A platform for numerical computations
with special application to preconditioning
A platform for numerical computations
with special application to preconditioning

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr. R.A. van Santen, voor een
commissie aangewezen door het College
voor Promoties in het openbaar te verdedigen
op donderdag 4 december 2003 om 16.00 uur

door

Wienand Daniël Drenth

geboren te Nijmegen
Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. R.M.M. Mattheij
en
prof.dr. H.A. van der Vorst

Copromotor:
dr. J.M.L. Maubach
In loving memory of my mother
Preface

It seemed that out of battle I escaped
Down some profound dull tunnel, long since scooped
Through granites which titanic wars had groined.

Yet also there encumbered sleepers groaned,
Too fast in thought or death to be bestirred.
Then, as I probed them, one sprang up, and stared
With piteous recognition in fixed eyes,
Lifting distressful hands, as if to bless.
And by his smile, I knew that sullen hall,
By his dead smile I knew we stood in Hell.

Wilfred Owen, Strange meeting

Research resulting in a dissertation is almost never a solitary business. Henceforth this particular one as well presents the results of research performed by several people. However, life itself is not always on the hand of doing research, and can turn lonely and dark. Sometimes events take such an unexpected and complicated turn that ”simple“ research related issues do not seem to matter anymore. This happened in April 2001 when my mother suddenly died. At that moment research and its importance became very relative to me. It took me pain and much time to find a proper direction forward, and pick up research again. And even more important, and perhaps at a higher cost, to find a new direction in life itself again. It is often said that time heals many wounds, but then time should be abundant. Writing these words this dissertation came to a good end. Though it is with mixed feelings, as the relativity remains.

The research described in this thesis was started in the summer of 1998. It formed part of a NWO project titled “Development of an interactive environment for numerical algorithms in large scale scientific computing“, and was carried out in cooperation with Utrecht University. The environment mentioned eventually became the NumLab workbench, or just NumLab. As the research progressed over the years the direction was shifted more toward the development of numerical algorithms. Notwithstanding this change, those algorithms were developed with the interactive environment in mind as eventual workbench.
The years I was working on the above mentioned project were also instructive on a wider field. Since I received my education at a non-technical university (Nijmegen), working at a technical university gave me the opportunity to learn and see other aspects of (mathematical) research. Looking back now, in the comfortable position of speaking about past events only, I considered it as a very interesting period. It certainly helps to widen ones horizon to taste a little on the various directions (theoretical and practical) that can be given in a particular research area. Moreover, a great variety of topics to concentrate on certainly helped to make work pleasant. Hence I found myself coding parts of NumLab on the one hand, but also working with pencil and paper to do mathematics on the other hand. But at other moments when I was teaching calculus to first-year students other challenges had to be faced.

Acknowledgments

As this thesis could not have been written without the help and collaboration of others, there are several people I would like to thank. First of all many thanks to my promoter prof.dr. R.M.M. Mattheij. Despite the difficulties encountered in the past years, there is much to his credit that brought this research and thesis to a good end. I must thank my supervisor dr. J.M.L. Maubach as well. His enthusiasm was unrivaled and the lengthy discussions we had on various research topics contributed greatly to the contents of this thesis. Finally, the comments made by prof.dr. H.A. van der Vorst from Utrecht University, and prof.dr.ir. J.J. van Wijk and prof.dr. W.H.A. Schilders from Eindhoven University on parts of my thesis were also helpful and contributed in improving its contents.

Work without colleagues is no real work at all, as these individuals contribute to a large extend to the habitat in which one lives and works. My former colleagues in the Scientific Computing Group did create a very agreeable and pleasant environment to live and work in. This being from an academic point of view, but certainly also from a human point of view. I hope all goes well for you in the years to come.

Finally I would like to thank my parents for their support in the past years to bolster both the intellectual and inner man. I think without them the challenge to write this thesis was a much harder undertaking. It fills me with sorrow only my father can witness the final steps. My mother, I hope you can see it all, wherever you are now.
## Contents

Preface v

1 Introduction 1
   1.1 Computational platforms 1
   1.2 Problem setting 3
   1.3 Outline 5

2 Direct methods: Efficient Gaussian elimination 7
   2.1 Introduction 7
   2.2 Substructuring 8
   2.3 An efficient node ordering for the refinement along a line 10
   2.4 Numerical examples and conclusions 18

3 Iterative methods and preconditioners 23
   3.1 Iterative solution methods 23
   3.2 Complexity of iterative methods 27
   3.3 Estimates for the spectral condition number 29
   3.4 Preconditioning 31
      3.4.1 Preconditioning methods 32
      3.4.2 Optimal order techniques for diffusive problems 33
      3.4.3 Incomplete factorization methods 34
   3.5 Approximate inverse preconditioning 34
      3.5.1 Frobenius norm minimization 35
      3.5.2 Factorized sparse approximate inverses 37
      3.5.3 Practical use of approximate inverses 38
   3.6 Diagonal approximate inverses 38

4 The approximate inverse 41
   4.1 The convection diffusion discretization 41
   4.2 Eigenvalues and eigenvectors of tridiagonal matrices 43
   4.3 The spectral condition number of Toeplitz matrices 45
   4.4 A preconditioner based on approximate inverses 47
   4.5 The spectral condition number related to a refined grid 55
## Contents

B Turing completeness 129  
B.1 Alphabets and language ...................................................... 129  
B.2 The Turing machine ........................................................... 129  
B.3 Primitive recursive functions ................................................. 132  
B.4 \(\mu\)-Recursive functions .................................................. 135

Bibliography 139

Index 147

Summary 149

Samenvatting 151

Curriculum Vitae 153
Chapter 1

Introduction

1.1 Computational platforms

The past few decades has witnessed a revolutionary increase in usage of computer systems in scientific research. As systems became more attractive with respect to price and performance, simulations run on computer platforms became a good alternative to real experiments. Typical application areas are civil engineering, medical sciences, hydrodynamics or aerodynamics.

Modern computer applications can accurately model complex problems with large amounts of parameters and variables, and generate large sets of computed data. These can have sizes of many gigabytes like in computational fluid dynamics simulations, as e.g. are met in weather forecasts. Since the performance of computers has drastically improved, it has now become relatively easy to run simulations and produce large datasets in a fairly short runtime.

One important aspect of analyzing the produced datasets and acquiring insight in the simulated problem is scientific visualization. A most common way is to plot the computed data, and examine this for further analysis. One step further is tracking, which gives more interaction with the simulation. At each (time) step of the simulation, the data produced is directly visualized, graphical or numerical, for inspection. This way the simulation can be monitored during runtime and the process can be stopped when the computed data is considered to be invalid. In order to avoid these invalid results, the whole process can be restarted after changing the input.

However, though tracking is a nice feature to control the simulation in a more interactive manner, this option is not sufficient when many input parameters are present. For example, when the simulation is to be run with different parameters it has to be halted, reconfigured and restarted. Interactive steering is a means to control the (visualization and simulation) parameters interactively and to overcome the problem of halting, reconfiguring and restarting. Parameters can be changed during the simulation itself, and the visual and numerical tracking offers immediate feedback.

One of the biggest problems of early simulation applications was the monolithic and specialized nature of the software. These applications were designed for a very restricted set of problems and did lack a certain amount of tracking and steering. In principle input and output data was read and written to file for further analysis. This
approach to modeling and simulating is very uneconomical and inflexible and has prompted the appearance of libraries for greater reusability of application software and application knowledge. However, a researcher is predominantly occupied with performing research in an often very complex and specialized environment, often having no application software at hand. Moreover, the researcher does not have the time to develop, implement and test a sophisticated software platform for the particular research area, not to mention the possibility that the researcher does not have the required programming skills. So, what is needed is a highly flexible, visual and easy-to-use software platform and programming environment that allows a ( inexperienced) researcher to reuse existing components in an easy way to run simulations. On top of this the researcher should be able to extend the existing platform and environment by newly developed simulation software for future use.

Building such custom tailored software by reusing existing components has been considered at several places. The most successful solutions come in the form of visual programming environments, such as AVS [97] or Iris Explorer [50]. But also packages that offer a whole range of specialized routines, such as Matlab [67] or Mathematica [107], are sophisticated tools for scientific applications. For such platforms there are several aspects that are important. First of all there have to be components. These are the reusable building bricks from which new applications can be made. Secondly there are control mechanisms that drive and steer the components that are put together in an application. Thirdly, a data exchange model is used to communicate and transfer between the various primitive units. Obviously, this model should be generic enough in order to communicate with non-platform software. Finally a user interface should be available that offers the researcher the possibility to track and steer the simulation.

In the future one may imagine that for an industrial problem, a computer will “read” a mathematical model and discretization from a book or paper and construct a software application for all required calculations. In this thesis we make a step into that direction. The software itself may have various representations, from classical source code line representation up to a modern interactive visual one. The latter representation facilitates rapid alterations and adaptations for research purposes and application development.

The need for such platforms is apparent for research groups who spend considerable time on the development of application software. A first step toward such a platform was made in [29, 61]; this platform was called Numlab. Numlab allowed a certain amount of flexibility as components of an application could be changed, and communicated with existing mathematical packages such as Matlab and Mathematica. However, this version of Numlab was based on an inhouse developed interpreter language which made the platform monolithic in some sense. One of the drawbacks was that this did not allow implementation of additional tools in an easy way. It can be compared to a library that needs to close down before a new book can be added: The book has to be placed on the proper shelf and the inventory list must be updated by hand. Then the library can reopen. In this thesis we present a concrete
and interactive visual workbench (Lab) for numerical computations (Num) and visualization: NumLab. It contains an application called network editor ([93]) and an existing C++ interpreter ([27]). Source code is created with a computer mouse and simple clicks: The user selects components from libraries (called modules) and connects inputs and outputs. In NumLab a new book can be added without the need to close down the library. The book is placed on the correct shelf and the inventory list is updated automatically.

1.2 Problem setting

Since NumLab is quite an ambitious project, we shall concentrate ourselves on a particular problem from real life. Our goal is to extend NumLab in such a way that we have all the necessary tools to tackle this problem. A typical problem of what is to be solved using the NumLab workbench is that of the moisture and salt ion transportation in a brick wall. The prediction of salt ion transport is important, because unbalanced salt concentrations damage bricks (see [11, 59, 62, 63, 80]).

Another problem is the shrinkage that occurs during drying. Moisture concentration gradients in the material and corresponding gradients in the amount of shrinkage will lead to drying stresses. Controlling these stresses is important, since they can lead to deformation and/or cracking of the product. A correct description of the evolution of moisture concentration profiles in the material is complicated by the influence of shrinkage and stresses on mass transfer.

The simplest moisture transport mathematical model in [59, 62] is a non-trivial elliptic model for the moisture content: A brick of length \( L_1 \) and height \( L_2 \) is modeled with domain \( \Omega = (0, L_1) \times (0, L_2) \). Its moisture content \( u(x) \) satisfies

\[
(1.1) \quad u_t = \nabla \cdot (D(u) \nabla u) \quad \text{in } \Omega,
\]
and \( u(x) \) satisfies some mixed boundary conditions on \( \partial \Omega \). The initial solution \( u_0 \) has a transition layer at \( x_1 = L_1 \). The non-linearity of the diffusion coefficient is schematically represented in figure 1.1.

Typically a problem like (1.1) is first discretized by a numerical method like the Finite Element Method (FEM) or the Finite Difference Method (FDM). To this end one needs a grid, i.e., a set of points connected by lines which, e.g., give rise to elements (FEM) which together cover the domain. Because of the typical form of the PDE, we need a much finer grid at the boundary of a "brick" than in the interior. In figure 1.2 we have depicted two typical choices for such a mesh, embedded in a larger frame of more bricks. The (non-linear) equations describing the moisture content at the various nodes are then solved by e.g. Newton’s method. At the heart of this method we encounter a linear system that we can write in generic form as

\[
Ax = b.
\]

Since the grid is not uniform such linear systems will be ill-conditioned in general. This hampers solving (1.2) efficiently.

Rather than intending to solve a problem like (1.1), we consider the more generic linear diffusion problem

\[
\begin{align*}
-\nabla \cdot a \nabla u + cu &= f & \text{in } \Omega = (0,1)^d \subset \mathbb{R}^d, \\
u &= g & \text{at } \partial \Omega.
\end{align*}
\]  

(1.3)

The linear system arising from discretizing (1.3) may be ill-conditioned due to two sources. In general, a non-smooth source term \( f \) will result in a non-smooth solution \( u \) of (1.3). In order to get an accurate solution, grids with local refinements as in figure 1.2 are required. A consequence of such refinements is that the resulting linear system, as in (1.2), is unbalanced because of the large differences in scale. In fact this is similar to a very inhomogeneous grid due to a rapidly varying diffusion coefficient \( a \). As remarked above, this means that the linear system becomes ill-conditioned. Another problem is that due to the relatively large number of gridpoints near the boundary the number of unknowns may rise enormously. Hence the problem size

![Figure 1.2: A piece of wall with an unstructured and structured grid.](image)
increases likewise which has its effects on efficient implementation and computation. Both phenomena, an ill-conditioned system and a large number of unknowns, lead to a larger computational complexity in general. In order to reduce this, techniques called preconditioning have been developed to reduce this complexity. However, for the problem at hand no really efficient techniques exist.

In this thesis we therefore will concentrate on constructing methods to solve linear systems arising from configurations like the ones shown in figure 1.2. More in particular, we focus on a typical brick with such a highly non-uniform mesh and will develop new ways to tackle the ill-conditioning and thus improve efficiency of iterative solvers. This then will be the test to demonstrate our platform NumLab. The construction, outline and testing then constitutes the second major aspect of this thesis.

1.3 Outline

As noted above we deal with two major subjects in this thesis. The first of these is the NumLab workbench to facilitate numerical simulations. Secondly we extend NumLab by adding tools that are designed for solving the problem as described in the previous section. Before we consider in detail the NumLab workbench, we must first construct, discuss and analyse the tools we want to add.

First in Chapter 2 a direct method is presented for the solution of convection dominated problems. In particular, we focus on grids that have refinement layers which are reminiscent of the brick-wall configuration. For such refinements this chapter shows that a complete Gaussian factorization can be efficient with respect to both fill-in as well as the band-width if the degrees of freedom are numbered such that they exploit the special grid structure. This efficient way to factorize the matrix means that inverses can be computed with an almost optimal number of floating points operations (flops).

Next, in Chapter 3, iterative methods are examined; these are usually more efficient than Gaussian elimination considered before. The main idea of such iterative methods is that the solution is computed by successive updating the existing approximation. If the method is successful, the approximation is close enough to the exact solution after a number of iterations, given a desired accuracy for the computed solution. This process of iterating and constructing the solution is described for a variety of iterative methods. Whereas with Gaussian factorization it is known beforehand that the solution can be found after the successful factorization, which can be viewed as one iteration, for iterative methods it is not known a priori how many iterations are required to obtain the solution. A priori knowledge of this is important, since the number of iterations determines the computational complexity. Unfortunately, an explicit formula for the minimum number of iterations needed is known for few iterative methods only. Another important notion in iterative methods is the condition number of the matrix. This condition number can be seen as a measure for the "quality" of the matrix and is important for error estimates. We also look at methods that can improve the computational complexity of solving the problem. When solving a problem by an iterative method, the computational complexity often grows polynomially with the number of unknowns. This is undesirable. Ideally the com-
Chapter 1: Introduction

Computational complexity should be of the same order of magnitude as the number of unknowns. In general terms, this means that every unknown has to be "looked at" only once in order to obtain the solution. Thus, what we want is an efficient way to improve this computational complexity. This is called preconditioning. What can be judged from this term is that the original problem is adapted such that the new problem is better conditioned. This in turn might lead to fewer iterations and hence a lower computational complexity. However, not all types of problems are that simple to precondition. In this chapter we address several methods that result in a theoretically ideal computational complexity for a specific type of problems. As our problem of interest does not have such properties, we concentrate on a special class of preconditioning techniques which are based on approximated inverses (Section 3.5).

In Chapter 4 we look into these types of techniques in more detail. Of particular interest for this chapter is a slightly modified form of the preconditioner. For this so-called modified approximate inverse, several results are obtained in this chapter. Numerical examples show that these also hold for the original unmodified version which is used in computations.

The approximate inverse related preconditioner is examined further in Chapter 5. Here it is shown that by applying this preconditioner, the resulting new problem can be decoupled for one dimensional problems. This phenomenon gives rise to a recursive solution method by successive application of the approximate inverse. Unfortunately, in two dimensions this cannot be generalized straightforwardly. We point at the properties that are inherited from the one dimensional case, and which are not. This leads to a different solution procedure and we provide suggestions for utilizing approximate inverse in two dimensions in a recursive manner.

The remainder of this thesis is devoted to the construction and use of NumLab. To start with, in Chapter 6 we discuss the basic ideas of our platform. NumLab enables us to implement and test new ideas by reusing existing components and building new components easily either from scratch or by putting together existing components. In NumLab, mathematical concepts, like operators and solvers, are implemented using a uniform interface with a generic formulation. This enables one to change components and parameters in a simulation. Also complex numerical tools to solve complex problems can be composed with just a few components. Finally, all tools can be used for visual programming which gives the researcher an overview of the simulation and direct ways to interact with the simulation.

In Chapter 7 we continue with the NumLab workbench, and examine how we can implement the new tools presented in Chapter 4. This chapter is the synthesis of our computational platform (NumLab) in which (complex) mathematical tools (the approximate inverse preconditioner) can be utilized in an easy manner. We seize the opportunity to present several numerical examples to show the effectiveness of the preconditioner from Chapter 4 for certain types of problems. In NumLab, running another example can be obtained by simply changing a parameter in the required module. This is detailed as well.

Finally, Chapter 8 summarizes the most important conclusions of this thesis. Furthermore some recommendations for future work are addressed.
Chapter 2

Direct methods: Efficient Gaussian elimination

In this chapter we discuss an efficient implementation of Gaussian elimination, suited for the solution of convection dominated problems. The domains of interest are such that they can be divided into similar substructures. Brick walls as have been discussed in Section 1.2 are a good example of such domains. The layer between two substructures (two bricks for example) may be such that local refinement is needed to capture the solution accurately. This part of a brick wall is simplified, but the characteristics of uniform coarse parts with refinement layers in between are inherited. For such locally refined grids we present a numbering of the degrees of freedom with favourable properties. When \( k \) levels of refinement are used the amount of fill-in is \( O(k2^k) \). The original number of degrees of freedom \( n \) is \( O(2^k) \), which implies that the complexity of the method is \( O(n \log n) \). This fill-in is comparable to the fill-in-optimized minimum degree algorithm \([46, 95]\). The bandwidth is comparable to the bandwidth-optimizing reverse Cuthill-McKee ordering \([45]\). Another benefit is that Gaussian elimination is not only applicable to Poisson type problems, but also to convection diffusion problems.

2.1 Introduction

The local bisection refinement, see \([70]\), best preserves the regular substructures of the domain of interest, much better than the more common Delaunay type gridding (see \([56]\)). Both grids in figure 1.2 – left Delaunay, right bisection – are obtained using the same input data. The bisection refinement preserves the substructures best, and in fact leads to semi-regular grids.

An even more important reason to use grids created by bisection refinement is that semi-regular grids can also also be obtained in three and more dimensions, as well as that efficient storage scheme and optimal order preconditioners exist for such refinement along lines, see \([66]\).

Several efficient solvers have been published for elliptic problems that are discretized with finite elements on refined grids. For instance, the optimal AMLI \([7, 8]\), and optimal BPX \([24]\). In principle, these solvers can solve elliptic problems without domain decomposition. When convection dominates, the solvers no longer appear
to be efficient.

In this chapter we are interested in the subdomains which contain a line along which all mesh refinement is situated (see figure 2.1). For this special subdomain, we show that a Gaussian elimination can be performed such that a near optimal band-width and near optimal fill-in is achieved. The results are based on a proper numbering of the degrees of freedom. The minimum degree algorithm [46] has somewhat less fill-in, even though no optimal tie-breaking strategy is known when different degrees of freedom have the same connectivity. It should be noted that the total amount of fill-in is quite sensitive to the tie-breaking strategy.

When solving the linear system of equations, our exact factors can be replaced by an incomplete version (see for instance [58, 74, 39, 78]), combined with Krylov-space based iterative solvers [86, 102]. However, because the focus of this chapter is on demonstrating that we get optimal fill-in and band width matrix factors for the full Gaussian elimination process, we do not provide an analysis of the incomplete case.

The substructures of interest are introduced in Section 2.2. In Section 2.3 a numbering of degrees of freedom is presented for which Gaussian elimination leads to an amount of fill-in close to that of the fill-in-optimized minimum degree algorithm [46], and on top leads to a band-width which is close to that of the reverse Cuthill-McKee ordering [45]. In the final section some examples are given to illustrate the new numbering scheme.

### 2.2 Substructuring

In this chapter, we first restrict our attention to the bisection refinement along a straight line as in figure 2.1.

![Figure 2.1: Uniform refined quadrilateral with local refinement.](image)

Furthermore we shall specifically focus on domain decompositions with building block sketched in figure 2.2, that is, each subdomain will contain either a uniform grid, or all or half of the refinement.
The subdomains we examine in detail contain all refinement and are concentrated around the refinement along a straight line. For these subdomains, we denote the amount of horizontal blocks (layers) by $K$, and $k$ denotes the level of refinement. The related grids are called $G_{K,k}$.

Let $k \geq 1$. Each horizontal block is obtained by applying $2(k - 1)$ levels of bisection refinement. Its connectivity-graph is called $B_k$. (Each two subsequent refinements shrink the smallest edge-length by a factor two.) Figure 2.3 and figure 2.4 show the refined grids related to $K = 4$ and $K = 8$ for both $k = 1$ and $k = 2$. Let $B_k$ denote the connectivity graph associated to a horizontal block. With $B_k$, we associate the coarse grid element-size $H = 1/K$ and smallest element-size $h = H \cdot 2^{-k}$. These connectivity graphs $B_k$ are discussed in more detail in the next section.
2.3 An efficient node ordering for the refinement along a line

In this section we examine complete factorization based techniques, because optimal order methods cannot be used for the problem of interest introduced in Section 1.2. Because the amount of fill-in depends on the numbering of the degrees of freedom (see [38]), we examine three numbering schemes:

(i) The Reverse Cuthill-McKee ordering [45];

(ii) Fill-in-optimized minimum degree ordering [95];

(iii) The (LR) numbering scheme proposed in this section in (2.2):
First numbering orthogonal to the line of refinement (shock) followed by numbering along the line of refinement.

It turns out that the numbering scheme (iii) proposed in (2.2) and (ii) leads to comparable amounts of fill-in, much less than the fill-in related to scheme (i). In the limit scheme (ii) tends to have less fill-in than the our scheme (iii). Of course, our scheme is faster: We do not compute but impose a numbering.

The real advantage of scheme (iii) shows when skyline storage formats are used. This storage scheme stores a sparse matrix – or its factor – using just one integer index $k_i$ per row $i$ (see for instance [85]). Large software libraries such as ISSL use the skyline because this is the format which allows the fastest possible matrix-vector multiplication. The reason is obvious: Per row all matrix entries $a_{i,i}, \ldots a_{k_i,i}$ as well as $a_{i,k_i}, \ldots a_{i,i}$ are stored as vectors of numbers. This section shows that when using the skyline storage for the LU factors, scheme (iii) uses much less memory than the minimal degree scheme (ii). The reason for this is that the band-width of the “minimal degree permuted matrix” is much larger than the band-width of the matrix with our scheme (iii).

The remainder of this section is organized as follows. First, we calculate the exact amount of fill-in created by the application of Gaussian elimination to the stiffness matrix $A_n$ on the grids in figure 2.3 and figure 2.4. Next, the amount of fill-in is compared to the amount of entries in the original stiffness matrix $A_{n'}$ and is shown to be logarithmically dependent on the level of refinement. The corresponding amount of fill-in is compared to the amount of fill-in resulting from the symmetric minimum degree ordering, as implemented in Matlab.

The convective case is not examined since the induced extra non-zero matrix entries have little or no effect on the order of the amount of fill-in. In the following discussion the degrees of freedom and the amount(s) of fill-in counts employ a certain substructure induced by the bisection refinement.

First we have a closer look at the connectivity graphs $B_k$ related to the horizontal layers in figures 2.3 and 2.4. Note that the connectivity graphs in figures 2.5 and 2.6
related to horizontal layers in figures 2.3 and 2.4 are mutually related. Figure 2.6 shows that $B_{k+1}$ consists of two sub-graphs of type $B_k$, and of two identical caps $C$, one on each side. The caps are related to a grid that contains three triangles.

The edges of all non-directed graphs (see for instance figure 2.5) correspond to the non-zero stiffness matrix entries, and the vertices to the degrees of freedom. Thus, the diagonal edges from the central division line are not present (because all related entries are zero). When counting degrees of freedom, we do not count the vertices situated at the top, and when counting fill-in, we neither count the fill-in caused by vertices at the top. Therefore the related collection of edges at the top is shown as dotted lines.

We next give a result for the number of degrees of freedom related to $B_k$.

**Theorem 2.1.** Let $N_k$ denote the amount of degrees of freedom related to $B_k$, excluding the degrees of freedom related to vertices situated at the top. Then

$$N_k = 5 \cdot 2^k - 2. \tag{2.1}$$

*Proof.* First consider the case $k = 1$. According to figure 2.5, the amount of degrees of freedom is 8, which is equal to $5 \cdot 2^1 - 2$.

Next, assume that our claim holds for some $k > 0$, so let $N_k = 5 \cdot 2^k - 2$. Due to the construction of $B_{k+1}$ in figure 2.6, $N_{k+1}$ is double the amount of $N_k$, with an additional two degrees of freedom situated at the bottom corners. Hence

$$N_{k+1} = 2(5 \cdot 2^k - 2) + 2 = 5 \cdot 2^{k+1} - 2.$$

For $K > 1$, the $K$ graphs $B_k$ are superimposed. The related amount of degrees of freedom $N_{K,k}$ is $K \cdot N_k$ plus the amount of vertices on the lower bottom line of $B_k$, which is $3 + 2k$. Thus, the total degrees of freedom is given by

$$N_{K,k} = K \cdot (5 \cdot 2^k - 2) + 3 + 2k.$$
Chapter 2: Direct methods: Efficient Gaussian elimination

Note that we pretend not to eliminate the vertices at the Dirichlet boundaries (top and bottom). The elimination would make the fill-in count below even more elaborate, but has little effect on the amount of fill-in. Thus, as a matter of fact, we count the fill-in of a stiffness matrix $A_n$ induced by Neumann boundary conditions on the domain’s top and bottom.

The Gaussian elimination process applied to $A_n$ leads to the standard upper and lower triangular factors such that $A_n = L_n U_n$. We count the amount of potential non-zero entries $\rho$ in $U_n$, excluding its diagonal. The actual amount of fill-in follows from $\rho$ and the amount of non-zero entries in $A_n$. Because the grid depends on $K$ and $k$, so do $A_n$ and $\rho$; we shall use the notation $A_{K,k}$ and $\rho_{K,k}$ respectively when necessary.

The amount of entries $\rho$ depends on the numbering of the degrees of freedom (numbering of the vertices of the graphs $B_k$). We distinguish the following numbering schemes:

\begin{equation}
\begin{aligned}
(LR) & \quad \text{First from left to right, next from bottom to top;} \\
(BT) & \quad \text{First from bottom to top, next from left to right.}
\end{aligned}
\end{equation}

The (LR) scheme is the one which leads to a nearly optimal band-width combined with nearly optimal fill-in. The (BT) scheme performs poorly with respect to both fill-in and band-width. The main reason that the (LR) numbering performs so well turns out to be that the degrees of freedom are numbered first in a direction orthogonal to the resolved line, and next in the direction tangent to this line. This numbering scheme shortens possible paths which can lead to the creation of fill-in.

In [45] paths have been characterized that can cause fill-in. Let $i < k$ be the numbers of two degrees of freedom. A related (potential) non-zero fill-in is created in $U_{K,k}$ during Gaussian elimination if there is a fill-in path: A path from vertex $v_i$ to vertex $v_k$ through vertices $v_0, \ldots, v_{i-1}$ ([76, Th. 3.3]). All counts for $\rho$ depend on the (LR) numbering. When graph $B_k$ is related to a middle layer of a grid $G_{K,k}$, its fill-in paths can leave and reenter through its bottom horizontal line. Such paths are called reentrant paths, all other paths are called non-reentrant, or internal.

**Definition 2.2.** For $k > 0$, define

(i) $P_k$, the amount of non-top vertices induced fill-in paths (inc. reentrant) of $B_k$;

(ii) $T_k$, the amount of top vertices induced fill-in paths of $B_k$;

(iii) $R_k$, the amount of bottom vertices reentrant fill-in paths of $B_k$.

Then the amount of non-zero entries $\rho_{K,k}$ is given by

\begin{equation}
\rho_{K,k} = K \cdot P_k + T_k - R_k.
\end{equation}

In Theorem 2.3 below where an explicit expression for $\rho_{K,k}$ for the (LR) scheme is given, several types of fill-in paths are distinguished:
• To the right, or diagonally or vertically up (also non-zero entry in $A_n$);
• To the left, and next diagonally or vertically up;
• If possible, first down from $v_i$, next left, and finally up to a vertex on a horizontal line above $v_i$;
• If possible, first down from $v_i$, next right, and finally up to a vertex on a horizontal line above or containing $v_i$.

**Theorem 2.3.** Let $A_{K,k}$ be the standard stiffness matrix obtained on the grids $G_{K,k}$ in figures 2.3 – 2.4, using an (LR) numbering scheme. Assume that $A_{K,k} = L_{K,k} U_{K,k}$ is factored by Gaussian elimination. Then the amount of potential non-zero entries in the strictly upper triangular part of $U_{K,k}$ is

$$
\rho_{K,k} = K \cdot \left( (10k + 16) \cdot 2^k - 4k - 7 \right) + 2(1 + k).
$$

**Proof.** The refined grid consists of $K$ horizontal layers, each related to a graph $B_k$. From (2.3) it follows that we have to count all of (i) $P_k$, (ii) $T_k$ and (iii) $R_k$.

First, consider case (i). To determine $P_k$, we first count the non-zero entries related to the bottom corner vertices of $B_k$. This amount equals

$$
4k + 7.
$$

The proof now employs induction with respect to the level $k$. Note that fill-in paths counted for $P_k$ can be reentrant.

For $k = 1$, the left corner vertex $v^l_1$ has 6 fill-in paths ending at higher numbered vertices, and the right corner vertex $v^r_1$ has 5 such paths, two of which are depicted in figure 2.7.

![Figure 2.7: One fill-in path from $v^l_1$, and one from $v^r_1$.](image-url)
Thus, for $k = 1$, the total amount of fill-in paths is:

$$6 + 5 = 4 \cdot 1 + 7$$

Next, assume that (2.5) holds for some $k \geq 1$. According to figure 2.6, $B_{k+1}$ has two subgraphs of type $B_k$, and two cap graphs $C$, each introducing a new lower corner vertex $v_{k+1}^l$ and $v_{k+1}^r$. Now $v_{k+1}^l$ has a fill-in path to all vertices $v_k^l$, and $v_{k+1}^r$ has a fill-in path to all vertices $v_k^r$. Each has two additional fill-in-paths with the two new top left and right vertices of graph $B_{k+1}$. So the new bottom left and right corner vertices have a total of

$$4k + 7 + 2 + 2 = 4(k + 1) + 7$$

fill-in paths.

For case (i), we next assert that:

(2.6)

$$P_k = (10k + 16) \cdot 2^k - 4k - 7.$$  

Again, the proof makes use of induction with respect to $k$. First, let $k = 1$. The vertex-wise fill-in path count in figure 2.8 shows that the total amount, for 8 vertices, is given by

$$7 \cdot 5 + 1 \cdot 6 = (10 \cdot 1 + 16) \cdot 2^1 - 4 \cdot 1 - 7.$$  

![Figure 2.8: The amount of fill-in paths for each vertex in $B_1$.](image)

Now, suppose (2.6) holds for some $k \geq 1$. Recall that $B_{k+1}$ has two subgraphs of type $B_k$ and two cap graphs $C$ (see figure 2.6). Thus, the total amount of fill-in paths is given by

$$2 \cdot (10k \cdot 2^k + 16 \cdot 2^k - 4k - 7)$$

fill-in paths from for 2 graphs $B_k$ (see (2.1));

$$2 \cdot 2 \cdot (5 \cdot 2^k - 2)$$

2 extra fill-in paths for 2 graphs $B_k$;

$$4(k + 1) + 7$$

fill-in paths for the new bottom corners (see (2.5)).
As desired, this adds up to \((10(k + 1) + 16) \cdot 2^{k+1} - 4(k + 1) - 7\).

Next, consider case (ii). Here we count \(T_p\), the amount of fill-in paths related to top vertices of \(B_k\):

\[
T_p = (k + 1)(2k + 3).
\]

In order to see that this is correct, first look at \(B_1\). Figure 2.9 shows the amount of extra paths for each individual vertex at the top. The amount equals

\[
4 + 3 + 2 + 1 + 0 = 10 = (1 + 1)(2 \cdot 1 + 3).
\]

Next, assume (2.7) holds for certain \(k \geq 1\). Note that the amount of vertices situated at the top horizontal line (also at bottom horizontal line) of \(B_k\) is given by

\[
2k + 3.
\]

Then the fill-in paths from the vertices at the top of \(B_{k+1}\) are:

- \((k + 1)(2k + 3)\) from top vertices of upper \(B_k\), inside \(B_k\) (see (2.7));
- \(2k + 3\) 1 extra from top vertices of upper \(B_k\) to new upper right corner (see (2.8));
- \(2(k + 1) + 3 - 1\) paths extra from new left vertex.

Again, this adds up to the desired result \(((k + 1) + 1)(2k + 1) + 3\). Keeping (2.7) in mind, the total amount of fill-in paths in case (ii) is:

\[
(10k + 16) \cdot 2^k - 4k - 7 + ((k + 1)(2k + 3)).
\]

Finally, consider case (iii). Here, we count the amount of reentrant fill-in paths \(R_k\)
related to bottom vertices of graph $B_k$. To this end, we first count the total amount of paths related to bottom vertices, and next count and subtract the amount of internal paths.

The total amount of fill-in paths related to bottom vertices of $B_k$ is given by

\[(2.9) \quad 4k^2 + 13k + 9.\]

For $k = 1$, this holds, counting all such paths in figure 2.5. Assume this amount is correct for certain $k \geq 1$. Then for $k + 1$ we have

- $4k^2 + 13k + 9$ for each bottom vertex in bottom $B_k$ (see 2.9);
- $2 \cdot (2k + 3)$ for each bottom vertex in bottom $B_k$ (see (2.8));
- $(4k + 1) + 7$ extra from two new vertices $C$ graphs.

The total amount now is $4(k + 1)^2 + 13(k + 1) + 9$ indeed.

Next we count the amount of internal paths

\[(2.10) \quad 2k^2 + 10k + 8.\]

For $k = 1$, this holds (see figure 2.5). Assume this amount is correct for certain $k \geq 1$. Then for $k + 1$:

- $2k^2 + 10k + 8$ for each bottom vertex in bottom $B_k$ (see 2.10);
- $2k + 3 - 1$ for each bottom vertex in bottom $B_k$ except right corner (see (2.8));
- $2$ for the new right bottom corner vertex in $B_k$;
- $2k + 3 + 2$ for the new right bottom corner vertex (in $C$ graph).

From (2.10) and (2.9) we obtain thus

$$R_k = (4k^2 + 13k + 9) - (2k^2 + 10k + 8) = 2k^2 + 3k + 1.$$

Finally, note that

$$T_k - R_k = (k + 1)(2k + 3) - (2k^2 + 3k + 1) = 2(1 + k),$$

which shows that (2.4) holds.

