Everything You Always Wanted to Know About FLENS, But Were Afraid to Ask

Michael Lehn [1]

February 2, 2008

[1]Ulm University, Institute for Numerical Mathematics

Contents

1 Introduction

Scientific computing is concerned with the construction of mathematical models, numerical solution techniques and the application of computers to simulate and optimize scientific and engineering problems. Within the last 40 years, scientific computing has changed dramatically. Vast advances were made in the development of hardware architectures. However, despite the steady increment of available computing power, the major breakthroughs in scientific computing were attained after the advent of modern numerical methods. This progress made it possible for the complexity of models to be increased enormously. As a result, nowadays certain scientific problems can be simulated with an accuracy that was unthinkable just a decade ago. In its course of evolution, scientific computing has become indispensable in more and more fields, such as finance and medicine.

The insatiable greed for treating more and more complex problems requires the continuous advancement of numerical methods. Further, the implementation of a modern numerical algorithm has to be efficient and is expected to achieve peak performance on the available hardware resources. For these reasons the development of software is an essential component within scientific computing. The proper realization and design of such software requires an interdisciplinary cooperation of experts from at least three domains:

1. Experts from the **underlying problem domain** are crucial for the adequate creation of mathematical models. These experts are not necessarily specialists in high-performance computing nor numerical analysis. However, they have to be able to apply (and to some degree extend) the software for their research.

2. Specialists in **high-performance computing (HPC)** who are capable of implementing a numerical algorithm such that the available hardware can be exploited to achieve peak performance. Scientific software has to be designed in such a way that these highly optimized implementations can be integrated easily.

3. In order to take advantage of new advances in **numerical analysis**, it is crucial for a scientific software package to enable researchers in these fields to contribute state of the art methods. In this respect, rapid adoption of such methods can only be expected if the implementation can be accomplished easily. In particular, it must not require a deep knowledge of the complete software package.

Typical tasks in the development of scientific software are related to all these domains; for example, the implementation of a modern numerical method, subsequently tuning this implementation and finally applying the method to a concrete problem. Using numerical methods and reference implementations from popular textbooks is often not sufficient for the development of serious scientific software. In general, state of the art algorithms are not covered by these books and neither are the corresponding implementations suited to achieve high-performance. Software packages like Mathematica [90], Maple [48] and Matlab [49] provide high-level programming languages that can support the initial development of new numerical methods and the rapid implementation of prototypes. However, these standard packages are often not sufficient as HPC kernels and neither are they intended for the realization of scientific software that has to deal with large scaled problems on an HPC plattform. Achieving optimal efficiency and scalability requires the adaption of algorithms to a specific problem. This in particular includes the adaption of data structures to exploit special features of the problem at hand. Furthermore, taking full advantage of the underlying hardware often requires the complete re-implementation of numerical routines in low-level programming languages. Ideally, it should be possible for experts to contribute to the development of scientific software while working solely on their own area of expertise. In order to allow such a division of work and to enhance the developmental process advanced tools and techniques are required.

Object-oriented programming languages like C++ exhibit features that can ease the development, maintainability, extensibility and usability of complex software packages. However, most libraries in the field of high performance computing are still implemented in non-object-oriented languages like Fortran or C. This is mainly due to the fact that runtime overhead resulting from abstraction is sometimes hard to avoid. At this point it should be mentioned that in the scientific computing community (and especially in the field of high-performance computing) C++ used to have a bad reputation due to a bad runtime performance. In particular, C++ libraries could often hardly compete with established Fortran libraries.

This work addresses how the C++ programming language can be extended through libraries to enhance and simplify the development of scientific software. Efforts made in this respect resulted in the design and implementation of the C++ library **FLENS** (*Flexible Library for Efficient Numerical Solutions*) [45] as presented in this work. Further, concepts and techniques that allow the effective collaboration of specialists from different disciplines within the development of scientific software will be demonstrated based on FLENS. The performance issue mentioned above was in particular addressed throughout the design and implementation of FLENS.

This work is organized as follows. **Chapter 2** briefly summarizes related work that influenced the design and implementation of FLENS. **Chapter 3** addresses the challenge of utilizing external high-performance libraries that are implemented in non-object-oriented languages. In particular, this chapter illustrates how low-level abstractions for data structures and routines can enhance the usability of these libraries without sacrificing efficiency. **Chapter 4** illustrates the elementary concepts that were realized in the design of the FLENS library to achieve extensibility, flexibility, reusability and ease of use without sacrificing performance. Techniques for extending FLENS as well as concepts for the implementation of flexible and reusable methods based on FLENS, are briefly outlined in **Chapter 5**.

Considering some standard numerical problems, the techniques indicated in chapter 5 will be illustrated in more detail in the following chapters. **Chapter 6** illustrates the implementation of numerical solvers for the Dirichlet-Poisson problem. Most notable in this respect is the implementation of the multigrid method presented in this chapter. The possibility of attaining parallelization without modifying the implementation of the multigrid method makes the potential of FLENS vivid. In order to further demonstrate the reusability, the implementation of the multigrid method will be applied in **Chapter 7** to numerically solve the Navier-Stokes equations in two space dimensions on a simple domain. The subject of **Chapter 8** is the implementation of the Finite Element Method for numerically solving the Dirichlet-Poisson Problem on bounded Lipschitz domains with piecewise polygonal boundaries. For this toy example, the chapter presents a compact and expressive implementation and hereby indicates how FLENS could be utilized in teaching. Furthermore, sparse matrix types in FLENS and solvers for sparse systems of linear equations are illustrated.

Conclusions in **Chapter 9** round off this work. Up-to-date benchmark results and the complete source code of FLENS (including all the examples illustrated in this work) are available on

`http://flens.sf.net/`

2 Related Work

The development of FLENS was influenced in various ways by other works as outlined in the following. These works originate from different domains which loosely constitute the sections of this chapter.

Section 2.1 summarizes libraries and techniques from the field of high performance computing, while Section 2.2 introduces software packages that provide level-languages especially designed for numerical and scientific applications. As briefly outlined in Section 2.3, a gainful utilization of object-oriented languages for scientific computing is no trivial task. Further, libraries that have contributed notable techniques to overcome certain issues are introduced in this section. Section 2.4 is dedicated to research focused on improving and ensuring software quality. Section 2.5 highlights the origin of the FLENS library. Finally, Section 2.5 outlines how FLENS could be utilized by other libraries which are well established in the field of scientific computing.

2.1 Libraries for High Performance Computing (HPC)

In the field of HPC various libraries exist that are well established and successful in this domain. In the following, different HPC libraries are summarized that influenced the development and design of FLENS.

2.1.1 BLAS and LAPACK

BLAS (Basic Linear Algebra Subprograms) [58] specifies a set of kernel routines for linear algebra operations such as vector and matrix multiplications. Although a reference implementation exists, BLAS actually constitutes an application programming interface. BLAS became a de facto standard in the field of scientific and high performance computing for which various implementations exist. Nowadays highly optimized BLAS implementations for all relevant hardware architectures are available. Thus, using BLAS routines as building blocks for scientific software ensures portability. Further, software based on BLAS can achieve peak performance on every platform.

LAPACK (Linear Algebra PACKage) [3] is a Fortran library for numerical computing. Among others, LAPACK provides routines for solving systems of linear equations, least-square problems, eigenvalue and singular value problems. Almost all LAPACK routines make use of BLAS to perform critical computations. This enables LAPACK to provide optimal performance while remaining portable.

Development of BLAS and LAPACK origins in the 1970s. Motivation for the development of BLAS and LAPACK was to provide efficient and portable implementations for solving systems of linear equations and least-square problems. In such implementations certain linear algebra operations are frequently recurring. Good engineering practice suggests specifying a set of routines providing the needed functionality. Keeping the number of specified routines low enhances portability. At the same time implementations dedicated to a particular platform can exploit the underlying hardware architecture without sacrificing compatibility. The original BLAS [44] specified Fortran routines for scalar and vector operations (e.g. sum of vectors, dot-products, vector norms, . . .) and was successfully exploited by LINPACK [25] the predecessor of LAPACK.

In order to take advantage of more modern hardware architectures (e.g. vector computers and parallel computers) in the 1980s the original BLAS was extended. This resulted in BLAS Level 2 [24] specifying matrix-vector operations and BLAS Level 3 [23] for matrix-matrix operations. Combining and extending the functionality of EISPACK [70] (a library for eigenvalue and singular value problems) and LINPACK finally resulted in LAPACK.

LAPACK and BLAS routines are intended for merely dense and banded matrices. The type of matrix elements can be real or complex with single or double precision. In recent years, BLAS and LAPACK specifications were extended for sparse vectors/matrices as well as for element types with arbitrary or extended precision. However only few (commercial) implementations for these extensions are currently available.

Integration of BLAS and LAPACK into FLENS is primarily covered in Chapters 3 and Chapter 4. As exemplified in Chapter 5, extensions of BLAS and LAPACK (e. g. new matrix types, new element types) can be easily incorporated into FLENS.

2.1.2 Automatic Performance Tuning of High Performance Libraries

Modern hardware architectures exhibit a complexity that makes it hard to implement high performance libraries. Cache and pipeline effects are almost unpredictable without an extraordinary knowledge of the underlying architectures. Another relevant factor for the performance is the utilized compiler. Many hardware vendors also develop their own compilers for this reason which enables them to realize optimal implementations of high performance libraries.

Without these resources and possibilities performance tuning requires randomly adjusting code and measuring the results [85]. *Explorative optimization* is an approach that can be outlined as 'systematically adjusting code and measuring the results'. For an algorithm certain factors are identified that are relevant for performance. For loop structures such factors might be related to, for example, loop unrolling or tiling. The library gets tuned for a specific hardware platform by systematically changing these parameters and measuring the resulting performance. Using this optimization method, libraries can be developed (at least to a certain extent) independent of a concrete architecture, while a near-optimal performance can be achieved after tuning.

PHiPAC (Portable High Performance Ansi C) [7, 8, 9] and ATLAS (Automatically Tuned Linear Algebra Subprograms) [88, 89] are BLAS implementations based on the explorative optimization approach. The same technique is used by FFTW (Fastest Fourier Transform in the West) [30] for the implementation of the *Fast Fourier Transforms.* PHiPAC, ATLAS and FFTW are all capable of competing with vendor tuned implementations and in some cases are even superior. Moreover, all these libraries are open source, free and non-commercial.

ATLAS and PHiPAC can be utilized in FLENS as BLAS implementations. Depending on matrix and vector sizes for example, different BLAS implementations exhibit different performance results. The explorative optimization approach could be used in FLENS to choose the optimal implementations for a given problem size. Examples for integrating FFTW into FLENS are given in Chapter 6.

2.1.3 HPC Libraries for Parallel Computers with Distributed Memory

ScaLAPACK (Scaleable LAPACK) [59, 37, 13] includes a subset of LAPACK routines that were redesigned for parallel computers with distributed memory (e. g. a computer cluster, that is, a group of computers interconnected via network). Fundamental building blocks of the ScaLAPACK library are:

1. PBLAS (Parallel BLAS) a distributed memory version of the Level 1, 2 and 3 BLAS.
2. BLACS (Basic Linear Algebra Communication Subprograms) a library for handling interprocessor communication.

One of the design goals of ScaLAPACK was to have the ScaLAPACK routines resemble their LAPACK equivalents as much as possible.

Using FLENS to implement parallel numerical routines is addressed in Chapters 6 and 7. The concepts applied there are similar to those used in the realization of ScaLAPACK. In particular, it is possible to integrate ScaLAPACK into FLENS.

2.1.4 Direct Solvers for Sparse Linear Systems of Equations

Numerical methods for solving partial differential equations often lead to huge linear systems of equations, where the coefficient matrix is sparse. Solvers for these systems can be classified as either iterative or direct solvers. While Chapters 6 and 7 are mainly focused on iterative solvers, Chapter 8 illustrates the possibility of integrating direct solvers into FLENS. A general overview and comparison of different direct solver libraries is given in [33]. Some of these libraries are briefly described in the following:

1. SuperLU [46, 20, 19] is a general purpose library for the direct solution of large, sparse, non-symmetric systems of linear equations on high performance machines. The library is written in C and is callable from either C or Fortran.

2. UMFPACK [16, 15, 18, 17] is a set of routines for solving unsymmetric sparse linear systems using the so called *Unsymmetric MultiFrontal* method. The library is implemented in C.

3. PARDISO (PArallel DIrect SOlver) [63, 64] is a thread-safe, high-performance, robust, memory efficient and easy to use software for solving large sparse symmetric and unsymmetric linear systems of equations on shared memory multiprocessors. The library is based on C and Fortran source code.

As described in Chapter 8, the efficient storage of sparse matrices is crucial for the overall performance. Depending on the numerical application different formats are favorable. A good introduction on these formats and their implementation is given in [22].

2.2 High-level Languages for Numerical Computations

Software packages like Matlab [55, 49] and Octave [27] provide high-level languages for numerical computations. The Octave package is a non-commercial, open source project which is mostly compatible with the commercial Matlab package. The scope of these packages can be extended and covers, e. g. , numerical linear algebra, numerical solvers for ordinary and partial differential equations and numerical optimization.

Due to the dedicated programming language, numerical algorithms can be implemented based on a syntax which is close to the mathematical notation. The expressive syntax not only allows rapid prototyping but also makes it easier to ensure the correctness of the resulting implementation. Both packages use external libraries (e. g. BLAS and LAPACK implementations, libraries for numerical optimizations) to carry out numerical computations. Hence, they can be considered as high-level interfaces to these libraries[1]. On the one hand, Matlab and Octave benefit from the performance provided by these external libraries. On the other hand, in general not all features of these libraries can be exploited. For instance, only two matrix types are supported, namely dense and sparse matrices. Other matrix types (e. g. for matrices with band or block structure) that are supported by BLAS and LAPACK are not available in Matlab and Octave. The abandonment of a greater variety of matrix types is in favor for simplicity and ease of use.

Like Matlab and Octave, FLENS aims to provide an expressive and intuitive syntax. For this purpose, FLENS adapted various ideas from these packages. Among others, this is reflected in the usage of overloaded operators for basic linear algebra operations or the notation to refer to matrix and vector slices (cp. Chapter 4). However, the primary intension of FLENS is distinct. FLENS is intended to be a library for scientific and high performance computing. Instead of

[1] Actually the initial motivation to develop Matlab was to provide students a simple interface to LINPACK and EISPACK [55].

completely hiding internal technical details, this aim requires that users of FLENS have the possibility to completely comprehend, follow and control what is going on. Sacrificing efficiency or transparency for the sake of simplicity is not acceptable in the field of high performance computing. How FLENS realized an interface satisfying these requirements is illustrated in Chapters 3 and 4.

2.3 Scientific Computing in Object-Oriented Languages

Object-oriented languages exhibit features to ease the usage of techniques like encapsulation, inheritance, modularity and polymorphism. These features ease the realization of large complex libraries. However, most libraries in the field of high performance computing are still implemented in non-object-oriented languages. This is mainly due to the fact that runtime overhead resulting from abstraction is sometimes hard to avoid.

Beside object-oriented programming, the C++ programming languages [75] supports procedural and generic programming. In particular generic programming can be utilized for the avoidance of runtime overhead due to abstraction. One of the first C++ libraries that could compete in the field of high performance library was BLITZ++ ([79, 86, 83]). The BLITZ++ library applied techniques like *expression templates* [81], *meta programming* [82] and *traits* [56]. For small matrices and vectors a speedup factor of up to ten could be achieved. While BLITZ++ introduced revolutionary techniques, it only supported a small subset of the linear algebra operations specified by BLAS. It provides vector types and arrays, but no further specialized matrix classes. In particularly, there is no support for sparse data types. A further issue is the complexity of the expression template technique (cp. Section 4.3.2). Todd Verhuizen, the creator of BLITZ++ and inventor of the expression template technique, states that this is a "... very elaborate, complex solution which is difficult to extend and maintain. It is probably not worth your effort to take this approach." [84].

Currently, one of the most widespread C++ libraries for scientific computing is uBLAS [87]. While the library is implemented using many of the concepts introduced by BLITZ++ (in particular expression templates) it offers a full support of BLAS. In addition uBLAS provides different sparse matrix implementations and corresponding linear algebra operations. However, like for the BLITZ++ library, extending uBLAS (e. g. adding new matrix types) is a non-trivial task.

A more complete list of object-oriented libraries and software can be found on the *Object-Oriented Numerics* web site [80]. This list also includes libraries implemented in programming languages different from C++.

FLENS adapted and extended some basic ideas from the expression templates technique for the realization of its high-level BLAS interface described in Chapter 4. The modifications are in particular focused on achieving a higher flexibility and simplifying extensibility.

2.4 Software Quality

Software Engineering is the computer science discipline concerned with developing large applications. Among others, software engineering covers techniques and methods for defining software requirements, performing software design, software construction and software testing. All these techniques and methods are intended for the production of high quality software. The IEEE Standard 729-1983 [41] states a rather general definition of software quality as follows:

1. The totality of features and characteristics of a software product that bear on its ability to satisfy given needs; e. g. conform to specifications.

2. The degree to which software possesses a desired combination of attributes.

3. The degree to which a customer or user perceives that software meets his or her compositions expectations.

4. The composite characteristics of software that determine the degree to which the software in use will meet the expectations of the customer.

In order to measure software quality Betrand Meyer [53] specifies the following factors:

Correctness is the ability of software products to perform their exact tasks, as defined by their specifications.

Robustness is the ability of software systems to react appropriately to abnormal conditions.

Extendibility is the ease of adapting software products to changes of specifications.

Reusability is the ability of software elements to serve for the construction of many different applications.

Compatibility is the ease of combining software elements with others.

Efficiency is the ability of a software system to place as few demands as possible on hardware resources, such as processor time, space occupied in internal and external memories, bandwidths used in communication devices.

Portability is the ease of transferring software products to various hardware and software environments.

Ease of use is the ease with which people of various backgrounds and qualifications can learn to use software products and apply them to solve problems. It also covers the ease of installation, operation and monitoring.

Functionality is the extent of possibilities provided by a system.

Timeliness is the ability of a software system to be released when or before its users want it.

Obviously all these definitions are stated rather generally and need to be substantiated for each concrete type of software. Quality of scientific software was in particular discussed, investigated and exemplified in [50].

Libraries can be used to extend the scope of programming languages. Thus, the quality of software packages can benefit from the quality of these libraries. Based on *Ulm's Oberon System*[2], techniques and concepts for the design of high quality libraries were demonstrated in [10].

The FLENS library extends the scope of the C++ programming languages for features that are relevant for scientific computing. The quality of FLENS gets discussed and analyzed in this work based on the above quality factors.

2.5 FLENS: Previous Releases

The original FLENS library [73] was developed by Alexander Stippler in 2003. The present work describes its successor that resulted from a complete redesign and re-implementation. While the scope and intension of these two libraries are different, the original FLENS library had some essential influence on its successor.

The original FLENS library was mainly intended to build a base for the implementation of various applications concerning adaptive wavelet methods for solving elliptic partial differential equations. In this field the implementation of numerical methods requires rather flexible matrix and vector types. For instance, indices are typically multidimensional, and the sparsity patterns of matrices and vectors are a priori unknown. This requires quite flexible indexing mechanisms

[2]`http://www.mathematik.uni-ulm.de/oberon/`

and index types, as well as sophisticated data structures in order to combine fast element access and compact storage.

Soon after its release the original FLENS library was not only used for the development of adaptive wavelet methods, but also applied in other fields of scientific computing (e. g. numerical optimization [69] or numerical finance [91], [61]). A further field of application was teaching. FLENS was used as a tool for numerical programming in both undergraduate and graduate courses at Ulm University. However, as the field of applications steadily became wider, deficiencies of the original FLENS design became apparent. Design issues were related to efficiency, flexibility, extensibility and reusability. Such deficiencies are typical for a first design of a complex library. Further, for the development of the original FLENS no computer cluster was available. Therefore parallelization could not be thoroughly considered in the initial design. In order to resolve these issues a complete redesign and re-implementation became inevitable. For the relaunch of the FLENS project the experiences made with the original FLENS library were invaluable and influenced most of its design decisions.

2.6 Related Scientific Libraries

Compared to its original releases the new FLENS library aims to be a much more general and flexible building block for scientific and high performance computing. In the field of adaptive wavelet methods it serves as a building block for LAWA (Library for Adaptive Wavelet Applications) [72] which is now actively developed by Alexander Stippler. LAWA and FLENS were used in [77] to demonstrate and exemplify adaptive wavelet methods for solving elliptic partial differential equations.

For the numerical treatment of partial differential equations various libraries exist that are highly sophisticated. UG [5], DUNE [6] and ALBERTA [66] for instance, provide numerical methods that are state of the art in this domain. Also serving as a building block for such libraries in the future was an ambitious aim in the development of FLENS.

In order to serve as an effective building block, the scope of FLENS was designed orthogonal to that covered by the libraries mentioned above. FLENS solely intends to extend the C++ programming languages for features that are frequently needed in the development of scientific software. Libraries specialized in a particular field can take advantage of these extensions within their development process.

3 Low-Level Interface for BLAS and LAPACK

For any library intended as a serious platform for scientific computing, interfacing with other external libraries is a highly relevant and essential feature. Such external libraries are often specialized in treating only certain domain specific problems and can be considered as experts in this field. This chapter illustrates the low-level interface of FLENS for BLAS and LAPACK, where the term 'low-level' reflects the degree of abstraction realized in the interface. Most of the issues addressed in this chapter occur in a similar form when interfacing with other external libraries. This includes the handling of specific data structures or the situation where external libraries are implemented in programing languages that are different from C++.

BLAS and LAPACK implementations support different formats for storing matrices and vectors. These formats are commonly denoted as *storage schemes*. In Section 3.1 some of the most important storage schemes are introduced. As shown there, many technical details as well as programming language specific features are relevant for the proper handling and usage of storage schemes. Outlined in Section 3.2, this motivates the implementation of C++ classes encapsulating such schemes. Sections 3.3 and 3.4 give a general overview of the functionality and the usage of BLAS and LAPACK. Further, these sections address the issue of calling Fortran routines from within C++ programs. The chapter is concluded by a brief discussion on software quality. This discussion in particular addresses the need for 'low-level' interfaces.

3.1 Storage Schemes for Matrices and Vectors

BLAS provides linear algebra routines for different matrix types: triangular, symmetric, hermitian and general (i. e., non-square or unsymmetric). Storage of elements can be organized following different schemes – in the following denoted as *storage schemes*. What particular storage scheme is favorable depends on the matrix structure as well as the numerical methods operating on the matrix. While in some cases compact storage of a matrix is essential, in others it is crucial to have fast access to single elements, whole rows or columns or to operate efficiently on matrix slices.

It is important to point out the difference between storage schemes and matrix types. Each storage scheme merely defines a certain format specifying how to store matrix elements in memory. However, the storage scheme itself does not specify a particular matrix type. For illustration consider the general matrix

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & 0 \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ 0 & a_{3,2} & a_{3,3} & a_{3,4} \\ 0 & 0 & a_{4,3} & a_{4,4} \end{pmatrix}.$$

The very same storage scheme used to store elements of A can be used to represent symmetric, hermitian or triangular matrices. Referencing only elements from the storage scheme belonging to either the upper or lower triangular part allows to represent symmetric matrices (elements actually stored in memory are indicated red):

$$S_U = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & 0 \\ a_{1,2} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{1,3} & a_{2,3} & a_{3,3} & a_{3,4} \\ 0 & a_{2,4} & a_{3,4} & a_{4,4} \end{pmatrix} \quad \text{and} \quad S_L = \begin{pmatrix} a_{1,1} & a_{2,1} & 0 & 0 \\ a_{2,1} & a_{2,2} & a_{3,2} & 0 \\ 0 & a_{3,2} & a_{3,3} & a_{4,3} \\ 0 & 0 & a_{4,3} & a_{4,4} \end{pmatrix}.$$

Analogously, the scheme can be interpreted to represent upper or lower triangular matrices. Triangular matrices can further be assumed to be unit triangular such that diagonal elements are actually not referenced. In connection with the LU decomposition (see Section 3.4.1) it is in particular useful to interpret the storage scheme of a general matrix also as

$$U = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & 0 \\ 0 & a_{2,2} & a_{2,3} & a_{2,4} \\ 0 & 0 & a_{3,3} & a_{3,4} \\ 0 & 0 & 0 & a_{4,4} \end{pmatrix} \quad \text{or} \quad L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ a_{2,1} & 1 & 0 & 0 \\ 0 & a_{3,2} & 1 & 0 \\ 0 & 0 & a_{4,3} & 1 \end{pmatrix}.$$

Recapitulating and simplifying one can note that — in the sense of BLAS and LAPACK — matrix types are compositions of storage schemes together with some additional informations specifying how to interpret these schemes.

With respect to software quality such a loose coupling between storage scheme and matrix type is not optimal. In Chapter 4 matrix and vector classes will be introduced to firmly link these ideas together.

3.1.1 Memory Layout of Arrays in C and Fortran

The memory layouts of multidimensional arrays of C++ and Fortran differ. This difference is important for any C/C++ library interfacing with Fortran libraries. This is because data structures are passed via pointers from the C/C++ library to Fortran routines. Hence, data structures setup within the C/C++ library have to comply to the memory layout expected by Fortran. In the following these differences in the memory layout are illustrated in more detail.

In C (and therefore also in C++) the memory layout of multidimensional arrays is deliberately[1] reflected by the syntax used for the declaration of arrays and the notation for element access (see Table 3.1). The underlying concept is commonly known as the K&R method[2] of reducing arrays to pointers.

	declaration	element access of $a_{i,j}$
C	`float a[m][n]`	`a[i-1][j-1]`
Fortran	`REAL a(m,n)`	`a(i,j)`

Table 3.1: Syntax for arrays in C and Fortran.

An n-dimensional array is a one-dimensional array containing elements that are $(n-1)$-dimensional arrays. Further an array is treated as a pointer to the first element and the notation `a[i]` is equivalent to `*(a+i)` dereferencing an address computed by address arithmetic. As this implies that `*a = a[0]` refers to the first element, the indices of C arrays must be zero-based.

For matrices the most important consequence is their row-wise storage in two-dimensional C arrays. This is best illustrated by considering storage of a matrix $A \in \mathbb{R}^{m \times n}$ in a two-dimensional C array `a[m][n]`. The element $a_{i,j}$ with $1 \le i \le m$, $1 \le j \le n$ gets stored in `a[i-1][j-1]`. As the first index refers to a row it is evident that `a[i-1]` points to the first element of the i-th row. Figure 3.1 depicts the memory layout of a dynamically allocated two-dimensional array `a` storing $A \in \mathbb{R}^{3\times 4}$. In this case the array is, e. g., of type `float **a` and `a` is a pointer to an array containing pointers to the rows of A.

[1]This might have been one reason for C to become a preferred language for system programming.
[2]referring to Brian W. Kernighan und Dennis M. Ritchie

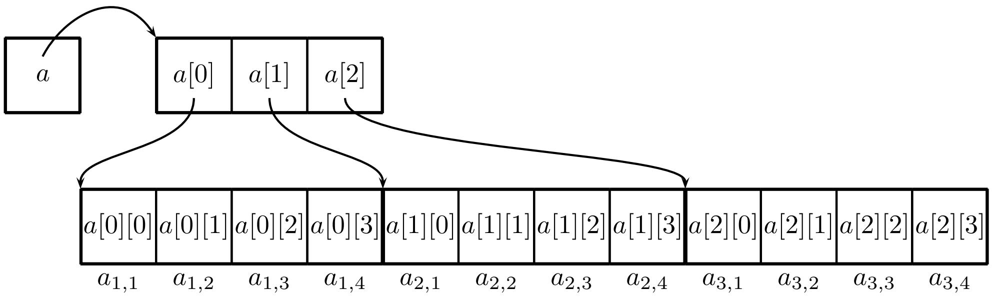

a
$a[0]$
$a[1]$
$a[2]$
$a[0][0]$ $a[0][1]$ $a[0][2]$ $a[0][3]$ $a[1][0]$ $a[1][1]$ $a[1][2]$ $a[1][3]$ $a[2][0]$ $a[2][1]$ $a[2][2]$ $a[2][3]$
$a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,4}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$ $a_{2,4}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$ $a_{3,4}$

Figure 3.1: Memory layout of a dynamically allocated, two-dimensional array in C.

Fortran's syntax for arrays is more aligned to the mathematical notation. By default, Fortran arrays are indexed with base one. Opposed to C, in Fortran[3] matrices are internally stored column-wise. The same memory layout can be achieved with C by swapping the meaning of the indices, i. e. the first index referring to columns, the second to rows. This simply means that storage of a matrix A in a two-dimensional Fortran array equals storage of A^T in a two-dimensional C array. Figure 3.2 shows how to dynamically allocate a two-dimensional array in C in order to achieve a compatible memory layout for Fortran.

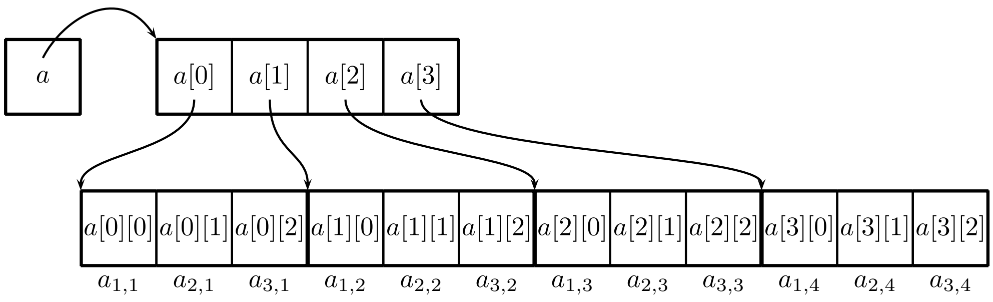

a
$a[0]$
$a[1]$
$a[2]$
$a[3]$
$a[0][0]$ $a[0][1]$ $a[0][2]$ $a[1][0]$ $a[1][1]$ $a[1][2]$ $a[2][0]$ $a[2][1]$ $a[2][2]$ $a[3][0]$ $a[3][1]$ $a[3][2]$
$a_{1,1}$ $a_{2,1}$ $a_{3,1}$ $a_{1,2}$ $a_{2,2}$ $a_{3,2}$ $a_{1,3}$ $a_{2,3}$ $a_{3,3}$ $a_{1,4}$ $a_{2,4}$ $a_{3,4}$

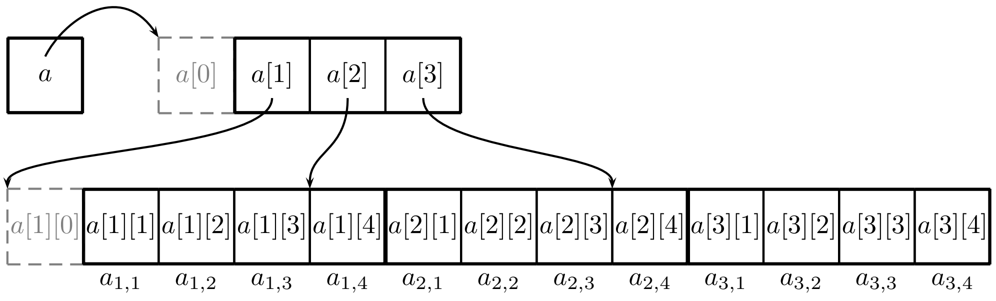
Figure 3.2: Achieving a Fortran compatible memory layout with C.

3.1.2 Arrays with Arbitrary Index Base

For the implementation of numerical algorithms the fact that C arrays are zero based can become inconvenient and therefore lead to an error prone code. Using dynamically allocated arrays allows the realization of arbitrary index bases. Figure 3.3 shows how to store $A \in \mathbb{R}^{3\times 4}$ row-wise in a two-dimensional array `a` such that row and column indices — to be precise *valid* indices — start at one. Compared to Figure 3.1 pointers are shifted by '-1'. Obviously this can be generalized to achieve arbitrary index ranges, however in a realization without special precautions this may easily lead to error prone code. Moreover, typical errors resulting from 'out of range' indices are extremely hard to debug.

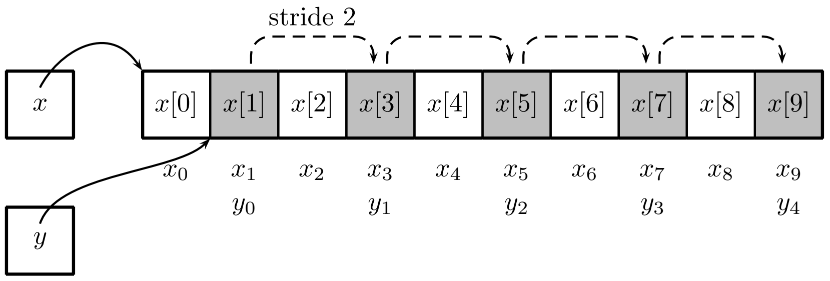
The gathered knowledge about arrays now allows one to introduce the storage schemes used in BLAS and LAPACK and to illustrate how to set up corresponding structures in C. This in turn will be the base for a solid design and implementation of C++ classes realizing these schemes.

[3]In Fortran — which is older than C — column-wise storage is merely a standardized convention. For C the storage order is a inherent consequence of the K&R method.



a
a[0]
a[1]
a[2]
a[3]
a[1][0]
a[1][1]
a[1][2]
a[1][3]
a[1][4]
a[2][1]
a[2][2]
a[2][3]
a[2][4]
a[3][1]
a[3][2]
a[3][3]
a[3][4]
$a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,4}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$ $a_{2,4}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$ $a_{3,4}$

Figure 3.3: Two-dimensional C array with index base one.



stride 2
x
y
x[0] x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8] x[9]
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9
y_0 y_1 y_2 y_3 y_4

C/C++ Code:
`double x[10];`
`double *y = x+1;`
`int strideY = 2;`
`int lengthY = 5;`

Figure 3.4: Vector and Vector View.

3.1.3 Vector Storage Scheme

Vector elements are stored linearly in memory separated from each other by a constant *stride*. Using a one-dimensional array results in a contiguous storage where the stride equals one. Strides different from one naturally arise when dealing with vector slices. Hereby a vector slice is determined by three pieces of information: a pointer to the first element, length of the slice and the stride between its elements. Figure 3.4 illustrates an example where `x` is a contiguously stored vector of length 10 and `y` refers to a slice of `x` referencing every second element. The i-th element of `y` gets dereferenced by `y[i*stride]` for $0 \leq i < 5 =$ `lengthY`.

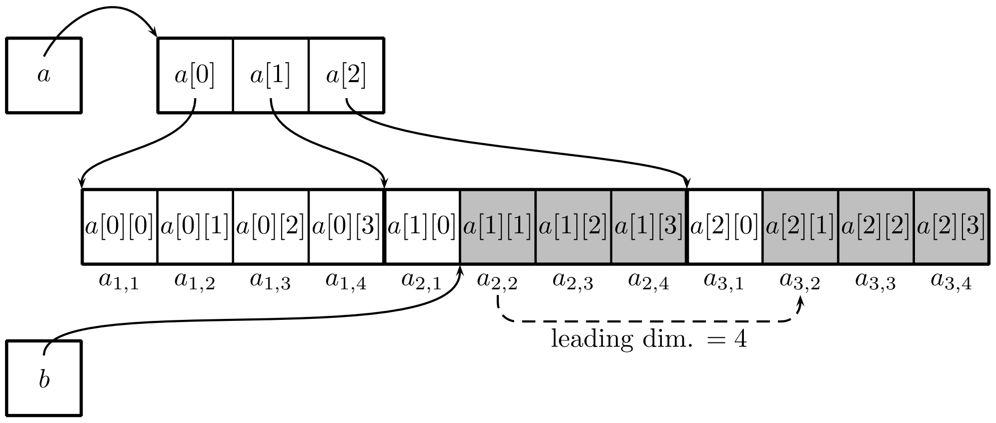
FLENS adopted the term *view* from database systems to refer to slices. This terminology reflects that a slice is not copied and stored in extra allocated memory. Instead, slices directly refer to parts of another vector and changing elements of the slice changes elements in the 'original' vector. The same terminology is used for matrix slices.

3.1.4 Full Storage Scheme

In the full storage scheme, all $m \cdot n$ elements of a $m \times n$ matrix are stored in a two-dimensional array. The storage is said to be *row major* or *column major* if elements are stored row or column wise respectively. Using shifted pointers, an implementation can provide arbitrary index bases for rows and columns.

Full storage allows one to operate on matrix views that are sub-matrix blocks. Hereby the term 'block' refers to sub-matrices of form $B = (a_{i,j})$ with $i_0 \leq i \leq i_1$ and $j_0 \leq j \leq j_1$. For this purpose four pieces of information are needed: a pointer `b` to the first element (a_{i_0,j_0}), the number of rows and columns of the view (that is $i_1 - i_0 + 1$ and $j_1 - j_0 + 1$ respectively) and further the so called *leading dimension*. In a row major storage, the leading dimension denotes the stride between column elements, and for column major storage the stride between row elements.

Obviously the leading dimension is given by the dimensions of the original matrix. In a row major storage, the leading dimension equals the number of columns, and analogously in a column major storage, the number of rows. Figure 3.5 illustrates this concept for a 2×3 matrix view B referencing a block within a 3×4 matrix A.



a
$a[0]$
$a[1]$
$a[2]$
$a[0][0]$ $a[0][1]$ $a[0][2]$ $a[0][3]$ $a[1][0]$ $a[1][1]$ $a[1][2]$ $a[1][3]$ $a[2][0]$ $a[2][1]$ $a[2][2]$ $a[2][3]$
$a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,4}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$ $a_{2,4}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$ $a_{3,4}$
leading dim. = 4
b

Figure 3.5: Matrix view of a row major full storage scheme.

Accessing elements of B requires some pointer arithmetic as summarized in Table 3.2. Due to the additional integer operations element access for views is considerably slower as compared to 'non-views'. If fast element access is relevant, setting up an additional array with pointers to

storage order	$B_{i,j}$	leading dimension (`ld`)
column major	`b[j*ld+i]`	row element stride
row major	`b[i*ld+j]`	column element stride

Table 3.2: Element access for full storage schemes

rows of the view (or columns in column major schemes) may pay off.

As can be seen in Figure 3.5, elements of matrix views are in general not contiguously in memory. However some numerical libraries (e. g. FFTW [30]) explicitly rely on this.

3.1.5 Band Storage Scheme

The band storage format intends to store a $m \times n$ matrix A with k_l sub-diagonals and k_u super-diagonals[4] compactly in a two-dimensional array `ab`:

1. Columns of A are stored in columns of `ab` and
2. diagonals of A in rows of `ab`. More precisely, the first row of `ab` stores the highest super-diagonal of A.

BLAS and LAPACK functions require that `ab` is stored in column major order. In general, array `ab` will contain a few additional dummy elements (as indicated by '$*$'):

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & 0 & 0 \\ a_{2,1} & a_{2,2} & a_{2,3} & 0 \\ 0 & a_{3,2} & a_{3,3} & a_{3,4} \\ 0 & 0 & a_{4,3} & a_{4,4} \\ 0 & 0 & 0 & a_{5,4} \end{pmatrix}, \quad \texttt{ab}_{\text{(Col. Major)}}:$$

$*$	$a_{1,2}$	$a_{2,3}$	$a_{3,4}$
$a_{1,1}$	$a_{2,2}$	$a_{3,3}$	$a_{4,4}$
$a_{2,1}$	$a_{3,2}$	$a_{4,3}$	$a_{5,4}$

The above two rules define the column major band storage scheme for A. Recalling that row major storage of A has to be equivalent to column major storage of A^T it follows that:

[4]only $k_l, k_u \geq 0$ is supported in BLAS and LAPACK.

	$*$	$a_{2,1}$	$a_{3,2}$	$a_{4,3}$	$a_{5,4}$
`ab`$_{\text{(Row Major)}}$:	$a_{1,1}$	$a_{2,2}$	$a_{3,3}$	$a_{4,4}$	$*$
	$a_{1,2}$	$a_{2,3}$	$a_{3,4}$	$*$	$*$

For an implementation of the band storage scheme it is favorable to use for `ab` an implementation of the full storage scheme with arbitrary index ranges:

	Dimensions of `ab`	row indices of `ab`	column indices of `ab`	a_{ij}
col. major	$(k_u + k_l + 1) \times n$	$-k_u, \dots, k_l$	$=$ col. indices of A	`ab(i-j,j)`
row major	$(k_u + k_l + 1) \times m$	$-k_l, \dots, k_u$	$=$ row indices of A	`ab(j-i,i)`

The band storage format allows one to support matrix views referencing a subset of adjacent bands in the original matrix. A banded sub-matrix B of A with $\tilde{k}_l$ sub-diagonals and $\tilde{k}_u$ super-diagonals where $k_l \le \tilde{k}_l \le \tilde{k}_u \le k_u$ can be realized through appropriate views of `ab`. Hereby views of `ab` reference whole rows. In column major ordering these are rows with indices $-\tilde{k}_u, \dots, \tilde{k}_l$, whereas in row major ordering $-\tilde{k}_l, \dots, \tilde{k}_u$.

3.1.6 Packed Storage Scheme

Triangular, symmetric and hermitian matrices can be stored most compactly using the packed storage scheme. Either the upper or lower triangular part of the matrix gets stored. Elements are stored contiguously — either row or column wise — in a one-dimensional vector `ap`:

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ 0 & a_{2,2} & a_{2,3} \\ 0 & 0 & a_{3,3} \end{pmatrix},$$

`ap`$_{\text{(Row Major)}}$:	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{2,2}$	$a_{2,3}$	$a_{3,3}$
`ap`$_{\text{(Col. Major)}}$:	$a_{1,1}$	$a_{1,2}$	$a_{2,2}$	$a_{1,3}$	$a_{2,3}$	$a_{3,3}$

Formulas for element access depend on whether the upper or lower triangular part of A is stored in `ap` as well as the storage order. If row and column indices of A start at zero then using for `ab` a vector with index base zero leads to the following formulas:

storage order	triangular part	$a_{i,j}$ stored in
row major	upper	`ap`$(j + i(2n - i - 1)/2)$
row major	lower	`ap`$(j + i(i + 1)/2)$
col. major	upper	`ap`$(i + j(j + 1)/2)$
col. major	lower	`ap`$(i + j(2n - j - 1)/2)$

For row and column indices starting at i_0 and j_0 respectively replace i with $i - i_0$ and j with $j - j_0$. Due to the required integer arithmetic arbitrary element is relatively slow. However consecutive element access as required for a triangular solver (i. e. solving a system of linear equations where the coefficient matrix is triangular) is fast.

3.2 Realization of Storage Schemes in C++

Implementing classes for storage schemes makes it a lot easier to use BLAS or LAPACK functions. Testing these classes thoroughly assures correct memory allocation and element access. In debug mode, methods for element access can check index ranges. In conjunction with support for arbitrary index ranges, correct allocation as well as index checks are essential to ensure correctness. In general, supporting vector or matrix views also requires some sort of memory management. To be more precise, if different objects share some dynamically allocated block of memory, it has to be guaranteed that this memory block gets deallocated by the last referencing object.

3.2.1 Vector Storage: `class Array`

Beside the vector length and index base the `Array` class encapsulates a pointer to the allocated memory[5]:

```
template <typename T>
class Array
{
    public:
        Array();

        Array(int length, int firstIndex=1);

        // ...

    private:
        int _length, _firstIndex;
        T *_data;
};
```

Elements are stored contiguously in memory. The pointer is shifted such that `_data[_firstIndex]` references the first element:

```
_data = static_cast<T *>(calloc(length, sizeof(T))) - _firstIndex;
assert(_data+_firstIndex);
```

In a more general implementation additional template parameters for the `Array` class can serve to treat allocation of memory more flexibly: E. g. using `malloc` or `memalign` instead of `calloc` or other implementations for dynamic memory allocation.

Only in debug mode, index ranges are checked for element access such that in non-debug mode there is no overhead compared to ordinary C arrays:

```
template <typename T>
T &
Array<T>::operator()(int index)
{
    assert(index>=_firstIndex);
    assert(index<_firstIndex+_length);

    return _data[index];
}
```

Methods to retrieve vector length, stride between elements as well as a pointer to the first vector element can be used for calling BLAS and LAPACK functions (see Section 3.3):

```
template <typename T>
class Array
{
    public:

        int length() const;

        int stride() const;

        const T *data() const;
        {
            return &_data[_firstIndex];
        }

        T *data();

        // ...
};
```

[5]Names of private variables have a preceding ‘_’ by the FLENS coding style convention.

3.2.2 Vector Storage and Views: class ArrayView and ConstArrayView

The Array class further provides methods for the creation of views. These views are either of type ArrayView or ConstArrayView depending on whether a view gets created from within a non-const or const context:

```
template <typename T>
class Array
{
    public:

        ConstArrayView<T>
        view(int from, int to, int stride=1, int firstViewIndex=1) const;

        ArrayView<T>
        view(int from, int to, int stride=1, int firstViewIndex=1);

        // ...
};
```

The view objects created will reference those elements of the original array indexed there by from, from + stride, from + 2*stride,..., to. The stride can neither be zero nor negative as this is not supported by most BLAS implementations. The index base of the array view is firstViewIndex and can differ from the index base used in the original array. ArrayView provides the same interface, i. e. the same methods, as class Array. While ConstArrayView — to ensure *const correctness* — defines only the const methods of the Array interface, such that underlying data can not be modified.

As was shown in Subsection 3.1.3 ArrayView and ConstArrayView objects have to store the length of the array view, its index base, the stride between elements and a (shifted) pointer to access the first element of the view. The address of the memory block allocated by the original Array object is stored in pointer _storage:

```
template <typename T>
class ArrayView
{
    // ...

    private:
        void *_storage;
        T *_data;
        int _length, _stride, _firstIndex;
};
```

The reference counter for _storage gets incremented during the construction of view objects, as here for ArrayView objects:

```
template <typename T>
ArrayView<T>::ArrayView(void *storage, T *data,
                        int length, int stride, int firstIndex)
  : _storage(storage), _data(data),
    // ...
{
    RefCounter::attach(_storage);
}
```

Class RefCounter is internal part of FLENS an provides a simple reference counting implementation. Method RefCounter::attach(addr) increments and RefCounter::detach(addr) decrements a reference counter for address addr. detach returns true if no further object instance exists using the underlying address. Whether the destructor of the array view releases memory depends on the reference counter:

```
template <typename T>
ArrayView<T>::~ArrayView()
{
    if (RefCounter::detach(_storage)) {
        free(_storage);
    }
}
```

Note that the `ConstArrayView` must not deallocate memory as this would violate const correctness. Instead an assertion gets triggered such that in debug mode it will be tested to ensure that the last referencing object is not a const view object. The destructor of the `Array` class is implemented analogously to `ArrayView` but can use the `data()` method to get a pointer to the allocated memory block:

```
template <typename T>
Array<T>::~Array()
{
    if (RefCounter::detach(data())) {
        free(data());
    }
}
```

The implementation of reference counting used here is kept very simple. This is sufficient as no recursive dependencies can occur. Much more sophisticated techniques like *garbage collection* [42] are integral parts of some other programming languages and for this reason make memory management there much safer. Further, using the user-defined reference counting strategy only across three (not too complex) classes — combined with some test cases — gives a fair chance to ensure a correct implementation.

Note that accessing elements of an array view requires pointer arithmetic and can imply a considerable run-time overhead:

```
template <typename T>
T &
ArrayView<T>::operator()(int index)
{
    assert(index>=_firstIndex);
    assert(index<_firstIndex+_length);

    return _data[_firstIndex + _stride*(index-_firstIndex)];
}
```

3.2.3 Interface of Vector Storage Schemes

Classes `Array`, `ArrayView` and `ConstArrayView` do not have a common base class but instead are only related by meeting certain interface conventions. Such a convention is most relevant for generic programming. The interface of the vector storage schemes loosely can be grouped into categories[6]:

1. Initialization, element access and changing vector size/index base:

`int`
`firstIndex() const;`
	First valid index.

`int`
`lastIndex() const;`
	Last valid index.

`const T&`
`operator()(int index) const;`
	Element access.
	Debug mode: Checks for valid indices.

`T&`
`operator()(int index);`	

[6]`ConstArrayView` only provides the const methods as mentioned above.

`void`
`resize(int length,`
`       int firstIndex=1);`
	Changing array size or index base. Previously stored data will not be copied!

2. Creation of views: View objects can be used like regular arrays. However, no additional memory gets allocated. Instead view objects merely reference elements of a previously allocated array. The index bases of the original array and its views can differ.

`ConstArrayView<T>`
`view(int from, int to, int stride=1,`
`     int firstViewIndex=1) const;`
	Creates a view referencing elements of the original array indexed by `from, from+stride, from+2*stride,...,to`.
	The index base of the view is `firstViewIndex`.

`ArrayView<T>`
`view(int from, int to, int stride=1,`
`     int firstViewIndex=1);`	

3. Access to internal data structures: in particular these methods are suited for calling BLAS and LAPACK functions (see Section 3.3 and 3.4).

