  = true;
518       projectionReady_ = true;
519     }
520   }
521
522   //  initSubspace
```

Initializes subspace_. For each configured DOF type, calls the configured getSubspace_ function with indices and items. Non-empty vectors are added to subspace_.

```
523
524   void DeflationPrecon :: initSubspace_ ()
525
526   {
527     jem::System::info () << myName_
528       << " :  Initializing  '" << subspaceType_ << "' deflation subspace ...\n";
529
530     jem::System::info () << " * processes: "  << context_->size() << '\n';
531     jem::System::info () << " * dimensions: " << coords_.size( 0 ) << '\n';
532
533     IntVector  idofs  ( dofs_->dofCount() );
534     IntVector  iitems ( dofs_->dofCount() );
535
536     int vectorCount = 0;
537     for ( int iname = 0; iname < subspaceDofs_.size(); iname++ )
538     {
539       String name = subspaceDofs_[ iname ];
540       jem:: Slice compare;
541       if ( name.back() == '*' )
542       {
543         compare = jem::Slice( 0, name.size() - 1 );
544       }
545       else
546       {
547         compare = ALL;
548       }
549
550       int size = 0;
551       for ( int itype = 0; itype < dofs_->typeCount(); itype++ )
552       {
553         String name2 = dofs_->getTypeName( itype );
554         if ( name2[ compare ] == name[ compare ] )
555         {
556           jem:: Slice range( size , idofs . size () );
557           size += dofs_->getDofsForType( idofs[ range ], iitems [ range ], itype );
558         }
559       }
560
561       int added = 0;
562       if ( size > 0 )
563       {
```

```
564       Matrix vectors( (this->*getSubspace_)( idofs[ slice( 0, size ) ], iitems[ slice( 0, size ) ] ) );
565       subspace_.reshape( vspace_->size(), vectorCount + vectors.size( 1 ) );
566       for ( int icol = 0; icol < vectors.size( 1 ); icol++ )
567       {
568         Vector column( vectors( ALL, icol ) );
569         conmat_->initLhs( column, column );
570         double norm2 = vspace_->product( column, column );
571         if ( ! jem::Float::isTiny( norm2 ) )
572         {
573           subspace_( ALL, vectorCount ) = column;
574           vectorCount++;
575           added++;
576         }
577       }
578     }
579
580     int discarded = subspace_.size( 1 ) - vectorCount;
581     jem::System::info() << " * added " << added << " of "
582       << added + discarded << " type `" << name << "' vectors\n";
583   }
584   subspace_.reshape( vspace_->size(), vectorCount );
585
586   jem::System::info() << " * deflation vectors: " << vectorCount << '\n';
587
588   jem::System::info() << myName_
589     << " : Deflation subspace complete.\n";
590 }
591
592 //  getEmptySubspace
```

Constructs "Empty" subspace — literally.

```
593
594 Matrix DeflationPrecon::getEmptySubspace_
595
596   ( const IntVector&      idofs,
597     const IntVector&      iitems ) const
598
599 {
600   return Matrix();
601 }
602
603 //  getSubdomainSubspace
```

Constructs "Subdomain" subspace. Creates a set of subdomain-constant vectors by setting the column corresponding based on the rank to zero. Linear vectors are created similarly by copying the coords_ to a position based on the rank.

```
604
605 Matrix DeflationPrecon::getSubdomainSubspace_
606
607   ( const IntVector&      idofs,
608     const IntVector&      iitems ) const
609
610 {
611   int stride = 1;
```

```
612     Vector shift;
613     if ( subspaceProps_.linear )
614     {
615       shift.resize( coords_.size( 0 ) );
616       shift = 0.0;
617       for ( int i = 0; i < idofs.size(); i++ )
618       {
619         shift += coords_( ALL, iitems[ i ] );
620       }
621       shift /= double( idofs.size() );
622       stride += coords_.size( 0 );
623     }
624
625     int icol = context_->myRank() * stride;
626     Matrix vectors( vspace_->size(), context_->size() * stride );
627     vectors = 0.0;
628
629     for ( int i = 0; i < idofs.size(); i++ )
630     {
631       vectors( idofs[ i ], icol ) = 1.0;
632       if ( subspaceProps_.linear )
633       {
634         vectors( idofs[ i ], slice( icol+1, icol+stride ) ) =
635           coords_( ALL, iitems[ i ] ) - shift;
636       }
637     }
638
639     void (VectorExchanger::*exchange)( const Vector& vec );
640     if ( subspaceProps_.blend )
641     {
642       exchange = &VectorExchanger::scatter;
643     }
644     else
645     {
646       exchange = &VectorExchanger::exchange;
647     }
648
649     Vector weights( vspace_->size() );
650     weights = 1.0;
651     (*vex_.*exchange)( weights );
652
653     for ( int icol = 0; icol < vectors.size( 1 ); icol++ )
654     {
655       (*vex_.*exchange)( vectors( ALL, icol ) );
656       vectors( ALL, icol ) /= weights;
657     }
658
659     return vectors;
660   }
661
662   //  getBasisSubspace