Remark 2.4. Theorem 2.3 shows that the amount of entries in the factors of $A_n$ is $O(k \cdot 2^k)$. The amount of entries in $A_n$ depends in a different manner on $k$:

Theorem 2.5. Let $A_{K,k}$ be the standard stiffness matrix obtained on the grids $G_{K,k}$ in figures 2.3 – 2.4. Then the amount of non-zero entries in $A_{K,k}$ is given by

\[(2.11) \quad A_{K,k} = 25 \cdot 2^{k-1} - 7.\]
Proof. As before we use the fact that each graph $B_{k+1}$ contains two $B_k$ subgraphs, and cap graphs $C$, as in figures 2.5 and 2.6. For $k = 1$, figure 2.10 shows that $A_{K,1}$ contains 18 non-zero entries.

![Figure 2.10](image)

Figure 2.10: Amount of non-zero entries of $A_{K,1}$.

Next assume (2.11) holds for a certain $k \geq 1$. Then, looking at figure 2.11 we deduce that the non-zero entries in $A_{K,k+1}$ are counted as

- $2 \cdot (A_{K,k})$ one for each subgraph $B_k$;
- 3 for the new right bottom corner vertex $a$ (in $C$ graph);
- 2 for the new left bottom corner vertex $b$ (in $C$ graph);
- $2 \cdot 1$ one each extra for vertices $c$ and $d$.

![Figure 2.11](image)

Figure 2.11: Amount of non-zero entries of $A_{K,k+1}$.

This adds up to the desired amount for $k + 1$ in (2.11).

Remark 2.6. Theorem 2.5 shows that the amount of entries in $A_{K,k}$ is $O(2^k)$. Thus using Gaussian elimination for the (LR) scheme produces factors which are roughly $k$ times larger than the matrix itself. In the typical case, where the average amount
of entries per row of \( A \) is small, a factor \( k = 10 \) or \( k = 20 \) is a reasonable fill-in. In fact it can compensate for \( k = 10 \) or \( k = 20 \) iterations, had an iterative method been used to solve \( Ax = b \). Note that \( k = 20 \) corresponds to a difference in scales of \( 10^{-6} \).

### 2.4 Numerical examples and conclusions

In this last section we give two examples in order to illustrate the (LR) node numbering scheme discussed in Section 2.3, and compare its efficiency to the reverse Cuthill-McKee and minimum degree ordering schemes.

**Example 2.7.** The first example considers Gaussian elimination of matrices \( A_{K,k} \) related to the grids shown in figures 2.3 and 2.4. Figure 2.12 depicts the amount of non-zero entries in the factors for the (LR), (BT) and minimum degree schemes. As

![Figure 2.12: Fill-in for \( K = 4, \ k = 0, \ldots, 7 \).](image)

figure 2.12 shows the (LR) scheme’s fill-in is close to the minimum degree scheme’s fill-in. Because an exact count of minimum degree scheme’s fill-in is not available in the literature, we cannot do comparisons for \( k \to \infty \).

Next we compare the band-width related to the (LR), minimum degree and reverse Cuthill-McKee schemes. This is shown in figures 2.13 and 2.14.
As will be clear from these figures, the envelope of the factors of the reordered minimum degree matrix is almost maximal, i.e. almost identical to the total amount of degrees of freedom. Thus, using a minimum degree ordering, matrix-vector implementations for $L$ and $U$ should make use of column indexing. The $L$ and $U$ factors from the (LR) scheme do not require such an indexing, whence matrix-vector multiplication is likely to be faster. Furthermore, the band-width related to the (LR) scheme is comparable to the band-width optimizing reverse Cuthill-McKee scheme.
Example 2.8. In this example we examine a brick-like structure as in figure 2.15, and in particular we look at the refinement layer near the boundary. This is the type of grid we had in mind in Section 1.2 as representation of a typical brick with a highly non-uniform mesh toward the boundary. The initial uniform grid in these tests has size $19 \times 19$, and at each refinement step one level of bisection refinement is applied to the layer of elements closest to the boundary. Figure 2.16 shows in detail this layer of interest, with several refinement steps.

First we compare the band-width related to the (LR), minimum degree and reverse Cuthill-McKee ordering schemes. Results are shown in figures 2.17 and 2.18.
The number of non-zeros in the LU factors for the three schemes is compared in figure 2.19 for a sequence of bisections. From these figures and results we see that the (LR) scheme compares well with the minimum degree scheme with respect to fill-in. When we compare the envelopes of the matrix, we see that for both ordering schemes the envelope for the last couple of rows and columns is maximal. However, with respect to the remainder of the matrix the (LR) scheme’s envelope seems to be more favourable when storing the matrix.
in skyline format. Also compared to the band-width minimizing reverse Cuthill-McKee scheme, the (LR) scheme has a much smaller envelope except for the last couple of rows and columns. The maximal envelope of the (LR) scheme for these rows and columns is due to our ordering of the degrees of freedom: Since our grid is circular, the nodes that are numbered first are coupled to the nodes numbered last. However, the number of rows that is filled in this way is smaller than the number of bisection steps.

In this chapter we have shown that a major part of the stiffness matrix factors related to figures 2.3 and 2.4 can be efficiently stored in skyline format, using our (LR) numbering scheme for the degrees of freedom. Thus optimal fast matrix-vector multiplications are possible. Furthermore, the (LR) numbering scheme leads to matrices with much smaller bandwidth, and we have proved that the related fill-in is logarithmic in the amount of levels of refinement. Also for grids as in figure 2.15 the (LR) numbering scheme compares well with the minimum degree and reverse Cuthill-McKee schemes, certainly when using the skyline format for storing the factors.
Chapter 3

Iterative methods and preconditioners

In this chapter we address the problem of computing the solution of a linear system of equations resulting from the discretization of a convection diffusion equation. There are two main classes of algorithms to compute the solution of a linear system: *direct methods* and *iterative methods*. The Gaussian elimination technique discussed in Chapter 2 is an example of a direct method. In this chapter we focus on iterative methods. We start with introducing the underlying problem and indicate some iterative methods. In particular we discuss the complexity of such methods and point at the important factors determining this complexity. Methods to reduce this complexity are dealt with next. This includes approximate inverse techniques as they will be the methods of choice in Chapter 4.

3.1 Iterative solution methods

In this thesis we will exclusively study linear systems of equations arising from the convection diffusion equation

\[
-a \Delta u + b \cdot \nabla u + cu = f \quad \text{in } \Omega = (0, 1)^d \subset \mathbb{R}^d,
\]

\[
u = g \quad \text{at } \partial \Omega.
\]

After discretization we typically obtain an \( n \times n \) linear system of equations

\[
A_n x_n = b_n.
\]

For the following discussion the actual type of equation and discretization method as such is not of importance. In contrast to Chapter 2 we will discuss iterative methods to solve an equation like (3.2) here. Let \( x_n^{(0)} \) denote an initial guess and \( r_n^{(0)} := b_n - A_n x_n^{(0)} \) the residual. Then the sequence \( x_n^{(k)}, k = 1, 2, \ldots \) denotes the iterates computed during the iterative process, with residuals \( r_n^{(k)} := b_n - A_n x_n^{(k)} \). Finally, let \( e_n^{(k)} := x_n^{(k)} - x_n \) be the error at iteration step \( k \).

Iterative solvers are classified as *stationary* or *nonstationary*. Stationary methods are based on a splitting of the matrix \( A_n \), denoted as

\[
A_n = P_n - Q_n,
\]
where $P_n$ is non-singular. For such methods the iteration can be expressed in the simplified form

\begin{equation}
P_n x^{(k+1)}_n = Q_n x^{(k)}_n + b_n,
\end{equation}

or

\begin{equation}
x^{(k+1)}_n = P_n^{-1}Q_n x^{(k)}_n + P_n^{-1}b_n.
\end{equation}

Now it is easy to see that for the error we have

\begin{equation}
e^{(k)}_n = (P_n^{-1}Q_n)^k e^{(0)}_n.
\end{equation}

As an example of stationary iterative methods consider the Successive Over Relaxation Method (SOR). Here $A_n$ is split up as $A_n = D_n - L_n - U_n$ where $D_n$, $-L_n$ and $-U_n$ denote the diagonal, lower-triangular and upper-triangular parts of $A_n$ respectively, and let $\omega > 0$. For the SOR method we take $P_n = \frac{1}{\omega}D_n - L_n$ and $Q_n = \frac{1-\omega}{\omega}D_n + U_n$. The parameter $\omega$ is called the relaxation parameter. Then the SOR iterative method can be written as

\begin{equation}
x^{(k+1)}_n = (D_n - \omega L_n)^{-1}(\omega U_n + (1-\omega)D_n)x^{(k)}_n + \omega(D_n - \omega L_n)^{-1}b_n.
\end{equation}

For the iteration matrix we have

\begin{equation}
P_n^{-1}Q_n = (D_n - \omega L_n)^{-1}(\omega U_n + (1-\omega)D_n).
\end{equation}

The value of $\omega$ is important for the convergence of the algorithm. Let $\rho$ denote the spectral radius of the Jacobi matrix $J(A_n) := D_n^{-1}(L_n + U_n)$, i.e. $\rho = \max_{\lambda \in \sigma(J(A_n))} |\lambda|$ where $\sigma(J(A_n))$ is the spectrum of $J(A_n)$. Then the theoretically optimal value for $\omega$ is given by (cf. [9])

\[
\omega_{\text{opt}} = \frac{2}{1 + \sqrt{1 - \rho^2}}.
\]

There exists a host of such stationary methods; however they are not of much importance for our particular problem here. We are actually more interested in nonstationary iterative methods. Though these are usually more difficult to implement, they can be very effective. The difference between stationary and nonstationary methods is that in nonstationary methods the computations involve information that changes at each iteration. The principle of these methods is that a projection process is employed one way or another. As such, at each iteration $k$ the new residual $r^{(k)}_n$ is computed such that it is orthogonal to a subspace $K_k$ of dimension $k$ formed in the $k-1$ previous iterations. In other words, we construct a new approximate solution $x^{(k)}_n$ such that

\[b_n - A_n x^{(k)}_n \perp K_k.
\]

Of particular interest are subspaces $K_k$ defined by

\begin{equation}
K_k = \text{span}\{r^{(0)}_n, A_n r^{(0)}_n, \ldots, A_n^{k-1}r^{(0)}_n\}.
\end{equation}
This is the so called \textit{Ritz-Galerkin approach}. Subspaces defined this way are called \textit{Krylov subspaces}, and methods based on orthogonalisation of the new residual over the current subspace are called \textit{Krylov subspace methods}. It should be noted that other methods can be constructed by selecting other subspaces for the orthogonality condition.

The best known of these Krylov subspace methods is the \textit{Conjugate Gradient Method} (CG) (see [51]) for symmetric positive definite systems. This method generates a sequence of vectors that satisfy the Ritz-Galerkin condition in (3.6). In short, the CG algorithm can be described as follows (see for example [9, 84]). Let $p_n^{(k)}$ be the search direction vector at iteration step $k$. Then the iterate $x_n^{(k)}$ is updated as

$$x_n^{(k+1)} = x_n^{(k)} + \alpha_k p_n^{(k)}$$

for a scalar $\alpha_k$. Then for the new residual $r_n^{(k+1)}$

$$r_n^{(k+1)} = r_n^{(k)} - \alpha_k A_n p_n^{(k)}.$$ 

With $\alpha_k = (r_n^{(k)}, r_n^{(k)}) / (p_n^{(k)}, A_n p_n^{(k)})$ we minimize the error $(r_n^{(k+1)}, A_n^{-1} r_n^{(k+1)})$ over the existing subspace $K_k$. Finally the search direction is updated as well

$$p_n^{(k+1)} = r_n^{(k+1)} + \beta_k p_n^{(k)},$$

with $\beta_k = (r_n^{(k+1)}, r_n^{(k+1)}) / (r_n^{(k)}, r_n^{(k)})$. This ensures that $r_n^{(k+1)}$ and $r_n^{(k)}$ are orthogonal.

For convergence estimates of the CG method the \textit{spectral condition number} $\kappa_2$ of $A_n$ is used. For a symmetric positive definite matrix $A_n$, if $\lambda_n$ and $\lambda_1$ are the maximum and minimum eigenvalues of $A_n$ respectively, then

$$\kappa_2(A_n) = \frac{\lambda_n}{\lambda_1}.$$ 

Then it can be shown that after $k$ iterations the error $e_n^{(k)}$ satisfies

$$\|e_n^{(k)}\|_{A_n} \leq 2 \left( \frac{\sqrt{\kappa_2(A_n)} - 1}{\sqrt{\kappa_2(A_n)} + 1} \right)^k \|e_n^{(0)}\|_{A_n},$$

where $\|y_n\|_{A_n} := (y_n, A_n y_n)_2$ (see for example [76]).

For non-symmetric matrices the CG method is not suitable because the residual vectors cannot be made orthogonal. One approach to solve iteratively for non-symmetric matrices is the \textit{Bi-Conjugate Gradients Method} (Bi-CG) (see for example [103]). Implicitly this algorithm not only solves the original system $A_n x_n = b_n$, but also its dual linear system $A_n^T \bar{x}_n = \bar{b}_n$. Explicitly the standard CG algorithm is augmented...
by a second sequence of residuals \( \tilde{r}_n^{(k)} \) and search directions \( \tilde{p}_n^{(k)} \) based on multiplication by \( A_n^T \) and arbitrary \( r_n^{(0)} \), i.e.

\[
(3.12) \quad \tilde{r}_n^{(k+1)} = \tilde{r}_n^{(k)} - \alpha_k A_n^T \tilde{p}_n^{(k)}
\]

and

\[
(3.13) \quad \tilde{p}_n^{(k+1)} = \tilde{r}_n^{(k+1)} + \beta_k \tilde{p}_n^{(k)}.
\]

Here \( \alpha_k = (\tilde{r}_n^{(k)}, r_n^{(k)})/(\tilde{p}_n^{(k)}, A_n p_n^{(k)}) \) and \( \beta_k = (\tilde{r}_n^{(k+1)}, r_n^{(k+1)})/(\tilde{r}_n^{(k)}, r_n^{(k)}) \). For symmetric \( A_n \), Bi-CG reduces to the standard CG, at twice the cost of CG.

However, multiplication by the matrix \( A_n^T \) may be cumbersome to implement efficiently. This has led to the **Conjugate Gradient Squared Method** (CGS) (see [89]) which is applicable for non-symmetric matrices, but does not need operations with \( A_n^T \). This is achieved as follows. First we observe that the residual \( r_n^{(k)} \) and the search direction \( p_n^{(k)} \) can be expressed as

\[
(3.14) \quad r_n^{(k)} = P_k (A_n) r_n^{(0)} \quad \text{and} \quad p_n^{(k)} = Q_{k-1} (A_n) r_n^{(0)},
\]

where \( P_k \) and \( Q_{k-1} \) are polynomials of degree \( k \) and \( k - 1 \) respectively. Then

\[
(3.15) \quad \alpha_k = \frac{(P_k (A_n^T) \tilde{r}_n^{(0)}, P_k (A_n) r_n^{(0)})}{(Q_{k-1} (A_n^T) \tilde{r}_n^{(0)}, A_n Q_{k-1} (A_n) r_n^{(0)})} = \frac{(\tilde{r}_n^{(0)}, P_k (A_n)^2 r_n^{(0)})}{(\tilde{r}_n^{(0)}, A_n Q_{k-1} (A_n)^2 r_n^{(0)})}.
\]

This leads to the idea of looking at residuals that satisfy

\[
\tilde{r}_n^{(k)} = P_k (A_n)^2 r_n^{(0)}.
\]

Let \( \tilde{r}_n \) be an arbitrary vector such that \( (\tilde{r}_n, r_n^{(0)}) \neq 0 \). Then the CGS algorithm consists of the following steps. Let \( \beta_{k-1} = (\tilde{r}_n, r_n^{(k-1)})/(\tilde{r}_n, r_n^{(k-2)}) \). Then let \( u_n^{(k)} \) be an additional vector given by

\[
(3.16) \quad u_n^{(k)} = r_n^{(k-1)} + \beta_{k-1} q_n^{(k-1)}.
\]

The search direction \( p_n^{(k-1)} \) is now updated according to

\[
(3.17) \quad p_n^{(k)} = u_n^{(k)} + \beta_{k-1} (q_n^{(k-1)} + \beta_{k-1} p_n^{(k-1)}).
\]

With \( v_n^{(k)} = A_n p_n^{(k)} \) and \( \alpha_k = (\tilde{r}_n, r_n^{(k-1)})/(\tilde{r}_n, v_n^{(k)}) \), let

\[
q_n^{(k)} = u_n^{(k)} - \alpha_k v_n^{(k)}.
\]

The new approximate solution and new residual are then computed as

\[
(3.18) \quad x_n^{(k)} = x_n^{(k-1)} + \alpha_k (u_n^{(k)} + q_n^{(k)}).
\]
The quadratic nature of the recursion is expressed in the extra vectors needed for the computation of the new search direction, approximation and residual. In practice it is often observed that the CGS method converges twice as fast as the Bi-CG method, though this has not been proved yet. A disadvantage of CGS is that it usually shows irregular convergence behaviour which can lead to cancellation of the iterative process. Because of its speed, however, the CGS method will be our iterative method of choice for our numerical examples in Chapter 7.

3.2 Complexity of iterative methods

The efficiency of an iterative method depends on certain characteristics of the linear system and on characteristics of the method itself. In general, an ill-conditioned linear system will lead to slow convergence of the iterative method used. This means that a large number of iterations is needed in order to achieve the required accuracy of the solution. A large number of iterations implies a large number of floating point operations, which in turn increases computational time. Besides causing a large number of flops, an ill-conditioned linear system also affects the accuracy of the solution. Ideally, we want the number of flops to be proportional to the number of degrees of freedom $n$, provided we only have a single processor computer at hand. However, this ideal situation is rarely achieved. In order to tackle the problem of a large number of flops and reduce the number of iterations, and hence the number of flops, linear systems are preconditioned. Basically this means a basis transformation: A second linear operator is constructed (the preconditioner) such that the preconditioned linear system, i.e. the system obtained after the basis transformation, is better conditioned. Preconditioning is described in detail in Section 3.4.

The efficiency of an iterative method is measured by the total workload or computational complexity $w(\epsilon)$. This is the number of floating point operations (flops) required for finding an approximation $x_n^{(k)}$ for (3.2) such that $\|r_n^{(k)}\| \leq \epsilon$. For iterative methods, the total workload depends on the number of flops per iteration, and on the number of iterations $\lambda(\epsilon)$ needed to achieve the required accuracy. For so called sparse systems, i.e., with only a few non-zero entries per row, as we have in our case the number of flops per iteration is typically $O(n)$; indeed an iteration involves only a number of vector updates and matrix-vector multiplications (see for example [9]). This gives

$$w(\epsilon) = \lambda(\epsilon)O(n),$$

which indicates that the number of iterations is the important factor for determining the total workload. Note that this definition of the total workload does not allow to assess iterative methods that have different number of flops each iteration step.

Ideally we want the number of flops to be proportional to the number of degrees of freedom $n$, (i.e. assuming we only have a single processor computer at hand).
Chapter 3: Iterative methods and preconditioners

Hence, an iterative method is called \textit{optimal} if the total workload \( w(\epsilon) \) is linearly proportional to the number of degrees of freedom \( n \), i.e.

\[ w(\epsilon) = C \cdot n. \]

(3.20)

On a parallel computer with \( p \) processors the total workload would ideally be

\[ w(\epsilon) = \frac{C}{p} \cdot n. \]

If we elaborate further on this, and assume we can employ \( n \) processors in parallel, then \( w(\epsilon) = O(1) \). This does not take into account inter-processor communication. Furthermore this estimate does not hold when \( n \to \infty \).

\textit{Suboptimal} order methods have a workload that typically grows faster than \( n \). If \( w(\epsilon) = C \cdot n \log(n) \) we have a method that is only marginally suboptimal. Really suboptimal methods have a workload that is polynomially dependent on \( n \), i.e. for some \( \theta > 0 \) we have

\[ w(\epsilon) = C \cdot n^{1+\theta}. \]

The number of iterations \( \lambda(\epsilon) \) is a most important factor in determining the workload. This number depends on the specific iterative method used. For few iterative methods the specifics that determine the worst case of the number of iterations \( k \) needed to achieve \( \|r_n^{(k)}\| \leq \epsilon \) are explicitly known.

1. SOR: the convergence of this method depends strongly on the choice of the relaxation parameter \( \omega \). The optimal value \( \omega_{\text{opt}} \) for this parameter depends on the spectral radius of the Jacobi matrix, and in general this optimal value is not easy to compute. For the Poisson problem on a uniformly refined two dimensional grid, this value \( \omega_{\text{opt}} \) is easy to compute, and is found to be \( 2/(1 + \sin(\pi h)) \). For the spectral condition number of the iteration matrix in (3.5) we find \( \rho(P_n^{-1}Q_n) \approx 1 - 2\pi h \) (see [76]). Then the number of iterations \( \lambda(\epsilon) \) needed such that \( e_n^{(k)} \leq (P_n^{-1}Q_n)^k e_n^{(0)} \) for \( k > \lambda(\epsilon) \) is of order \( \log(\epsilon)/\log(1 - 2\pi h) \), under the assumption that \( \|e_n^{(0)}\| = 1 \). Typically we have \( \epsilon = O(h^2) \) so that \( \log(\epsilon) = -\log(n) \). So, \( \lambda(\epsilon) \) is estimated by \( n \log(n) \). Using the fact that SOR costs \( n \) flops per iteration, we find for the total workload \( w(\epsilon) = n^2 \log(n) \).

2. CG: rate of convergence depends on the spectral condition number of the matrix \( A_n \). Typically for a Laplace type operator \( (a = 1, b = 0, c = 0 \text{ in (3.1)}) \) in two dimensions we have \( \kappa_2(A_n) = O(n) \) (see Section 3.3 for a proof). Then for given tolerance \( \epsilon \) the number of iterations \( \lambda(\epsilon) \) required such that

\[ \|x_n - x_n^{(k)}\|_A \leq \epsilon \|x_n - x_n^{(0)}\|_A \text{ for } k > \lambda(\epsilon), \]

given by

\[ \lambda(\epsilon) \leq \sqrt{\kappa_2(A_n)} \log(2/\epsilon) + 1. \]

(see [5]). Hence, with \( \kappa_2(A_n) = O(n) \) and \( O(n) \) flops per iteration, for such a problem we have \( w(\epsilon) \doteqdot n^2 \) which is suboptimal.
Estimates for the spectral condition number

For other methods mentioned before there are no estimates for the worst case number of iterations needed.

3.3 Estimates for the spectral condition number

In the previous section we mentioned the dependence of certain iterative methods on the spectral condition number of the matrix. This spectral condition number is also used in error estimates. In this section we give several results on estimates for the spectral condition number. The results are for a symmetric positive definite linear system originating from a finite element discretization in particular.

If we take \( b = 0 \) and \( c = 1 \) in (3.1) we can define two bilinear forms. For the second order term \( a \Delta u \) we define the stiffness bilinear form

\[
(A, v) := \int_{\Omega} a \nabla u \nabla v,
\]

and the mass bilinear form

\[
(C, v) := \int_{\Omega} cu,
\]

for the mass term \( cu \). Because \( c = 1 \) we will omit the \( c \) in the following discussion.

Let \( \{\phi_i\}_{i=1}^n \) be a basis, spanning a finite dimensional Hilbert space \( H \) and define the linear operators

\[
[A_n]_{i,j} := A(\phi_j, \phi_i) = \int_{\Omega} a \nabla \phi_j \nabla \phi_i, \quad i, j = 1, \ldots, n,
\]

and

\[
[C_n]_{i,j} := C(\phi_j, \phi_i) = \int_{\Omega} \phi_j \phi_i, \quad i, j = 1, \ldots, n.
\]

Finally, let \( \lambda_{\text{min}} \) denote the smallest eigenvalue of the continuous eigenproblem

\[-\Delta u = \lambda u, \text{ in } \Omega,\]

where \( u \) satisfies the homogeneous imposed Dirichlet boundary conditions.

For discretization we cover the domain \( \Omega \) by a grid \( \Omega_h \). An element of \( \Omega_h \) is denoted by \( e \) and we define aspect ratios

\[
h := \min_e \{\text{diam}(e)\}, \quad H := \max_e \{\text{diam}(e)\}.
\]

Let \( a_{\text{min}} := \min_{x \in \Omega} \{a(x)\} \) and \( a_{\text{max}} := \max_{x \in \Omega} \{a(x)\} \). Furthermore, let \( p_{\text{min}} \) and \( p_{\text{max}} \) be the minimum and maximum number of elements respectively which share any specific degree of freedom.

Then for \( \kappa_2(A_n) \) we have the well known result, cf. [42, 43].
Chapter 3: Iterative methods and preconditioners

**Theorem 3.1.**

\[(3.23)\]

\[\kappa_2(A_n) \leq \gamma \frac{H^d}{h^d} \frac{H^{-2}}{\lambda_{\min}} \frac{a_{\max}}{a_{\min}} \frac{p_{\max}}{p_{\min}},\]

where the constant \(\gamma\) does not depend on the aspect ratios \(h\) and \(H\).

For a proof one may consult [42, pp. 164 -167] or [5, pp. 232-240].

For the mass matrix \(C_n\) an estimate for its spectral condition number is given by

**Theorem 3.2.**

\[(3.24)\]

\[\kappa_2(C_n) \leq \gamma_c \frac{H^d}{h^d} \frac{p_{\max}}{p_{\min}},\]

Thus the mass matrix has a condition number of order \(O(1)\), if \(H^d/h^d\) is bounded for \(n \to \infty\).

Combining the two previous results gives

**Corollary 3.3.**

\[(3.25)\]

\[\kappa_2(A_n + C_n) = O(H^{d-2}) + O(H^d).\]

**Proof.** From theorems 3.1 and 3.2 we find

\[((A_n + C_n)u, u)_2 \leq \gamma_1 H^{d-2} + \gamma_2 H^d,\]

and

\[((A_n + C_n)u, u)_2 \geq \gamma_3 h^d.\]

The constants \(\gamma_1, \gamma_2\) and \(\gamma_3\) do not depend on aspect ratios \(H\) and \(h\). This completes the proof. \(\square\)

If the factors \(a_{\max}/a_{\min}\) and \(p_{\max}/p_{\min}\) are known and independent of \(H\) and \(h\), then the result in Theorem 3.1 can be reformulated as

\[(3.26)\]

\[\kappa_2(A_n) = O(H^{d-2} \frac{H^d}{h^d}).\]

The previous result (3.26) is less interesting than it may seem at first sight. Consider the Laplace operator, i.e. (3.1) with \(a = 1\) and \(b = 0, c = 0\), in two dimensions. For a uniform grid we have the estimate \(\kappa_2(A_n) = O(h^{-2}) = O(n)\). Numerical observations show that this estimate is sharp. However, for non-uniform grids this estimate is not sharp. We demonstrate this for two grids with local refinements as in figures 3.1 and 3.2 by direct computation. In both examples the size of \(H\) remains unchanged and \(h\) is halved when an extra level of refinement is added. From (3.23) it would follow that an extra level of refinement results in an increase of the condition number by a factor 4. However, from table 3.1 we observe that \(\kappa_2(A_n) = O(l)\) for a grid as in figure 3.1; here \(l\) is the level of refinement. For the second grid the results in table 3.2 seem to indicate that \(\kappa_2(A_n) = O(n)\).
3.4 Preconditioning

In this section we examine ways to obtain better convergence for iterative methods. Improvements can be obtained by what is called \textit{preconditioning}. Basically this means that a new linear operator \(M_n\) is constructed, called a \textit{preconditioner}, such that multiplying \(A_n\) by \(M_n\) results in a lower total workload \(w(\epsilon)\). Multiplying can be done
either from the right or from the left. In order to do so, it is necessary that both the
collection of and the multiplication by \( M_n \) is relatively cheap. In the following
we first describe how to use such a preconditioner in general. Then we give a sur-
vey of existing techniques. This starts with methods that result in an optimal order
workload, followed by other techniques resulting in suboptimal order.

### 3.4.1 Preconditioning methods

Two classes of methods for the construction of a preconditioner \( M_n \) can be distin-
guished: explicit methods and implicit methods. For explicit methods we compute an
approximation \( M_n \) to \( A_n^{-1} \), and solve the preconditioned linear system

\[
M_n A_n x_n = M_n b_n.
\]

Since both \( M_n \) and \( A_n \) are explicitly available, only matrix-vector multiplications
are required in iterations. Alternatively we can solve a right preconditioned linear
system

\[
A_n M_n y_n = b_n, \quad x_n = M_n u_n.
\]

For implicit methods \( M_n \) approximates \( A_n \). This gives the preconditioned system

\[
M_n^{-1} A_n x_n = M_n^{-1} b_n.
\]

Here it is seen that in an iterative method each matrix-vector multiplication requires
the solution of a linear system involving \( M_n \). The necessity of cheaply solving a
system with \( M_n^{-1} \) in order to optimize the workload is evident. We also have the
right preconditioned version, i.e.

\[
A_n M_n^{-1} y_n = b_n, \quad x_n = M_n^{-1} y_n.
\]

For completeness we mention another variant of preconditioning, viz. one that pre-
serves symmetry in case \( A_n \) itself is symmetric. In this situation \( M_n \) is available in a
factored form, i.e.

\[
M_n = L_n L_n^T.
\]

The preconditioner is then split between left and right, and the system to solve be-
comes:

\[
L_n^{-1} A_n (L_n^{-1})^T y_n = L_n^{-1} b_n, \quad x_n = (L_n^{-1})^T y_n.
\]

Despite the various ways to create a new, preconditioned, linear system of equations,
the goal is to achieve a better conditioned linear operator such that iterative methods
converge in less steps. For methods depending on the spectral condition number of
the matrix this means (for (3.29))

\[
\kappa_2(M_n^{-1} A_n) \ll \kappa_2(A_n)
\]

asymptotically in \( n \). Of course, provided construction of and multiplication by \( M_n \)
is cheap.
3.4.2 Optimal order techniques for diffusive problems

Consider the situation of a uniformly refined grid, such that the ratio $H/h$ is fixed for each grid level. These methods are all of multilevel type, and the most important ones are:

- The Algebraic Multi-level Iterative (AMLI) methods. First appeared in [7, 8];
- The Bramble, Pasciak and Xu (BPX) preconditioner. Published in [24];
- Multigrid methods. See for example [25].

Despite many differences all three methods employ a breakdown of the finest grid into coarser grids. This results in a recursive structure of the preconditioner. The basic idea is that a specific relation between two successive levels of refinement, cq. grids, is exploited and used to represent the solution in terms of this breakdown in refinement levels. Recursion in the sense of exploiting self-similar substructures was used in Chapter 2, which resulted in a nearly optimal order method ($O(n \log n)$). In Chapter 5 a recursive nature of grids is exploited as well in a very straightforward manner. This results in a nearly optimal order method as well.

Secondly there are optimal order methods for non-uniform grids where $H/h \to \infty$ with increasing level of refinement. These are also of the multilevel type:

- Related to the AMLI method is a method that make use of special 2-d bisection refinement for AMLI permitted coarse grids. See [66];
- For the BPX method non-uniform grids are also allowed. However, bisection refinement is not allowed, and only the highest level element can be further refined. See [19, 35].

It should be noted that the type of refinement used is important for both methods.

**Remark 3.5.** Optimal order methods can be used for non-symmetric and indefinite elliptic problems too. There exists an extensive literature on such problems. In [41] a two-level preconditioner for grids with local refinement is constructed. Under the assumptions that the coarse grid size $H$ is small enough, and that the ratio $H/h$ is bounded, an optimal order preconditioner can be obtained. Multigrid is also employed for non-symmetric and indefinite elliptic problems and has been extensively studied; see for example [21, 23, 32, 33, 88, 108]. One of the recurring assumptions is that the coarsest grid is sufficiently fine in order to ensure a uniform convergence rate. It also appears that computation of the restriction and prolongation operators are more costly than solving the original problem.

**Remark 3.6.** Unfortunately, there are situations when optimal order preconditioners cannot be employed. In those situations one has to revert to preconditioning methods that are suboptimal. For symmetric elliptic boundary value problems in a finite element context, a preconditioner based on domain decomposition was introduced...
Chapter 3: Iterative methods and preconditioners

in [22]. This method, called BPS, has the following estimate of the condition number of the preconditioned system

\[ \kappa(M_n^{-1}A_n) = O(1 + (\ln(H/h))^2). \]

Here \( H \) denotes the size of the subdomains, and \( h \) is the mesh-size.

3.4.3 Incomplete factorization methods

An important class of preconditioners are the incomplete factorizations methods. These preconditioners are based on direct solution methods (cf. Chapter 2), where part of the computation is skipped. This results in incomplete factors \( L_n \) and \( U_n \) such that \( L_n U_n \approx A_n \). These are then employed as preconditioners as in (3.31). For literature see [74] and [31, 75, 84, 103, 104].

Given the incomplete calculation of the factors \( L_n \) and \( U_n \), a lot of different strategies have been studied. The most straightforward strategy is to compute factors that have the same sparsity pattern \( S \) as \( A_n \). This method is called ILU(0). An immediate improvement over ILU(0) are techniques that allow fill-in according to certain criteria. An approach is to allow fill-in at certain positions such that it leads to more non-zero bands in the factors. This is similar to omitting fill-in when the fill-in is outside a prescribed sparsity pattern, which is called the drop by position strategy. These methods are called ILU(\( p \)) where \( p \) denotes the level of fill-in allowed. Besides dropping fill-in based on a criteria for the position of fill-in, fill-in can also be omitted if its absolute value is below a certain threshold \( \tau \), which is a drop by size strategy. These methods are referred to as ILU(\( \tau \)) or ILUT, where the \( \tau \) denotes the threshold. Finally we mention the Modified ILU (MILU) method ([49]), which employs a form of additional correction to the diagonal entries. This correction leads to \( L_n U_n w_n \approx A_n w_n \) for almost constant \( w_n \). The efficiency of the MILU method strongly depends on the amount of correction applied (see [101]). Let \( 0 \leq \alpha \leq 1 \) be a parameter steering the amount of correction. For \( \alpha = 0 \) we obtain the ILU(0) method and \( \alpha = 1 \) results in the standard MILU method. Experiments indicate the \( \alpha = .95 \) leads to a very substantial reduction in the number of iterations. Of interest is to note that the MILU method is employed as smoother for multigrid methods (see [90, 106]).

3.5 Approximate inverse preconditioning

In the previous section we gave a summary of few of the most well known (implicit) preconditioning methods. In this section we look at explicit preconditioners. Although implicit preconditioners have been successfully employed in a number of applications, there are two main reasons to examine explicit preconditioners. First of all, with the introduction of high-performance architectures straightforward implementation of implicit preconditioners could lead to lowering of performance. Especially ILU-type preconditioners are troublesome to implement in parallel, because of the recursive nature of the computation. Another drawback of ILU-type preconditioners is the possibility of breakdowns during the factorization process, due to zero
An approximate inverse preconditioner typically occurs when the matrix $A_n$ is non-symmetric and/or indefinite. Even if some care is taken during the factorization, there is no guarantee that the resulting preconditioners will work properly.