`int`
`length() const;`
	Length of array.

`int`
`stride() const;`
	Stride in memory between array elements.

`const T *`
`data() const;`
	Pointer to the first element in the array.

`T *`
`data();`	

4. Constructors and assignment operators: Instead of listing the signatures these are best documented by depicting functionality and providing examples of their usage. Class `Array` provides constructors for the creation of array objects of a particular size (including size zero) and index base (by default 1):

```
Array<double> x,          // length 0 (no memory allocated),
              y(5),       // length 5, index base 1
              z(5,0);     // length 5, index base 0
```

Objects of type `ArrayView` can only be created from `Array` objects by calling one of its `view(..)` methods[7]. The creation of view objects that are not 'linked' with any `Array` object is not allowed:

```
ArrayView<double> a;                    // error
ArrayView<double> b =  y.view(1,3);     // ok, refers to y(1), y(2), y(3)
```

Arrays can be copied using the assignment operator. In an assignment the left hand side gets automatically resized if necessary and possible. Hereby 'necessary' means that the left and right hand side arrays are of different lengths. Whereas 'possible' refers to the fact that array views can not be resized. Note that if the left and right hand sides are of equal length but use different index bases there will be no resizing and in particular the index bases are not changed. However, if resizing is performed the left hand side will also adapt the index base of the right hand side:

[7]The `view` methods internally use a constructor of the `ArrayView` class not intended for public usage

```
x = y;           // resize x: new length is 5, index base 1.
                 //   -> x(1) = y(1), x(2) = y(2), ...
z = y;           // no resize
                 //   -> z(0) = y(1), z(1) = y(2), ...
b = y;           // error: can not resize b to length 5

x = b;           // ok, resize x: new length 3, index base 1
                 //   -> x(1) = b(1) [ = y(1) ],
                 //      x(2) = b(2) [ = y(2) ],
                 //      x(3) = b(3) [ = y(3) ]
```

Note that during an assignment there is no check for overlapping views. This is mainly due to performance reasons. Without caution of the user this can lead to unexpected results (see discussion in Section 3.2.7):

```
ArrayView<double> v1 = y.view(1,3);   // refers to y(1), y(2), y(3)
ArrayView<double> v2 = y.view(3,5);   // refers to y(3), y(4), y(5)
v2 = v1;                              //    y(3) = y(1),
                                      //    y(4) = y(2),
                                      // -> y(5) = y(3) [ = y(1) ]
```

5. Public typedefs and traits: For generic programming a well-defined set of typedefs and traits is essential. Consider a template function to find and return the absolute largest value of an array. The fact that all array classes provide a typedef `ElementType` for the element type allows the generic implementation:

```
template <typename ARRAY>
typename ARRAY::ElementType
max(ARRAY &x)
{
    // find and return max. value of x
}
```

The array `x` can be of type `Array`, `ArrayView` or `ConstArrayView`.

Typedefs `View` and `ConstView` define the type of views returned by the `view` methods. Typedef `NoView` defines the type of a regular array using its own memory. This is required for using views in generic programming:

```
template <typename ARRAY>
typename ARRAY::ElementType
dummy(ARRAY &x)
{
    typename ARRAY::View      v1 = x.view(1,3);
    typename ARRAY::ConstView v2 = x.view(1,3);
    typename ARRAY::NoView    y  = x.view(1,3);
}
```

While `v1` and `v2` reference slices of `x` array `y` contains its own copy thereof. Note that in this example `x` can either be of type `Array` or `ArrayView`. If `x` was of type `ConstArrayView` then creation of view `v1` would violate const-correctness. Such an attempt would result (as expected and intended) in a compile time error as an `ArrayView` can be converted into a `ConstArrayView` but not vice versa.

If typedefs are not suitable then trait classes can be defined to retrieve type information about template parameters at compile time. While this is not needed for the array classes implementations of other schemes as described in the next sections make use of traits.

3.2.4 Full Storage: C++ Interface and Implementation Notes

Template class `FullStorage` implements the full storage scheme described in Section 3.1.4. Beside a template parameter specifying the element type a second parameter defines the ordering of elements, which can be either row or column major. The later template parameter has to be the value of enumeration type

```
enum StorageOrder {
    RowMajor,
    ColMajor
};
```

declared inside the FLENS library. Objects of type `FullStorage` store the number of rows and columns, the index bases for rows and columns as well as a pointer to the stored elements:

```
template <typename T, StorageOrder Order>
class FullStorage
{
    public:
        FullStorage();

        FullStorage(int numRows, int numCols,
                    int firstRow=1,
                    int firstCol=1);
        // ...

    private:
        int _numRows, _numCols, _firstRow, _firstCol;
        T **_data;
};
```

Memory gets allocated dynamically during construction of the object. Elements are stored in a two-dimensional array as motivated and described in Section 3.1.4. The following code segment exemplifies allocation for column major ordering using shifted pointers to achieve general index bases:

```
if (Order==ColMajor) {
    _data = static_cast<T **>(calloc(_numCols, sizeof(T *))) - _firstCol;
    assert(_data+_firstCol);

    _data[_firstCol] = static_cast<T *>(calloc(_numCols*_numRows,sizeof(T)))
                     - _firstRow;
    assert(_data[_firstCol]+_firstRow);

    for (int i=1; i<_numCols; ++i) {
        _data[_firstCol+i] = _data[_firstCol] + i*_numRows;
    }
}
```

Views are supported through template classes `FullStorageView` and `ConstFullStorageView`. As was the case for arrays these view classes also need to store a pointer to the original data, which is required for reference counting. In addition the leading dimension has to be stored as was shown in Section 3.1.4:

```
template <typename T, StorageOrder Order>
class FullStorageView
{
    // ...
    private:
        void *_storage;
        T **_data;
        int _numRows, _numCols;
        int _leadingDimension;
        int _firstRow, _firstCol;
};
```

The interface for the packed storage scheme is documented in Appendix A.1.

3.2.5 Band Storage: C++ Interface and Implementation Notes

Template class `BandStorage` implements the band storage scheme. As was shown in Section 3.1.5 an implementation of this scheme can use a full storage scheme to store bands of the matrix. Hereby one can take advantage of the flexible implementations provided for the full storage scheme.

Template parameters for `BandStorage` specify the element type and the storage ordering. The index base for rows and columns has to be the same. Matrix dimensions and index base are stored explicitly in `BandStorage` objects. The number of sub- and super-diagonals can be retrieved implicitly from the `FullStorage` object storing the matrix elements:

```
template <typename T, StorageOrder Order>
class BandStorage
{
    public:
        BandStorage();

        BandStorage(int numRows, int numCols,
                    int numSubDiags, int numSuperDiags,
                            int indexBase=1);

        // ...

    private:
        int _numRows, _numCols, _indexBase;
        FullStorage<T, ColMajor> _data;
};
```

In numerical applications referencing single diagonals or multiple adjacent diagonals can be of interest. Diagonals can be referenced through array views as introduced in Section 3.2.2. For referencing adjacent bands the view classes `BandStorageView` and `ConstBandStorageView` are provided. In contrast to class `BandStorage` these classes use classes `FullStorageView` and `ConstFullStorageView` respectively. This means that for band storage schemes all tasks related to views — in particular tedious and possibly error prone issues like reference counting — can be delegated to a few, well-tested classes.

The interface for the band storage scheme is documented in Appendix A.2.

3.2.6 Packed Storage: C++ Interface and Implementation Notes

Beside element type and storage order template class `PackedStorage` requires a parameter to specify whether the upper or lower part of a matrix gets stored. For this reason FLENS declares the enumeration type

```
enum StorageUpLo {
    Upper,
    Lower
};
```

According to the storage order, elements are stored contiguously in a one-dimensional array as described in Section 3.1.6. Index bases for rows and columns have to be the same and the matrix has to be square (hence `dim` specifies the number of rows and columns):

```
template <typename T, StorageOrder Order, StorageUpLo UpLo>
class PackedStorage
{
    public:
        PackedStorage();

        PackedStorage(int dim, int indexBase=1);

        // ...

    private:
        int _dim, _indexBase;
        Array<T> _data;
};
```

Views are not supported for the packed storage scheme.

The interface for the packed storage scheme is documented in Appendix A.3.

3.2.7 Discussion: Software Quality of Storage Scheme Classes

In the following, the classes outlined in this section are briefly analyzed with respect to software quality. The most relevant impact is related to the following software quality factors:

1. **Correctness/Ease of use**: As was shown, allocation of storage schemes and memory management for views require one to incorporate very specific and technical details. Even worse, software bugs in this respect are typically hard to debug. Encapsulating storage schemes in classes does not only improve the ease of use, but also has a positive impact on correctness. Bundling all technical, low-level code in only a small number of classes allows specific and targeted testing. As a result, storage classes can be used as reliable building blocks.

   In Section 3.2.3 it was shown that 'overlapping views' can lead to unexpected results. While it would be possible to treat such cases in debug mode this is not realized in FLENS. This is due to the fact that such a check would imply a considerable runtime overhead in debug mode. But keeping the runtime overhead of the debug mode reasonable is a key requirement to ensure that this mode will be used in practice. One possible way out would be to allow different debug levels in the future.

2. **Efficiency/Robustness**: With respect to views a certain runtime overhead is induced due to reference counting. In most numerical applications it is not required that the lifetime of view objects can exceed the original object. At least it can be avoided rather easily. If runtime overhead due to the creation and destruction of views should become relevant one could consider the following policy: Only in debug mode checks ensure that actually no view object lives longer than its creator. This checks use the current mechanism for reference counting. While in non-debug mode all code related to reference counting would be omitted.

   View classes for the full storage scheme induce a further runtime overhead. This is because during instantiation an array with pointers to either rows or columns gets setup. If random access of elements is relevant for views this pays off. But still there might be cases where this overhead could be undesirable.

Whether quality requirements are fulfilled sufficiently depends on how the single factors are weighted. Furthermore, the weighting in general will vary between different types of application. This makes it obvious that a single implementation can never be considered as optimal.

This suggests to provide different implementations of a certain storage scheme. Each of these implementations can then be realized following a different efficiency policy. If all implementations comply to the same interface, these implementations are exchangeable. Obviously this is of great practical importance. While developing a scientific software, this allows using initially a default implementation. Once the correctness of the software is guaranteed, the implementation can be replaced with a specifically tuned version. In turn, the time required to ensure the correctness can be reduced by providing different implementations, realizing different levels of checks in debug mode.

3.3 BLAS Routines and their Embedding into C++

FLENS uses CBLAS a standardized C interface to BLAS. While BLAS itself only supports column major ordering, CBLAS does also support row major ordering. This section gives only a brief overview about the functionality provided by BLAS. A detailed documentation of BLAS and CBLAS can be found in [14] and a quick reference guide in [60].

3.3.1 Naming Conventions for Functions in BLAS and CBLAS

Names of CBLAS functions are related to correspondent BLAS routines as follows: the name of a CBLAS function is prefixed with `cblas_` followed by the name of the BLAS function in lowercase letters. For example, the CBLAS function `cblas_dgemv` interfaces with the BLAS routine `DGEMV`.

Following a strict naming convention the name of a BLAS routine encodes further information. The first letter represents the data type:

	C type	C++ type
s	single precision (`float`)	`float`
d	double precision (`double`)	`double`
c	single precision complex (real and imaginary part of type `float`)	`complex<float>`
z	double precision complex (real and imaginary part of type `double`)	`complex<double>`

Note that ANSI/ISO C does not specify complex data types. CBLAS expects that real and imaginary parts of a complex number are stored in consecutive memory locations. This conforms to the implementation of the STL class `complex<T>`.

For BLAS routines operating on matrices the next two letters indicate the matrix argument type[8]:

	Full Storage	Band Storage	Packed Storage
General	ge	gb	-
Symmetric	sy	sb	sp
Hermitian	he	hb	hp
Triangular	tr	tb	tp

To be more precise, these two letters indicate how functions interpret a particular storage scheme. An '`sb`-function' will treat a band storage scheme as a symmetric matrix (typically this function then will only access either the upper or lower part of the scheme). While a '`gb`-function' would treat the *same* scheme as a general matrix with band structure.

Remaining letters depict the operation performed. For instance `mv` stands for *matrix-vector product* and `mm` for *matrix-matrix product*. Consequently the CBLAS routine `cblas_dgemv` implements the matrix-vector product for a general matrix with full storage and elements of double precision.

Depending on the operations performed BLAS can be grouped into three levels. In the following these levels are further considered.

3.3.2 Level 1 BLAS

Level 1 BLAS provides vector-vector and scalar-vector operations. Letters `s`, `d`, `c`, `z` specifying the element type are only indicated by `X`:

operation	mathematical notation	CBLAS function
vector norms	$\text{nrm2} \leftarrow \|\|x\|\|_2$	`cblas_Xnrm2`
	$\text{asum} \leftarrow \|\|\text{re}(x)\|\|_1 + \|\|\text{im}(x)\|\|_1$	`cblas_Xasum`
	$\text{amax} \leftarrow \max(\|\text{re}(x_i)\| + \|\text{im}(x_i)\|)$	`cblas_Xamax`

[8]For routines that expect more than one matrix these letters indicate the most significant matrix type

dot product	dot $\leftarrow x^T y$ dot $\leftarrow x^H y$	`cblas_Xdot` `cblas_Xdotc`
vector assignment	$y \leftarrow x$	`cblas_Xcopy`
vector scaling	$x \leftarrow \alpha x$	`cblas_Xscal`
scaled vector addition	$y \leftarrow \alpha x + y$	`cblas_Xaxpy`

An expression like $z = \alpha x + \beta y$ can be evaluated by

$$\begin{array}{ll} z \leftarrow x & (\texttt{cblas\_Xcopy}) \\ z \leftarrow \alpha z & (\texttt{cblas\_Xscale}) \\ z \leftarrow \beta y + z & (\texttt{cblas\_Xaxpy}) \end{array}$$

Characteristic for the interface of all these functions is how vectors are passed. One has to pass the vector length, the stride between elements and a pointer to the first element. For example, the signature of `cblas_Xcopy` is

```
void
cblas_dcopy(const int      N,      // vector length
            const double   *X,     // pointer to first element of x
            const int      incX,   // stride between elements of x
            double         *Y,     // pointer to first element of y
            const int      incY);  // stride between elements of y
```

This works seamlessly with the array classes introduced in Section 3.2.3. Such that the assignment `y = x` can be implemented like this:

```
// debug mode: check x.length() == y.length()
cblas_dcopy(x.length(), x.data(), x.stride(),
                        y.data(), y.stride());
```

3.3.3 Level 2 BLAS: Matrix-Vector Operations

Matrix-vector operations specified by Level 2 BLAS involve matrix-vector products, triangular solve (forward-/backward substitution) as well as rank one and two updates. In the following op(A) can denote A, A^T or A^H:

operation	mathematical notation	CBLAS function

matrix-vector products	$y \leftarrow \alpha \text{op}(A)x + \beta y$
	$y \leftarrow \alpha A x + \beta y$
	$y \leftarrow \alpha A x + \beta y$
	$x \leftarrow \text{op}(A)x$
		`cblas_X{gemv, gbmv}`
		`cblas_X{hemv, hbmv, hpmv}`
		`cblas_X{semv, sbmv, spmv}`
		`cblas_X{trmv, tbmv, tpmv}`

triangular solve	$x \leftarrow A^{-1}x$ $x \leftarrow A^{-T}x$ $x \leftarrow A^{-H}x$	`cblas_X{trsv, tbsv, tpsv}`
rank one update	$A \leftarrow \alpha x y^T + A$	`cblas_Xger`
symmetric ($A = A^T$) rank one and two updates	$A \leftarrow \alpha x x^T + A$ $A \leftarrow \alpha x y^T + y x^T + A$	`cblas_Xsyr` `cblas_Xsyr2`
...	...	...

CBLAS defines the enumeration type

```
enum CBLAS_ORDER {CblasRowMajor, CblasColMajor};
```

to specify storage ordering. As was already noted in Section 3.1 in order to associate a storage scheme with a specific matrix type additional information is required. Such information gets encoded based on further enumeration types. For symmetric, hermitian and triangular matrices enumeration type

```
enum CBLAS_UPLO  {CblasUpper, CblasLower};
```

is used to declare that matrix elements are stored in either the upper or lower triangular part of the scheme. For triangular matrices the enumeration type

```
enum CBLAS_DIAG  {CblasUnit, CblasNonUnit};
```

allows one to specify whether the matrix is unit or non-unit triangular respectively. If a triangular matrix is supposed to be unit triangular then diagonal elements from the storage scheme will not be referenced by CBLAS functions.
Further op(A) can be defined as either A, A^T or A^H through

```
enum CBLAS_TRANSPOSE {CblasNoTrans, CblasTrans, CblasConjTrans};
```

Functionality and usage of Level 2 BLAS functions can be illustrated quite comprehensively for function `cblas_dtbmv`. It allows one to compute either $x \leftarrow Ax$ or $x \leftarrow A^T x$ where A is a banded triangular matrix. Because A is triangular, elements of vector x can be overwritten successively with the result of the matrix-vector product. The C++ interfaces for the band storage and vector storage schemes allow one to provide all further argument parameters for `cblas_dtbmv`:

```
// matrix-vector product: x = op(A)*x
void
cblas_dtbmv(const enum CBLAS_ORDER      Order,    // storage order
            const enum CBLAS_UPLO       upLo,     // access upper or lower part
            const enum CBLAS_TRANSPOSE  transA,   // op(A) = A or A^T
            const enum CBLAS_DIAG       diag,     // unit- or non-unit diagonal
            const int                   n,        // matrix dimension
            const int                   k,        // total number of diagonals
            const double               *A,        // pointer to matrix elements
            const int                   lda,      // leading dimension
            double                     *x,        // pointer to vector elements
            const int                   incX);    // stride in vector x
```

Also note that the vector length of x is assumed to equal the dimension of A.

3.3.4 Level 3 BLAS: Matrix-Matrix Operations

Level 3 BLAS provides functions for matrix-matrix products, symmetric rank k and $2k$ updates and triangular solve for multiple right hand sides. Only matrices with full storage are supported:

operation	mathematical notation	CBLAS function
matrix-matrix product	$C \leftarrow \alpha\mathrm{op}(A)\mathrm{op}(B) + \beta C$	`cblas_X{gemm, symm, hemm}`
symmetric ($A = A^T$)	$C \leftarrow \alpha AA^T + \beta C$	
rank k and $2k$ updates	$C \leftarrow \alpha A^T A + \beta C$	`cblas_Xsyrk`
	$C \leftarrow \alpha AB^T + \bar{\alpha} BA^T + C$	
	$C \leftarrow \alpha A^T B + \bar{\alpha} B^T A + C$	`cblas_Xsyr2k`
hermitian ($A = A^H$)	$C \leftarrow \alpha AA^T + \beta C$	
rank one and two updates	$C \leftarrow \alpha A^T A + \beta C$	`cblas_Xherk`
	$C \leftarrow \alpha x AB^H + \bar{\alpha} BA^H + C$	
	$C \leftarrow \alpha x A^H B + \bar{\alpha} B^H A + C$	`cblas_Xher2k`

Interface for `cblas_dgemm` computing $C \leftarrow \alpha\mathrm{op}(A)\mathrm{op}(B) + \beta C$:

```
void
cblas_dgemm(const enum CBLAS_ORDER      Order,    // storage order
            const enum CBLAS_TRANSPOSE  transA,   // op(A) = A or A^T
            const enum CBLAS_TRANSPOSE  transB,   // op(B) = B or B^T
            const int              m,       // number of rows in C
            const int              n,       // number of columns in C
            const int              k,       // number of columns of op(A)
            const double           alpha,   // scalar alpha
            const double          *A,       // pointer to matrix elements
            const int              lda,     // leading dim. of A
            const double          *B,       // pointer to matrix elements
            const int              ldb,     // leading dim. of B
            const double           beta,    // scalar beta
            double                *C,       // pointer to matrix elements
            const int              ldc);    // leading dim. of C
```

Note that the number of columns in op(A) has to equal the number of rows in op(B).

3.3.5 C++ Interface for BLAS

FLENS provides for each CBLAS function a corresponding C++ wrapper function. One purpose of these C++ wrapper functions is to map FLENS enumeration types onto CBLAS enumeration types:

```
void
tbmv(StorageOrder Order, StorageUpLo UpLo, Transpose Trans, UnitDiag Diag,
     int n, int k, const double *A, const int lda, double *X, int incX)
{
    CBLAS_ORDER order     = (Order==RowMajor) ? CblasRowMajor : CblasColMajor;
    CBLAS_UPLO upLo       = (UpLo==Upper)     ? CblasUpper    : CblasLower;
    CBLAS_TRANSPOSE trans = (Trans==NoTrans)  ? CblasNoTrans  : CblasTrans;
    CBLAS_DIAG diag       = (Diag==NonUnit)   ? CblasNonUnit  : CblasUnit;

    cblas_dtbmv(order, upLo, trans, diag, n, k, A, lda, X, incX);
}
```

A prefix analogously to `cblas_` can be omitted as these functions are defined inside `namespace flens`. Further the wrapper functions are overloaded with respect to the element type:

```
void
tbmv(StorageOrder Order, StorageUpLo UpLo, Transpose Trans, UnitDiag Diag,
     int n, int k, const float *A, const int lda, float *X, int incX)
{
// call cblas_stbmv
}
```

Hence also a prefix specifying the element type can be left out.

Usage of BLAS can now be handled more flexibly. In the following example function `example_bs` performs two matrix-vector multiplications (interpreting a band storage scheme first as upper triangular then as lower unit triangular matrix):

```
template <typename BS, typename ARRAY>
void
example_bs(const BS &A, ARRAY &x)
{
    assert(A.numRows()==A.numCols());
    assert(A.numRows()==x.length());

    // compute:  x = upperTriangular(A)*x
    tbmv(StorageInfo<BS>::order, Upper, NoTrans, NonUnit,
         A.numRows(), A.numSubDiags()+A.numSuperDiags()+1,
         A.data(), A.leadingDimension()
         x.data(), x.stride());

    // compute:  x = lowerUnitTriangular(A)*x
    tbmv(StorageInfo<BS>::order, Lower, NoTrans, Unit,
         A.numRows(), A.numSubDiags()+A.numSuperDiags()+1,
         A.data(), A.leadingDimension()
         x.data(), x.stride());
}
```

Note that `A` can be of type `BandStorage`, `BandStorageView` or `ConstBandStorageView`, while `x` can only be of type `Array` or `ArrayView`. Element types of `A` and `x` have to be the same.

The wrapper functions sketched here can be regarded as low-level C++ wrapper functions for BLAS. Except for the element type there is no further abstraction provided. As can be seen from the example above, providing the same functionality of `example_bs` for packed or full storage schemes would require new implementations of similar functions.

Obviously this set of wrapper functions is still not sufficient to thoroughly support generic programming. This aim will be achieved through another set of wrapper function introduced in Section 4.3.1. These wrapper function can be considered as high-level wrappers and are built on top of the low-level wrappers introduced here.

3.4 LAPACK

LAPACK implements more than 300 functions such that it is impossible to provide here a comprehensive overview on its functionality. This section gives only a brief overview about some LAPACK functions related to solving systems of linear equations. While this shows only a small fraction of LAPACK, it allows one to reflect important requirements for the implementation of efficient numerical libraries.

Even for a single numerical algorithm LAPACK provides quite a variety of implementations. For a concrete problem one can choose that implementation which is most suitable to exploit particular properties of this problem. Note that this is in diametral contrast to the idea of providing one general implementation that is applicable on all types of problems. Instead one ideally would like for each concrete problem a hand-optimized implementation. Choosing one of the first numerical methods taught in undergraduate classes allows one most easily to exemplify and motivate how to adapt algorithms to specific problems.

3.4.1 Motivation: LU Factorization

Consider the system of linear equations

$$Ax = b \tag{3.1}$$

with given coefficient matrix $A \in \mathbb{R}^{n\times n}$, right-hand side $b \in \mathbb{R}^n$ and unknowns $x \in \mathbb{R}^n$. Using Gaussian elimination one can find a permutation matrix P, a lower unit triangular matrix L and an upper triangular matrix U such that

$$A = PLU. \tag{3.2}$$

Equation 3.2 constitutes the LU factorization of A. Obviously 3.1 is equivalent to

$$PLUx = b \tag{3.3}$$

and can be solved in three steps:

1. compute $\tilde{b} = P^{-1}b = P^T b$,
2. solve $Ly = \tilde{b}$ for y,
3. solve $Ux = y$ for x.

3.4.2 Practical Aspects of the LU Factorization

Permutation matrix P is defined through row-interchanges carried out during the factorization. Therefore a favorable representation of P is a vector containing the pivot indices. A pivot vector $p = (p_1, \dots, p_n)$ documents that during the factorization the i-th row was interchanged with the p_i-th row. More formally, p stores a sequence of transpositions. Let $\tau_{i,j}$ denote the transposition interchanging the i-th and j-th element of a n-tupel. Then p represents the permutation $\pi = \tau_{1,p_1} \circ \cdots \circ \tau_{n,p_n}$. An implementation can compute $b \leftarrow Pb$ efficiently in a single loop:

```
for i = 1,..,n
     swap b(i) and b(p(i))
```

As the inverse permutation is $\pi^{-1} = \tau_{n,p_n} \circ \cdots \circ \tau_{1,p_1}$ traversing the same loop backwards allows one to compute $b \leftarrow P^{-1}b$:

```
for i = n,..,1
     swap b(i) and b(p(i))
```

In general coefficient matrix A can be overwritten with L and U during the factorization, that is

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{pmatrix} \quad \rightsquigarrow \quad A_{LU} = \begin{pmatrix} u_{1,1} & u_{1,2} & u_{1,3} & u_{1,4} \\ l_{2,1} & u_{2,2} & u_{2,3} & u_{2,4} \\ l_{3,1} & l_{3,2} & u_{3,3} & u_{3,4} \\ l_{4,1} & l_{4,2} & l_{4,3} & u_{4,4} \end{pmatrix}.$$

The total number of floating-point operation is of order $\frac{2}{3}n^3 + \mathcal{O}(n^2)$.

If A is a tridiagonal matrix, than the number of the floating-point operation reduces to $\mathcal{O}(n)$. However, matrix U will have in general two super-diagonals. For an efficient implementation this requires that the storage scheme holding A allows one to overwrite its second super-diagonal (as indicated by '$*$'):

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & * & 0 & 0 \\ a_{2,1} & a_{2,2} & a_{2,3} & * & 0 \\ 0 & a_{3,2} & a_{3,3} & a_{3,4} & * \\ 0 & 0 & a_{4,3} & a_{4,4} & a_{4,5} \\ 0 & 0 & 0 & a_{5,4} & a_{5,5} \end{pmatrix} \quad \rightsquigarrow \quad A_{LU} = \begin{pmatrix} u_{1,1} & u_{1,2} & u_{1,3} & 0 & 0 \\ m_{2,1} & u_{2,2} & u_{2,3} & u_{2,4} & 0 \\ 0 & m_{3,2} & u_{3,3} & u_{3,4} & u_{3,5} \\ 0 & 0 & m_{4,3} & u_{4,4} & u_{4,5} \\ 0 & 0 & 0 & m_{5,4} & u_{5,5} \end{pmatrix}.$$

Elements of L are stored (in general rearranged due to pivoting) on the sub-diagonal of A_{LU}.

This can be generalized for a general banded matrix with k_l sub-diagonals and k_u super-diagonals. The total number of floating-point numbers is approximately $2n(k_l + k_u + 1)k_l$. That is, of order $\mathcal{O}(n)$ if k_l, k_u are much smaller than n. Triangular matrices U and L will have $k_l + k_u$ super-diagonals and k_l sub-diagonals respectively.

3.4.3 LAPACK Routines for *LU* Factorization and Triangular Solvers

The same naming conventions used in BLAS are used by LAPACK. Where appropriate LAPACK provides implementations that are especially adopted for tridiagonal matrices. This is for instance indicated by '`gt`' in the routine name standing for 'general tridiagonal matrix'. For tridiagonal matrices diagonals are simply stored in separate vectors such that no further storage scheme needs to be introduced.

For solving systems of linear equations using the *LU* factorization LAPACK provides three groups of functions:

1. ***LU* Factorization: Compute $A \leftarrow LU$ and p**
   Routines `Xgetrf`, `Xgbtrf` and `Xgttrf` overwrite a coefficient matrix A with its *LU* decomposition and provides a pivot vector p as output. As mentioned above for banded and tridiagonal matrices the storage scheme holding A has to be sufficiently large.

2. **Triangular Solver**
   Once the factorization is computed, triangular solver `Xgetrs`, `Xgbtrs` or `Xgttrs` can be used to solve the system of linear equations. These solvers can also be used for multiple right-hand sides, i. e. to solve

$$AX = B, \quad X = (x_1, \ldots, x_k),\ B = (b_1, \ldots, b_k)\ \in\ \mathbb{R}^{n \times k}$$

   simultaneously for X. Hereby the right-hand side B will be overwritten with solution X.

3. Driver Routines: `Xgesv`, `Xgbsv` and `Xgtsv` are driver routines combining factorization and triangular solver from above. Arguments are coefficient matrix A, right-hand side B and a index vector p. On exit these will be overwritten with the factorization, solution and the pivot indices respectively.

3.4.4 C++ Interface for LAPACK

FLENS can be linked with any LAPACK implementation and does not depend on external C-interfaces such as CLAPACK[57]. Instead, FLENS directly interfaces with the Fortran routines of LAPACK. This is because C-interfaces for LAPACK — as opposed to CBLAS for BLAS — do not provide an additional benefit like support for row major storage. Only few LAPACK implementations (e. g. ATLAS [88]) do actually support row major storage but usually only for a small subset of LAPACK.

FLENS interfaces with LAPACK by declaring for each routine a corresponding external function, e. g.

```
extern "C" {
    void sgetrf_(int *m, int *n, float *a, int *lda, int *ipiv, int *info);

    void dgetrf_(int *m, int *n, double *a, int *lda,  int *ipiv, int *info);

    ...
}
```

Common to most LAPACK functions is the last argument `info`. On return it contains a value used for diagnostics. Note that Fortran routines have to receive all argument parameters by reference, i. e. via pointers. For each external declaration FLENS implements a C++ wrapper function. These allow that arguments can be passed by value and the diagnostics `info` becomes the return value. These functions are further overloaded with respect to the element type:

```
// sgetrf
int
getrf(int m, int n, float *a, int lda, int *ipiv)
{
    int info;
    sgetrf_(&m, &n, a, &lda, ipiv, &info);
    return info;
}

// dgetrf
int
getrf(int m, int n, double *a, int lda, int *ipiv)
{
    // ...
}
```

The overloaded `getrf` functions have input parameters:

`m`	numbers of rows in matrix A,
`n`	numbers of columns in matrix A,
`a`	pointer to elements of A,
`lda`	leading dimension of `a`.

and output parameters:

`a`	overwritten by the LU factorization,
`ipiv`	pointer to array storing the pivots of the factorization,
`info`	integer used for diagnostics: `info = 0` indicates successful execution, `info =-i` that the `i`-th parameter had an illegal value and `info = i` that $u_{ii} = 0$.

Using the C++ classes for the full storage scheme and vector storage scheme the *LU* factorization of a matrix can be computed by

```
FullStorage<double, ColMajor> A(5,5);
Array<int>                    p(5);

// init A

int info = getrf(A.numRows(), A.numCols(),
                 A.data(), A.leadingDimension(),
                 p.data());
if (info!=0) {
    // error handling
}
```

Note that for most LAPACK implementations only column major storing is supported. Furthermore p has to be an array with element stride equal to one. Such consistency checks will be performed by higher-level LAPACK wrappers introduced in Section 4.4 from the next chapter.

3.5 Discussion on Software Quality: Improvements and Deficiencies

The interface for BLAS and LAPACK presented in this chapter consists of two parts: classes for storage schemes and C++ wrapper functions for BLAS and LAPACK routines.

As was already discussed in Section 3.2.7, notable benefits result from the storage classes with respect to correctness, robustness and ease of use. The proper allocation and handling of the underlying data structures is not only an error prone task but also requires a good understanding of many technical details. Checks in debug mode (e. g. for valid indices) allow for tracking down typical errors while recompiling in non-debug mode still provides optimal performance.

Wrapper functions for BLAS and LAPACK exhibit a rather low level of abstraction. In order to be independent of concrete BLAS and LAPACK implementations FLENS defines its own enumeration types, e. g., to indicate the storage order (`StorageOrder`) or storage of lower or upper triangular parts (`StorageUpLo`). The wrapper functions are overloaded with respect to the element type such that a wrapper like `axpy` provides a uniform interface for the BLAS functions `saxpy`, `daxpy`, `caxpy` and `zaxpy`. But beside this, the interface of a wrapper function is almost identical with the interface of the underlying BLAS or LAPACK function.

Due to the low level of abstraction, an implementation of a wrapper function only requires a few lines of code. Given the considerable amount of BLAS and LAPACK functions, it is a crucial requirement that the implementation of wrapper functions is a trivial task[9]. Further, this guarantees that non of the features provided by BLAS and LAPACK gets lost due to improper abstraction.

When interfacing with other external libraries the practical need for low level interfaces remains apparent. Programmers familiar with an external library and the relevant data structures can provide a interface without deep knowledge of FLENS. Obviously this is crucial for maintainability. For this reason the BLAS and LAPACK interface presented in this chapter was realized independent of the remaining FLENS library. Thus, it easily could be separated and maintained as a pure C++ interface for BLAS and LAPACK (analogously to pure C interfaces like CBLAS and CLAPACK).

On the flip side, deficiencies remain that are related to reusability and flexibility. Consider a numerical algorithm that merely requires the computation of a matrix-vector product. A flexible and reusable realization should allow that the same implementation can be used for different matrix types. However, this is not possible as the various BLAS wrappers (e. g. `gemv`, `gbmv`, `symv`, `sbmv`, `trmv`, . . .) have different signatures. A higher level BLAS interface introduced in the following chapter resolves this issue.

[9]In principle it would be possible to automatically generate most of the FLENS wrapper functions.

4 FLENS: Design and Implementation Concepts

This chapter describes the most fundamental design and implementation concepts realized in the FLENS library. These concepts are mainly exemplified by considering the integration of the functionality provided by BLAS and LAPACK. The generality of the realized approach will be demonstrated in the following chapters.

Section 4.1 introduces the techniques used for the realization of matrix and vector class hierarchies. Section 4.2 then illustrates how matrix and vector types for BLAS and LAPACK are integrated into these hierarchies. Section 4.3 describes a high-level interface for linear algebra operations. This interface in particular can serve as a high-level interface for BLAS. The section further addresses how FLENS avoids runtime overhead due to abstraction. Section 4.4 briefly outlines the high-level interface for LAPACK which is realized in FLENS. The chapter is concluded by a discussion on improvements and remaining deficiencies with respect to software quality.

4.1 Implementation of Class Hierarchies in Scientific Libraries

In the following, the FLENS class hierarchies for matrices are introduced. The matrix hierarchy is organized based on two levels of specialization. The first level specifies a matrix to be either general, triangular, symmetric or hermitian. On a second level these can be further specialized with respect to properties that are relevant for a concrete implementation. For instance, a triangular matrix with band structure and a symmetric block matrix are specializations of a triangular and symmetric matrix respectively.

An extract of the FLENS matrix hierarchy is shown in Figure 4.1. From the diagram one can read that class `TriangularMatrix` is a specialization of `Matrix`. Class `TbMatrix` in turn is a specialization of `TriangularMatrix`.

FLENS realizes its class hierarchies by applying the *Curiously Recurring Template Pattern (CRTP)* [78]. In many applications this design pattern is adopted merely for the sake of efficiency. This is because the CRTP allows to achieve static polymorphism and therefore allows to avoid virtual functions. In FLENS the realization of the CRTP in addition addresses the ease of integrating new matrix and vector types. Integration of new types in particular requires interoperability with existing types and reusability of algorithms. As will become evident in Section 4.3 FLENS provides for this purpose a coherent mechanism breaking the complexity of extending the library.

However, in order to apply the CRTP effectively to matrix and vector types some technical hurdles have to be overcome. After briefly introducing the CRTP the remaining section addresses these obstacles together with the solutions realized in FLENS.

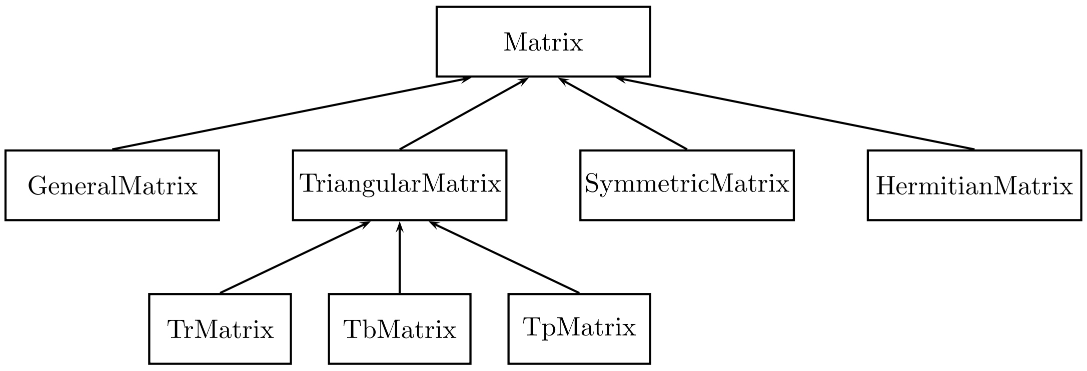


Figure 4.1: Extract of the FLENS matrix hierarchy.

4.1.1 Curiously recurring template pattern (CRTP)

Consider a base class `Base` declaring an abstract method `aMethod` implemented in a derived class `Implementation`. For this example Listing 4.1 illustrates the basic idea of applying the CRTP to avoid virtual functions. The key point is that the base class receives the type of the derived class as template parameter (line 17). This allows the base class `Base` to provide a method `impl` (line 5) for statically casting an instance to its derived type. Using the `impl` method a call of `aMethod` gets dispatched (line 12) to the implementation provided by the derived class (line 21). As the type of the derived type is known at compile time this can be inlined by the compiler.

```
 1 template <typename Impl>
 2 class Base
 3 {
 4     public:
 5         Impl &impl()
 6         {
 7             return static_cast<Impl &>(*this);
 8         }
 9
10         void aMethod()
11         {
12             impl().aMethod();
13         }
14 };
15
16 class Implementation
17     : public Base<Implementation>
18 {
19     public:
20
21         void aMethod()
22         {
23             // ...
24         }
25 };
```

Listing 4.1: Curiously recurring template pattern (CRTP)

4.1.2 CRTP and Derived Classes with Template Parameters

Consider that a derived class has at least one template parameter. If methods declared in the base class depend on this template parameter the CRTP can not be applied in a straightforward manner. Essentially it has to be possible to retrieve template parameters of the derived class through typedefs in the base class. However, a first and obvious attempt illustrated in the following code snippet fails:

```
 1 template <typename Impl>
 2 class Base
 3 {
 4     public:
 5         typedef typename Impl::T_Type T_Type;
 6
 7     // ...
 8 };
 9
10 template <typename T>
11 class Implementation
12     : public Base<Implementation<T> >
13 {
14     public:
15         typedef T T_Type;
16
17     // ...
18 };
```

The typedef inside the base class (line 5) falls back on the typedef in the derived class (line 15). Obviously this can not work. In order to instantiate for example objects of type `Derived<double>` the compiler first has to parse `Base<Derived<double> >`. However, `Base<Derived<double> >` in turn depends on `Derived<double>::T_Type`. Hence instantiation is impossible due to this recursive dependency.

'Outsourcing' the typedef from `Derived` is a possible way out. Technically this can be achieved using a trait class `TypeInfo`:

```
 1 template <typename I>
 2 struct TypeInfo
 3 {
 4 };
 5
 6 template <typename Impl>
 7 class Base
 8 {
 9     public:
10         typedef typename TypeInfo<Impl>::T_Type T_Type;
11
12     // ...
13 };
```

The typedef in line 10 can now be defined before any instantiation takes place. Merely a forward declaration of `Derived` is required:

```
template <typename T>
class Implementation;

template <typename T>
struct TypeInfo<Implementation<T> >
{
    typedef T T_Type;
};
```

Except for the need of specializing the `TypeInfo` trait the derived class can now be defined as before:

```
template <typename T>
class Implementation
    : public Base<Implementation<T> >
{
    // ...
};
```

4.1.3 CRTP and Multilevel Inheritance

Support for inheritance trees more than one level deep constitutes the next obstacle. Consider the class hierarchy with two-level inheritance shown in Figure 4.2. Classes `Base` and `Derived` declare abstract functions that are implemented in class `Implementation`. For exemplification

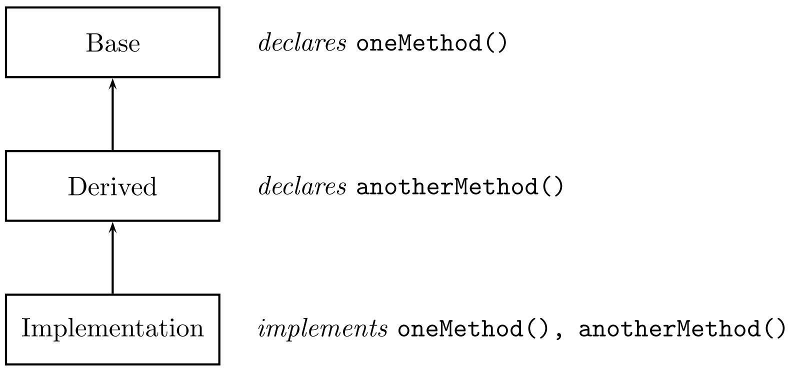

Base
declares oneMethod()
Derived
declares anotherMethod()
Implementation
implements oneMethod(), anotherMethod()

Figure 4.2: Two Level Inheritance Tree.

consider functions `dummy1` and `dummy2`:

```
template <typename I>
void
dummy1(Base<I> &base)
{
    base.oneMethod();
}

template <typename I>
void
dummy2(Derived<I> &derived)
{
    derived.oneMethod();
    derived.anotherMethod();
}
```

Applying the CRTP to achieve static polymorphism requires that objects of type `Base` and `Derived` can be converted to an object of type `Implementation` by a static cast . In general this requires that the type of the most derived class is known to the base class at compile-time.

In a one-level inheritance tree the type of the implementation coincides with the template parameter provided to the base class. In a multi-level inheritance the corresponding type can not be retrieved directly as can be seen from the above example:

`Implementation<T>` is derived from `Derived<Implementation<T> >` and in turn

`Derived<Implementation<T> >` is derived from `Base<Derived<Implementation<T> > >`.

To retrieve the implementation type the `TypeInfo` trait can be extended to provide a corresponding `Impl` typedef. Again this trait needs to be specialized for each derived class. For class `Derived` such a specialization can be defined by

```
template <typename I>
class Derived;

template <typename I>
struct TypeInfo<Derived<I> >
{
    typedef typename TypeInfo<I>::T_Type  T_Type;
    typedef typename TypeInfo<I>::Impl    Impl;
};
```

Note that typedefs in line 7 and 8 are in turn defined using the `TypeInfo` trait. This requires the specialization of `TypeInfo` for the implementing class:

```
template <typename T>
class Implementation;

template <typename T>
struct TypeInfo<Implementation<T> >
{
    typedef T                   T_Type;
    typedef Implementation<T>   Impl;
};
```

Obviously this can be applied analogously to inheritance trees that are more than two levels deep. `TypeInfo` definitions for abstract classes get resolved iteratively by the specialization of `TypeInfo` for the implementing class.

The CRTP can now be applied as before. The base class provides the method `impl` statically converting objects into their derived type:

```
template <typename I>
class Base
{
    public:
        typedef TypeInfo<I>::Impl    Impl;
        typedef TypeInfo<I>::T_Type T_Type;

        Impl &impl() {
            return static_cast<Impl &>(*this);
        }

        void oneMethod() {
            impl().oneMethod();
        }
};
```

Analogously class `Derived` declares the abstract method `anotherMethod`:

```
template <typename I>
class Derived
    : public Base<Derived<I> >
{
    public:

        void anotherMethod() {
            this->impl().anotherMethod();
        }
};
```

Implementations of `oneMethod()` and `anotherMethod()` are provided by the most specialized class in the usual manner:

```
template <typename T>
class Implementation
    : public Derived<Implementation<T> >
{
    public:
        void oneMethod() {
            // ...
        }

        void anotherMethod() {
            // ...
        }
};
```

As demonstrated next the CRTP in this form is suitable for the realization of efficient matrix or vector hierarchies.

4.2 Matrix and Vector Classes for BLAS and LAPACK

While FLENS aims to be more than just a high-level interface to BLAS and LAPACK this was considered as a minimal requirement. This section is dedicated to the implementation of matrix and vector types based on the storage classes introduced in Chapter 3. These types are in particular tailored for the interaction with BLAS and LAPACK.

4.2.1 Design Pattern

As was shown in Section 3.3 BLAS distinguishes between eleven matrix types. Adapting the BLAS naming conventions FLENS defines corresponding matrix classes:

	Full Storage	Band Storage	Packed Storage
General	`GeMatrix`	`GbMatrix`	-
Symmetric	`SyMatrix`	`SbMatrix`	`SpMatrix`
Hermitian	`HeMatrix`	`HbMatrix`	`HpMatrix`
Triangular	`TrMatrix`	`TbMatrix`	`TpMatrix`

FLENS separates interface and implementation of matrix types following the *bridge pattern* [31]. For instance, class `GbMatrix` acts as interface for a general matrix with band structure. Possible implementations are provided through classes realizing the band storage scheme like `BandStorage`, `BandStorageView` or `ConstBandStorageView` introduced in Chapter 3. As there is no need to exchange the implementation for a matrix or vector type at runtime the bridge pattern can be implemented using templates [78]:

```
template <typename BS>
class GbMatrix
    : GeneralMatrix<GbMatrix<BS> >
{
    public:
        typedef typename GbMatrix<BS>::ElementType  T;

        T
        operator() (int i, int j)
        {
            return _bs(i,j);
        }

        int
        numRows() const
        {
            return _bs.numRows();
        }

        // ...

    private:
        BS _bs;
};

template <typename BS>
struct TypeInfo<GbMatrix<BS> >
{
    typedef GbMatrix<BS> Impl;
    typedef typename BS::ElementType ElementType;
};
```

Functionality gets directly delegated to the encapsulated storage scheme. Corresponding function calls can be inlined by a compiler such that there is no inevitable runtime overhead due to abstraction.

Encapsulated storage schemes are often denoted as *engines* reflecting the exchangeability and the responsibility to fulfill specific tasks.

As was noted in Section 3.1.1 a certain storage scheme can represent various matrix types. Working with storage schemes directly the consistent and correct interpretation of a scheme is completely up to the user. Through a matrix class such as `GbMatrix` a storage scheme gets statically tagged as a general matrix with band structure. This improves type safety and therefore allows a compiler to check and enforce the correct interpretation. Further the use of this design pattern allows realization of a simplified BLAS interface as will be shown in Section 4.3.1.

For other matrix types besides the storage scheme, further information has to be encapsulated. For a symmetric matrix with banded structure the corresponding matrix type also needs to store whether the upper or lower triangular part of the storage scheme gets referenced[1]:

```
template <typename BS>
class SbMatrix
    : SymmetricMatrix<SbMatrix<BS> >
{
    public:

        StorageUpLo
        upLo() const
        {
            return _upLo;
        }

        // ...

    private:
        BS _bs;
        StorageUpLo _upLo;
};

// TypeInfo definition ...
```

Analogously to matrix types, class `DenseVector` serves as interface for dense vectors suited to use `Array`, `ArrayView` or `ConstArrayView` as implementation.

4.2.2 Engine Interface

Interoperability with other libraries requires access to internal data structures of vectors and matrices. For this purpose FLENS provides appropriate methods in its matrix and vector classes. Method `engine` provides access to the underlying storage scheme. For convenience methods like `leadingDimension` provide direct access to frequently needed components of the storage scheme.

```
template <typename FS>
class SyMatrix
{
    public:

        int
        leadingDimension() const
        {
            return _fs.leadingDimension();
        }

        // ...

        FS &
        engine();

        const FS &
```

[1]The same applies to hermitian matrices. Triangular matrices in addition have to store whether the diagonal should be assumed to be unit or not.

```
        engine() const;

    private:
        FS _fs;
        StorageUpLo _upLo;
};
```

4.2.3 Matrix and Vector Views

FLENS makes it possible that the same storage scheme can represent different matrix (or vector) types at the same time. Due to the clean separation of storage schemes and matrix types this can easily be realized.