        Constructs "Basis" subspace based on Equation 3.4.

663
```

```
664 int intpow( int base, int exponent )
665
666 {
667   int retval = 1;
668   while ( exponent )
669   {
670     retval *= base;
671     exponent --;
672   }
673   return retval;
674 }
675
676 Matrix DeflationPrecon::getBasisSubspace_
677
678   ( const IntVector&      idofs,
679     const IntVector&      iitems ) const
680
681 {
682   Matrix vectors( vspace_->size(), intpow( subspaceProps_.nodes, coords_.size( 0 ) ) );
683   vectors = 0.0;
684
685   for ( int i = 0; i < idofs.size(); i++ )
686   {
687     vectors( idofs[ i ], ALL ) = 1.0;
688     for ( int idim = 0, stride = 1;
689           idim < coords_.size( 0 );
690           idim++, stride *= subspaceProps_.nodes
691         )
692     {
693       double pos = ( subspaceProps_.nodes - 1 ) * coords_( idim, iitems[ i ] );
694       int    div = int( pos );
695       double mod = pos - div;
696       for ( int icol = 0; icol < vectors.size( 1 ); icol++ )
697       {
698         int node = ( icol / stride ) % subspaceProps_.nodes;
699         if ( node == div )
700         {
701           vectors( idofs[ i ], icol ) *= 1.0 - mod;
702         }
703         else if ( node == div + 1 )
704         {
705           vectors( idofs[ i ], icol ) *= mod;
706         }
707         else
708         {
709           vectors( idofs[ i ], icol ) = 0.0;
710         }
711       }
712     }
713   }
714
715   return vectors;
716 }
717
```

718 // **initProjection**

Constructs the projection_ vectors $(\boldsymbol{Y}^T\boldsymbol{A}\boldsymbol{Z})^{-1}\boldsymbol{Y}^T$*, with* $\boldsymbol{Y}$ *obtained from the configured getProjection_.*

719

```
720 void DeflationPrecon :: initProjection_ ()
721
722 {
723   jem::System::info () << myName_
724                        << " : Initializing '"
725                        << projectionType_
726                        << "' projection operator ...\n";
727
728    projection_ . resize ( subspace_.shape() );
729
730   Matrix vectors ( (this->*getProjection_)() );
731   Matrix coarsemat( subspace_. size ( 1 ), subspace_. size ( 1 ) );
732   Vector v( vspace_->size() );
733
734   for ( int i = 0; i < subspace_.size ( 1 ); i++ )
735   {
736     conmat_->matmul( v, subspace_( ALL, i ) );
737     vspace_->project( coarsemat( i, ALL ), v, vectors );
738   }
739
740   jem::System::info () << " * inverting coarse matrix\n";
741
742   double d;
743   if ( ! jem::numeric::LUSolver:: invert ( coarsemat, d ) )
744   {
745     throw jem::Exception( myName_, "singular deflation matrix" );
746   }
747
748   jem::numeric::matmul( projection_, vectors, coarsemat );
749
750   jem::System::info () << myName_
751                        << " : Projection operator complete.\n";
752 }
753
```

754 // **getStandardProjection**

Returns $\boldsymbol{Z}$ *for "Standard" projection.*

```
755
756 Matrix DeflationPrecon :: getStandardProjection_ () const
757
758 {
759   return subspace_;
760 }
761
```

762 // **getGMRESProjection**

Returns $\boldsymbol{A}\boldsymbol{Z}$ *for "GMRES" projection.*

```
763
764 Matrix DeflationPrecon :: getGMRESProjection_ () const
```

```
765
766 {
767   Matrix vectors( subspace_.shape() );
768
769   for ( int i = 0; i < subspace_.size( 1 ); i++ )
770   {
771     conmat_->matmul( vectors( ALL, i ), subspace_( ALL, i ) );
772   }
773
774   return vectors;
775 }
776
777 //   getResidual
```

Returns residual of lhs for given rhs. Not used by the preconditioner itself, only by the solver and only for user output. Defined here because the solver lacks conmat_.