A viable alternative is to construct a preconditioner $G_n$ that is an approximation to $A_n^{-1}$. As mentioned in Subsection 3.4.1 this preconditioner requires only matrix-vector multiplications. Added benefit is that matrix-vector multiplications are well suited for parallelism. Ideally, we would like $G_n$ to resemble $A_n^{-1}$ as much as possible, as this obviously would result in a preconditioned linear system that is close to the identity matrix. However, the inverse of a sparse matrix $A_n$ is full in general, which necessitates a more careful approach in the construction of such an explicit preconditioner. These considerations have led to the development of so-called approximate inverse preconditioners. Basically the approximate inverse is constructed by minimizing a consistent norm of the difference between the identity $I_n$ and the preconditioned linear system. The motivation for looking at such preconditioners is the following:

For the continuous differential and boundary operators $L$ and $B$ respectively, the Green's functions give a representation of the inverse operator. So, intuitively, discrete versions of the Green's functions will be good approximations to $A_n^{-1}$ where $A_n$ is a discretization of $L$ and $B$. However there are several restrictions. Green's functions are rarely available in closed form. Even if an explicit formula for the Green's functions is available, a discrete version, for example by sampling, is not always beneficial. One major reason is that Green's functions have global support which will result in a full approximate inverse whereas an efficient preconditioner should be sparse. Also, in two dimensions Green's functions have a singularity at the source point, which makes sampling more difficult. The problem of a full approximate inverse may be avoided by sampling the Green's function in few points only, so that a sparse approximate inverse is obtained. However, this turns out not to be satisfactory either. Yet, let us call the approximate inverse $G_n$. For simplicity we will also call this matrix the Green's matrix.

In the following, we discuss several approximate inverse strategies, but only the most relevant details are given. Most, if not all, literature on approximate inverses deals with implementation and application only, which in fact means a lack of theoretical results. The method as such, however, is important for our discussion. In Chapter 4 we will discuss our choice of $G_n$ in greater detail.

### 3.5.1 Frobenius norm minimization

Historically Frobenius norm minimization is the first approximate inverse technique that was proposed (see for instance [12]). The main idea is to compute a sparse matrix $G_n \approx A_n^{-1}$ as the solution of the constrained minimization problem

\begin{equation}
\min_{G_n \in \mathcal{S}_n} \| I_n - A_n G_n \|_F.
\end{equation}

Here $\mathcal{S}_n$ denotes a set of sparse matrices with sparsity pattern $S$, and $\| \cdot \|_F$ denotes the Frobenius norm. Let $e_i$ denote the $i$th column of $I_n$ and $g_i$ the $i$th column of $G_n$. 


Chapter 3: Iterative methods and preconditioners

Since
\[ \| I_n - A_n G_n \|_F^2 = \sum_{i=1}^{n} \| e_i - A_n g_i \|_2^2, \]
the minimization problem (3.33) to compute \( G_n \) is replaced by solving \( n \) distinct linear least squares problems, each with constraints on the sparsity of \( g_i \). In order to compute the \( g_i \) we minimize
\[ \| e_i - A_n g_i \|_2^2 \]
given the sparsity pattern for \( g_i \). This boils down to solving
\[ (3.34) \quad [A_n g_i]_j = [e_i]_j, \quad (j, i) \in S. \]

This method has the highest potential for parallelism. The approximate inverse technique described in Chapter 4 is based on Frobenius norm minimization and solving (3.34) for obtaining \( G_n \) in particular.

This Frobenius norm minimization can be generalized into a weighted variant (see [4]). Given a symmetric positive definite matrix \( H_n \), the norm to be minimized is
\[ (3.35) \quad \| I_n - A_n G_n \|_{F_H}^2 \equiv \text{tr}((I_n - A_n G_n)H_n(I_n - A_n G_n)^T). \]
The \( n \) linear systems needed to compute the columns of \( G_n \) are
\[ A_n^T H_n A_n g_i = A_n^T H_n e_i, \quad i = 1, \ldots, n. \]

Clearly, \( H_n = I_n \) gives back (3.33).

If \( A_n \) is symmetric positive definite, with \( H_n = A_n^{-1} \) the approximate inverse \( G_n \) satisfies
\[ [A_n G_n]_{i,j} = \delta_{i,j}, \quad (i, j) \in S. \]

One of the important issues is the choice of the sparsity pattern \( S \). If \( S \) is given or known a priori, then computation of \( G_n \) is straightforward. This computation is highly parallelizable since the computation of \( g_i \) is a local process only.

The role of \( S \) is intended to remove small entries of \( A_n^{-1} \), and retain the large entries, that influence the quality of the preconditioner. However, for an arbitrary sparse matrix it is not known in advance which entries of its inverse are large. This makes the choice of \( S \) difficult. In order to overcome this problem, a common choice is to take the sparsity pattern of \( A_n \) for \( S \). If there are still large elements in \( A_n^{-1} \) located outside the non-zero pattern of \( A_n \), another choice is to take the sparsity pattern of \( A_n^p \), \( p \geq 2 \). In general this means that more information of \( A_n^{-1} \) is captured in the approximate inverse \( G_n \). However, there is no guarantee it will work properly. Also the cost for computing and storing the approximate inverse grows rapidly for increasing \( p \).

As the problem of prescribing a good sparsity pattern \( S \) for \( G_n \) remained, adaptive strategies have been developed, see [34, 47]. The approach in [47] is probably the most successful and is better known as the SParse Approximate Inverse (SPAI) preconditioner. The strategy is to start with an initial sparsity pattern and to enlarge it when
entries found outside the sparsity pattern at that moment are considered to be large according to some threshold. See [17, 47] for more details of the SPAI preconditioner.

3.5.2 Factorized sparse approximate inverses

A big disadvantage of the Frobenius norm based approximate inverse is that $G_n$ does not preserve symmetry and/or positive definiteness of $A_n$ in general, even if $A_n$ is. This means it cannot be used with the CG method to solve symmetric positive definite problems. We consider therefore a method that does have this property. It is based on computing a factorized sparse approximate inverse using incomplete inverse factorizations.

Assume that $A_n$ can be factored as $L_n D_n U_n$ where $L_n$ is lower triangular, $D_n$ is diagonal and $U_n$ is upper triangular. Then $A_n^{-1} = U_n^{-1} D_n^{-1} L_n^{-1}$. The factorized sparse approximate inverse is constructed by computing its factors $Z_n$ and $W_n$ which are sparse approximations to $U_n^{-1}$ and $L_n^{-1}$ respectively. Then as approximate inverse we take

$$G_n = Z_n D_n^{-1} W_n.$$  

Several approaches have been developed to compute the approximate factors $Z_n$ and $W_n$.

A first class of methods does not need a factorization of $A_n$, but constructs the factorized approximate inverse directly from $A_n$. This includes the Factorized Sparse Approximate Inverse (FSAI) method introduced in [60] which can be described as follows. Assume that $A_n$ is symmetric positive definite, and let $S_L$ denote a prescribed lower triangular sparsity pattern including the main diagonal. Then the lower triangular matrix $\hat{Z}_n$ is computed by solving

$$[A_n \hat{Z}_n]_{i,j} = \delta_{i,j}, \ (i,j) \in S_L.$$  

Let $\hat{D}_n := (\text{diag}(\hat{Z}_n))^{-1}$ and $Z_n = \hat{D}_n^{1/2} \hat{Z}_n$, then the preconditioned linear system $Z_n A_n Z_n^T$ is symmetric positive definite, and diagonal entries equal to 1. A common choice for $S_L$ is to take

$$S_L := \{(i, j) : [A_n]_{i,j} \neq 0, i \geq j\},$$

i.e. the non-zeros in the lower triangular part of $A_n$. For a better approximation, albeit at a higher cost, another choice is to take the non-zeros in the lower triangular part of $A_n^p$ for $p \geq 2$. The main drawback of FSAI is that it relies on a prescribed sparsity pattern, and is not well suited for general sparse matrices. The method can be extended to the case of non-symmetric matrices. However, the solvability of the local linear systems and the non-singularity of the approximate inverse is no longer guaranteed.

A second method of computing a factorized approximate inverse is discussed in [14]. The advantage of this method is that a prescribed sparsity pattern is not required.
Chapter 3: Iterative methods and preconditioners

This method is based on incomplete (bi)conjugation and generally referred to as \textit{AINV}. The algorithm computes two sets of vectors, \( \{w_i\}_{i=1}^n \) and \( \{z_i\}_{i=1}^n \) that are \( A_n \)-biconjugate, i.e.

\[(3.39) \quad z_i^T A_n w_j = 0 \Leftrightarrow i \neq j.\]

Let

\[ Z_n = [z_1, \ldots, z_n], \quad \text{and} \quad W_n = [w_1, \ldots, w_n], \]

then

\[(3.40) \quad W_n^T A_n Z_n = D_n = \text{diag}(p_1, \ldots, p_n) \]

where

\[ p_i = w_i^T A_n z_i \neq 0. \]

\( W_n \) and \( Z_n \) are non-singular and hence

\[ A_n^{-1} = Z_n D_n^{-1} W_n^T \]

which gives us a factorization of \( A_n^{-1} \).

In general \( Z_n \) and \( W_n \) are dense, and to get a sparse approximate inverse, the matrices \( Z_n \) and \( W_n \) are computed incompletely. This gives incomplete factors \( \hat{Z}_n, \hat{W}_n \) and an approximation \( \hat{D}_n \approx D_n \). The factorized approximate inverse takes the form

\[ G_n = Z_n \hat{D}_n^{-1} W_n^T. \]

The advantage of factorized sparse approximate inverse techniques over the SPAI technique in Subsection 3.5.1 is that the former can be used as preconditioners for the CG method for solving symmetric positive definite problems.

3.5.3 Practical use of approximate inverses

We conclude this section with some comments on the use of the approximate inverse preconditioning techniques. As has been mentioned before, the approximate inverses were developed to overcome breakdown of preconditioners such as the ILU(0). As such the setting in which approximate inverses are presented in literature is rather practical and implementation oriented. In [15] several tests were done to compare the ILU(0) and the SPAI and AINV as preconditioner. The experiments were performed on sparse matrices, mostly from the Harwell-Boeing collection. These non-symmetric matrices originate from a wide range of applications such as oil reservoir simulation and circuit design. Several observations can be made. AINV and SPAI fail for a few test problems, but for the remainder of the tests the robustness of the AINV and SPAI is comparable with the ILU(0). Rates of convergence are also comparable; AINV seems to perform a bit better than SPAI. The time for the iterative part however, shows that SPAI and AINV are much better than ILU(0) due to the good vectorization properties of the approximate inverses. The price to pay is the expensive construction time for SPAI and AINV with the first one being even much more expensive than the latter. It should be noted that SPAI was designed for parallel implementations which has not been taken into account in these tests. For other tests in different settings see for instance [13, 14, 16, 34].
3.6 Diagonal approximate inverses

In the discussion of the choice for $S$, we mentioned that the sparsity pattern of $A_n$ is a common choice, or alternatively that of $A_p^n$ for $p \geq 2$. It is also possible to construct a diagonal approximate inverse, which, in fact, inherits the sparsity pattern of $A_0^n$. Let this diagonal approximate inverse be denoted by $D_n$. For right preconditioning we observe that the columns of $A_n D_n$ are scaled versions of the columns of $A_n$ since $[A_n D_n]_{i,j} = [A_n]_{i,j} [D_n]_{j,j}$. Similarly, for the left preconditioned system $D_n A_n$ the rows are scaled. With such scalings it is possible to lower the condition number of $A_n$. This is called equilibration and has been studied in considerable detail in [10, 98, 100, 99].

From [98] we derive several results on what can be obtained optimally with equilibration: Let $k \cdot k_p$ be any Hölder norm, $p > 0$ or $p = 1$, or the Frobenius norm.

For the left preconditioned linear system we then have

**Theorem 3.7.** Let $\kappa(A_n) = \|A_n\|_\infty \|A_n^{-1}\|_p$. Then for a diagonal matrix $D_n$, $\kappa(D_n A_n)$ is minimal if in $D_n A_n$ all rows have identical 1-norm.

For the right preconditioned case:

**Theorem 3.8.** Let $\kappa(A_n) = \|A_n\|_1 \|A_n^{-1}\|_p$. Then $\kappa(A_n D_n)$ is minimal if in $A_n D_n$ all columns have identical 1-norm.

However, in practice we are more interested in the spectral condition number $\kappa_2$ of the matrix. Then, based on $\frac{1}{\sqrt{n}} \cdot \| \cdot \|_\infty \leq \| \cdot \|_2 \leq \sqrt{n} \cdot \| \cdot \|_\infty$, we have

**Corollary 3.9.** Let $\kappa(A_n) = \|A_n\|_2 \|A_n^{-1}\|_p$ and let $\mathcal{D}_n$ denote the set of $n \times n$ diagonal matrices. Then

$$\kappa(D_n A_n) \leq \sqrt{n} \min_{D_n \in \mathcal{D}_n} \kappa(D_n A_n)$$

if all rows in $D_n A_n$ have identical 2-norm, and

$$\kappa(A_n D_n) \leq \sqrt{n} \min_{D_n \in \mathcal{D}_n} \kappa(A_n D_n)$$

if all columns in $A_n D_n$ have identical 2-norm.

For sparse matrices that are symmetric positive definite the last result can be refined (see [98]):

**Corollary 3.10.** Let $A_n$ be symmetric positive definite and $\kappa(A_n) = \|A_n\|_2 \|A_n^{-1}\|_2$. Assume that $A_n$ has at most $m$ non-zeros in any row. Then

$$\kappa(A_n) \leq m \min_{D_n \in \mathcal{D}_n} \kappa(D_n A_n D_n)$$

if the main diagonal of $A_n$ is constant.

In [98] no information is provided about the optimality of $D_n$. These results indicate that diagonal preconditioning is most efficient when this preconditioner equilibrates the rows or columns of the given linear operator $A_n$. 
Chapter 4

The approximate inverse

In this chapter we look further into the approximate inverse technique introduced in Section 3.5. In the first section we consider a discretization for a general convection diffusion equation on a non-uniform grid. For a certain class of tridiagonal matrices, which also arise from the previously discussed discretizations, the eigenvalues and eigenvectors can be determined. This theory leads to Toeplitz matrices which are one of the few types of matrices for which the spectrum, and henceforth the spectral condition number, can be computed.

The spectral condition numbers of our preconditioned linear systems $A_nG_n$ and $G_nA_n$ are difficult to compute, since they are not symmetric. However, as it turns out, these preconditioned linear systems resemble a Toeplitz matrix. For the one dimensional case we alter the approximate inverse based preconditioner such that the products $A_nG_n$ and $G_nA_n$ are Toeplitz for the pure diffusive problems. Using Toeplitz matrix results summarized in [20] the spectral condition number can be derived. For the product $G_nA_n$ we give an estimate of the spectral condition number, since in this case the inverse is explicitly known. The spectral condition number of the other product $A_nG_n$ is estimated in an alternative way, which is described at the end of the chapter.

4.1 The convection diffusion discretization

Let $a$, $b$ and $c$ be non-negative real numbers, $g$ a function, and consider the BVP:

$$-a\frac{d^2u}{dx^2} + b\frac{du}{dx} + cu = g(x), \quad x \in \Omega = (0, 1),$$

$$u(0) = u(1) = 0.$$

Let $n$ be a positive integer, let $\psi \in (0, 1]$. Define $f := 1/\psi^{n+1}$ and $F$ the mapping

$$F: x \mapsto \frac{f(1-x) - f}{1 - f}.$$
Finally, let
\[
x_i := F(i/(n+1)) = \frac{f^{(1-i/(n+1))} - f}{1 - f}, \quad \forall i = 0, \ldots, n + 1.
\]
Then \(x_0 = 0, x_{n+1} = 1,\) and
\[
\psi = \frac{x_{i+1} - x_i}{x_i - x_{i-1}}, \quad \forall i = 1, \ldots, n.
\]
Indeed,
\[
\frac{x_{i+1} - x_i}{x_i - x_{i-1}} = \frac{f^{1-(i+1)/(n+1)} - f^{1-i/(n+1)}}{f^{1-i/(n+1)} - f^{1-(i-1)/(n+1)}} = \frac{f^{1-1/(n+1)} - 1}{1 - f^{1/(n+1)}} = \frac{\psi - 1}{1 - \psi^{-1}}
\]
\[
= \frac{\psi - 1}{\psi} \cdot \frac{\psi - 1}{\psi - 1} = \psi.
\]
Thus grid points \(x_i\) are denser for \(x_i\) close to 1. Define
\[
h_i := x_{i+1} - x_i, \quad \forall i = 0, \ldots, n,
\]
and let
\[
H := h_0 \text{ and } h := h_n,
\]
then
\[
\frac{x_{i+1} - x_i}{x_i - x_{i-1}} = \psi \Rightarrow h_i = \psi^i H, \quad \forall i = 0, \ldots, n.
\]
Thus,
\[
h = \psi^n H.
\]
If we take \(\Omega = (\alpha, \beta),\) then \(H\) can simply be found, for
\[
\beta - \alpha = \sum_{i=0}^n (x_{i+1} - x_i) = \sum_{i=0}^n h_i = \sum_{i=0}^n \psi^i H = H \cdot \frac{1 - \psi^{n+1}}{1 - \psi} \Rightarrow H = (\beta - \alpha) \frac{1 - \psi}{1 - \psi^{n+1}}.
\]
Now we proceed as follows: Let \(u_i := u(x_i).\) Using (4.3), we find
\[
x_{i-1} = x_i - h_i/\psi, \quad x_{i+1} = x_i + h_i.
\]
Using a Taylor expansion and a bit of algebra, a central difference approximation of \(u \mapsto u''\) is (see [69])
\[
\frac{2}{h_{i+1} + h_i} \left( \frac{u_{i+1} - u_i}{h_i} - \frac{u_i - u_{i-1}}{h_{i-1}} \right) \approx u''(x_i) \Rightarrow \frac{1}{h_i^2} \frac{2\psi}{1 + \psi} (\psi u(x - h_i/\psi) - (1 + \psi)u(x) + u(x + h_i)) = u''(x_i) + O(h_i).
Thus discretizing the operator \( u \mapsto -au'' \) results in the difference equation

\[
\frac{1}{h_i^2} \cdot \frac{2\psi}{1 + \psi} \cdot a \left( -\psi u_{i-1} + (1 + \psi)u_i - u_{i+1} \right).
\]

The related stencil is given by

\[
\frac{1}{h_i^2} \cdot \frac{2\psi}{1 + \psi} \cdot a \left[ -\psi, (1 + \psi), -1 \right], \quad \forall i = 1, \ldots, n.
\]

For a first order downwind discretization of \( u \mapsto bu' \) the stencil is

\[
\frac{1}{h_i} \cdot b \left[ -1, 1, 0 \right].
\]

Thus, the stencil for the finite difference operator is given by

\[
(4.6) \quad \frac{1}{h_i^2} \cdot \frac{2\psi}{1 + \psi} \cdot a \left[ -\psi, (1 + \psi), -1 \right] + \frac{1}{h_i} \cdot b \left[ -1, 1, 0 \right] + c \left[ 0, 1, 0 \right], \quad i = 1, \ldots, n.
\]

After using the boundary values for \( u_0 \) and \( u_{n+1} \) we can collect these difference equations in the usual way in a matrix \( A_n \).

### 4.2 Eigenvalues and eigenvectors of tridiagonal matrices

In this section we compute the eigenvalues and eigenvectors related to the discretization of convection and diffusion.

First, consider forward differences: Then \( A_n \in \mathbb{R}^{n \times n} \) is typically given by

\[
A_n = \begin{bmatrix}
  a_0 & a_1 & 0 & \cdots & 0 \\
  a_1 & a_0 & a_1 & \cdots & 0 \\
  0 & a_0 & a_1 & \cdots & 0 \\
  \vdots & \vdots & \ddots & \ddots & \vdots \\
  0 & 0 & \cdots & a_0 & a_1 \\
  \end{bmatrix}, \quad a_0, a_1 \in \mathbb{R}, \ a_1 \neq 0.
\]

Clearly, \( A_n \) resembles a Jordan matrix in a trivial way. Its eigenvalue is \( a_0 \) (algebraic multiplicity \( n \)), and its single eigenvector is \( e_1 := [1, 0, \ldots, 0]^T \) (geometric multiplicity 1).

Next consider backward differences: Then a typical form of \( A_n \) is

\[
A_n = \begin{bmatrix}
  a_0 & 0 & a_1 & \cdots & 0 \\
  a_1 & a_0 & 0 & \cdots & 0 \\
  \vdots & \vdots & \ddots & \ddots & \vdots \\
  a_{n-2} & a_{n-1} & a_0 & 0 & a_1 \\
  a_{n-1} & a_{n-2} & a_0 & a_1 & a_0 \\
\end{bmatrix}, \quad a_{n-1}, a_0 \in \mathbb{R}, \ a_{n-1} \neq 0.
\]
Here $A_n$ is again similar to a Jordan matrix. Its eigenvalue is $a_0$ (algebraic multiplicity $n$), and its single eigenvector is $e_n := [0, \ldots, 0, 1]^T$ (geometric multiplicity 1).

To finish with, we consider central differences for the double derivative:

For some $a_{-1}, a_0, a_1 \in \mathbb{R}$, $a_{-1} \neq 0$, $a_1 \neq 0$, and $A_n \in \mathbb{R}^{n \times n}$ typically has the form

$$A_n = \begin{bmatrix} a_0 & a_1 & & & & \\ a_{-1} & a_0 & a_1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & a_{-1} & a_0 & a_1 & \\ & & & a_{-1} & a_0 \end{bmatrix}.$$

For this tridiagonal matrix we can find the eigenvalues and eigenvectors as follows. Let $x_n = \begin{bmatrix} x_1, \ldots, x_n \end{bmatrix}^T \in \mathbb{R}^n$ be such that $A_n x_n = \lambda x_n$. Then

$$\begin{cases} a_0 x_1 + a_1 x_2 = \lambda x_1, \\
a_{-1} x_{k-1} + a_0 x_k + a_1 x_{k+1} = \lambda x_k, & k = 2, \ldots, n-1, \\
a_{-1} x_{n-1} + a_0 x_n = \lambda x_n. \\
x_0 = x_{n+1} = 0 \end{cases}$$

Introducing $x_0 = x_{n+1} = 0$, we find that $\{x_k\}_{k=0}^{n+1}$ satisfies the recurrence relation

$$\begin{cases} x_0 = 0, \\
a_{-1} x_{k-1} + (a_0 - \lambda)x_k + a_1 x_{k+1} = 0, & k = 1, \ldots, n. \\
x_{n+1} = 0. \end{cases}$$

Using this relation, we proceed as follows. The characteristic polynomial of the recursion is

$$x^2 + \frac{a_0 - \lambda}{a_{-1}} x + \frac{a_1}{a_{-1}} = 0.$$

First let the roots $\lambda_1$ and $\lambda_2$ of the characteristic polynomial be different. Then there are $\alpha_1, \alpha_2 \in \mathbb{R}$, so that the solution of (4.7) is given by

$$x_i = \alpha_1 \lambda_1^i + \alpha_2 \lambda_2^i, \quad i = 0, \ldots, n+1.$$

Using the fact that $x_0 = x_{n+1} = 0$ we find

$$x_{n+1} = \alpha_1 (\lambda_1^{n+1} - \lambda_2^{n+1}) = 0.$$

Note that $\alpha_1 = -\alpha_2 = 0$ would correspond to the trivial solution, which we therefore exclude.

Hence

$$\lambda_1^{n+1} = \lambda_2^{n+1} \Rightarrow \left(\frac{\lambda_1}{\lambda_2}\right)^{n+1} = 1 \Rightarrow \frac{\lambda_1}{\lambda_2} = e^{2\pi i \frac{n}{n+1}} = e^{\frac{2\pi i}{n+1}}.$$

Using $\lambda_1 \lambda_2 = a_1/a_{-1}$, i.e.,

$$\lambda_1^2 = \lambda_1 \lambda_2 \cdot \frac{\lambda_1}{\lambda_2} = a_1/a_{-1} \cdot e^{\frac{2\pi i}{n+1}},$$
The spectral condition number of Toeplitz matrices

we find
\[ \lambda_1 = \pm \sqrt{a_1/a_{-1} \cdot e^{\pi i}}, \]
\[ \lambda_2 = \pm \sqrt{a_1/a_{-1} \cdot e^{-\pi i}}. \]

Since
\[ \frac{a_0 - \lambda}{a_{-1}} = -(\lambda_1 + \lambda_2) = - \pm \sqrt{a_1/a_{-1} \cdot 2 \cos\left(\frac{p\pi}{n+1}\right)} \]
we obtain
\[ \lambda = a_0 \pm 2 \sqrt{a_{-1}a_1} \cos\left(\frac{p\pi}{n+1}\right). \]

If \( a_{-1}/a_1 < 0 \) we have complex eigenvalues, viz.
\[ \lambda = a_0 + 2i \cdot \sqrt{|a_{-1}a_1|} \cos\left(\frac{p\pi}{n+1}\right). \]

Finally \( \lambda_1 = \lambda_2 \) cannot occur as this would imply that \( x_i = (\alpha_1 + \alpha_2 \cdot i) \lambda_i^1 \), i.e., \( \alpha_1 = \alpha_2 = 0 \Rightarrow x = 0 \).

The above results are summarized in the following corollary.

**Corollary 4.1.** For \( a_{-1}a_1 \neq 0 \):
\[
\sigma(A_n) = \left( a_0 + 2 \sqrt{a_{-1}a_1} \cos\left(\frac{p\pi}{n+1}\right) \right)^n_{p=1} 
\subset (a_0 - 2 \sqrt{a_{-1}a_1}(1 - 2 \left(\frac{\pi}{n+1}\right)^2 + O(\frac{\pi}{n+1})^4), a_0 + 2 \sqrt{a_{-1}a_1})
\subset (a_0 - 2 \sqrt{a_{-1}a_1}, a_0 + 2 \sqrt{a_{-1}a_1}).
\]

The related eigenvectors \( x_p \) are:
\[ x_p = \left( \sqrt{\frac{a_1}{a_{-1}}} \sin\left(\frac{\pi p}{n+1}\right) \right)^{i=0} \]

### 4.3 The spectral condition number of Toeplitz matrices

In this section we summarize a number of properties for so called *Toeplitz matrices*. For a state of the art overview of properties see [20]. For Toeplitz matrices the spectral condition number can be computed with the use of the \( \| \cdot \|_\infty \) norm. This is the main result of interest in relation to the approximate inverse preconditioner, which is discussed in the next section.

**Definition 4.2.** Let \( b := (\ldots, b_{-2}, b_{-1}, b_0, b_1, b_2, \ldots)^T \in \mathbb{R}^\infty \), a real vector in \( l^1 \). The infinite matrix \( B_n \) defined by
\[
B_n = \begin{bmatrix}
  b_0 & b_1 & b_2 & \ldots \\
  b_{-1} & b_0 & b_1 & \ldots \\
  b_{-2} & b_1 & b_0 & \ldots \\
  \vdots & \vdots & \vdots & \ddots
\end{bmatrix}
\]
is called a Toeplitz matrix. The numbers \( \ldots, b_{-2}, b_{-1}, b_0, b_1, b_2, \ldots \) do not depend on \( n \).
A finite section of a Toeplitz matrix, i.e. the matrix (see [20, (2.5)])

\[
B_n = \begin{bmatrix}
    b_0 & b_1 & \ldots & b_{n-1} \\
    b_{-1} & b_0 & \ddots & \vdots \\
    \vdots & \ddots & \ddots & b_1 \\
    b_{-n+1} & \ldots & b_{-1} & b_0
\end{bmatrix}, \quad b_i \in \mathbb{R},
\]

is also called a Toeplitz matrix.

For Toeplitz matrices the spectral condition number can be computed using the \(\|\cdot\|_\infty\) norm. This is due to a special relation between the largest and smallest singular values \(\sigma_1(B_n)\) and \(\sigma_n(B_n)\) of a matrix \(B_n\) on the one hand and the values of \(\|B_n\|_\infty\) and \(\|B_n^{-1}\|_\infty\) on the other hand respectively.

For the largest singular value we have (see [20, (2.5) and Theorem 4.13])

**Theorem 4.3.** For \(b \in l^1\) the following holds

\[
\lim_{n \to \infty} \sigma_n(B_n) = \lim_{n \to \infty} \|B_n\|_\infty = \sum_{i=-\infty}^{\infty} |b_i|.
\]

There is a similar result for the smallest singular value \(\sigma_1\) (see [20, Theorem 4.3 (4.8)])

**Theorem 4.4.** If \(\sigma_1(B_n)\) is such that for all \(n \in \mathbb{N}\), \(\sigma_1(B_n) \geq \delta > 0\) then

\[
\lim_{n \to \infty} \sigma_1(B_n) = \lim_{n \to \infty} \frac{1}{\|B_n^{-1}\|_\infty}.
\]

Therefore, if \(B_n\) is Toeplitz, we have

\[
\lim_{n \to \infty} \kappa_2(B_n) = \lim_{n \to \infty} \frac{\sigma_n(B_n)}{\sigma_1(B_n)} = \frac{\lim_{n \to \infty} \sigma_n(B_n)}{\lim_{n \to \infty} \sigma_1(B_n)} = \sum_{i=-\infty}^{\infty} |b_i|/ \lim_{n \to \infty} \|B_n^{-1}\|_\infty.
\]

**Example 4.5.** The matrix \([-1, 2, -1]\) has singular values that are equal to some of its eigenvalues (see (4.8)) of which the largest one converges to 4 for \(n \to \infty\).

**Example 4.6.** In the case of no refinement (i.e. \(\psi = 1\)) we have \(h_i = H = h = 1/(n+1)\). Then the matrix \(A_n\) based on (4.6) is Toeplitz,

\[
A_n = \frac{1}{h^2} \cdot a [-1, 2, -1] + \frac{1}{h} \cdot b [-1, 1, 0] + c [0, 1, 0]
= [- \frac{a}{h^2} - \frac{b}{h}, \frac{2a}{h^2} + \frac{b}{h} + c, - \frac{a}{h^2}],
\]

for each value of \(n\). However we have \(A_n \neq A_n^T\), so \(\kappa_2(A_n)\) is not simple to compute. From (4.8) we find that

\[
\sigma(A_n) = \left(2 \frac{a}{h^2} + \frac{b}{h} + c + 2 \frac{1}{h^2} \sqrt{\left(\frac{a^2}{h^2} + \frac{ab}{h} \cos\left(\frac{p\pi}{n+1}\right)\right)}\right)_p^{n}.
\]
However,

$$\kappa_2(A_n) = \sigma_n(A_n)/\sigma_1(A_n),$$

where $\sigma_1 \leq \sigma_2 \leq \ldots \leq \sigma_n$ are the singular values of $A_n$ (positive square roots of eigenvalues of $A_nA_n^T$) are not known – because $A_n$ is not symmetric. The theorems stated above cannot be used because the entries of $A_n$ depend on $n$. For fixed $n$ and refinement ($\psi < 1$) the matrix $A_n$ does not have constant diagonals and cannot be called Toeplitz.

4.4 A preconditioner based on approximate inverses

Now consider the application of the approximate inverse technique using Frobenius norm minimization that was introduced in Subsection 3.5.1. This technique calculates a preconditioner $G_n$ such that $A_nG_n$ approximates $I_n$ given a certain sparsity pattern $S$ that $G_n$ should satisfy. First we describe computation of $G_n$ for a general sparsity pattern, and introduce necessary notations. Then, for the convection diffusion operator we examine the structure of $G_n$ in detail where $G_n$ inherits the sparsity structure of $A_n$. For the pure diffusion case $A_nG_n$ is almost Toeplitz. We compute a sparse matrix $G_n$ such that $A_nG_n$ approximates $I_n$ and is Toeplitz. For this product we calculate the spectrum and spectral condition number.

Let $S$ be a given sparsity pattern. Then in order to compute the columns of $G_n$ we have to solve

$$[A_n g_i]_j = [e_i]_j, \ (j,i) \in S, \ i = 1, \ldots, n.$$ 

Let $S_i$ denote the sparsity pattern of $g_i$, i.e.,

$$S_i := \{ j : (j,i) \in S \},$$

and let its elements be numbered as $j_1, \ldots, j_{m_i}$. Let $A_{(i)}$ be an $m_i \times m_i$ sub-matrix of $A_n$, defined by

$$[A_{(i)}]_{p,q} := [A_n]_{j_p,j_q}, \ p,q = 1, \ldots, m_i,$$

and likewise let $e_{(i)}$ be the unit vector of dimension $m_i$ given by

$$[e_{(i)}]_p := \delta_{j_p,i}, \ p = 1, \ldots, m_i.$$

Then solving the large $n \times n$ linear system reduces to solving the $m_i \times m_i$ linear system

$$A_{(i)} g_{(i)} = e_{(i)}, \ i = 1, \ldots, n.$$ 

Provided that $A_{(i)}$ is non-singular we thus obtain a vector $g_{(i)}$. This shows that if $i \notin S_i$, $e_{(i)} = 0$ and hence $g_i = 0$ as well. So we may assume that $i \in S_i$.

From $g_{(i)}$ we construct $g_i$ via

$$[g_i]_{j_p} = \begin{cases} [g_{(i)}]_p & \text{if } j_p \in S_i, \\ 0 & \text{otherwise}. \end{cases}$$
Finally, the approximate inverse, the Green’s matrix, $G_n$ is constructed with the $g_i$ as its columns, and $G_n$ satisfied the prescribed sparsity pattern $S$.