Consider as an example the *LU* factorization of a square matrix *A*. First the matrix gets initialized and then factorized. Using the factorization to solve a system of linear equations the lower-unit and upper triangular part of the square matrix are reinterpreted as triangular matrices. In order to realize this the triangular matrices need to reference the underlying storage scheme of *A*. If *A* is of type

```
GeMatrix<FullStorage<double, ColMajor> >      A(N,N);
```

then methods `upper` and `lower` create corresponding triangular matrices referencing — not copying — the storage scheme of *A*

```
TrMatrix<FullStorageView<double, ColMajor> >  U = A.upper();
TrMatrix<FullStorageView<double, ColMajor> >  L = A.lowerUnit()
```

Using standardized typedefs of the FLENS matrix types allows for the creation of triangular (lines 5-6), symmetric (line 8), hermitian and general matrices referencing the same storage scheme[2]:

```
1 typedef GeMatrix<FullStorage<double, ColMajor> >   GEMatrix;
2
3 GEMatrix A(N,N);
4
5 GEMatrix::TriangularView U = upper(A);
6 GEMatrix::TriangularView L = lowerUnit(A);
7
8 GEMatrix::SymmetricView  S = upper(A);
```

This allows a reusable implementation of numerical methods that are independent of particular matrix or vector types through generic programming:

```
template <typename MatA, typename MatB>
void
someNumericalMethod(MatA &A, const MatB &B)
{
    // ...
    typename MatA::TriangularView      U = upper(A);
    typename MatB::ConstSymmetricView  S = upper(B);
    // ...
}
```

Note that these typedefs in turn rely on typedefs of the underlying storage scheme (line 10-12):

```
1 template <typename FS>
2 class GeMatrix
3     : public GeneralMatrix<GeMatrix<FS> >
4 {
5     public:
6         // shortcut for element type
7         typedef typename GeMatrix<FS>::ElementType  T;
8
```

[2]Function `upper(A)` just calls `A.upper()` and is provided for the sake of a more readable and natural syntax.

```
        // view types from FS
        typedef typename FS::ConstView          ConstFSView;
        typedef typename FS::View               FSView;
        typedef typename FS::NoView             FSNoView;

        // view types for GeMatrix
        typedef TrMatrix<ConstFSView>           ConstTriangularView;
        typedef TrMatrix<FSView>                TriangularView;
        typedef TrMatrix<FSNoView>              TriangularNoView;
        // ...
};
```

If a triangular part should be copied instead of referenced the `NoView` variant can be used:

```
GEMatrix::TriangularNoView T = upper(A);  // copy the upper triangular of A
```

For referencing slices of matrices FLENS provides a convenient syntax. For example, a vector referencing the third column can be created by

```
GEMatrix::VectorView col = A(_,3);  // reference 3rd column
```

Hereby `_` is denoted as *range operator* and in this case used to select all rows of `A`. To select parts of the column variants `_(from, to)` and `_(from, stride, to)` can be used:

```
// reference the first five elements of 3rd column
GEMatrix::VectorView col2 = A(_(1,5),3);

// reference every second element of 3rd column
GEMatrix::VectorView col3 = A(_(1,2,N),3);
```

A more thorough overview of the range operator and handling of views can be found in the FLENS tutorial [45].

4.3 Linear Algebra Operations

In FLENS functions for linear algebra operations provide a simple and convenient interface. These functions provide the same functionality as the BLAS routines but can be seamlessly extended to support other matrix and vector types. Compared to the BLAS interfaces the number of arguments is reduced and in particular arguments related to technical details of storage schemes are completely omitted.

Using overloaded operators in C++ for linear algebra operations simplifies the task to implement reusable numerical methods. However, it is not trivial to achieve this without sacrificing efficiency. FLENS combines and extends different existing approaches for this purpose. The mechanism implemented in FLENS is in particular designed for the integration of new matrix or vector types as well as new types of linear algebra operations.

4.3.1 FLENS Interface for BLAS

In Section 3.3.5 a simple BLAS wrapper was introduced. Overloaded versions make it possible to skip the first letters from the function names that usually indicates the element type. The design of the FLENS matrix and vector classes allows for further simplifications without a negative impact on performance. Consider the BLAS routine `gemv` from Section 3.3.5 for the matrix-vector product of the form

$$y \leftarrow \alpha \mathrm{op}(A)x + \beta y, \quad \mathrm{op}(A) = A, A^T \text{ or } A^H. \tag{4.1}$$

Letters `ge` in the function name are used to indicate that matrix A uses the full storage format. Now this information can be moved into the signature of the function. Furthermore FLENS types like `GeMatrix` or `DenseVector` encapsulate all technical details describing the underlying storage schemes like leading dimension, strides or storage order. This suggests to implement another wrapper `mv` on top of `gemv`. It's main benefit is a slimed down signature and therefore simplified usage:

```
GeMatrix<FullStorage<double, ColMajor> > A(m,n);
DenseVector<Array<double> >              y(m), x(n);
double                                   alpha, beta;

// init A, x

mv(trans, alpha, A, x, beta, y);
```

The wrapper itself merely calls the adequate BLAS function. Required parameters for the BLAS function are retrieved from matrix and vector objects:

```
template <typename ALPHA, typename MA, typename VX,
          typename BETA, typename VY>
void
mv(Transpose trans,
   ALPHA alpha, const GeMatrix<MA> &A, const DenseVector<VX> &x,
   BETA beta, DenseVector<VY> &y)
{
    // check pre-conditions

    gemv(StorageInfo<MA>::order,
         trans, A.numRows(), A.numCols(),
         alpha,
         A.data(), A.leadingDimension(),
         x.data(), x.stride(),
         beta,
         y.data(), y.stride());
}
```

Reasonable preconditions like appropriate dimensions of A and x are placed before calling the BLAS function. FLENS further checks size of left-hand side arguments and performs a resize if needed:

```
assert(x.length()==((trans==NoTrans) ? A.numCols()
                                     : A.numRows()));
int yLength = (trans==NoTrans) ? A.numRows() : A.numCols();
if (y.length()!=yLength) {
    y.resize(yLength);
}
```

Wrappers for other BLAS functions are implemented following the same pattern, that means:

1. type information moves from the function name to the signature,
2. preconditions are checked through assertions,
3. left-hand side arguments are resized if needed.

Depending on the matrix type signatures of the wrappers are as follows:

matrix type	signature	functionality
general	`mv(trans, alpha, A, x, beta, y)`	$y \leftarrow \alpha \ \mathrm{op}(A)x + \beta y$ $\mathrm{op}(A) = A, A^T$ or A^H
hermitian	`mv(       alpha, A, x, beta, y)`	$y \leftarrow \alpha Ax + \beta y$
symmetric	`mv(       alpha, A, x, beta, y)`	$y \leftarrow \alpha Ax + \beta y$
triangular	`mv(trans,        A, x          )`	$x \leftarrow Ax$

User-defined matrix and vector types can be integrated by a user-defined wrapper with appropriate signature.

More comfortable and safer access of BLAS without performance loss is only one advantage. An even more important benefit is eligibility for generic programming. Consider the conjugated gradient method for solving a system of linear equations $Ax = b$, where A is symmetric and positive definite. The most essential operation is the matrix-vector product. Using the wrapper the method can be implemented independent of concrete types:

```
template <typename MA, typename VX, typename VY>
void
cg(const SymmetricMatrix<MA> &A,
   const Vector<VX> &x,
   Vector<VY> &b)
{
    ...

    // Ap = A*p
    mv(1, A, p, 0, Ap);

    ...
}
```

Section 4.3.2 illustrates how FLENS supports notations like `Ap =A*p` for the matrix-vector product. An essential issue in this respect is that without precautions such convenient notations would easily introduce a considerable runtime overhead.

Impact on Software Quality

The introduced types of BLAS wrappers have a positive impact on software quality:

1. **Efficiency**: Checks for correct dimensions of the right-hand side arguments are only performed in debug mode. Only checking dimensions of the left-hand side arguments and resizing them if needed impose a marginal additional runtime overhead in non-debug mode.

2. **Robustness/Correctness/Ease of use**: Checking for consistent dimensions is a major feature. To this extent such tests are not possible in underlying BLAS implementations as they receive data structures through pointers. BLAS wrappers as well as the matrix and vector classes encapsulate and hide technical details regarding storage formats. These pieces can easily be tested thoroughly to provide solid and easy to use building blocks for numerical applications.

3. **Reusability/Extensibility**: Numerical algorithms can be implemented using generic programming and therefore can be reused for different types of vectors and matrices. Hereby an algorithm depends on a specific set of operations (e. g. matrix-vector product, vector sums). Proper documentation of this set in addition to a specification of their interface allows for the application of user-defined matrix or vector types. Due to the design of the FLENS matrix or vector hierarchy, such interfaces can be kept very simple and therefore easy to support.

4.3.2 Overloading Operators: Existing Approaches

Using overloaded operators for linear algebra operations increases expressiveness and readability of code. For a numerical library this is obviously a nice feature. Mathematical notations are directly reflected in the source code. Code written by others becomes more comprehensible even for people not so familiar with C++ or the FLENS library. But it also can speed up and simplify implementation of numerical applications. With respect to software quality an expressive notation can be more than just an ‘eye candy feature’. Whether the implementation of a numerical algorithm is correct can only be checked by a human familiar with the mathematics. For those people an expressive notation in the implementation serves as a tool to ease this task. However, quality of scientific software is only then increased if there is no negative impact on performance. Maintaining performance in turn increases the complexity of a library tremendously. This not only endangers the extensibility of a library but also makes it hard to guaranty its correctness. It turns out that these restrictions can not be fulfilled trivially. Existing approaches that attempt to overcome this problems are introduced in the following.

Performance Issues

Supporting overloaded operators for linear algebra operations can easily lead to serious performance penalties. In order to illustrate this it is already sufficient to consider an expression like $y = A^T x$. In an implementation this can efficiently be coded using an appropriate `mv` function. For A being a general matrix and using $\alpha = 1, \beta = 0$ this would result in

```
mv(Trans, 1, A, x, 0, y);
```

The underlying implementation of `mv` neither has to create any temporary objects nor setup the transpose A^T explicitly. While this should be an obvious requirement for the evaluation it is not easily met when overloaded operators are used.

In order to allow an expressive notation like

```
y = transpose(A)*x;
```

a function `transpose` and an overloaded multiplication operator has to be implemented. Naive implementations perform the underlying operation immediately and return the results:

```
const Matrix
transpose(const Matrix &A)
{
    // create new matrix for result
    // assign transpose of A to result
    // return result
}

const Vector
operator*(const Matrix &A, const Vector &x)
{
    // create new vector for result
    // compute matrix-vector product
    // return result
}
```

This would result in an evaluation of the expression that is equivalent to

```
t1 = transpose(A);
t2 = t1 * x;
y  = t2;
```

creating two temporary objects `t1` and `t2`. Setting up the transpose explicitly is hereby the most crucial part.

Closures for Linear Algebra Expressions

The performance issue arising from overloaded operators was addressed by Stroustrup in [75]. The suggested solution makes it possible that the above example can be evaluated by a single call of the `mv` function without additional runtime overhead.

The basic idea was to adopt the concept of *closures* known from functional programming. A closure object contains all information needed such that an underlying operation can be performed at a later point in time. Applying this, functions and operators for linear algebra operations return such closure objects. If operands are itself closures this leads to *composite closures*. Evaluation of closures gets delayed until their assignment to the left-hand side.

For the above example the `transpose` function is called first. It receives a matrix object and returns a closure of type `TransMatrix` keeping a reference to the operand:

```
struct TransMatrix
{
    TransMatrix(const Matrix &A) : _A(A) {}

    const Matrix &_A;
};

```

```
const TransMatrix
transpose(const Matrix &A)
{
    return TransMatrix(A);
}
```

Next the transpose-closure gets multiplied by a vector object. The resulting composite closure is of type `TransMatrixMultVector` with references to the matrix and vector operand:

```
struct TransMatrixMultVector
{
    TransMatrixMultVector(const TransMatrix &At, const Vector &x)
     : _At(At._A), _x(x)
    {}

    const Matrix &_At;
    const Vector &_x;
};

const TransMatrixMultVector
operator*(const TransMatrix &At, const Vector &x)
{
    return TransMatrixMultVector(At, x);
}
```

The assignment operator of the vector class has to accept this composite closure as a right hand side argument. Its implementation can then call function `mv` to trigger evaluation of the closure:

```
Vector &
Vector::operator=(const TransMatrixMultVector &AtX)
{
    mv(Trans, 1, AtX._At._A, AtX._x, 0, *this);
    return *this;
}
```

For exemplification, the evaluation of `y=transpose(A)*x` results in

```
y.operator=(TransMatrixMultVector(TransMatrix(A),x));
```

and can be resolved by a compiler through inlining into a direct call of

```
mv(Trans, 1, A, x, 0, y);
```

Hence there is no inherent runtime overhead. It is notable that inlining can only be performed due to the fact that closure types directly reflect the type of linear algebra operation and its operands.

While this technique shows that efficiency and expressive notations not necessarily exclude each other it is far from being a feasible method for practical usage. With the number of supported operations and combinations of operations, the number of required closure classes would explode.

Expression Templates

The template programming technique known as expression templates was developed independently by Todd Veldhuizen [81] and David Vandevoorde [78]. Motivation again was avoidance of temporary objects as well as minimization of memory read and write accesses during the evaluation of linear algebra expressions.

The linear combination of vectors $r = \alpha x + \beta y + \gamma z$ is an ideal example for the efficiency of the expression template technique. Overloaded operators can be used to code the expression by

```
r = alpha*x + beta*y + gamma*z
```

while behind the scene evaluation happens iteratively in a single loop. Thereby each vector element gets accessed only once, either for a memory read or write access:

```
for (int i = 1; i<=n; ++i) {
    r(i) = alpha*x(i) + beta*y(i) + gamma*z(i);
}
```

The technique requires that interface and implementation of vectors are separated following the bridge pattern. For the sake of simplicity, engines considered here are only supposed to provide methods for element access and vector length.

Again, evaluation of the closure is triggered in the assignment operator[3] by accessing elements of the right-hand side (as shown below in more detail):

```
template <typename Engine>
class Vector
{
    public:
        //...

        template <typename RHS>
        Vector<Engine>
        operator=(const Vector<RHS> &rhs)
        {
            for (int i=1; i<=rhs.length(); ++i) {
                engine()(i) = rhs(i);
            }
            return *this;
        }

        //...
};
```

Closures are realized through additional engine types[4] and the above example leads to closure classes `AddClosure` and `ScaleClosure`. `AddClosure` encapsulates the addition of two vectors whereas `ScaleClosure` encapsulates the multiplication of a vector with a scalar value. As before closure objects in general keep references to their operands. Call operator `()` used for element access returns the result of the underlying operation:

```
template <typename L, typename R>
class AddClosure
{
    public:
        AddClosure(const L &left, const R &right)
            : _left(left), _right(right)
        {
            assert(left.length()==right.length());
        }

        int
        length() const
        {
            return _left.length();
        }

        double
        operator()(int i) const
        {
            return _left(i) + _right(i);
        }

    private:
        const L &_left;
        const R &_right;
};
```

[3]To be precise, the assignment operator for right-hand side vectors running a different engine.

[4]As a closure can not be an *l-value* only const methods have to be implemented.

The `ScaleClosure` class can be implemented almost analogously, but with one notable difference. Storing const references to C/C++ built-in types would be inefficient [54]. Hence the closure stores a copy of the scaling factor:

```
template <typename S, typename V>
class ScaleClosure
{
    public:
        ScaleClosure(S scalar, const V &vector)
            : _scalar(scalar), _vector(vector)
        {
        }

        int
        length() const
        {
            return _vector.length();
        }

        double
        operator()(int i) const
        {
            return _scalar * _vector(i);
        }

    private:
        S         _scalar;   // keep copy of scaling factor
        const V &_vector;
};
```

As closures are regular vector engines composite closures do not require any extra treatment. In fact creation of composite closures happens automatically through overloaded operators:

```
template <typename L, typename R>
Vector<AddClosure<L, R> >
operator+(const Vector<L> &l, const Vector<R> &r)
{
    return AddClosure<L, R>(l.engine(), r.engine());
}

template <typename S, typename V>
Vector<ScaleClosure<S, V> >
operator*(S scalar, const Vector<V> &vector)
{
    return ScaleClosure<S, V>(scalar, vector.engine());
}
```

For illustration one can consider sequentially how for `r = alpha*x + beta*y + gamma*z` the overloaded operators create a composite closure of the right-hand side. For vectors `x`, `y` and `z` using class `Array<double>` as engine and scalar values of type `double` the single steps are:

1. `operator*()` returns for `alpha*x`, `beta*y` and `gamma*z` respectively vector closures of type

   ```
   Vector<
       ScaleClosure<double, Array<double> >
       >
   ```

2. `operator+()` returns for `alpha*x + beta*y` a vector closure of type

   ```
   Vector<
       AddClosure<
           ScaleClosure<double, Array<double> >,
           ScaleClosure<double, Array<double> >
           >
       >
   ```

3. `operator+()` returns for `alpha*x + beta*y + gamma*z` a vector closure of type

```
Vector<
    AddClosure<
        AddClosure<
            ScaleClosure<double, Array<double> >,
            ScaleClosure<double, Array<double> >
            >,
        ScaleClosure<double, Array<double> >
        >
    >
```

Obviously type names like these could scare many people to death. It is therefore notable to point out again that these types are created automatically and only the compiler actually has to deal with them.

Advantages and Limitations of Expression Templates

Expression templates make the handling of closures and in particular of composite closures feasible. For each elementary type of operation only one closure has to be implemented. In some cases this technique can even outperform FORTRAN code [86].

However a few issues remain when using expression templates in a full featured numerical library:

1. In certain cases expression templates can beat BLAS routines in terms of performance. A prime example is the computation vector sums

   $$y = x_1 + \cdots + x_n$$

   where n is sufficiently large. With expression templates computation is performed through a single loop. Using BLAS routines the computation needs to be performed by adding the vectors successively one by one:

   $$y \leftarrow x_1,\; y \leftarrow y + x_2,\; \ldots,\; y \leftarrow y + x_n.$$

   In both cases temporary objects are avoided. But in the latter case the number of memory accesses is disproportionally higher[5].

   However, practical applications require that expressions occurring most frequently are evaluated most efficiently. In many numerical algorithms these expression can be evaluated by a single or at most a few BLAS routines. In this case, the BLAS routine also provides minimal memory access. In addition, BLAS implementations that exploit specific hardware features are already available. Competing with BLAS routines in this field seems not to promise any real pay-off. In the best case the same performance can be reached.

2. Expression templates permit a rather easy method to treat sums and linear combinations of vectors and matrices. However it requires quite some effort to treat matrix-vector or matrix-matrix products:

   In order to treat expressions like in `y = A*x` where `A` is a general matrix a vector closure engine can be implemented conforming to the above interface: the i-th element of the closure is the dot product of the i-th row of `A` with `x`. Length of the closure equals the number of rows of `A`.

   However, this method would fail for `x = A*x` as here `x` also occurs on the left-hand side. In this case the result of `A*x` has to be stored first in a temporary vector `t`. Afterwards `x` can be overwritten by `t`.

[5]With expression templates there will be $n \cdot \dim(y)$ memory read and $\dim(y)$ write accesses. Usage of BLAS routines will result in $n \cdot \dim(y)$ memory read but also $n \cdot \dim(y)$ write accesses.

3. Numerical methods for solving PDEs require solving systems of linear equations where the matrices are of particular structure. In order to solve these certain iterative solvers can provide optimal convergence rates. For the actual runtime performance of these iterative solvers the efficient implementation of the matrix-vector product is crucial.

   In some cases the integration of new, user-defined matrix or vector types can be a real challenge. If the new types require change of the engine interface, all existing engine implementations are affected - including the closure engines. Then supporting an operation for a new type requires that the developer takes the complete expression template mechanism into account. This means implementing an operation like the matrix-vector product for a new type can not be added to the library in a selective and independent manner. Already supporting addition of sparse vector efficiently would require modification of the closure interface. This is necessary as for sparse vectors it only makes sense to iterate over non-zero elements. With increasing diversity of matrix or vector types the interface design and implementation of closure engines becomes a more and more complex task.

The issue that implementations of closures become more and more complicated when the number of supported matrix and vectors types gets increased is addressed in the FLENS approach.

4.3.3 Overloading Operators: FLENS Approach

FLENS combines certain aspects of the ideas illustrated in Section 4.3.2. Composite closures are created automatically as this was the case for expression templates. Evaluation of closures is performed using the BLAS wrappers introduced in Section 4.3.1. The evaluation mechanism is both transparent and efficient. In particular, the FLENS approach provides a simple mechanism to add and integrate new matrix and vector types seamlessly into a numerical application framework.

Closures

Closures are needed to delay evaluation of linear algebra operations that result in either a matrix and vector type. Therefore it makes sense to distinguish between *matrix closures* and *vector closures*. Concepts for both types of closures correspond to each other such that in the following only vector closures are considered. Class `VectorClosure` receives three template parameters:

```
template <typename Op, typename L, typename R>
    class VectorClosure;
```

The first parameter specifies the type of operation, the remaining the operand types. Hence the vector closure type contains all static information about the encapsulated delayed operation. Types for linear algebra operations are defined through empty structs:

```
struct OpAdd  {};   // +
struct OpSub  {};   // -
struct OpMult {};   // *
// ...
```

That means for the matrix-vector product Ax the closure gets instantiated with `OpMult` for `Op` and types of A and x for `L` and `R` respectively.

Vector closures are derived from the vector base class. This reflects the fact that evaluation of vector closures results in a particular vector type. Therefore such a closure itself can be regarded as a vector type. Furthermore this will allow automatic creation and eases later treatment of closures.

```
template <typename Op, typename L, typename R>
class VectorClosure
    : public Vector<VectorClosure<Op, L, R> >
{
    public:
        VectorClosure(typename Ref<L>::Type l,
                      typename Ref<R>::Type r);

        typename Ref<L>::Type  left() const;

        typename Ref<R>::Type  right() const;

    private:
        typename Ref<L>::Type _left;
        typename Ref<R>::Type _right;
};
```

Similar to the expression templates example, operands are referenced unless they are built-in types. As also suggested in [78] such a distinction can be realized through a trait class, here denoted as `Ref`. By default

```
typename Ref<L>::Type _left;
```

is equivalent to

```
const L &_left;
```

For operand types that should be copied (e. g. built-in types like `float`, `double`, . . .) specializations of `Ref` are used to achieve the expansion

```
        L  _left;
```

Deriving the vector closure from base class `Vector` further requires the definition of an appropriate `TypeInfo` trait. Hereby only defining the element type might be not trivial:

```
template <typename Op, typename L, typename R>
struct TypeInfo<VectorClosure<Op, L, R> >
{
    typedef VectorClosure<Op, L, R> Impl;
    typedef typename Promotion<typename L::ElementType,
                               typename R::ElementType>::Type ElementType;
};
```

If both operands have the same element type then (by default) this is also the element type of the result. FLENS allows operands to have different element types. Consider the sum of two vectors where one vector has elements of type `double` and the other of type `float`. Through the `Promotion` trait it is possible to define that the result has elements of type `double`. Examples on how to support mixed element types using FLENS are presented in Chapter 5.

Overloading Operators

All operators and functions for linear algebra operations that result in either a matrix or vector can now be overloaded following the same pattern:

1. The return type is a const closure. Depending on the operation this is either a matrix or vector closure.
2. Matrix and vector operands are of type `Matrix` and `Vector` respectively. This means all matrix and vector types integrated in the FLENS hierarchy are automatically accepted.

For the sum of two vectors this leads to the declaration of

```
// x + y
template <typename V1, typename V2>
    const VectorClosure<OpAdd, typename V1::Impl, typename V2::Impl>
    operator+(const Vector<V1> &x, const Vector<V2> &y);
```

Its implementation just instantiates and returns the corresponding vector closure

```
    typedef VectorClosure<OpAdd, typename V1::Impl, typename V2::Impl> VC;
    return VC(x.impl(), y.impl());
```

Like in the expression template approach, composite closures are created automatically. Considering the sum of vectors `r = x + y + z` (all of type `DenseVector<Array<double> >`) the right-hand side sequentially results in a composite closure as follows:

1. `operator+()` returns for `x + y` a vector closures of type

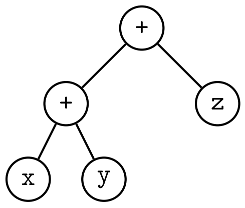
```
VectorClosure<OpAdd,
              DenseVector<Array<double> >,
              DenseVector<Array<double> >
             >
```
(4.2)

2. `operator+()` returns for `x + y + z` a vector closure of type

```
VectorClosure<OpAdd,
              VectorClosure<OpAdd,
                            DenseVector<Array<double> >,
                            DenseVector<Array<double> >
                           >,
              DenseVector<Array<double> >
             >
```
(4.3)

Denoting by `rhs` the latter closure object, operands `x`, `y` and `z` can be retrieved by 'traversing' the composite closure:

`x` is reference by `rhs.left().left()`,
`y` is reference by `rhs.left().right()`,
`z` is reference by `rhs.right()`.



This motivates to represent closures as trees. In this context such trees are generally denoted as *expression trees.*

Closure Evaluation: The Basic Idea

Again evaluation of the closure has to be triggered through the assignment operator defined in the vector class. This could now be realized analogously to the Stroustrup's example shown in Section 4.3.2: an assignment operator in the `DenseVector` dedicated to the vector closure type (4.3) can access each summand and evaluate the closure using the BLAS wrappers

```
copy(x, r);       // r  = x
axpy(1, y, r);    // r += y
axpy(1, z, r);    // r += z
```
(4.4)

Hence the integration of closures into the FLENS matrix or vector class hierarchy, the evaluation of closures can be handled more flexible. Instead of several assignment operators dedicated to certain closure types only the implementation of one assignment operator is required. This operator expects an arbitrary vector type (which here in particular includes vector closures) and delegates assignment to a copy function:

```
template <typename Engine>
    template <typename Imp>
DenseVector<Engine> &
DenseVector<Engine>::operator=(const Vector<Imp> &y)
{
    copy(y.impl(), *this);          // i.e. *this = y.impl()
    return *this;
}
```

The copy function receives as first argument `y.impl()`, i. e. the vector closure for `x+y+z` whose type is given in (4.3). Hence BLAS wrappers are extended for an overloaded version of `copy` that can handle such a vector closure[6]:

[6]Note that this function is implemented independent of class `DenseVector` and therefore allows for the use of any kind of vector implementation.

```
template <typename VL, typename VR, typename VY>
void
copy(const VectorClosure<OpAdd, VL, VR> &x, Vector<VY> &y)
{
    copy(x.left(), y.impl());
    axpy(1, x.right(), y.impl());
}
```

Evaluation of the closure happens recursively. For `r = x + y + z` the resulting function calls can be depicted in stack form as

$$\begin{array}{l}
\texttt{r.operator=(x+y+z, r);}\\
\qquad\texttt{copy(x+y+z, r);}\\
\qquad\qquad\texttt{copy(x+y, r);}\\
\qquad\qquad\qquad\underline{\texttt{copy}}\texttt{(x, r);}\\
\qquad\qquad\qquad\underline{\texttt{axpy}}\texttt{(1, y, r);}\\
\qquad\qquad\underline{\texttt{axpy}}\texttt{(1, z, r);}
\end{array}$$

Functions that perform the actual computations are underlined. Obviously evaluation is equivalent to and after inlining even identical to (4.4). Benchmarks on the FLENS website [45] demonstrate that indeed no runtime overhead gets induced. Moreover, for current compilers typically only very moderate optimization flags (e. g. `-DNDEBUG` and `-O3`) are sufficient.

Closure Evaluation: FLENS Policy

The sole reason to use overloaded operators in a numerical library has to be enhanced expressiveness of numerical code. This has to be achieved while obeying the constrains:

1. No inherent run-time overhead compared to direct usage of the BLAS wrappers.
2. Not misleading to produce inefficient code and in particular side-effects that create temporary objects.
3. The usage of overloaded operators makes it easier for outsiders to understand the mathematical functionality. But for an insider, i. e. the programmer it has to be comprehensible as to what exactly is going on behind the scene. That means it has to be absolutely transparent which functions are used, in which order and with which parameters for the evaluation of a linear algebra expression. In other words, there has to be a clear one-to-one mapping between notation and the underlying evaluation mechanism.

An example of how a library providing overloaded operators can mislead a user to produce inefficient code is an expression like

```
q = R*(x - y);
```

If a library in general allows such an expression the internal evaluation needs to be either equivalent to

```
t = x - y;
q = R*t;
```

or to

```
q = R*x
q -= R*y;
```

In the first case this means that the evaluation requires the creation of a temporary object. This would be a side-effect not visible to the user. Consider that the expression gets evaluated inside a loop such that in each iteration a temporary object gets created. Then a user wants to store the intermediate result of `b - A*x` in a vector allocated only once outside the loop:

```
// allocate r
while (...) {
    r = b - A*x;
    q = R*r;
}
```

In the second case the evaluation does not conform numerically to the expression[7] `q = R*(b - A*x)`. Such an evaluation would not only lead to numerically incorrect results but also would realize an extremely obscure evaluation mechanism.

The way FLENS evaluates closures makes it absolutely transparent as to how expressions are evaluated. It also allows for the prevention of unintended creation of temporary objects. Expressions like `R*(b - A*x)` in general lead to an error at compile time but can be supported if the user provides a function for the evaluation of the corresponding closure.

Closure Evaluation: The General Realization in FLENS

Reviewing again the above example for the sum of three vectors will illustrate the general concept realized in FLENS. The basic idea was to use:

1. BLAS wrappers for `copy` and `axpy` from Section 4.3.1 for the actual computations and
2. an additional version of `copy` (dedicated to a certain type of vector closure) controlling how composite closures get 'decomposed' and evaluated.

BLAS wrappers handling matrix and vector closures are provided by the FLENS library. These wrappers rely on BLAS wrappers for concrete matrix and vector types that conform to the same functionality and naming scheme of BLAS. Table 4.2 enlists wrapper functions for scaling and adding vectors as well as carrying out matrix-vector products. For operations that can be carried

BLAS wrapper	signature	functionality
scal	`scal(alpha, x)`	$y \leftarrow \alpha\, x$
copy	`copy(x, y)`	$y \leftarrow x$
axpy	`axpy(alpha, x, y)`	$y \leftarrow x$
mv	`mv(trans, alpha, A, x, beta, y)`	$y \leftarrow \alpha\, \text{op}(A)x + \beta y$ $\text{op}(A) = A, A^T$ or A^H

Table 4.2: Extract of wrapper functions for BLAS level 1 and 2.

out by BLAS routines FLENS already provides such wrapper functions. For new user-defined matrix or vector types, this set can easily be extended. To be more explicit, only this set needs to be extended. The whole mechanism of overloaded operators and handling closures already provided by FLENS remains unaffected by new matrix or vector types.

4.4 LAPACK Interface

As for BLAS routines, wrappers for LAPACK can provide a much more convenient interface. First of all less parameters are required as details about storage schemes are encapsulated in the FLENS matrix types. Further it allows a uniform interface for routines implementing the same numerical method optimized for different matrix types. Listing 4.2 shows LAPACK wrappers for the *LU* factorization of general matrices.

```
template <typename FS>
int
trf(GeMatrix<FS> &A, DenseVector<Array<int> > &P)
{
```

[7]Note that for floating point numbers `a*(b+c)` in general is not equal to `a*b + a*c`!

```
    return getrf(A.numRows(), A.numCols(),
                 A.data(), A.leadingDimension(),
                 P.data());
}

template <typename BS>
int
trf(GbMatrix<BS> &A, DenseVector<Array<int> > &P)
{
    return gbtrf(A.numRows(), A.numCols(),
                 A.numSubDiags(), A.numSuperDiags()-A.numSubDiags(),
                 A.data(), A.leadingDimension(),
                 P.data());
}
```

Listing 4.2: LAPACK wrappers for the *LU* factorization of general matrices.

For general matrices based on full storage and band storage schemes the factorization gets carried out by functions `getrf` and `gbtrf` respectively. Note that the low-level LAPACK wrappers introduced in Section 3.4 overloaded these functions with respect to the matrix element type.

4.5 Discussion on Software Quality: Improvements and Deficiencies

This chapter illustrated fundamental concepts realized in the design and implementation of FLENS. Several techniques were applied to combine flexibility, extensibility and ease of use without sacrificing efficiency. This was possible because of a thoroughly orthogonal design and a proper combination of techniques.

Virtual functions in C++ can lead to a considerable runtime overhead [26]. This was one reason why C++ has gained a bad reputation within the high-performance computing community. In FLENS the *Curiously Recurring Template Pattern (CRTP)* was used and adapted to realize multilevel class hierarchies for matrix and vector types. One advantage of applying the CRTP is the avoidance of virtual functions. It was illustrated how the *bridge pattern* can be used to integrate matrix and vector types for BLAS and LAPACK into the FLENS class hierarchy. It was further shown how the CRTP based class hierarchy can be combined with concepts like *closures* and *expression templates* to allow a convenient interface for BLAS and LAPACK. In particular, overloaded operators can be used to provide an expressive and intuitive notation for linear algebra operations. It was hereby demonstrated how the ease of use can be improved dramatically without any negative impact on performance (cp. the benchmarks on [45]).

The expressiveness provided by FLENS enhances the correctness of a numerical implementation. Furthermore, because of the high degree of abstraction it eases the possibility of implementing reusable numerical methods. In this chapter features and concepts of FLENS were mainly exemplified for BLAS and LAPACK. However, FLENS is more than just a high-level C++ interface for these libraries. FLENS provides a simple and transparent mechanism for extending the set of matrix and vector types as well as the set of supported linear algebra operations. All these features will be illustrated in more detail in the following chapters.

As was shown in Section 4.3.3, several efforts were realized in FLENS in order to prevent obscure side-effects like the unintended creation of temporary objects. For example, a linear algebra expression like `x = A*x` gets rejected by FLENS if the evaluation would require the implicit creation of a temporary object. For programmers who are inexperienced in scientific computing this might seem to be an immoderate restriction at first. Nevertheless, for the serious development of scientific software the transparency of an implementation is crucial.

5 Realizing Reusable Numerical Algorithms

This chapter briefly illustrates various techniques for extending the FLENS library and realizing reusable implementations of numerical linear algebra algorithms. On the one hand, reusability implies that the same implementation can be used to treat different — but in general similar — types of problems. On the other hand, reusability in the context of scientific software implies that performance optimizations can be applied afterwards, in a way that does not require to modify the original implementation. For the sake of simplicity and compactness, all these different topics are covered exemplary in this chapter.

Section 5.1 illustrates how to extend FLENS for a new matrix type. This extension also includes the support of linear algebra operations that are not part of standard BLAS implementations. Section 5.2 briefly outlines how to support element types of matrices and vector that are of arbitrary precision. Using the conjugated gradient method as an example, Sections 5.3 and 5.2 demonstrate the capabilities provided by FLENS for realizing reusable numerical software components. How optimizations can be applied orthogonal to existing implementations is shown in Section 5.5.

5.1 User Defined Matrix and Vector Types

As a simple example for a new matrix type consider diagonal matrices. The code snippet in Listing 5.1 illustrates some typical functionality that a user might expect from a corresponding implementation: access to diagonal elements for initialization (lines 8-10) and computation of matrix-vector products (lines 12 and 13).

```
 1 int n=10;
 2
 3 DiagonalMatrix<double>       D(n);
 4 DenseVector<Array<double> > x(n), y(n), r;
 5
 6 for (int i=1; i<=n; ++i) {
 7     D(i) = i;
 8     x(i) = n-i;
 9 }
10
11 y = D*x;
12 r = y - D*x;
```

Listing 5.1: Example for the usage of a diagonal matrix type.

As described in Section 4.1, the integration of the new matrix type into the FLENS hierarchy requires defining a corresponding `TypeInfo` trait:

```
template <typename T>
struct DiagonalMatrix;

template <typename T>
struct TypeInfo<DiagonalMatrix<T> >
```

```
{
    typedef DiagonalMatrix<T>  Impl;
    typedef T                  ElementType;
};
```

Objects of type `DiagonalMatrix` are derived from `SymmetricMatrix` and internally store the diagonal elements in a vector of type `DenseVector`. Constructors, operators for element access and method `dim` delegate the functionality to the encapsulated vector:

```
template <typename T>
struct DiagonalMatrix
    : public SymmetricMatrix<DiagonalMatrix<T> >
{
    DenseVector<Array<T> > d;

    DiagonalMatrix(int n)
        : d(n)
    {
    }

    int
    dim() const
    {
        return d.length();
    }

    const T &
    operator()(int index) const
    {
        return d(index);
    }

    T &
    operator()(int index)
    {
        return d(index);
    }
};
```

What remains to get the code in Listing 5.1 to work is an implementation of the matrix-vector product. Recall that no operators need to be overloaded and that the creation and treatment of closures is already handled by FLENS as described in Section 4.3. Merely an appropriate specialization of an `mv` function needs to be implemented which is capable of multiplying a diagonal matrix with a dense vector[1]:

```
template <typename ALPHA, typename MD, typename VX, typename BETA, typename VY>
void
mv(ALPHA alpha, const DiagonalMatrix<MD> &D, const DenseVector<VX> &x,
   BETA beta, DenseVector<VY> &y)
{
    assert(D.dim()==x.length());
    assert(D.dim()==y.length());
    assert(x.firstIndex()==1);
    assert(y.firstIndex()==1);

    for (int i=1; i<=D.dim(); ++i) {
        y(i) = D(i)*x(i);
    }
}
```

The above example demonstrates a basic philosophy of FLENS: 'only implement what is needed'. This allows for the complexity of an implementation to remain proportional to the complexity of the addressed problem. However, if a user requires in addition a matrix-matrix product then

[1]Obviously it would be easy to further optimizes this implementation and make it more flexible.

this can be easily added. This in turn guarantees the extensibility of existing implementations. Further, optimizations of the above matrix-vector product can be applied without deep knowledge of FLENS[2].

5.2 Supporting Element Types with Arbitrary Precision

GMP[34] is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers and floating point numbers. Data types from GMP can be used as element types for matrices and vectors. However, the standard BLAS and LAPACK implementations only[3] support the built-in types `float`, `double`, `complex<float>` and `complex<double>`. Fortunately the C++ interface of the GMP library already provides overloaded operators for arithmetic operations. This allows a simple generic implementation of BLAS and LAPACK, e. g. :

```
template <typename ALPHA, typename X, typename Y>
void
axpy(ALPHA alpha, const DenseVector<X> &x, DenseVector<Y> &y)
{
    // assertions
    for (int i=x.firstIndex(), I=y.firstIndex(); i<=x.lastIndex(); ++i) {
        y(I) += alpha*x(I);
    }
}
```

5.3 The Conjugated Gradient (cg) Method

The cg-method is an iterative method to solve a system of linear equations

$$Ax = b, \quad A \in \mathbb{R}^{n\times n},\ x, b \in \mathbb{R}^n \tag{5.1}$$

where the coefficient matrix A is symmetric positive definite. Using an initial guess $x^{(0)}$ the cg-method produces iterates

$$x^{(i)} \in x^{(0)} + K_i(b - Ax^{(0)}, A), \quad K_i(r, A) := \operatorname{span}\{r, Ar, \dots, Ar^{(i-1)}\}$$

such that $\left(x^{(i)} - x^*\right)^T A \left(x^{(i)} - x^*\right)$ is minimized where x^* denotes the exact solution. The subspace $K_i(r, A)$ of $\mathbb{R}^n$ is denoted as *Krylov subspace* of matrix A for the initial residual r[4]. Pseudocode for the cg-method is given in Algorithm 5.1 (cp. [74]).

Algorithm 5.1 (Conjugated Gradient method).

```
Start: Choose initial guess $x^{(0)} \in \mathbb{R}^n$ and set $p^{(0)} := r^{(0)} := b - Ax^{(0)}$
For $k = 0, 1, \dots$
    If $p^{(k)} = 0$:
        $x^{(k)}$ is solution of $Ax = b$. Stop.
    Otherwise compute
        $\alpha := \frac{\left(r^{(k)}\right)^T r^{(k)}}{\left(p^{(k)}\right)^T A p^{(k)}}$
```

[2]The above implementation of `mv` could internally call a function from an external library.

[3]Commercial implementations like for instance the *Intel Math Kernel Library* actually support arbitrary precision and interval arithmetic.

[4]The cg-method is commonly denoted as *Krylov method*, reflecting the type of subspaces considered to produce the iterates. The cg-method is also denoted as a *non-stationary method*, reflecting the fact that information used to produce an iterate changes in each iteration.

$$
\begin{aligned}
&x^{(k+1)} := x^{(k)} + \alpha p^{(k)}\\
&r^{(k+1)} := r^{(k)} - \alpha_k A p^{(k)}\\
&\beta_k := \frac{\left(r^{(k+1)}\right)^T r^{(k+1)}}{\left(r^{(k)}\right)^T r^{(k)}}\\
&p^{(k+1)} := r^{(k+1)} + \beta_k p^{(k)}
\end{aligned}
$$

It can be shown that Algorithm 5.1 produces after, at most n iterations the exact solution x^*. However, due to round off errors in general more iterations are required in an implementation. A typical stopping criterion is to require that $(r^{(k)})^T r^{(k)}$ has fallen below a given threshold. For the cg-method (cp. [74], [21]) the error estimate

$$||x^{(k)} - x^*||_A \leq 2 \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right)^k ||x^{(0)} - x^*||_A \tag{5.2}$$

holds, where $\kappa = \kappa(A)_2 = ||A||_2 ||A^{-1}||_2$ denotes the condition of A and $||x||_A = \sqrt{x^T A x}$ the energy norm. As can be seen from (5.2) the rate of convergence depends on the condition of A. As outlined in the following section, the rate of convergence can be increased by preconditioning.

Listing 5.2 shows an implementation of the cg-method in FLENS. Due to the use of overloaded operators the core of the implementation (lines 9-26) directly reflects the above pseudo-code. The `cg_` trait used in lines 5 and 6 for the local definition of vectors and scalar valued variables is explained in more detail below. The most expressive operation in each iteration of the cg-method is the computation of $Ap^{(k)}$. As this matrix-vector product is needed twice (line 17 and 19) the result gets stored in vector `Ap` (line 16).

```
 1 template <typename MA, typename VX, typename VB>
 2 int
 3 cg(const MA &A, VX &x, const VB &b, double tol, long maxIterations)
 4 {
 5     typename cg_<VB>::T           alpha, beta,
 6                                   rNormSquare, rNormSquarePrev;
 7     typename cg_<VB>::AuxVector   Ap, r, p;
 8
 9     r = b - A*x;
10     p = r;
11     rNormSquare = r*r;
12     for (long k=1; k<=maxIterations; ++k) {
13         if (rNormSquare<=tol) {
14             return k-1;
15         }
16         Ap = A*p;
17         alpha = rNormSquare/(p * Ap);
18         x += alpha*p;
19         r -= alpha*Ap;
20
21         rNormSquarePrev = rNormSquare;
22         rNormSquare = r*r;
23         beta = rNormSquare/rNormSquarePrev;
24         p = beta*p + r;
25     }
26     return maxIterations;
27 }
```

Listing 5.2: Generic implementation of the cg-method.

In most cases the vector type of `b` can be used for local vectors[5] and its element type for scalar valued variables. This is the default behavior of the `cg_` trait:

```
28 template <typename A>
29 struct cg_
```

[5]Note that this is not the case for the type of `x`. If `x` is a sparse vector the product `A*x` in general will be a dense vector.

```
{
    typedef A AuxVector;
    typedef typename A::ElementType T;
};
```

However, if b is a vector view, its type is not suitable. This case is treated by the specialization:

```
template <typename I>
struct cg_<DenseVector<I> >
{
    typedef typename DenseVector<I>::NoView AuxVector;
    typedef typename DenseVector<I>::ElementType T;
};
```

In order to apply the cg-method on user defined matrix or vector types the implementation of corresponding BLAS functions is required (cp. Section 5.1). This reflects a general concept: each generic implementation of an iterative method depends on a certain set of BLAS functions. Further this shows that performance tuning of an iterative method can be directly achieved by tuning the underlying BLAS implementation.

5.4 The Preconditioned Conjugated Gradient Method

The fact that the convergence rate depends on the condition of the coefficient matrix is exploited by the *preconditioned conjugated gradient (pcg) method* (see e. g. [74]). Assume that a symmetric positive definite matrix B exists approximating A^{-1} such that $BA \approx I$ (and in particular $\kappa(BA)_2 \ll \kappa(A)_2$). As matrix

$$\tilde{A} := B^{-\frac{1}{2}}(BA)B^{\frac{1}{2}} = B^{\frac{1}{2}}AB^{\frac{1}{2}}$$

results from a similarity transform, it follows that $\kappa(\tilde{A})_2 = \kappa(BA)_2 \ll \kappa(A)_2$. If $\tilde{x}^*$ solves

$$\tilde{A}\tilde{x} = \tilde{b}, \quad \tilde{b} := B^{\frac{1}{2}}b \tag{5.3}$$

then $x^* = B^{\frac{1}{2}}\tilde{x}^*$ solves (5.1). Applying the cg-method to (5.3) leads to Algorithm 5.2 which is adapted from [74]. The algorithm does not require to setup $\tilde{A}$ and $\tilde{b}$ explicitly and computes the solution of (5.1). Compared to Algorithm 5.1 each iteration requires an additional matrix-vector multiplication with the preconditioner B.

Algorithm 5.2 (Preconditioned Conjugated Gradient method).

```
Start: Choose initial guess $x^{(0)} \in \mathbb{R}^n$ and set $r^{(0)} := b - Ax^{(0)}$, $p^{(0)} := Br^{(0)}$
For $k = 0, 1, \dots$
    If $p^{(k)} = 0$:    $x^{(k)}$ is solution of $Ax = b$. Stop.
    Otherwise compute
        $\alpha := \frac{\left(r^{(k)}\right)^T q^{(k)}}{\left(p^{(k)}\right)^T Ap^{(k)}}$
        $x^{(k+1)} := x^{(k)} + \alpha_k p^{(k)}$
        $r^{(k+1)} := r^{(k)} - \alpha_k Ap^{(k)}$
        $q^{(k+1)} := Br^{(k+1)}$
        $\beta_k := \frac{\left(r^{(k+1)}\right)^T q^{(k+1)}}{\left(r^{(k)}\right)^T q^{(k)}}$
        $p^{(k+1)} := q^{(k+1)} + \beta_k p^{(k)}$
```

The generic implementation of the pcg-method (Listing 5.3) has an additional template parameter `Prec` for the preconditioner `B`. Beside this, the implementation follows the same pattern that was used for the cg-method: using the same trait for local variables (lines 6-7) and storing the matrix-vector product in a local vector (line 19).

```
 1 template <typename Prec, typename MA, typename VX, typename VB>
 2 int
 3 pcg(const Prec &B, const MA &A, VX &x, const VB &b,
 4     double tol, long maxIterations)
 5 {
 6     typename cg_<VB>::T pNormSquare, alpha, beta, rq, rqPrev;
 7     typename cg_<VB>::AuxVector r, q, p, Ap;
 8
 9     r = b - A*x;
10     q = B*r;
11     p = q;
12
13     rq = r*q;
14     for (long k=1; k<=maxIterations; ++k) {
15         pNormSquare = p*p;
16         if (pNormSquare<=tol) {
17             return k-1;
18         }
19         Ap = A*p;
20         alpha = rq/(p*Ap);
21         x += alpha*p;
22
23         r -= alpha*Ap;
24         q = B*r;
25
26         rqPrev = rq;
27         rq = r*q;
28         beta = rq/rqPrev;
29         p = beta*p + q;
30     }
31     return maxIterations;
32 }
```

Listing 5.3: Generic implementation of the pcg-method.