```
778
779 double DeflationPrecon::getResidual
780
781   ( const Vector& lhs,
782     const Vector& rhs ) const
783
784 {
785   Vector u( lhs.size() );
786   Vector b( lhs.size() );
787   Vector r( lhs.size() );
788
789   conmat_->initLhs( u, lhs );
790   conmat_->getRhs( b, rhs );
791   conmat_->matmul( r, u );
792   jem::numeric::axpy( r, b, -1.0, r );
793
794   return sqrt( vspace_->product( r, r ) );
795 }
796
797 //   improve
```

Improves the initial lhs $\boldsymbol{x}_0$ by adding the projection $\mathbb{P}_{\mathbb{Z}}(\boldsymbol{x} - \tilde{x}_0)$

```
798
799 void DeflationPrecon::improve
800
801   ( const Vector& lhs,
802     const Vector& rhs ) const
803
804 {
805   JEM_PRECHECK( subspaceReady_ && projectionReady_ );
806
807   Vector u( lhs.size() );
808
809   conmat_->initLhs( u, lhs );
810
811   if ( subspace_.size( 1 ) > 0 )
812   {
```

```
813     conmat_->getRhs( largevec_, rhs );
814     vspace_->project( smallvec_, largevec_, projection_ );
815     jem::numeric::matmul( largevec_, subspace_, smallvec_ );
816     u += largevec_;
817   }
818
819   conmat_->getLhs( lhs, u );
820 }
821
822 //  matmul
```

Computes lhs $= (\boldsymbol{I} - \mathbb{P}_{\mathcal{Z}})\boldsymbol{M}^{-1}$ *rhs.*

```
823
824 void DeflationPrecon::matmul
825
826   ( const Vector&       lhs,
827     const Vector&       rhs   ) const
828
829 {
830   JEM_PRECHECK( subspaceReady_ && projectionReady_ );
831
832   precon_->matmul( lhs, rhs );
833
834   if ( subspace_.size( 1 ) > 0 )
835   {
836     conmat_->matmul( largevec_, lhs );
837     vspace_->project( smallvec_, largevec_, projection_ );
838     jem::numeric::matmul( largevec_, subspace_, smallvec_ );
839     lhs -= largevec_;
840   }
841 }
842
843 //  DeflationSolver constructor
```

Initializes solver_ and precon_, and connects two events from the nested solver to local handlers.

```
844
845 const char* DeflationSolver::TYPE_NAME = "Deflation";
846 const char* DeflationSolver::SOLVER_PROP = "solver";
847 const char* DeflationSolver::PRECON_PROP = "precon";
848
849 DeflationSolver::DeflationSolver
850
851   ( const String&       name,
852     Ref<Solver>         solver,
853     Ref<DeflationPrecon>  precon ) :
854
855     Super( name )
856
857 {
858   solver_ = jem::dynamicCast<IterativeSolver>( solver );
859   precon_ = precon;
860
861   JEM_PRECHECK( solver_ != jem::NIL );
```

```
862   JEM_PRECHECK( precon_ != jem::NIL );
863
864   connect( solver_->nextIterEvent, this, &Self::nextIterHandler_ );
865   connect( solver_->restartEvent,  this, &Self::restartHandler_  );
866 }
867
868 DeflationSolver::~DeflationSolver ()
869
870 {}
871
872 //  configure, getConfig
```

Configures the nested solver based on run-time properties.

```
873
874 void DeflationSolver::configure ( const Properties& props )
875
876 {
877   Super::     configure( props );
878   solver_->configure( props );
879
880   Properties myProps = props.findProps( myName_ );
881
882   plotFilename_ = "/dev/null";
883   myProps.find( plotFilename_, "plotfile" );
884   plotFile_ = jem::newInstance<FileWriter>( plotFilename_ );
885 }
886
887 void DeflationSolver::getConfig ( const Properties& props ) const
888
889 {
890   Super::     getConfig( props );
891   solver_->getConfig( props );
892
893   Properties myProps = props.makeProps( myName_ );
894
895   myProps.set( "plotfile", plotFilename_ );
896 }
897
898 //  improve
```

Solves the linear system by first calling the deflation preconditioner's and next the nested solver's improve function. All the rest is user feedback.