In order to compute the approximate inverse of (4.6) we take

$$S = \{(i, j) : |A_{ij}| \neq 0\}.$$

Since $A_n$ is tridiagonal, we have to consider only the equations for $x_i, x_{i-1}$ and $x_{i+1}$. For computing the $i$th column of $G_n$ we have to solve $A_i g_i = e_2$. Here $A_{i}$ is the $3 \times 3$ sub-matrix of $A_n$, and $g_i$ a vector related to node $i$ as explained in Section 3.5. Since $h_{i-1} = h_i/\psi$ and $h_{i+1} = h_i\psi$ we have to solve a system dependent on $h_i$ only:

$$\begin{bmatrix}
    c + \frac{2a\psi^3}{h_i} & \frac{-2a\psi^3}{(1+\psi)h_i} & 0 \\
    \frac{2a\psi^3}{h_i} & c + \frac{2a\psi^3}{(1+\psi)h_i} & \frac{-2a\psi}{h_i} \\
    0 & \frac{-2a\psi}{(1+\psi)h_i} & c + \frac{2a\psi}{h_i} + \frac{b}{h_i}
\end{bmatrix} g_i = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}.$$

Except for $i = 1$ and $i = n$ we find

$$g_i = \frac{2a\psi^2(2a+h_i(b+c\psi h_i))}{(1+\psi)h_i^4} \begin{bmatrix} 2a\psi^3+h_i(2a+h_i(b+c\psi h_i)) \\ (2a\psi+b(1+\psi)h_i)(2a\psi^3+h_i(b+c\psi h_i)) \\ \psi(1+\psi)h_i^4 \end{bmatrix} / \det(A_i).$$

The vector $g_{(1)}$ related to node 1 is found by solving the $2 \times 2$ linear system of equations

$$\begin{bmatrix}
    c + \frac{2a\psi^3}{h_i^2} & \frac{-2a\psi^3}{(1+\psi)h_i^2} \\
    \frac{-2a\psi^3}{(1+\psi)h_i^2} & c + \frac{2a\psi^3}{(1+\psi)h_i^2} + \frac{b}{h_i}
\end{bmatrix} g_{(1)} = \begin{bmatrix} 1 \\ 0 \end{bmatrix},$$

and likewise for $g_{(n)}$ related to node $n$ we have to solve

$$\begin{bmatrix}
    c + \frac{2a\psi^3}{h_n^2} & \frac{-2a\psi^3}{(1+\psi)h_n^2} \\
    \frac{-2a\psi^3}{(1+\psi)h_n^2} & c + \frac{2a\psi^3}{(1+\psi)h_n^2} + \frac{b}{h_n}
\end{bmatrix} g_{(n)} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

We find

$$g_{(1)} = \frac{1}{\det(A_{(1)})} \begin{bmatrix} c + \frac{2a\psi^3}{h_1} + \frac{b}{\psi h_1} \\ \frac{2a\psi^3}{(1+\psi)h_1^2} + \frac{b}{\psi h_1} \end{bmatrix}, \text{ and } g_{(n)} = \frac{1}{\det(A_{(n)})} \begin{bmatrix} c + \frac{2a\psi^3}{h_n} + \frac{b}{\psi h_n} \\ \frac{2a\psi^3}{(1+\psi)h_n^2} + \frac{b}{\psi h_n} \end{bmatrix}.$$

We are interested in the matrix $A_n G_n$, a five diagonal matrix. In the following we have a more detailed look at its entries. The $i$th column of $A_n G_n$ is computed.
through with entries:

\[
\begin{bmatrix}
\vdots & \vdots \\
\vdots & a_{i-2,i-2} & a_{i-2,i-1} & 0 & 0 & 0 \\
a_{i-1,i-2} & a_{i-1,i-1} & a_{i-1,i} & 0 & 0 & 0 \\
0 & a_{i,i-1} & a_{i,i} & a_{i,i+1} & 0 & 0 \\
0 & 0 & a_{i+1,i} & a_{i+1,i+1} & a_{i+1,i+2} & \vdots \\
0 & 0 & 0 & a_{i+2,i+1} & a_{i+2,i+2} & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots 
\end{bmatrix}
\begin{bmatrix}
\vdots \\
g_{(i),1} \\
g_{(i),2} \\
g_{(i),3} \\
g_{(i),4} \\
\vdots 
\end{bmatrix}
= 
\begin{bmatrix}
\vdots \\
a_{i-2,i-1} \cdot g_{(i),1} \\
a_{i-1,i} \cdot g_{(i),1} \\
a_{i,i+1} \cdot g_{(i),1} \\
a_{i+2,i+2} \cdot g_{(i),1} \\
\vdots 
\end{bmatrix}.
\]

The upper non-zero co-diagonal is found to be

\[
a_{i-2,i-1} \cdot g_{(i),1} = \frac{-2 a \psi^5}{(1 + \psi) h_i^2} g_{(i),1}
\]

\[
= \frac{-2 a \psi^5}{(1 + \psi) h_i^2} \frac{2 a \psi^2}{(1 + \psi) h_i^4} (2 a + h_i (b + c \psi h_i)) / \det(A_{(i)}).
\]

In order to simplify the calculations, consider the purely diffusive problem, i.e. $b = c = 0$. Then

\[
\det(A_{(i)}) = \frac{8a^3 \psi^3 (1 + \psi^2)}{h_i^6 (1 + \psi)^2}, \quad i = 2, \ldots, n - 1,
\]

and for $i = 1$ and $i = n$

\[
\det(A_{(1)}) = \frac{4a^2 (1 + \psi + \psi^2)}{h_1^4 (1 + \psi)^2}, \quad \det(A_{(n)}) = \frac{4a^2 \psi^4 (1 + \psi + \psi^2)}{h_n^4 (1 + \psi)^2}.
\]

We find

\[(4.11) \quad g_{(i)} = h_i^2 \cdot \frac{1 + \psi}{2a(1 + \psi^2)} \left[ \frac{1/\psi}{1 + \psi}/\psi \right], \quad i = 2, \ldots, n - 1,
\]

and

\[
g_{(1)} = h_1^2 \cdot \frac{1 + \psi}{2(1 + \psi + \psi^2)} \left[ \frac{(1 + \psi)/\psi}{1} \right], \quad \quad g_{(n)} = h_n^2 \cdot \frac{1 + \psi}{2(1 + \psi + \psi^2)} \left[ \frac{1/\psi}{1 + \psi}/\psi \right].
\]

Then we find for the upper co-diagonal

\[
a_{i-2,i-1} \cdot g_{(i),1} = \frac{-2 a \psi^5}{(1 + \psi) h_i^2} g_1 = \frac{-2 a \psi^5}{(1 + \psi) h_i^2} \cdot \frac{1 + \psi}{2a(1 + \psi^2)} \cdot 1/\psi = -\psi^4/(1 + \psi^2),
\]

and likewise for the lower co-diagonal

\[
a_{i+2,i+1} \cdot g_{(i),3} = \frac{-2 a}{(1 + \psi) \psi^2 h_i^2} \cdot \frac{1 + \psi}{2a(1 + \psi^2)} = -\frac{1}{\psi^2(1 + \psi^2)}.
\]
Chapter 4: The approximate inverse

Note that these representations are independent of $h_i$.

Except for the left upper and right lower corners, $A_nG_n$ is a Toeplitz matrix. Therefore, let $B^r_n$ be a Toeplitz matrix defined by

$$B^r_n := \begin{bmatrix} \frac{1}{\psi^2(1 + \psi^2)} & 0, 1, 0, -\frac{\psi^4}{1 + \psi^2} \end{bmatrix}.$$  \hfill (4.12)

Then $B^r_n$ approximates $A_nG_n$ and is a Toeplitz matrix for each $n$. For the spectrum of $B^r_n$, we have from (4.8)

$$\sigma(B^r_n) \subset (1 - 2\sqrt{\frac{\psi^4}{(1 + \psi^2)^2\psi^2}}, 1 + 2\sqrt{\frac{\psi^4}{(1 + \psi^2)^2\psi^2}}) = (1 - 2\frac{\psi}{1 + \psi^2}, 1 + 2\frac{\psi}{1 + \psi^2}).$$

A similar analysis as carried out for the product $A_nG_n$ can be performed for the $G_nA_n$. To this end define a Toeplitz matrix $B^l_n$ resembling $G_nA_n$ except for the corners. We obtain

$$B^l_n := \begin{bmatrix} \frac{\psi^2}{1 + \psi^2} & 0, 1, 0, -\frac{1}{1 + \psi^2} \end{bmatrix},$$  \hfill (4.13)

and

$$\sigma(B^l_n) \subset (1 - 2\sqrt{\frac{\psi^2}{(1 + \psi^2)^2\psi^2}}, 1 + 2\sqrt{\frac{\psi^2}{(1 + \psi^2)^2\psi^2}}) = (1 - 2\frac{\psi}{1 + \psi^2}, 1 + 2\frac{\psi}{1 + \psi^2}).$$

We shall illustrate the approximate inverses and the connection to the Toeplitz matrices by two examples.

**Example 4.7.** Let $a = 1, b = c = 0$, and $\psi = 1$. In this example we look for a matrix $G_n$ such that $A_nG_n$ is Toeplitz, which means it satisfies (4.12). So,

$$B^r_n = A_nG_n = \begin{bmatrix} 1 & 0 & -\frac{1}{2} \\ 0 & 1 & \ddots \\ -\frac{1}{2} & \ddots & \ddots & \ddots \\ & \ddots & 1 & 0 \\ -\frac{1}{2} & 0 & 1 & \end{bmatrix}.$$  \hfill (4.12)

Note that $A_nG_n$ is the discretization of the Laplace operator on a coarser grid. The degrees of freedom decouple after right multiplication by $G_n$, as is witnessed by the zeros on the first lower and upper diagonal. Furthermore, the two resulting matrices have a structure like $A_n$.

For $G_n$ we find
A preconditioner based on approximate inverses

\[ \bar{G}_n = \frac{1}{2n} \begin{bmatrix} \ast & 1 & \ast \\ \ast & 2 & \ddots & \vdots \\ \ast & 1 & \ddots & \ddots & 1 & \ast \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\ \ast & 1 & \ast \\ \end{bmatrix} \]

which is similar to \( G_n \) except for full first and last columns. The first column of \( \bar{G}_n \) is given by

\[ [\bar{G}_n]_{i,1} = \begin{cases} i(\frac{1}{n} - 1) + 2 & \text{for } i \leq 3, \\ \frac{i}{n} - 1 & \text{for } i > 3 \end{cases} \]

The last column of \( \bar{G}_n \) is the reverse of the first column. We see in this example that a relatively easy modification of the approximate inverse \( G_n \) results in a Toeplitz product \( A_n \bar{G}_n \).

In the next example we examine the situation for a refined grid.

**Example 4.8.** Let \( a = 1, b = c = 0, \) and \( \psi = 1/2 \). Then equation \( i \) \((i = 1, \ldots, n)\) reads (see (4.6))

\[
\frac{\psi}{1 + \psi} \begin{bmatrix} -\psi, (1 + \psi), -1 \end{bmatrix} = \frac{1}{H^2} \begin{bmatrix} 2i \frac{1}{(\psi^2)^i} \end{bmatrix} \begin{bmatrix} -\psi, (1 + \psi), -1 \end{bmatrix} = \frac{1}{H^2} 4^i \begin{bmatrix} 2i \frac{1}{3} \begin{bmatrix} -\frac{1}{2}, -\frac{2}{3}, -1 \end{bmatrix} = \frac{1}{H^2} 4^i \begin{bmatrix} -\frac{1}{3}, 1, -\frac{2}{3} \end{bmatrix}. \]

Thus for \( A_n \) we have

\[
A_n = \frac{4}{H^2} \begin{bmatrix} 1 & -\frac{2}{3} & \ast \\ -\frac{4}{3} & 4 & -\frac{8}{3} \\ \vdots & \vdots & \ddots & \ddots & \ddots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots \\ \frac{1}{3} & \frac{1}{3} 4^{n-1} & 4^{n-2} & \frac{2}{3} \end{bmatrix} \in \mathbb{R}^{n \times n}.
\]
Chapter 4: The approximate inverse

The related approximate inverse is readily constructed from (4.11),

\[ G_n = \frac{H^2}{4}. \]

\[ \begin{bmatrix}
\frac{9}{7} & \frac{6}{20} & \cdots \\
\frac{3}{7} & \frac{9}{20} & \cdots \\
\vdots & \ddots & \ddots \\
\frac{3}{80} & \cdots & \frac{6}{20+4^{n-1}} \\
\vdots & \ddots & \ddots \\
\frac{9}{56+2^n} & \cdots & \frac{3}{20+4^{n-1}} \\
\frac{3}{56+2^n} & \cdots & \frac{9}{20+4^{n-1}} \\
\end{bmatrix} \in \mathbb{R}^{n \times n}. \]

Note that the first and last column have a deviating structure. The two related preconditioned linear systems \( A_n G_n \) and \( G_n A_n \) are explicitly given by

\[ (4.16) \quad A_n G_n = \begin{bmatrix}
1 & 0 & -\frac{1}{20} \\
0 & 1 & 0 & \ddots \\
-\frac{48}{21} & 0 & \ddots & -\frac{1}{20} \\
-\frac{32}{10} & \ddots & 0 & -\frac{1}{28} \\
\vdots & \ddots & 1 & 0 \\
-\frac{32}{10} & 0 & 1 \\
\end{bmatrix}, \]

and

\[ (4.17) \quad G_n A_n = \begin{bmatrix}
\frac{31}{35} & \frac{12}{35} & -\frac{4}{5} \\
-\frac{6}{35} & \frac{39}{35} & 0 & \ddots \\
-\frac{1}{5} & 0 & 1 & -\frac{4}{5} & \ddots \\
-\frac{1}{5} & 1 & 0 & -\frac{4}{5} \\
\vdots & \ddots & 0 & \frac{39}{35} & -\frac{12}{35} \\
-\frac{1}{5} & \frac{6}{35} & \frac{31}{35} & \frac{39}{35} \end{bmatrix}. \]

It is easily seen these two products are almost Toeplitz.
A preconditioner based on approximate inverses

Related to the first product (4.16) is the Toeplitz matrix

$$B_n^r = \begin{bmatrix}
1 & 0 & -\frac{1}{20} \\
0 & 1 & 0 & \ddots \\
-\frac{33}{10} & 0 & \ddots & -\frac{1}{20} \\
-\frac{33}{10} & \ddots & \ddots & \ddots \\
\ddots & 1 & 0 & \ddots \\
-\frac{33}{10} & 0 & 1 & \ddots \\
\end{bmatrix}.$$ 

The modified approximate inverse $G_n$ such that $B_n^r = A_n G_n$ is found to be

$$G_n = \frac{H^2}{4}.$$ 

Here we omit the explicit computation of the first and last column which is a very technical calculation (cf. (4.14) in the previous example).

Related to the other product in (4.17) is the Toeplitz matrix

$$B_n^l = \begin{bmatrix}
1 & 0 & -\frac{4}{5} \\
0 & 1 & 0 & \ddots \\
-\frac{4}{5} & 0 & \ddots & -\frac{4}{5} \\
-\frac{4}{5} & 1 & \ddots & -\frac{4}{5} \\
\ddots & 1 & 0 & \ddots \\
-\frac{4}{5} & 0 & 1 & \ddots \\
\end{bmatrix}.$$
The matrix $G_n$ such that (4.19) holds is given by

$$G_n = \frac{H^2}{4} \cdot \begin{bmatrix}
* & * & * & \ldots & * \\
\frac{3}{5} & \frac{9}{20} & \frac{6}{80} & & \\
 & \frac{3}{20} & \frac{9}{80} & \ddots & \\
 & & \frac{3}{80} & \ddots & \frac{6}{20n+4} \\
& & & \ddots & \frac{9}{20n+4} \\
* & \ldots & * & \ldots & * \\
\end{bmatrix}.$$  

For the first and last rows the same remark can be made as above for (4.18). $B_l^n$ cannot be the discretization of a convection diffusion operator on a uniform grid, as can be verified by some algebra.

To finish this example, we look at the spectrum of $B_l^n$. For simplicity we assume $n$ to be even. A close observation of (4.19) shows that the degrees of freedom decouple into the odd and even numbered ones. This decoupling is the subject of Chapter 5, so we will only use this observation. The degrees of freedom can be permuted according to this decoupling, and $B_l^n$ is then rewritten in permuted form as

$$B_l^n = \begin{bmatrix} B_l^{n/2} \\
B_l^{n/2} \end{bmatrix}$$

where

$$B_l^{n/2} = \begin{bmatrix}
1 & -\frac{4}{5} \\
-\frac{1}{5} & 1 & \ddots \\
& \ddots & \ddots & -\frac{4}{5} \\
& & -\frac{1}{5} & 1 \\
\end{bmatrix}.$$  

Using (4.8) the eigenvalues of this $n/2 \times n/2$ matrix are

$$\sigma(B_l^{n/2}) = \left(1 + \frac{4}{5} \cos\left(\frac{i\pi}{n+1}\right)\right)^{n/2} \cup \left(\frac{1}{5}, \frac{9}{5}\right),$$

where the interval does not depend on $n$ or $\frac{H}{n}$. It is easily seen that $\sigma(B_l^n) = \sigma(B_l^{n/2})$ and that each eigenvalue has algebraic multiplicity 2. However, boundedness of the spectrum does not mean that the condition number is $O(1)$ for $\psi \downarrow 0$, since $B_l^n$ is not symmetric. Because in general the worst case estimate for the number of iterations depends on the spectral condition number, we calculate this number for matrix $B_l^n$ in the next section.
4.5 The spectral condition number related to a refined grid

In the previous section we found that the preconditioned linear system resembles a Toeplitz matrix $B_n$. However, since both $B_l^n$ and $B_r^n$ are non-symmetric the spectral condition number cannot easily be estimated. In [69] it is shown that with the use of Green’s functions certain classes of tridiagonal matrices have inverses that can be computed explicitly. This comes in handy as the relation (4.10) requires an explicit inverse. Note that the calculation of $\kappa_2(B_l^n)$ and $\kappa_2(B_r^n)$ require different approaches, because of different matrix properties.

First we summarize a result from [69]. Let $\Omega = (0, 1)$ be partitioned as $0 = x_0 < x_1 < \ldots < x_{n+1} = 1$. Let $h_i = x_i - x_{i-1}$. Assume that

$$u \mapsto -\frac{d^2u}{dx^2},$$

is discretized as

$$\frac{2h_{i+1}}{h_{i+1} + h_i}u_{i-1} + 2u_i - \frac{2h_i}{h_{i+1} + h_i}u_{i+1} = h_i h_{i+1} f(x_i, u(x_i), u'(x_i)).$$

Let $A_n \in \mathbb{R}^{n \times n}$ be the matrix

$$A_n = \left[ \begin{array}{cc} -\frac{2h_{i+1}}{h_{i+1} + h_i}, & 2, \frac{2h_i}{h_{i+1} + h_i} \end{array} \right].$$

Then (see [69, Corr. 5.5])

**Property 4.9.** For the matrix $A_n$ defined as above, its inverse is explicitly given by

$$[A_n^{-1}]_{i,j} = \begin{cases} \frac{h_{j+1} + h_j}{2h_{j+1}h_j} (1 - x_j)x_i, & i \leq j \\ \frac{h_{j+1} + h_j}{2h_{j+1}h_j} x_j (1 - x_i), & i \geq j \end{cases}.$$

We will use this result to compute the spectral condition number of $B_l^n$, without convection, related to a refined grid with refinement factor $\psi$. Since the above result holds for tridiagonal matrices, we assume that $B_l^n$ has been permuted and partitioned, i.e.,

$$B_l^n = \left[ \begin{array}{cc} B_{n/2}^1 & \cdot \\ \cdot & B_{n/2}^1 \end{array} \right].$$

So in the following we estimate the spectral condition condition for $B_{n/2}^1$, which, of course, is equal to that of $B_l^n$ itself. From the construction of $B_l^n$ in (4.13), there exist $b_{-1}$ and $b_1$ such that

$$B_{n/2}^1 = \left[ \begin{array}{c} b_{-1}/2, 1, b_1/2 \end{array} \right].$$

Then for $(B_{n/2}^1)^{-1}$ we have
Property 4.10. Let $\psi \leq 1$ and $\Psi := \psi^2$. Then $B_{n/2}^1$ is of the form (4.22), and

$$[(B_{n/2}^1)^{-1}]_{i,j} = \begin{cases} 
\Psi^{-j}(1 + \Psi)(1 + \Psi + \ldots + \Psi^{n/2}) & (1 - \tilde{x}_j)\tilde{x}_i, \quad i \leq j \\
\Psi^{-j}(1 + \Psi)(1 + \Psi + \ldots + \Psi^{n/2}) & \tilde{x}_j(1 - \tilde{x}_i), \quad i \geq j.
\end{cases}$$

Proof. First we show that $B_{n/2}^1$ is of a similar form as $A_n$. In particular we should have

$$b_{-1} = -\frac{2h_{i+1}}{h_{i+1} + h_i} \quad \text{and} \quad b_1 = -\frac{2h_i}{h_{i+1} + h_i},$$

cf. (4.22). For ease of writing we omit the tilde from now on. The required form in (4.25) holds if

$$\frac{b_{-1}}{b_1} = \frac{h_{i+1}}{h_i}.$$ 

In our case

$$b_{-1} = -\frac{2\psi^2}{1 + \psi^2}, \quad b_1 = -\frac{2}{1 + \psi^2} \Rightarrow \frac{b_{-1}}{b_1} = \frac{-2\psi^2}{\frac{1 + \psi^2}{2}} = \psi^2.$$ 

Let $\Psi := \psi^2$ and $m := n/2$. Because

$$1 = \sum_{i=1}^{m+1} h_i = h_1 \sum_{i=0}^{m} (\Psi)^i = h_1 \frac{1 - (\Psi)^{m+1}}{1 - \Psi},$$

we find

$$h_1 = \frac{1 - \Psi}{1 - (\Psi)^{m+1}},$$

and

$$h_i = \frac{1 - \Psi}{1 - \Psi^{m+1}} \Psi^{i-1}.$$ 

Combining this with the identity $\tilde{x}_i = \sum_{j=1}^{i} h_j$ we find

$$\tilde{x}_i = \frac{1 - \Psi}{1 - \Psi^{m+1}} \sum_{j=1}^{i} \Psi^{j-1} = \frac{1 - \Psi}{1 - \Psi^{m+1}} \frac{1 - \Psi^i}{1 - \Psi} = \frac{1 - \Psi^i}{1 - \Psi^{m+1}}.$$ 

For $i = m + 1$ we get $\tilde{x}_{m+1} = 1$. Having acquired expressions for $h_i$ and $\tilde{x}_i$, we use (4.23) to compute $(B_{n/2}^1)^{-1}$, and this completes the proof. □

In order to estimate $\|(B_{n/2}^1)^{-1}\|_\infty$ we have to look at the row sums. Once more let $m := n/2$. For row sum $r_i$ of the $i$th row we find

$$r_i = \frac{1 + \Psi}{1 - \Psi - \Psi^{m+1} + \Psi^{m+2}((m + 1)(1 - \Psi^i) + i(\Psi^{m+1} - 1)).}$$
Now we can estimate $\max_i r_i$. For the leading term (which is independent of $i$) we see
\[
\lim_{n \to \infty} \frac{1 + \Psi}{1 - \Psi - \Psi^{m+1} + \Psi^{m+2}} = \frac{1 + \Psi}{1 - \Psi}.
\]
The second major part gives
\[
(m + 1)(1 - \Psi^i) + i(\Psi^m - 1) \leq (m + 1)(1 - \Psi^i) \leq m + 1.
\]
So we have
\[
\|(B_{n/2}^i)^{-1}\|_\infty \leq \frac{1 + \Psi}{1 - \Psi} \left( \frac{n}{2} + 1 \right).
\]
For the initial Toeplitz matrix $B_n^i$ we summarize the results in the following corollary:

**Corollary 4.11.**
\[
\|(B_n^i)^{-1}\|_\infty \leq \frac{1 + \Psi}{1 - \Psi} \left( \frac{n}{2} + 1 \right).
\]

**Proof.** Because of the partitioning of $B_n^i$ in (4.24) we have
\[
\|(B_n^i)^{-1}\|_\infty = \|(B_{n/2}^i)^{-1}\|_\infty.
\]
This completes the proof.

Having established a bound for $\|(B_n^i)^{-1}\|_\infty$ we can estimate the spectral condition number of $B_n^i$.

**Property 4.12.**
\[
(4.26) \quad \kappa_2(B_n^i) = O(n), \quad n \to \infty.
\]

**Proof.** We have $\kappa_2(B_n^i) = \|B_n^i\|_\infty \|(B_n^i)^{-1}\|_\infty$. It is readily seen that $\|B_n^i\|_\infty = 2$, and this combined with the previous result $\|(B_n^i)^{-1}\|_\infty = O(n)$ the result in (4.26) is proven.

In Chapter 7 we give some examples to demonstrate that this estimate is sharp for $n \to \infty$.

The above theory is only valid in situations where all row sums are zero (except for the corner points). Therefore, because $B_n^i_t$ does not satisfy this condition we shall investigate a possible alternative in order to obtain estimates for the spectral condition number. Assume that $B_n^i$ is permuted and partitioned like (4.24), i.e.,
\[
\tilde{B}_n^i = \begin{bmatrix} B_{n/2}^i & B_{n/2}^i \end{bmatrix}.
\]
Here $B_{n/2}^r$ is tridiagonal with entries given by
\[ B_{n/2}^r = \left[ -\frac{1}{\psi^2(1+\psi^2)}, 1, -\frac{\psi^4}{1+\psi^2} \right]. \]

Let $D_{n/2}$ be a diagonal matrix defined by
\[ [D_{n/2}]_{i,i} := \left( \frac{1}{\psi^2} \right)^{i-1}, \quad i = 1, \ldots, n/2. \]

Then we have
\[ D_{n/2}^{-1} B_{n/2}^r D_{n/2} = (B_{n/2}^l)^T, \]
and thus
\[ B_{n/2}^r = D_{n/2} (B_{n/2}^l)^T D_{n/2}^{-1}. \]

Now we can find an estimate for $\kappa_2(B_{n/2}^r)$, and henceforth for $\kappa_2(B_{n/2}^r)$. Using the result $\kappa_2(B_{n/2}^l) = O(n/2)$ and $\kappa_2(D_{n/2}) = \psi^{-n}$, then the following bounds for $\kappa_2(B_{n/2}^r)$ are easily established:
\[ 1 \leq \kappa_2(B_{n/2}^r) \leq O\left( \frac{n}{2} \left( \frac{1}{\psi^2} \right)^n \right), \quad n \to \infty. \]

It should be noted that the above estimate is very rough. From numerical results it seems more likely that
\[ \kappa_2(B_{n/2}^r) \sim O \left( \frac{1}{\psi} \right)^n, \quad n \to \infty. \]

In this chapter we gave an estimate for the spectral condition number of the preconditioned system based on a modified approximate inverse. The results and observations in this chapter are for the pure diffusion problem only. However, intuitively these results are valid for more general cases as well, which is demonstrated by numerical examples in Chapter 7.

To finish this chapter we give an example to exemplify the estimates found.

**Example 4.13.** Let $a = 1, b = c = 0$, and $\psi = 1/2$, cf. Example 4.8. Then we look at tridiagonal matrices $B_n^r$ and $B_n^l$ with entries as in (4.12) and (4.13), i.e.,
\[ B_n^r = \left[ -\frac{32}{10}, 1, -\frac{1}{20} \right] \]
and
\[ B_n^l = \left[ -\frac{1}{5}, 1, -\frac{4}{5} \right]. \]
The spectral condition number related to a refined grid

Furthermore, let $D_n$ be a diagonal matrix defined by

$$[D_n]_{i,i} := 4^{i-1}.$$  

Then, using 4.27, $B_n^r = D_n(B_n^l)^T D_n^{-1}$.

The spectral condition numbers for $B_n^r$, $B_n^l$ and $D_n$ are given below in table 4.1.

<table>
<thead>
<tr>
<th>$n$</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\kappa_2(B_n^r)$</td>
<td>1086.09</td>
<td>4390.73</td>
<td>1.77$\cdot$10^4</td>
<td>7.10$\cdot$10^4</td>
<td>2.85$\cdot$10^5</td>
<td>1.14$\cdot$10^6</td>
<td>4.58$\cdot$10^6</td>
</tr>
<tr>
<td>$\kappa_2(B_n^l)$</td>
<td>9.20</td>
<td>11.38</td>
<td>13.55</td>
<td>15.71</td>
<td>17.87</td>
<td>20.03</td>
<td>22.18</td>
</tr>
<tr>
<td>$\kappa_2(D_n)$</td>
<td>$4^4$</td>
<td>$4^5$</td>
<td>$4^6$</td>
<td>$4^7$</td>
<td>$4^8$</td>
<td>$4^9$</td>
<td>$4^{10}$</td>
</tr>
<tr>
<td>$\kappa_2(B_n^r)/\kappa_2(D_n)$</td>
<td>4.24</td>
<td>4.29</td>
<td>4.32</td>
<td>4.34</td>
<td>4.35</td>
<td>4.36</td>
<td>4.37</td>
</tr>
</tbody>
</table>

Table 4.1: Condition numbers for $\kappa_2(B_n^r)$, $\kappa_2(B_n^l)$ and $\kappa_2(D_n)$.

From this table we see that the results for $\kappa_2(B_n^l)$ are according to (4.26). Another observation is that $\kappa_2(B_n^r)$ resembles the estimate in (4.29).
Chapter 5

Recursive solution methods

In this chapter we examine the application of the approximate inverse preconditioner for convection diffusion problems as introduced in Chapter 4. In particular we look at the decoupling mentioned in Example 4.8 and (4.24) in more detail.

In the first part of this chapter we consider a one dimensional setting. Using characteristics of the approximate inverse $G_n$, we show that the preconditioned linear operator $A_nG_n$ decouples into two submatrices. On the resulting subproblems, the same decoupling can be obtained using the approximate inverse preconditioner again. This repeated decoupling gives rise to a recursive solution algorithm. The computational complexity of this algorithm is shown to be $O(n \log(n))$ with $n$ the number of degrees of freedom on the finest grid.

Secondly we examine the approximate inverses in two dimensions. Though the construction of the approximate inverse $G_n$ is similar to the 1-D case, the final result is somewhat different. The main difference is that the decoupling depends on the type of operator, as well as on the grid topology (tensor grid, grid of triangles etc.). For a discretization of the Laplace operator on a uniform grid an interesting observation is made. Even if there is no repeated decoupling, repeated application of approximate inverses seems to lead to approximations of the Laplace operator with a larger support.

5.1 Decoupling in one dimension

Consider problem (3.1). We discretize this by a finite element method with linear basis functions, or a finite difference method resulting in a three point finite difference stencil. The corresponding $n \times n$ matrix $A_n$ is tridiagonal.

Let $\mathcal{S}$ denote the sparsity pattern of $A_n$ (cf. Section 3.5), i.e.,

$$\mathcal{S} := \{(i, j) : [A_n]_{i,j} \neq 0\}. \quad (5.1)$$

Because $A_n$ is symmetric, $(i, j) \in \mathcal{S}$ implies $(j, i) \in \mathcal{S}$. The sparsity pattern of $A_n^2$, denoted by $\mathcal{S}^2$, is then given by

$$\mathcal{S}^2 = \{(i, j) : \exists k : (i, k) \in \mathcal{S} \land (j, k) \in \mathcal{S}\}. \quad (5.2)$$
This is easily seen since $[A^2_{n}]_{i,j} = \sum_k a_{ik}a_{kj}$ and the observation that $a_{ik}a_{kj}$ is non-zero only if $i - 2 \leq j \leq i + 2$. Since the approximate inverse $G_n$ is constructed with sparsity pattern $S$, the sparsity pattern of $A_nG_n$, say $\bar{S}$, will be a subset of $S^2$. Because $A_nG_n$ satisfies

$$(5.3) \quad [A_nG_n]_{i,j} = \delta_{i,j} \text{ for } (i,j) \in S,$$

we have

$$\bar{S} = \{(i,j) : j = i - 2 \vee j = i + 2 \vee j = i, 1 \leq j \leq n\}.$$ From this we see that the set $\{1, \ldots, n\}$ decouples into sets $\hat{S}_1$ and $\hat{S}_2$ defined by

$$\hat{S}_1 = \{i : i = 2k + 1, 0 \leq k \leq \left\lceil \frac{n}{2} \right\rceil\}$$

and

$$\hat{S}_2 = \{i : i = 2k, 1 \leq k \leq \left\lfloor \frac{n}{2} \right\rfloor\}.$$ We summarize this in the following.

Property 5.1. For $A_n$ and $G_n$ as defined above the matrix $A_nG_n$ is reducible. This implies there exist $\bar{S}_1$ and $\bar{S}_2$, such that

$$\bar{S} = \bar{S}_1 \cup \bar{S}_2, \text{ and } \bar{S}_1 \cap \bar{S}_2 = \emptyset,$$

that are given by

$$\bar{S}_1 = \{(i,j) : i \in \hat{S}_1 \wedge [A_nG_n]_{i,j} \neq 0\},$$

and

$$\bar{S}_2 = \{(i,j) : i \in \hat{S}_2 \wedge [A_nG_n]_{i,j} \neq 0\}.$$ The result in Property 5.1 can easily be extended to the $p$-approximate inverse preconditioner. Let $S^p$ denote the sparsity pattern of $A^p_n$, i.e.,

$$S^p := \{(i,j) : \exists k_1, \ldots, k_{p-1} : (i,k_1) \in S \wedge (k_1, k_2) \in S \wedge \ldots \wedge (k_{p-1}, j) \in S\},$$
or in term of the band-width

$$S^p := \{(i,j) : i - p \leq j \leq i + p, 1 \leq j \leq n\}.$$ Since

$$(5.4) \quad [A_nG_n]_{i,j} = \delta_{i,j} \text{ for } (i,j) \in S^p,$$

we have

$$S^p = \{(i,j) : j = i - p - 1 \vee j = i + p + 1 \vee j = i, 1 \leq j \leq n\}.$$ From this we see that $\{1, \ldots, n\}$ decouples into $p + 1$ disjoint sets:

$$\hat{S}_q = \{i : k(p + 1) + q, 0 \leq k \leq \left\lfloor \frac{n}{p} \right\rfloor, q = 1, \ldots, p + 1$$

So, Property 5.1 can be generalized as
Property 5.2. For $G_n$ constructed such that

$$[A_n G_n]_{i,j} = \delta_{i,j} \text{ for } (i, j) \in S^p,$$

there exist sets $\tilde{S}_q$, $q = 1, \ldots, p + 1$, such that

$$\tilde{S} = \bigcup_{q=1}^{p+1} \tilde{S}_q, \text{ and } \bigcap_{q=1}^{p+1} \tilde{S}_q = \emptyset.$$

Figures 5.1 and 5.2 below illustrate the decoupling for an 1-approximate inverse and a 2-approximate inverse respectively.

![Figure 5.1: Sparsity patterns for unpermuted and permuted $A_n G_n$. 1d Laplace, FEM, uniform grid, $p = 1$.](image)

The analysis above shows that the sparsity pattern of $A_n$ only determines whether $A_n G_n$ decouples or not. From this we see that this decoupling holds for all tridiagonal matrices, and as such decoupling takes place for convection diffusion problems.

### 5.2 A solver based on recursive decoupling

We can now employ the decoupling described in Section 5.1 to construct a solver based on a recursive application of this decoupling. We do this in Subsection 5.2.1. In Subsection 5.2.2 we analyse the complexity of this method.

#### 5.2.1 Description of the solver

Let $k$ denote the index of the recursion level. The level with the finest grid is numbered 0, with $n_0$ degrees of freedom, and we want to solve

$$A_0 x_0 = b_0.$$
Chapter 5: Recursive solution methods

The level at which the recursion ends is numbered $K$, with $n_K$ degrees of freedom. Throughout this section, $A_k$ denotes and $n_k \times n_k$ matrix, and $b_k$ and $x_k$ are vectors of dimension $n_k$.

Suppose we want to solve the problem

\begin{equation}
A_k x_k = b_k.
\end{equation}

We construct the matrix $G_k$ with $p = 1$ as described previously, and use $G_k$ as right preconditioner. So, we solve the preconditioned system

\begin{equation}
A_k G_k y_k = b_k,
\end{equation}

with

\begin{equation}
x_k = G_k y_k.
\end{equation}

In the previous section we showed that $G_k$ causes a decoupling of the nodes, with two sets $\hat{S}_1$ and $\hat{S}_2$. Consider the permutation $\pi_k := \{ \hat{S}_1, \hat{S}_2 \}$ and related permutation matrix $P_k$. We now use $P_k$ to decouple the system. We then obtain

\begin{equation}
P_k A_k G_k P_k^{-1} P_k y_k = P_k b_k,
\end{equation}

or

\begin{equation}
\begin{bmatrix}
A_{k+1,1} & A_{k+1,2} \\
\end{bmatrix}
\begin{bmatrix}
y_{k+1,1} \\
y_{k+1,2} \\
\end{bmatrix} =
\begin{bmatrix}
b_{k+1,1} \\
b_{k+1,2} \\
\end{bmatrix}.
\end{equation}

Note that the matrix $A_k$ is partitioned according to the decoupling of the nodes. The vectors $y_k$ and $B_k$ are partitioned correspondingly.

Figure 5.2: Sparsity patterns for unpermuted and permuted $A_n G_n$. 1d Laplace, FEM, transient grid, $p = 2$. 
Here the two new matrices $A_{k+1,i}$ are again tridiagonal and the same procedure can be repeated for construction and application of matrices $G_{k+1,i}$. The two subsolutions $y_{k+1,1}$ and $y_{k+1,2}$ computed at this stage, possibly by going some levels further, constitute the solution $x_k$ via

$$x_k = G_k P_k^{-1} \begin{bmatrix} y_{k+1,1} \\ y_{k+1,1} \end{bmatrix}. \tag{5.7}$$

The recursion ends at level $K$ when the order of the matrices is small enough to be solved by a direct method. All subsolutions are put together using (5.7) in order to obtain the solution.