For the practical application the question remains how to chose a preconditioner $B \approx A^{-1}$. A simple type of preconditioning is the so called *diagonal* or *Jacobi preconditioner*[6] where $B = \operatorname{diag}(A)^{-1}$. A straight forward way to apply the Jacobi preconditioner would be to initialize a diagonal matrix (line 7) with the reciprocals of the diagonal elements (lines 11-13) and pass it to the pcg-method (line 15):

```
 1 typedef SbMatrix<BandStorage<double, ColMajor> > Mat;
 2 typedef DenseVector<Array<double> >             Vec;
 3
 4 int n = 128;
 5 Mat                  A(n, Upper, 1);
 6 Vec                  b(5), x(5);
 7 DiagonalMatrix<Vec> B(n);
 8
 9 // init A, b
10
11 for (int i=1; i<=n; ++i) {
12     B(i) = 1/A(i,i);
13 }
14
15 int it = pcg(B, A, x, b);
```

While in this case setting up the preconditioner matrix explicitly is acceptable, this is not feasible for more sophisticated types of preconditioning. The following alternative to this approach

[6]The denotation 'Jacobi' preconditioner will be motivated in Section 6.7.

demonstrates an implementation concept that will be frequently used in the following chapters. Each type of preconditioner gets realized through a dedicated matrix type, in this case `JacobiPrec`:

```
1 template <typename MA>
2 class JacobiPrec;
3
4 template <typename MA>
5 struct TypeInfo<JacobiPrec<MA> >
6 {
7     typedef JacobiPrec<MA> Impl;
8     typedef double         ElementType;
9 };
```

As can be seen from the above declaration, this new matrix type receives the type of the coefficient matrix A as a template parameter. The preconditioner class itself is derived from `SymmetricMatrix` (line 12) and merely keeps a reference to the coefficient matrix (line 20):

```
10 template <typename MA>
11 class JacobiPrec
12     : public SymmetricMatrix<JacobiPrec<MA> >
13 {
14     public:
15         JacobiPrec(const MA &_A)
16             : A(_A)
17         {
18         }
19
20         const MA  &A;
21 };
```

For the `JacobiPrec` class the matrix-vector product $q = Bp$ can then be implemented as follows:

```
template <typename MA, typename VB, typename VX>
void
mv(double alpha, const JacobiPrec<MA> &B, const DenseVector<VB> &p,
   double beta, DenseVector<VX> &x)
{
    assert(alpha==1.);
    assert(beta==0.);

    for (int i=p.firstIndex(); i<=p.lastIndex(); ++i) {
            p(i) = p(i)/B.A(i,i);
    }
}
```

From a performance perspective it might be more favorable to store the reciprocals of the diagonal elements of A^{-1} in an extra vector. This is due to the fact that floating point divisions are more expensive than multiplications. As in each iteration of the pcg-method a matrix-vector product has to be computed, the additional memory required will most likely pay off. Obviously the above implementation of the `JacobiPrec` and the corresponding `mv` function can easily be adopted to take this into account. However, it is notable that in both cases neither the implementation of the pcg-method nor the usage of the pcg-method gets affected by such performance tweaks:

```
typedef SbMatrix<BandStorage<double, ColMajor> > Mat;
typedef DenseVector<Array<double> >              Vec;

int n = 128;
Mat             A(n, Upper, 1);
Vec             b(5), x(5);
JacobiPrec<Mat> B(A);

// init A, b

int it = pcg(B, A, x, b);
```

5.5 Extending the Set of Basic Linear Algebra Operations

As was mentioned in the end of Section 5.1 performance tuning of iterative methods can typically be achieved by tuning the underlying BLAS implementation. In some cases further improvements can be obtained by extending the set of operations supported by BLAS. Consider the computation of the residual

```
r = b - A*x;
```

In general this computation can be carried out equivalently to

```
r  = b;
r -= A*x;
```

which in turn is equivalent to

```
copy(b, r);
mv(NoTrans, -1, A, x, 1, r);
```

For certain types of matrices it can be much more efficient to compute the residual through a dedicated function, as for instance

```
residual(b, A, x, r);
```

For diagonal matrices and dense vectors a simple but efficient implementation is given through

```
template <typename VB, typename MD, typename VX, typename VR>
  void
  residual(const DenseVector<VB> &b, const DiagonalMatrix<MD> &D,
               const DenseVector<VX> &x,
               DenseVector<VR> &r)
  {
      for (int i=x.firstIndex(); i<=x.lastIndex(); ++i) {
          r(i) = b(i) - D(i)*x(i)
      }
  }
```

Listing 5.4: Special purpose implemetation to compute the residual for a diagonal coefficient matrix and dense vectors.

The mechanism realized in FLENS for linear algebra operations can be extended such that the expression `r = b-A*x` automatically gets evaluated:

1. through a specialized `residual` function if such a function is available and
2. otherwise through `copy` and `mv` functions.

In order to extend FLENS in this respect one has to be familiar with the concept of closures realized in FLENS. This is shown in the remaining section. However, once completed it becomes absolutely invisible to other users. Merely providing an implementation like in Listing 5.4 is required. Recall that in FLENS the expression `b - A*x` results in a closure of type

```
VectorClosure<OpSub,
              VB,
              VectorClosure<OpMult,
                            MA,
                            VX>
             >
```

where `VB`, `MA` and `VX` represent types of `b`, `A` and `x` respectively. This closure gets assigned to vector `r` through a call of a `copy` function.

```
// r = b -A*x
template <typename VB, typename MA, typename VX, typename VR>
void
copy(const VectorClosure<OpSub, VB, VectorClosure<OpMult, MA, VX> > &b_Ax,
     Vector<VR> &r)
{
    residual(b_Ax.left(), b_Ax.right().left(), b_Ax.right().right(), r.impl());
}
```

If a specialized implementation of `residual` exists for the corresponding matrix and vector types this implementation gets called and otherwise a default implementation:

```
template <typename VB, typename MA, typename VX, typename VR>
void
residual(const Vector<VB> &b, const Matrix<MA> &A, const Vector<VX> &x,
         Vector<VR> &r)
{
    typedef typename VR::ElementType T;

    copy(b.impl(), r.impl());
    mv(NoTrans, T(-1), A.impl(), x.impl(), T(1), r.impl());
}
```

All these redirections can be inlined by the compiler such that no run-time overhead gets induced.

6 The Finite Difference Method (FDM) and Poisson Solvers

In this chapter the Dirichlet-Poisson problem in one and two space dimensions serves as a model problem. The problem is only considered on simple domains (the unit interval in one dimension and the unit square in two dimensions) such that an appropriate discretization can be obtained using a simple finite difference approximation. The simplicity of this model problem is intended in order to keep the main focus on the implementation of different numerical solvers for the discretized problem. In this respect the implementation of the multigrid method presented in this chapter is most notable. The very same implementation can be used to solve the discretized problem not only in one or two space dimensions but also either serial or parallel.

Section 6.1 introduces the one-dimensional model problem on the unit interval and its discretization. This section also addresses different issues that are relevant for the later implementation of numerical solvers. Direct solvers are presented in Section 6.2 while stationary iterative solvers are discussed in Section 6.3. The implementation of direct and stationary iterative solvers serve as building blocks for the multigrid method covered in Section 6.4. Having motivated the different types of solvers for one space dimension, the underlying concepts are extended to two dimensions in Section 6.5. Parallelization of the multigrid method is addressed in Section 6.6. Section 6.7 briefly outlines various possibilities and aspects of applying the preconditioned cg-method that was already introduced in the previous chapter. The components developed for the multigrid method immediately allow the parallelization of the cg-method and provide additional preconditioners.

6.1 Initial Model Problem

On the unit interval, the one-dimensional Dirichlet-Poisson problem reads:

$$\begin{aligned} -u''(x) &= f(x) \quad x \in (0,1), && (6.1)\\ u(0) &= d_0, && (6.2)\\ u(1) &= d_1. && (6.3) \end{aligned}$$

This states a two-point boundary value problem on the interval $\Omega = (0,1)$ with Dirichlet boundary conditions. For $f \in C^0(\bar{\Omega})$ a unique solution $u^* \in C^2(\Omega) \cap C^0(\bar{\Omega})$ is given by (cp. [76] and [62])

$$u^*(x) = d_0 + (d_1 - d_0)x + \int_0^1 G(x,s)f(s)\,\mathrm{d}s,$$

where

$$G(x,s) := \begin{cases} s(1-x) & \text{if } 0 \le s \le x,\\ x(1-s) & \text{if } x < s \le 1, \end{cases}$$

defines *Green's function* for the above problem.

6.1.1 Finite Difference Approximation

The basic idea of the *Finite Difference Method (FDM)* is to replace derivatives with difference quotients. Thus a continuous problem as given through equations (6.1) to (6.3) gets approximated by a system of algebraic equations.

For a suitable grid size $h = \frac{1}{N+1}$ the continuous problem will be discretized on the equidistant grid

$$\bar{\Omega}_h = \{x_i = ih \mid i = 0, \dots, N+1\} \tag{6.4}$$

illustrated in Figure 6.1. The set of interior grid points is

$$\Omega_h = \{x_i = ih \mid i = 1, \dots, N\} \tag{6.5}$$

such that $\partial\Omega_h = \bar{\Omega}_h \backslash \Omega_h$ denotes grid points on the boundary. The second derivative in (6.1) gets approximated on Ω_h using second central differences, i. e.

$$u''(x_i) \approx \frac{u(x_{i+1}) - 2u(x_i) + u(x_{i-1})}{h^2}. \tag{6.6}$$

Numerical solutions are real valued, discrete functions that are defined on $\bar{\Omega}_h$ and in the following denoted as *grid functions.* The set of all grid functions is denoted as

$$V_h = \{v \mid v : \bar{\Omega}_h \to \mathbb{R}\} \tag{6.7}$$

and

$$V_{0,h} = \{v \in V_h \mid v|_{\partial\Omega_h} = 0\} \tag{6.8}$$

the subset of V_h containing grid functions that are zero at the boundaries. As a grid function $v_h \in V_h$ is specified by its values on grid points it can be represented as vector

$$v_h = (v_0, \dots, v_{N+1})^T := (v(x_0), \dots, v(x_{N+1}))^T \in \mathbb{R}^{N+2}$$

and analogously $v_{0,h} \in V_{0,h}$ as

$$v_{0,h} = (v_1, \dots, v_N)^T := (v(x_1), \dots, v(x_N))^T \in \mathbb{R}^N.$$

The finite difference approximation of the Dirichlet-Poisson problem can now be stated as:

Find $u_h = (u_0, \dots, u_{N+1}) \in V_h$ satisfying

$$\begin{aligned} -\frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} &= f(x_i) \quad i = 1, \dots, N, && (6.9)\\ u_0 &= d_0, && (6.10)\\ u_{N+1} &= d_1. && (6.11) \end{aligned}$$

The free variables of u_h are in the following denoted by $u_{0,h} = (u_1, \dots, u_N)^T$. Defining

$$T_N := \begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix} \in \mathbb{R}^{N \times N}, \quad q_h := \begin{pmatrix} f_1 + \frac{d_0}{h^2} \\ f_2 \\ \vdots \\ f_{N-1} \\ f_N + \frac{d_1}{h^2} \end{pmatrix} \in \mathbb{R}^N \tag{6.12}$$

allows for rewriting the system (6.9)-(6.11) as a system of linear equations

$$A_h u_{0,h} = q_h \quad \text{with } A_h := \frac{1}{h^2} T_N. \tag{6.13}$$

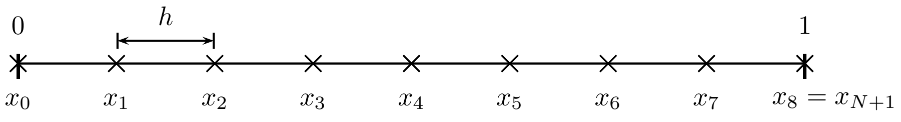


Figure 6.1: Equidistant grid for the one-dimensional Dirichlet-Poisson problem.

The right-hand side $q_h = (q_1, \dots, q_N)^T$ incorporates the boundary conditions and can be defined more formally by

$$q_i := f_i + \frac{1}{h^2}(\delta_{i,1}d_0 + \delta_{i,N}d_1). \tag{6.14}$$

Convergence of the numerical solution can be guaranteed if $f \in C^2(\bar{\Omega})$ (see e. g. [76]). For $v \in V_h$ the discrete sup-norm is defined by

$$||v_h||_{h,\infty} := \max_{i=0,\dots,N+1} |v(x_i)|. \tag{6.15}$$

Let u^* and u_h^* denote exact solutions of the continuous and discretized problem respectively then

$$||u^* - u_h^*||_{h,\infty} \leq \frac{||f''||_\infty}{96} h^2. \tag{6.16}$$

For $u_h, v_h \in V_h$ the inner product

$$(u_h, v_h)_h := h\left(\frac{u_0 v_0}{2} + \sum_{i=1}^{N} u_i v_i + \frac{u_{N+1} v_{N+1}}{2}\right) \tag{6.17}$$

induces a further norm on V_h, the so called *energy norm*

$$||u_h||_h := \sqrt{(u_h, u_h)_h}. \tag{6.18}$$

6.1.2 Implementation Notes

In the subsequent sections solvers for $A_h u_{0,h} = q_h$ are discussed and implementations thereof are introduced. Grid functions are represented through objects of type `DenseVector<..>` with index range `0..N+1`. For the sake of simplicity the element type is fixed to `double`. The right-hand side q_h gets assembled by the implementations according to (6.14). Parameters expected by solvers need to conform to the following conventions:

`rh`: integer containing the reciprocal of h, i. e. $N + 1$,

`f`: dense vector containing values[1] $(*, f_1, \dots, f_N, *)$ and

`u`: a dense vector initialized with $(d_0, *, \dots, *, d_1)$.

Function

```
template <typename U>
    double
    dp1d_normInf(const DenseVector<U> &u);
```

computes the discrete L_∞-norm defined in (6.15) and

```
template <typename U>
    double
    dp1d_norm2Sqr(const DenseVector<U> &u);
```

[1]As in Chapter 3 arbitrary values are indicated by $*$.

the square of the discrete L_2-norm defined in (6.18).

In the context of iterative methods, computation of the residual $r_h := q_h - A_h u_h$ in order to estimate the accuracy of a solution is frequently needed. Due to the special structure of A_h this computation can be carried out without setting up the matrix explicitly:

```
int rhh = rh*rh; // rhh = 1/h^2
int N = rh-1;
for (int i=1; i<=N; ++i) {
    r(i) = q(i) + rhh*(u(i-1)-2*u(i)+u(i+1));
}
```

This is realized by function[2] `dp1d_residual(rh, f, u, r)`. With respect to memory consumption and computational cost this is by far more efficient than setting up a banded matrix storing A_h and computing the residual using BLAS functions. The gain in efficiency achieved by exploiting the structure of A_h and providing a dedicated implementation is notable.

Exploiting the structure of a problem is crucial for all numerical implementations. This is taken into account in all implementations throughout this chapter:

1. Whenever there is a benefit for efficiency, algorithms are realized through single purpose implementations that are dedicated to one particular problem. Although the same numerical algorithm gets applied, different implementations are used to solve the problem in one and more dimensions.

2. Implementations are kept independent from FLENS unless there is a true benefit in using FLENS specific features. In this case types like `DenseVector` are merely used to provide a convenient reference implementation and could easily be replaced through an external implementation.

This sets up an empirical test case to measure the relevance of FLENS in numerical applications.

6.1.3 Relevance of FLENS

A major strength in the design of FLENS is the ability to easily define new matrix/vector types and to allow that related linear algebra operations are evaluated through dedicated functions. An obvious advantage is the enhanced readability of code without sacrificing efficiency. Defining the matrix type `DirichletPoisson1D` allows that the expression

```
r = f - A*u;
```

gets evaluated by directly calling `dp1d_residual(rh, f, u, r)`. Defining further for grid functions a dedicated type `GridVector1D` allows the computation of the discrete L_2-norm by

```
alpha = sqrt(r*r);
```

based on a call of `normL2Sqr(r)`.

In the remaining, implementations of `DirichletPoisson1D` and `GridVector1D` are merely outlined. Definitions of the `TypeInfo` traits are completely omitted. The coefficient matrix A_h in equation (6.13) is uniquely defined by its dimension. The corresponding FLENS type is derived from `SymmetricMatrix` and merely stores an integer containing $1/h = N + 1$:

```
class DirichletPoisson1D
    : public SymmetricMatrix<DirichletPoisson1D>
{
    public:
        DirichletPoisson1D();

        DirichletPoisson1D(int _rh);

        int rh;
};
```

Listing 6.1: FLENS matrix type for the coefficient matrix A_h.

[2] Arguments are assumed to comply with the specification defined above for the solvers.

The vector type for grid functions encapsulates beside an integer for $1/h = N+1$ a dense vector:

```
class GridVector1D
    : public Vector<GridVector1D>
{
    public:

        GridVector1D(int _rh)
            : rh(_rh), values(_(0,rh))
        {}

        // ... define operators =, +=, -=, ...

        int     rh;

        typedef DenseVector<Array<double> >  Grid;
        Grid    grid;
};
```

Listing 6.2: FLENS vector type representing grid functions V_h.

Operators treating closures are implemented according to the pattern described in Chapter 5. Wrappers for linear algebra operations can delegate functionality to existing functions, e. g.

```
void
copy(const GridVector1D &x, GridVector1D &y)
{
    y.rh = x.rh;
    copy(x.grid, y.grid);
}
```

6.2 Direct Solvers

In this section so called *direct solvers* for (6.13) are described. For direct solvers the total number of required operations is known a priori and only depends on the dimensions of the coefficient matrix. In general, for an $N \times N$ coefficient matrix at least $\mathcal{O}(N^3)$ operations are required. Because of some special properties of the coefficient matrix A_h it is possible to provide solvers that require much less operations. The first solver illustrated in this section is based on the Cholesky factorization and exploits the structure of A_h. The second solver utilizes the fact that A_h arises from the discretization of the Poisson problem and exploits the fact that properties of the continuous problem (6.1) to (6.3) are carried over.

6.2.1 Cholesky Factorization

For the specific tridiagonal matrix T_N defined in (6.12) the Cholesky factorization $T_N = L_N D_N L_N^T$ is explicitly known. Matrices L_N and D_N are given by

$$L_N = \begin{pmatrix} 1 & & & & \\ l_1 & 1 & & & \\ & l_2 & \ddots & & \\ & & \ddots & 1 & \\ & & & l_{N-1} & 1 \end{pmatrix}, \quad l_k = -\frac{k}{k+1} \tag{6.19}$$

and

$$D_N = \operatorname{diag}(d_1, \dots, d_N), \quad d_k = \frac{k+1}{k}. \tag{6.20}$$

Applying the factorization to (6.13) gives

$$L_N D_N L_N^T u_{0,h} = h^2 q_h$$

and can be solved in two steps as illustrated in Algorithm 6.1. Due to the sparsity of L_h the computation of $u_{0,h}$ requires $\mathcal{O}(N)$ operations.

Algorithm 6.1 (Cholesky Solver: One-Dimensional Dirichlet-Poisson Problem).

1. *Solve $L_h v = h^2 q_h$ by forward substitution:*

$$v_1 = h^2 q_1 = h^2 f_1 + d_0, \tag{6.21}$$

$$v_k = h^2 q_k - l_{k-1} v_{k-1} = h^2 f_k + \frac{k-1}{k} v_{k-1}, \quad k = 2, \ldots, N-1, \tag{6.22}$$

$$v_N = h^2 q_N - l_{N-1} v_{N-1} = h^2 f_N + d_1 + \frac{N-1}{N} v_{N-1}. \tag{6.23}$$

2. *Solve $D_h L_h^T u_{0,h} = v$ by backward substitution.*

$$u_N = \frac{v_N}{d_N} = \frac{N}{N+1} v_N, \tag{6.24}$$

$$u_k = \frac{v_k - d_k l_k u_{k+1}}{d_k} = \frac{k}{k+1}(v_k + u_{k+1}), \quad k = N-1, \ldots, 1. \tag{6.25}$$

Implementation

The forward substitution is carried out in lines 9-13. Hereby vector `u` sequentially gets overwritten with $v_1, \ldots, v_N$ according to (6.21)-(6.23). The backward substitution (6.24)-(6.25) is realized in lines 16-18 and overwrites `u` in reverse order such that in the k-th step values of $v_1, \ldots, v_k$ are still contained in `u(1),..,u(k)`.

```
 1 template <typename F, typename U>
 2 void
 3 dp1d_cholesky(int rh, const DenseVector<F> &f, DenseVector<U> &u)
 4 {
 5     int N = rh - 1;
 6     double hh = 1./(rh*rh);
 7
 8     // forward substitution
 9     u(1) = hh*f(1)+u(0);
10     for (int k=2; k<N; ++k) {
11         u(k) = hh*f(k) + u(k-1)*(k-1)/k;
12     }
13     u(N) = hh*f(N)+u(N+1) + u(N-1)*(N-1)/N;
14
15     // backward substitution
16     u(N) = u(N)*N/(N+1);
17     for (int k=N-1; k>=1; --k) {
18         u(k) = (u(k)+u(k+1))*k/(k+1);
19     }
20 }
```

Listing 6.3: Direct solver for the one-dimensional Dirichlet-Poisson problem based on the Cholesky factorization of T_N.

6.2.2 Fast Poisson Solver

Eigenvalues $\lambda_1, \ldots, \lambda_N$ and eigenvectors $v_N^1, \ldots, v_N^N$ of T_N are explicitly known to be[3]

$$\lambda_k = 2\left(1 - \cos\frac{\pi k}{N+1}\right), \quad v_N^k = \left(\sin\frac{\pi\, k\, l}{N+1}\right)_{l=1}^{N}. \tag{6.26}$$

[3]Note that functions $v_k(x) = sin(k\pi x)$ are eigenfunctions of the Laplace operator $L = \frac{\partial}{\partial x}$ fulfilling homogeneous boundary conditions on $\bar{\Omega}$, i.e. $v_k(0) = v_k(1) = 0$.

Eigenvectors v_N^k are also called the *frequency modes* of A_h reflecting that these are discrete sin-functions. Denoting $V = (v_N^1, \ldots, v_N^N)$ and $\Sigma = \operatorname{diag}(\lambda_1, \ldots, \lambda_N)$ the system of linear equations (6.13) can be factorized by

$$\frac{1}{h^2} V^{-1} \Sigma V u_{0,h} = q_h.$$

As T_N is symmetric, its eigenvectors are orthogonal and (see [38], [52])

$$V^{-1} = \frac{2}{N+1} V^T = \frac{2}{N+1} V.$$

This allows for solving the problem by computing

$$u_{0,h} = \frac{2h^2}{N+1} V \Sigma^{-1} V q_h. \tag{6.27}$$

Although V is dense only $\mathcal{O}(N \log_2 N)$ instead of $\mathcal{O}(N^2)$ operations are required to compute $u_{0,h}$ in this particular case. This is due to the fact that the matrix-vector products involving V and $\frac{2}{N+1}V$ correspond to a *Discrete Sine Transform (DST)* and its inversion. Both can be computed using the *Fast Fourier Transform (FFT)*. The resulting Algorithm 6.2 is known as *Fast Poisson Solver* and computes (6.27) in three steps.

Algorithm 6.2 (Fast Poisson Solver: One-Dimensional Dirichlet-Poisson Problem).

1. *compute* $\hat{q}_h = V q_h$ *using a DST* ($\mathcal{O}(N \log_2 N)$ *operations),*
2. *compute* $\tilde{q}_h = \Sigma^{-1} \hat{q}_h$ *(N operations),*
3. *compute* $u_{0,h} = \frac{2h^2}{N+1} V \tilde{q}_h$ *using an inverse DST* ($\mathcal{O}(N \log_2 N)$ *operations).*

Implementation

In the following implementation the open source library FFTW [30] is used to carry out the DSTs. All transforms supported by the FFTW need to be performed following the same pattern.

First a so called plan needs to be created that contains all information needed by the FFTW to compute the transform. Beside the type of transformation this includes the size of the data and its location in memory. The algorithms used by the FFTW can be adapted to the data size in order to improve efficiency. Therefore two options are provided for creating the plan:

1. `FFTW_MEASURE` causes the FFTW to perform several test runs and to measure execution times. Depending on the underlying hardware and the data size this might take a few seconds. In order to perform this benchmarks the memory containing the data will be overwritten.

2. `FFTW_ESTIMATE` does not initiate any computations but the resulting plan is probably sub-optimal.

Once a plan is created it can be executed multiple times. If the Poisson problem needs to be solved for multiple right-hand sides it might pay off to use the `FFTW_MEASURE` option in the initialization. This suggests to split the implementation of the Fast Poisson Solver into two functions. An initializer creates and returns a plan for the DST used by a second function realizing Algorithm 6.2:

```
unsigned fftw_flags = FFTW_MEASURE;
fftw_plan p = dp1d_fastpoissonsolver_init(rh, u, fftw_flags);

// init u, f ...

dp1d_fastpoissonsolver(rh, p, f, u);
```

Note that this implementation requires that both functions receive the same vector `u`. This restriction is for the sake of simplicity and for avoiding going too much into the details of the FFTW interface. For a more flexible implementation this restriction can easily be removed.

The FFTW library provides the function `fftw_plan_r2r_1d` to create plans for transforms where input and output is real-valued. Together with the option `FFTW_RODFT00` this is used in the function `dp1d_fastpoissonsolver_init` (Listing 6.4) to create a plan for a DST. The created plan is defined to be an *in-place* transform, this means the input data will be overwritten with the output. This allows that both transforms in Algorithm 6.2 can be realized using the same plan.

```
 1 template <typename U>
 2 fftw_plan
 3 dp1d_fastpoissonsolver_init(int rh, DenseVector<U> &u,
 4                             unsigned fftw_flags = FFTW_MEASURE)
 5 {
 6     int N = rh - 1;
 7     return fftw_plan_r2r_1d(N, &u(1), &u(1), FFTW_RODFT00, fftw_flags);
 8 }
```

Listing 6.4: Fast Poisson Solver (Initializer)

For a realization of Algorithm 6.2 it is important to consider that the FFTW library generally does not normalize any transform. For the DST this implies that components of vector u are transformed according to

$$u_k \to 2\sum_{j=1}^{N} u_j \sin\left(\frac{\pi\, j\, k}{N+1}\right), \quad k = 1, \ldots, N.$$

Using the above notation this means that executing the transform plan will overwrite the vector `u` according to

$$u \to 2V\, u.$$

Incorporating that characteristic of the FFTW, the following strategy is realized in the function `dp1d_fastpoissonsolver` (see Listing 6.5) in order to implement Algorithm 6.2:

Lines	Action	Result: `u` contains
18-20	initialization	$h^2 q_h$
23	execute transform	$2h^2 \hat{q}_h$
26-28	scale components of `u`	$2h^2 \tilde{q}_h$
31	execute transform	$4Vh^2\tilde{q}_h = 2(N+1)u_{0,h}$
32	scale `u`	$u_{0,h}$

Using other libraries to perform the DST this strategy needs to be adapted accordingly.

```
 9 template <typename F, typename U>
10 void
11 dp1d_fastpoissonsolver(int rh, fftw_plan &plan,
12                        const DenseVector<F> &f, DenseVector<U> &u)
13 {
14     double hh = 1./(rh*rh);
15     int N = rh - 1;
16 
17     // setup q from f and boundary nodes in u
18     u(_(1,N)) = hh*f(_(1,N));
19     u(1) += u(0);
20     u(N) += u(N+1);
21 
22     //- step 1 -
23     fftw_execute(plan);
24 
```

```
    //- step 2 -
    for (int k=1; k<=N; ++k) {
        u(k) /= 2*(1-cos(k*M_PI/(N+1)));
    }

    //- step 3 -
    fftw_execute(plan);
    u(_(1,N)) /= 2*(N+1);
}
```

Listing 6.5: Fast Poisson Solver for the one-dimensional Dirichlet-Poisson problem.

6.2.3 Embedding Direct Solvers into FLENS

Direct solver classes introduced in the following serve as abstractions for the above low-level implementations. Direct solvers will play an essential role for the implementation of the multigrid method illustrated and discussed in Section 6.4. The design of these classes is therefore focused on their later usage as components within the multigrid method.

All direct solver classes provide the same interface such that they are exchangeable in a generic implementation. Usage is separated into initialization and applying the underlying method:

```
// init solver
DirectSolver ds(A, f, u);

// apply solver
ds.solve();
```

The solver object internally keeps a reference to `u` and `f`. Overwriting these vectors and reapplying the `solve` method allows for solving the system of linear equations for multiple right-hand sides.

Applications using a direct solver can specify the particular type of solver through appropriate typedefs. For the Dirichlet-Poisson problem in one dimension `DirectSolver` for instance can be defined as

```
typedef FastPoissonSolver<DirichletPoisson1D, GridVector1D>  DirectSolver;
```

or

```
typedef Cholesky<DirichletPoisson1D, GridVector1D>           DirectSolver;
```

Exemplarily for the Fast Poisson Solver Listing 6.6 shows the declaration of a direct solver class. The class is templated with respect to matrix and vector type.

```
template <typename MatrixType, typename VectorType>
class FastPoissonSolver
{
    public:
        FastPoissonSolver(const MatrixType &A, const VectorType &f,
                          VectorType &_u);

        void
        solve();

    private:
        const VectorType &f;
        VectorType       &u;
        fftw_plan        plan;
};
```

Listing 6.6: Fast Poisson Solver for the one-dimensional Dirichlet-Poisson problem.

For solving the one-dimensional Dirichlet-Poisson problem class `FastPoissonSolver` is specialized for matrix type `DirichletPoisson1D` and vector type `GridVector1D`. The constructor initializes the solver and the `solve` method calls the `dp1d_fastpoissonsolver` function:

```
template <>
FastPoissonSolver <DirichletPoisson1D ,
                  GridVector1D >::FastPoissonSolver(const DirichletPoisson1D &A,
                                                    const GridVector1D &_f,
                                                    GridVector1D &_u)
    : f(_f), u(_u)
{
    plan = dp1d_fastpoissonsolver_init(f.rh, u.grid);
}

template <>
void
FastPoissonSolver <DirichletPoisson1D , GridVector1D >::solve()
{
    dp1d_fastpoissonsolver(u.rh, plan, f.grid, u.grid);
}
```

Listing 6.7: Fast Poisson Solver for the one-dimensional Dirichlet-Poisson problem.

6.3 Stationary Iterative Methods

The methods in this section are based on an additive decomposition $A = L + D + R$ where D, L and R represent the diagonal, strictly lower and strictly upper parts of A respectively. Initially the methods are introduced to solve a system of linear equations

$$Au = q,$$

where A is a square matrix not necessarily related to A_h defined in (6.13).

Using an initial guess $u^{(0)}$ the methods produce a sequence of vectors $u^{(1)}, u^{(2)}, \ldots$ converging to the solution if certain criterions are met. These criterions for convergence are in particular examined for $A = A_h$.

6.3.1 Jacobi Method

In matrix terms the Jacobi method is given as

$$u^{(n+1)} = \underbrace{-D^{-1}(L+U)}_{=:J_A} u^{(n)} + D^{-1}q.$$

Let u^* denote the exact solution then $u^* = J_A u^* + D^{-1}q$. Hence the error after n iterations is given as

$$\begin{aligned} e^{(n)} &:= u^* - u^{(n)} \\ &= J_A\left(u^* - u^{(n-1)}\right) = \cdots = (J_A)^n\left(u^* - u^{(0)}\right) \\ &= (J_A)^n e_0. \end{aligned}$$

As J_A is symmetric the Jacobi method converges if $\rho(J_A) < 1$, where

$$\rho(J_A) := \max\{|\lambda| : \lambda \text{ eigenvalue of } J_A\}$$

is the spectral radius of J_A. Eigenvectors of T_N and J_{A_h} are the same and eigenvalues are related by

$$\lambda_k(J_{A_h}) = 1 - \frac{\lambda_k(T_N)}{2} = \cos\left(\frac{k\pi}{N+1}\right).$$

This guarantees that the Jacobi method will eventually converge. However in general, convergence will be rather slow. Representing the initial error with respect to the eigenvectors, i. e. , $e_0 = \sum_{k=1}^{N} c_k v_N^k$ it follows that

$$e^{(n)} = \sum_{k=1}^{N} c_k \lambda_k (J_{A_h})^n v_N^k = \sum_{k=1}^{N} c_k \cos^n \left(\frac{k\pi}{N+1} \right) v_N^k. \tag{6.28}$$

This shows that from the initial error the lowest and highest frequency modes v_N^k, i. e. , when $k \approx 1$ or $k \approx N$, are damped slowly by a factor close to 1. Whereas frequency modes for $k \approx \frac{N+1}{2}$ are damped quickly. This smoothing property is the foundation for the multigrid method discussed in Section 6.4.

Implementation

For (6.13) the Jacobi method in component form is given as

$$u_i^{(n+1)} = \frac{1}{2}\big(u_{i-1}^{(n)} + u_{i+1}^{(n)} + h^2 f_i\big), \quad i = 1, \dots, N, \tag{6.29}$$

with

$$u_0^{(n)} := d_0 \quad \text{and} \quad u_{N+1}^{(n)} := d_1, \tag{6.30}$$

incorporating the boundary condition in each iteration according to (6.14). Due to the special structure of A_h an implementation can overwrite $u_{0,h}^{(n)}$ sequentially with $u_{0,h}^{(n+1)}$ using two scalar valued auxiliary variables:

```
template <typename F, typename U>
void
dirichlet_poisson_1d_jacobi(const DenseVector<F> &f, DenseVector<U> &u)
{
    const int N = f.length()-2;
    const double hh = 1./((N+1)*(N+1));

    double ui, ui_1 = u(0);
    for (int i=1; i<=N; ++i) {
        ui = u(i);
        u(i) = 0.5*(ui_1+u(i+1)+hh*f(i));
        ui_1 = ui;
    }
}
```

Listing 6.8: Jacobi-iteration for the one-dimensional Dirichlet-Poisson problem.

Note that only those values of `u` are overwritten that are corresponding to values in the interior of Ω. Hence it is not required to enforce boundary conditions (6.30) afterwards.

The Smoothing Property

For a vivid demonstration of the smoothing property we chose $f \equiv 0$ and $d_0 = d_1 = 0$ such that $u_h^* = u^* \equiv 0$ is the exact solution. Hence the error $e_h = u_h^{(n)} - u^*$ coincides with the current iteration $u_h^{(n)}$. In a first example we use as an initial guess

$$u_k^{(0)} = \sin(\omega_1 \pi x) + \sin(\omega_2 \pi x) \quad \text{with } k_1 \approx 1 \text{ and } k_2 \approx \frac{N+1}{2}$$

consisting of two frequency modes.

Figure 6.2 shows that the higher frequency is damped after only three iterations while the lower frequency remains almost unaffected after 20 iterations. On the other hand, Figure 6.3 shows for

$$u_k^{(0)} = \sin(\omega_1 \pi x) + \sin(\omega_2 \pi x) + \sin(\omega_3 \pi x) \quad \text{with } k_1 \approx 1,\ k_2 \approx \frac{N+1}{2} \text{ and } k_3 \approx N+1,$$

that the highest frequencies are also damped very slowly.

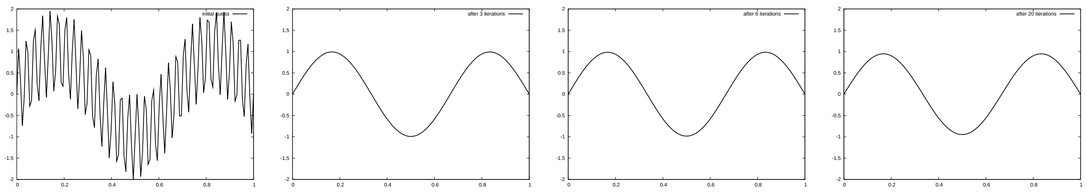


Figure 6.2: Jacobi method for $N = 127$: $u^{(0)} = v_3 + v_{60}$, $u^{(3)}$, $u^{(6)}$ and $u^{(20)}$.

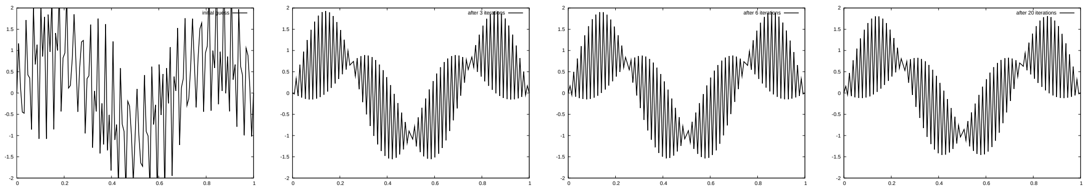


Figure 6.3: Jacobi method for $N = 127$: $u^{(0)} = v_3 + v_{60} + v_{123}$, $u^{(3)}$, $u^{(6)}$ and $u^{(20)}$.

6.3.2 Weighted Jacobi Method

For the multigrid method, smoothers damping the highest frequencies are desired. The weighted Jacobi method given by

$$\begin{aligned} u^{(n+1)} &= \omega(\underbrace{J_A u^{(n)} + D^{-1}q}_{\text{Jacobi method}}) + (1-\omega)u^{(n)} \\ &= \underbrace{(I - \omega D^{-1}A)}_{=:J_{A,\omega}} u^{(n)} + \omega D^{-1} q \end{aligned} \tag{6.31}$$

is a modification to achieve this. Eigenvalues of $J_{A_h,\omega}$ and T_N are related by

$$\lambda_k(J_{A_h,\omega}) = 1 - \frac{\omega}{2}\lambda_k(T_N) = 1 - \omega + \omega\cos\left(\frac{k\pi}{N+1}\right)$$

and eigenvectors are identical. After n iterations the initial error is reduced by

$$e^{(n)} = \sum_{k=1}^{N} c_k \lambda_k(J_{A_h,\omega})^n v_N^k = \sum_{k=1}^{N} c_k \left(1 - \omega + \omega\cos\left(\frac{k\pi}{N+1}\right)\right)^n v_N^k. \tag{6.32}$$

For the multigrid method introduced in Section 6.4, stationary iterative methods are used as smoothing operators. In this respect it is favorable to choose ω such that high frequencies are damped the fastest. This leads to

$$1 - \omega + \omega\cos\left(\frac{\pi N}{N+1}\right) \overset{!}{\approx} 0 \;\Rightarrow\; \omega = \frac{1}{2}.$$

From (6.31) and (6.29) one obtains the component form of the weighted Jacobi:

$$u_i^{(n+1)} = (1-\omega)u_i^{(n)} + \frac{\omega}{2}(u_{i-1}^{(n)} + u_{i+1}^{(n)} + h^2 f_i), \quad i = 1, \dots, N.$$

Implementation and Demonstration of the Smoothing Property

Listing 6.9 shows a simple implementation of the weighted Jacobi method. As can be seen in Figure 6.4 for $\omega = 1/2$ also the highest frequency modes are damped quickly from the initial error. Obviously the weighted Jacobi method requires two additional floating point multiplications

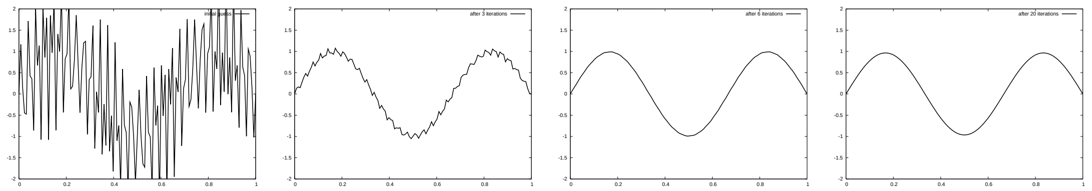


Figure 6.4: Weighted Jacobi method for $N = 127$: $u^{(0)} = v_3 + v_{60} + v_{123}$, $u^{(3)}$, $u^{(6)}$ and $u^{(20)}$

(line 14). These additional computations are traded for enhanced smoothing properties. For numerical applications it is interesting to investigate whether this tradeoff actually pays off. This in turn makes it desirable that different types of smoothers can be implemented as exchangeable components. How FLENS allows to construct such components in a straightforward manner will be illustrated in Subsection 6.3.4.

```
 1 template <typename F, typename U>
 2 void
 3 dirichlet_poisson_1d_weighted_jacobi(const double omega,
 4                                      const DenseVector<F> &f,
 5                                      DenseVector<U> &u)
 6 {
 7     const int N = f.length()-2;
 8     const double hh = 1./((N+1)*(N+1));
 9     const double c = omega/2;
10
11     double ui, ui_1 = u(0);
12     for (int i=1; i<=N; ++i) {
13         ui = u(i);
14         u(i) = (1-omega)*ui + c*(ui_1+u(i+1)+hh*f(i));
15         ui_1 = ui;
16     }
17 }
```

Listing 6.9: Weighted Jacobi-iteration for the one-dimensional Dirichlet-Poisson problem.

6.3.3 Gauss-Seidel Method

In matrix terms the Gauss-Seidel method is defined by

$$u^{(n+1)} = \underbrace{-(L+D)^{-1}U}_{=:G_A}\, u^{(n)} + (L+D)^{-1}q.$$

For $A = A_h$ it can be shown that $\rho(G_A) = \rho(J_A)^2$ such that the Gauss-Seidel method converges twice as fast as the Jacobi method [52].

Implementation

The component form of the Gauss-Seidel method simplifies to

$$u_i^{(n+1)} = \frac{1}{2}\left(u_{i-1}^{(n+1)} + u_{i+1}^{(n)} + h^2 f_i\right), \quad i = 1, \dots, N.$$

As opposed to the Jacobi method the Gauss-Seidel method requires computing components of the new iterate in a specific order.

An implementation can use the same vector to store $u^{(n)}$ and $u^{(n+1)}$. The component $u_i^{(n)}$ can be overwritten with $u_i^{(n+1)}$ such that no temporary variables are needed:

```
template <typename F, typename U>
void
```

```
dirichlet_poisson_1d_gauss_seidel(const DenseVector<F> &f, DenseVector<U> &u)
{
    const int N = f.length()-2;
    const double hh = 1./((N+1)*(N+1));

    for (int i=1; i<=N; ++i) {
        u(i) = 0.5*(u(i-1)+u(i+1)+hh*f(i));
    }
}
```

Listing 6.10: Gauss-Seidel-iteration for the one-dimensional Dirichlet-Poisson problem.

6.3.4 Embedding Stationary Iterative Methods into FLENS

Reflecting the observed smoothing properties, stationary iterative methods formally can be formally regarded as smoothing operators

$$S_{A_h,f_h} : u_h^{(n)} \to u_h^{(n+1)}.$$

This motivates the embedding of these methods into FLENS such that it is possible to code such smoothing operations by

```
Smoother S(A, f);

u = S*u;
```

Exemplarily an implementation of a smoother class is illustrated for the Gauss-Seidel method. In this case the type `Smoother` is defined as

```
typedef GaussSeidel<DirichletPoisson1D, GridVector1D>  Smoother;
```

All smoother classes are templated with respect to the type of coefficient matrix and vector type, such that this typedef can be exchanged in favor of another method. As can be seen from Listing 6.11, the class `GaussSeidel` is a simple specialization of a FLENS matrix keeping references to the coefficient matrix and the right-hand side vector.

```
template <typename M, typename V>
class GaussSeidel
    : public GeneralMatrix<GaussSeidel<M,V> >
{
    public:
        typedef M MatrixType;
        typedef V VectorType;

        GaussSeidel(const M &_A, const V &_f);

        const M &A;
        const V &f;
};
```

Listing 6.11: FLENS matrix type representing the Gauss-Seidel method as smoothing operator.

Redirection of the matrix-vector multiplication to carry out an iteration of the Gauss-Seidel method merely requires the implementation of an appropriate `mv` function. As Listing 6.12 shows, such an implementation basically consists of a call to the low-level implementation of the Gauss-Seidel method from Listing 6.10.

```
void
mv(Transpose trans, double alpha,
   const GaussSeidel<DirichletPoisson1D, GridVector1D> &GS,
   const GridVector1D &u_1, double beta, GridVector1D &u)
{
    assert(trans==NoTrans);
    assert(alpha==1.);
    assert(beta==0.);
    assert(ADDRESS(u)==ADDRESS(u_1));

    dp1d_gauss_seidel(GS.A.rh, GS.f.grid, u.grid);
}
```

Listing 6.12: Applying the Gauss-Seidel method as smoothing operator.

Unsupported operations like `u = 0.7*S*u` or `v = S*u` where `u` and `v` are intercepted by assertions in debug mode. If desired, these kind of operations can be supported additionally in a more flexible implementation.

Based on the above abstraction for smoothing operators class `StationaryIterativeSolver` allows for using stationary iterative methods as actual solvers. The class is templated with respect to the smoothing operator and provides the same interface for solving the system of linear equations as classes introduced in Section 6.2:

```
typedef StationaryIterativeSolver<Smoother>  Solver;

Solver sv(A, f, u);
sv.solve();
```

Analogously to direct solvers an object of type `StationaryIterativeSolver` keeps references to the coefficient matrix, the right-hand side and the solution vector. In addition a smoothing operator and a local vector `r` internally used to store the residual are allocated:

```
template <typename Method>
class StationaryIterativeSolver
{
    public:
        typedef typename Method::MatrixType MatrixType;
        typedef typename Method::VectorType VectorType;

        StationaryIterativeSolver(const MatrixType &A, const VectorType &f,
                                  VectorType &_u);

        void
        solve();

    private:
        Method            S;
        const MatrixType  &A;
        const VectorType  &f;
        VectorType        &u, r;
};
```

Listing 6.13: Applying the Gauss-Seidel method as smoothing operator.

The `solve` method applies the smoothing operator until either a maximum number of iterations is reached or the norm of the residual falls below machine accuracy:

```
template <typename Method>
void
StationaryIterativeSolver<Method>::solve()
{
    double eps = std::numeric_limits<double>::epsilon();
    int maxIt = 500;

    r = f - A*u;
    for (int i=1; (i<=maxIt) && (normL2(r)>eps); ++i) {
        u = S*u;
        r = f - A*u;
    }
}
```

Listing 6.14: Using the Gauss-Seidel method as solver.

6.4 Multigrid Method

Applying the FDM to solve the Poisson problem provides an ideal example to illustrate basic ideas and concepts of the multigrid method. In this section we use a discretization of (6.1)-(6.3) that allows a hierarchical set of grids[4]

$$\bar{\Omega}_h \supset \bar{\Omega}_{2h} \supset \bar{\Omega}_{4h} \supset \cdots \supset \bar{\Omega}_{2^p h}. \tag{6.33}$$

The multigrid method takes advantage of the smoothing properties observed in Section 6.3 for the stationary iterative methods. The multigrid method further requires operators to approximate grid functions on coarser and finer grids:

$$R_h^{2h} : V_{0,h} \to V_{0,2h}, \quad P_{2h}^h : V_{0,2h} \to V_{0,h}$$

denoted as *restriction* and *prolongation* operators respectively[5]. A thorough and more general study of the multigrid method is given in [12], [52] and [36].

6.4.1 Basic Idea

Assume that for (6.13) an initial guess $u_{0,h}^{(0)}$ is given such that the error

$$e_{0,h} := u_{0,h}^* - u_{0,h}^{(0)} = v_N^k = \left(\sin\left(\frac{\pi k l}{N+1} \right) \right)_{l=1}^N$$

consists of a single eigenvector of A_h. Using the Jacobi method this error would be damped after each iteration by factor $\lambda_k(J_{A_h}) = \cos\left(\frac{k\pi}{N+1}\right)$. From (6.4.1) it follows that the residual

$$r_h = q_h - A_h u_{0,h}^{(\nu)} = A_h u_{0,h}^* - A_h u_{0,h}^{(\nu)} = A_h e_h^{(\nu)} = \lambda_k(A_h)\, v_N^k$$

is a scalar multiple of v_N^k. Obviously the problem (6.13) is equivalent to

$$A_h e_{0,h} = r_h$$

and solutions related by $u_{0,h}^* = u_{0,h}^{(0)} + e_{0,h}$. As will be shown in the following, an approximate solution of (6.4.1) can be obtained by solving

$$A_{2h} u_{0,2h} = q_{2h}, \quad q_{2h} := R_h^{2h} r_h. \tag{6.34}$$

Defining the restriction by taking only values of r_h at grid points of Ω_{2h} it follows that

$$q_{2h} := r_h|_{\Omega_{2h}} = \lambda_k(A_h) \left(\sin\left(\frac{2\pi k l}{N+1} \right) \right)_{l=1}^{\frac{N+1}{2}-1} \tag{6.35}$$

$$= \lambda_k(A_h) \left(\sin\left(\frac{\pi k l}{n+1} \right) \right)_{l=1}^{n} = \lambda_k(A_h) v_n^k \tag{6.36}$$

is a scalar multiple of the k-th eigenvector of A_{2h}. Hence the exact solution of (6.34) is given by

$$\begin{aligned} u_{0,2h}^* &= \frac{\lambda_k(A_h)}{\lambda_k(A_{2h})}\, v_n^k = 4\, \frac{\lambda_k(T_N)}{\lambda_k(T_n)}\, v_n^k = 4\, \frac{1-\cos\left(\frac{\pi k}{N+1}\right)}{1-\cos\left(\frac{\pi k}{n+1}\right)}\, v_n^k = 4\, \frac{\sin^2\left(\frac{\pi k}{2(N+1)}\right)}{\sin^2\left(\frac{\pi k}{N+1}\right)}\, v_n^k \\ &= 4\, \frac{\sin^2\left(\frac{\pi k h}{2}\right)}{\sin^2(\pi k h)}\, v_n^k. \end{aligned}$$

[4]In Section 7.5 the multigrid method will be applied for a non-hierarchical set of grids.

[5]Defining these operators on $V_{0,h}$ is sufficient for the methods introduced in this section. For the so called *full multigrid method* these also have to be defined on V_h.