```
899
900 void DeflationSolver::improve
901
902   ( const Vector& lhs,
903     const Vector& rhs )
904
905 {
906   precon_->init();
907
908   jem::System::info() << myName_
909     << ": Solving for precision " << getPrecision() << " ...\n";
910
```

```
911     Vector nil( lhs.size() );
912     nil = 0.0;
913     double residual = precon_->getResidual( nil, rhs );
914     jem::System::info() << String::format( " * right hand side : %.3e\n", residual );
915     rscale_ = 1.0 / residual;
916
917     residual = precon_->getResidual( lhs, rhs );
918     jem::System::info() << String::format( " * initial residual : %.3e\n", residual );
919
920     *plotFile_ << "\n\n# NON DEFLATED\n";
921     *plotFile_ << 0 << ' ' << residual * rscale_ << '\n';
922
923     precon_->improve( lhs, rhs );
924
925     residual = precon_->getResidual( lhs, rhs );
926     jem::System::info() << String::format( " * after deflation : %.3e\n", residual );
927
928     *plotFile_ << "\n# START ITERATIONS\n";
929     *plotFile_ << 0 << ' ' << residual * rscale_ << '\n';
930
931     solver_->improve( lhs, rhs );
932
933     residual = precon_->getResidual( lhs, rhs );
934     jem::System::info() << String::format( " * after iterations : %.3e\n", residual );
935
936     jem::System::info() << myName_
937       << " : Done solving.\n";
938   }
939
940   //   solve
```

Same as improve, but starts improving from an empty vector instead of lhs.

```
941
942   void DeflationSolver::solve
943
944     ( const Vector& lhs,
945       const Vector& rhs )
946
947   {
948     lhs = 0.0;
949     improve( lhs, rhs );
950   }
951
952   //   nextIterHandler, restartHandler
```

Event handlers connected to corresponding events from the nested solver. Re-emitted after writing convergence data to the configured plotFile_.

```
953
954   void DeflationSolver::nextIterHandler_
955
956     ( int                   iiter,
957       double                resid )
958
959   {
```

```
960     * plotFile_ << iiter << ' ' << resid << '\n';
961     nextIterEvent.emit( iiter, resid, *this );
962   }
963
964   void DeflationSolver::restartHandler_
965
966     ( int                    iiter,
967       double                 resid )
968
969   {
970     * plotFile_ << "\n# RESTART ITERATIONS\n";
971     restartEvent.emit( iiter, resid * rscale_, *this );
972   }
973
974   //  bridge functions
```

Forwarded directly to the nested solver.

```
975
976   void DeflationSolver::setMaxIterCount ( int n )
977
978   {
979     solver_->setMaxIterCount( n );
980   }
981
982   int DeflationSolver::getMaxIterCount () const
983
984   {
985     return solver_->getMaxIterCount();
986   }
987
988   void DeflationSolver::setMode ( int mode )
989
990   {
991     solver_->setMode( mode );
992   }
993
994   int DeflationSolver::getMode () const
995
996   {
997     return solver_->getMode();
998   }
999
1000  void DeflationSolver::setPrecision ( double eps )
1001
1002  {
1003    solver_->setPrecision( eps );
1004  }
1005
1006  double DeflationSolver::getPrecision () const
1007
1008  {
1009    return solver_->getPrecision();
1010  }
1011
```

1012 // **makeNew**

Creates the deflation solver. First aqcuires the matrix, constraints and vector space from globdat and creates the nested preconditioner. Then uses these in the creation of the deflation preconditioner, in turn used in the creation of the nested solver. Both are used in the formation of the deflation solver.