The algorithm can easily be extended to the case $p > 1$. Here $G_k$ causes a decoupling into $p + 1$ sets $S_q$, $q = 1, \ldots, p + 1$. With a permutation $\pi_k$ related to this decoupling we end up with the generalized form of (5.6),

$$\begin{bmatrix} A_{k+1,1} \\ \vdots \\ A_{k+1,p+1} \end{bmatrix} \begin{bmatrix} y_{k+1,1} \\ \vdots \\ y_{k+1,p+1} \end{bmatrix} = \begin{bmatrix} b_{k+1,1} \\ \vdots \\ b_{k+1,p+1} \end{bmatrix}. \tag{5.8}$$

Below we give an implementation of the algorithm to solve $A_0x_0 = b_0$ in pseudocode.

```
x = SOLVE(A,b,p)
IF order of A small enough:
    use direct method to compute x;
ELSE
    decouple problem into p+1 subproblems:
    A_i and b_i for i = 1 \ldots p+1,
    with permutation P;
    x_i = SOLVE(A_i,b_i,p) for i = 1 \ldots p+1;
    use P to obtain x from p+1 subsolutions;
```

### 5.2.2 Complexity of the solver

We now examine the workload of the recursive algorithm. The workload $w_k$ at level $k$ is related to the workload at the next level $k + 1$ through the simple recurrence relation

$$w_k = l_k + 2w_{k+1}.$$ 

Here $l_k$ denotes the cost to construct $G_k$, to compute the two subblocks and to reconstruct the solution from the two sub solutions. The construction of the approximate inverse $G_k$ typically involves solving $n_k$ linear systems of dimension $3 \times 3$. Note that $l_k$ is linearly proportional to $n_k$, say $l_k = cn_k$, $c \in \mathbb{R}$. Without loss of generality, we assume $n_0 = 2^K$ for the initial number of unknowns at level 0. Since the decoupling divides the nodes evenly we have $n_{k+1} = \frac{n_k}{2}$. 
So the recurrence relation becomes \( w_k = c2^{K-k} + 2w_{k+1} \). For the workload at the lowest level \( K \) we assume \( w_K = F \). The solution is found to be

\[
(5.9) \quad w_k = \frac{c}{2}(K - k)2^{K-k} + F2^{K-k}.
\]

This shows that for solving \( A_0x_0 = b_0 \) \( w_0 = O(n_0 \log(n_0)) \). Henceforth this algorithm is not optimal. However, there are several issues that can be exploited in order to improve the workload of the algorithm.

Note that \( n_k \) small subproblems need to be solved for the construction of \( G_k \). If this is done in parallel the factor \( l_k \) will no longer be linearly proportional to \( n_k \) and we may assume \( l_k = c \). Then (5.9) becomes \( w_k = (F + c)2^{K-k} - c \), which is of the order \( O(n_k) \).

Another remark is that similarity of the subsystems can be used to reduce computational cost. For a stiffness matrix related to a uniform grid for instance, the related \( n_k \) subsystems that need to be solved to obtain \( G_k \) come in three types only: one for each boundary, and one for all interior points. This implies that only a few subsystems actually need to be solved, and hence the factor \( l_k \) will be \( O(1) \).

For this generalized version where \( p > 1 \) it is easily seen that the workload \( w_k \) at level \( k \) is estimated by

\[
(5.10) \quad w_k = l_k + (p + 1)w_{k+1}.
\]

Here \( l_k \) denotes the effort to compute \( G_k \), obtain the decoupling, and reconstruct the solution. Construction of \( G_k \) involves solving \( N_k \) linear systems of dimension \((2p+1) \times (2p+1)\). If \( p \) is large, computation of the local Green’s functions \( g_i \) is expensive. Also storage requirements for \( G_k \) increase with increasing \( p \). However, a larger \( p \) may be beneficial since the criteria on when to solve the subsystems by a direct method is reached in fewer steps.

### 5.3 Decoupling in two dimensions

Consider now problem (3.1) in two dimensions with \( b = 0 \) and \( c = 0 \). Let us cover the domain \( \Omega \) with a uniform grid, with \( m \) degrees of freedom in both directions, so \( n = m^2 \). A finite difference discretization, or a finite element discretization results in the linear system of equations

\[
(5.11) \quad A_n x_n = b_n.
\]

As in Section 5.1, let \( S \) denote the sparsity pattern of \( A_n \)

\[
(5.12) \quad S := \{(i, j) : [A_n]_{i,j} \neq 0\}.
\]

Because of symmetry we have

\[
(i, j) \in S \iff (j, i) \in S.
\]

In the next lemma we look at the sparsity pattern \( S^2 \) of \( A_n^2 \) and how it relates to \( S \).
Property 5.3. Let $S^2$ denote the sparsity pattern of $A_n^2$. Then we have

$$S^2 := \{(i, j) : \exists k : (i, k) \in S \land (j, k) \in S\}.$$ 

Proof. Let $l$ be a number such that $(i, l) \in S$ and $(j, l) \in S$ for some $i$ and $j$. Because of symmetry we also have $(l, j) \in S$ and it follows that $a_{il}a_{lj} \neq 0$. Entry $[A_n^2]_{i,j}$ is defined by $\sum_{k} a_{ik}a_{kj}$. Since for $k = l$ we have a non-zero contribution to this sum, the pair $(i, j)$ related to entry $[A_n^2]_{i,j}$ is contained in the sparsity pattern. 

This means that the sparsity pattern of $A_n^2$ is the sparsity pattern of $A_n$ complemented by the "neighbours' neighbours": If there are numbers $i, j, k$ such that $(i, j) \in S$, $(j, k) \in S$ and $(i, k) \notin S$, then $(i, k) \in S^2$. Since this statement on the sparsity pattern of $A_n^2$ holds for any two different matrices with identical sparsity patterns, it is also true for $A_nG_n$ with the extra knowledge that for some $(i, j) \in S^2$ it holds that $[A_nG_n]_{i,j} = 0$. This is because of the property

$$[A_nG_n]_{i,j} = \delta_{i,j} \text{ for } (i, j) \in S.$$ 

Then the sparsity pattern $\tilde{S}$ of $A_nG_n$ can be defined as

$$\tilde{S} := \{(i, j) : \exists k, k \neq i : (i, k) \in S \land (j, k) \in S \land [A_nG_n]_{i,k} = [A_nG_n]_{j,k} = 0\}.$$ 

So, a degree of freedom $i$ gets disconnected from its neighbours, but becomes connected to the neighbour's neighbours, as shown in figure 5.3 for a uniform grid. A neighbour is no longer a neighbour after the right multiplication by $G_n$. The left picture in figure 5.3 is correct for a FDM discretization of $-a\Delta u + b \cdot \nabla u + cu$, as well for a FEM discretization of $-a\Delta u + cu$.

![Initial coupling of central dof](initial.png) ![New coupling of central dof](new.png)

Figure 5.3: Initial connectivity in $A_n$ and new connectivity in $A_nG_n$.

Next we examine whether there is a decoupling of the degrees of freedom as seen in the one dimensional case. We have a uniform grid with $m$ nodes in both directions, and we number the points from 1 to $m^2$ from bottom to top and from left to right. For node $i$ let $S_i$ denote the set of nodes that are neighbours of $i$, i.e.,

$$S_i = \{i + 1, i + m, i - 1, i - m\},$$
provided all entries exist. For example, if \(i \mod n = 1\), then \(i - 1 \not\in S_i\), or if \(i \mod n = 0\), then \(i + 1 \not\in S_i\).

Given the same assumptions on existence, for the elements of \(S_i\) we have:

\[
S_{i+1} = \{i + 2, i + m + 1, i, i - m + 1\},
\]
\[
S_{i+m} = \{i + m + 1, i + 2m, i + m - 1, i\},
\]
\[
S_{i-1} = \{i, i - 1 + m, i - 2, i - 1 - m\},
\]
\[
S_{i-m} = \{i - m + 1, i, i - m - 1, i - 2m\}.
\]

In \(A_n G_n\), the set of neighbours for node \(i\) is \(S_{i+1} \cup S_{i+m} \cup S_{i-1} \cup S_{i-m}\).

Let \(\hat{S}_i\) denote the set of nodes \(j\) such that \(i\) and \(j\) are connected,

\[
\hat{S}_i := \{j : j = i + 2km + 2l, k, l \in \mathbb{Z}, 1 \leq j \leq m^2\} \cup
\{j : j = i + (2k + 1)m + 2l + 1, k, l \in \mathbb{Z}, 1 \leq j \leq m^2\}
\]

and consider in particular the sets \(\hat{S}_1\) and \(\hat{S}_2\). Then we have

\[
\hat{S}_1 = \{i : i = 2km + 2l + 1, 0 \leq k, l \leq \left\lfloor \frac{m}{2} \right\rfloor - 1\} \cup
\{i : i = (2k + 1)m + 2l + 2, 0 \leq k, l \leq \left\lfloor \frac{m}{2} \right\rfloor - 1\},
\]

and

\[
\hat{S}_2 = \{i : i = 2km + 2l + 2, 0 \leq k \leq \frac{m}{2} - 1, 0 \leq l \leq \left\lfloor \frac{m}{2} \right\rfloor - 1\} \cup
\{i : i = (2k + 1)m + 2l + 3, 0 \leq k \leq \left\lfloor \frac{m}{2} \right\rfloor - 1, 0 \leq l \leq \left\lfloor \frac{m}{2} \right\rfloor - 1\}.
\]

It is easy to see that \(\hat{S}_1 \cap \hat{S}_2 = \emptyset\) and that \(\hat{S}_1 \cup \hat{S}_2 = \{i\}_{i=1}^{m^2}\).

This result shows that the grid decouples into two disjoint sets of degrees of freedom. For \(\hat{S}\) this gives (cf. Property 5.1):

**Property 5.4.** For a uniform grid, matrices \(A_n\) and \(G_n\) and sparsity pattern \(\hat{S}\) as defined previously, the matrix \(A_n G_n\) is reducible. So there exist \(\hat{S}_1\) and \(\hat{S}_2\) for which it holds that

\[
\hat{S} = \hat{S}_1 \cup \hat{S}_2, \quad \text{and} \quad \hat{S}_1 \cap \hat{S}_2 = \emptyset,
\]

and are given by

\[
\hat{S}_1 = \{(i, j) : i \in \hat{S}_1 \land [A_n G_n]_{i,j} \neq 0\},
\]

and

\[
\hat{S}_2 = \{(i, j) : i \in \hat{S}_2 \land [A_n G_n]_{i,j} \neq 0\}.
\]

For a uniform grid the right multiplication by \(G_n\) results in a decoupling of the degrees of freedom into two sets, as illustrated in figure 5.4, which shows the sparsity patterns of \(A_n\) and \(A_n G_n\).
Decoupling in two dimensions

Remark 5.5. A consequence of the above property is that the matrix \( A_n G_n \) decouples into two matrices, say \( B_1 \) and \( B_2 \). Now it is interesting to see that the rows of these matrices satisfy the stencil

\[
\begin{bmatrix}
-\frac{1}{12}, & -\frac{1}{6}, & -\frac{1}{6}, & -\frac{1}{12}, & 1, & -\frac{1}{12}, & -\frac{1}{6}, & -\frac{1}{6}, & -\frac{1}{12} \\
\end{bmatrix}.
\]

This stencil corresponds to a nine-point finite difference approximation for the Laplace operator given by

\[
\nabla^2 u(x, y) = \frac{1}{8} \left( \frac{1}{h^2} (u(x, y) - u(x, y - 2h)) + \frac{1}{6} u(x - h, y - h) - \frac{1}{6} u(x + h, y - h) - \frac{1}{12} u(x - h, y + h) - \frac{1}{12} u(x + h, y + h) - \frac{1}{6} u(x, y + 2h) \right).
\]

So far we have considered only matrices \( G_n \) that inherit the sparsity pattern of \( A_n \). In section 5.1 construction of \( G_n \) was examined where the sparsity pattern of \( A_p n \), \( p > 1 \) was used as a template. So, let \( S^p \) denote the sparsity pattern of \( A^p_n \).

\[
(5.15) \quad S^p = \{(i, j) : \exists k_1, \ldots, k_{p-1} : (i, k_1) \in S \land (k_1, k_2) \in S \land \ldots \land (k_{p-1}, j) \in S\},
\]

and let \( G_n \) be such that (5.4) is satisfied. Let \( \tilde{S}^p \) denote the sparsity pattern of \( A_n G_n \). For \( p = 1 \) we showed that \( S \) decouples into two disjoint sets. However for \( p = 2 \) there is no decoupling. In order to see this, we show that a node \( i \) remains connected to its neighbours. Let

\[
(5.16) \quad \tilde{S}_i = \{i\} \cup \{i + am + b : -3 \leq a, b \leq 3, |a| + |b| = 3\}.
\]

For \( i + 2m + 1 \) in this set, we take a closer look at \( \tilde{S}_{i+2m+1} \). We see \( i + m + 3 \in \tilde{S}_{i+2m+1} \), and for \( \tilde{S}_{i+m+3} \) we find \( i + 1 \in \tilde{S}_{i+m+3} \). So, we have \( (i, i + 2m + 1) \in \tilde{S}^2 \), \( (i + 2m + 1, i + m + 3) \in \tilde{S}^2 \) and \( (i + m + 3, i + 1) \in \tilde{S}^2 \). From this it follows that \( (i, i + 1) \in \tilde{S}^2 \). Similarly we can show that \( i \) is connected to its other neighbours as well.

Figure 5.4: Sparsity patterns for \( A_n \), \( A_n G_n \), and \( A_n G_n \) after permutation.
If we take $p = 3$, then for node $i$

\[(5.17) \quad \tilde{S}_i = \{i\} \cup \{i + am + b : -4 \leq a, b \leq 4, |a| + |b| = 4\}.\]

For this case $\tilde{S}_3$ happens to decouple into the odd and even numbered degrees of freedom, similar to $\tilde{S}$, and not into four sets as seen in Section 5.1. These two observations for $p = 2$ and $p = 3$ lead to a generalization of Property 5.4:

**Property 5.6.** Let $p \geq 1$ and let $G_n$ be constructed such that property (5.4) is satisfied. Let $S'p$ denote the sparsity pattern of $A_nG_n$. If $p$ is odd, then there exist two sets $\tilde{S}_1$ and $\tilde{S}_2$ such that

\[\tilde{S}^p = \tilde{S}_1 \cup \tilde{S}_2, \quad \text{and} \quad \tilde{S}_1 \cap \tilde{S}_2 = \emptyset.\]

$\tilde{S}_1$ and $\tilde{S}_2$ are as in Property 5.4. For even $p$ there is no decoupling.

We are also interested to see whether this decoupling can be repeated for the right picture in figure 5.3. Here we have two disjoint sets of nodes, $\tilde{S}_1$ and $\tilde{S}_2$, and associated a submatrix of $A_nG_n$ to each, say $B_1$ and $B_2$. For both submatrix corresponding approximate inverses $G_1$ and $G_2$ are computed, and we perform the same analysis on neighbours as done for $A_nG_n$ above. It turns out that two neighbouring nodes do not decouple under the action of the approximate inverse. This is best made clear in figure 5.5.

![Graph of two neighbours](image)

**Figure 5.5:** Two neighbours $a$ and $b$ get not decoupled.

Finally, we take a look at the case where there is a refinement. For convenience, we consider the situation of only one refinement point, as in figure 5.6. Following the same analysis as performed previously, it is seen that neighbours remain neighbours due to this refinement. This is illustrated in the figure by the dotted and solid lines.
5.4 Breakdown of uniform 2-D grids

In this section we look at a breakdown of uniform grids as a result of the decoupling technique discussed in the previous section. We first introduce a notation in order to distinguish between different grids. Based on this breakdown we formulate an algorithm.

We start with a grid $G_0$. Since we want to decouple repeatedly it is convenient to introduce levels. The first grid $G_0$ is associated with level 0, and the two grids originating from this grid, called $G_{1,1}$ and $G_{1,2}$, are obviously connected to level 1. Note that in literature the coarsest grid is usually associated with the lowest level number (0 for example), and the initial grid with some level number $K$. We reverse the notation, since it is not known a priori how many times the decoupling can be repeated. Let $k$ denote the current level with grid $G_k$. Of course, at level $k$ there are multiple grids, but we will consider one only and omit any designation other than one related to the level. To distinguish between grids, we introduce a triple of integers $(m, m_1, m_2)$ which are functions of $k$. Here $m$ denotes the number of rows, $m_1$ the number of nodes in odd numbered rows, and $m_2$ the number of nodes in even numbered rows. The rows are numbered in increasing order. For $k$ even the grid is rectangular, and the number of columns per row is always $m_1$, and hence $m_2 = m_1$. If $k$ is odd, the grid is said to be oriented diagonally. So, from this triple $(m, m_1, m_2)$ we can tell what sort of grid we are dealing with.

Suppose now that $k$ is even. Then we have a rectangular grid $G_k$ with grid parameters $(m, m_1, m_1)$ ($m_2 = m_1$ for rectangular grids). The two grids that originate from $G_k$ after the decoupling are denoted by $G_{k+1,1}$ and $G_{k+1,2}$. These grids are orientated diagonally. Then the grid parameters for these new grids are

$$G_{k+1,1} : (m, \left\lfloor \frac{m_1}{2} \right\rfloor, \left\lfloor \frac{m_1}{2} \right\rfloor);$$
$$G_{k+1,2} : (m, \left\lceil \frac{m_1}{2} \right\rceil, \left\lceil \frac{m_1}{2} \right\rceil).$$
For odd \( k \), we have a diagonally orientated grid \( G_k \) with parameters \((m, m_1, m_2)\). The grids \( G_{k+1,1} \) and \( G_{k+1,2} \) resulting from the decoupling are rectangular and have parameters as follows:

\[
G_{k+1,1} : \left( \left\lfloor \frac{m}{2} \right\rfloor, m_1, m_1 \right);
\]

\[
G_{k+1,2} : \left( \left\lfloor \frac{m}{2} \right\rfloor, m_2, m_2 \right).
\]

Table 5.1 shows the grids that originate from an initial \( 7 \times 7 \) grid, represented by the triple \((m, m_1, m_2)\). In figures 5.7 and 5.8 the decoupling is made visible by depicting the nodes of the initial grid and its two successors.

![Table 5.1: Breakdown of a 7 x 7 grid after three decompositions, grid parameters are (m, m1, m2) shown.](image)

![Figure 5.7: Decomposition of the rectangular (7, 7, 7) grid into two diagonally oriented sets of nodes.](image)

![Figure 5.8: Decomposition of the diagonal (7, 4, 3) grid into two rectangularly oriented sets of nodes.](image)

Related to grid \( G_k \) is the stiffness matrix \( A_k \) based on a finite element of finite difference discretization on that grid. The stiffness matrices related to the grid \( G_{k+1,1} \) and

\[
...\]
Three algorithms based on forced repeated decoupling

$G_{k+1,2}$ are denoted by $A_{k+1,1}$ and $A_{k+1,2}$ respectively. The order $n_k$ of a stiffness matrix $A_k$ related to a grid $G_k$ with parameters $(m, m_1, m_2)$ are easily found to be

\[
k \text{even: } m \cdot m_1; \]
\[
k \text{odd: } \left\lceil \frac{m}{2} \right\rceil m_1 + \left\lceil \frac{m}{2} \right\rceil m_2.
\]

Now we have $A_k$ and $G_k$ we find that the new matrix $A_k G_k$ decouples into parts $\overline{A}_{k,1,1}$ and $\overline{A}_{k,1,2}$.

5.5 Three algorithms based on forced repeated decoupling

In the previous section we introduced definitions for grids, and explained how two new grids $G_{k+1,1}$ and $G_{k+1,2}$ originate from initial grid $G_k$. This creation of two new grids implies a decoupling of the original degrees of freedom into two disjoint sets. With a proper permutation based on this decoupling the related linear operator $A_k G_k$ can be written as a $2 \times 2$ block matrix.

At level $k$ we have to solve the system of equations

\[(5.18) \quad A_k x_k = b_k.\]

We construct the Green’s matrix $G_k$. Next, $G_k$ is applied as a right preconditioner, and we solve

\[(5.19) \quad A_k G_k y_k = b_k,\]

with

\[(5.19) \quad x_k = G_k y_k.\]

The multiplication of $A_k$ by the Green’s matrix $G_k$ results in a decoupling of the degrees of freedom into two disjoint sequences $\hat{S}_1$ and $\hat{S}_2$. Consider the permutation $\pi_k := (\hat{S}_1 \hat{S}_2)$ and the corresponding permutation matrix $P_k$. The permutation of variables reorders the degrees of freedom such that the two sets are numbered one after each other. We apply this permutation in order to get the decoupled system. This is the linear system

\[(5.20) \quad P_k A_k G_k P_k^T P_k y_k = P_k b_k.\]

Note that $P_k^T = P_k^{-1}$. This system is a $2 \times 2$ block system where all sub blocks are of level $k + 1$:

\[(5.21) \quad \begin{bmatrix}
\overline{A}_{k+1,1} & \overline{A}_{k+1,2}
\end{bmatrix}
\begin{bmatrix}
y_{k+1,1} \\
y_{k+1,2}
\end{bmatrix}
= \begin{bmatrix}
b_{k+1,1} \\
b_{k+1,2}
\end{bmatrix}.\]

Here

\[
\begin{bmatrix}
\overline{A}_{k+1,1} & \overline{A}_{k+1,2}
\end{bmatrix}
:= P_k A_k G_k P_k^T,
\]
Chapter 5: Recursive solution methods

\[
\begin{bmatrix}
y_{k+1,1} \\
y_{k+1,2}
\end{bmatrix} := P_k y_k
\]

and

\[
\begin{bmatrix}
b_{k+1,1} \\
b_{k+1,2}
\end{bmatrix} := P_k b_k.
\]

There are two independent linear subsystems,

\[
\begin{align*}
\overline{A}_{k+1,1} y_{k+1,1} &= b_{k+1,1}, \\
\overline{A}_{k+1,2} y_{k+1,2} &= b_{k+1,2},
\end{align*}
\]

to be solved, instead of one system in (5.18). These are related to two sub grids of \( G_k \), called \( G_{k+1,1} \) and \( G_{k+1,2} \) respectively. At this new level \( k + 1 \) the decoupling procedure is to be repeated. Unfortunately this cannot be accomplished with the matrices \( A_{k+1,i} \), as outlined in the previous section. However, given the regularity of the new grids \( G_{k+1,1} \) and \( G_{k+1,2} \) it is possible to repeat the decoupling.

In the following three algorithms are presented that make use of this.

5.5.1 A recursive preconditioner: I

A first idea is to find suitable preconditioners for the two linear systems in (5.22). Related to grids \( G_{k+1,1} \) and \( G_{k+1,2} \) we construct Laplace matrices \( A_{k+1,1} \) and \( A_{k+1,2} \) such that these matrices can be decoupled using Green’s matrices \( G_{k+1,1} \) and \( G_{k+1,2} \). So, the Laplace matrices \( A_{k+1,1} \) and \( A_{k+1,2} \) are utilized as preconditioners for \( \overline{A}_{k+1,1} \) and \( \overline{A}_{k+1,2} \) respectively. The next step in the algorithm is to solve the preconditioned systems

\[
\begin{align*}
A_{k+1,1}^{-1} \overline{A}_{k+1,1} y_{k+1,1} &= A_{k+1,1}^{-1} b_{k+1,1} \\
A_{k+1,2}^{-1} \overline{A}_{k+1,2} y_{k+1,2} &= A_{k+1,2}^{-1} b_{k+1,2}.
\end{align*}
\]

Both \( A_{k+1} \) and \( \overline{A}_{k+1} \) are discretizations of the Laplace operator: \( A_{k+1} \) is related to the five point discretization, and \( \overline{A}_{k+1} \) to a nine point discretization. So, since \( A_{k+1} \) and \( \overline{A}_{k+1} \) are discretizations of the same operator, it makes sense to use one of the two as preconditioner for the other. Furthermore, we have seen that the matrix \( A_{k+1} \) related to the five point stencil decouples after multiplication with the corresponding Green’s matrix \( G_{k+1} \).

In iterative methods systems of equations with \( A_{k+1,1} \) and \( A_{k+1,2} \) have to be solved in order to apply this preconditioner. In order to solve these systems, the same process of computing Green’s matrices \( G_{k+1,1} \) and \( G_{k+1,2} \), and decoupling of the the degrees of freedom is repeated. The two subsolutions \( y_{k+1,1} \) and \( y_{k+1,2} \) are computed at this stage, possibly by going some levels further. Using (5.19) the solution \( x_k \) is obtained via

\[
x_k = G_k P_k^{-1} \begin{bmatrix} y_{k+1,1} \\ y_{k+1,2} \end{bmatrix}.
\]

The recursion terminates at level \( K > 0 \) when the dimensions of the linear systems \( A_K \) at that level are such that the systems of equations can be solved using a direct
Three algorithms based on forced repeated decoupling

method. The $2^K$ partial solutions are assembled to obtain the solution of the original problem $A_0x_0 = b_0$.

Below we give an implementation of the algorithm described above in (5.18) - (5.24), called SOLVE-I, by the following pseudocode:

```plaintext
x = SOLVE-I(A, rhs)
IF order of A small enough:
    use direct method to compute x;
ELSE
    split A into two subproblems:
        A1p, rhs1 and preconditioner A1,
        A2p, rhs2 and preconditioner A2,
        with permutation P;
    x1 = CGS(A1p, rhs1, SOLVE-I(A1), EPS, MAXIT);
    x2 = CGS(A2p, rhs2, SOLVE-I(A2), EPS, MAXIT);
    use permutation P to obtain x from x1 and x2;
END
```

The tolerance EPS and maximum number of iterations MAXIT used by the inner CGS iterative method are shown explicitly since these two parameters influence speed and accuracy of SOLVE-I.

There are two ways to use the SOLVE-I algorithm: directly as a solver, or as a preconditioner. When used as a solver, the call is directly to the algorithm:

```plaintext
x = SOLVE-I(A, rhs).
```

When used as a preconditioner inside an iterative method, the use of SOLVE-I is as follows:

```plaintext
x = CGS(A, rhs, SOLVE-I, EPS, MAXIT).
```

5.5.2 A recursive preconditioner: II

In the SOLVE-I algorithm in the previous subsection the recursive nature follows from (5.23). Since both $A_{k+1,i}$ and $\overline{A}_{k+1,i}$ are matrices related to different discretizations of the Laplace operator, we consider substitution of $\overline{A}_{k+1,i}$ by $A_{k+1,i}$. With this approach, matrices $\overline{A}_{k+1,i}$ need not to be computed explicitly. Another benefit is that $A_{k+1,i}$ required less storage since these matrices are sparser.

So, the linear systems in (5.23) with $\overline{A}_{k+1,i}$ are replaced by $A_{k+1,i}$ resulting in

\[
\begin{align*}
A_{k+1,1}y_{k+1,1} &= b_{k+1,1} \\
A_{k+1,2}y_{k+1,2} &= b_{k+1,2}
\end{align*}
\]

In this alternative in the recursive step we solve

$$A_{k+1}y_{k+1} = b_{k+1}$$
instead of

\[ \overline{A}_{k+1}y_{k+1} = b_{k+1}. \]

An updated version of the algorithm, now called \texttt{SOLVE-II} reads as follows:

\[
x = \text{SOLVE-II}(A, \text{rhs})
\]

IF order of \( A \) small enough:

use direct method to compute \( x \);

ELSE

split \( A \) into two subproblems:

\( A_1 \) and \( \text{rhs1} \),

\( A_2 \) and \( \text{rhs2} \),

with permutation \( P \);

\( x_1 = \text{CGS}(A_1, \text{rhs1}, \text{SOLVE-II}(A_1), \text{EPS}, \text{MAXIT}) \);

\( x_2 = \text{CGS}(A_2, \text{rhs2}, \text{SOLVE-II}(A_2), \text{EPS}, \text{MAXIT}) \);

use permutation \( P \) to obtain \( x \) from \( x_1 \) and \( x_2 \);

END

5.5.3 A recursive preconditioner: III

The above algorithm can be altered even further, by simply replacing the calls to an iterative method by a call to the algorithm itself. This results in the following algorithm:

\[
x = \text{SOLVE-III}(A, \text{rhs})
\]

IF size of \( A \) small enough:

use direct method to compute \( x \);

ELSE

construct two matrices related to splitting of \( A \):

\( A_1 \) and \( \text{rhs1} \),

\( A_2 \) and \( \text{rhs2} \),

with permutation \( P \);

\( x_1 = \text{SOLVE-III}(A_1, \text{rhs1}) \);

\( x_2 = \text{SOLVE-III}(A_2, \text{rhs2}) \);

use permutation \( P \) to obtain \( x \) from \( x_1 \) and \( x_2 \);

END

To summarize, we have three versions of the \texttt{SOLVE} algorithm:

1. \textbf{[SOLVE-I]} The algorithm with inner iterative solvers to solve

\[ A_{k+1}^{-1}\overline{A}_{k+1}y_{k+1} = A_{k+1}^{-1}b_{k+1}; \]

2. \textbf{[SOLVE-II]} The algorithm with inner iterative solvers to solve

\[ A_{k+1}y_{k+1} = b_{k+1}; \]

3. \textbf{[SOLVE-III]} The algorithm with a recursive call to itself.
5.6 Complexity of the algorithms

The work at level \( k \) to solve the system \( A_k x_k = b_k \) depends on the tolerance \( \epsilon \) (EPS in the SOLVE algorithms in Section 5.5) and is denoted by \( w_k(\epsilon) \). The work at this level, except for the work required to solve the two subproblems, is denoted by \( l_k \). This involves construction of the Green’s matrix and the related permutation. The work required by the iterative method used (CGS for example) is denoted by \( g_k \).

Finally, let \( \bar{\lambda}_k(\epsilon) \) denote the average number of iterations needed for solving the two subproblems. Then we obtain the following estimate for level \( k \),

\[
(5.26) \quad w_k(\epsilon) = l_k + 2\bar{\lambda}_{k+1}(\epsilon)(g_{k+1} + w_{k+1}(\epsilon)).
\]

For CGS we assume \( \bar{\lambda}(\epsilon) \) iterations. Apart from \( w_{k+1}(\epsilon) \) as indicated above each iteration costs \( 18n_{k+1} \) flops, see [9]. Furthermore, \( l_k = 3n_k \) (approximation for the work to obtain the permutation, but still linear in the amount of degrees of freedom). We assume \( n_{k+1} = \frac{a}{2}n_k \), and henceforth \( n_k = 2^{K-k}n_K \) where \( n_K \) denotes the amount of degrees of freedom at level \( K \) when a direct method is used. The work at level \( K \) is fixed: \( w_K(\epsilon) = w_K \).

Then the workload \( w_k(\epsilon) \) at level \( k \) is expressed as follows:

\[
(5.26) \quad w_k(\epsilon) = aw_{k+1}(\epsilon) + bn_K2^{K-k}.
\]

Here

\[
a = 2\bar{\lambda}(\epsilon), \quad b = 3 + 18\bar{\lambda}(\epsilon).
\]

We can solve this recursion:

\[
w_k(\epsilon) = a^{K-k}w_K + bn_K2^{K-k}\left(\frac{a}{2}\right)^{K-k} - 1 \quad \frac{a}{2} - 1.
\]

Whence for the total workload \( w_0(\epsilon) \) to solve \( A_0 x_0 = b_0 \) we have

\[
(5.27) \quad w_0(\epsilon) \simeq O(\bar{\lambda}(\epsilon)^K(w_K + n_0)).
\]
Chapter 6
NumLab concepts

The scientific computing numerical and visualization algorithms are becoming more and more complex. The construction of such algorithms from source code becomes expensive: Interpretation is too slow, compilation prevents interactive steering and overview gets lost. To make it worse, there is no formal computer language specification of complex mathematical notions such as partial differential equations, ordinary differential equations, boundary value problems, etc.

With this situation in mind this chapter discusses the concepts of the NumLab computational platform. The aim of NumLab is to aid in the rapid (better) software construction for complex scientific computations and visualizations. Though software might also be written with the available tools on the market, down to the use of assembler code, this would take an insubordinate amount of time. Our goal is to write the same software much faster and in a convenient manner. To aid the rapid software development, NumLab offers toolboxes of high level operators and invariance of important operator properties under operator composition.

First a brief introduction of existing architectures is given, followed by the goals to be achieved by NumLab.

6.1 Introduction

From a structural point of view, software environments for numerical computations can be classified into three categories (see for instance [82]): Libraries, turnkey systems, and application frameworks.

Libraries for numerical algorithms such as LAPACK [2], NAGLIB [79], or IMSL [53], or for visualization such as OpenGL [55], Open Inventor [105], or VTK [87], provide services in the form of data structures and functions. Libraries are usually easy to extend with new data types and functions. However, using libraries to build a complete computational or visualization application requires involved programming.

Turnkey systems, such as Matlab [67], Mathematica [107], or the many existing dedicated numerical simulators on the market, are simpler to use than libraries to build a complete application. However, extending the functionality of such systems is usually limited to a given application domain, as in the case of the dedicated simulators, or to a fixed set of supported data types, as in the case of the Matlab programming environment.
Application (computational) frameworks, such as the Diffpack and SciLab systems for solving differential equations [26, 54] or the Orange system for experimental mathematics [48] combine the advantages of the libraries and turnkey systems. On one hand, frameworks have an open structure, similar to libraries, so they can be extended with new components, such as solvers, matrix storage schemes, or mesh generators. On the other hand, some frameworks (notably visualization) offer an easy manner to construct a complete application that combines visualization, numerical computations, and user interaction. This is usually provided by means of visual programming tools such as Matlab’s Simulink [67] or the dataflow network editing tools of the AVS [97], IRIS Explorer [1], or Orange [48] frameworks. In these frameworks, applications are constructed by assembling visual representations (icons) of the computational or visualization components in a network. Program execution is implemented in terms of computational operations on the network nodes and data flows between these nodes respectively.

With the above in mind, let us consider how the NumLab environment integrates the advantages of the above architectures. On the level of libraries, NumLab’s C++ routines call Fortran, Pascal, C and C++. Next, similar to a turnkey system, NumLab offers full integration of visualization and numerical computation and implements communication with other environments such as Simulink [67] and MathLink [107]. On the application framework level, NumLab provides interactive application construction with its visual programming dataflow system Vission [92, 94]. Furthermore, NumLab provides an object-level (subroutine-level) make-concept which allows for interactive program validation.

In order to better address NumLab’s merits on all levels, we need a closer look at computational frameworks. Though efficient and effective, most existing computational frameworks are limited in several respects. First, limitations exist from the perspectives of the end user, application designer, and component developer [3, 37, 82, 94]. First, few computational frameworks facilitate convenient interaction between visualization (data exploration) and computations (numerical exploration), both essential to the end user. Secondly, from the application designer’s perspective, the visual programming facility, often provided in visualization frameworks such as AVS or Explorer [1, 97], is usually not available for numerical frameworks. Conversely, it is quite difficult to integrate large scale computational libraries in visualization frameworks.

Finally, from the numerical component developer perspective, understanding and extending a framework’s architecture is still (usually) a very complex task, albeit noticeably simplified in object-oriented environments such as [26, 87].

Next to limitation with respect to the three types of users, many computational frameworks are constrained in a more structural manner: Similar mathematical concepts are not factored out into similar software components. As a consequence, most existing numerical software is heterogeneous, thus hard to deploy and understand. For instance, in order to speed up the iterative solution of a system of linear equa-
tions, a preconditioner is often used. Though iterative solvers and preconditioners fit into the same mathematical concept, that of an approximation $x$ which is mapped into a subsequent approximation $z = F(x)$, most computational software implements them incompatibly, so preconditioners cannot be used as iterative solvers and vice versa [26]. Another example emerges from finite element libraries. Such libraries frequently restrict reference element geometry and bases to a (sub)set of possibilities found in the literature. Because this set is hard coded, extensions to different geometries and bases for research purposes is difficult, or even impossible.

The design of NumLab addresses all the above problems. NumLab is a numerical framework that provides C++ software components (objects) for the development of a large range of interdisciplinary applications (PDEs, ODEs, non-linear systems, signal processing, and all combinations). Further, it provides interactive application design/use with its visual programming dataflow system Vission, data interchange (e.g. via Simulink and MathLink), and can be used both in a compiled and interpreted fashion. Its computational libraries factor out fundamental notions with respect to numerical computations (such as evaluation of operators $z = F(x)$ and their derivatives), which keeps the amount of basic components small. All components of these libraries are aware of dataflow, even in the absence of the Vission dataflow system, and can for instance call back to see whether provided data is valid.