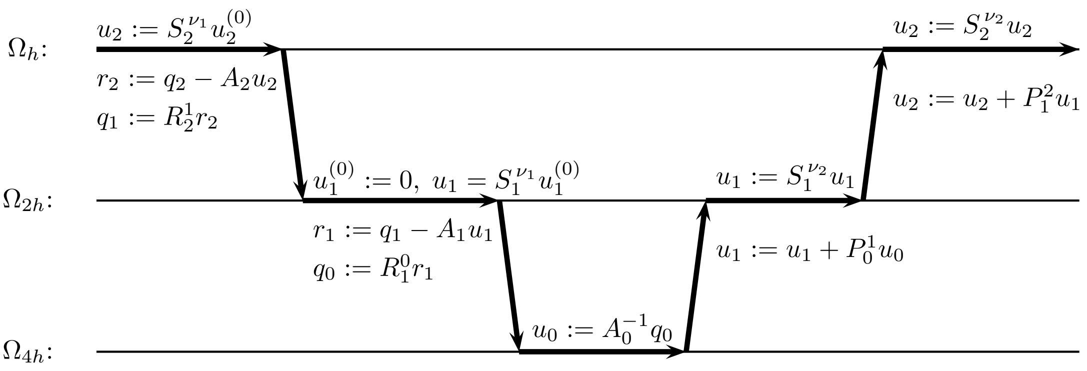


Figure 6.5: V-Cycle with three grids.

For $k \approx 1$ it follows that $u_{0,2h}^* \approx v_n^k$ such that the solution of (6.13) can be approximated by

$$u_{0,h}^* \approx u_{0,h}^{(0)} + P_{2h}^h u_{0,2h}^*.$$

To illustrate the basic idea of the multigrid method two observations are notable:

1. Compared to (6.13) problem (6.34) contains about half the number of unknowns.

2. Stationary methods applied to (6.34) might converge faster: Using an initial guess $u_{0,2h}^{(0)} \equiv 0$ the error gets damped by factor $\lambda_k(J_{A_{2h}}) = \cos\left(\frac{k\pi}{n+1}\right)$. So in particular low frequency modes in the error get damped faster.

   Consider the numerical example where $N = 127$ and $k = 9$ such that $\cos\left(\frac{k\pi}{N+1}\right) \approx 0.9757$, $\cos\left(\frac{k\pi}{n+1}\right) \approx 0.9040$ and $4 \cdot \sin^2\left(\frac{\pi k h}{2}\right) / \sin^2(\pi k h) \approx 1.0123$. After 6 iterations on $\bar{\Omega}_{2h}$ the error gets reduced by factor $1.0123 \cdot 0.9040^6 \approx 0.5525$. On $\bar{\Omega}_h$ the same reduction requires about 24 iterations (where each iteration requires twice as many operations).

6.4.2 Coarse Grid Correction and Multigrid V-Cycle

The idea of using the coarser grid Ω_{2h} to improve numerical solutions on Ω_h is realized in the *coarse grid correction* method introduced next. The multigrid V-cycle extends this concept to take advantage of a larger set of hierarchical grids. To allow a simpler notation let

$$A_l = A_{2^{p-l}h},\ u_l = u_{0,2^{p-l}h},\ q_l = q_{2^{p-l}h},\ r_l = r_{2^{p-l}h} \quad l = 0, \dots, p,$$

denote coefficient matrices, grid functions and vectors related to the different grids in (6.33). Hence, the finest and coarsest grid is identified by $l = p$ and $l = 0$ respectively. Analogously let

$$R_l^{l-1} = R_{2^{p-l}h}^{2^{p-l}2h}, \quad P_{l-1}^l = P_{2^{p-l}2h}^{2^{p-l}h}, \quad S_l = S_{A_{2^{p-l}h}, q_{2^{p-l}h}}$$

denote restriction, prolongation and smoothing operators respectively.

The *coarse grid correction* performs ν_1 pre-smoothing operations before solving the auxiliary problem on the coarse grid up to machine accuracy. After updating the numerical solution on the fine grid ν_2 post-smoothing operations are applied:

Algorithm 6.3 (Coarse Grid Correction).

1. *Pre-smooth initial guess:* $u_1 := S_1^{\nu_1} u_1$.

2. *Compute and restrict residual:* $r_1 := q_1 - A_1 u_1,\ q_0 := R_1^0 r_1$.

3. *'Solve' on Ω_{2h}: $u_0 := A_0^{-1} q_0$.*
4. *Update the approximation on Ω_h: $u_1 := u_1 + P_0^1 u_0$*
5. *Post-smooth: $u_1 := S_1^{\nu_2} u_1$.*

Step 3 can either be carried out by using a direct solver or by applying the same method to Ω_{2h} and Ω_{4h}. This leads to the *multigrid V-cycle* illustrated in Figure 6.5 for a set of three grids. For a set of p grids the algorithm can be defined recursively:

Algorithm 6.4 (V-Cycle).

```
procedure V-Cycle($l$)
    if $l = 0$
        $u_0 := A_0^{-1} q_0$        (solve exact)
    else
        $u_l := S_l^{\nu_1} u_l^{(0)}$
        $r_l := q_l - A_l u_l$
        $q_{l-1} := R_l^{l-1} r_l$
        $u_{l-1} := 0$
        call V-Cycle($l-1$)
        $u_l := u_l + P_{l-1}^l u_{l-1}$
        $u_l := S_l^{\nu_2} u_l$
    end if
```

The V-cycle gets started with $l = p$ after initializing the initial guess u_p and right-hand side q_p. Only on the coarsest grid the system of linear equations is solved exactly.

For each grid level $l = 0, \dots, p$ an implementation has to allocate vectors u_l, q_l, r_l. For the one-dimensional problem considered, the total amount of memory needed is $\mathcal{O}(N)$[6]. Using one of the stationary smoothers introduced in Section 6.3, the total number of operations performed in one V-cycle is $\mathcal{O}(N)$[7].

A most notable feature of the multigrid method is that the rate of convergence can be independent of the grid size $h = \frac{1}{N+1}$. For two grid levels, i. e., $p = 1$ applying one V-cycle[8] to the initial guess $u_1^{(0)}$ leads to

$$u_1 = \underbrace{S_1^{\nu_2}\left(I - P_0^1 A_0^{-1} R_1^0 A_1\right) S_1^{\nu_1}}_{:=M_{\nu_1,\nu_2}} u_1^{(0)} + S_1^{\nu_2} P_0^1 A_0^{-1} R_1^0 q_1 \tag{6.37}$$

such that convergence is guaranteed if $\rho\left(M_{\nu_1,\nu_2}\right) < 1$. It was shown in [36] that for certain smoothing, restriction and prolongation operators[9] the spectral radius is uniformly bound by

$$\rho\left(M_{\nu_1,\nu_2}\right) \le C_{\nu_1+\nu_2} < 1$$

where $C_{\nu_1+\nu_2}$ does not depend on h. Hence an approximate solution with accuracy ε can be obtained after $n_\varepsilon = \mathcal{O}(\log 1/\varepsilon)$ iterations. The analysis for the Dirichlet-Poisson discussed here is rather straightforward and can easily be extended to treat the problem in higher dimensions as long as the domains remain simple. The more general case of self-adjoint operators on general domains was studied in [36] and [51].

[6] In total vectors $u_0, \dots, u_p$ contain $\sum_{l=0}^{p} \left(2^{-l}(N+1) - 1\right) = \mathcal{O}(N)$ elements.

[7] On level l one smoothing operation requires is also $\mathcal{O}(2^{-l}N)$.

[8] This is commonly denoted as *Two-Grid Iteration.*

[9] For the one-dimensional Dirichlet-Poisson problem the weighted Jacobi method with $\omega = \frac{1}{2}$ was used as smoothing operator S_1. R_1^0 and P_0^1 were defined as so called weighted restriction and prolongation operators which are both defined in the next subsection

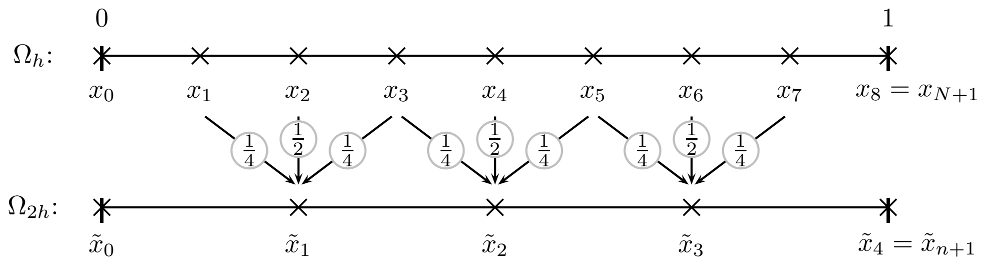


Figure 6.6: Weighted restriction $R_h^{2h} : V_{0,h} \to V_{0,2h}$.

6.4.3 Restriction and Prolongation Operators

In (6.35), the restriction was defined such that $v = (v_1, \dots, v_N) \in V_{0,h}$ and its approximation $\tilde{v} = (\tilde{v}_1, \dots, \tilde{v}_n) \in V_{0,2h}$ were related by

$$\tilde{v}_i = v_{2i}, \quad i = 1, \dots, n.$$

Obviously, this type of restriction operator inherits a loss of information.
For the Dirichlet-Poisson problem better convergence rates can be obtained by using the *weighted restriction* where

$$\tilde{v}_i = \frac{1}{4}v_{2i-1} + \frac{1}{2}v_{2i} + \frac{1}{4}v_{2i+1}, \quad i = 1, \dots, n. \tag{6.38}$$

Figure 6.6 illustrates the weighted restriction for $N = 7$. Denoting

$$R = \frac{1}{4}\begin{pmatrix} 1 & 2 & 1 & & & & \\ & & 1 & 2 & 1 & & \\ & & & & 1 & 2 & 1 \\ & & & & & & \dots \end{pmatrix} \in \mathbb{R}^{n \times N}$$

the weighted restriction can be expressed as $\tilde{v} = Rv$. An implementation for this particular matrix-vector product can be realized according to (6.38):

```
template <typename U, typename F, typename R>
void
dp1d_restriction(const DenseVector<V> &v, DenseVector<VC> &vc);
{
    int N = rh-1;
    for (int i=1, I=2*i; i<=N; ++i, I+=2) {
        vc(i) = 0.25*v(I-1) + 0.5*v(I) + 0.25*v(I+1);
    }
}
```

Listing 6.15: Weighted restriction: computing $vc := Rv$

A prolongation $v = (v_1, \dots, v_N) = P_{2h}^h(\tilde{v}_1, \dots, \tilde{v}_n) \in V_{0,2h}$ can be obtained by using linear interpolation, such that

$$\begin{aligned} v_{2i} &= \tilde{v}_i, && i = 1, \dots, n, \\ v_{2i+1} &= \frac{1}{2}(\tilde{v}_i + \tilde{v}_{i+1}), && i = 0, \dots, n. \end{aligned}$$

For $N = 7$ the corresponding prolongation stencil is illustrated in Figure 6.7.

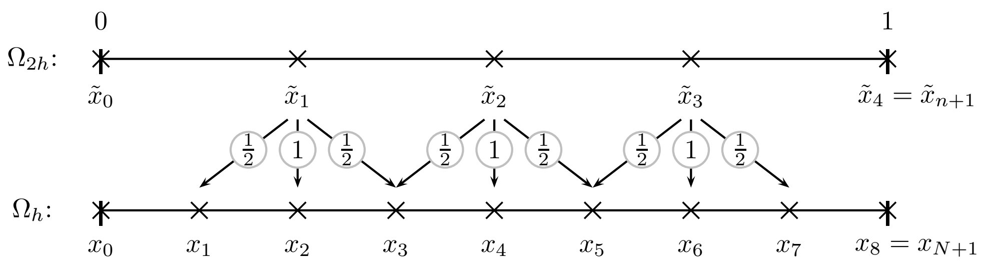


Figure 6.7: Prolongation $P_{2h}^h : V_{0,2h} \to V_{0,h}$.

The prolongation can be expressed as matrix-vector product $v = P\tilde{v}$ where

$$P = \frac{1}{2}\begin{pmatrix} 1 & & \\ 2 & & \\ 1 & 1 & \\ & 2 & \\ & 1 & 1 \\ & & \vdots \end{pmatrix} \in \mathbb{R}^{N\times n}.$$

For the multigrid method the relevant operation is the updating of a grid function v according to $v \to v + P_{2h}^h \tilde{v}$. A single purpose implementation of this operation is given by

```
template <typename VC, typename V>
void
dp1d_prolongation(const DenseVector<VC> &vc, DenseVector<V> &v)
{
    int N =rh-1;

    for (int i=1, I=2*i; i<=N; ++i, I+=2) {
        v(I-1) += 0.5*vc(i);
        v(I)   += vc(i);
        v(I+1) += 0.5*vc(i);
    }
}
```

Listing 6.16: Prolongation and update: compute $v := v + P_{2h}^h v_c$ where $v_c \in V_{0,2h}$ and $v \in V_{0,h}$.

6.4.4 Embedding Restriction and Prolongation Operators into FLENS

As is the case with the smoothing operators, the restriction and prolongation operators can be realized by defining appropriate FLENS matrix types. In this case empty matrix types are sufficient:

```
class Restriction  : public GeneralMatrix<Restriction> {};

class Prolongation : public GeneralMatrix<Prolongation> {};
```

Listing 6.17: Restriction and Prolongation operators.

The only purpose of these types is the ability to provide `mv` functions delegating the operations to appropriate low-level functions, e. g. the restriction to `dp1d_restriction`.

```
void
mv(Transpose trans, double alpha, const Restriction &R,
   const GridVector1D &v, double beta, GridVector1D &vc)
{
    // assertions ...

    dp1d_restriction(v.grid, vc.grid);
}
```

Listing 6.18: Matrix-vector multiplication delegated to the restriction implementation.

Proceeding analogously for the prolongation appliance of these operators can be coded by

```
Restriction  R;
Prolongation P;

GridVector1D v(rh), vc(rh/2);

vc = R*v;
v += P*vc;
```

Different types of restriction or prolongation operators can be supported by defining further matrix types. For each new restriction or prolongation type a corresponding `mv` version delegates the functionality to a specific implementation.

6.4.5 Implementation of the Multigrid V-Cycle

The most relevant components for the implementation of the multigrid method are the restriction, prolongation and smoothing operators as well as a solver applied on the coarsest grid. Abstractions made for these ingredients can be used for a generic implementation. The main advantage of such an implementation is an enormous degree of flexibility. Listing 6.19 shows how a particular multigrid method can be specified through a list of typedefs: For instance, in this setup the Gauss-Seidel method is used as smoother (line 4) and the fast poisson solver applied on the coarsest grid (line 5).

```
1 typedef DirichletPoisson1D                                MatType;
2 typedef GridVector1D                                      VecType;
3
4 typedef GaussSeidel<MatType, VecType>                     Smoother;
5 typedef FastPoissonSolver<MatType, VecType>               SVR;
6 typedef Restriction                                       R;
7 typedef Prolongation                                      P;
8
9 typedef MultiGrid<MatType, VecType, R, P, Smoother, SVR>  MG;
```

Listing 6.19: Exemplary setup of the multigrid method.

Before going into the detail of the multigrid implementation, the setup of relevant data structures is considered in Listing 6.20. The hierarchical set of grids in (6.33) is chosen such that finest and coarsest grids have mesh sizes $h = 2^{-l_{\max}}$ and $2^p h = 2^{-l_{\min}}$ respectively (lines 1-2). Vectors representing $\{f_l\}_{l=0}^p$, $\{r_l\}_{l=0}^p$ and $\{u_l\}_{l=0}^p$ as well as the set of coefficient matrices $\{A_l\}_{l=0}^p$ are realized through ordinary C arrays (lines 5-6). After allocation of these sets (lines 9-12) the right-hand side f_p and initial guess u_p are initialized for the specific Dirichlet-Poisson problem (line 15). The solver for the coarsest grid (line 17) gets passed to the multigrid implementation beside the set of grid vectors and coefficient matrices (line 18). The V-cycle gets applied until either a maximum number of iterations is reached or the residual has fallen below machine accuracy (line 20-22).

```
1 const int lMax = 10, lMin = 2;
2 const int p = lMax-lMin;
3 const int maxIt = 20;
4
5 MatType A[p+1];
6 VecType f[p+1], r[p+1], u[p+1];
7
8 // allocate:   A[l], f[l], r[l], p[l]
9 for (int l=p, rh=(1<<lMax); l>=0; --l, rh/=2) {
10     A[l] = DirichletPoisson1D(rh);
11     f[l] = r[l] = u[l] = GridVector1D(rh);
}

// initialize: f[p] and initial guess u[p]
problemSet1(f[p], u[p]);

SVR svr(A[0], f[0], u[0]);
MG mg(A, f, r, u, svr);

for (int it=1; (it<=maxIt) && (normL2(r[p])>eps); ++it) {
    mg.vCycle(p, 1, 1);
}
```

Listing 6.20: Exemplary setup of the multigrid method.

The multigrid method is implemented following a similar approach used for the direct solvers in Section 6.2.3 and the stationary solvers in Section 6.3.4. Required data structures are passed to the constructor of the multigrid class

```
MG mg(A, f, r, u, svr);
```

and referenced by the multigrid object. This simply makes these structures available to the `vCycle` method according to an internally specified naming convention. Due to the FLENS embedding of relevant operators the V-cycle can be implemented using a syntax that directly reflects the pseudo code in Algorithm 6.4:

```
template <typename Mat, typename Vec, typename Res, typename Pro,
          typename S, typename SVR>
void
MultiGrid<Mat, Vec, Res, Pro, S, SVR>::vCycle(int l, int v1, int v2)
{
    if (l==0) {
        svr.solve();
    } else {
        for (int v=1; v<=v1; ++v) {
            u[l] = S(A[l],f[l])*u[l];
        }
        r[l] = f[l] - A[l]*u[l];
        f[l-1] = R*r[l];

        u[l-1] = 0;
        vCycle(l-1,v1,v2);

        u[l] += P*u[l-1];
        for (int v=1; v<=v2; ++v) {
            u[l] = S(A[l],f[l])*u[l];
        }
    }
}
```

Listing 6.21: FLENS implementation of the multigrid V-cycle.

The ability of exchanging solver, smoother, restriction and prolongation is one obvious advantage achieved by this concept. Due to the technique realized in FLENS for the evaluation of linear algebra expressions the above implementation of the V-cycle is equivalent in terms of efficiency to an implementation that directly calls the low-level functions. A further benefit is shown in the next section where the very same implementation of the V-cycle is used to solve the Dirichlet-Poisson problem in higher dimensions.

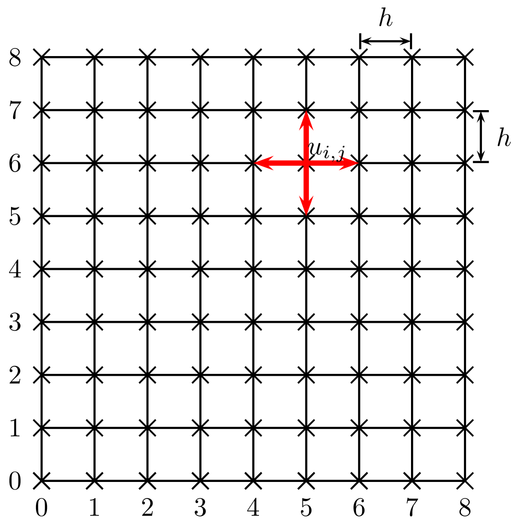


Figure 6.8: Equidistant grid $\bar{\Omega}$ for the two-dimensional Dirichlet-Poisson problem and the *five-point stencil* for the approximation of $u_{xx} + u_{yy}$ at (x_i, y_j).

6.5 The Dirichlet-Poisson Problem in Two Dimensions

In this section numerical methods introduced in Sections 6.2 to 6.4 are adapted to solve the two-dimensional Dirichlet-Poisson problem

$$\begin{aligned} -u_{xx} - u_{yy} &= f \quad \text{in } \Omega, && (6.39)\\ u &= d \quad \text{on } \partial\Omega. && (6.40) \end{aligned}$$

on domain $\Omega = (0,1)^2$. The FDM will be applied on the equidistant grid (Figure 6.8)

$$\bar{\Omega}_h = \{(x_i, y_i) = (ih, jh) : i = 0, \dots, N+1,\ j = 0, \dots, N+1\}, \quad h = \frac{1}{N+1}, \tag{6.41}$$

with interior grid points

$$\Omega_h = \{(x_i, y_i) = (ih, jh) : i = 1, \dots, N,\ j = 1, \dots, N\}. \tag{6.42}$$

Let V_h and $V_{0,h}$ denote sets of grid functions as specified in (6.7) and (6.8) respectively. Replacing on $\bar{\Omega}_h$ the partial derivatives in (6.39) with second order central differences leads to the finite difference approximation of problem (6.39)-(6.40):

Find $u_h \in V_h$ satisfying

$$\begin{aligned} -\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2} - \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2} &= f_{i,j} \quad i,\ j \in \{1, \dots, N\}, && (6.43)\\ u_{i,j} &= d_{i,j} \quad i \text{ or } j \in \{0, N+1\}. && (6.44) \end{aligned}$$

Numbering the grid points $\bar{\Omega}_h$ allows for representing of grid functions as vectors. Specifying a row-wise numbering a grid function $v_h \in V_h$ can be represented as

$$v_h = (v_{0,0}, \dots, v_{N+1,0}, v_{0,1}, \dots)^T = (v(x_0, y_0), \dots, v(x_{N+1}, y_0), v(x_0, y_1), \dots)^T \in \mathbb{R}^{(N+1)^2}$$

and $v_{0,h} \in V_{0,h}$ as vector

$$v_{0,h} = (v_{1,1}, \dots, v_{N,1}, v_{1,2}, \dots)^T = (v(x_1, y_1), \dots, v(x_N, y_1), v(x_1, y_2), \dots)^T \in \mathbb{R}^{N^2}.$$

For this row-wise numbering system (6.43)-(6.44) is equivalent to

$$A_h u_{0,h} = q_h \tag{6.45}$$

with coefficient matrix

$$A_h := \frac{1}{h^2} \begin{pmatrix} T_N + 2I & -I & & & \\ -I & T_N + 2I & -I & & \\ & \ddots & \ddots & \ddots & \\ & & -I & T_N + 2I & -I \\ & & & -I & T_N + 2I \end{pmatrix} \in \mathbb{R}^{N^2 \times N^2} \tag{6.46}$$

and right-hand side $q_h = (q_{1,1}, \dots, q_{N,1}, q_{1,2}, \dots)^T \in \mathbb{R}^{N^2}$ defined through

$$q_{i,j} := f_{i,j} + \frac{1}{h^2}(\delta_{i,1} d_{0,j} + \delta_{i,N} d_{N+1,j} + \delta_{j,1} d_{i,0} + \delta_{j,N} d_{i,N+1}). \tag{6.47}$$

Analogously to the one-dimensional problem two norms on V_h are defined by

$$||v_h||_{h,\infty} := \max_{i,j} |v_{i,j}| \tag{6.48}$$

and

$$||v_h||_h := h \left(\sum_{i,j} c_{i,j} \, (v_{i,j})^2 \right)^{1/2} \quad \text{where} \quad c_{i,j} = \begin{cases} 1 & \text{if } i,j \in \{1, \dots, N\}, \\ \frac{1}{2} & \text{else.} \end{cases} \tag{6.49}$$

6.5.1 Implementation Notes

By numbering grid points, a grid function can be formally treated as a vector. Independent of the problem dimensions this allows for rewriting the finite difference approximation of the Dirichlet-Poisson problem as a system of linear equations. For a general numerical analysis this is a crucial point. However, for the numerical realization fast access to values at grid points is crucial. For instance, the residual $r_h = q_h - A_h u_{0,h}$ can be computed efficiently by

```
for (int i=1; i<=N; ++i) {
    for (int j=1; j<=N; ++j) {
        r(i,j) = f(i,j)-rhh*(4*u(i,j)-u(i-1,j)-u(i+1,j)-u(i,j-1)-u(i,j+1));
    }
}
```

Listing 6.22 shows how the theoretical and practical aspects can be combined using FLENS. The formal treatment of grid functions as vectors is reflected by deriving class `GridVector2D` from base class `Vector`. Values at grid points are internally stored in a matrix to provide fast access.

```
class GridVector2D
    : public Vector<GridVector2D>
{
    public:
        typedef GeMatrix<FullStorage<double, RowMajor> >  Grid;

        GridVector2D(int _rh);

        // operators =, +=, -=, *=

        int    rh;
        Grid   grid;
};
```

Listing 6.22: Grid functions on two-dimensional domains.

Numerical computations can be carried out by low-level implementations receiving the grid function values stored in a matrix. For example, the function

```
template <typename U, typename F, typename R>
    void
    dp2d_residual(int rh, const GeMatrix<F> &f, const GeMatrix<U> &u,
                  GeMatrix<R> &r);
```

can compute the residual on grid points as shown above. Defining a matrix type[10] for the coefficient matrix in (7.49) the residual can be computed in a high-level implementation by

```
DirichletPoisson2D A(rh);
GridVector2D       r(rh), f(rh), u(rh);

// ...

r = f - A*u;
```

This shows that theoretical and practical aspects are combined orthogonally. Implementations that only rely on mathematical properties are not affected by decisions made regarding efficiency.

6.5.2 Direct Solvers

Direct solvers for the two-dimensional Dirichlet-Poisson problem are accessible through the same uniform interface provided for the one-dimensional problem:

```
DirichletPoisson2D A;
GridVector2D       f(rh), u(rh);

// init f, u

DirectSolver ds(A, f, u);
ds.solve();
```

Typedef `DirectSolver` can either be defined as

```
FastPoissonSolver<DirichletPoisson2D, GridVector2D>    DirectSolver;
```

or

```
Cholesky<DirichletPoisson2D, GridVector2D>             DirectSolver;
```

Both solvers embed low-level implementations that exploit the structure and special properties of the coefficient matrix A_h.

Cholesky Factorization

As can be seen from (7.49) in two dimensions the coefficient matrix A_h is a banded matrix with N sub- and super-diagonals respectively. For the Cholesky factorization $A_h = L_h D_h L_h^T$ it follows that L_h is a lower triangular matrix with N sub-diagonals. However, while A_h is sparse — with at most five non-zero entries per row — all sub-diagonals of L_h are fully occupied.

As can be inferred from Section 3.4.2 the number of operations required to perform the Cholesky factorization of A_h is $\mathcal{O}(N^4)$. Forward and backward substitution require $\mathcal{O}(N^3)$ operations each.

A solver using the Cholesky factorization can be implemented using the LAPACK functions `pbtrf` and `pbtrs`. Function `pbtrf` overwrites a matrix of type `SbMatrix` with its Cholesky factorization. Subsequently `pbtrs` can be used to solve the system of linear equations. Functions `dp2d_cholesky_init` and `dp2d_cholesky` are realized using these LAPACK functions:

```
template <typename A>
    void
    dp2d_cholesky_init(int rh, SbMatrix<A> &A);

```

[10]Class `DirichletPoisson2D` can be implemented analogously to class `DirichletPoisson1D`.

```
template <typename A, typename F, typename U>
    void
    dp2d_cholesky(int rh, const SbMatrix<A> &A, const GeMatrix<F> &F,
                  GeMatrix<U> &U);
```

Listing 6.23: Solver for the two-dimensional Dirichlet-Poisson problem based on the Cholesky factorization.

Fast Poisson Solver

The system of linear equations (6.45) is equivalent to the matrix equation

$$UT_N + T_N U = h^2 Q \quad \text{where } U = (u_{i,j})_{i,j=1,\dots N},\ Q = (q_{i,j})_{i,j=1,\dots N}. \tag{6.50}$$

To show this equivalence let

$$A \otimes B = \begin{pmatrix} a_{1,1}B & \dots & a_{1,n}B \\ \vdots & \ddots & \vdots \\ a_{m,1}B & \dots & a_{m,n}B \end{pmatrix} \in \mathbb{R}^{mp\times nq}$$

denote the Kronecker product for matrices $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{p\times q}$. Then

$$\mathbf{A}_h = I \otimes T_N + T_N \otimes I = \begin{pmatrix} T_N & & \\ & T_N & \\ & & \ddots \end{pmatrix} + \begin{pmatrix} 2I & -I & \\ -I & 2I & \ddots \\ & \ddots & \ddots \end{pmatrix}.$$

Related to the Kronecker product is the so called **Vec** operator, in the following defined as

$$\mathbf{Vec} : \mathbb{R}^{m\times n} \to \mathbb{R}^{mn},\ \mathbf{Vec}(A) := \begin{pmatrix} \mathbf{a_1^T} \\ \mathbf{a_2^T} \\ \vdots \\ \mathbf{a_m^T} \end{pmatrix} \quad \mathbf{a_k} = (a_{k,1}, \dots, a_{k,n}).$$

Operator **Vec** stacks the rows of a matrix. This conforms to the numbering chosen above to represent grid functions as vectors, such that

$$\mathbf{Vec}(U) = u_{0,h}.$$

The equivalence of (6.45) and (6.50) follows from the equivalence (see [32])

$$AXB = C \quad \Longleftrightarrow \quad (A \otimes B^T)\mathbf{Vec}(X) = \mathbf{Vec}(C).$$

Applying the factorization $T_N = V^{-1}\Sigma V$ to (6.50)

$$\begin{aligned} UT_N + T_N U = h^2 Q \quad &\Leftrightarrow \quad UV^{-1}\Sigma V + V^{-1}\Sigma V U = h^2 Q \\ &\Leftrightarrow \quad VUV^{-1}\Sigma + \Sigma VUV^{-1} = h^2 VQV^{-1} \\ &\Leftrightarrow \quad \widehat{U}\Sigma + \Sigma\widehat{U} = h^2\widehat{Q}, \quad \widehat{U} = VUV^{-1},\ \widehat{Q} = VQV^{-1} \end{aligned}$$

allows for stating the following Algorithm 6.5 for the computation of U.

Algorithm 6.5 (Fast Poisson Solver: Two-Dimensional Dirichlet-Poisson Problem).

1. *Compute $\widehat{Q} = VQV^{-1} = \frac{2}{N+1}VQV$:*
   Components of $\widehat{Q} = (\hat{q}_{i,j})$ are determined by

   $$\hat{q}_{i,j} = \frac{2}{N+1}\sum_{k=1}^{N}\sum_{l=1}^{N} q_{i,j} \sin\left(\frac{ik\pi}{N+1}\right) \sin\left(\frac{jl\pi}{N+1}\right).$$

   This corresponds to a two-dimensional DST of Q and can be computed with $\mathcal{O}(N^2 \log N)$ operations.

2. *Solve $\widehat{U}\Sigma + \Sigma\widehat{U} = h^2\widehat{Q}$:*
   Components of $\widehat{U} = (\hat{u}_{i,j})$ can be computed by

   $$\hat{u}_{i,j} = \frac{\hat{q}_{i,j}}{(\lambda_i + \lambda_j)} = \frac{\hat{q}_{i,j}}{\left(2 - 2\cos\left(\frac{i\pi}{N+1}\right)\right) + \left(2 - 2\cos\left(\frac{j\pi}{N+1}\right)\right)}$$

   in N^2 operations.

3. *Compute $U = V^{-1}\widehat{U}V = \frac{2}{N+1}VUV$:*
   Again the solution can be obtained through a two-dimensional DST in $\mathcal{O}(N^2 \log N)$ operations as

   $$u_{i,j} = \frac{2}{N+1}\sum_{k=1}^{N}\sum_{l=1}^{N} \hat{u}_{i,j} \sin\left(\frac{ik\pi}{N+1}\right) \sin\left(\frac{jl\pi}{N+1}\right).$$

Using the two-dimensional DST provided by the FFTW library the Fast Poisson Solver is implemented by functions:

```
template <typename U>
    fftw_plan
    dp2d_fastpoissonsolver_init(int rh, GeMatrix<U> &u,
                                unsigned fftw_flags = FFTW_ESTIMATE);

template <typename F, typename U>
    void
    dp2d_fastpoissonsolver(int rh, fftw_plan &plan,
                           const GeMatrix<F> &f, GeMatrix<U> &u);
```

Listing 6.24: Fast Poisson Solver for the two-dimensional Dirichlet-Poisson problem.

Like for the one-dimensional Fast Poisson Solver one has to take into account that the FFTW does not apply any scaling on the transform. Therefore Algorithm 6.5 needs to be adapted accordingly. Details about scaling factors can be found in the FFTW documentation[30].

6.5.3 Stationary Iterative Methods

Low-level implementations of the stationary iterative methods are embedded into FLENS following the same pattern illustrated in Section 6.3.4. This allows that each low-level implementation is immediately available as a building block for the multigrid method on an abstract level: For instance, as a smoother

```
Smoother  S(A, f);
u = S*u;
```

or as a solver

```
typedef StationaryIterativeSolver<Smoother>  Solver;
Solver svr(A, f, u);
svr.solve();
```

What concrete method gets applied can be specified through typedefs: For instance, the Gauss-Seidel method by

```
typedef GaussSeidel<DirichletPoisson2D, GridVector2D>  Smoother;
```

Jacobi:	$u_{i,j}^{(n+1)} = \frac{1}{4}\big(u_{i-1,j}^{(n)} + u_{i+1,j}^{(n)} + u_{i,j-1}^{(n)} + u_{i,j+1}^{(n)} + h^2 f_{i,j}\big)$
Weighted Jacobi:	$u_{i,j}^{(n+1)} = (1-\omega)u_{i,j}^{(n)} + \frac{\omega}{4}\big(u_{i-1,j}^{(n)} + u_{i+1,j}^{(n)} + u_{i,j-1}^{(n)} + u_{i,j+1}^{(n)} + h^2 f_{i,j}\big)$
Gauss-Seidel:	$u_{i,j}^{(n+1)} = \frac{1}{4}\big(u_{i-1,j}^{(n+1)} + u_{i+1,j}^{(n)} + u_{i,j-1}^{(n+1)} + u_{i,j+1}^{(n)} + h^2 f_{i,j}\big)$

Table 6.1: Stationary iterative methods for the two-dimensional Dirichlet-Poisson problem.

Jacobi and Gauss-Seidel Method

From the representation of A_h given in (6.5.2) it follows that eigenvectors and eigenvalues are related to T_N. Let $\lambda_1, \dots, \lambda_N$ and $v_1, \dots, v_N$ denote eigenvalues and vectors of T_N respectively, then

$$\begin{aligned} h^2 A_h \mathbf{Vec}(v_k v_l^T) &= \mathbf{Vec}\left(v_k v_l^T T_N + T_N v_k v_l^T\right) = \mathbf{Vec}\left(\lambda_l v_k v_l^T + \lambda_k v_k v_l^T\right) \\ &= (\lambda_l + \lambda_k)\mathbf{Vec}\left(v_k v_l^T\right). \end{aligned}$$

Convergence of the Gauss-Seidel method is therefore guaranteed since A_h is positive definite. For the Jacobi method it follows that eigenvectors of J_{A_h} are $\mathbf{Vec}(v_k v_l^T)$ and corresponding eigenvalues are given by

$$\lambda_{k,l}(J_{A_h}) = 1 - \frac{\lambda_{k,l}(h^2 A_h)}{4} = 1 - \frac{\lambda_l + \lambda_k}{4} = \frac{1}{2}\left(\cos\left(\frac{l\pi}{N+1}\right) + \cos\left(\frac{k\pi}{N+1}\right)\right).$$

Therefore smoothing properties observed in Section 6.3.1 can be shown analogously for the two-dimensional case. Table 6.1 summarizes methods introduced in Section 6.3 in component form. These representations easily allow to derive implementations adapted to the two-dimensional Poisson problem. An implementation of the Gauss-Seidel method is given in Listing 6.25.

```
template <typename F, typename U>
void
dp2d_gauss_seidel(int rh, const GeMatrix<F> &f, GeMatrix<U> &u)
{
    int N = rh-1;
    int rhh = rh*rh;
    double hh = 1./rhh;

    for (int i=1; i<=N; ++i) {
        for (int j=1; j<=N; ++j) {
            u(i,j) = 0.25*(u(i-1,j)+u(i+1,j)+u(i,j-1)+u(i,j+1)+hh*f(i,j));
        }
    }
}
```

Listing 6.25: Gauss-Seidel-iteration for the two-dimensional Dirichlet-Poisson problem.

Red-Black Gauss-Seidel

As can be seen in Table 6.1 the Gauss-Seidel method requires to compute new components of the iterate in the same order grid points were numbered. Choosing a different numbering scheme leads to variants like the *red-black Gauss-Seidel* method introduced in the following. Figure 6.9 illustrates a grid colored with alternate red and black points. This coloring scheme can be specified by:

$$\text{Denote grid point } (x_i, y_j) \text{ as } \begin{cases} \text{‘red’} & \text{if } i+j \text{ even,} \\ \text{‘black’} & \text{else.} \end{cases}$$

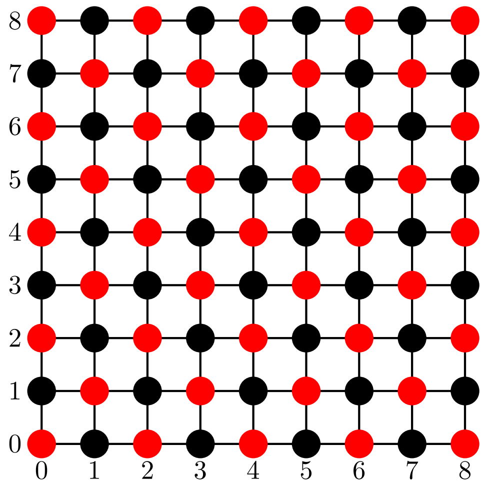


Figure 6.9: Red-Black ordering on $\bar{\Omega}_h$.

Numbering red points first and black points subsequently the component form of the Gauss-Seidel method changes to

$$u_{i,j}^{(n+1)} = \begin{cases} \frac{1}{4}\left(u_{i-1,j}^{(n)} + u_{i+1,j}^{(n)} + u_{i,j-1}^{(n)} + u_{i,j+1}^{(n)} + h^2 f_{i,j}\right), & \text{if } i+j \text{ even,} \\ \frac{1}{4}\left(u_{i-1,j}^{(n+1)} + u_{i+1,j}^{(n+1)} + u_{i,j-1}^{(n+1)} + u_{i,j+1}^{(n+1)} + h^2 f_{i,j}\right), & \text{else.} \end{cases}$$

This is implemented in two separate functions that need to be called consecutively. If this should be inconvenient a third function could bundle these calls. However splitting the functionality for the implementation provides a flexibility that simplifies parallelization as shown in the next section. Function `dp2d_gauss_seidel_red` in Listing 6.26 computes new components at all red points. Obviously these 'red components' can be computed in arbitrary order. This allows for a simple implementation using two for-loop blocks (lines 9-13 and 14-18). Each block uses increments of two but different starting points:

```
 1 template <typename F, typename U>
 2 void
 3 dp2d_gauss_seidel_red(int rh, const GeMatrix<F> &f, GeMatrix<U> &u)
 4 {
 5     int N = rh-1;
 6     int rhh = rh*rh;
 7     double hh = 1./rhh;
 8 
 9     // iterate over red points
10     for (int i=1; i<=N; i+=2) {
11         for (int j=1; j<=N; j+=2) {
12             u(i,j) = 0.25*(u(i-1,j)+u(i+1,j)+u(i,j-1)+u(i,j+1)+hh*f(i,j));
13         }
14     }
15     for (int i=2; i<=N; i+=2) {
16         for (int j=2; j<=N; j+=2) {
17             u(i,j) = 0.25*(u(i-1,j)+u(i+1,j)+u(i,j-1)+u(i,j+1)+hh*f(i,j));
18         }
19     }
20 }
```

Listing 6.26: Red-Black Gauss-Seidel-iteration for the two-dimensional Dirichlet-Poisson problem.

Analogously function `dp2d_gauss_seidel_black` treats black points.

Parallelization of the Gauss-Seidel method is a major application for the red-black variant. On a multiprocessor system with shared memory this is obvious. As components at red points can be updated in an arbitrary order this can be done in parallel and subsequently components at black points. For systems with distributed memory parallelization can be achieved by domain decomposition as discussed in Section 6.6. A further notable property of this method is that the residual vanishes at the black points. Using the red-black numbering the system (6.43)-(6.44) can be rewritten as

$$\begin{pmatrix} D_r & E_{r,b} \\ E_{b,r} & D_b \end{pmatrix} \begin{pmatrix} u_r \\ u_b \end{pmatrix} = \begin{pmatrix} q_r \\ q_b \end{pmatrix}. \tag{6.51}$$

with diagonal matrices D_r and D_b. The Gauss-Seidel method in matrix form is then given by

$$u_r^{(n+1)} = D_r^{-1}\left(q_r - E_{r,b}u_b^{(n)}\right), \tag{6.52}$$

$$u_b^{(n+1)} = D_b^{-1}\left(q_b - E_{b,r}u_r^{(n+1)}\right). \tag{6.53}$$

For the residual at black points it follows that

$$\begin{aligned} r_b^{(n+1)} &= q_b - E_{b,r}u_r^{(n+1)} - D_b u_b^{(n+1)} \\ &= q_b - E_{b,r}D_r^{-1}\left(q_r - E_{r,b}u_b^{(n)}\right) - D_b D_b^{-1}\left(q_b - E_{b,r}u_r^{(n+1)}\right) \\ &= -E_{b,r}D_r^{-1}\left(q_r - E_{r,b}u_b^{(n)}\right) + E_{b,r}u_r^{(n+1)} \\ &= -E_{b,r}D_r^{-1}\left(q_r - E_{r,b}u_b^{(n)}\right) + E_{b,r}D_r^{-1}\left(q_r - E_{r,b}u_b^{(n)}\right) = 0. \end{aligned}$$

A dedicated function `dp2d_residual_red` computing the residual only on red points can take advantage of this.

6.5.4 Multigrid Method

Components that remain to be provided for the multigrid method are restriction and prolongation operators. These are provided in the following form:

```
Restriction    R;
Prolonagtion   P;
GridVector2D   v(rh), vc(rh/2);

vc = R*v;
v += P*vc;
```

Different variants of restriction and prolongation can be provided through additional classes in a straightforward manner.

Prolongation Operator

Using linear interpolation, the prolongation $v_h = P\tilde{v}_{2h}$ is defined by

$$\begin{aligned} v_{2i,2j} &:= \tilde{v}_{i,j} && 1 \le i,j \le n = \tfrac{1}{2h} - 1, \\ v_{2i+1,2j} &:= \tfrac{1}{2}(\tilde{v}_{i,j} + \tilde{v}_{i+1,j}) && 1 \le i \le n-1,\ 1 \le j \le n, \\ v_{2i,2j+1} &:= \tfrac{1}{2}(\tilde{v}_{i,j} + \tilde{v}_{i,j+1}) && 1 \le i \le n,\ 1 \le j \le n-1, \\ v_{2i+1,2j+1} &:= \tfrac{1}{4}(\tilde{v}_{i,j} + \tilde{v}_{i+1,j+1}) && 1 \le i \le n-1,\ 1 \le j \le n-1. \end{aligned}$$

For the update operation $v_h \to v_h + P_{2h}^h \tilde{v}_{2h}$ the contributions of $\tilde{v}_{i,j}$ to values of v_h are illustrated in Figure 6.10. An implementation based on this stencil for the prolongation operator is shown in Listing 6.27.

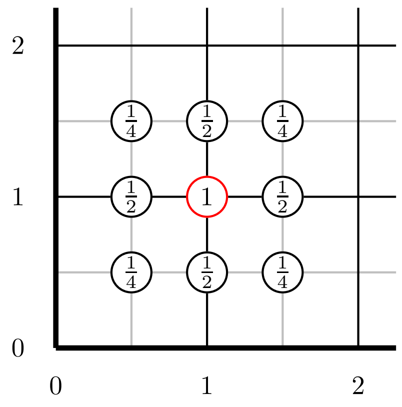


Figure 6.10: Stencils for the prolongation. The red node in the middle contributes to circled nodes on the fine grid.

```
template <typename VC, typename V>
void
dp2d_prolongation(const GeMatrix<VC> &vc, GeMatrix<V> &v)
{
    int N = vc.lastRow()-1;
    for (int i=1, I=2; i<=N; ++i, I+=2) {
        for (int j=1, J=2; j<=N; ++j, J+=2) {
            v(I-1,J-1)+=.25*vc(i,j); v(I-1,J)+=.5*vc(i,j); v(I-1,J+1)+=.25*vc(i,j);
            v(I  ,J-1)+=.50*vc(i,j); v(I  ,J)+=   vc(i,j); v(I  ,J+1)+=.50*vc(i,j);
            v(I+1,J-1)+=.25*vc(i,j); v(I+1,J)+=.5*vc(i,j); v(I+1,J+1)+=.25*vc(i,j);
        }
    }
}
```

Listing 6.27: Prolongation in two dimensions.

Restriction Operators

The simplest form of restriction $\tilde{v}_{2h} = R_h^{2h} v_h$ is defined by the injection

$$\tilde{v}_{i,j} = v_{2i,2j}, \quad 1 \le i, j \le n = \frac{1}{2h} - 1.$$

Weighting the restriction according to

$$\tilde{v}_{i,j} = \frac{1}{8}(4v_{2i,2j} + v_{2i+1,2j} + v_{2i-1,2j} + v_{2i,2j+1} + v_{2i,2j-1}), \quad 1 \le i, j \le n,$$

is denoted as *half-weighted restriction*. This term reflects that only four neighboring points of (x_{2i}, y_{2j}) contribute to $\tilde{v}_{i,j}$ as illustrated in Figure 6.11. A simple implementation of the half-weighted restriction is shown in Listing 6.28.

```
template <typename U, typename F, typename R>
void
dp2d_restriction_hw(const GeMatrix<V> &v, GeMatrix<VC> &vc)
{
    int N = vc.lastRow()-1;
    for (int i=1, I=2; i<=N; ++i, I+=2) {
        for (int j=1, J=2; j<=N; ++j, J+=2) {
            vc(i,j) = (4*v(I,J)+v(I-1,J)+v(I,J-1)+v(I,J+1)+v(I+1,J))/8;
        }
    }
}
```

Listing 6.28: Half-weighted restriction in two dimensions.

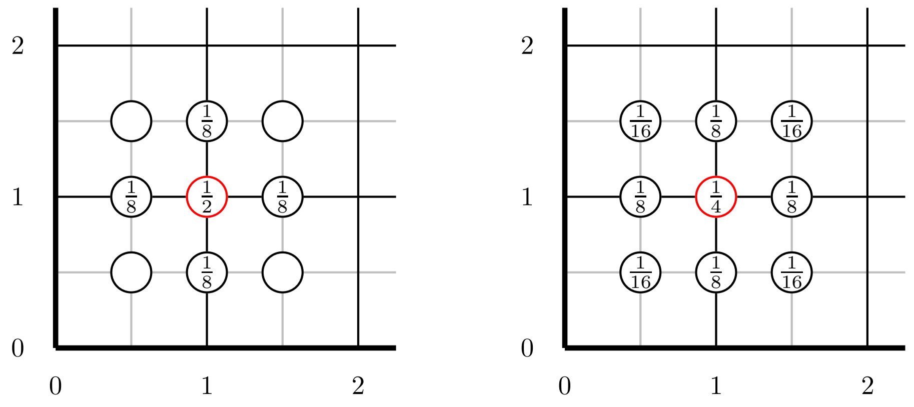


Figure 6.11: Stencils for the half-weighted restriction (left) and the full-weighted restriction (right). Circled nodes on the fine grid contribute to the red node in the middle.

Taking all eight neighboring points into account leads to the *full-weighted restriction* (Figure 6.11) defined by

$$\begin{aligned}\tilde{v}_{i,j} \;=\; \tfrac{1}{16}\; & \big(4v_{2i,2j} \\ & +2(v_{2i+1,2j} + v_{2i-1,2j} + v_{2i,2j+1} + v_{2i,2j-1}) \\ & +v_{2i+1,2j+1} + v_{2i-1,2j+1} + v_{2i+1,2j-1} + v_{2i-1,2j-1}\big), \quad 1 \le i,j \le n.\end{aligned}$$

Usually better convergence rates are achieved through the full-weighted restriction but requires more floating point operations. Using the red-black Gauss-Seidel method as smoother no additional operations are required compared to the half-weighted restrictions. Adapted implementations can take into account that the residual vanishes on black points after a smoothing operation.

6.6 Parallel Implementation of the Multigrid Method

This section describes the parallel implementation of the multigrid method on a cluster, that is, a group of computers interconnected via network. Each computer (in the following denoted as processor node or simply processor) in the cluster has its own memory and one or more processors. Up-to-date benchmarks demonstrating the scalability of the introduced parallel multigrid implementation can be found on the FLENS website [45].

The parallel implementation uses the *Single Program, Multiple Data (SPMD)* technique. Data interchanges between processor nodes are provided by the network. This data communication between these nodes is organized using the Message Passing Interface (MPI). A kind introduction and comprehensive reference of MPI is given in [35]. Concepts for the realization of parallel numerical algorithms are illustrated in much more detail in [2] and [67].