```
1013
1014 Ref<Solver> DeflationSolver::makeNew
1015
1016   ( const String&      name,
1017     const Properties&  conf,
1018     const Properties&  props,
1019     const Properties&  params,
1020     const Properties&  globdat )
1021
1022 {
1023   Ref<AbstractMatrix> matrix;
1024   Ref<Constraints> cons;
1025   Ref<VectorSpace> vspace;
1026   Ref<Preconditioner> precon = newPrecon(
1027     jive::util::joinNames( name, PRECON_PROP ),
1028     conf,
1029     props,
1030     params,
1031     globdat
1032   );
1033
1034   params.get( matrix, SolverParams::MATRIX      );
1035   params.get( cons,   SolverParams::CONSTRAINTS );
1036   params.get( vspace, SolverParams::VECTOR_SPACE );
1037
1038   Ref<DeflationPrecon> deflation = jem::newInstance<DeflationPrecon>(
1039     name,
1040     matrix,
1041     cons,
1042     vspace,
1043     precon
1044   );
1045
1046   params.set( SolverParams::PRECON, deflation );
1047   Ref<Solver> solver = SolverFactory::newInstance(
1048     jive::util::joinNames( name, SOLVER_PROP ),
1049     conf,
1050     props,
1051     params,
1052     globdat
1053   );
1054
1055   return jem::newInstance<Self>(
1056     name,
1057     solver,
1058     deflation
1059   );
1060 }
```

```
1061
1062 //   declare
         Registers makeNew at the solver factory as type "Deflation"
1063
1064 void DeflationSolver :: declare  ()
1065 {
1066   SolverFactory :: declare (  TYPE_NAME, & makeNew );
1067   SolverFactory :: declare (  CLASS_NAME, & makeNew );
1068 }
1069
1070 JIVE_END_PACKAGE( solver )
```

Bibliography

[1] John D. Anderson, Jr. *Fundamentals of Aerodynamics*. McGrawl-Hill, New York, 2001.

[2] S. Scott Collis. The dg/vms method for unified turbulence simulation. In *32nd AIAA Fluid Dynamics Conference and Exhibit*, June 2002.

[3] V. Faber and T. Manteuffel. Necessary and sufficient conditions for the existence of a conjugate gradient method. *SIAM Journal on Matrix Analysis and Applications*, 21:356–362, 1984.

[4] J. Frank and C. Vuik. On the construction of deflation-based preconditioners. *SIAM Journal on Scientific Computing*, 23(2):442–462, 2000.

[5] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore and London, second edition, 1989.

[6] Thomas J.R. Hughes, Luca Mazzei, and Kenneth E. Jansen. Large eddy simulation and the variational multiscale method. *Computing and Visualization in Science*, vol. 3(no. 1/2):47–59, 2000.

[7] E.F. Kaasschieter. Preconditioned conjugate gradients for solving singular systems. *Journal of Computational and Applied Mathematics*, 24:265–275, 1988.

[8] George Em Karniadakis and Spencer J. Sherwin. *Spectral/hp element methods for CFD*. Oxford University Press, Oxford, 1999.

[9] Ronald B. Morgan. A restarted gmres method augmented with eigenvectors. *SIAM Journal on Matrix Analysis and Applications*, 16(4):1154–1171, 1995.

[10] The message passing interface (mpi) standard. Website: http://www-unix.mcs.anl.gov/mpi.

[11] E.A. Munts, S.J. Hulshoff, and R. de Borst. A space-time variational multiscale discretization for les. In *34th AIAA Fluid Dynamics Conference and Exhibit*, June 2004.

[12] R. Nabben and C. Vuik. A comparison of deflation and coarse grid correction applied to porous media flow. *SIAM Journal on Numerical Analysis*, 42(4):1631–1647, 2004.

[13] Stephen B. Pope. *Turbulent Flows*. Cambridge University Press, Cambridge, 2000.

[14] Lewis F. Richardson. The supply of energy from and to atmospheric eddies. *Proceeding of the Royal Society*, A97, 1920.

[15] Y. Saad, M. Yeung, J. Erhel, and F. Guyomarch. A deflated version of the conjugate gradient algorithm. *SIAM Journal on Scientific Computing*, 21(5):1909–1926, 2000.

[16] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, 2000.

[17] Barry F. Smith, Petter E. Bjørstad, and William Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, Cambridge, 1996.

[18] Jarno Verkaik. Deflated krylov-schwarz domain decomposition for the incompressible navier-stokes equations on a colocated grid. Master's thesis, Delft University of Technology, 2002.

[19] C. Vuik, A. Segal, and J.A. Meijerink. An efficient preconditioned cg method for the solution of a class of layered problems with extreme contrasts in the coefficients. *Journal of Computational Physics*, 152:385–403, 1999.

Index