In order to achieve its rapid software construction goal NumLab offers C++ libraries

(L1) with a uniform cross-language interface;

(L2) with factored out common numerical concepts ([72] and figure 6.1);

(L3) which are based on blue-prints (templates), ([73]);
(L4) with high level components (PDE solvers, complex vector field visualizations, see figure 6.1 ([71]));

(L5) which can be used for interpretation ([27]);

(L6) which can all be used for visual programming,

and in addition:

(A1) a visual programming application called Vission ([93]).

There are libraries based on both public and commercial software, and for both scientific computing and visualization purposes.

The libraries have been designed with other important goals in mind as well, but most of these are out of the scope of this thesis: parallelism, generation of C++ from visual networks and vice versa, Turing completeness. The choice of an existing computer language was based on the desire not to add another language to the tower of Babel – different from the Lawrence Livermore BABEL project which introduces a new scientific interface definition language (SIDL). The choice for C++ was simple: it is a kind of computer language superset.

The properties of the NumLab libraries above ensure:

(M1) abstraction from data representation and much less different manuals to read;

(M2) second opinions (different solver/visualization) at about no cost;

(M3) speed due to on demand generation of specialized code from generic blueprints;

(M4) composition of complex solvers with just a few components;

(M5) low learning curve (comparable to Matlab);

(M6) overview and convenience.

The Vission application ensures crash-safe operation across multiple libraries ([81]).

The work required for the creation of the NumLab libraries can also be split corresponding to the six points as follows:

(W1) encapsulation of all libraries and generation of data-conversion code;

(W2) factorization of concepts across libraries;

(W3) refactorization of libraries and production of in-house written libraries;

(W4) production of in-house written prototype libraries;

(W5) restriction to CINT C++ subset [27];

(W6) restriction to the Vission (C++) concepts [93].
In practice the restriction to the CINT C++ subset has not lead to restrictions on the encapsuled libraries: In C++ a certain result can be achieved with different algorithms (different pieces of code) and so far we have not encountered problems we could not solve.

The restrictions to the concepts of the Vission application were harder to deal with due to the Vission’s research status. For instance, Vission does not offer naked pointers as C++ does. This complicates matters: Libraries such as the professional Visualization ToolKit VTK libraries could not be encapsuled one to one with the source code, which leads to maintenance problems (each new version of VTK requires a thorough inspection). The fact is that it is possible to also introduce naked pointers in Vission, but this would not solve all problems with libraries that return pointers. The encapsulation of public domain and commercial libraries consumes a lot of time but can, will and is automated for all or part of the task. In fact, most (eliminated) weaknesses in the NumLab libraries have been found in an attempt to automate the encapsulation.

The development of numerical libraries which function “as expected” in the visual data-flow application (A1) consumed a large part of the amount of time dedicated to the development of the NumLab libraries. The reason is that it was difficult to write numerical libraries which lived up to the envisioned advantages: The visual representation of a computer program should be a kind of network which should offer:

(N1) structure: a network which contains function/object instances and argument connections/alterations;

(N2) structure: data travels from the top of this network (source) and gets altered on its journey down to the bottom (output);

(N3) overview: a network with just a few components can form a complex application;

(N4) overview: network details can be hidden and be shown on demand;

(N5) interaction: a network accepts graphical input next to the standard console or file input;

(N6) interaction: while it is executed a network can be altered in order to speed up the computations.

In order to achieve this, two major problems had to be addressed. First, visualization of low level components (such as a sorting algorithm) is

1. slow due to the interpretation,

2. and tends to become a visual mess of connections due to the multiple use of a datum.
Visualization of all levels of numerical components is further difficult because

3. (iterative) code tends to rebind \( x^{(k)} \mapsto x^{(k+1)} \), for which no standard visual realization exist,

4. and therefore control loops are non-trivial to realize.

To overcome the low level problems, NumLab as well as VTK/Inventor only offers high level visual components such as linear-system solvers, ODE solvers, PDE solvers, etc.. To overcome the second problem, libraries such as VTK and Inventor make the problem the visual application’s problem. For visualization problems, that seems fine, but for numerical problems, where fixed point iterations and hence loops dominate the scene, this is impossible. Hence, NumLab offers primitives for loop control.

This thesis adds both to the NumLab concepts and its content. As part of the work, iterative solution routines and preconditioners were added to NumLab. The construction of libraries for the Vission application are commented at in the following order. First, Section 6.2 presents the NumLab foundations, which are principles to which components of NumLab adhere. More in detail, Section 6.3 comments on the NumLab elements and Section 6.4 examines NumLab operators. These are divided into the categories systems of equations in Subsection 6.4.1, solvers and preconditioners in Subsection 6.4.2, partial differential equations in Subsection 6.4.3 and ordinary differential equations in Subsection 6.4.4. Next, Section 6.5 explains how NumLab factors out common components. Details with regard to the implementations are provided in subsections.

6.2 The foundation of NumLab

NumLab offers various elements \( x \) of linear vector spaces \( V \) and operators \( F \) between such spaces. For all elements \( x, y \in V \) and a scalar \( c \) it offers

1. All vectorspace operations: \( x + y, c \cdot x \), etc..

For each operator \( F \), it offers

2. Evaluation: \( z = F(x) \) which returns an element.

And for each differentiable operator \( F \) and element \( x \), it offers:

3. Jacobian determination: \( Z = dF(x) \) which returns a linear operator.

The Jacobian determination can be exact or numerical and should be the Frechet derivative; NumLab cannot examine continuity of the derivative.

The NumLab concept is the simplest possible: For an evaluation \( z = F(x) \), each operator \( F \) can use other elements \( \{x_i\} \), other elements \( \{z_i\} \), as well as other operators \( \{F_i\} \); for a visual representation see figure 6.2.
The NumLab operator concept is more basic than for instance the DiffPack one. There, specific operators cannot use all other operators. For instance, a linear solver operator can use a preconditioner but not other operators. Thus, as an example, DiffPack must and does offer SOR both as a linear solver and as a preconditioner. In contrast, NumLab offers just one operator SOR which can both be used as a solver or as a preconditioner, as will be explained in Subsection 6.4.2. Sometimes operators use more input data. In case of an iterative solver for example, this may include an initial value, a preconditioner and parameters to specify the stopping criteria of the solver.

The NumLab implementation (toolboxes) implement the vector spaces $\mathbb{R}^N$, the vector spaces of (piecewise) functions defined on partitioned geometries as well vector spaces of functions on $\mathbb{R}^d$. Because most of NumLab’s applications are in finite volume/difference/elements, the remainder of this chapter will focus on $\mathbb{R}^N$ and spaces of (piecewise) functions defined on partitioned geometries.

### 6.3 The NumLab elements

Here we consider the mathematical framework for spaces $V$ in more detail. In general, let $\Omega$ be the bounded polygonal/polyhedral domain of interest, with smooth enough boundary $\partial \Omega$. The linear vector space $V = V_1 \times \cdots \times V_n$ is a cross-product space of $n$ spaces ($n$ is the amount of degrees of freedom). Each space $V_i$ is spanned by basis functions $\{v_{ij}\}_{j=1}^{N_i}$ where $v_{ij} : \Omega \to \mathbb{R}$. An element $x \in V$ is a vector function from $\Omega$ to $\mathbb{R}^n$, and is written as $x = [x_1, \ldots, x_n]$, a vector of component functions. Each component $x_i \in V_i$ is a linear combination of basis functions, for all $c \in \Omega$

$$x_i(c) = \sum_{j=1}^{N_i} x_{ij}(t) v_{ij}(c). \quad (6.1)$$

Each element $x_i$ is associated to a unique scalar vector $X_i = [x_{i1}, \ldots, x_{iN_i}] \in \mathbb{R}^{N_i}$. In turn, $X$ denotes the aggregate of the vectors $X = [X_1, \ldots, X_n]$, and $X_{ij} = [X_{ij}]$. Summarized, we have vector functions $x = [x_1, \ldots, x_n]$ and related vectors of coefficient vectors $X = [X_1, \ldots, X_n]$.

Whenever $n = 1$, we use a more standard notation. In this case, the space is $V$, etc.
spanned by basis functions $\{v_j\}_{j=1}^N$, and elements $x \in V$ are related to coefficient vector $x = [x_1, \ldots, x_N]$.

For most finite element computations, the basis functions $v_{ij}$ of $V_i$ have local support. However, basis functions have global support in spectral finite elements computations. The local supports, also called elements, are created with the use of a triangulation algorithm. On the lowest level, NumLab offers vectors in $\mathbb{R}^n$ and block matrices in $\mathbb{R}^{n \times m}$ for coefficient vectors $\{v_{ij}\}$ and Jacobian matrices. Of course, each matrix block has full, diagonal or sparse representations.

### 6.4 The NumLab operators

This section discusses the NumLab operators. Here is a small overview on the NumLab operators:

1. **Problem-specific operators**: Transient Finite Element, Volume, Difference operators $F$ for transient boundary value problems (BVPs); Operators which formulate systems of ordinary differential equations (ODEs); operators which act on linear operators (for instance image filters);

2. **Problem-specific solvers for systems of ODEs**: Time-step and time-integration operators formulated with the use of (parts of) the problem-specific operators mentioned above. The former operators require non-linear solvers for the computation of solutions;

3. **Solvers for systems of non-linear equations**: Such systems are operators, and their solution is reduced to the solution of a sequence of linear systems;

4. **Solvers for systems of linear equations**: Such systems are also operators $F(x) = Ax - b$. Their solution is reduced to a sequence of operator evaluations and vector space operations.

The reduction from one type of operator to another is commented on in the subsections below, in the reverse order of the itemization above. Thus, Subsection 6.4.1 examines systems of (non-)linear equations, Subsection 6.4.2 examines (non-)linear solvers and preconditioners, Subsection 6.4.4 considers the reduction of systems of ODEs to non-linear systems, and Subsection 6.4.3 deals with an initial boundary value problem. The presented mathematical reductions are de facto standards, new is NumLab’s software implementation which is a one to one mapping of these techniques.

### 6.4.1 Systems of equations

NumLab has no special representation for (non-)linear systems of equations, such entities are formulated with the use of operators. In NumLab a matrix is a linear operator $F$ which is of the form $x \mapsto Ax$, where $A$ is the canonical coefficient matrix representation of $F$ (obtained with the canonical basis of vector space $V$). Next, if
F is a linear operator and f an element, a linear system of equations is represented with an affine operator G of the form

\( x \mapsto F(x) - f. \)

In NumLab, an image is a linear operator which simplifies image processing (see Figure 6.4m).

Non-linear systems of equations are represented by an operator \( x \mapsto F(x) \) such that a solution of \( F(x) = 0 \) solves the non-linear system.

### 6.4.2 Solvers and Preconditioners

In NumLab, all solvers are represented as operators as well. Each solver \( S \) for the problem \( z = F^{-1}(x) \) calculates an approximation of \( F^{-1}(x) \) for a specific argument \( x \) and extra argument \( F \). The operator \( S \) approximates \( F^{-1}(x) \) as follows: it is implemented as a fixed point iteration \( z^{(k+1)} = S(z^{(k)}) \) with initial value \( z^{(0)} := z \) and final value \( z = z^{(K)} \). By construction, it is not continuous due to a stopping tolerance \( \epsilon > 0 \) and the maximum amount of iterations, and it is not possible to calculate \( dS \) for a specific \( x \) and \( F \).

All of NumLab’s iterative solvers adhere to the following principle: If \( z^{(0)} \in V \) for some space \( V \), then

\[ z^{(0)} \xrightarrow{S} z^{(1)} \xrightarrow{S} z^{(2)} \xrightarrow{S} z^{(3)} \xrightarrow{S} \cdots, \]

is a sequence in \( V \), i.e., each solver maps \( V \) into \( V \). This holds even if \( V \) is constrained, such as \( V = \{[1, 2] + \lambda[3, 4] : \lambda \in \mathbb{R} \} \subset \mathbb{R}^2 \). This is of importance because Dirichlet or Neumann boundary conditions as will be discussed in Subsection 6.4.3 lead to constrained spaces \( V \). The idea behind (6.3) is that all iterations should map an approximate solution to a potentially better approximate solution of the problem of interest. This at least requires that all \( z^{(k)} \), \( k > 0 \) are in the same space as \( z^{(0)} \).

Furthermore, because this holds, we can switch and mix linear solvers on demand.

As an example, let \( F \) be an affine operator (linear system) and consider the Richardson iterative solver \( R \) for the computation of \( z = F^{-1}(x) \). This solver is defined with the use of a fixed point iteration:

\[ z^{(k+1)} = z^{(k)} - (F(z^{(k)}) - x), \]

The evaluation \( z = R(x) \) of the Richardson operator \( R \) is implemented with the use of NumLab’s basic operations as follows:

\[ r^{(1)} = F(z^{(k)}); \]

\[ r^{(2)} = r - x; \]

\[ z^{(k+1)}^{(2)} = z^{(k)} - r; \]
Chapter 6: NumLab concepts

with initial value \( z^{(0)} := z \) and final value \( z := z^{(K)} \). Here \(^{(1)}\) denotes _operator evaluation_ and \(^{(2)}\) _vector space operation_, where the right hand part is assigned to the left hand side. The computation of the Jacobian is not offered because \( R \) is not differentiable.

Because iterative solvers can be slow, as detailed in Chapter 3, it is common to use a preconditioner. Richardson method with preconditioner (operator) \( P \) passed as an extra operator is:

\[
(6.6) \quad z^{(k+1)} = z^{(k)} - P(F(z^{(k)}) - x).
\]

Now the implementation of the evaluation \( z = R(x) \) of the preconditioned Richardson operator \( R \) is:

\[
\begin{align*}
    r^{(1)} & \equiv F(z^{(k)}); \\
    r^{(2)} & \equiv r - x; \\
    s^{(1)} & \equiv P(r); \\
    z^{(k+1)}^{(2)} & \equiv z^{(k)} - s.
\end{align*}
\]

The NumLab toolbox only offers the preconditioned Richardson method with default preconditioner \( x \mapsto x \). Each operator (such as \( x \mapsto U^{-1}L^{-1}x \)) can be used as a preconditioner. Likewise, each solver can use itself as preconditioner, if its maximum amount of iterations is set to for instance 1. It should be noted that different solvers use preconditioners in a different manner.

Though preconditioning in NumLab is simple, this does not solve the problem of proper preconditioning. This has been addressed in Chapter 3 and the application designer should keep these mathematical restrictions in mind, when designing a suitable solver for the problem at hand.

Similar to linear solvers, NumLab also formulates non-linear solvers as operators. Problems with non-linear operators which do not provide derivative evaluation, can be solved with the use of a fixed point method (comparable to the Richardson method above), or with a combinatorial fixed point method [96] (a multiDimensional variant of the bisection method). Non-linear operators that provide derivative evaluation can also be solved with (damped, inexact) Newton methods (see [36, 40]).

As an example, let \( x \) be an element and \( F \) an operator, and consider the NumLab undamped Newton’s method \( U \) for the calculation of \( z = F^{-1}(x) \): The fixed point iteration is:

\[
(6.8) \quad z^{(k+1)} = z^{(k)} - dF^{-1}(z^{(k)})(F(z^{(k)}) - x),
\]

and could be implemented as:

\[
\begin{align*}
\mathbf{r}^{(1)} & = \mathbf{F}(\mathbf{z}^{(k)}); \\
\mathbf{r}^{(2)} & = \mathbf{r} - \mathbf{x}; \\
\mathbf{s}^{(1)} & = \text{inv}(d\mathbf{F}(\mathbf{z}^{(k)}))(\mathbf{r}); \\
\mathbf{z}^{(k+1)}^{(2)} & = \mathbf{z}^{(k)} - \mathbf{r},
\end{align*}
\]

(6.9)

where \( \mathbf{F} \mapsto \text{inv}(d\mathbf{F}(\mathbf{z}^{(k)})) \) returns the inverse (linear operator) of the Jacobian of \( \mathbf{F} \) at \( \mathbf{z}^{(k)} \). However, NumLab implements Newton’s method differently. It uses a linear solver \( \mathbf{R} \) in order to approximate \( d\mathbf{F}(\mathbf{z}^{(k)})^{-1}(\mathbf{r}) \):

\[
\begin{align*}
\mathbf{r}^{(1)} & = \mathbf{F}(\mathbf{z}^{(k)}); \\
\mathbf{r}^{(2)} & = \mathbf{r} - \mathbf{x}; \\
\mathbf{s}^{(1)} & = \mathbf{R}(\mathbf{r}, \text{Jacobian}(\mathbf{F})); \\
\mathbf{z}^{(k+1)}^{(2)} & = \mathbf{z}^{(k)} - \mathbf{r}.
\end{align*}
\]

(6.10)

The linear solver is passed to Newton’s method as an extra operator, its Jacobian matrix is calculated and passed to Richardson method. The Richardson solver is also passed as an extra operator, which in turns could have been passed a preconditioner as an extra operator.

### 6.4.3 Partial differential equations

In order to show how NumLab formulates partial differential equations (PDEs) as operators, consider a PDE. Let \( d \) be a positive integer and \( \Omega \subset \mathbb{R}^d \) be the bounded region of interest, and let \( \partial \Omega \) denote its boundary. The problem of interest is to find a solution \( \mathbf{u} \) defined on \( \Omega \) that satisfies

\[
\begin{align*}
-\Delta \mathbf{u} & = \mathbf{h} \quad \text{in } \Omega, \\
\mathbf{u} & = \mathbf{g} \quad \text{at } \partial \Omega.
\end{align*}
\]

(6.11)

For the sake of simplicity of presentation, all boundary conditions are of Dirichlet type. The NumLab construction of the operator depends on the discretization method and on the specific properties desired from the operator: Whenever possible, NumLab operators are maximal monotone (see [64]). This ensures that the Jacobian is a positive (semi-) definite linear operator. For a finite element discretization of (6.11) NumLab defines a maximal monotone operator as follows. First, \( \Omega \) is covered with elements (see [70]) which serve to define a basis \( \{v_j\}_{j=1}^N \) (see [57]), which generates the vector space \( V \) of interest. In standard Galerkin finite elements (see [5]), the solution \( \mathbf{u} \) to (6.11) is assumed to be in the space \( V \), i.e., \( \mathbf{u} = \sum_j u_j v_j \) for some coefficient vector \( \mathbf{u} = [u_1, \ldots, u_N] \in \mathbb{R}^N \). Define \( \mathbf{E} : V \to V \) and \( \mathbf{f} \in V \) as follows:

\[
\mathbf{E}(\mathbf{u}) := \sum_i \left( \int \nabla \left( \sum_{j=1}^N u_j v_j \right) \nabla v_i - h v_i \right) v_i, \quad \mathbf{f} = \sum_i \left( \int h v_i \right) v_i.
\]

(6.12)
Now recall that for an approximate solution \( u^{(0)} \in V^g \) an iterative solver must produce a sequence

\[
(6.13) \quad u^{(0)} \xrightarrow{(S)} u^{(1)} \xrightarrow{(S)} u^{(2)} \xrightarrow{(S)} u^{(3)} \xrightarrow{(S)} \ldots \in V^g,
\]

and recall that NumLab’s iterative methods (such as Richardson method and all methods in [9]) perform calculations on an unconstrained vector space \( \mathbb{R}^N \). These solvers cannot perform calculations on non-linear subsets of \( \mathbb{R}^N \) such as affine subsets.

In NumLab, this problem is solved as follows: Partial differential operators are defined in such a manner that all required computations can take place in a linear subset (i.e., a subspace) of \( \mathbb{R}^N \) which all solvers (for instance all methods in [9]) can handle. For this linear BVP, as well as for all non-linear BVP’s discretized with the finite element method, we proceed as follows. For \( \gamma : \partial \Omega \rightarrow \mathbb{R} \) define

\[
(6.14) \quad V^\gamma = \{ u \in V : u = \gamma \text{ at } \partial \Omega \}
\]

and let \( P \) denote the projection of \( V \) onto \( V^0 \). The NumLab operator which represents the finite element discretization of (6.11) is defined through its evaluation and computation of its Jacobian:

1. \( z = F(u) := P(E(u) - f) \in V \)
2. \( Z = dF(u) := I - P + PdE(u)P^T \in V \mapsto V \).

In this manner

1. \( F : V \mapsto V^0 \);
2. \( dF : V \mapsto V^0 \times V^0 \) and \( dF \) is invariant on \( V - V^0 \);
3. \( dF \) is the derivative of \( F \) in \( V^0 \);
4. \( dF \) is maximal monotone;
5. The concept (6.13) holds for all linear and non-linear solvers in NumLab;
6. \( F(x) = 0 \iff P(x) \) solves the homogeneous problem related to (6.11),

In NumLab the linear vector space \( V \) is represented with \( \mathbb{R}^N \), \( V^0 \) is a chosen linear subspace.

If we want to solve the transient version

\[
(6.15) \quad \frac{\partial}{\partial t} u(t) = \Delta u + f \quad \text{in } \Omega,
\]

\[
u = g \quad \text{at } \partial \Omega,
\]

with a Method of Lines (MOL) discretization, we substitute \( F \) for \( E \) in the next section. As an alternative one can simultaneously discretize in time and space (see for instance [6]).
6.4.4 Ordinary differential equations

Standard discretizations of ordinary differential equations can also be formulated as operators whose evaluation reduces to a sequence of vector space operations and function evaluations. For instance, let $F$ be a discrete NumLab operator, and consider the initial value problem: Find $x(t)$ for which:

$$
\frac{d}{dt} x(t) + F(t, x(t)) = 0 \quad (t > 0), \quad x(0) = x_0.
$$

Let $h > 0$ denote the discrete time-step, and define $t_k = kh$ for all $k = 0, 1, 2, \ldots$. Next, let $V$ be the space which is used to approximate $x(t)$ at a specific time $t$, let $\{v_j\}_j$ be the basis used to this end and let $M$ be the linear operator:

$$
[M]_{i,j} = (v_j, v_i).
$$

Provided with an approximation $x^{(k)} \in V$ of $x(t_k)$, a fixed-step Euler backward method determines an approximation $x^{(k+1)} \in V$ of $x(t_{k+1})$

$$
M(x^{(k+1)} - x^{(k)}) = -hE(t_{k+1}, x^{(k+1)}),
$$

which can be rewritten as

$$
M(x^{(k+1)} - x^{(k)}) + hE(t_{k+1}, x^{(k+1)}) = 0.
$$

Define the operator $E$ as follows:

$$
E(x) := M(x - x^{(k)}) + hE(t_{k+1}, x).
$$

Then $x^{(k+1)}$ is a solution of $E(x) = 0$. Of course, $E$ depends on the extra argument values $x^{(k)}$, $t_k$ and $h$. Each time step we have to solve $E(x) = 0$ for which we can use a non-linear solver examined above. Note that $F$ is maximal monotone if $E$ is because $M$ is a positive semi-definite linear operator. Likewise, in a finite difference setting, if $E$ is diagonal dominant, so is $F$. If $E$ is an $L$-matrix, so if $F$, etc..

In general, if a NumLab operator depends on another one, its implementation will keep all properties invariant. Both on a global scale and a lower level (element/stencil-wise) scale.

The approach above generalizes in a straightforward manner to all explicit methods, such as Runge-Kutta type methods [30], as well as to all implicit discretization methods, such as Backward Difference Formulas (BDF) [44]. Furthermore, this approach generalizes to the case of fixed and variable timestep integrators (see PEC and PECE [68]). An example is the solution of a Lotka-Volterra predator-prey problem, shown in Fig. 6.4 l.
6.5 The NumLab factored out common components

For all offered ODE/PDE discretizations NumLab factors out common components. For the sake of demonstration, we focus on the case of a finite element discretization such as in Section 6.4.3. In NumLab, a finite element function $x \in V$ is constructed with a sequence of entities: In this list, on each line, each first item is input to NumLab, and all others are derived (invisible):

1. boundary specification $\mapsto$ geometry specification $\mapsto$
2. reference element $\mapsto$
3. element generator $\mapsto$ computational grid (discrete geometry) $\mapsto$
4. reference functions $\mapsto$
5. boundary conditions $\mapsto$ linear and constrained spaces $\mapsto$ sampler $\mapsto$
6. symbolic function $\mapsto$
7. element $x$ in the space $V$.

For each input alternatives are possible, for instance one can use quadrilateral instead of triangular elements. A NumLab network which demonstrates all of these input choices can be seen in figure 6.3. The various parts are encircled and numbered as follows: The grid is defined first (1), followed by the space (2). Then the operator of interest (3) is constructed, followed by the solver (4). In this example we have chosen for the CGS iterative method which is preconditioned by an incomplete factorization. Finally, the visualization pipeline is indicated by (5).

NumLab’s basic requirements, such as (6.13), are elegantly and efficiently captured by using an object-oriented approach to software design [18, 28, 77, 83]. Consequently, we have implemented our numerical software framework as an object-oriented library written in the C++ language [91]. The sections which follow provide more (implementational) detail about the different factored out components.

6.5.1 The Grid module

Recall the NumLab element definitions in Section 6.3. To be able to define local support for the basis functions $v_{ij}$ later on, we need to discretize the function’s domain $\Omega$. This is modeled in the software framework by the Grid module, which covers the function’s domain with elements $e$. This Grid module takes a Contour as input, which describes the boundary $\partial \Omega$ of $\Omega$. The default contour is the unit square’s contour.

In NumLab, the grid covers regions $\Omega$ in any dimension (e.g. 2D planar, manifold or 3D spatial), and consists of a variety of element shapes, such as triangles, quadrilaterals, tetrahedra, prisms, hexahedrals, $n$-simplices (see [70]), and so on. All grids implement a common interface. This interface provides several but few services. These include: Iteration over the grid elements and their related vertices, topological queries such as the element which contains a given point. The amount of services
The NumLab factored out common components

Figure 6.3: A NumLab finite differences network.
is a minimum: Modules which use a grid generator and need more service must compute the required relations from the provided information.

Specific grid generator modules produce grids in different manners. NumLab contains Delaunay generators, simplicial generators, and regular generators, and "generators" which read an existing grid from a file. An example generator is illustrated in figure 6.4k, which shows a cubic finite element interpolant on a 2-manifold in $\mathbb{R}^3$.

### 6.5.2 The Space module

The linear vector space $V$ is implemented by the software module `Space`. `Space` takes a `Grid` and `BoundaryConditions` as inputs. The grid’s discretization in combination with the boundary conditions are used to build the supports of its basis functions $v_{ij}$. The default boundary conditions are Dirichlet type conditions for all solution components. None, Robin, Neumann and vectorial boundary conditions are specified per boundary part. Recall that elements in $V$ do not have to satisfy the Dirichlet boundary conditions.

Because `Grid` has a minimal interface, some information, required by `Space` for the construction of the basis functions, is not provided. Whenever this happens, `Space` internally computes the required information with the use of `Grid`’s services.

A specific `Space` module implements a specific set of basis functions, such as constant, linear, quadratic, or even higher order polynomial degree, matched to the elements’ geometry. The interface of the `Space` module follows the mathematical properties of the vector space $V$ presented so far: Elements $x, y \in V$ can be added together or scaled by real values. Furthermore, elements $v_{ij}$ of $V$ are functions, and $V$ permits evaluation at points $c \in \Omega$ of such functions and their derivatives.

It should be kept in mind that elements of $V$ are functions, not linear combinations of functions. Therefore, the name `Space` is somewhat misleading. However, for the brevity of demonstration, the name `Space` will also be used in the sequel.

In most cases the required basis functions have local support, also called `element-wise` support. The restriction of global basis function $v_{ij}$ to support $e$ is said to be local function $v_{ir}$. In software, this is coded as follows: For space component $i$ (so $V_i$), element $e$, and local basis function $r$ thereon, $j := j(i, r)$ induces basis function $v_{ij}$. The software implementation is on element-level for efficiency purposes: Given a point $c \in \Omega$, `Space` determines which support $e$ contains $c$ for the evaluation of $v_{ij}(c)$.

### 6.5.3 The Function module

As discussed, a vector function $x : \Omega \mapsto \mathbb{R}^n$ in a space $V$ generated by $v_{ij}$ is uniquely related to a coefficient vector $X$ with coefficients $X_{ij}$. Based on this observation, NumLab software module `Function` implements a vector function $x$ as a block vector of real-valued coefficients $X_{ij}$, combined with a reference to the related `Space`, which contains related functions $v_{ij}$.

The `Function` module provides services to evaluate the function and its derivatives at a given point $c \in \Omega$. To this end, both $x$’s coefficient vector $X$ and the point $c$ are...
6.5.4 The operator module

As described previously, an operator $F : V \mapsto W$ maps an element $x \in V$ to an element $z \in W$. The evaluation $z = G(x)$ computes the coefficients $z_{ij}$ of $z$ from the coefficients $x_{ij}$ of $x$, as well as from the bases $\{v_{ij}\}$ and $\{w_{ij}\}$ of $V$ and $W$ respectively. Next to the evaluation of $G$, derivatives such as the Jacobian operator $DG$ of $G$ are evaluated in a similar manner. Such derivatives are important in several applications. For example, they can be used in order to find a solution of $G(z) = x$, by Newton’s method.

The software implementation of the operator notion follows straightforwardly the mathematical concepts introduced in Section 6.4. The implementation is done by the Operator module, which offers two services: evaluation of $z = G(x)$, coded as $G$.eval($z$, $x$), and of the Jacobian of $G$ in point $y$, $z = DG(y)x$, coded as $G$.getJ($y$).eval($z$, $x$). To evaluate $z = G(x)$, the Operator module takes two Function objects $z$ and $x$ as input and computes the coefficients $z_{ij}$ using the coefficients $x_{ij}$ and the bases of the Space objects $z$ and $x$ carry with them. It is important that both the ‘input’ $z$ and the ‘output’ $x$ of the Operator module are provided, since it is in this way that Operators determine the spaces $V$, respectively $W$.

To evaluate $z = DG(y)x$, the Operator proceeds similarly. Internally, $DG(y)$ is usually implemented as a coefficient matrix, and the operation $DG(y)x$ is a matrix-vector multiplication. However, the implementation details are hidden from the user ($DG(y)x$ may be computed element-wise, i.e. matrix-free), who works only with the Function and Operator mathematical notions.

Specific Operator implementations differ in the way they compute the above two evaluations. For example, a simple Diffusion operator $z = G(x)$ may operate on a scalar function and produce a function $z$ where $z_i = x_{i-1} - 2x_i + x_{i+1}$. A generic Linear operator may produce a vector of coefficients $z = Ax$ where $A$ is a matrix. A Summator operator $z = G_1(x) + G_2(x)$ may take two inputs $G_1$ and $G_2$ and produce a vector of coefficients $z_i = [G_1(x)]_i + [G_2(x)]_i$. Remark that the modules passed to the Space module referred to by $x$. In turn, the Space module returns the value of $x(c)$. This is computed following the definition $x(c) = \sum_j x_{ij} v_{ij}(c)$, as described in the previous section. The computation of the partial derivatives of a given function $x$ in a point $c$ follows a similar implementation.

Providing evaluation of functions $x \in V$ and of their derivatives at given points is, strictly speaking, the minimal interface the Space module has to implement. However, it is sometimes convenient to be able to evaluate a function at a point given as an element number and local coordinates within that element. This is especially important for efficiency in the case where one operation is iterated over all elements of a Grid, such as in the case of numerical integration. If the Space module allows evaluating functions at points specified as elements and local element coordinates, the implementation of the numerical integration is considerably faster than when point-to-element location has to be performed. Consequently, we also provided the Space module with a function evaluation interface which accepts an element number and a point defined in the element local coordinates.
implementing the Linear and Summator operators actually have two inputs each. In both cases the function $x$ is the first input, while the second is the matrix $A$ for the Linear operator and the operators $G_1$ and $G_2$ for the Summator operator. These values could be as well hard-coded in the operator implementation. In both cases however, we see Operator as a function of a single variable $x$, as described in the mathematical framework.

6.5.5 The Solver module

We model the solving of $G(z) = x$ by the module Solver in our software framework. Mathematically, Solver is similar to an operator $S: V \rightarrow W$, where $V$ and $W$ are function spaces. The interface of Solver provides evaluation at functions $x \in W$, similarly to the Operator module. The implementation of the Solver evaluation operation $z = S(x)$ should provide an approximation $z$ to $z \approx F^{-1}(x)$. However, Solver does not provide evaluation of its Jacobian, as this may be undesirably complex to compute in the general case.

Practically, Solver takes as input an initial guess Function object $x$ and an Operator object $G$. Its output $z$ is such that $G(z) = x$. The operations done by the solver are either vector space operations or Operator evaluations, or evaluations of similar operators $G(z)$. In the actual implementation, this is modeled by providing the Solver module with one or more extra inputs of type Solver. In this way, one can for example connect a nested chain of preconditioners to an iterative solver module. The implementation of a specific Solver follows straightforwardly from its mathematical description. Iterative solvers such as Richardson, SOR, CG, Bi-CG, with or without preconditioners, are easily implemented in this software framework.

The framework makes no distinction between a solver and a preconditioner, as discussed in Section 6.4. The sole difference between a solver and a preconditioner in this framework is semantic, not structural. A solver is supposed to produce an exact solution of $G(z) = 0$ (up to a desired numerical accuracy), whereas the preconditioner is supposed to return an approximate one. Both are implemented as Solver modules, which allows easy cascading of a chain of preconditioners to an iterative solver as well as using preconditioners and solvers interchangeably in applications. Furthermore, the framework makes no structural distinction between direct and iterative solvers. For example, an ILUSolver module is implemented to compute an incomplete LU factorization (cf. Subsection 3.4.3) of its input operator $G$. The ILUSolver module can be used as a preconditioner for a ConjugateGradient solver module. In the case the ILUSolver is not connected to the ConjugateGradient module’s input, the latter performs non preconditioned computations. Alternatively, a LUSolver module is implemented to provide a complete LU factorization of its input operator $G$ (see Chapter 2). The LUSolver can be used either directly to solve the equation $G(z) = x$, or as preconditioner for another Solver module.

6.5.6 NumLab components

The natural modeling of the mathematics in terms of class hierarchies, the object-oriented design allows users to easily extend the current framework with new soft-
ware modules. Implementing a new solver, preconditioner, or operator usually involves writing only a few tens of lines of C++ to extend an existing one. The same approach also facilitates the reuse of existing numerical libraries such as LAPACK [2] or Templates [9] by integrating them in the current object-oriented framework.

Summarizing, the main classes of NumLab modules are:

- **Grid**: Produces a grid from a set of parameters. Examples are 2D and 3D grid generators for regular and unstructured grids, and grid file readers;

- **Function**: Several specific functions \(v_{ij}\) are generated, such as cosines, or piecewise (non-)conforming polynomial functions in several dimensions;

- **Space**: There is a single Space class, but a multitude of basis functions are implemented. This module takes parameters defining the characteristics of the space used.

- **Operator**: This produces an operator that maps elements between two spaces, with the operators specifics given as input parameters. Operators for several ODEs, PDEs, and non-linear systems have been implemented, such as Laplace, Stokes, Navier-Stokes, and elasticity problems. Next, several operators for matrix manipulation and image processing have been implemented. For example, matrix sparsity patterns can be easily visualized, as in other applications like Matlab (figure 6.4j).

- **Solver**: A range of iterative solvers including CG, Bi-CG, CGS, Bi-CGStab, etc. etc. are implemented. Solver specifics are given as input parameters. Several preconditioners such as ILU are also provided as Solver specializations, following the common treatment of solver and preconditioner modules previously described.

### 6.6 NumLab implementation

To conclude this chapter we comment on the implementation of the NumLab workbench as an indication of the complexity to achieve the six points M1 – M6. The current NumLab workbench version consists of the following amounts of lines of code:

- Computational libraries \(O(130,000)\) – contributed as part of this thesis;

- Visualization libraries \(O(70,000)\);

- Network editor \(O(40,000)\).