6.6.1 Processor Topology and Domain Decomposition

Domain Ω is decomposed and distributed across $p = p_1 p_2$ processor nodes arranged in an $p_1 \times p_2$ Cartesian topology. To be more precise, the domain gets subdivided into local domains

$$\Omega^{(k,l)} = \Omega \cap \left[\frac{k}{p_1}, \frac{k+1}{p_1}\right) \times \left[\frac{l}{p_2}, \frac{l+1}{p_2}\right), \quad 0 \le k < p_1,\ 0 \le l < p_2. \tag{6.54}$$

and processor node (k,l) computes the unknowns on the local grid points

$$\Omega_h^{(k,l)} = \Omega^{(k,l)} \cap \Omega_h. \tag{6.55}$$

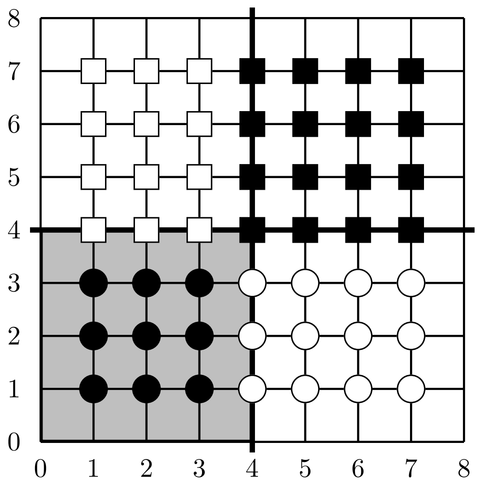


Figure 6.12: Decomposition of Ω_h into the four subdomains $\Omega^{(0,0)}$ (shaded gray), $\Omega^{(1,0)}$, $\Omega^{(0,1)}$ and $\Omega^{(1,1)}$. Symbols ●, ❍, □ and ■ indicate unknowns assigned to the different processes.

For $N+1 = 8$ and $p_1 = p_2 = 2$ Figure 6.12 illustrates the subdivision and indicates how grid points are assigned to different processor nodes. In the following it is assumed that the number of available processors are of the form $p_1 = 2^{l_1}, p_2 = 2^{l_2}$ and the grid size h chosen such that $N+1 = 2^L$ where $l_1, l_2 \leq L$. Local grid points assigned to process node (k,l) are then given by

$$\Omega_h^{(k,l)} = \left\{ \left(\frac{k}{p_1} + ih, \frac{l}{p_2} + jh \right) \;\middle|\; i = \delta_{k,0}, \ldots, \frac{N+1}{p1} - 1,\ j = \delta_{l,0}, \ldots, \frac{N+1}{p2} - 1 \right\}. \tag{6.56}$$

6.6.2 Local Grids with Ghost Nodes

Assume that the red-black Gauss-Seidel method is used as smoother in the multigrid method. Due to the structure of A_h, updating the iterate at the grid point (x_i, y_i) requires that values on the grid points

$$N_5(i,j) = \{(x_i, y_j), (x_{i+1}, y_j), (x_{i-1}, y_j), (x_i, x_{j+1}), (x_i, y_{j-1})\}$$

are available. The prolongation $v_h = P_{2h}^h \tilde{v}_{2h}$ requires for the computation of $v_{i,j}$ that values of $\tilde{v}_{2h}$ are available on grid points

$$N_9(i,j) = N_5(i,j) \cap \{(x_{i+1}, y_{j+1}), (x_{i-1}, y_{j+1}), (i-1, j+1), (i-1, j-1)\}.$$

The same applies to the full-weighted restriction. Hence the computation of unknowns on $\Omega_h^{(k,l)}$ requires that process node (k,l) has read access to additional nodes that are assigned to different local grids. These additional nodes are denoted as *ghost nodes.* For a subdivision into four local grids Figure 6.13 illustrates ghost nodes needed by each processor. Local grids extended by required ghost nodes are formally specified by

$$\hat{\Omega}_h^{(k,l)} = \left\{ \left(\frac{k}{p_1} + ih, \frac{l}{p_2} + jh \right) \;\middle|\; i = i_0^{(k)}, \ldots, i_1^{(k)},\ j = j_0^{(l)}, \ldots, j_1^{(l)} \right\}, \tag{6.57}$$

where

$$i_0^{(k)} = \delta_{k,0} - 1, \quad i_1^{(k)} = \frac{N+1}{p1} - \delta_{k,N+1}, \tag{6.58}$$

and

$$j_0^{(k)} = \delta_{l,0} - 1, \quad j_1^{(l)} = \frac{N+1}{p2} - \delta_{l,N+1}. \tag{6.59}$$

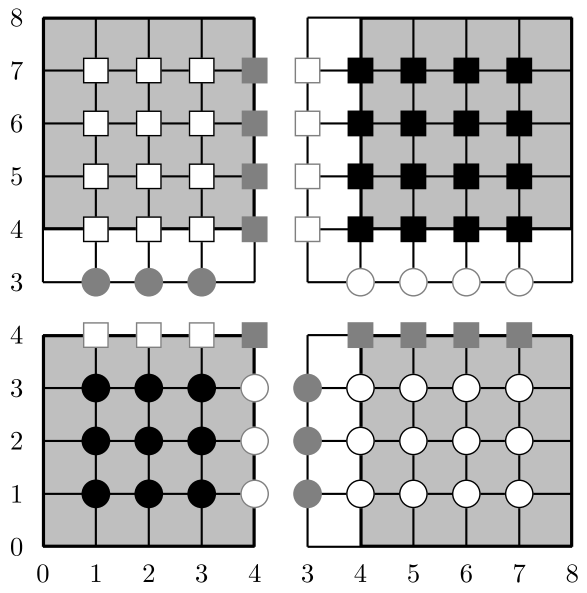


Figure 6.13: Decomposition of Ω_h. Ghost nodes required on local grids are indicated by grayed out nodes.

In each read access to a ghost node it has to be guaranteed that the value coincides with that of the original node. This requires communication between the processors in the form of data exchange. For parallel implementations the overhead due to interprocess communication is a crucial issue with respect to scalability. Obviously the amount of overhead not only depends on the amount of data that needs to be send and received. But it also depends on the number of processors involved in updating ghost nodes of a particular local grid. As can be seen from Figure 6.13 process node (k, l) needs to receive values on ghost nodes from five processors[11]:

Ghost nodes of process node (k, l)	receive update from process node
$(x_{i_0+1}, y_{j_0}), \ldots, (x_{i_1-1}, y_{j_0})$	south process node: $(k, l-1)$
$(x_{i_0+1}, y_{j_1}), \ldots, (x_{i_1-1}, y_{j_1})$	north process node: $(k, l+1)$
$(x_{i_0}, y_{j_0+1}), \ldots, (x_{i_0}, y_{j_1-1})$	west process node: $(k-1, l)$
$(x_{i_1}, y_{j_0+1}), \ldots, (x_{i_1}, y_{j_1-1})$	east process node: $(k+1, l)$
(x_{i_1}, y_{j_1})	north-east process node: $(k+1, l+1)$

Direct communication with the north-east process node $(k+1, l+1)$ for a single value can be avoided. Note that ghost node (x_{i_1}, y_{j_1}) on processor (k, l) as well as ghost node (x_{i_0+1}, y_{j_1}) on processor $(k+1, l)$ refer both to the same grid point (x_{i_0}, y_{j_0}) on processor $(k+1, l+1)$. This suggests that processor (k, l) receives this ghost node indirectly from the east process node as shown in Table 6.2. Complementary Table 6.3 lists what each process node needs to send in this case.

In general a guarantee that process (k, l) receives the correct value from $(k+1, l+1)$ requires a strict coordination of the interprocess communication. Processor $(k+1, l)$ first needs to receive the value from $(k+1, l+1)$ before sending it to processor (k, l). Such a synchronization in turn can have a notable negative impact on performance.

However, for the parallel implementation of the multigrid this type of synchronization can be avoided in some cases. Using a smoother like the red-black Gauss-Seidel method in each iteration

[11]Cases where process nodes do not exist in certain directions are ignored. Indices i_0, i_1, j_0, j_1 refer to the local grid $\hat{\Omega}_h^{(k,l)}$.

Ghost nodes of process node (k,l)	receive update from process node
$(x_{i_0+1}, y_{j_0}), \dots, (x_{i_1-1}, y_{j_0})$	south process node: $(k, l-1)$
$(x_{i_0+1}, y_{j_1}), \dots, (x_{i_1-1}, y_{j_1})$	north process node: $(k, l+1)$
$(x_{i_0}, y_{j_0+1}), \dots, (x_{i_0}, y_{j_1-1})$	west process node: $(k-1, l)$
$(x_{i_1}, y_{j_0+1}), \dots, (x_{i_1}, y_{j_1-1}), (x_{i_1}, y_{j_1})$	east process node: $(k+1, l)$

Table 6.2: Receiving updates of ghost nodes.

From process node (k,l) send values on grid points	receiving process node
$(x_{i_0+1}, y_{j_0+1}), \dots, (x_{i_1-1}, y_{j_0+1})$	south process node: $(k, l-1)$
$(x_{i_0+1}, y_{j_1-1}), \dots, (x_{i_1-1}, y_{j_1-1})$	north process node: $(k, l+1)$
$(x_{i_0+1}, y_{j_0+1}), \dots, (x_{i_0+1}, y_{j_1-1}), (x_{i_0+1}, y_{j_1})$	west process node: $(k-1, l)$
$(x_{i_1-1}, y_{j_0+1}), \dots, (x_{i_1-1}, y_{j_1-1})$	east process node: $(k+1, l)$

Table 6.3: Sending updates of boundary nodes.

ghost nodes need to be updated twice. Without synchronization updating the ghost points twice guarantees that process (k,l) has received the correct value. Further, the only operations that require an up-to-date value in (x_{i_1}, y_{j_1}) are the restriction and prolongation operators that are performed after a smoothing step.

6.6.3 Implementation of a Distributed Grid Vector

Using the MPI library process nodes are identified by ranks, which are integers within the range from 0 to $p-1$. Processors can be organized logically in a $p_1 \times p_2$ Cartesian topology using the MPI functions `MPI_Cart_create`, `MPI_Cart_coords` and `MPI_Cart_shift`. Listing 6.29 shows the wrapper class `MpiCart` hiding the technical details.

```
enum CartDir { North=0, East=1, South=2, West=3 };

class MpiCart
{
    public:
        MpiCart();

        MpiCart(int argc, char **argv);

        MpiCart(const MpiCart &rhs);

        MPI::Cartcomm comm;
        int numRows, numCols;
        int row, col;
        int neighbors[4];
};
```

Listing 6.29: Abstraction of the Cartesian processor topology.

The meaning of the member variables is described in Table 6.4.

The number of total process nodes and the logical layout of the processor grid can be specified through parameters on the command line, e.g.

```
mpirun -np 32 ./myApp 4 8
```

`comm`	MPI communication object used for interprocess communication.
`(row, col)`	Position within the processor topology. `row` and `col` are within the range $0, \dots, p_1 - 1$ and $0, \dots, p_2 - 1$ respectively.
`numRows, numCols`	Number of rows and columns in the processor grid, i. e. p_1 and p_2 respectively.
`neighbors`	Ranks of neighboring process nodes. E. g. `neighbors[East]` is the rank of the east node.

Table 6.4: Attributes of `MpiCart` objects.

starts the application `myApp` on 32 process nodes organized in a 4×8 grid. Arguments from the command line are handled during the construction of the `MpiCart` object. The `MpiCart` object needs to be created before any other MPI function gets called:

```
int
main(int argc, char **argv)
{
    MpiCart mpiCart(argc, argv);

    // ..
}
```

Class `DistributedGridVector2D` shown in Listing 6.32 implements grid functions with values distributed across a processor grid. Each instance holds values on a local grid $\hat{\Omega}_h^{(k,l)}$ defined in (6.57). These values are stored in a local matrix `grid`. For a simple and efficient implementation it is of advantage that all local `grid` matrices have the same size independent of the process node. Therefore the index range of `grid` is always $[-1, \dots, (N+1)/p_1] \times [-1, \dots, (N+1)/p_2]$. Values on $\hat{\Omega}_h^{(k,l)}$ are stored in the sub-matrix indexed by $[i_0, \dots, i_1] \times [j_0, \dots, j_1]$ where i_0, i_1 and j_0, j_1 are initialized according to (6.58) and (6.59) respectively. An instance of `mpiCart` is used for the interprocess communication. Method `setGhostNodes` updates the ghost nodes and uses therefore the user defined MPI data types `MpiRow` and `MpiCol`. The technique realized for updating ghost nodes is described in more detail below.

```
class DistributedGridVector2D
    : public Vector<DistributedGridVector2D>
{
    public:
        typedef GeMatrix<FullStorage<double, RowMajor> > Grid;

        DistributedGridVector2D(const MpiCart &_mpiCart, int rh);

        // ...

        void
        setGhostNodes();

        MpiCart          mpiCart;
        int              rh, N;
        int              i0, i1, j0, j1;
        Grid             grid;
        MPI::Datatype    MpiRow, MpiCol;
};
```

Listing 6.30: Grid vector distributed accross the processor grid.

Data exchange between west and east process nodes is illustrated in Figure 6.14 and between north and south nodes in Figure 6.15. Comparing these figures with Table 6.3 and Table 6.2 obviously more data gets exchanged than actually required. With respect to performance this is negligible but simplifies the implementation. For sending and receiving rows and columns it is convenient to define appropriate user defined MPI data types. As the local grid is stored row-wise, for rows such a type can be defined through

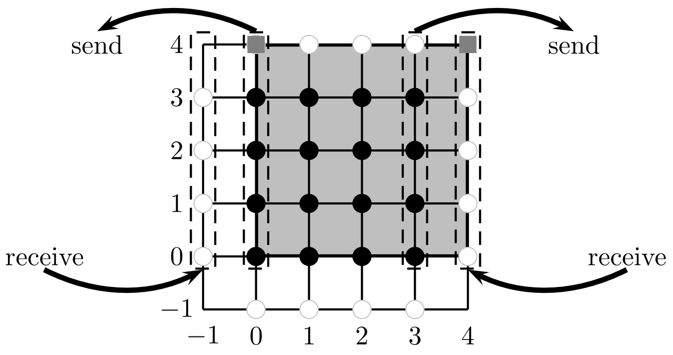


Figure 6.14: Exchange of ghost nodes between processors in west and east direction.

```
MpiRow = MPI::DOUBLE.Create_contiguous(i1-i0-1);
MpiRow.Commit();
```

This specifies that `MpiRow` is a data type consisting of $i_1 - i_0 - 1$ elements of type `double` located contiguously in memory. Analogously `MpiCol` defined by

```
MpiCol = MPI::DOUBLE.Create_vector(j1-j0, 1, grid.leadingDimension());
MpiCol.Commit();
```

refers to $j_1 - j_0$ elements of type `double` with stride equal to the leading dimension of `grid`. The implementation of the method `setGhostNodes` is shown in Listing 6.32. For sending (lines 29-32) and receiving (34-37) non-blocking MPI functions are chosen. This enhances performance and prevents deadlocks. The underlying MPI implementation can carry out the data exchange in the most favorable order. The method suspends execution (line 39) until the data exchange is completed.

```
20 void
21 DistributedGridVector2D::setGhostNodes()
22 {
23     MPI::Request  req[8];
24     MPI::Status   stat[8];
25
26     const int *neighbors = mpiCart.neighbors;
27     const int m = i1-i0-2;
28     const int n = j1-j0-2;
29     req[0] = mpiCart.comm.Isend(&grid(0,n), 1, MpiCol, neighbors[East],  0);
30     req[1] = mpiCart.comm.Isend(&grid(0,0), 1, MpiRow, neighbors[South], 0);
31     req[2] = mpiCart.comm.Isend(&grid(0,0), 1, MpiCol, neighbors[West],  0);
32     req[3] = mpiCart.comm.Isend(&grid(m,0), 1, MpiRow, neighbors[North], 0);
33
34     req[4] = mpiCart.comm.Irecv(&grid(  0,n+1), 1, MpiCol, neighbors[East],  0);
35     req[5] = mpiCart.comm.Irecv(&grid( -1,  0), 1, MpiRow, neighbors[South], 0);
36     req[6] = mpiCart.comm.Irecv(&grid(  0, -1), 1, MpiCol, neighbors[West],  0);
37     req[7] = mpiCart.comm.Irecv(&grid(m+1,  0), 1, MpiRow, neighbors[North], 0);
38
39     MPI::Request::Waitall(8, req, stat);
40 }
```

Listing 6.31: Grid vector distributed over the processor grid.

Listing 6.32 shows the implementation of the discrete L_2 norm for a distributed grid vector. Each process node first computes the square of its local norm, i. e. $||v_h|_{\Omega_h^{(k,l)}}||_h^2$ (line 5). Then results from all process nodes are summed up (line 8) and the square root returned (line 9).

```
41 double
42 normL2(const DistributedGridVector2D &v)
```

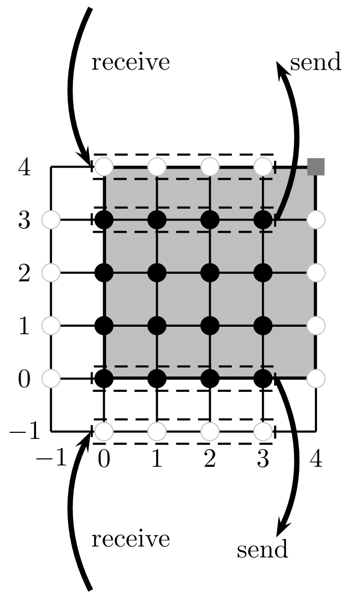


Figure 6.15: Exchange of ghost nodes between processors in north and south direction.

```
{
    double h = 1./v.rh;
    double r = dp2d_norm2sqr(v.grid);

    double R;
    v.mpiCart.comm.Allreduce(&r, &R, 1, MPI::DOUBLE, MPI::SUM);
    return h*sqrt(R);
}
```

Listing 6.32: Norm of a distributed grid vector.

6.6.4 Direct Solvers

Parallel implementations of the Cholesky factorization is provided by the ScaLAPACK Library [59] and for the DST for example by the FFTW library. Based on these implementations, direct solvers can be integrated as illustrated in Section 6.2 and Section 6.5.2. However, it is notable that both solvers require a processor grid of the form $1 \times p$. Implementation of these solvers are in the following not further considered as this merely exhibits technical details of these libraries.

6.6.5 Red-Black Gauss-Seidel

An implementation of the red-black Gauss-Seidel method for distributed grid vectors can be realized be reusing the non-parallel implementation as illustrated in Listing 6.33.

```
void
mv(Transpose trans, double alpha,
   const GaussSeidelRedBlack<DirichletPoisson2D, DistributedGridVector2D> &GS,
   const DistributedGridVector2D &u_1, double beta, DistributedGridVector2D &u)
{
    dp2d_gauss_seidel_red(GS.A.rh, GS.f.grid, u.grid);
    u.setGhostNodes();
    dp2d_gauss_seidel_black(GS.A.rh, GS.f.grid, u.grid);
    u.setGhostNodes();
}
```

Listing 6.33: Grid vector distributed over the processor grid.

After applying the Gauss-Seidel method on red and black points the ghost nodes need to be updated, respectively. However, this implementation requires some minor modification of the implementation introduced in Listing 6.26. For the sake of simplicity, the implementation of `dp2d_gauss_seidel_red` assumes that indices start at 1 and further that the grid is square. As the same kind of modification is required to solve the Dirichlet-Poisson on a non-square domain $\Omega = (a, b) \times (c, d)$ this should not be considered as a conceptual issue.

6.6.6 Restriction and Prolongation

In the multigrid method the restriction operator is only applied when ghost nodes are up-to-date therefore no interprocess communication is required:

```
void
mv(Transpose trans, double alpha, const Restriction &R,
   const DistributedGridVector2D &v, double beta, DistributedGridVector2D &vc)
{
    dp2d_restriction_hw(v.grid, vc.grid);
}
```

Listing 6.34: Restriction operator for distributed grid vectors.

The implementation of the prolongation operator in Listing 6.27 requires that for a grid point all eight neighboring grid points are available. As can be seen from Figure 6.12 for local grids this is not the case at boundaries on the top and right side. Providing additional implementations for these cases still allows for reusing the non-parallel implementation `dp2d_prolongation`.

```
void
mv(Transpose trans, double alpha, const Prolongation &P,
   const DistributedGridVector2D &vc, double beta, DistributedGridVector2D &v)
{
    dp2d_prolongation(vc.grid, v.grid);
    dp2d_prolongation_north(vc.grid, v.grid);
    dp2d_prolongation_east(vc.grid, v.grid);
    dp2d_prolongation_north_east(vc.grid, v.grid);
    v.setGhostNodes();
}
```

Listing 6.35: Prolongation operator for distributed grid vectors.

6.7 Conjugated Gradient Method

Due to the sparsity of the coefficient matrix the conjugated gradient method is another favorable solver for the Dirichlet-Poisson problem. In order to use the implementation introduced in Section 6.7, merely specialized matrix-vector products have to be provided. Such specializations can be realized analogously to the functions computing the residual[12]. The parallelization of the cg-method immediately follows from the parallelization of the matrix-vector product. Thus, like the generic implementation of the multigrid method only one implementation of the cg-method is sufficient for solving the Dirichlet-Poisson problem in one and two space dimensions either serial or parallel.

Stationary iterative solvers can be integrated in FLENS in order to serve as preconditioners. For the initial guess $u^{(0)} \equiv 0$ applying one step of the Jacobi method in order to solve $Au = q$ leads to

$$u^{(1)} = -D^{-1}(L + U)u^{(0)} + D^{-1}q = D^{-1}q, \tag{6.60}$$

[12]Actually the residual functions could have been implemented based on implementations for the matrix-vector product. In this case the cg-method could be used out of the box. This approach was not followed in this chapter for the sake of demonstrating that optimized functions for computing the residual can be used.

such that $D^{-1} = \mathrm{diag}(A)^{-1}$ can be considered as an approximation of A^{-1}. This gives the Jacobi preconditioner introduced in Section 5.2. As was illustrated in Chapter 5 the preconditioner can be easily integrated into FLENS allowing the following usage of the pcg-method implementation:

```
// ...
JacobiPrec<DirichletPoisson2D> J(A);

int it = pcg(B, A, x, b);
```

An example implementation is given in the FLENS tutorial [45]. Notable is the fact that the preconditioner directly benefits from any kind of optimization or parallelization of the underlying Jacobi method. Symmetric variants of stationary methods can be used to obtain further preconditioners analogously. For instance, the *symmetric Gauss-Seidel method* (cp. [52], [4])

$$u^{(n+1)} = -(U+D)^{-1}L(L+D)^{-1}Uu^{(n)} + (U+D)^{-1}D(L+D)^{-1}q,$$

leads to $B = (U+D)^{-1}D(L+D)^{-1}$.

Symmetric stationary iterative solvers can easily be integrated into FLENS to serve as smoothers for the multigrid method. This allows using the the multigrid method as a preconditioner.

6.8 Summary of the Chapter

The capabilities of FLENS to combine reusability, flexibility and efficiency were demonstrated by using the Dirichlet-Poisson problem as model problem. Combing these different aspects was possible due to the orthogonal design of the library. Reusability and flexibility was exploited on different levels. On the one hand, the same implementation of the multigrid or the cg-method could be reused to solve the problem in different space dimension and even parallelization did not require any modifications. On the other hand, it was shown for instance, how an optimized implementation of the Gauss-Seidel method could not only be used for the multigrid method as a smoother and direct solver but also as a preconditioner for the cg-method. Further, the total efficiency did benefit from the reusability. For example, performance tuning of the Gauss-Seidel method instantly tunes the performance of the multigrid and the cg-method.

7 Application of the FDM: Navier-Stokes Equations

Computational fluid dynamics (CFD) is one of the branches of fluid mechanics that uses numerical methods and algorithms to solve and analyze problems that involve fluid flows. Governing equations for the motion of fluid substances such as liquids and gases are the Navier-Stokes equations. This chapter exemplifies how components developed in the previous chapter can be applied to some simple problems in the field of CFD. The numerical treatment of the Navier-Stokes equations follows [47] very closely.

Section 7.1 briefly introduces the Navier-Stokes equations and Section 7.2 the so called *lid-driven cavity problem* which is a standard test problem in CFD. The time and space discretization of the Navier-Stokes equations gets covered in Sections 7.3 and 7.4 respectively. This results in a simple solver for the two-dimensional Navier-Stokes equations. As will be shown there, the computation of a numerical solution requires to solve a Neumann-Poisson problem. Section 7.5 illustrates how the implementation of the multigrid method from Chapter 6 can be utilized for this purpose. Section 7.6 outlines the parallelization of the Navier-Stokes solver. Finally, Section 7.7 illustrates how the components developed in this chapter can be applied to solve the so called *Pot and Ice* problem.

7.1 Navier-Stokes Equations

The Navier-Stokes equations establish that changes in momentum in infinitesimal volumes of fluid are the sum of forces acting inside the fluid. Therefore the equations basically constitute an application of Newton's second law. Relevant forces are dissipative viscous forces (similar to friction), changes in pressure and other body forces acting inside the fluid such as gravity. For an incompressible and Newtonian[1] fluid the flow inside a domain $\Omega \subset \mathbb{R}^n$ ($n \in \{2,3\}$) over time $t \in \mathbb{R}^+$ is governed by the equations

$$\begin{aligned} \frac{\partial}{\partial t}\vec{u} + (\vec{u}\cdot\nabla)\vec{u} + \nabla p &= \frac{1}{\mathrm{Re}}\Delta\vec{u} + \vec{F}, && (7.1)\\ \nabla\cdot\vec{u} &= 0 && (7.2)\end{aligned}$$

where

$\vec{u} : \Omega \times \mathbb{R}^+ \to \mathbb{R}^n$ denotes the velocity field of the fluid,

$p : \Omega \times \mathbb{R}^+ \to \mathbb{R}$ the pressure and

$\vec{F} : \Omega \times \mathbb{R}^+ \to \mathbb{R}^n$ the forces acting inside the fluid.

[1]A Newtonian fluid is characterized by a linear relation between sheer rate and sheer stress, e. g. $\tau_x = \eta\frac{\mathrm{d}\vec{u}}{\mathrm{d}y}$ where τ_x denotes the sheer stress in x direction and η the dynamic viscosity [39]. Examples for Newtonian fluids are water and air whereas blood is a typical non-Newtonian fluid.

Obviously equations (7.1) and (7.2) specify the pressure p only up to a constant value. Necessary for the existence and uniqueness of a solution are initial and boundary conditions for the velocity field $\vec{u}$. The initial condition

$$\vec{u}(\vec{x},0) = \vec{u}_0(\vec{x}), \quad \vec{x} \in \Omega \tag{7.3}$$

describing the flow for $t = 0$ has to satisfy (7.2). In the following boundary conditions are considered only for the special case where the fluid is surrounded by solid, impermeable walls. In this case the boundary condition specifies the velocity in normal and tangential direction on the boundary $\partial\Omega$ for $t \geq 0$. For $\vec{x} \in \partial\Omega$ let $\vec{u}_n(\vec{x},t)$ denote the velocity in normal direction and $\vec{u}_t(\vec{x},t) = \vec{u}(\vec{x},t) - \vec{u}_n(\vec{x},t)$ the velocity in tangential direction. The fact that no fluid can pass through the wall is specified by the boundary condition

$$\vec{u}_n(\vec{x},t) = 0, \quad \vec{x} \in \partial\Omega,\ t \geq 0. \tag{7.4}$$

The velocity tangential to the wall can be specified either by so called *no-slip* or *free-slip* conditions. The no-slip condition

$$\vec{u}_t(\vec{x},t) = 0, \quad \vec{x} \in \partial\Omega,\ t \geq 0 \tag{7.5}$$

states that the fluid 'sticks' on the wall such that the tangential velocity vanishes at the boundary. Conversely the free-slip condition[2]

$$\frac{\partial \vec{u}_t(\vec{x},t)}{\partial \vec{n}} = 0, \quad \vec{x} \in \partial\Omega,\ t \geq 0 \tag{7.6}$$

states that there is no friction loss at the wall.

Equations (7.1) to (7.6) specify the Navier-Stokes equations in non-dimensional form. This form is obtained by introducing a characteristic length L and a characteristic velocity u_∞ for the flow. Quantities in the Navier-Stokes equations are measured in units that are based on L and u_∞. For example, the velocity and the time is measured in units of u_∞ and L/u_∞ respectively. Denoting the constant density of the fluid by ρ_∞ the pressure and the force are measured in units of $1/\rho_\infty u_\infty^2$ and $L/\rho_\infty u_\infty^2$ respectively. The dimensionless Reynolds number

$$\mathrm{Re} = \frac{\rho_\infty\, u_\infty\, L}{\eta} \tag{7.7}$$

indicates the ratio between inertial forces ($u_\infty L$) and viscous forces (η/ρ_∞) where η denotes the dynamic viscosity.

7.2 Motivation: The Lid-Driven Cavity Problem

The lid-driven cavity problem has long been used as a test or validation case for new methods in computational fluid dynamics. The problem geometry is simple and two-dimensional. Figure 7.1 shows the standard case of the problem where fluid is contained in a square domain Ω with no-slip boundary conditions. The boundary has three non-moving sides while the 'lid' is moving tangentially.

Let $\vec{u} = (u,v)^T$ denote the velocity field in two space dimensions. The component form of equation (7.2) is given as

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0. \tag{7.8}$$

Applying (7.8) to the convective term $(\vec{u}\cdot\nabla)\vec{u}$ gives

$$(\vec{u}\cdot\nabla)\vec{u} = \begin{pmatrix} u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} \\ u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} \end{pmatrix} = \begin{pmatrix} \frac{\partial(u^2)}{\partial x} + \frac{\partial(uv)}{\partial y} \\ \frac{\partial(uv)}{\partial x} + \frac{\partial(v^2)}{\partial y} \end{pmatrix}. \tag{7.9}$$

[2]$\partial/\partial\vec{n}$ denotes the normal derivative.

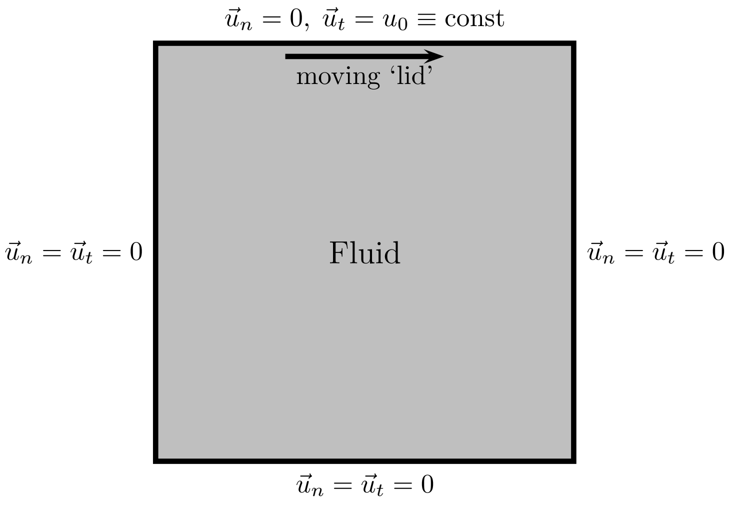

$\vec{u}_n = 0,\ \vec{u}_t = u_0 \equiv \text{const}$
moving 'lid'
$\vec{u}_n = \vec{u}_t = 0$
Fluid
$\vec{u}_n = \vec{u}_t = 0$
$\vec{u}_n = \vec{u}_t = 0$

Figure 7.1: The two-dimensional lid-driven cavity problem.

From this follows that the component form of equations (7.1) and (7.2) can be rewritten as

$$\frac{\partial u}{\partial t} + \frac{\partial (u^2)}{\partial x} + \frac{\partial (uv)}{\partial y} = f_x - \frac{\partial p}{\partial x} + \frac{1}{\text{Re}}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \tag{7.10}$$

$$\frac{\partial v}{\partial t} + \frac{\partial (uv)}{\partial x} + \frac{\partial (v^2)}{\partial y} = f_y - \frac{\partial p}{\partial y} + \frac{1}{\text{Re}}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) \tag{7.11}$$

where $\vec{F} = (f_x, f_y)^T$ denotes the gravity field.

Choosing the width of the square domain as characteristic length L does scale the domain such that $\Omega = (0,1)^2$. Then the boundary conditions for the lid-driven cavity problem can be stated as

$$u(x,1,t) = u_0, \quad u(x,0,t) = v(x,0,t) = v(x,1,t) = 0 \qquad x \in [0,1], \tag{7.12}$$

$$u(0,y,t) = u(1,y,t) = v(0,y,t) = v(1,y,t) = 0 \qquad y \in [0,1]. \tag{7.13}$$

To gain an impression on the meaning of the Reynolds number consider water as fluid. At 20^oC the dynamic viscosity of pure water is $\eta = 0.001\,\text{Ns/m}^2$ and its density $\rho_\infty = 999.2\,\text{Kg/m}^3$. Measuring the velocity u_0 in units of $u_\infty = 0.001\,\text{m/s}$ and lengths in units of $L = 1\,\text{m}$ leads to $\text{Re} \approx 1000$.

7.3 Time Discretization and the Projection Method

Time derivatives are approximated at discrete points of time $\{t_n\}_{n\geq 0}$ where $t_0 = 0$. Time discretizations of the velocity and pressure fields at time t_n are denoted by $\vec{u}^{(n)} = u(\,\cdot\,, t_n)$ and $p^{(n)} = p(\,\cdot\,, t_n)$ respectively. Using the explicit Euler method a semi-discrete version[3] of (7.1) can be written as

$$\frac{\vec{u}^{(n+1)} - \vec{u}^{(n)}}{\delta t_n} = \frac{1}{\text{Re}}\Delta \vec{u}^{(n)} + \vec{F}^{(n)} - \left(\vec{u}^{(n)} \cdot \nabla\right) \vec{u}^{(n)} - \nabla p^{(n+1)} \tag{7.14}$$

where $\vec{F}^{(n)} = \vec{F}(\,\cdot\,, t_n)$ and $\delta t_n = t_{n+1} - t_n$. Equation (7.14) is explicit for $\vec{u}$ and implicit for p. Rearranging terms gives the following semi-discrete version of equations (7.1) and (7.2):

$$\vec{u}^{(n+1)} = \vec{u}_*^{(n+1)} - \delta t_n \nabla p^{(n+1)}, \tag{7.15}$$

$$\nabla \cdot \vec{u}^{(n+1)} = 0 \tag{7.16}$$

[3]Discrete in time but continuous in space.

where

$$\vec{u}_*^{(n+1)} = \vec{u}^{(n)} + \delta t_n \left(\frac{1}{\mathrm{Re}} \Delta \vec{u}^{(n)} + \vec{F}^{(n)} - \left(\vec{u}^{(n)} \cdot \nabla \right) \vec{u}^{(n)} \right) \tag{7.17}$$

can be considered as an approximation of $\vec{u}^{(n+1)}$. As $\vec{u}_*^{(n+1)}$ only depends on values at time t_n it can be computed explicitly. Subsequently $\vec{u}^{(n+1)}$ gets computed by projecting $\vec{u}_*^{(n+1)}$ into the space of divergence-free vector fields. As can be seen from (7.15) this projection is achieved through a pressure correction. Applying (7.16) to equation (7.15) defines the correction term through

$$0 = \nabla \cdot \vec{u}^{(n+1)} = \nabla \cdot \left(\vec{u}_*^{(n+1)} - \delta t_n \nabla p^{(n+1)} \right). \tag{7.18}$$

Rearranging terms, this states the Poisson equation

$$-\Delta p^{(n+1)} = -\frac{1}{\delta t_n} \nabla \cdot \vec{u}_*^{(n+1)} \tag{7.19}$$

for the pressure gradient. Boundary conditions for (7.19) can be retrieved by imposing the boundary conditions

$$\vec{u}_*^{(n+1)} = \vec{u}^{(n+1)} = \vec{u}(\,\cdot\,, t_{n+1}) \quad \text{on } \partial\Omega \tag{7.20}$$

for the approximate velocity field. From (7.15) it follows that

$$\frac{\partial}{\partial \vec{n}} p^{(n+1)} = \nabla p^{(n+1)} \cdot \vec{n} = \frac{1}{\delta t_n} \left(\vec{u}_*^{(n+1)} - \vec{u}^{(n+1)} \right) \cdot \vec{n} = 0 \quad \text{on } \partial\Omega \tag{7.21}$$

where $\vec{n}$ denotes the outer normal vector on $\partial\Omega$. Because of these *homogenous Neumann boundary conditions* the pressure is only defined up to a constant. Imposing as a further condition

$$\int_\Omega p^{(n+1)}(\vec{x}) \, \mathrm{d}\vec{x} = 0 \tag{7.22}$$

the pressure filed $p^{(n+1)}$ is uniquely defined through (7.19), (7.21) and (7.22).

In the following the projection method is applied to equations (7.10) and (7.11) specifying the Navier-Stokes equations in two space dimension. The resulting semi-discrete equations are written in component form, which makes it easier to motivate the space discretization introduced in the next section. The computations required to advance the velocity field $\vec{u}^{(n)} = (u^{(n)}, v^{(n)})^T$ to time $t = t_{n+1}$ can be subdivided into the following three steps:

1. Compute the approximate velocities

$$u_*^{(n+1)} = u^{(n)} + \delta t_n \left(\frac{1}{\mathrm{Re}} \left(\frac{\partial^2 u^{(n)}}{\partial x^2} + \frac{\partial^2 u^{(n)}}{\partial y^2} \right) + f_x^{(n)} - \frac{\partial \left(u^{(n)} \right)^2}{\partial x} - \frac{\partial \left(u^{(n)} v^{(n)} \right)}{\partial y} \right) \tag{7.23}$$

$$v_*^{(n+1)} = v^{(n)} + \delta t_n \left(\frac{1}{\mathrm{Re}} \left(\frac{\partial^2 v^{(n)}}{\partial x^2} + \frac{\partial^2 v^{(n)}}{\partial y^2} \right) + f_y^{(n)} - \frac{\partial \left(u^{(n)} v^{(n)} \right)}{\partial x} - \frac{\partial \left(v^{(n)} \right)^2}{\partial y} \right) \tag{7.24}$$

2. Solve the Neumann-Poisson problem

$$-\frac{\partial^2 p}{\partial x^2} - \frac{\partial^2 p}{\partial y^2} = -\frac{1}{\delta t_n} \left(\frac{\partial u_*^{(n+1)}}{\partial x} + \frac{\partial v_*^{(n+1)}}{\partial y} \right) \quad \text{in } \Omega, \tag{7.25}$$

$$-\frac{\partial p}{\partial \vec{n}} = 0 \quad \text{on } \partial\Omega, \tag{7.26}$$

$$\int_\Omega p(\vec{x}) \, \mathrm{d}\vec{x} = 0 \tag{7.27}$$

for $p^{(n+1)}$.

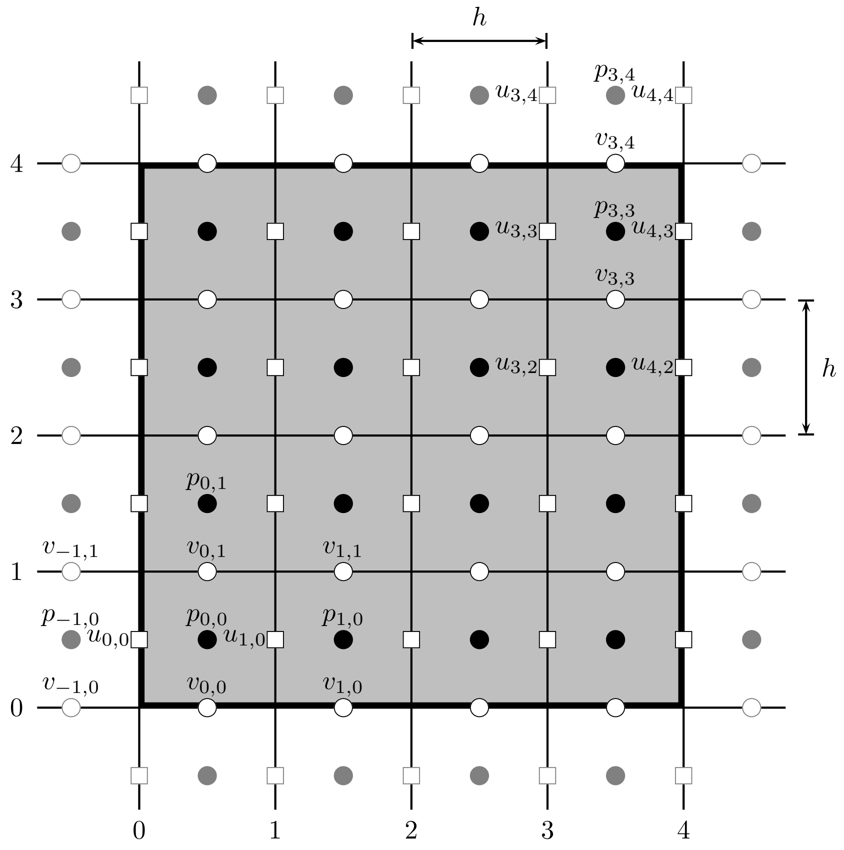


Figure 7.2: Staggered grids used for the discretization of the velocity and pressure fields.

3. Apply a pressure correction projecting the approximate velocities into the subspace of divergence-free functions:

$$u^{(n+1)} = u_*^{(n+1)} - \delta t_n \frac{\partial p^{(n+1)}}{\partial x} \tag{7.28}$$

$$v^{(n+1)} = v_*^{(n+1)} - \delta t_n \frac{\partial p^{(n+1)}}{\partial y} \tag{7.29}$$

7.4 Space Discretization and Implementation

This section illustrates how FLENS can be utilized to realize a simple implementation of a Navier-Stokes solver. At first, the semi-discrete equations that resulted from the projection method are discretized in space. Implementations of the particular discretization schemes will be introduced side by side. Finally, these implementations will be used as components for the Navier-Stokes solver.

7.4.1 Staggered Grids

In order to prevent pressure oscillations so called staggered grids are used for the space discretization. Staggered grids result from the grid Ω_h defined in Section 6.5 by shifting the grid points by half the grid spacing in x- or y-direction. For $s_x, s_y \in \left\{0, \frac{1}{2}\right\}$ staggered grids are formally defined through

$$\Omega_h^{(s_x,s_y)} := \left\{(x_i, y_i) = \big((i+s_x)\,h, (j+s_y)\,h\big) : i = \delta_{s_x,0}, \dots, N,\ j = \delta_{s_y,0}, \dots, N\right\}. \tag{7.30}$$

For the numerical treatment of the boundary conditions the grid $\Omega_h^{(s_x,s_y)}$ has to be extended by ghost nodes that are located outside the domain Ω. The formal definition of this extended grid is given through

$$\hat{\Omega}_h^{(s_x,s_y)} := \Big\{(x_i, y_i) = \big((i+s_x)\,h, (j+s_y)\,h\big) : i = i_0^{(s_x)}, \dots, i_1^{(s_x)},\, j = j_0^{(s_y)}, \dots, j_1^{(s_y)}\Big\} \tag{7.31}$$

where

$$i_0^{(s_x)} = \delta_{s_x,0} - 1, \quad i_1^{(s_x)} = N+1 \tag{7.32}$$

and

$$j_0^{(s_y)} = \delta_{s_y,0} - 1, \quad j_1^{(s_y)} = N+1. \tag{7.33}$$

Denoting corresponding spaces of grid functions by

$$V_h^{(s_x,s_y)} = \Big\{v \;\Big|\; v : \hat{\Omega}_h^{(s_x,s_y)} \to \mathbb{R}\Big\} \tag{7.34}$$

the semi-discrete functions in equations (7.23) to (7.28) are approximated by grid functions as follows:

1. The pressure filed $p^{(n)}$ gets approximated by $p_h^{(n)} \in V_h^{\left(\frac{1}{2},\frac{1}{2}\right)}$,
2. velocities $u^{(n)}$ and $u_*^{(n)}$ by $u_h^{(n)}, u_{*,h}^{(n)} \in V_h^{\left(0,\frac{1}{2}\right)}$ respectively and
3. velocities $v^{(n)}$ and $v_*^{(n)}$ by $v_h^{(n)}, v_{*,h}^{(n)} \in V_h^{\left(\frac{1}{2},0\right)}$.

The location of grid points that are relevant for the space discretization is illustrated in Figure 7.2.

Implementation

Listing 7.1 illustrates the implementation of grid functions defined on staggered grids. Template parameters (line 1) specify shifts of grid points in x- and y-direction. These shifts are defined through boolean values where `false` and `true` define a shift of 0 and $0.5h$ respectively. Analogously to the implementation of grid functions illustrated in the previous chapter in Listing 6.22 values on grid points are stored internally in a matrix (lines 13, 15) to provide fast element access. Dimensions of this internal matrix are defined in the constructor (lines 6-7) according to (7.32) and (7.33). As before, deriving the implementing class from the vector base class (line 3) realizes that grid functions are mathematically represented as vectors.

```
 1 template <bool DirectionX, bool DirectionY>
 2 class StaggeredGridVector2D
 3     : public Vector<StaggeredGridVector2D<DirectionX, DirectionY> >
 4 {
 5     public:
 6         StaggeredGridVector2D(int _rh)
 7             : rh(_rh), grid(_((DirectionX) ? -1: 0, rh),
 8                             _((DirectionY) ? -1: 0, rh))
 9         {}
10
11         // ...
12
13         typedef GeMatrix<FullStorage<double, RowMajor> >  Grid;
14         int     rh;
15         Grid    grid;
16 };
```

Listing 7.1: Gridfunctions on staggered grids.

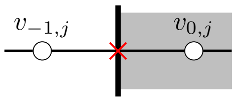


Figure 7.3: Treatment of boundary conditions: at the left boundary the velocity in y-direction is obtained at the red cross through linear interpolation.

7.4.2 Boundary Conditions for the Lid-Driven Cavity Problem

Values of $u_h^{(n)}$ and $v_h^{(n)}$ at grid points located on the boundary are specified through (7.12) and (7.13) to be

$$u_{0,j}^{(n)} = 0, \qquad u_{N+1,j}^{(n)} = 0, \quad j = 0, \dots, N, \tag{7.35}$$
$$v_{i,N+1}^{(n)} = 0, \qquad v_{i,0}^{(n)} = 0, \quad i = 0, \dots, N. \tag{7.36}$$

Where grid points are not directly located on the boundary the boundary conditions are enforced by setting the neighboring ghost nodes. As illustrated in Figure 7.3 values on the boundary are retrieved through linear interpolation. Applying this to enforce the boundary conditions for $u_h^{(n)}$ at the top and bottom side and for $v_h^{(n)}$ at the left and right side leads to

$$u_{i,N+1}^{(n)} = 2u_0 - u_{i,N}^{(n)}, \qquad u_{i,-1}^{(n)} = -u_{i,0}^{(n)}, \quad i = 0, \dots, N, \tag{7.37}$$
$$v_{-1,j}^{(n)} = -v_{0,j}^{(n)}, \qquad v_{N+1,j}^{(n)} = -v_{N,j}^{(n)}, \quad j = 0, \dots, N. \tag{7.38}$$

Implementation

Listing 7.2 illustrates an implementation to enforce the boundary conditions as specified by equations (7.35) to (7.38).

```
template <typename U, typename V>
void
setBoundaryCondition(int rh, GeMatrix<U> &u, GeMatrix<V> &v, double u0)
{
    int N = rh - 1;

    u(0,_) = 0; u(N+1,_) = 0;
    v(_,0) = 0; v(_,N+1) = 0;

    u(_,-1)  = -u(_,0);
    u(_,N+1) = 2*u0;
    u(_,N+1) -= u(_,N);

    v( -1,_) = -v(0,_);
    v(N+1,_) = -v(N,_);
}
```

Listing 7.2: Enforcing boundary conditions for the lid-driven cavity problem.