The computational and visualization libraries serve code from:

- VTK for visualization \(O(400,000)\) – current version 80%;

- Open Inventor for visualization \(O(400,000)\) – current version 100%;
• LAPACK BLAS for computations $O(1.000.000)$ - current version 10%;

• SEPRAN for computations $O(1.000.000)$ - current version 1%.

A NumLab application which makes use of computations and visualization is linked to an average of 40 to 50 different libraries.

To finish with, the NumLab workbench runs on Linux and Silicon Graphics systems with the appropriate versions of all libraries. In order to facilitate its maintenance, processes are automated as much as is possible. The emphasis is on the automated construction of wrappers for external libraries such as VTK, Open Inventor and SEPRAN.
Figure 6.4: Visualization of various numerical computations in the NumLab environment.
Chapter 7

NumLab preconditioners

In this chapter we examine the implementation of the approximate inverse preconditioner of Chapter 4 as part of the NumLab workbench. The NumLab concept has been described in Chapter 6. In this chapter we show how the approximate inverse preconditioner is implemented as part of NumLab and how this new tool is used for research purposes. The implementation of our new tool, the approximate inverse preconditioner, is described in Section 7.1. As detailed in Chapter 1 we are interested in grids with a brick-like structure, and in particular in grids with several levels of refinement. For such grids and the model problem in (7.1) of interest we illustrate the effectiveness of the approximate inverse preconditioner by means of several examples. In Section 7.2 some one dimensional examples are considered first, followed in Section 7.3 and Section 7.4 by more realistic two and three dimensional examples.

7.1 Introduction

In this section we describe how preconditioning techniques can be embedded inside the NumLab workbench, and in particular the approximate inverse preconditioner. Recall from Chapter 3 the convection diffusion equation

\[
-a \Delta u + \mathbf{b} \cdot \nabla u + cu = f \quad \text{in } \Omega = (0,1)^d \subset \mathbb{R}^d, \\
u = g \quad \text{at } \partial \Omega, 
\]

and the resulting linear system of equations

\[
A_n x_n = b_n 
\]

obtained after discretization of (7.1).

The approximate inverse preconditioner \( G_n \) from Chapter 4 is implemented in a new module. Given the input of a linear operator \( A_n \), this module first constructs the approximate inverse \( G_n \) and subsequently evaluates a vector \( x_n \) whenever called upon. We apply this preconditioner to speed up the CGS iterative method as detailed in Section 3.1. As the implementation into NumLab can be copied line by line, we only give the lines relevant for the preconditioning steps. For a full implementation of the CGS algorithm see [103]. The matrix-vector multiplications are denoted in operator notation as detailed in Section 6.2. So, the matrix-vector multiplication
\( x_n = G_n z_n \) is denoted as \( x = G(z) \). Recall from Subsection 6.4.2 that \((1)\) denotes an operator evaluation and \((2)\) denotes a vector space operation. Then, for the right preconditioned system we solve the linear system

\[
A_n G_n y_n = b_n, \quad x_n = G_n y_n.
\]

Using the notation of (6.7) we obtain, given an initial value \( x^{(0)} \),

\[
\begin{align*}
    r^{(0)} & \overset{(1)}{=} b - A(x^{(0)}); \\
    \vdots & \quad \\
    p & \overset{(1)}{=} G(p^{(k)}); \\
    v & \overset{(1)}{=} A(p); \\
    \vdots & \\
    u & \overset{(1)}{=} G(u^{(k)} + q^{(k)}); \\
    x^{(k)} & \overset{(2)}{=} x^{(k-1)} + \alpha_k u; \\
    u & \overset{(1)}{=} A(u); \\
    r^{(k)} & \overset{(2)}{=} r^{(k-1)} - \alpha_k u; \\
    \vdots & 
\end{align*}
\]

(7.3)

After convergence we find the final value \( x^{(K)} \).

For the left preconditioned case we solve the linear system

\[
G_n A_n x_n = G_n b_n,
\]

and translating this into pseudocode gives, given an initial value \( x^{(0)} \),

\[
\begin{align*}
    r^{(0)} & \overset{(1)}{=} G(b - A(x^{(0)})); \\
    \vdots & \\
    v_1 & \overset{(1)}{=} A(p^{(k)}); \\
    v & \overset{(1)}{=} G(v_1); \\
    \vdots & \\
    u & \overset{(2)}{=} u^{(k)} + q^{(k)}; \\
    x^{(k)} & \overset{(2)}{=} x^{(k-1)} + \alpha_k u; \\
    u & \overset{(1)}{=} A(u); \\
    u & \overset{(1)}{=} G(u); \\
    r^{(k)} & \overset{(2)}{=} r^{(k-1)} - \alpha_k u; \\
    \vdots & 
\end{align*}
\]

(7.4)
with final value $x^{(K)}$. Note the two extra matrix-vector multiplications compared to the unpreconditioned CGS algorithm. In figure 7.1 a NumLab network is given showing the approximate inverse component `OperatorIteratorLinearAI` connected to the CGS iterative solver. The various distinctive parts of the network are indicated. If we want to run another test on a different domain, in the part labeled (1) we have to change one of the parameters of the module `GeometryTransformationTensor`; see for example figure 7.10 for an opened interactor. Also, if we would like to change the operator, we have to alter the module called `DifferenceTransport`. This is shown for example in figure 7.3.

### 7.2 Numerical results in one dimension

In this section and the following we present numerical examples to illustrate the effectiveness of the approximate inverse preconditioner, introduced in Section 3.5 and described in detail in Chapter 4. This is done by implementing the necessary modules inside the NumLab workbench as new tools. This includes a new operator module that constructs the approximate inverse and evaluates it. As all NumLab implementations are made with visual programming in mind, this new module has a user-interface to control and steer its operation. For the approximate inverse module one could think of means to adapt the sparsity pattern of the approximate inverse $G_n$, or even change the type of approximate inverse technique used. Note, however, that depending on the level of flexibility wanted, every approximate inverse technique can be made into a single module. Since we consider Frobenius norm minimization to compute $G_n$, cf. Subsection 3.5.1, we have a module that is specialized in this task only.

As has been mentioned in Chapter 1 we assume that the inhomogeneity of the source term $f$ results in a solution $u$ with layers toward the boundary. This in turn forces us to utilize grids with local refinements as for example shown in figure 7.4 and figure 7.11. For example, the function

$$u(x) = \frac{\pi}{2} + \arctan\left(\frac{x - 1}{0.01}\right)$$

has a steep layer near $x = 1$ and as such a computational domain as in figure 7.4 is required to capture this steep gradient accurately. Note however that for the spectral condition numbers the solution is not important. For solving the linear systems iteratively we take $x_n^{(0)} = 0$ as initial guess.

In the examples to follow we perform tests as follows. First we are interested in the spectral condition numbers of the matrices $A_n$, $A_n G_n$ and $G_n A_n$. In the tables with numerical results these tests are denoted by $\kappa_2$-1, $\kappa_2$-2 and $\kappa_2$-3 respectively. Secondly we use the CGS iterative method to solve the three corresponding linear systems of equations. These are

**CGS-1**: the original system of equations

$$A_n x_n = b_n,$$
Figure 7.1: NumLab network showing the approximate inverse module.
CGS-2: the right preconditioned system
\[ A_n G_n y_n = b_n, \quad x_n = G_n y_n, \]
and finally

CGS-3: the left preconditioned system
\[ G_n A_n x_n = G_n b_n. \]

In order to make comparisons we use the ILU(0) preconditioner to precondition the three linear systems and see how the CGS method performs. This results in three more tests:

PCGS-1: Let \( K_1 \) denote the ILU(0) factorization of \( A_n \). Then we solve
\[ K_1^{-1} A_n x_n = K_1^{-1} b_n. \]

PCGS-2: Here let \( K_2 \) denote the ILU(0) factorization of \( A_n G_n \). The system of equations we solve is
\[ K_2^{-1} A_n G_n y_n = K_2^{-1} b_n, \quad x_n = G_n y_n. \]

PCGS-3: Let \( K_3 \) denote the ILU(0) factorization of \( G_n A_n \). The system to solve is
\[ K_3^{-1} G_n A_n x_n = K_3^{-1} G_n b_n. \]

In the tables a bar (”-“) means that the iterative solver did not converge properly. For the tests without ILU(0) preconditioner this means that the CGS solver broke down because of irregular convergence behaviour. For the tests with additional ILU(0) this failure is caused because the factors of the incomplete factorization are ill-conditioned. This is one of the reasons why approximate inverse preconditioners are studied (cf. Section 3.5).

In the one dimensional case all matrices \( A_n \) are tridiagonal and we take the sparsity pattern of \( A_n \) for the construction of \( G_n \). Hence, construction of \( G_n \) requires solving \( n \times 3 \) linear systems. The computational complexity for constructing this \( G_n \) is approximately \( 15n \). The computational cost for one CGS iteration is approximately \( 14n \) for the unpreconditioned case, and \( 20n \) for the preconditioned case (see e.g. [9]). Here we have taken into account that \( G_n \) is an explicit preconditioner, and as such only requires matrix-vector multiplications for its application. So the work for constructing \( G_n \) is approximately equivalent to 1 iteration with the CGS method. Here we use Gaussian elimination to solve the \( n \) linear systems, and do not take into consideration similarity of these subsystems. The computational complexity for the construction of the ILU(0) preconditioner for a tridiagonal matrix is approximately \( 3n \). For the tests where we apply ILU(0) to \( A_n G_n \) and \( G_n A_n \) the cost for constructing the incomplete factorization is the same. We illustrate this by two examples.
Example 7.1. In this example we consider the grid depicted in figure 7.2 with refinement factor $\psi = 1/2$. Let $n > 0$ and let $f := 1/\psi^{n+1}$. Then the grid points $x_i$ are given by (cf. (4.2))

$$x_i = \frac{f(1-i/(n+1)) - f}{1 - f}, \quad \forall i = 0, \ldots, n + 1.$$  

After using the boundary conditions in $x_0$ and $x_{n+1}$ we obtain an $n \times n$ linear system of equations. First we consider the pure diffusive case, $a = 1$ and $b = c = 0$. The results are given in table 7.1.

<table>
<thead>
<tr>
<th>$n$</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\kappa_2$</td>
<td>119.92</td>
<td>497.60</td>
<td>2027.81</td>
<td>8188.27</td>
<td>3.29$\times$10$^4$</td>
<td>1.32$\times$10$^5$</td>
<td>5.28$\times$10$^5$</td>
<td>2.12$\times$10$^6$</td>
</tr>
<tr>
<td>$\kappa_2$</td>
<td>13.75</td>
<td>39.41</td>
<td>60.15</td>
<td>168.92</td>
<td>249.04</td>
<td>696.75</td>
<td>1014.12</td>
<td>2833.04</td>
</tr>
<tr>
<td>$\kappa_2$</td>
<td>2.87</td>
<td>5.33</td>
<td>4.86</td>
<td>7.56</td>
<td>6.97</td>
<td>9.76</td>
<td>9.12</td>
<td>11.94</td>
</tr>
<tr>
<td>CGS-1</td>
<td>5</td>
<td>7</td>
<td>10</td>
<td>13</td>
<td>18</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CGS-2</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CGS-3</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>9</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>PCGS-1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>PCGS-2</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PCGS-3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1: Results for grid 7.1, $\psi = 1/2$, diffusion only.

We see from this table that $\kappa_2(A_n)$ is $O(h^{-2})$, and that the CGS method breaks down for $n \geq 9$. For the two preconditioned linear systems, $\kappa_2(G_nA_n)$ behaves as predicted by the estimate in Property 4.26, despite the fact we consider here $G_nA_n$ instead of the modified matrix $B_n$ in (4.13). For the other product $A_nG_n$ we observe $\kappa_2(A_nG_n) \approx O(h^{-1})$ as in (4.29). For the tests with the ILU(0) preconditioner we see that this preconditioner breaks down because of ill-conditioned factors when applied to $A_n$ and $A_nG_n$. Judging from the number of iterations needed to converge the combination of the ILU(0) preconditioner and the matrix $G_nA_n$ seems to be a real winner. It should be noted that additional to the construction of $G_n$ the construction of the incomplete factors of $G_nA_n$ is required, which means an increase in computational complexity. However, this does not amount to the complexity of 1 extra iteration.

Next we consider the same grid but now with a convective term $b\nabla u$ added. For our NumLab network this means we have to alter an input parameter in the module that defines the differential operator. This is illustrated in figure 7.3 in which we have
altered the x-convection in order to run another test with convective coefficient $b = 100$. The results are in table 7.2.

![Figure 7.3: Interactor for changing the equation.](image)

<table>
<thead>
<tr>
<th>$n$</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\kappa_2$-1</td>
<td>32.18</td>
<td>92.91</td>
<td>294.68</td>
<td>1021.96</td>
<td>3777.55</td>
<td>1.45$\times10^4$</td>
<td>5.68$\times10^5$</td>
<td>2.25$\times10^6$</td>
</tr>
<tr>
<td>$\kappa_2$-2</td>
<td>18.11</td>
<td>78.36</td>
<td>83.00</td>
<td>343.78</td>
<td>362.26</td>
<td>1456.81</td>
<td>1543.67</td>
<td>5963.71</td>
</tr>
<tr>
<td>$\kappa_2$-3</td>
<td>2.39</td>
<td>3.16</td>
<td>3.25</td>
<td>3.78</td>
<td>3.95</td>
<td>4.47</td>
<td>4.88</td>
<td>5.63</td>
</tr>
<tr>
<td>CGS-1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>11</td>
<td>14</td>
<td>17</td>
<td>24</td>
<td>32</td>
</tr>
<tr>
<td>CGS-2</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>CGS-3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>PCGS-1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PCGS-2</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PCGS-3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 7.2: Results for grid 7.1, $\psi = 1/2$, diffusion and convection.

The results for this test with convection are slightly better than the results for the non-convective case. However, the growth of the condition numbers seems to be identical. Also the performance and breakdown of the ILU(0) preconditioner matches the previous results. With respect to commenting on the number of iterations the CGS iterative method needs to converge, we must take into account the irregular convergence behaviour of the method. Nevertheless, in both tests we observe that both preconditioned systems converge faster than the unpreconditioned system.
Chapter 7: NumLab preconditioners

Example 7.2. To conclude with the one dimensional examples we look at the grid in figure 7.4 which resembles a simple representation of a brick. Let \( m > 0 \) such that \( n = 2m + 1 \) and let \( f := 1/\psi^{m+1} \). Then the grid points \( x_i \) for this grid are given by

\[
x_i = \begin{cases} 
    1/2 - \frac{f^{(2i/(n+1))}}{2(1-f)}, & 0 \leq i < m + 1, \\
    1/2, & i = m + 1, \\
    1/2 + \frac{f^{(2-2i/(n+1))}}{2(1-f)}, & m + 1 < i \leq n + 1.
\end{cases}
\]

(7.6)

We consider the purely diffusive case. The results are displayed in table 7.3.

<table>
<thead>
<tr>
<th>( \kappa_2 )</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>9</th>
<th>11</th>
<th>13</th>
<th>15</th>
<th>17</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \kappa_2 )</td>
<td>4.57</td>
<td>21.53</td>
<td>99.14</td>
<td>425.24</td>
<td>1761.62</td>
<td>7171.82</td>
<td>2.89 \times 10^4</td>
<td>1.16 \times 10^7</td>
<td>4.66 \times 10^7</td>
</tr>
<tr>
<td>( \kappa_2 )</td>
<td>1.50</td>
<td>6.33</td>
<td>16.75</td>
<td>37.58</td>
<td>76.40</td>
<td>164.02</td>
<td>321.86</td>
<td>679.88</td>
<td>1319.49</td>
</tr>
</tbody>
</table>

| \( \kappa_2 \) | 1.68| 3.12| 4.29| 5.37| 6.72| 7.54| 9.03| 9.72| 11.28 |

| CGS-1          | 2   | 3   | 5   | 7   | 14  | 14  | -   | 22  | -   |
| CGS-2          | 2   | 3   | 4   | 5   | 12  | 7   | 12  | 11  | 17  |
| CGS-3          | 2   | 3   | 4   | 5   | 8   | 7   | 10  | 9   | 11  |
| PCGS-1         | 1   | 1   | -   | -   | -   | -   | -   | -   | -   |
| PCGS-2         | 1   | 1   | -   | -   | -   | -   | -   | -   | -   |
| PCGS-3         | 1   | 1   | 1   | 1   | 1   | 1   | 1   | 1   | 1   |

Table 7.3: Results for grid 7.4, \( \psi = 1/2 \), diffusion only.

We observe a growth of \( \kappa_2(\mathbf{A}_n) \) proportional to \( h^{-2} \), of \( \kappa_2(\mathbf{A}_n\mathbf{G}_n) \) proportional to \( h^{-1} \) and of \( \kappa_2(\mathbf{G}_n\mathbf{A}_n) \) proportional to \( \log(h^{-1}) \). Also in this example we observe breakdown of the CGS method for the unpreconditioned system, and faster convergence for the two preconditioned systems. Furthermore, preconditioning \( \mathbf{A}_n \) and \( \mathbf{A}_n\mathbf{G}_n \) with ILU(0) fails because of ill-conditioned factors. This nullifies the fact that preconditioning \( \mathbf{A}_n \) with ILU(0) works for two problem sizes. Again, ILU(0) applied to \( \mathbf{G}_n\mathbf{A}_n \) (PCGS-3) is the best.

In figure 7.5 the norms of the residuals at each iteration step are given for \( n = 13 \) and \( n = 15 \). The irregular convergence of the CGS method for solving the unpreconditioned system (CGS-1) is clearly visible in the left hand figure; the stagnation of the iterative process becomes clear in the right hand figure. Furthermore we see that the iterative process for solving the two preconditioned systems converges much smoother.
Numerical results in one dimension

The computational complexity of CGS-2, CGS-3 and PCGS-3 is shown in figure 7.6. Clearly PCGS-3 performs best, and the other two are comparable to each other.

The reduction in total workload is obvious when we compare these figures to CGS-1. For example for \( n = 17 \), CGS-1 needs 22 iterations to obtain convergence and this amounts to approximately 300\( n \) flops. Although we look at a particular problem, we leave \( n \) explicitly as this makes it easier to compare the respective workloads. The
computational complexity for solving CGS-3 is close to $200n$ flops. For PCGS-3 the costs are approximately $40n$ flops and this is clearly much better than CGS-1 and CGS-3, despite the additional construction of the incomplete factorization. Note also that since the number of iterations for PCGS-3 seems to be constant regardless of the number of degrees of freedom, this combination of approximate inverse and incomplete factorization preconditioning results in an optimal computational complexity (see (3.20)).

7.3 Numerical results in two dimensions

In this section we examine tensor grids with refinement to one or more borders, as shown in figures 7.7, 7.8 and 7.11 below. These grids are direct extensions to the one dimensional grids in the previous section and are better representations of the brick-like structure we had in mind in Section 1.2. With respect to executing such tests in NumLab, the only change is made in the module that generates the grid. As discussed previously, this module for the grid can either be a purpose made module or a generic module in which a parameter can be set to get the grid of interest.

In all examples below the matrix $A_n$ is pentadiagonal, and the approximate inverses $G_n$ inherit the sparsity pattern of $A_n$. This way the computational cost for one CGS iteration is approximately $18n$ without preconditioner, and $28n$ with preconditioner. The construction of $G_n$ requires solving $n$ $5 \times 5$ linear systems. The computational complexity for this is approximately $65n$ flops which is a little more than 2 iterations with CGS iterative method. Again we assume use of Gaussian elimination for solving these systems, and we do not take into account similarity of subsystems that might reduce the complexity. In this case the construction of the ILU(0) preconditioner requires approximately $6n$ flops.

![Figure 7.7: A tensor grid with refinement to $x = 1, \psi = 1/2$.]

**Example 7.3.** In this example we consider a grid as in figure 7.7 with refinement toward the line $x = 1$. Let $n_x > 0$ denote the number of gridpoints in the $x$-direction
Numerical results in two dimensions

and let \( f := 1/\psi^{n+1} \). Then the \( x \)-coordinates in figure 7.7 are given by

\[
(7.7) \quad x_i = \frac{f(1-i/(n_x+1)) - f}{1 - f}, \quad \forall i = 0, \ldots, n_x + 1.
\]

Let \( n_y \) denote the number of gridpoints in the \( y \)-directions. The \( y \)-coordinates are then given by \( y_i = i/(n_y + 1), \ i = 0, \ldots, n_y + 1 \). Then \( n = n_x n_y \) is the total number of degrees of freedom. In this example and the following \( n_x = n_y \). We examine the condition numbers and number of CGS iterations for a pure diffusive problem.

<table>
<thead>
<tr>
<th>2-D, number of unknowns (number of refinement steps)</th>
<th>16 (4)</th>
<th>25 (5)</th>
<th>36 (6)</th>
<th>49 (7)</th>
<th>64 (8)</th>
<th>81 (9)</th>
<th>100 (10)</th>
<th>121 (11)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \kappa_2 )-1</td>
<td>62.53</td>
<td>244.71</td>
<td>973.81</td>
<td>3895.24</td>
<td>1.56x10^4</td>
<td>6.24x10^4</td>
<td>2.50x10^5</td>
<td>9.99x10^5</td>
</tr>
<tr>
<td>( \kappa_2 )-2</td>
<td>5.03</td>
<td>11.08</td>
<td>24.68</td>
<td>53.54</td>
<td>111.48</td>
<td>231.74</td>
<td>476.91</td>
<td>974.03</td>
</tr>
<tr>
<td>( \kappa_2 )-3</td>
<td>2.63</td>
<td>3.73</td>
<td>4.72</td>
<td>6.19</td>
<td>7.39</td>
<td>8.98</td>
<td>10.27</td>
<td>11.99</td>
</tr>
<tr>
<td>CGS-1</td>
<td>10</td>
<td>50</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CGS-2</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>15</td>
<td>16</td>
<td>20</td>
<td>21</td>
<td>22</td>
</tr>
<tr>
<td>CGS-3</td>
<td>8</td>
<td>10</td>
<td>11</td>
<td>14</td>
<td>14</td>
<td>18</td>
<td>17</td>
<td>19</td>
</tr>
<tr>
<td>PCGS-1</td>
<td>7</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PCGS-2</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PCGS-3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 7.4: Results for grid 7.7, \( \psi = 1/2 \), diffusion only.

From table 7.4 we observe the following for the spectral condition numbers. \( \kappa_2(A_n) \) grows proportional to \( h^{-2} \) and \( \kappa_2(G_n A_n) \) grows proportional to \( h^{-1} \) which means a reduction by an order of \( h^{-1} \). \( G_n A_n \) seems to be conditioned best with \( \kappa_2(G_n A_n) = O(\log(h^{-1})) \). With respect to convergence of the CGS method we see that the method breaks down for solving with \( A_n \) for \( l \geq 6 \) with \( l \) being the number of refinement steps. Solving the systems with \( A_n G_n \) and \( G_n A_n \) results in an equivalent number of iterations. This changes when an ILU(0) preconditioner is considered. \( A_n \) with ILU(0) breaks down quickly because the factors are unstable. This is also the case for \( A_n G_n \) with additional ILU(0) preconditioner. In contrast, the ILU(0) factorization of \( G_n A_n \) does not show these instabilities and helps to reduce the number of iterations further. Also the growth of the number of iterations for this test seems less than for the test without additional ILU(0) preconditioner.
Chapter 7: NumLab preconditioners

Figure 7.8: A tensor grid with refinement to $x = 1$ and $y = 1$, $\psi = 1/2$.

**Example 7.4.** Next we examine grids as in figure 7.8 with refinement to the lines $x = 1$ and $y = 1$. The $x$-coordinates are as in (7.7) and the $y$-coordinates are computed in the same way. We consider the purely diffusive case first. The results are given in table 7.5.

<table>
<thead>
<tr>
<th>$n \ (l)$</th>
<th>16 (4)</th>
<th>25 (5)</th>
<th>36 (6)</th>
<th>49 (7)</th>
<th>64 (8)</th>
<th>81 (9)</th>
<th>100 (10)</th>
<th>121 (11)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\kappa_1$-1</td>
<td>121.61</td>
<td>506.14</td>
<td>2066.41</td>
<td>8352.85</td>
<td>$3.36 \times 10^4$</td>
<td>$1.35 \times 10^5$</td>
<td>$5.40 \times 10^6$</td>
<td>$2.16 \times 10^6$</td>
</tr>
<tr>
<td>$\kappa_2$-2</td>
<td>8.03</td>
<td>20.89</td>
<td>40.12</td>
<td>92.64</td>
<td>172.70</td>
<td>391.37</td>
<td>718.89</td>
<td>1612.62</td>
</tr>
<tr>
<td>$\kappa_3$-3</td>
<td>2.78</td>
<td>3.83</td>
<td>4.80</td>
<td>5.90</td>
<td>6.92</td>
<td>8.03</td>
<td>9.08</td>
<td>10.19</td>
</tr>
</tbody>
</table>

CGS-1 | 11 | 21 | 37 | - | - | - | - | - |
CGS-2 | 9 | 13 | 14 | 16 | 17 | 19 | 20 | 23 |
CGS-3 | 8 | 11 | 11 | 13 | 15 | 16 | 17 | 18 |
PCGS-1 | - | - | - | - | - | - | - | - |
PCGS-2 | 5 | 7 | - | - | - | - | - | - |
PCGS-3 | 4 | 5 | 5 | 5 | 6 | 6 | 6 | 6 |

Table 7.5: Results for grid 7.8, $\psi = 1/2$, diffusion only.

Next we are interested to see the influence of a convective term $b \cdot \nabla u$. Therefore consider convection in both directions with $b = [100, 100]^T$. In figure 7.9 the corresponding NumLab interactor is shown, with the parameters $x$-convection and $y$-convection set to 100. The results are given in table 7.6.
Despite the difference in differential operators we can make the same observations for both tests. As in example 7.3, $\kappa_2(A_n)$ is proportional to $h^{-2}$, and $\kappa_2(A_nG_n)$ and $\kappa_2(G_nA_n)$ are proportional to $h^{-1}$ and $\log(h^{-1})$ respectively. Also the results for the CGS method are comparable. In both the non-convective and convective case we observe breakdown of the CGS method for solving with $A_n$ for $l \geq 7$. The preconditioned systems do not suffer from this breakdown. Despite the fact that convergence of the CGS method is irregular the results seem to indicate that the number of iterations needed for convergence is approximately proportional to the number of refinement levels $l$. When applying the ILU(0) preconditioner to $A_n$ and $A_nG_n$, we observe also breakdown because of ill-conditioned factors. As before, ILU(0) applied to $G_nA_n$ gives the best results, despite the additional cost for constructing the incomplete factorization.
Example 7.5. To conclude this section we examine a sequence of grids with a brick like refinement as in figure 7.11 with $\psi = 1/2$. This is in fact our problem of interest. The $x$-coordinates for this grid are as in (7.6); the $y$-coordinates are obviously defined similarly. In figure 7.10 we see that the grid parameters, `AccumulationEndPoints`, are set such that we have the required grid. We consider the purely diffusive problem only. The computed spectral condition numbers and number of iterations are given in table 7.7.
Numerical results in two dimensions

<table>
<thead>
<tr>
<th>$n$ ($l$)</th>
<th>9 (1)</th>
<th>25 (2)</th>
<th>49 (3)</th>
<th>81 (4)</th>
<th>121 (5)</th>
<th>169 (6)</th>
<th>225 (7)</th>
<th>289 (8)</th>
<th>361 (9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\kappa_2^{-1}$</td>
<td>4.58</td>
<td>21.63</td>
<td>100.11</td>
<td>431.12</td>
<td>1790.46</td>
<td>7299.92</td>
<td>2.95 $10^4$</td>
<td>1.19 $10^5$</td>
<td>4.75 $10^5$</td>
</tr>
<tr>
<td>$\kappa_2^{-2}$</td>
<td>1.63</td>
<td>3.42</td>
<td>7.87</td>
<td>18.69</td>
<td>42.15</td>
<td>87.39</td>
<td>184.73</td>
<td>373.88</td>
<td>776.25</td>
</tr>
<tr>
<td>$\kappa_2^{-3}$</td>
<td>1.62</td>
<td>2.28</td>
<td>3.20</td>
<td>4.34</td>
<td>5.48</td>
<td>6.63</td>
<td>7.78</td>
<td>8.91</td>
<td>10.02</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>CGS-1</th>
<th>CGS-2</th>
<th>CGS-3</th>
<th>PCGS-1</th>
<th>PCGS-2</th>
<th>PCGS-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n=121$</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>9</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>$n=169$</td>
<td>9</td>
<td>12</td>
<td>11</td>
<td>16</td>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 7.7: Results for grid 7.11, $\psi = 1/2$, diffusion only.

The observations for this example with respect to the spectral condition number are the same as for examples 7.3 and 7.4 and therefore can be omitted. With respect to the convergence of the CGS method the number of iterations for CGS-3 seems to be proportional to the number of refinement levels $l$ as well. We see in this example that the ILU(0) preconditioner applied to $A_{n}$ (PCGS-1) does not break down immediately. We have given the norms of the residuals in figure 7.12 for two grids. From both figures it is clear that the iteration process of the CGS method for solving the unpreconditioned system (CGS-1) is very irregular. Convergence for the two preconditioned systems is much better. In case of CGS-3 convergence is even without irregularities.

Figure 7.12: The norms of residuals computed during the CGS iterative process (cf. table 7.7).
As in the one dimensional case, the left preconditioned system $G_n A_n$ shows the
best results, both with and without an additional ILU(0) preconditioner. For the
reduction of the computational complexity we examine Example 7.5.

![Figure 7.13: The computational complexity for the tests in Example 7.5.](image_url)

Clearly, PCGS-3 performs the best of the three, and CGS-2 and CGS-3 are comparable
to each other as could be seen from the iteration count for both methods. For example
for $n = 121$, CGS-1 needs 70 iterations for convergence. This costs approximately
1200\(n\) flops. CGS-3 performs much better with 420\(n\) flops, despite the construction
of the approximate inverse $G_n$. With 210\(n\) flops PCGS-3 performs most favourable.
Also here the number of iterations hardly seems to grow when the problem size is
increased. Hence, the total workload is primarily dependent on $n$ only which makes
this combination close to optimal.

### 7.4 Numerical results in three dimensions

To conclude this chapter we present several three dimensional examples. Again, to
perform these only a minor alteration needs to be made in our reference network in
figure 7.1.

The grids of interest are once more tensor product grids. The finite difference stenc-
il is a 7 point stencil resulting in a 7-diagonal matrix $A_n$. In this case the cost for
one CGS iteration is $22n$ for the unpreconditioned system, and $36n$ for the precon-
ditioned case. Now construction of $G_n$ requires solving $7 \times 7$ linear systems, and
the total computational cost is around $150n$ which equals approximately 4 CGS it-
erations. However, this more expensive construction of $G_n$ is compensated for by
convergence of the preconditioned linear systems, and the possibility to utilize the
ILU(0) as additional preconditioner. For 7-diagonal matrices constructing the ILU(0)
preconditioner requires approximately $12n$ flops.

**Example 7.6.** First we consider grids with refinements to the planes $x = 1$, $y = 1$ and
$z = 1$, with refinement factor $\psi = 1/2$. For $n_x = n_y = n_z > 0$, the coordinates for
Numerical results in three dimensions

all three directions are defined as in (7.7). The total number of degrees of freedom $n$ is given by $n = n_x n_y n_z$. Again, we examine the spectral condition number of $A_n$, $A_n G_n$ and $G_n A_n$, and consider the performance of the CGS method to solve the corresponding linear systems. First we look at the purely diffusive case. The results for a sequence of grids are shown in table 7.8.

<table>
<thead>
<tr>
<th>$n$ (l)</th>
<th>3-D, number of unknowns (number of refinement steps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>27 (3)</td>
<td>125 (5)</td>
</tr>
<tr>
<td>$\kappa_{2-1}$</td>
<td>28.36</td>
</tr>
<tr>
<td>$\kappa_{2-2}$</td>
<td>3.11</td>
</tr>
<tr>
<td>$\kappa_{2-3}$</td>
<td>1.85</td>
</tr>
</tbody>
</table>

Table 7.8: Results for a 3-D tensor grid with refinement to $x = 1, y = 1, z = 1$, $\psi = 1/2$, diffusion only.

Next we consider the same grid, but now with a convective term $b \cdot \nabla u$ added with $b = [100, 100, 100]^T$. See figure 7.14 for the change made in the NumLab network in order to achieve this. For the same sequence of grids we computed the spectral condition numbers, and solved the three linear systems with the CGS method. The results are in table 7.9.

Figure 7.14: Interactor for changing the equation.
Chapter 7: NumLab preconditioners

3-D, number of unknowns (number of refinement steps)

<table>
<thead>
<tr>
<th>n (l)</th>
<th>27 (3)</th>
<th>125 (5)</th>
<th>343 (7)</th>
<th>729 (9)</th>
<th>1331 (11)</th>
<th>2197 (13)</th>
<th>3375 (15)</th>
<th>4913 (17)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\kappa_2)-1</td>
<td>16.07</td>
<td>166.60</td>
<td>2143.91</td>
<td>3.23 \times 10^4</td>
<td>5.11 \times 10^5</td>
<td>8.16 \times 10^6</td>
<td>1.31 \times 10^7</td>
<td>2.09 \times 10^9</td>
</tr>
<tr>
<td>(\kappa_2)-2</td>
<td>8.16</td>
<td>51.83</td>
<td>271.05</td>
<td>1143.05</td>
<td>4627.43</td>
<td>1.87 \times 10^4</td>
<td>7.53 \times 10^4</td>
<td>3.02 \times 10^5</td>
</tr>
<tr>
<td>(\kappa_2)-3</td>
<td>3.26</td>
<td>6.35</td>
<td>8.77</td>
<td>10.18</td>
<td>10.93</td>
<td>11.59</td>
<td>12.49</td>
<td>13.57</td>
</tr>
<tr>
<td>CGS-1</td>
<td>10</td>
<td>36</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CGS-2</td>
<td>5</td>
<td>9</td>
<td>13</td>
<td>20</td>
<td>23</td>
<td>26</td>
<td>31</td>
<td>35</td>
</tr>
<tr>
<td>CGS-3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>16</td>
<td>18</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>PCGS-1</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PCGS-2</td>
<td>3</td>
<td>6</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PCGS-3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 7.9: Results for a 3-D tensor grid with refinement to \(x = 1, y = 1, z = 1\), \(\psi = 1/2\), diffusion and convection.

One can make similar remarks about these two examples as were made for the examples in one and two dimensions. \(\kappa_2(A_n)\) is once more proportional to \(h^{-2}\), \(\kappa_2(A_nG_n)\) to \(h^{-1}\) and \(\kappa_2(G_nA_n)\) to \(\log(h^{-1})\). Furthermore, the CGS method breaks down for \(A_n\), and ILU(0) is unstable for both \(A_n\) and \(A_nG_n\).

Example 7.7. As a final example we consider tensor grids with refinement to all six borders of the cube. This resembles the discretization of a brick 3-D. In figure 7.18 a network is shown with the modules that create this three dimensional tensor grid. The module GeometryContourCube defines the domain \(\Omega\), in this case \((0,1)^3\). The tensor grid itself is shown as well. The coordinates in all three directions are as in (7.6), for \(n_x = n_y = n_z > 0\). First we look at the diffusive case. The results are given in table 7.10.