7.4.3 Numerical Approximation of $u_*^{(n+1)}$ and $v_*^{(n+1)}$

In the following only the computation of $u_{*,h}^{(n+1)}$ approximating $u_*^{(n+1)}$ is illustrated in more detail. A corresponding approximation of $v_*^{(n+1)}$ can be obtained analogously. The approximation of $u_*^{(n+1)}$ results from (7.28) by discretizing the spacial derivatives. Let $[\,\cdot\,]_{x,y}$ indicate the approximation of derivatives at (x, y) then for $i = 1, \dots, N$ and $j = 0, \dots, N$ an approximation of

$u_*^{(n+1)}$ at the grid point $(ih,(j+0.5)h)$ is specified through

$$\begin{aligned} u_{*,i,j}^{(n+1)} &= u_{i,j}^{(n)} + \delta t_n \left(\frac{1}{\mathrm{Re}} \left[\frac{\partial^2 u^{(n)}}{\partial x^2} + \frac{\partial^2 u^{(n)}}{\partial y^2} \right]_{ih,(j+0.5)h} + f_x^{(n)}(ih,(j+0.5)h) \right. \\ & \left. - \left[\frac{\partial \left(u^{(n)}\right)^2}{\partial x} \right]_{ih,(j+0.5)h} - \left[\frac{\partial \left(u^{(n)} v^{(n)}\right)}{\partial y} \right]_{ih,(j+0.5)h} \right) \end{aligned} \tag{7.39}$$

Approximations of the different terms can be obtained as follows:

1. Applying second central differences to the first term leads to

$$\left[\frac{\partial^2 u^{(n)}}{\partial x^2} + \frac{\partial^2 u^{(n)}}{\partial y^2} \right]_{ih,(j+0.5)h} = \frac{u_{i-1,j}^{(n)} - 2u_{i,j}^{(n)} + u_{i+1,j}^{(n)}}{h^2} + \frac{u_{i,j-1}^{(n)} - 2u_{i,j}^{(n)} + u_{i,j+1}^{(n)}}{h^2}. \tag{7.40}$$

2. Applying second central differences to the interpolation of $u_h^{(n)}$ at the cell centers (indicated in Figure 7.4 by x) leads to

$$\left[\frac{\partial \left(u^{(n)}\right)^2}{\partial x} \right]_{ih,(j+0.5)h} = \frac{1}{4h} \left(\left(u_{i+1,j}^{(n)} + u_{i,j}^{(n)} \right)^2 - \left(u_{i,j}^{(n)} + u_{i-1,j}^{(n)} \right)^2 \right). \tag{7.41}$$

For high Reynolds numbers or high velocities using second central differences leads to numerical instabilities. A thorough discussion about the numerical discretization of convective terms can be found for instance in [47], [28] or [29]. Following closely the methods introduced in [47] the convective terms are discretized using the weighted donor-cell scheme

$$\begin{aligned} \left[\frac{\partial \left(u^{(n)}\right)^2}{\partial x} \right]_{ih,(j+0.5)h} &= \frac{1}{4h} \left(\left(u_{i,j}^{(n)} + u_{i+1,j}^{(n)} \right)^2 - \left(u_{i-1,j}^{(n)} + u_{i,j}^{(n)} \right)^2 \right) \\ &+ \frac{\gamma}{4h} \left(\left| u_{i,j}^{(n)} + u_{i+1,j}^{(n)} \right| \left(u_{i,j}^{(n)} + u_{i+1,j}^{(n)} \right) \right) \\ &- \frac{\gamma}{4h} \left(\left| u_{i-1,j}^{(n)} + u_{i,j}^{(n)} \right| \left(u_{i-1,j}^{(n)} + u_{i,j}^{(n)} \right) \right). \end{aligned} \tag{7.42}$$

where $\gamma \in [0,1]$. Obviously the second central differences are obtained for $\gamma = 0$. A criterion for the choice of γ is given below in (7.44).

3. Applying the weighted donor-cell scheme analogously to the third term leads to

$$\begin{aligned} \left[\frac{\partial \left(u^{(n)} v^{(n)}\right)}{\partial y} \right]_{ih,(j+0.5)h} &= \frac{1}{4h} \left(\left(u_{i,j}^{(n)} + u_{i,j+1}^{(n)} \right) \left(v_{i-1,j+1}^{(n)} + v_{i,j+1}^{(n)} \right) \right. \\ & \left. - \left(u_{i,j-1}^{(n)} + u_{i,j}^{(n)} \right) \left(v_{i,j}^{(n)} + v_{i-1,j}^{(n)} \right) \right) \\ &+ \frac{\gamma}{4h} \left(\left(u_{i,j}^{(n)} + u_{i,j+1}^{(n)} \right) \left| v_{i-1,j+1}^{(n)} + v_{i,j+1}^{(n)} \right| \right) \\ &- \frac{\gamma}{4h} \left(\left(u_{i,j-1}^{(n)} + u_{i,j}^{(n)} \right) \left| v_{i,j}^{(n)} + v_{i-1,j}^{(n)} \right| \right) \end{aligned} \tag{7.43}$$

According to [47] and [40] the weighting parameter has to be chosen such that

$$\gamma \geq \max_{i,j} \left(\left| \frac{u_{i,j}^{(n)} \delta t_n}{h} \right|, \left| \frac{v_{i,j}^{(n)} \delta t_n}{h} \right| \right) \tag{7.44}$$

is satisfied.

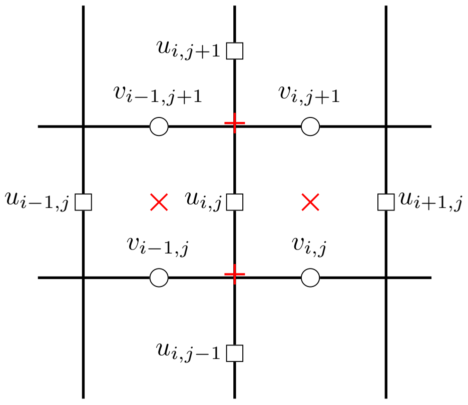


Figure 7.4: Interpolation of $u_h^{(n)}$ and $v_h^{(n)}$.

Implementation

The computations of $u_{*,h}^{(n+1)}$ and $v_{*,h}^{(n+1)}$ are implemented in function `computeImpulse`[4]. The function receives the velocities $u_h^{(n)}$ and $v_h^{(n)}$ through matrices `u` and `v` (line 3) where the matrix index ranges are assumed to conform (7.32) and (7.33). Force fields f_x and f_y are assumed to be constant. The approximate velocities $u_{*,h}^{(n+1)}$ and $v_{*,h}^{(n+1)}$ are stored in matrices `us` and `vs` respectively which get passed to the functions as a non-const reference. As can be seen from Listing 7.3 the implementation follows closely the notation used for the discretization: (7.40) is realized in lines 11-12, (7.42) in lines 14-18 and (7.43) in lines 20-24.

```
 1 template <typename U, typename V, typename US, typename VS>
 2 void
 3 computeImpulse(int rh, const GeMatrix<U> &u, const GeMatrix<V> &v,
 4                double dt, double gamma, double re, double fx, double fy,
 5                GeMatrix<US> &us, GeMatrix<VS> &vs)
 6 {
 7     double rhh = rh*rh;
 8 
 9     for (int i=u.firstRow()+1; i<=u.lastRow()-1; ++i) {
10         for (int j=u.firstCol()+1; j<=u.lastCol()-1; ++j) {
11             double ddux = (u(i+1,  j) - 2*u(i,j) + u(i-1,j))*rhh;
12             double dduy = (u(  i,j+1) - 2*u(i,j) + u(  i,j-1))*rhh;
13 
14             double duu = sqr(u(  i,j) + u(i+1,j))
15                        - sqr(u(i-1,j) + u(  i,j))
16                 +gamma*( std::abs(u(  i,j)+u(i+1,j))*(u(  i,j)-u(i+1,j))
17                         -std::abs(u(i-1,j)+u(  i,j))*(u(i-1,j)-u(  i,j)));
18             duu *= rh/4;
19 
20             double duv = (v(i,j+1)+v(i-1,j+1))*(u(i,  j)+u(i,j+1))
21                        - (v(i,  j)+v(i-1,  j))*(u(i,j-1)+u(i,  j))
22                 +gamma*( std::abs(v(i,j+1)+v(i-1,j+1))*(u(i,  j)-u(i,j+1))
23                         -std::abs(v(i,  j)+v(i-1,  j))*(u(i,j-1)-u(i,  j)));
24             duv *= rh/4;
25 
26             us(i,j) = u(i,j) + dt*((ddux+dduy)/re - duu -duv +fx);
27         }
28     }
29 
30     // ...
31 }
```

Listing 7.3: Computation of the approximate velocities in x and y direction.

[4]The function name reflects that $u_{*,h}^{(n+1)}$ results from integrating volume forces over the time-interval δt_n.

7.4.4 Computation of $p^{(n+1)}$

Discretizing the right-hand side of (7.25) using first central differences, i. e.

$$\begin{aligned} q_{i,j}^{(n+1)} &= -\frac{1}{\delta t_n}\left[\frac{\partial u_*^{(n+1)}}{\partial x} + \frac{\partial v_*^{(n+1)}}{\partial y}\right]_{((i+0.5)h,(j+0.5)h)} \\ &= -\frac{1}{\delta t_n}\left(\frac{u_{*,i+1,j}^{(n+1)} - u_{*,i,j}^{(n+1)}}{h} + \frac{v_{*,i,j+1}^{(n+1)} - v_{*,i,j}^{(n+1)}}{h}\right) \end{aligned} \tag{7.45}$$

and second central differences for the left-hand side leads to the system of equations

$$\frac{1}{h^2}\left(4p_{i,j}^{(n+1)} - p_{i-1,j}^{(n+1)} - p_{i+1,j}^{(n+1)} - p_{i,j-1}^{(n+1)} - p_{i,j+1}^{(n+1)}\right) = q_{i,j}^{(n+1)} \quad i,j \in \{0,\dots,N\}. \tag{7.46}$$

From the discretization of the homogeneous Neumann boundary conditions (7.26) it follows that

$$p_{i,-1}^{(n+1)} = p_{i,0}^{(n+1)}, \quad p_{i,N+1}^{(n+1)} = p_{i,N}^{(n+1)} \qquad i \in \{0,\dots,N\}, \tag{7.47}$$

$$p_{-1,j}^{(n+1)} = p_{0,j}^{(n+1)}, \quad p_{N+1,j}^{(n+1)} = p_{N,j}^{(n+1)} \qquad j \in \{0,\dots,N\}. \tag{7.48}$$

Hence, a grid function p_h that satisfies the Neumann boundary conditions gets uniquely described by its values on interior grid points and can be represented as

$$p_h = (p_{0,0},\dots,p_{N,0},p_{0,1},\dots)^T \in \mathbb{R}^{(N+1)^2}.$$

From (7.47) and (7.48) it follows that for the system of equations in (7.46) the coefficient matrix is given by

$$\tilde{A}_h := \frac{1}{h^2}\begin{pmatrix} \tilde{T}_N + 2I & -I & & & \\ -I & \tilde{T}_N + 2I & -I & & \\ & \ddots & \ddots & \ddots & \\ & & -I & \tilde{T}_N + 2I & -I \\ & & & -I & \tilde{T}_N + 2I \end{pmatrix} \in \mathbb{R}^{(N+1)^2 \times (N+1)^2} \tag{7.49}$$

where

$$\tilde{T}_N = \begin{pmatrix} 1 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 1 \end{pmatrix} \in \mathbb{R}^{(N+1)\times(N+1)}. \tag{7.50}$$

Using numerical quadrature for the discretizing condition of (7.27) leads to

$$0 = \sum_{i=0}^{N}\sum_{j=0}^{N} p_{i,j}^{(n+1)} = e^T p_h^{(n+1)} \tag{7.51}$$

where $e = (1,\dots,1)^T \in \mathbb{R}^{(N+1)^2}$. In order to allow a more compact notation let

$$P_h := \left\{ p_h \in \mathbb{R}^{(N+1)^2} \ : \ e^T p_h \right\} \tag{7.52}$$

denote the subspace satisfying the orthogonality condition (7.51). Then a discrete version of the Neumann-Poisson problem stated in equations (7.25) to (7.27) can be written as:

$$\text{Find } p_h^{(n+1)} \in P_h \text{ such that } \tilde{A}_h\, p_h^{(n+1)} = q_h^{(n+1)}. \tag{7.53}$$

By examining the eigenvalues and eigenvectors of $\tilde{A}_h$ criterions for the existence and uniqueness of a solution can be given. Eigenvalues $\tilde{\lambda}_0, \dots, \tilde{\lambda}_N$ and eigenvectors $\tilde{v}_0, \dots, \tilde{v}_N$ of $\tilde{T}_N$ are explicitly known to be

$$\tilde{\lambda}_k = 2\left(1 - \cos\left(\frac{\pi k}{N+1}\right)\right), \quad \tilde{v}_k = \left(\cos\left(\frac{\pi k\left(l+\frac{1}{2}\right)}{N+1}\right)\right)_{l=0}^{N}. \tag{7.54}$$

It can be shown that the set of eigenvectors form an orthogonal basis of $\mathbb{R}^{N+1}$. As was shown in Section 6.5.3 it follows that eigenvalues and eigenvectors of $\tilde{A}_h$ are

$$\tilde{\lambda}_{k,l} = \frac{\tilde{\lambda}_k + \tilde{\lambda}_k}{h^2} \quad \text{and} \quad \tilde{v}_{k,l} = \mathbf{Vec}\left(\tilde{v}_k \tilde{v}_l^T\right) \quad k, l = 0, \dots, N. \tag{7.55}$$

Obviously $\tilde{\lambda}_{0,0}$ is the only zero-valued eigenvalue of $\tilde{A}_h$. Therefore a unique solution of (7.53) exists if and only if the right-hand side q_h is orthogonal to eigenvector $\tilde{v}_{0,0} = e$ spanning the kernel of $\tilde{A}_h$. From (7.45) it follows that this condition for solvability, i. e.

$$0 = \mathbf{Vec}\left(\tilde{v}_{0,0}\right)^T q_h^{(n+1)} = e^T q_h^{(n+1)} \quad \Leftrightarrow \quad \sum_{i=0}^{N}\sum_{j=0}^{N} q_{i,j}^{(n+1)} = 0, \tag{7.56}$$

is satisfied if $u_{*,h}^{(n+1)}$ satisfies the type of boundary conditions stated in (7.35) and (7.36).

The system of linear equations stated in (7.53) can be solved using the multigrid method. Definition of the appropriate smoothing, restriction and prolongation operators and an outline for an implementation is illustrated in more detail in Section 7.5.

Implementation

As illustrated in Listing 7.4 the right-hand side q_h for the Neumann-Poisson problem gets assembled according to (7.45).

```
template <typename US, typename VS, typename RHS>
void
computePoissonRhs(int rh, const GeMatrix<US> &us, const GeMatrix<VS> &vs,
                  double dt, GeMatrix<RHS> &rhs)
{
    for (int i=rhs.firstRow()+1; i<=rhs.lastRow()-1; ++i) {
        for (int j=rhs.firstCol()+1; j<=rhs.lastCol()-1; ++j) {
            rhs(i,j) =-rh*((us(i+1,j)-us(i,j))+(vs(i,j+1)-vs(i,j)))/dt;
        }
    }
}
```

Listing 7.4: Computation of the right-hand side for the Neumann-Poisson problem.

7.4.5 Computation of $u^{(n+1)}$

Using first central differences for the discretization of the pressure correction defined in (7.28) and (7.29) leads to

$$u_{i,j}^{(n+1)} = u_{*,i,j}^{(n+1)} - \delta t_n \left(p_{i,j}^{(n+1)} - p_{i-1,j}^{(n+1)}\right), \tag{7.57}$$

$$v_{i,j}^{(n+1)} = v_{*,i,j}^{(n+1)} - \delta t_n \left(p_{i,j}^{(n+1)} - p_{i,j-1}^{(n+1)}\right). \tag{7.58}$$

Implementation

As indicated in Listing 7.7 the pressure correction can be implemented completely analogously to the discrete formulation given in (7.57) and (7.58).

```
template <typename US, typename VS, typename P, typename U, typename V>
void
computeVelocity(int rh, const GeMatrix<US> &us, const GeMatrix<VS> &vs,
                const GeMatrix<P> &p, double dt,
                GeMatrix<U> &u, GeMatrix<V> &v)
{
    for (int i=u.firstRow()+1; i<=u.lastRow()-1; ++i) {
        for (int j=u.firstCol()+1; j<=u.lastCol()-1; ++j) {
            u(i,j) = us(i,j) - rh*dt*(p(i,j)-p(i-1,j));
        }
    }

    // ...
}
```

Listing 7.5: Obtaining $u^{(n+1)}$ by applying the pressure correction.

7.4.6 Stability Conditions and Time Step Control

Depending on the spacial discretization time steps δt_n have to satisfy certain stability conditions. Necessary for the condition is the so called *Courant-Friedrichs-Levy* (CFL) condition

$$\max|u_h^{(n)}|\,\delta t_n < h, \quad \max|v_h^{(n)}|\,\delta t_n < h. \tag{7.59}$$

This condition applies, generally, to explicit schemes for hyperbolic partial differential equations. Physically, this condition indicates that fluid particles should not travel more than one spacial step size h in one time step δt_n. The stability condition

$$\delta t_n < \mathrm{Re}\,\frac{h^2}{4} \tag{7.60}$$

applies to explicit schemes for hyperbolic partial differential equations [29]. For the Navier-Stokes equation all these conditions are necessary. Defining time steps by

$$\delta t_n := \tau \min\left\{\mathrm{Re}\,\frac{h^2}{4},\ \frac{h}{\max|u_h^{(n)}|},\ \frac{h}{\max|v_h^{(n)}|}\right\} \tag{7.61}$$

where $\tau \in (0,1]$ is a safety factor ensures that the above conditions are met.

Implementation

Function `computeTimeStep` computes δt_n according to (7.61). As indicated in Listing 7.6 the function receives the discrete velocity field, the Reynolds number, the grid h and returns the new time step.

```
template <typename U, typename V>
double
computeTimeStep(int rh, const GeMatrix<U> &u, const GeMatrix<V> &v,
                double re, double delta)
{
    // ...
}
```

Listing 7.6: Computation of time steps δt_n.

7.4.7 FLENS Implementation of the Lid-Driven Cavity Problem

So far functions were implemented in a 'FLENS independent style'. This means, functions like

```
template <typename U, typename V, typename US, typename VS>
void
computeImpulse(int rh, const GeMatrix<U> &u, const GeMatrix<V> &v,
               double dt, double gamma, double re, double fx, double fy,
               GeMatrix<US> &us, GeMatrix<VS> &vs)
{
    // ...
}
```

could have been implemented analogously in programming languages like C or Fortran. Therefore, these function can be considered as low-level implementations.

Vector type `StaggeredGridVector2D` was defined in Listing 7.1 to represent grid functions on staggered grids. In the following, typedefs

```
typedef StaggeredGridVector2D<false, true>  GridVectorU;
typedef StaggeredGridVector2D<true, false>  GridVectorV;
typedef StaggeredGridVector2D<true, true>   GridVectorP;
```

will be used for convenience. As was thoroughly illustrated in Chapter 6, the low-level implementations can be integrated into FLENS in the usual manner:

```
void
computeImpulse(const GridVectorU &u, const GridVectorV &v,
               double dt, double gamma, double re, double fx, double fy,
               GridVectorU &us, GridVectorV &vs)
{
    // ... call low-level component ...
}
```

For the sake of simplicity physical parameters are assumed to be defined as global variables:

```
double re = 1000;
double pr = 7;
double delta = 0.15;
double gamma = 0.9;
double gx = 0;
double gy = -9.81;
```

Listing 7.7: Obtaining $u^{(n+1)}$ by applying the pressure correction.

Combing the single components elaborated in this section, Algorithm 7.1 outlines an explicit finite difference scheme in order to obtain a numerical solution of the lid-driven cavity problem.

Algorithm 7.1 (Explicit Finite Difference Scheme).

```
Initialize: $t \leftarrow 0$ and $n \leftarrow 0$.
While $t < t_{max}$
    For $u^{(n)}$ and $v^{(n)}$ set boundary condition according to (7.35), (7.36) and (7.38).
    Compute $\delta t_n$ according to (7.61).
    Compute $u_{*,h}^{(n+1)}$ and $v_{*,h}^{(n+1)}$ according to (7.39).
    Compute $q^{(n+1)}$ according to (7.45).
    Compute $p^{(n+1)}$ by solving (7.53).
    Compute $u^{(n+1)}$ and $v^{(n+1)}$ according to (7.57) and (7.58).
    $t \leftarrow t + \delta t_n$, $n \leftarrow 0$.
```

The implementation outlined in Listing 7.8 closely follows Algorithm 7.1. However, the computation of $p^{(n+1)}$ is only indicated in line 13. This requires solving the system of linear equations in problem (7.53) and actually constitutes the most time consuming operation. The following section covers the computation of $p^{(n+1)}$ in more detail.

```
 1 GridVectorU  u(rh), us(rh);
 2 GridVectorV  v(rh), vs(rh);
 3 GridVectorP  &p = u_mg[p_mg], &rhs = f_mg[p_mg];
 4
 5 // ...
 6
 7 while (t<tMax) {
 8     setBoundaryCondition(u, v, 1);
 9     dt = computeTimeStep(u, v, re, pr, delta);
10     computeForce(u, v, dt, gamma, re, gx, gy, us, vs);
11     computePoissonRhs(us, vs, dt, rhs);
12
13     // compute p (using the multigrid method)
14
15     computeVelocity(us, vs, p, dt, u, v);
16     t += dt;
17 }
```

Listing 7.8: Implementation of Algorithm 7.1.

7.5 Multigrid Method for the Neumann-Poisson Problem

This section illustrates the application of the multigrid method for solving (7.53). For convenience the notation used in Section 7.4.4 gets adapted as follows:

1. $p_h^* \in P_h$ denotes the exact solution, i. e. $\tilde{A}_h p_h^* = q_h$.
2. $p_h^{(0)}$ denotes the initial guess and $p_h^{(1)}, p_h^{(2)}, \dots$ a sequence of vectors eventually converging to the exact solution.

7.5.1 Smoothing Operators: Stationary Iterative Methods

As defined in Section 6.3, let $\tilde{A}_h = L + D + U$ denote an additive decomposition. Eigenvalues of the Jacobi iteration matrix

$$J_{\tilde{A}_h} = -D^{-1}(L+U) \tag{7.62}$$

are related to $\tilde{A}_h$ by

$$\lambda_{k,l}(J_{\tilde{A}_h}) = 1 - \frac{\lambda_{k,l}(h^2\tilde{A}_h)}{4} = \frac{1}{2}\left(\cos\left(\frac{\pi l}{N+1}\right) + \cos\left(\frac{\pi k}{N+1}\right)\right), \quad k,l = 0,\dots,N \tag{7.63}$$

and eigenvectors of $J_{\tilde{A}_h}$ and $\tilde{A}_h$ are the same. From (7.63) it immediately follows that $\rho(\tilde{A}_h) = 1$ such that convergence of the method is not obviously guaranteed. The error after ν iterations is

$$e^{(\nu)} = p_h^* - p_h^{(\nu)} = \left(J_{\tilde{A}_h}\right)^{\nu}\left(p_h^* - p_h^{(0)}\right) = \left(J_{\tilde{A}_h}\right)^{\nu} e^{(0)}. \tag{7.64}$$

Representing the initial error with respect to the eigenvectors, i. e.

$$e^{(0)} = \sum_{k=0}^{N}\sum_{l=0}^{N} c_{k,l}\tilde{v}_{k,l} \tag{7.65}$$

leads to

$$e^{(\nu)} = \sum_{k=0}^{N}\sum_{l=0}^{N} c_{k,l}\lambda_{k,l}(J_{\tilde{A}_h})^{(\nu)}\tilde{v}_{k,l}. \tag{7.66}$$

For an initial guess $p_h^{(0)} \in P_h$ it follows that $c_{0,0} = 0$ such that

$$\lim_{\nu\to\infty} \left| c_{k,l} \lambda_{k,l}(J_{\tilde{A}_h})^{(\nu)} \right| = 0 \quad k,l = 0,\dots,N. \tag{7.67}$$

Hence the Jacobi method does converge if all computation can be carried out exactly.

For numerical implementations involving roundoff errors convergence can be guaranteed by projecting each iterate of the Jacobi method into P_h. Using the Gram-Schmidt method to enforce the orthogonality condition (7.51) leads to the projection

$$P_e : \mathbb{R}^{(n+1)^2} \to P_h, \quad x \mapsto P_e x = \left(I - \frac{ee^T}{e^T e} \right) x = x - \frac{e^T x}{e^T e}\, e. \tag{7.68}$$

Applying the projection after each iteration the thus modified Jacobi method can be formally stated as

$$p_h^{(\nu+1)} = P_e \left[J_{\tilde{A}_h} p_h^{(\nu)} + D^{-1} q_h \right] = P_e J_{\tilde{A}_h} p_h^{(\nu)} + P_e D^{-1} q_h. \tag{7.69}$$

Convergence of (7.69) can now be shown directly: Eigenvalues of P_e are

$$\lambda_{k,l}(P_e) = \begin{cases} 0, & k = l = 0, \\ 1, & \text{else} \end{cases} \tag{7.70}$$

and eigenvectors of P_e coincide with those of $\tilde{A}_h$. From this follows that

$$\lambda_{k,l}(P_e J_{\tilde{A}_h}) = \begin{cases} 0, & k = l = 0, \\ \lambda_{k,l}(J_{\tilde{A}_h}), & \text{else} \end{cases} \tag{7.71}$$

such that for the iteration matrix of (7.69) follows that $\rho(P_e J_{\tilde{A}_h}) < 1$.

Applying the projection (7.68) analogously to the Gauss-Seidel method leads to

$$p_h^{(\nu+1)} = P_e \left[G_{\tilde{A}_h} p_h^{(\nu)} + (L+D)^{-1} q_h \right] = P_e G_{\tilde{A}_h} p_h^{(\nu)} + P_e (L+D)^{-1} q_h \tag{7.72}$$

where

$$G_{\tilde{A}_h} = -(L+D)^{-1} U. \tag{7.73}$$

As shown in [52] eigenvalues of $G_{\tilde{A}_h}$ and $J_{\tilde{A}_h}$ are related by $\lambda_{k,l}(G_{\tilde{A}_h}) = \lambda_{k,l}(J_{\tilde{A}_h})^2$ such that $\rho(P_e G_{\tilde{A}_h}) = \rho(P_e J_{\tilde{A}_h})^2 < 1$ and convergence of (7.72) is guaranteed.

Implementation

```
template <typename Q, typename P>
void
dp2d_gauss_seidel(int rh, const GeMatrix<Q> &q, GeMatrix<U> &p)
{
    np2d_setBoundaryConditions(p);
    dp2d_gauss_seidel(rh, q, p);
    np2d_normalize(p);
}
```

Listing 7.9: Red-Black Gauss-Seidel-iteration for the two-dimensional Neumann-Poisson problem.

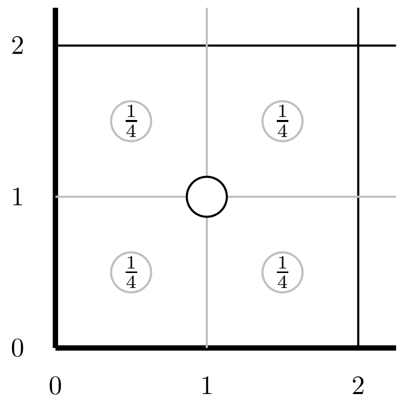


Figure 7.5: Stencil for the restriction. The value of the node on the coarse grid (black circle) is the average of nodes on the fine grid (gray circles).

7.5.2 Restriction Operator

In order to motivate a definition of a restriction operator

$$R_h^{2h} : V_h^{\left(\frac{1}{2},\frac{1}{2}\right)} \to V_{2h}^{\left(\frac{1}{2},\frac{1}{2}\right)} \tag{7.74}$$

consider a grid point $(\tilde{x}_i, \tilde{y}_i) = \big((2i+1)h, (2j+1)h\big) \in \Omega_{2h}^{\left(\frac{1}{2},\frac{1}{2}\right)}$. As can be seen from Figure 7.5 the nearest grid points on the finer grid $\Omega_h^{\left(\frac{1}{2},\frac{1}{2}\right)}$ are

$$\begin{aligned} (x_{2i}, y_{2i}) &= ((2i+0.5)\,h, (2i+0.5)\,h)\,, & (x_{2i+1}, y_{2i}) &= ((2i+1.5)\,h, (2i+0.5)\,h)\,, \\ (x_{2i}, y_{2i+1}) &= ((2i+0.5)\,h, (2i+1.5)\,h)\,, & (x_{2i+1}, y_{2i+1}) &= ((2i+1.5)\,h, (2i+1.5)\,h)\,. \end{aligned}$$

This suggests to define the restriction by averaging values on the finer grid accordingly. For $v = (v_{i,j})_{i,j=0}^N \in V_h^{\left(\frac{1}{2},\frac{1}{2}\right)}$ an approximation $\tilde{v} = (\tilde{v}_{i,j})_{i,j=0}^n \in V_{2h}^{\left(\frac{1}{2},\frac{1}{2}\right)}$ is obtained by

$$\tilde{v}_{i,j} = \frac{1}{4}\left(v_{2i,2j} + v_{2i+1,2j} + v_{2i,2j+1} + v_{2i+1,2j+1}\right), \quad i,j = 0,\ldots,n. \tag{7.75}$$

If computations are carried out exactly it follows that

$$v \in P_h \quad \Rightarrow \quad \tilde{v} = R_h^{2h} v \in P_{2h}. \tag{7.76}$$

For the multigrid method this guarantees the solvability of the restricted problem. The right-hand side of $\tilde{A}_{2h} p_{2h} = q_{2h}$ is defined as the restriction $q_{2h} := R_h^{2h} r_h$ for a residual $r_h \in P_h$. Combining the restriction with a Gram-Schmidt step, i. e. defining

$$\tilde{R}_h^{2h} : V_h^{\left(\frac{1}{2},\frac{1}{2}\right)} \to V_{2h}^{\left(\frac{1}{2},\frac{1}{2}\right)}, \quad v \mapsto P_e R_h^{2h} v, \tag{7.77}$$

ensures that solvability is not compromised through roundoff errors.

Implementation

Listing 7.10 illustrates a simple implementation of the restriction stencil defined in (7.75).

```
template <typename V, typename VC>
void
np2d_restriction(const GeMatrix<V> &v, GeMatrix<VC> &vc)
{
    int i0 = vc.firstRow()+1;
    int i1 = vc.lastRow()-1;
    int j0 = vc.firstCol()+1;
    int j1 = vc.lastCol()-1;

    for (int i=i0, I=2*i0; i<=i1; ++i, I+=2) {
        for (int j=j0, J=2*j0; j<=j1; ++j, J+=2) {
            vc(i,j) = 0.25*(v(I,J) + v(I+1,J) + v(I,J+1) + v(I+1,J+1));
        }
    }
}
```

Listing 7.10: Computation of the restriction.

7.5.3 Prolongation Operator

The prolongation operator

$$P_{2h}^{h} : V_{2h}^{\left(\frac{1}{2},\frac{1}{2}\right)} \to V_{h}^{\left(\frac{1}{2},\frac{1}{2}\right)} \tag{7.78}$$

can be defined through bilinear interpolation as illustrated in Figure 7.6. In the prolongation each point on the coarse grid contributes to its four neighboring grid points. Denoting as before the grid function on the coarse and fine grids by $\tilde{v}$ and v the corresponding stencil of prolongation operator is given by

$$\begin{aligned}
v_{2i,2j} &= \frac{3}{4}\left(\frac{1}{4}\tilde{v}_{i-1,j} + \frac{3}{4}\tilde{v}_{i,j}\right) + \frac{1}{4}\left(\frac{1}{4}\tilde{v}_{i-1,j-1} + \frac{3}{4}\tilde{v}_{i,j-1}\right), && (7.79)\\
v_{2i,2j+1} &= \frac{1}{4}\left(\frac{1}{4}\tilde{v}_{i-1,j+1} + \frac{3}{4}\tilde{v}_{i,j+1}\right) + \frac{3}{4}\left(\frac{1}{4}\tilde{v}_{i-1,j} + \frac{3}{4}\tilde{v}_{i,j}\right), && (7.80)\\
v_{2i+1,2j} &= \frac{3}{4}\left(\frac{3}{4}\tilde{v}_{i,j} + \frac{1}{4}\tilde{v}_{i+1,j}\right) + \frac{1}{4}\left(\frac{3}{4}\tilde{v}_{i,j-1} + \frac{1}{4}\tilde{v}_{i+1,j-1}\right), && (7.81)\\
v_{2i+1,2j+1} &= \frac{3}{4}\left(\frac{3}{4}\tilde{v}_{i,j} + \frac{1}{4}\tilde{v}_{i+1,j}\right) + \frac{1}{4}\left(\frac{3}{4}\tilde{v}_{i,j+1} + \frac{1}{4}\tilde{v}_{i+1,j+1}\right). && (7.82)
\end{aligned}$$

where $i, j = 0, \dots, n$.

Implementation

Listing 7.11 outlines an implementation of the prolongation operator. The stencil (7.82) is realized in lines 12-13.

```
 1 template <typename VC, typename V>
 2 void
 3 np2d_prolongation(const GeMatrix<VC> &vc, GeMatrix<V> &v)
 4 {
 5     int i0 = vc.firstRow()+1;
 6     int j0 = vc.firstCol()+1;
 7     int i1 = vc.lastRow()-1;
 8     int j1 = vc.lastCol()-1;
 9
10     for (int i=i0, I=2*i0; i<=i1; ++i, I+=2) {
11         for (int j=j0, J=2*j0; j<=j1; ++j, J+=2) {
12             v(I+1,J+1)+= 0.75*(0.75*vc(i,  j) + 0.25*vc(i+1,  j))
13                        + 0.25*(0.75*vc(i,j+1) + 0.25*vc(i+1,j+1));
14
15             // analogously update v(I,J), v(I,J+1), v(I+1,J)
16
17         }
18     }
19     np2d_bc(v);
20 }
```

Listing 7.11: Computation of the prolongation.

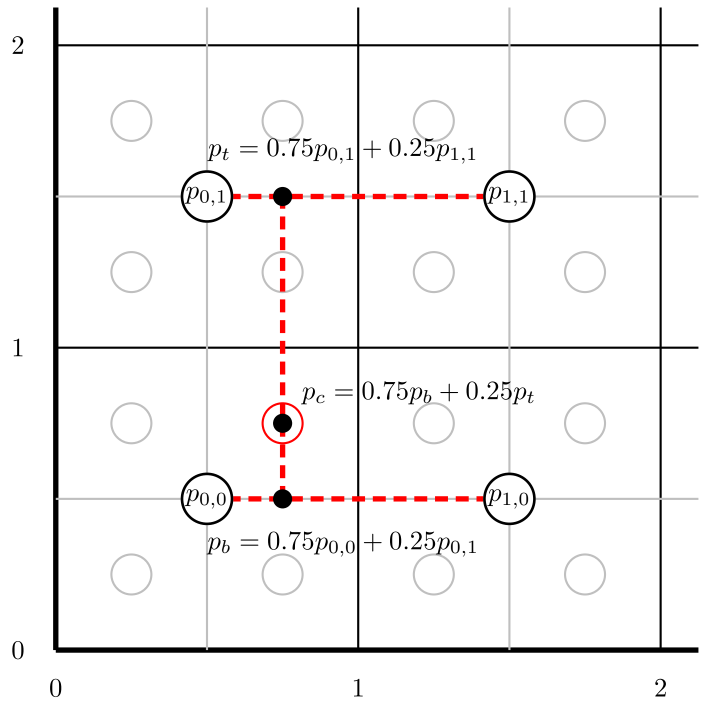

$p_t = 0.75p_{0,1} + 0.25p_{1,1}$
$p_{0,1}$
$p_{1,1}$
$p_c = 0.75p_b + 0.25p_t$
$p_{0,0}$
$p_{1,0}$
$p_b = 0.75p_{0,0} + 0.25p_{0,1}$
0
1
2

Figure 7.6: Stencil for the restriction. The value of the node on the coarse grid (black circle) is the average of nodes on the fine grid (gray circles).

7.6 Parallelization

The parallelization of the Navier-Stokes solver introduced in Section 7.4 following the same pattern illustrated in Section 6.6. The implementation of `DistributedGridVector2D` can be easily generalized for staggered grids as outlined in Listing 7.12.

```
template <bool DirectionX, bool DirectionY>
class DistributedStaggeredGridVector2D
    : public Vector<DistributedStaggeredGridVector2D>
{
    public:
        typedef GeMatrix<FullStorage<double, RowMajor> > Grid;

        DistributedStaggeredGridVector2D(const MpiCart &_mpiCart, int rh);

        // ...

        void setGhostNodes();

        MpiCart         mpiCart;
        int             rh, N;
        int             i0, i1, j0, j1;
        Grid            grid;
        MPI::Datatype   MpiRow, MpiCol;
};
```

Listing 7.12: Staggered grid vector distributed accross the processor grid.

Typedef for grid functions are adapted accordingly:

```
typedef DistributedStaggeredGridVector2D<false, true>  GridVectorU;
typedef DistributedStaggeredGridVector2D<true, false>  GridVectorV;
typedef DistributedStaggeredGridVector2D<true, true>   GridVectorP;
```

The components developed in Section 7.4 can be reused. The wrappers for these components are almost identical except for a subsequent update of ghost nodes:

```
void
computeImpulse(const GridVectorU &u, const GridVectorV &v,
               double dt, double gamma, double re, double fx, double fy,
               GridVectorU &us, GridVectorV &vs)
{
    // ... call low-level component ...
    us.setGhostNodes();
    vs.setGhostNodes();
}
```

The parallelization of the multigrid method requires the parallelization of the smoothing, restriction and prolongation operators. This can be accomplished analogously while reusing the implementations developed in Section 7.5.

7.7 The ‘Pot and Ice’ Problem

The *International Young Physicists’ Tournament* (IYPT)[5] is a competition among teams of secondary school students in their ability to solve complicated scientific problems, to present solutions to these problems in a convincing form and to defend them in scientific discussions, called ‘Physics Fights’. Since 1995 a German team has participated in the IYPT and achieved considerable results. However, for the tournament in 2003 they failed to solve the following problem:

> **Pot and Ice**: It is sometimes argued that to cool a pot efficiently one should put ice above it. Estimate to what extent this is more efficient than if the ice is put under the pot.

In 2006 this problem was treated numerically in collaboration with Bernd Kaifler, a physics student at Ulm University and once member of the German team in 2003. Components from the previous sections can be used for the implementation of a numerical simulation. In the remaining, a two-dimensional model for the ‘Pot and Ice’ problem and its numerical treatment will be outlined only briefly.

Like for the lid driven cavity problem the fluid gets considered on the unit square $\Omega = (0,1)^2$. Beside the velocity and pressure fields the model also incorporates the temperature field

$$T : \Omega \times \mathbb{R}^+ \to \mathbb{R}.$$

In addition to equations (7.10) and (7.11) the flow is governed by the *energy equation*

$$\frac{\partial T}{\partial t} + \vec{u} \cdot \nabla T = \frac{1}{\mathrm{Re}\,\mathrm{Pr}} \Delta T \tag{7.83}$$

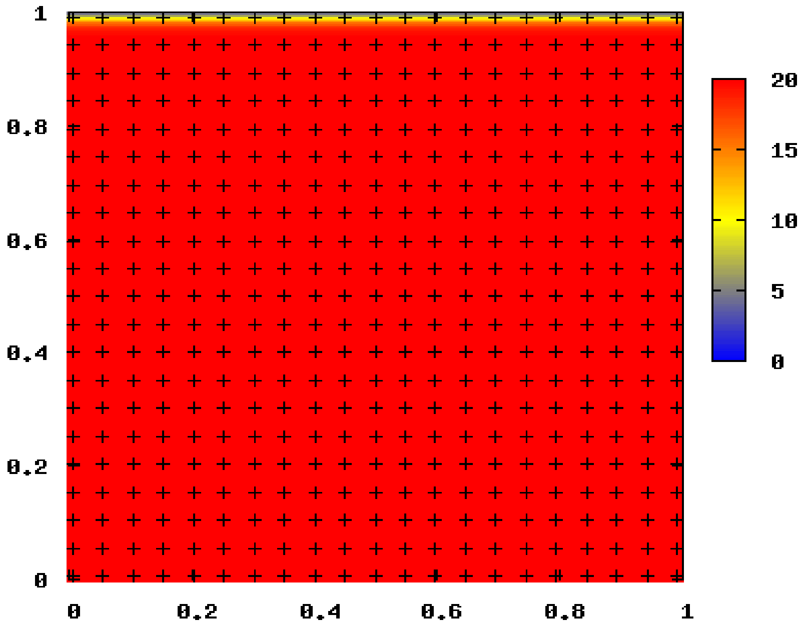
where

$$\mathrm{Pr} = \frac{\nu}{\alpha}$$

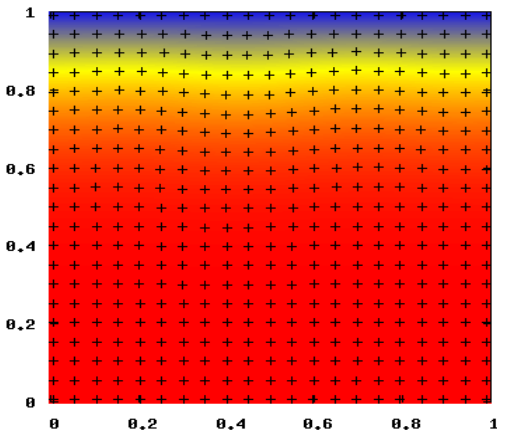
denotes the dimensionless *Prandtl number* [39] which approximates the ratio of viscosity and *thermal diffusivity* α. Equations (7.10), (7.11) and (7.83) are coupled as the volume forces now also include buoyant forces. Using the so called *Boussinesq approximation*[47] the forces on the right-hand side of (7.10) and (7.11) are changed to

$$\vec{F} = \begin{pmatrix} f_x \\ f_y \end{pmatrix} = \begin{pmatrix} f_x \\ f_y(1 - \beta T) \end{pmatrix}, \tag{7.84}$$

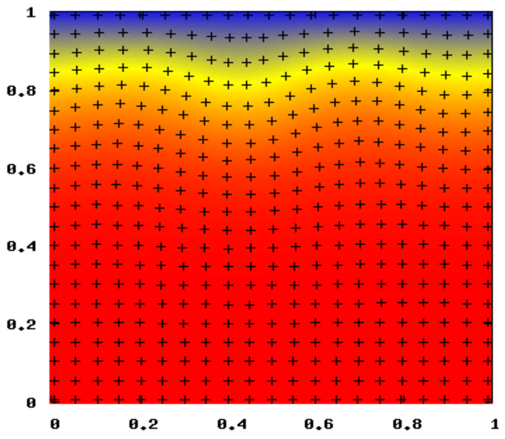
[5]http://www.iypt.org



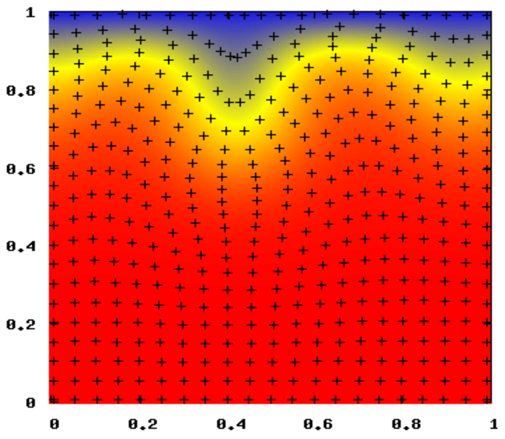
(a) $t = 0.1$



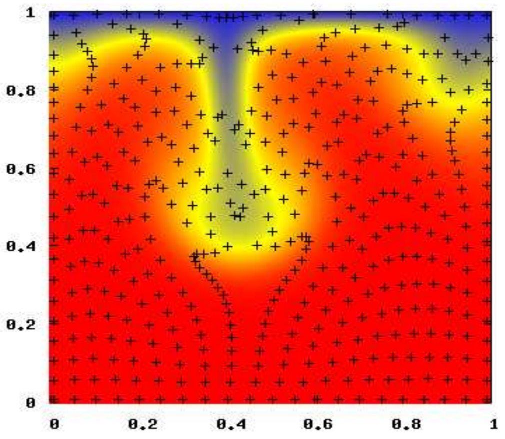
(b) $t = 20$



(c) $t = 21$

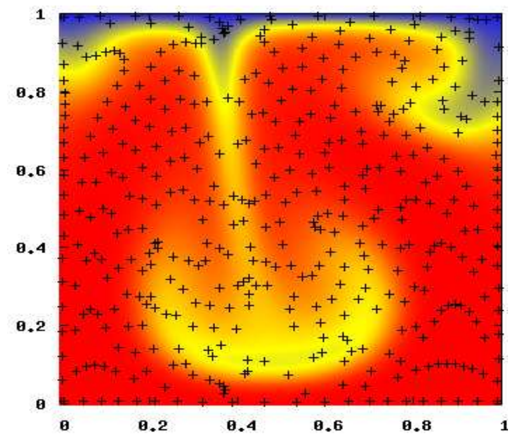


(d) $t = 22$

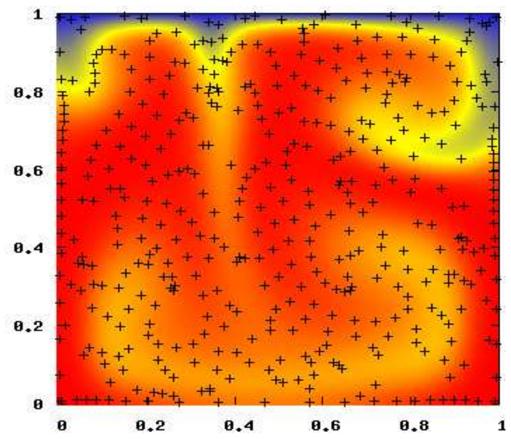


(e) $t = 23$

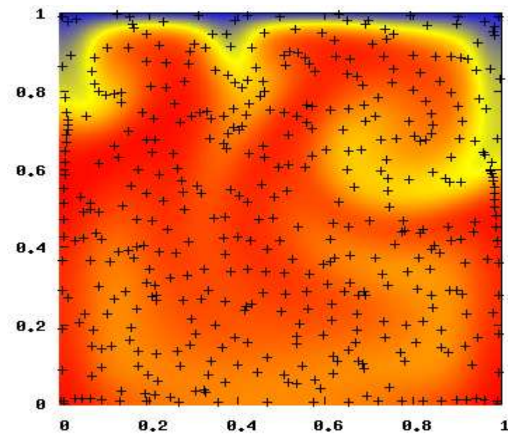
Figure 7.7: Numerical simulation of the ‘Pot and Ice’ problem.



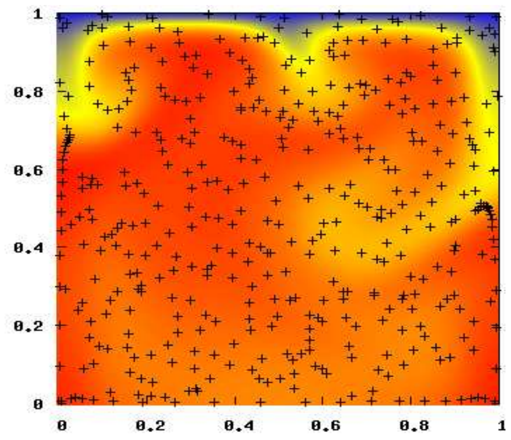
(a) $t = 24$



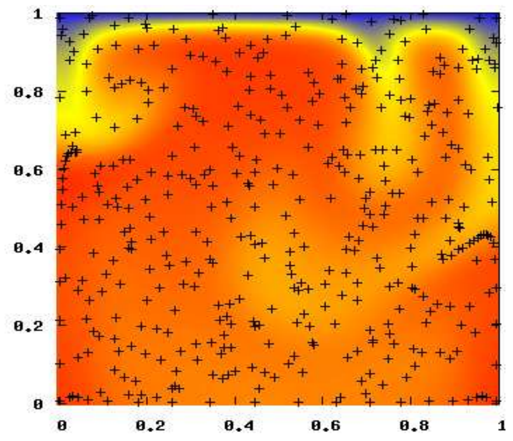
(b) $t = 25$



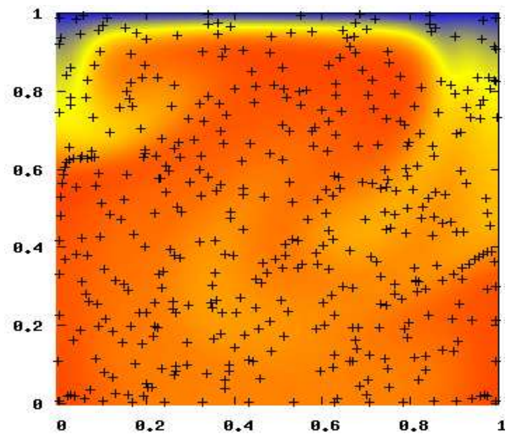
(c) $t = 26$



(d) $t = 27$



(e) $t = 28$



(f) $t = 29$

Figure 7.8: Numerical simulation of the 'Pot and Ice' problem.

where β denotes the dimensionless *thermal expansion coefficient.* Initial and boundary conditions for the velocity field are

$$\begin{aligned} \vec{u}(\vec{x},0) &= \vec{0}, \quad \vec{x} \in \Omega, && (7.85) \\ \vec{u}(\vec{x},t) &= \vec{0}. \quad \vec{x} \in \partial\Omega, t \geq 0 && (7.86) \end{aligned}$$

Let Γ_0 denote the cooled side of the pot (e. g. $\Gamma_0 = \{(x,1) : 0 \leq x \leq 1\}$ if ice is put above). Then Dirichlet boundary conditions are imposed on Γ_0 for cooling the fluid

$$T(\vec{x},t) = 0, \quad \vec{x} \in \Gamma_0, t \geq 0 \tag{7.87}$$

and Neumann boundary conditions

$$\frac{T(\vec{x},t)}{\partial\vec{n}} = 0, \quad \vec{x} \in \partial\Omega\ \Gamma_0, t \geq 0, \tag{7.88}$$

on the remaining sides are used to model an isolated pot. The initial temperature of the fluid is defined through the initial condition

$$T(\vec{x},0) = T_0, \quad \vec{x} \in \Omega. \tag{7.89}$$

The discretization of the above model can be carried out analogously to the lid driven problem using the finite difference method. Figures 7.7 and 7.8 contain different snapshots of a numerical simulation where a pot with water is cooled from above. In this simulation the water initially had a temperature of 20^o C. Subfigures 7.7(a) and 7.7(b) illustrate a typical phenomenon for this problem: At first, a horizontal layer of cooled water spreads out at the top. This is solely caused by diffusion and for several time steps there is no convection in the fluid. Within this period of time the pot gets cooled only slowly. However, as can be deduced from Subfigures 7.7(c) and 7.7(d) this is an unstable state. In physical experiments small agitations or imperfections of the fluid cause the sudden creation of convection. In numerical simulations this is caused by round off errors.