3-D, number of unknowns (number of refinement steps)

<table>
<thead>
<tr>
<th>n (l)</th>
<th>27 (1)</th>
<th>125 (2)</th>
<th>343 (3)</th>
<th>729 (4)</th>
<th>1331 (5)</th>
<th>2197 (6)</th>
<th>3375 (7)</th>
<th>4913 (8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\kappa_2)-1</td>
<td>4.58</td>
<td>21.73</td>
<td>100.99</td>
<td>436.25</td>
<td>1815.35</td>
<td>7410.06</td>
<td>2.99 \times 10^4</td>
<td>1.20 \times 10^5</td>
</tr>
<tr>
<td>(\kappa_2)-2</td>
<td>1.60</td>
<td>2.89</td>
<td>5.97</td>
<td>14.20</td>
<td>32.64</td>
<td>68.88</td>
<td>143.50</td>
<td>295.01</td>
</tr>
<tr>
<td>(\kappa_2)-3</td>
<td>1.55</td>
<td>2.22</td>
<td>3.01</td>
<td>4.16</td>
<td>5.36</td>
<td>6.56</td>
<td>7.73</td>
<td>8.85</td>
</tr>
<tr>
<td>CGS-1</td>
<td>4</td>
<td>10</td>
<td>29</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CGS-2</td>
<td>4</td>
<td>8</td>
<td>11</td>
<td>13</td>
<td>16</td>
<td>16</td>
<td>17</td>
<td>19</td>
</tr>
<tr>
<td>CGS-3</td>
<td>4</td>
<td>8</td>
<td>10</td>
<td>11</td>
<td>14</td>
<td>13</td>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>PCGS-1</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PCGS-2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PCGS-3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 7.10: Results for a 3-D tensor grid with brick refinement, \(\psi = 1/2\), diffusion only.

Next we examine the same sequence of grids, but now with a convective term added. Hence let \(b = [100, 100, 100]^T\) (see figure 7.14). The results for these tests are in table 7.11.
Numerical results in three dimensions

<table>
<thead>
<tr>
<th>( n ) (l)</th>
<th>27 (1)</th>
<th>125 (2)</th>
<th>343 (3)</th>
<th>729 (4)</th>
<th>1331 (5)</th>
<th>2197 (6)</th>
<th>3375 (7)</th>
<th>4913 (8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \kappa_2 )-1</td>
<td>4.57</td>
<td>14.75</td>
<td>44.91</td>
<td>137.26</td>
<td>440.70</td>
<td>1513.76</td>
<td>5522.96</td>
<td>2.10 \times 10^4</td>
</tr>
<tr>
<td>( \kappa_2 )-2</td>
<td>1.76</td>
<td>4.38</td>
<td>11.62</td>
<td>28.24</td>
<td>65.32</td>
<td>147.74</td>
<td>319.15</td>
<td>654.33</td>
</tr>
<tr>
<td>( \kappa_2 )-3</td>
<td>1.95</td>
<td>3.47</td>
<td>5.15</td>
<td>6.79</td>
<td>8.26</td>
<td>9.61</td>
<td>10.96</td>
<td>12.18</td>
</tr>
</tbody>
</table>

Table 7.11: Results for a 3-D tensor grid with brick refinement, \( \psi = 1/2 \), diffusion and convection.

For both examples the conclusions and observations for the spectral condition number are similar to those made for the previous examples. We see that the ILU(0) preconditioner applied to \( A_n \) performs slightly better than the problems preconditioned by the approximate inverse. However, for larger problem sizes the latter method is far more robust, as was also seen in the previous examples. Also here PCGS-3 shows the best results, but for the larger sized problems the incomplete factors appear to be ill-conditioned as well.

![Figure 7.15](image_url)  
(a) \( n = 343 \)  
(b) \( n = 729 \)

Figure 7.15: The norms of residuals computed during the CGS iterative process (cf. table 7.10).

For the tests on the purely diffusive problem the norms of the residuals for two choices of grids are shown in figure 7.15. Clearly CGS-3 shows the best convergence...
behaviour, and does not suffer from irregularities. CGS-2 is not much worse than CGS-3, but has some irregularities. The norms of the residuals for CGS-1 behave very irregularly, and in the right hand figure for \( n = 729 \) the iterative process does not even converge. In figure 7.16 the norms of the residuals are shown for two larger sized problems. This clearly shows the smooth convergence of the CGS method for problems CGS-2 and CGS-3. For these two examples the CGS method does not converge for CGS-1, for which hence only part of the iterative process is shown.

![Figure 7.16](image.png)

Figure 7.16: The norms of residuals computed during the CGS iterative process (cf. table 7.10).

To conclude this example we examine the computational complexity of the several tests performed. In figure 7.17 the number of flops is given for CGS-2, CGS-3 and PCGS-3, as these give the best results. Clearly, PCGS-3 has the lowest computational complexity. But in case of the test with convection, the factors of the ILU(0) preconditioner appear to be ill-conditioned for larger sized problems.

For the purely diffusive problem we take a closer look at the problem with \( n = 343 \); we leave the \( n \) in the following for ease of comparison. Then the computational complexity for CGS-3 amounts to \( 500n \) flops. Compared to the \( 640n \) flops for CGS-1 the profit seems not much. However, CGS-1 breaks down eventually. As previously the best results come from PCGS-3 for which the total costs are approximately \( 300n \) flops for this grid. However, as we see in the example for the convection convection problem, PCGS-3 appears to be prone to breakdown for larger sized problems as well. For these problems CGS-3 does not suffer breakdown.
Numerical results in three dimensions

Figure 7.17: The computational complexity for the tests in Example 7.7.
Chapter 7: NumLab preconditioners

Figure 7.18: A NumLab network for a three dimensional grid with brick-like refinement, see Example 7.7.
Conclusions and future work

Preconditioning techniques are important for efficiently solving linear systems of equations. This holds in particular for problems for which standard methods do not work satisfactorily. Among such problems are those for which the non-smooth solution requires employing a computational grid with local refinements. Apart from the need for efficient preconditioners and solution methods, the possibility of testing newly developed tools without too much effort is equally important. This has lead to the NumLab computational platform. Both topics have been discussed in this thesis. In this final chapter we give the major conclusions firsts, followed by several recommendations for future research.

8.1 Conclusions

The major conclusions are the following.

- A Gaussian elimination strategy based on a special node numbering scheme was considered for grids with refinements along a line. This strategy resulted in an $O(n \log n)$ computational complexity and compares well with the fill-in-optimized minimum degree algorithm.

- The standard estimate for the spectral condition number is not sharp for grids with local refinements.

- Related to the brick wall problem we have a convection diffusion equation which after discretization leads to a matrix $A_n$. For this matrix an approximate inverse preconditioner $G_n$ based on Frobenius norm minimization was considered. In a one dimensional setting theoretical estimates for the spectral condition number of a modified version $B_n$ of the left preconditioned matrix $G_n A_n$, we obtained $\kappa_2(B_n) = O(n)$.

Numerical tests showed that this estimate is sharp in one, two and three dimensions. We observed:

\[
\begin{align*}
\kappa_2(A_n) & \doteq O(h^{-2}), \\
\kappa_2(A_n G_n) & \doteq O(h^{-1}), \\
\kappa_2(G_n A_n) & \doteq O(\log(h^{-1})).
\end{align*}
\]
Chapter 8: Conclusions and future work

- In one and in two dimensions right multiplication of $A_n$ by the approximate inverse matrix $G_n$ results in a reducible matrix $A_n G_n$. This matrix can be permuted and written as a $2 \times 2$ block matrix. This gives rise to various recursive algorithms that employ this decoupling. Moreover, when $A_n$ is a discretization of the Laplace operator, so are the two submatrices of $A_n G_n$.

- Solving iteratively the unpreconditioned and preconditioned linear systems showed that the original unpreconditioned problem is prone to break down due to irregular convergence behaviour. Both preconditioned problems seem to perform similarly. Additionally, an incomplete factorization of the matrix $A_n$ results in ill-conditioned factors. Hence, solving the original problem preconditioned by ILU(0) is subject to breakdown as well. This is also seen when the right preconditioned system with $A_n G_n$ is (additionally) preconditioned by an incomplete factorization. In contrast, solving the left preconditioned system shows much improvement when preconditioned by ILU(0). This combination of an approximate inverse preconditioner and an incomplete factorization appears to be the best strategy with respect to computational complexity. However, for large problem sizes this incomplete factorization gives ill-conditioned factors as well. Because of this phenomenon the right preconditioned system by the approximate inverse preconditioner is considered to be most robust overall.

- The NumLab workbench offers a powerful tool to run numerical simulations. Given the generic interface and the implementation of mathematical concepts as operators, there is a maximum amount of flexibility and freedom in changing the various modules that together constitute a simulation.

8.2 Directions for the future

For both major topics some recommendations for future work can be given. As for the mathematical component there are several aspects that need further research. In Chapter 2 only complete Gaussian elimination was considered, which gave rise to a direct solver. Of interest is to examine the effectiveness of incomplete factorizations (ILU) (Subsection 3.4.3) employed as preconditioner for the numbering scheme presented in that chapter. Given the fact that the fill-in is almost optimal for a complete factorization, incomplete factorization methods of the ILU($p$) and ILUT variant are thought to be beneficial.

Secondly, for the approximate inverse preconditioner discussed in Chapter 4 we only gave theoretical results for the one dimensional case. Of interest is to examine in detail such preconditioners in higher dimensions and establish estimates for the (spectral) condition numbers of the matrices $A_n G_n$ and $G_n A_n$. Furthermore, the decoupling phenomenon in Chapter 5 should be exploited further, especially in two dimensions. Another point for future work is examination of the approximate inverse preconditioner constructed as an approximate left inverse, i.e., $G_n A_n \approx I_n$.

With respect to the NumLab workbench there are several steps in front of us to-
ward an even more complete workbench. A first start is to merge available (parallel) Finite Element assemblers (such as SEPRAN), integrating problem-specialized iterative solvers and preconditioners.

Of particular research interest are the optimal multigrid/multilevel type solvers (see Section 3.4). These can be used within the NumLab framework, but right now no special support is provided. In order to offer convenient multigrid/multilevel solver modules, a few modules need to be extended and others must be added. The grid and solution data types must be extended to handle internal stacks of grids and solutions. Restriction and prolongation require new modules, and application-specific preconditioners are desirable. The strong coupling between the grid and basic iterative solvers need not be a problem: operator evaluation can be grid based.

From a practical point of view, solving large PDE problems in NumLab is still slower than specialized toolkits. This is due to the generic nature of the NumLab modules that cannot make assumptions about specific data storage or discretization properties provided by other modules. This problem can be tackled in several ways: implementing less generic (optimized) modules, re-engineering the generic modules’ implementations to make more extensive use of data caching, or parallelizing the numerical code. A second limitation involves the need to program new Operator subclasses, e.g., to model new PDEs (see Section 6.5.4). A better approach would be to design generic Operators that accept their definition via a symbolic, interpreted notation. Implementing such generic Operators would raise the same efficiency problems outlined above.

Finally, from a technical point of view, separating the address space of the graphical editor from the address space where all modules execute should be another future goal. In this way NumLab applications will be more stable and less prone to break down when contributed research-modules fail. Also, the computational modules and visualization viewer module should be run on different threads so data can be visualized while computations continue.
Appendix A

Pentadiagonal Toeplitz matrices

In this appendix we give an result from [52] related to pentadiagonal Toeplitz matrices.

First note that the product of two Toeplitz matrices is close to, but not a Toeplitz matrix. Let $A$ and $B$ be two $5 \times 5$ matrices given by

$$A = \begin{bmatrix}
  x & 1 & 0 & 0 & 0 \\
  1 & x & 1 & 0 & 0 \\
  0 & 1 & x & 1 & 0 \\
  0 & 0 & 1 & x & 1 \\
  0 & 0 & 0 & 1 & x
\end{bmatrix}, \quad \text{and} \quad B = \begin{bmatrix}
  y & 1 & 0 & 0 & 0 \\
  1 & y & 1 & 0 & 0 \\
  0 & 1 & y & 1 & 0 \\
  0 & 0 & 1 & y & 1 \\
  0 & 0 & 0 & 1 & y
\end{bmatrix}.$$  

Then for the product we have

$$AB = \begin{bmatrix}
  1 + xy & x + y & 1 & 0 & 0 \\
  x + y & 2 + xy & x + y & 1 & 0 \\
  1 & x + y & 2 + xy & x + y & 1 \\
  0 & 1 & x + y & 2 + xy & x + y \\
  0 & 0 & 1 & x + y & 1 + xy
\end{bmatrix}$$

which is not a Toeplitz matrix.

Then we have a result from [52]. Let $a, b, x \in \mathbb{C}$. Assume that

$$A_n(a,b) = \begin{bmatrix}
  a & b & 1 & 0 & 0 \\
  b & a & b & 1 & 0 \\
  1 & b & a & b & 1 \\
  0 & 1 & b & a & b \\
  0 & 0 & 1 & b & a \\
  & & & & \\
\end{bmatrix} \in \mathbb{R}^{n \times n},$$
and that

\[ B_n(x) = \begin{bmatrix} x & 1 & 0 & 0 \\ 1 & x & 1 & 0 \\ 0 & 1 & x & 1 \\ 0 & 0 & 1 & x \\ \vdots & \ddots & & \ddots \end{bmatrix} \in \mathbb{R}^{n \times n}, \]

Assume that \( b^2 \geq 4(a - 2), 2b \leq a + 2 \) and \( a > 2b - 3/2 \), and that \( A_n(a, b) = B_n(x) \circ B_n(y) - E_n \), where \( E_n \) is zero, except for entries \( [E_n]_{1,1} = [E_n]_{n,n} = 1 \). Then

\[ \|A_n^{-1}\|_\infty \leq \|B_n(x)^{-1}\|_\infty \cdot \|B_n(y)^{-1}\|_\infty \]

and

\[ \lim_{n \to \infty} \|A_n^{-1}\|_\infty = \frac{1}{(x - 2)(y - 2)}. \]
Appendix B

Turing completeness

This appendix demonstrates that with the addition of certain basic modules, the NumLab visual programming language of Chapter 6 is Turing complete. In Section B.1 the concepts of alphabet and language are introduced first, followed in Section B.2 by Turing machines. Then Section B.3 presents the NumLab module design for primitive recursive functions. The last Section B.4 presents the NumLab module design for μ-recursive functions. Able to imitate μ-recursive functions, NumLab is Turing complete.

B.1 Alphabets and language

The definitions of string, alphabet and language are taken from [65, pp. 29–31]:

Definition B.1.

- An alphabet is finite set of symbols, such as the Roman alphabet \{a, b, c, \ldots , z\} or the binary alphabet \{0, 1\}.
- A string over an alphabet \(\Sigma\) is a finite sequence of symbols from alphabet \(\Sigma\).
- The set of all strings - including the empty string - over an alphabet \(\Sigma\) is denoted by \(\Sigma^*\).

These definitions permit the definition of a language:

Definition B.2. A language is set of strings over an alphabet \(\Sigma\).

Note that in particular a language over \(\sigma\) is a subset of \(\Sigma^*\). Turing machines are defined using languages.

B.2 The Turing machine

We start by introducing the concept of a Turing machine. We describe a Turing machine keeping figure B.1 in mind.

A Turing machine consists of a tape and a finite-state machine, called control unit. The control unit disposes over a head to read from and/or write to the tape. In each step, the control unit reads the tape and then performs the following tasks:
Appendix B: Turing completeness

3.1 Finite control

Put the control unit in a new state;

Either:

- Write a symbol on the current square on the tape or
- Move the read/write head one position to the left (‘L’) or to the right (‘R’).

The tape has a left end, but is unbounded on the right side. However, in a finite numbers of steps, the machine can only visit a finite number of squares on the tape. (In case the machine tries to move its head to the left off the end of the tape, it ceases to operate).

Initially, the tape contains only symbols at the left end. The rest of the tape consists of blank symbols. The machine is free to alter its input or write on the blank end of the tape. The message (data) left at the end of the computation is called the answer. The end of computations is reached when the control unit reaches the halt state. The blank symbol will be denoted by #.

As an example, consider figure B.2. The Turing machine shown computes \( f(x) = \sin(x) \). Here \( x = 3.14 \) is the input the tape. So, to start with, the tape contains the
The Turing machine symbols 3, 4, 1 and 4, and the control unit is in state \( s_1 \). At the end of the computations, the control unit is in the halt state \( h \) and has left behind the answer \( 0 \) on the tape.

The following definition of a Turing machine comes from [65]:

**Definition B.3.** A Turing machine is a quadruple \( (K, \Sigma, \delta, s) \) where

- \( K \) is a finite set of states, not containing the halt state denoted by \( h \);
- \( \Sigma \) is an alphabet, containing the blank symbol \( \# \), but not containing the symbols \( L \) and \( R \);
- \( s \in K \) is the initial state;
- \( \delta \) is a function from \( K \times \Sigma \) to \( (K \cup h) \times (\Sigma \cup \{L, R\}) \).

With the use of this definition, we can define when a (mathematical) function is Turing computable – required to prove the Turing completeness of NumLab:

**Definition B.4.** Turing computable functions:

Let \( \Sigma_0 \) and \( \Sigma_1 \) be alphabets not containing the blank symbol \( \# \). Let \( f \) be a a function from \( \Sigma_0^* \) to \( \Sigma_1^* \). A Turing machine \( M = (K, \Sigma, \delta, s) \) is said to compute \( f \) if \( \Sigma_0, \Sigma_1 \subseteq \Sigma \) and for any \( w \in \Sigma_0 \), if \( f(w) = u \) then

\[
(s, \#w\#) \xrightarrow{\delta}^* (h, \#u\#).
\]

(B.1)

If such a Turing machine \( M \) exists, then \( f \) is said to be a Turing computable function.

Because a Turing machine can carry out any computation that can be carried out by any similar type of automata, and because these automata seem to capture the essential features of real computing machines, we take the Turing machine to be a precise formal equivalent of the intuitive notion of “algorithm”. Following Church’s Thesis or Church-Turing’s Thesis, nothing will be considered an algorithm if it cannot be rendered as a Turing machine. It is a thesis, not a theorem, because it is not a mathematical result: It simply asserts that a certain informal concept corresponds to a certain mathematical object. It is theoretically possible, however, that Church’s Thesis could be overthrown at some future date, if someone were to propose an alternative model of computation that was publicly acceptable as fulfilling the requirement of “finite labour at each step” and yet was provably capable of carrying out computations that cannot be carried out by any Turing machine. No one considers this likely.

A language is called Turing complete, if it can generate all Turing computable functions (see also [65]). In this small section, it is shown that NumLab is Turing complete, if a few fundamental modules are added.
Appendix B: Turing completeness

B.3 Primitive recursive functions

The primitive recursive functions are defined by three types of initial functions and two combining rules. These can all be presented in a straightforward manner.

**Definition B.5.** The initial functions are the following three functions:

- The 0-place function $\zeta$ is the function from $\mathbb{N}^0$ to $\mathbb{N}$ such that

  \[
  \zeta() = 0.
  \]

- Let $k \geq 1$ and let $1 \leq i \leq k$. Then the $i$-th $k$-place projection function $\pi^k_i$ is the function from $\mathbb{N}^k$ to $\mathbb{N}$ such that

  \[
  \pi^k_i(n_1, \ldots, n_k) = n_i, \quad \text{for any } n_1, \ldots, n_k \in \mathbb{N}.
  \]

**Remark B.6.** Point of notation: Hereafter we write $\bar{n}$ for the $k$-tuple $(n_1, \ldots, n_k)$. Thus the above statement would be rewritten

  \[
  \pi^k_i(\bar{n}) = n_i, \quad \text{for any } \bar{n} \in \mathbb{N}^k.
  \]

- The successor function $\sigma$ is the function from $\mathbb{N}$ to $\mathbb{N}$ such that

  \[
  \sigma(n) = n + 1, \quad \text{for any } n \in \mathbb{N}.
  \]

We introduce the related three NumLab basic modules, and an additional one. These modules are:

- A zero module;
- An increment module;
- A decrement module;
- A decision or switch module,

and shown in figure B.3.

![Figure B.3: Basic elements](image)

Obviously, the zero module itself is already the 0-place function $\zeta$. Figure B.4 shows the zero element from the initial functions
Furthermore the increment module acts as the successor function $\sigma$. Using these two modules, we already dispose over the natural numbers, by consecutive incrementing 0.

![Figure B.4: Initial function zero module](image)

The $k$-place projection is shown in figure B.5. The figure shows on the left side a $k$-place projection function $\pi^k$. In this particular case, three input values $n_1$, $n_2$ and $n_3$ are entered into the module. The 2-projection selects the second component out of three input values.

![Figure B.5: The 2-projection module](image)

On the right hand side, it shows how we built the $k$-place projection function $\pi^k$ from the axioms before. The module first selects $n_2$ out of $n_1$ and $n_2$, by setting the right value 0 on the decision module. Next, it selects $n_2$ from $n_2$ and $n_3$ in the same manner but now by setting a 1 on the second decision module. Finally, value $n_2$ is exporting to the outside.

As an example, we provide a possible NumLab implementation of the decision module, using pseudo code:
class decision: public module
{
    void set(double *v) { this->v = v; }
    ...
    double update()
    {
        return (v->update()) ? then->update() : else->update();
    }
private:
    module *v;
    ...
}

We now proceed with the definitions of composition and Primitive Recursion

Definition B.7.

- Let \( l > 0 \) and \( k \geq 0 \), let \( g \) be an \( l \)-place function, and let \( h_1, \ldots, h_l \) be \( k \)-place functions. Let \( f \) be the \( k \)-place function such that, for every \( \bar{n} \in \mathbb{N}^k \),

\[
(B.6) \quad f(\bar{n}) = g(h_1(\bar{n}), \ldots, h_l(\bar{n})).
\]

Then \( f \) is said to be obtained from \( g, h_1, \ldots, h_l \) by composition.

- Let \( k \geq 0 \), let \( g \) be an \( k \)-place function, and let \( h \) be a \((k + 2)\)-place function. Let \( f \) be the \((k + 1)\)-place function such that for every \( \bar{n} \in \mathbb{N}^k \),

\[
(B.7) \quad f(\bar{n}, 0) = g(\bar{n})
\]

and for every \( \bar{n} \in \mathbb{N}^k \) and \( m \in \mathbb{N} \)

\[
(B.8) \quad f(\bar{n}, m + 1) = h(\bar{n}, m, f(\bar{n}, m))
\]

Then \( f \) is said to be obtained from \( g \) and \( h \) by primitive recursion.

Definition B.8. A function is said to be a primitive recursive function if it is an initial function or can be generated from the initial functions by some sequence of operations of composition and primitive recursion. More succinctly, the primitive recursive functions are the smallest class of functions containing the initial function and closed under composition and primitive recursion.

Because all primitive recursive functions terminate, the set of all primitive recursive functions cannot represent the set of all Turing computable functions. Therefore, in order to obtain all computable functions, some extension must be made to the methods used thus far for defining functions.
B.4 $\mu$-Recursive functions

This section introduces $\mu$-recursive functions and presents a visual module design for their NumLab implementation. Because the functions in the set of $\mu$-recursive functions can imitate all Turing machines, NumLab is Turing complete with the addition of this type of module. First, we must define the concept of unbounded minimalisation:

**Definition B.9.** Let $k \geq 0$ and let $g$ be a $(k+1)$-place functions. Then the unbounded minimalisation of $g$ is that $k$-place function $f$ such that, for any $\bar{n} \in \mathbb{N}^k$

\[
\begin{align*}
\text{(B.9)} \\
\qquad f(\bar{n}) = \begin{cases} \\
\text{the least } m \text{ such that } g(\bar{n}, m) = 0 \text{ if such } m \text{ exists;} \\
0 \text{ otherwise.}
\end{cases}
\end{align*}
\]

The second clause guarantees that $f$ is everywhere defined, regardless of what $g$ is. We write

\[
\begin{align*}
\text{(B.10)} \\
\qquad f(\bar{n}) = \mu m [g(\bar{n}, m) = 0]
\end{align*}
\]

and say that $f$ is obtained from $g$ by unbounded minimalisation.

In general, the unbounded minimalisation of a primitive recursive function need to be primitive recursive, or indeed computable in any intuitive sense. The reason, as we shall show later, is that there is no general method of telling whether an $m$ of the required type exists. However, if $g$ has the property that such an $m$ exists for every $\bar{n}$, then $f$ is computable if $g$ is computable: Given $\bar{n}$, we simply need to evaluate all of $g(\bar{n}, 0), g(\bar{n}, 1), \ldots$ until we find $m$ such that $g(\bar{n}, m) = 0$. However, in this case $f$ need not, in general, be primitive recursive.

These ideas leads to the definition of regular functions:
Appendix B: Turing completeness

**Figure B.7**: The NumLab design of a primitive-recursive module

**Definition B.10.** A \((k + 1)\)-place function \(g\) is called a regular function if and only if, for every \(\bar{n} \in \mathbb{N}^k\), there is an \(m\) such that \(g(\bar{n}, m) = 0\). A function is \(\mu\)-recursive if and only if it can be obtained from the initial functions \(\zeta, \pi^k_1, \) and \(\sigma\) by the following operations:

- composition
- primitive recursion
- application of unbounded minimalisation to regular functions.

With this definition, each primitive recursive function is also \(\mu\)-recursive.

Figure B.8 shows the NumLab module design for a \(\mu\)-recursive function.

In order to finish this appendix, we refer to [65], which shows that the set of all \(\mu\)-recursive functions can imitate all Turing machines (so all computable functions). Thus, because NumLab can generate modules for all \(\mu\)-recursive functions, NumLab is Turing complete.
Figure B.8: The NumLab design of a $\mu$-recursive module


Bibliography


Index

alphabet .......................... 129
AMLI ............................... 33
approximate inverse ............... 35
  AINV ............................. 37
  Frobenius norm minimization 35
  FSAI .............................. 37
  SPAI .............................. 36
Bi-CG ............................. 25
blank symbol ...................... 130
BPX ............................... 33
CG ................................. 25
CGS ................................ 26
Church’s Thesis .................... 131
Church-Turing’s Thesis ............ 131
composition ........................ 134
computational complexity ......... 27
control unit ....................... 129
diagonally oriented grid ......... 71
equilibration ....................... 39
fill-in path ....................... 12
finite-state machine ............. 129
flops ................................ 27
Green’s functions ................ 35
Green’s matrix ................... 35
halt state .......................... 131
imitate ............................ 135
incomplete factorizations ....... 34
iterative methods
  Bi-CG ............................. 25
  CG ................................. 25
  CGS ............................... 26
  SOR ............................... 24
Jacobi matrix ..................... 24
Krylov subspace .................. 25
Krylov subspace method ......... 25
language ........................... 129
Left-right (LR) ordering .......... 10
maximal monotone ............... 89
Minimum degree ordering ....... 10
multigrid .......................... 33
network editor ..................... 3
nonstationary iterative methods . 23
optimal order method .......... 28
preconditioners
  AMLI ............................. 33
  BPS ............................... 34
  BPX ............................... 33
  incomplete factorizations .... 34
  multigrid ....................... 33
  preconditioning ............... 27, 31
  explicit ......................... 32
  implicit ......................... 32
  Primitive Recursion .......... 134
  primitive recursive function .. 134
regular function .................. 136
residual ............................ 23
Reverse Cuthill-McKee ordering . 10
Ritz-Galerkin approach .......... 25
<table>
<thead>
<tr>
<th>Index</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOR</td>
<td>24</td>
</tr>
<tr>
<td>spectral condition number</td>
<td>25</td>
</tr>
<tr>
<td>spectral radius</td>
<td>24</td>
</tr>
<tr>
<td>stationary iterative methods</td>
<td>23</td>
</tr>
<tr>
<td>string</td>
<td>129</td>
</tr>
<tr>
<td>suboptimal order method</td>
<td>28</td>
</tr>
<tr>
<td>tape</td>
<td>129</td>
</tr>
<tr>
<td>tie-breaking strategy</td>
<td>8</td>
</tr>
<tr>
<td>Toeplitz matrix</td>
<td>45</td>
</tr>
<tr>
<td>total workload</td>
<td>27</td>
</tr>
<tr>
<td>Turing complete</td>
<td>131</td>
</tr>
<tr>
<td>Turing machine</td>
<td>129</td>
</tr>
<tr>
<td>unbounded minimalisation</td>
<td>135</td>
</tr>
<tr>
<td>visualization</td>
<td></td>
</tr>
<tr>
<td>steering</td>
<td>1</td>
</tr>
<tr>
<td>tracking and monitoring</td>
<td>1</td>
</tr>
</tbody>
</table>
Summary

Often solving partial differential equations in engineering and sciences is complicated by the fact that their solutions exhibit sharp peaks or fronts. Any suitable discretization method for numerically solving such problems should use a grid taking this activity into account. As a result the mesh is usually quite far from being uniform. This in turn causes difficulties when solving the linear systems that arise from such discretized problems; often as a step in a Newton process when the problem at hand is non-linear. The resulting system is said to be ill-conditioned. Typically the solution is found by an iterative method. If the problem is ill-conditioned the convergence (if at all) will be slow.

In order to speed up convergence so called preconditioning techniques are widely used. We consider a class of problems, related to moisture and salt ion transportation in a brick wall, for which no effective preconditioner exists as of yet. This then gives the first objective of this thesis: to construct techniques that are able to tackle this problem. A first approach is based on Gaussian elimination for a particular numbering scheme for the degrees of freedom related to refinement along a line. Secondly we consider preconditioning techniques based on approximating the inverse. The advantage of this type of preconditioning is that application requires only matrix-vector multiplications. More importantly, for the problem above mentioned standard preconditioning techniques based on incomplete factorizations are prone to break down. For this approximate inverse preconditioner we derive several estimates for the spectral condition number in the one dimensional case. This analysis is based on a slightly modified variant so that Toeplitz matrix theory can be used. A further analysis of this preconditioner shows that the degrees of freedom decouple into two sets. In the one dimensional case we derive an algorithm based on recursive application of this decoupling after applying this approximate inverse preconditioner. For two dimensions this cannot be generalized straightforwardly and we provide suggestions for a recursive application of the approximate inverse.

Besides the need for efficient solvers, another important aspect is to have a computational platform to run those solvers and perform numerical simulations. This is our second objective. A suitable computational platform should enable the researcher to use existing software and offer the possibility to extend the platform with newly made software components for future use. Furthermore, there should be the possibility to track and steer the simulation at runtime. This thesis describes the so called NumLab environment that enables the researcher to implement and test new ideas.
by reusing existing components and building new components from scratch or from existing components. Mathematical concepts like operators and linear solvers are implemented using a generic interface, which enables interactive change of components and parameters.

We illustrate the foregoing by a series of numerical examples. The approximate inverse preconditioner is tested using the NumLab environment. The numerically computed spectral condition numbers turn out to agree nicely with the theoretical estimates. To illustrate the robustness our results are compared to an incomplete factorization preconditioner. The latter breaks down, as does the iterative solver applied to the linear system of equations without preconditioning.
Samenvatting

Het oplossen van partiële differentiaalvergelijkingen uit technische en wetenschappelijke toepassingen wordt vaak bemoeilijkt doordat de oplossingen op bepaalde gebieden scherpe pieken of fronts vertonen. Elke geschikte discretisatiemethode om zulke problemen numeriek op te lossen, behoort daartoe een rekenrooster te kiezen dat deze kenmerken in ogenschouw neemt. Als gevolg hiervan is het rooster doorgaans verre van uniform. Dit op zijn beurt geeft moeilijkheden wanneer het lineaire stelsel dat volgt uit de discretisatie opgelost moet worden; vaak als stap in een Newton proces als het probleem in kwestie niet-lineair is. Het resulterende systeem wordt dan ook wel slecht geconditioneerd genoemd. In het algemeen wordt de oplossing gevonden met behulp van een iteratieve methode. Als het probleem slecht geconditioneerd is, dan zal de convergentie (zo die er al is) traag zijn.

Om de convergentie te versnellen wordt veelvuldig gebruikt gemaakt van zogenaamde preconditioneringstechnieken. Wij bekijken een klasse van problemen, geassocieerd aan vocht- en zouttransport in een muur van bakstenen, waarvoor op dit moment nog geen effectieve preconditioneerder bestaat. Dit geeft het eerste doel van dit proefschrift: het construeren van technieken die in staat zijn deze problemen aan tepakken. Een eerste aanpak is gebaseerd op Gauss eliminatie voor een specifieke nummering van de onbekenden die bij een verfijning langs een lijn horen. Ten tweede bekijken we preconditioneringstechnieken die gebaseerd zijn op het benaderen van de inverse. Het voordeel van dit type preconditioneerders is dat de toepassing ervan alleen matrix-vector vermenigvuldigingen vereist. Nog belangrijker is dat voor het bovengenoemde probleem standaard preconditioneringstechnieken gebaseerd op onvolledige factorisatie de neiging tot mislukken hebben. Voor deze benaderde inverse preconditioneerder leiden we voor het één dimensionale geval enkele schattingen voor het spectrale conditiegetal af. Deze analyse maakt gebruik van een gedeeltelijk aangepaste variant opdat Toeplitz matrixtheorie toegepast kan worden. Verdere bestudering van deze preconditioneerder leert dat de onbekenden ontkoppelen in twee verzamelingen. We leiden voor het ééndimensionale geval een algoritme af dat gebruik maakt van het recursief toepassen van deze ontkoppeling na toepassing van de benaderde inverse preconditioneerder. In twee dimensies in dit niet rechtstreeks generaliseerbaar. We geven enkele voorstellen voor het recursief gebruik van de benaderde inverse.

Naast de behoefte aan efficiënte oplosmethoden is het hebben van een rekenplatform voor het gebruiken van die oplosmethoden en het uitvoeren van numerieke simu-
latie een ander belangrijk aspect. Dit is ons tweede doel. Een geschikt rekenplatform zou de onderzoeker in staat moeten stellen om bestaande programmatuur te gebruiken en nieuw ontwikkelde programmatuur voor toekomstig gebruik aan het platform toe te voegen. Verder moet er ook de beschikking zijn over middelen om de simulatie te volgen en te sturen terwijl deze loopt. Dit proefschrift beschrijft de zogeheten NumLab-omgeving die de onderzoeker in staat stelt om nieuwe ideeën te implementeren en te testen door bestaande componenten te hergebruiken en nieuwe componenten te bouwen uit het niets of uit bestaande delen. Wiskundige concepten zoals operatoren en lineaire oplossers zijn geïmplementeerd met gebruikmaking van een generieke interface. Dit staat het interactief veranderen van componenten en parameters toe.

Het voorgaande illustreren we met een reeks numerieke voorbeelden. De benaderde inverse preconditioneerder wordt getest met gebruikmaking van de NumLab-omgeving. De numeriek berekende spectrale conditiegetallen zijn goed in overeenstemming met de theoretische schattingen. Om de robuustheid te illustreren vergelijken we onze resultaten met een onvolledige factorisatie preconditioneerder. De laatste blijkt niet robuust, net als de iteratieve oplosser toegepast op het ongepreconditioneerde lineaire stelsel van vergelijkingen.
Curriculum Vitae

The author of this thesis was born on January 9th, 1976 in the city of Nijmegen. The greater part of his childhood was spent in Deest, a small village on the south bank of the river Waal between Nijmegen and Tiel. After elementary school he went to the Pax Christi College in Druten in 1987 where he obtained his VWO diploma in 1993. Following this pre-university education the author went to study mathematics at the University of Nijmegen. He specialized in the subject of numerical analysis, with a strong component of functional analysis. He also followed several courses in Latin, and participated in a drama group called Moira. In June 1998 he finished his master’s thesis on the subject of least-squares methods for first order systems under supervision of prof. Axelsson. On August 1st, 1998 he assumed a PhD position at Eindhoven University of Technology in the Scientific Computing Group. Several results of his research are written in this thesis. As part of this research the 7th Copper Mountain Conference on Iterative Methods was attended in 2002.

Next to being a mathematician, Drenth has a strong interest in historical matters as well. This resulted in a cooperation with several others from around the world to study the history of the British Army. The author has a particular interest in the reserve forces of the British Army. His research in this field will result in a series of books to be published from 2005 on.