An implementation of a numerical scheme for the ‘Pot and Ice’ problem is included in the examples directory of FLENS. Videos visualizing the numerical simulation are hosted on the FLENS website [45].

7.8 Summary of the Chapter

This chapter illustrated how the FLENS library can be used as an effective building block for the development of scientific software in the field of *computational fluid dynamics* (CFD). It was in particular exemplified how the utilization of FLENS can enhance the collaboration between numerical analysts, specialist in high-performance computing and researchers in the CFD domain. Initially, mathematical models of some standard test problems in fluid dynamics were outlined. Subsequently, the finite difference method was applied to obtain a simple numerical scheme. Using FLENS as a building block enabled the rapid development a simple Navier-Stokes solver. This was possible for different reasons. On the one hand, the implementation of the numerical schemes could be realized close to the mathematical notation. On the other hand, FLENS provided reusable components like the multigrid method which could be easily adapted. While this illustrated the effective collaboration between the domains of CFA and numerical analysis in this development process, the performance aspect was initially ignored. However, FLENS endorsed the development of software components which can be tuned for performance subsequently. Furthermore, the parallelization of the Navier-Stokes solver could be realized immediately by reusing these components.

8 The Finite Element Method

In this chapter the two-dimensional Dirichlet-Poisson problem again serves as a model problem. Compared to Chapter 6 the considered domain is allowed to be more general to motivate the usage of the Finite Element Method (FEM) for discretization. The scope of this chapter was motivated by [1]. In this paper it was demonstrated how Matlab can be utilized to implement the FEM (for a slightly more general problem) in a very compact and expressive form. The presentation of an implementation based on FLENS intends to demonstrate how libraries dedicated to the FEM could take advantage of FLENS by using it as a building block. Further, the material developed for this chapter can be used for programming assignments within introductory courses on FEM.

Based on [11], Section 8.1 briefly covers the variational formulation of the Dirichlet-Poisson problem. Following [43], utilizing the FEM for the discretization and practical aspects for the later implementation are introduced in Sections 8.2 and 8.3.3. The implementation of the FEM is illustrated in Section 8.4. Sparse matrix types and solvers for sparse systems of linear equations in FLENS are outlined in Section 8.5.

8.1 The Dirichlet-Poisson Problem and its Variational Formulation

Consider a two-dimensional domain $\Omega \subset \mathbb{R}^2$ where Ω is assumed to be a bounded Lipschitz domain with piecewise polygonal boundary $\Gamma = \partial\Omega$. Defining L as the Laplace operator, i. e. $Lu := \Delta u$, the Dirichlet-Poisson problem reads:

For given functions $f \in C^0(\Omega)$ and $g \in C^0(\Gamma)$ find $u \in C^2(\Omega) \cap C^0(\bar{\Omega})$ that satisfies:

$$-Lu = f \quad \text{in } \Omega, \tag{8.1}$$

$$u = g \quad \text{on } \Gamma. \tag{8.2}$$

Existence of a solution u depends on the regularity of the boundary Γ. However, regularity of a bounded Lipschitz domain is not sufficient as shown by a counter example in [11] (page 33). Furthermore, even if a solution exists, higher derivatives of u might be unbounded, making convergence analysis of numerical methods impossible. This motivates to consider a generalized formulation of the problem.

8.1.1 Variational Formulation

For $u \in C^2(\Omega) \cap C^0(\bar{\Omega})$ it follows from Green's theorem that for all $v \in C_0^\infty(\Omega)$

$$(-Lu, v)_0 = (-\Delta u, v)_0 = \int_\Omega (-\Delta u)\, v \mathrm{d}x = \int_\Omega \nabla u \nabla v \,\mathrm{d}x =: a(u, v). \tag{8.3}$$

Using the bilinear form $a(\cdot,\cdot)$ it follows that a solution u of (8.1) and (8.2) satisfies

$$a(u, v) = (f, v)_0 \quad \text{for } v \in C_0^\infty(\Omega) \tag{8.4}$$

Equation (8.4) motivates the variational formulation. As shown for example in [11], a unique solution for problem (8.4) exists if u and v are taken from Sobolev spaces $H^1(\Omega)$ and $H_0^1(\Omega)$ respectively. This leads to the variational formulation of the Poisson problem (8.1) and (8.2):

Find $u \in U := H^1(\Omega)$ such that

$$\begin{aligned} a(u,v) &= (f,v)_0 \quad \text{for } v \in V := H_0^1(\Omega) & (8.5)\\ u - g &\in H_0^1(\Omega). & (8.6) \end{aligned}$$

For $g \in H^1(\Omega)$ and $f \in L^2(\Omega)$ this is a well-posed problem. Existence of a solution can be proven using the Lax-Milgram theorem. The space U is referred to as *trial space* and V as *test space*. Equation (8.6) can be regarded as weak formulation of the boundary condition (8.2) with g extended to $H^1(\Omega)$.

The weak boundary condition (8.6) motivates the substitution $\tilde{u} := u - g$ and leads to the homogenized variational problem, where test and trial spaces coincide:

Find $\tilde{u} \in U = V := H_0^1(\Omega)$ such that

$$a(\tilde{u},v) = (f,v)_0 - a(g,v) \quad \text{for } v \in V. \tag{8.7}$$

Hence we can concentrate on the homogeneous problem in the sequel. Then $u := \tilde{u} + g$ solves the variational problem associated to the Poisson problem.

8.1.2 Strong and Weak Solutions

A solution of (8.1) and (8.2) is denoted as a *strong* or *classic solution* whereas a solution of the associated variational problem (8.5) and (8.6) is referred to as a *weak solution*. These terms are justified as every strong solution is also a weak solution. Many practical problems incorporate restrictions that merely allow for proving the existence of a weak solution. For partial differential equations modeling physical problems, it often can be shown that weak solutions constitute adequate solutions of the original problem. Further, a weak solution of the Poisson problem that lies in the space $C^2(\Omega) \cap C^0(\bar{\Omega})$ is also a strong solution.

8.2 Discretization: Finite Element Method

Numerical solutions are obtained by solving a discretized version of (8.7). Following the Galerkin method, discretization is achieved by replacing trial and test spaces with finite dimensional spaces. Different methods for the construction of finite dimensional spaces approximating trial and test spaces exist. One of these methods is the Finite Element Method exemplified in the following.

Using the Finite Element Method, the finite dimensional spaces are constructed based on a triangulation of the domain Ω. In the following, $\mathcal{T}_h$ denotes a *regular triangulation* that consists merely of triangles while in general it also might contain quadrilaterals. The index h denotes the grain size of the mesh, e. g. the maximal diameter of a triangle. Regular triangulation means that

1. elements of $\mathcal{T}_h$ are closed triangles,
2. the domain Ω is covered by the triangles, i. e. $\bigcup_{T \in \mathcal{T}_h} T = \bar{\Omega}$ and
3. intersection of two different triangles results in either a point, a common edge or is empty.

A triangle $T \in \mathcal{T}_h$ is referred to as *element* and its vertices as *nodes*.

The simplest variant of the Finite Element Method is obtained by replacing the space $U = H_0^1(\Omega)$ with

$$U_h := \{\varphi \in C(\Omega) \,:\, \varphi_{|T} \text{ is linear on } T \in \mathcal{T}_h\}.$$

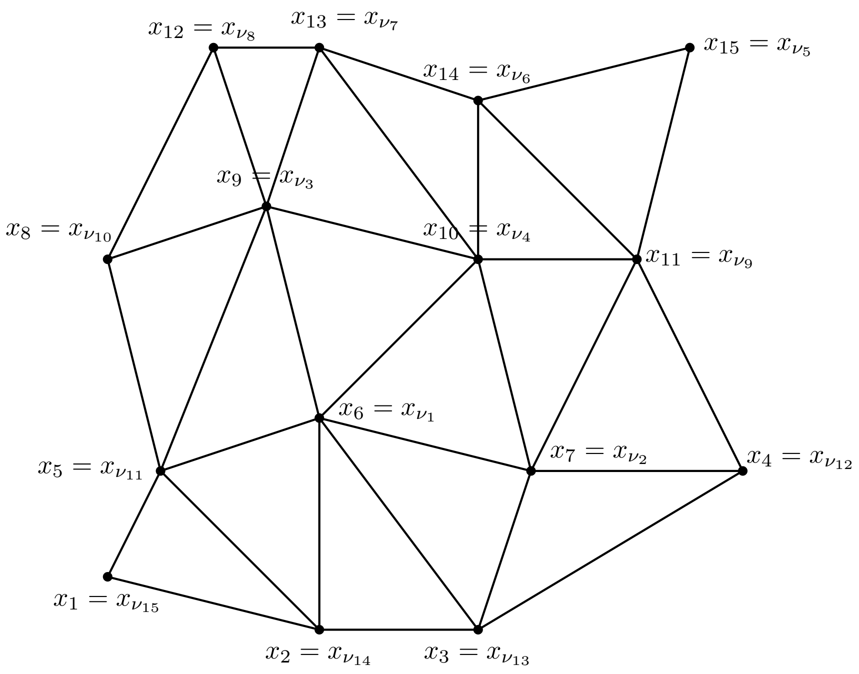


Figure 8.1: Example for a triangulation with a total of $N = 15$ nodes and $M = 4$ interior nodes. Interior nodes $\{x_6, x_7, x_9, x_{10}\}$ can formally be distinguished from boundary nodes $\{x_1, x_2, x_3, x_4, x_5, x_8, x_{11}, x_{12}, x_{13}, x_{14}, x_{15}\}$ using the permutation ν.

Using interpolation, a basis $(\varphi_1, \dots, \varphi_N)$ for U_h can be constructed such that

$$\varphi_i(x_j) = \delta_{i,j} \qquad j = 1, \dots, N,$$

for nodes $x_1, \dots, x_N \in \mathcal{T}_h$. This is called a *nodal basis* of U_h. Because of the boundary condition it is important to distinguish whether a node lies on the boundary or in the interior of the domain. As illustrated in Figure 8.1 this can be achieved by defining a permutation

$$\nu : \{1, \dots, N\} \to \{1, \dots, N\} \quad \text{such that} \begin{cases} x_{\nu_1}, \dots, x_{\nu_M} & \in \Omega, \\ x_{\nu_{M+1}}, \dots, x_{\nu_N} & \in \partial\Omega, \end{cases} \tag{8.8}$$

where $M < N$ and $\nu_l := \nu(l)$ for $l = 1, \dots, N$. Now elements of U_h can be represented[1] as

$$u_h(x) = \underbrace{\sum_{l=1}^{M} u_l \varphi_{\nu_l}(x)}_{=:\tilde{u}_h(x)} + \underbrace{\sum_{l=M+1}^{N} u_l \varphi_{\nu_l}(x)}_{=:g_h(x)} \tag{8.9}$$

The sum in (8.9) corresponds to the substitution used in (8.7). If g_h is chosen as the interpolant of the boundary function g, its coefficients are uniquely given as

$$u_l = g(x_{\nu_l}), \quad l = M+1, \dots, N. \tag{8.10}$$

Hence u_h is uniquely determined through $\tilde{u}_h \in V_h := \operatorname{span}\{\varphi_{\nu_l} : 1 \le l \le M\} \subset V$. Thus, the discretized problem of (8.7) can be stated as:

[1] Were possible and appropriate the following loose convention will be used: Indices k and l are used in conjunction with the reordering introduced by ν. Whereas i and j refer to the original index order.

Find $\tilde{u} = (u_1, \ldots, u_M) \in \mathbb{R}^M$ such that

$$\sum_{l=1}^{M} u_l \, a(\varphi_{\nu_l}, \varphi_{\nu_k}) = (f, \varphi_{\nu_k})_0 - \sum_{l=M+1}^{N} u_l \, a(\varphi_{\nu_l}, \varphi_{\nu_k}), \quad k = 1, \ldots, M. \tag{8.11}$$

Let $(u_1, \ldots, u_M)$ be the solution of (8.11) then $u_h(x_{\nu_l}) = u_l$ for $l = 1, \ldots, N$. Therefore, nodal values of the numerical solution u_h are given by

$$u_h(x_i) = u_{\eta_i}, \quad i = 1, \ldots, N, \tag{8.12}$$

where η denotes the inverse permutation of ν. For the later proceeding and in particular for the implementation, η is used to determine whether a node x_i lies on the boundary or in the interior of the domain:

$$x_i \in \begin{cases} \Omega, & \text{if } \eta_i \leq M, \\ \partial\Omega, & \text{if } \eta_i > M. \end{cases} \tag{8.13}$$

8.3 Assembling

Problem (8.11) is equivalent to a linear system of equations $Au = b$ with

$$A = \big(a(\varphi_{\nu_k}, \varphi_{\nu_l})\big)_{k,l} \in \mathbb{R}^{M \times M} \tag{8.14}$$

denoted as *stiffness matrix* and right-hand side

$$b = \Big((f, \varphi_{\nu_k})_0 - \sum_{l=M+1}^{N} a(\varphi_{\nu_l}, \varphi_{\nu_k})\, u_l\Big)_k \in \mathbb{R}^M. \tag{8.15}$$

For elliptic PDEs like the Poisson problem the stiffness matrix A is positive definite. This readily follows as for this problem the bilinear form is positive by definition (8.3). Further the choice of the nodal basis of V_h implies that A is sparse.

Computing the elements of A and b is referred to as *assembling*. Technical details used in the later implementation are illustrated in the following.

8.3.1 Evaluation of the Bilinear Form

Both the stiffness matrix (8.14) as well as the right-hand side (8.15) require the evaluation of

$$\begin{aligned} a(\varphi_i, \varphi_j) &= \int_\Omega \nabla\varphi_i(x) \cdot \nabla\varphi_j(x) \,\mathrm{d}x \\ &= \sum_{m=1}^{|\mathcal{T}|} \underbrace{\int_{T_m} \nabla\varphi_i(x) \cdot \nabla\varphi_j(x) \,\mathrm{d}x}_{=:A_{i,j}^{(m)}}. \end{aligned} \tag{8.16}$$

Due to its construction, a base function φ_i is only non-zero on triangles that contain x_i as a node. Therefore, a summand $A_{i,j}^{(m)}$ in (8.16) can only be non-zero if x_i and x_j are nodes of a triangle T_m. Hence it is favorable to iterate over all triangles $T_m \in \mathcal{T}$ as this allows for computing only the the non-zero summands $A_{i,j}^{(m)}$ in (8.16). For a triangle T_m with vertices $(x_{i_1}, x_{i_2}, x_{i_3})$ non-zero entries of $A^{(m)}$ are given by

$$A_{i_r,i_s}^{(m)} = \int_{T_m} \nabla\varphi_{i_r}(x) \cdot \nabla\varphi_{i_s}(x) \,\mathrm{d}x, \quad r, s = 1, 2, 3. \tag{8.17}$$

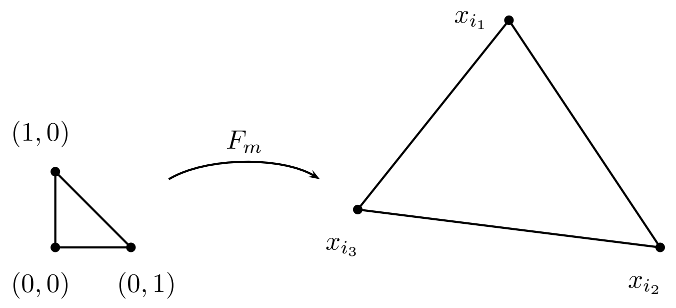


Figure 8.2: Mapping the reference triangle $\hat{T}$ onto a triangle T_m with vertices $(x_{i_1}, x_{i_2}, x_{i_3})$.

What remains is the numerical evaluation of the integral in (8.17) which is performed on a reference triangle $\hat{T}$ with vertices $\hat{x}_1 = (0,0)^T$, $\hat{x}_2 = (1,0)^T$ and $\hat{x}_3 = (0,1)^T$. Defining the affine-linear mapping (see Figure 8.2)

$$F_m(\hat{x}) := S\hat{x} + x_{i_1} \quad \text{with } S := (x_{i_2} - x_{i_1}, x_{i_3} - x_{i_1})$$

it follows that $F_m(\hat{T}) = T_m$ and in particular $F_m(\hat{x}_r) = x_{i_r}$ for $r = 1,2,3$. This mapping allows the reduction of a base function φ_{i_r} to a base function defined on $\hat{T}$ as follows:

$$N_r : \hat{T} \to \mathbb{R}, \quad N_r(\hat{x}) := \varphi_{i_r}(F_m(\hat{x})), \quad r = 1,2,3.$$

These base functions on $\hat{T}$ are explicitly known:

$$N_1(\hat{x}) = 1 - (1,1)\hat{x}, \quad N_2(\hat{x}) = (1,0)\hat{x}, \quad N_3(\hat{x}) = (0,1)\hat{x}.$$

Applying the mapping F_m to (8.17) leads to

$$A^{(m)}_{i_r,i_s} = \int_{\hat{T}} C\, \nabla N_r(\hat{x}) \cdot \nabla N_s(\hat{x})\, |\det(S)|\, \mathrm{d}\hat{x}, \tag{8.18}$$

where $C = (S^T S)^{-1}$. As the partial derivatives of N_r are constant the value of this integral can be stated explicitly in terms of C and S. Consequently all contributions arising from T_m can be computed in a single sweep (see [43]):

$$\tilde{A}^{(m)} := \left(A^{(m)}_{i_r,i_s}\right)_{r,s=1,2,3} = \gamma_1 S_1 + \gamma_2 S_2 + \gamma_3 S_3. \tag{8.19}$$

where $\gamma_1 = c_{11}|\det(S)|$, $\gamma_2 = c_{12}|\det(S)|$, $\gamma_3 = c_{13}|\det(S)|$ and

$$S_1 = \tfrac{1}{2}\begin{pmatrix} 1 & -1 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \quad S_2 = \tfrac{1}{2}\begin{pmatrix} 2 & -1 & -1 \\ -1 & 0 & 1 \\ -1 & 1 & 0 \end{pmatrix}, \quad S_3 = \tfrac{1}{2}\begin{pmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{pmatrix}.$$

8.3.2 Integrals for the Right-Hand Side

Assembling the right-hand side (8.15) requires the evaluation of

$$(f, \varphi_i)_0 = \int_\Omega f(x)\varphi_i(x)\, \mathrm{d}x = \sum_{m=1}^{|\mathcal{T}|} \underbrace{\int_{T_m} f(x)\varphi_i(x)\, \mathrm{d}x}_{=:b_i^{(m)}},$$

where i refers to an internal node (i.e. $\eta_i \leq M$). Again, a summand $b_i^{(m)}$ can only be non-zero if x_i is a node of triangle T_m. For a triangle $T_m = (x_{i_1}, x_{i_2}, x_{i_3})$ the relevant entries are computed on the reference triangle by

$$\begin{aligned} b_{i_r}^{(m)} &= \int_{T_m} f(x)\varphi_{i_r}(x)\,\mathrm{d}x \\ &= \int_{\hat{T}} \hat{f}(\hat{x}) N_r(\hat{x})\,|\det(S)|\,\mathrm{d}x, \quad r = 1,2,3 \end{aligned} \tag{8.20}$$

where $\hat{f}(\hat{x}) := f(F_m(\hat{x}))$. In general (8.20) can not be computed exactly and instead a quadrature formula is used. This leads to

$$\tilde{b}_r^{(m)} := b_{i_r}^{(m)} \approx \sum_{p=1}^{P} \omega_p \hat{f}(\hat{y}_p) N_r(\hat{y}_p)\,|\det(S)|, \quad r = 1,2,3 \tag{8.21}$$

with weights $\omega_1, \ldots, \omega_P$ and quadrature points $\hat{y}_1, \ldots, \hat{y}_P$. For the trapezoidal rule these are given as $\omega_p = \frac{1}{6}$ and $\hat{y}_p = \hat{x}_p$ for $p = 1,2,3$ and therefore

$$b_{i_r}^{(m)} \approx \frac{1}{6} \hat{f}(\hat{x}_r)\,|\det(S)|. \tag{8.22}$$

8.3.3 Assembling the System of Linear Equations

An assembling routine starts with an ‘empty’ stiffness matrix A and a zero-initialized right-hand side vector b. For each $T_m \in \mathcal{T}$ entries of $\tilde{A}^{(m)}$ and $\tilde{b}^{(m)}$ are computed and then used to sequentially update A and b.

Due to the fact that entries of $\tilde{A}^{(m)}$ and $\tilde{b}^{(m)}$ have to be addressed to corresponding entries of A and b, quite some effort in bookkeeping is required. If nodes of T_m are denoted as $(x_{i_1}, x_{i_2}, x_{i_3})$, a corresponding mapping is given by

$$\begin{aligned} \tilde{A}_{r,s}^{(m)} &= A_{i_r,i_s}^{(m)} = A_{\nu_k,\nu_l}^{(m)}, \\ \tilde{b}_r^{(m)} &= b_{i_r}^{(m)} = b_{\nu_k}^{(m)}, \end{aligned}$$

where $r, s \in \{1,2,3\}$ and $k := \eta_{i_r}, l := \eta_{i_s}$. From (8.14) and (8.15) it follows that

$$A_{\nu_k,\nu_l}^{(m)} \text{ contributes to } \begin{cases} A_{k,l} & \text{if } k, l \leq M, \\ b_k & \text{if } k \leq M \text{ and } l > M \end{cases} \tag{8.23}$$

and analogously

$$b_{\nu_k}^{(m)} \text{ contributes to } b_k \text{ if } k \leq M. \tag{8.24}$$

Pseudo-code for the assembling procedure is given in Algorithm 8.1. The algorithm takes advantage of the fact that $\tilde{A}^{(m)}$ is symmetric. Only entries $\tilde{A}_{r,s}^{(m)}$ and $\tilde{b}_r^{(m)}$ for $1 \leq s \leq r \leq 3$ are accessed.

Algorithm 8.1 (Assembling Stiffness Matrix and Right-Hand Side).

```
for each $T_m \in \mathcal{T}$
   denote nodes of $T_m$ as $(x_{i_1}, x_{i_2}, x_{i_3})$
   compute $\tilde{A}^{(m)}$ and $\tilde{b}^{(m)}$
   for $r = 1,2,3$
      $k := \eta_{i_r}$
      if $k \leq M$
         $b_k \to b_k + \tilde{b}_r^{(m)}$
      end
```

```
        for $s = r, ..., 3$
            $l := \eta_{i_s}$
            if $k > M$ and $l > M$
                next s
            end
            if $k \leq M$ and $l \leq M$
                $A_{\mathtt{k,l}} \to A_{\mathtt{k,l}} + \tilde{A}^{(m)}_{r,s}$
            else
                $\tilde{k} := \min\{k, l\}$
                $\tilde{l} := \max\{k, l\}$
                $b_{\tilde{k}} \to b_{\tilde{k}} - u_{\tilde{l}}\tilde{A}^{(m)}_{r,s}$
            end
        next s
    next r
next $T_m$
```

For the sake of efficiency an implementation has to exploit the sparsity as well as the symmetry of A. With respect to sparsity an implementation can utilize adequate matrix types as illustrated in Section 8.5. Symmetry can be exploited for example, by storing either the upper or lower triangular part of A. However, the concrete technical realization in general depends on the type of solvers that gets used for the sparse system of linear equations.

8.4 Implementation

An implementation of the finite element method basically consists of the following components:

1. A mesh generator creating a triangulation $\mathcal{T}$ for a given domain Ω.
2. Data structures for accessing and manipulating elements of the mesh.
3. Routines for assembling and solving the linear system of equations.
4. Post-processing routines, e. g. for the visualization of the numerical solutions.

Merely the implementation of the components outlined in 2. and 3. is covered in this work. The basic intention in this respect is to demonstrate how the extensibility of FLENS and the reusability of its components can be exploited. Further, mesh generation for rather general domains is a sophisticated topic on its own. The implementation therefore assumes that the triangulation can be created by an external software and is available in a common file format as outlined in Section 8.4.1.

The implementation of the FEM for the Dirichlet-Poisson problem basically consist of the `Mesh` and `FEM` classes introduced in Subsection 8.4.1 and 8.4.2 respectively. The classes are not part of the official FLENS release and merely intend to indicate how dedicated finite element libraries could make use of FLENS.

Within an introductory course for the FEM, the material covered in this section could be used for a one-week homework assignment. The `Mesh` class could be provided as a teaching aid and the students assigned to implement the methods of the `FEM` class.

8.4.1 The Mesh Class

A mesh specifies the triangulation of the domain Ω through a list of nodes and a related list of triangles. Hereby 'related' means that for nodes $(x_1, y_1), \dots, (x_N, y_N)$ a triangle $T \in \mathcal{T}$ is defined through a triple of node indices. For example, (i_1, i_2, i_3) refers to a triangle with vertices $(x_{i_1}, y_{i_1}), (x_{i_2}, y_{i_2})$ and (x_{i_3}, y_{i_3}).

Data structures for the triangulation are encapsulated in the `Mesh` class. The class design was driven by considering what operations are most essential for assembling the stiffness matrix and the right-hand side.

Interface

The interface is intended to provide the minimal set of methods needed to realize the FEM. Hence methods needed to perform a mesh refinement are not intended in this example. This means after creation of a `Mesh` object subsequent changes to the mesh are not possible, e.g. adding, removing or changing nodes or triangles. This is only a self-imposed restriction for the sake of keeping interface and implementation as simple as possible.

When a `Mesh` object gets instantiated, nodes and triangles are imported from files and methods provide read access to nodes and triangles. Further, the permutations ν and η defined in Section 8.2 as well as a method that returns the number of free nodes M are available through the following interface:

```
 1 class Mesh
 2 {
 3     public:
 4     // ...
 5
 6         Mesh(const char *nodesFile, const char *trianglesFile);
 7
 8         int numTriangles() const;
 9         Triangle triangle(int index) const;
10
11         int numNodes() const;
12         Node node(int index) const;
13
14         int numFreeNodes() const;
15         int nu(int index) const;
16         int eta(int index) const;
17
18     // ...
19 };
```

Listing 8.1: Mesh class for handling the triangulation.

So far types `Triangles` and `Node` (in lines 9 and 12) are not yet specified. However, for understanding the interface it is only relevant that these types comply to the following interface convention: Both classes have an overloaded call operator `()`. For a `Triangle` object `t` the node index of the r-th vertex can be retrieved by `t(r)`. Using the `node` method these indices can be used to retrieve corresponding node objects. Similarly the i-th coordinate of a `Node` object `n` can be accessed by `n(i)`. Usage becomes self-explanatory considering a little code example for printing coordinates of the triangle vertices line-by-line:

```
for (int m=1; m<=mesh.numTrinagles(); ++m) {
    Triangle t = mesh.triangle(m);
    for (int r=1; r<=3; ++r) {
        Node n = mesh.node(t(r));
        cout << "(" << n(1)
             << "," << n(2) << ") ";
    }
    cout << endl;
}
```

Obviously the restriction of interface to two dimensions could easily be generalized without much effort.

Implementation

For the following implementation of the `Mesh` class it is assumed that the triangulation is stored in two separate text files following a simple and very common format [43]. One file stores node coordinates line-by-line whereas the total number of nodes is stored in the first line:

```
<Number of nodes>
<x1> <y1>
<x2> <y2>
...
<xN> <yN>
```

The topology of the mesh is described in the second file. The first line contains the total number of triangles followed by lines of index triples referring to vertices:

```
<Number of triangles>
<t1_1> <t1_2> <t1_3>
<t2_1> <t2_2> <t2_3>
...
```

Most mesh generators support a slightly more general format in order to support more than two dimensions. However it is fairly easy to convert these formats using simple shell or PERL scripts.

The `Mesh` constructor reads the files and stores nodes and triangles row-wise in matrices `nodes` and `triangles`. Matrix types are in both cases of type `GeMatrix` using a full storage scheme. For `nodes` the element type is `double` while for `triangles` (storing only indices) it is `int`. For convenience typedefs `NodeTable` and `TriangleTable` are defined.

Now the i-th node is simply a vector view of the i-th row of `nodes`. Similarly vector views come in place to access single triangles. Hence types for `Node` and `Triangle` are defined as adequate types for vector views:

```
class Mesh
{
    public:
        typedef GeMatrix<FullStorage<double, RowMajor> >     NodeTable;
        typedef GeMatrix<FullStorage<int, RowMajor> >        TriangleTable;

        typedef const DenseVector<ConstArrayView<int> >      Triangle;
        typedef const DenseVector<ConstArrayView<double> >   Node;
        // ....

    private:
        NodeTable       nodes;
        TriangleTable   triangles;
        // ...
};
```

Listing 8.2: Typedefs in the mesh class for nodes and triangles.

The implementation of method `node(int)` simply creates and returns a view referencing the i-th row of `nodes`:

```
Node node(int i) const
{
    return nodes(i,_);
}
```

Listing 8.3: Mesh class: implementation of method `node(int)`.

Method `triangle` is implemented analogously. Permutations ν and η are internally realized through integer vectors. Vector `_nu` stores $\nu_1, \dots, \nu_N$ and `_eta` values $\eta_1, \dots, \eta_N$:

```
class Mesh
{
    // ...

    private:
    // ...
        int                         _numFreeNodes;
        DenseVector<Array<int> >    _nu, _eta;
};
```

Listing 8.4: Mesh class: Vectors for storing the permutations ν and η.

The number of free nodes M is stored in variable _numFreeNodes. Methods nu, eta and numFreeNodes are implemented as simple wrappers for accessing the internal data structures.

The permutation vectors are setup in the constructor after importing the mesh. Hereby the nodes belonging to the boundary are determined in a first step. This happens by iterating over all triangles and thereby counting occurrences of edges. If an edge only occurs once then both endpoints belong to the boundary. Based on the set of boundary nodes the reordering conforming to (8.8) can be determined in a second step.

8.4.2 FEM for the Poisson Problem

Methods for assembling the linear system of equations are gathered in class FEM. Solving is carried out using an external solver. Usage of the Mesh class allows an implementation of the assembling routine that directly reflects the pseudo-code Algorithm 8.1. Before going into relevant details of the implementation, a look on the FEM class interface already gives a picture of the basic idea.

Interface

Functions f and g for the Poisson problem (8.1, 8.2) can be implemented by the user outside the FEM class. These functions receive coordinates x as const Node object and return the function value as double.

Using the FEM class the Dirichlet-Poisson problem can be solved in three steps:

1. Creation of a FEM object which (besides some internal data structures) initializes a Mesh object.
2. Assembling the linear system of equations.
3. Solving these and writing the solution to a file.

Listing 8.5 illustrates the interface of the FEM class. Method assemble (line 11) assembles A and b. The stiffness matrix gets stored in a matrix of type SparseSymmetricMatrix<CSR<T> > (line 23). This is a FLENS matrix type for symmetric sparse matrices which gets introduced in more detail in Section 8.3.3. This should cause no confusion as its usage is very similar to those matrix types introduced in Chapter 4.

```
 1 class FEM
 2 {
 3     public:
 4         typedef Mesh::Triangle    Triangle;
 5         typedef Mesh::Node        Node;
 6         typedef double (*Func)(const Node &);
 7
 8         FEM(const char *nodesFile, const char *trianglesFile,
 9             const Func &f, const Func &g);
10
11         void assemble();
12
13         void solve();
14
15         void writeSolution(const char *filename);
16
17     private:
18         void compute_Am_bm(const Triangle &t);
19
20         Mesh                                    mesh;
21         Func                                    f, g;
22         DenseVector<Array<double> >             u, b;
23         SparseSymmetricMatrix<CSR<double> >     A;
24
25         DGeMatrix                               Am;
26         DenseVector<Array<double> >             bm;
27
```

```
        typedef GeMatrix<FullStorage<double, RowMajor> > DGeMatrix;
        std::vector<DGeMatrix>               S;

};
```

Listing 8.5: Class for the Finite Element Method.

The assemble method makes use of method `compute_Am_bm` which computes $\tilde{A}^{(m)}$ and $\tilde{b}^{(m)}$ according to (8.19) and (8.21). Matrices S_1, S_2 and S_3 from (8.19) are stored in a STL vector `S`, which gets initialized by the constructor. The result of `compute_Am_bm` gets stored in `Am` and `bm` serving as quasi-global variables within the `FEM` object.

Method `solve` calls an external solver for the linear system of equations to find a solution of (8.11). In a slightly more flexible implementation the solver might be passed via a template parameter or function pointer. According to (8.12) `writeSolution` writes the numerical solution to a file.

From the user's point of few the Poisson problem can be solved in 'three and a half lines' of code:

```
FEM fem("nodes", "triangles", f, g);
fem.assemble();
fem.solve();
fem.writeSolution("sol");
```

But obviously the real magic lies in the implementation of `compute_Am_bm` and `assemble` as shown next.

Implementation

In order to compute $\tilde{A}^{(m)}$ according to (8.19) matrix $C = (S^T S)^{-1}$ can be computed by applying Laplace's formula. For $S = (z_1, z_2) := (x_2 - x_1, x_3 - x_1)$ it follows that

$$C = \frac{1}{det(S)^2}\begin{pmatrix} z_2^T z_2 & -z_1^T z_2 \\ -z_2^T z_1 & z_1^T z_1 \end{pmatrix}.$$

For the computation of $\tilde{b^{(m)}}$ the trapezoid rule (8.22) is used. Following rather closely the mathematical notation the implementation is given by:

```
void FEM::compute_Am_bm(const Triangle &t)
{
    const Node &x1 = mesh.node(t(1)),
               &x2 = mesh.node(t(2)),
               &x3 = mesh.node(t(3));

    DenseVector<Array<double> > z1 = x2-x1,
                                z2 = x3-x1;

    double absDetZ = std::abs(det(z1,z2));

    double gamma[3];
    gamma[0] =  dot(z2, z2)/absDetZ,
    gamma[1] = -dot(z1, z2)/absDetZ,
    gamma[2] =  dot(z1, z1)/absDetZ;

    Am = gamma[0]*S[0] + gamma[1]*S[1] + gamma[2]*S[2];

    bm(1) = f(x1)*absDetZ/6;
    bm(2) = f(x2)*absDetZ/6;
    bm(3) = f(x3)*absDetZ/6;
}
```

Listing 8.6: Computing entries of the local stiffness matrix and right-hand side.

The following implementation of the assembling routine closely matches the pseudo-code given in Algorithm 8.1:

```
 1 void
 2 FEM::assemble()
 3 {
 4     int M = mesh.numInteriorNodes();
 5     for (int m=1; m<=mesh.numTriangles(); ++m) {
 6         Triangle t = mesh.triangle(m);
 7         compute_Am_bm(t);
 8         for (int r=1; r<=3; ++r) {
 9             int k=mesh.eta(t(r));
10             if (k<=M) {
11                 b(k) += bm(r);
12             }
13             for (int s=r; s<=3; ++s) {
14                 int l=mesh.eta(t(s));
15                 if ((k>M) && (l>M)) {
16                     continue;
17                 }
18                 if ((k<=M) && (l<=M)) {
19                     A(k,l) += Am(r,s);
20                 } else {
21                     int k_ = std::min(k,l);
22                     int l_ = std::max(k,l);
23                     b(k_) -= Am(r,s)*u(l_);
24                 }
25             }
26         }
27     }
28     A.finalize();
29 }
```

Listing 8.7: Assembling routine.

The fact that `A` is not of some general matrix type becomes only evident in the statement on line 28: '`A.finalize()`'. Details about concepts and the implementation of sparse matrix types in FLENS and solvers for corresponding linear systems of equations are given next.

8.5 Solvers for Sparse Linear Systems of Equations

Recall, in Chapters 6 and 7 the finite difference method has led to coefficient matrices exhibiting a very special structure. In these chapters dedicated matrix types were implemented exploiting this structure. By utilizing these matrix types it was possible that numerical methods for solving the systems of linear equations could achieve optimal efficiency.

The finite element method leads to sparse coefficient matrices with a more general structure. Hence matrix types that are more flexible than those introduced in Chapters 6 and 7 are required. An overview of different storage schemes that are suited for these types of matrices is given in [22] and [14]. Each format provides features making it more or less favorable for certain tasks:

- The *coordinate storage* allows the initialization of a sparse matrix in arbitrary order. This feature makes the format favorable for the assembling phase of the FEM. However, the format is suboptimal for the computation of the matrix-vector product.
- The *compressed row storage*[2] requires that initialization can be performed within a certain order. The storage is compact and linear algebra operations like the matrix-vector product can be realized very efficiently. Thus, iterative solvers like the cg-methods can be utilized immediately. In addition, this format is appropriate for direct sparse solvers. Obviously, compressed row storage is a favorable format for the solving phase.

The sparse matrix class introduced in this section uses the coordinate storage format during the initialization, and afterwards converts the format to compressed row storage.

[2]This scheme is also known as *compressed sparse row* (CSR) format.

8.5.1 Storage Schemes for Sparse Matrices

Basically all storage schemes for sparse matrices follow a similar concept. Non-zero elements of the sparse matrices are stored linear in memory together with at most a few additional zeros. Additional arrays are then used to describe the position of these values in the original matrix. In order to exemplify the different schemes the following 5×6 matrix

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & & & & \\ & a_{2,2} & & & & a_{2,6} \\ a_{3,1} & & a_{3,3} & a_{3,4} & & \\ & & & a_{4,4} & & \\ & a_{5,2} & & & a_{5,5} & \end{pmatrix} \tag{8.25}$$

with 10 non-zero entries will be utilized.

Coordinate Storage (COOR)

In COOR format a sparse $M \times N$ matrix with n non-zero entries $a_{i_1,j_1}, \ldots, a_{i_n,j_n}$ is stored using three vectors `values`, `rows` and `columns`. All vectors are of length n. Indices $i_1, \ldots, i_n$ are stored in `rows`, indices $j_1, \ldots, j_n$ are stored in `columns` and values $a_{i_1,j_1}, \ldots, a_{i_n,j_n}$ in `values`.

In the COORD representation of matrix A from (8.25) vectors `values`, `rows` and `columns` all have length 10 and a possible occupation of the vectors is:

	1	2	3	4	5	6	7	8	9	10
`values`	$a_{1,1}$	$a_{1,2}$	$a_{2,2}$	$a_{2,6}$	$a_{3,1}$	$a_{3,3}$	$a_{3,4}$	$a_{4,4}$	$a_{5,2}$	$a_{5,5}$
`rows`	1	1	2	2	3	3	3	4	5	5
`columns`	1	2	2	6	1	3	4	4	2	5

Compressed Row Storage (CRS)

In CRS format a sparse $M \times N$ matrix with n non-zero entries $a_{i_1,j_1}, \ldots, a_{i_n,j_n}$ is stored using three vectors `values`, `columns` and `rows` as follows:

- `values` is of length n and stores the non-zero elements $(a_{i_1,j_1}, \ldots, a_{i_n,j_n})$ of A. Elements are stored row-wise, that means for $1 \le k < l \le n$ it follows that either $i_k < i_l$ or $i_k = i_l$ and $j_k < j_l$.
- For each element `values(k)` the corresponding element `columns(k)` gives its column index, i. e. `columns` stores indices $(j_1, \ldots, j_n)$.
- `rows` is of length $M + 1$ and keeps track of which element in `values` belongs to which row. `rows(i)` refers to the first element in `values` belonging to the i-th row of A. This means `rows(i)` $= \min\{k \mid i_k = i\}$ for $i = 1, \ldots, M$.
  The number of non-zero elements of the i-th row is given as `rows(`i`+1) -rows(`i`)`. In order to keep this relationship valid for the last row, the value of `rows(`$M + 1$`)` has to be $n + 1$, i. e. the number of non-zeros plus one. This could also be interpreted as the beginning of an additional fictitious row.

  From the definition of `rows` it follows that empty rows are not eligible. In this case a fill-in zero is required.

In the CRS representation `values` and `columns` are of length 10, `rows` of length 6. Values are given as

	1	2	3	4	5	6	7	8	9	10
`values`	$a_{1,1}$	$a_{1,2}$	$a_{2,2}$	$a_{2,6}$	$a_{3,1}$	$a_{3,3}$	$a_{3,4}$	$a_{4,4}$	$a_{5,2}$	$a_{5,5}$
`columns`	1	2	2	6	1	3	4	4	2	5
`rows`	1	3	5	8	9	11				

Definition of a new `rows` vector enables the support of matrix views referencing consecutive rows. For instance

rows	5	8	9

together with `values` and `columns` from above represents a matrix view of A referencing rows 3 to 5. For the parallelization of the matrix vector product exactly these kind of views are needed.

A typical life cycle of a sparse matrix object was already observed in the above FEM example: first the matrix gets set up, second a system of linear equation has to be solved.

8.5.2 FLENS Implementation of Sparse Matrices

The matrix types for BLAS and LAPACK introduced in Chapter 4 were realized based on the storage schemes from Chapter 3. The concept of separating matrix types and storage schemes also gets applied in the realization of sparse matrices.

Implementation of the CRS Scheme

For symmetric, hermitian or triangular matrices it is sometimes favorable that only the upper or lower triangular part of the matrix gets stored. Whether the scheme stores a triangular part or not, can be specified through the following enumeration type:

```
enum CRS_Storage {
    CRS_General,
    CRS_UpperTriangular,
    CRS_LowerTriangular
};
```

For example, `CRS_General` specifies that the scheme represents elements from a general matrix, while `CRS_UpperTriangular` defines that only the upper triangular part is stored.

Listing 8.11 outlines the interface for the CRS class. Indices and elements are stored in dense vectors (lines 25-26). Beside the number of rows and columns the constructor (line 9) has a third and forth parameter to specify the approximate bandwidth and the index base respectively. The bandwidth parameter is only for optimization purposes, i. e. to minimize the number of required reallocations in the initialization phase (see the remarks about the `CRS_Initializer` implementation below). After the construction of a `CRS` objects the vectors `values`, `columns` and `rows` all have length zero. The required length of these vectors is unknown until the initialization is completed. The initialization is handled by an external class which is here of type `CRS_Initializer` (specified in line 6). The `CRS_Initializer` class will be introduced in more detail below. The initializer can be instantiated through the method `initializer` (line 15) and receives the `CRS` object it was assigned for initialization (line 17). The method `allocate` (line 21) can then be used by the initializer to reallocate vectors `values`, `columns` and `rows`.

```
 1 template <typename T, CRS_Storage Storage=CRS_General>
 2 class CRS
 3 {
 4     public:
 5         typedef T                               ElementType;
 6         typedef CRS_Initializer<T, Storage>     Initializer;
 7
 8         // -- constructors -----------------------------------------------------
 9         CRS(int numRows, int numCols, int approxBandwidth=1, int indexBase);
10
11         ~CRS();
12
13         // -- methods ----------------------------------------------------------
14         Initializer *
15         initializer()
16         {
17             return new Initializer(*this, _approxBandwidth);
18         }
```

```

        void
        allocate(int numNonZeros);

        // ... methods to retrieve number of rows, cols, iterators, ...

        DenseVector<Array<T> >    values;
        DenseVector<Array<int> >  columns, rows;

    private:
        int  _numRows, _numCols, _approxBandwidth, _indexBase;
};
```

Listing 8.8: Extract of the interface of the CRS storage scheme.

The `CRS_Initializer` class uses vector types provided from the *Standard Template Library* (STL) [71]. For this purpose the following data structures are used for storing indices and the value of a single matrix element $a_{i,j}$:

```
template <typename T>
struct CRS_Coordinate
{
    CRS_Coordinate(int _row, int _col, T _value);

    int row, col;
    T   value;
};
```

During the initializing phase the `CRS_Initializer` class uses the COOR format. Once the initialization is completed, the format gets converted into CRS. This requires sorting matrix elements and therefore again the STL library gets used. Utilizing the STL algorithms for sorting requires the definition of a less operator as illustrated in [75]. For two matrix elements a_{i_1,j_1} and a_{i_2,j_2} the call operator `()` in line 6 returns true, if either $i_1 < i_2$ or $i_1 = i_2$ and $j_1 < j_2$:

```
1 struct CRS_CoordinateCmp
2 {
3     // return true if a < b
4     template <typename T>
5     bool
6     operator()(const CRS_Coordinate<T> &a,
7                const CRS_Coordinate<T> &b) const;
8 };
```

The interface of the `CRS_Initializer` class is outlined in Listing 8.9. As was mentioned above, the constructor receives the `CRS` object (line 5) which has to be initialized and stores a reference of it (line 23). Matrix elements can be set using the `()` operator (line 13) and are stored in the STL vector `_coordinates` (lines 16-17) by appending new elements. Sorting elements is based on the less operator introduced above (as specified in line 19) and can be triggered through the `sort` method (line 10). Conversion into CRS format is handled in the destructor (line 7).

```
 1 template <typename T, CRS_Storage Storage>
 2 class CRS_Initializer
 3 {
 4     public:
 5         CRS_Initializer(CRS<T, Storage> &crs, int approxBandwidth);
 6
 7         ~CRS_Initializer();
 8
 9         void
10         sort();
11
12         T &
13         operator()(int row, int col);
14
15     private:
        typedef std::vector<CRS_Coordinate<T> > Coordinates;
        Coordinates _coordinates;

        CRS_CoordinateCmp _less;
        size_t            _lastSortedCoord;
        bool              _isSorted;

        CRS<T, Storage>  &_crs;
};
```

Listing 8.9: Initializer for the CRS format.

Sorting of the `_coordinates` vector is required at least once, namely when the destructor has to perform the conversion from COOR to CRS format. Several optimizations are implemented to speed up the initialization and conversion. The flag `_isSorted` indicates whether vector `_coordinates` is sorted or not. Integer `_lastSortedCoord` contains the position up to which the vector is already sorted. Whenever a new matrix element gets appended, it will be checked whether the sorting flag has to be changed. In case elements are appended in order, sorting gets completely avoided. In case the order gets corrupted, the vector might get sorted from time to time even before the destructor is triggered[3]. In this case the vector is already sorted up to position `_lastSortedCoord`. By first sorting the remaining elements the complete vector can be sorted by a subsequent merge sort.

Like in Chapter 4 the implementation of matrix types delegate most of the functionality to the implementation of the underlying storage scheme. Listing 8.10 illustrates the implementation for symmetric sparse matrices. The type of storage scheme can be specified through a template parameter (line 1) and an instance of this scheme gets stored internally (line 42). The constructor (line 12) instantiates an initializer of type `Engine::Initializer` (as specified in line 9). Thus, using `CRS` as storage implementation, the initializer is of type `CRS_Initializer`. During the initialization phase, elements can be accessed through the call operator `()` which delegates the call to the initializer (line 19). A call of method `finalize` (line 26) ends the initialization phase. By destroying the initializer (line 28) its destructor gets called. In case of the above `CRS` implementation this results in conversion of the underlying scheme from COOR into CRS.

```
 1 template <typename Engine>
 2 class SparseSyMatrix
 3     : public SymmetricMatrix<SparseSyMatrix<Engine> >
 4 {
 5     public:
 6         // shortcut for element type
 7         typedef typename SparseSyMatrix<Engine>::ElementType  T;
 8 
 9         typedef typename Engine::Initializer     Initializer;
10 
11         // -- constructors ---------------------------------------------------
12         SparseSyMatrix(int dim, int approxBandwidth=1);
13 
14         // -- operators ------------------------------------------------------
15         T &
16         operator()(int row, int col)
17         {
18             assert(!_initializer)
19             return _initializer->operator()(row,col);
20         }
21 
22         // ...
23 
24         // -- methods --------------------------------------------------------
25         void
26         finalize()
```

[3]Typically this in-between sorting will be triggered whenever a reallocation of the `_coordinates` is required. Initially the capacity of `_coordinates` is set to `approxBandwidth`. If this turns out to be not sufficient, the capacity is increased by `approxBandwidth` and sorting triggered.

```
        {
            delete _initializer;
            _initializer = 0;
        }

        // ...

        // -- storage scheme -----------------------------------------------
        const Engine &
        engine() const;

        Engine &
        engine();

    private:
        Engine       _engine;
        Initializer *_initializer;
};
```

Listing 8.10: Symmetric sparse matrix class.

The following code snippet uses the tridiagonal matrix defined in (6.13) to exemplify the usage of the `SparseSyMatrix` class. The approximate bandwidth is in this case 3 (line 2). After initialization (lines 4-8) the matrix gets finalized (line 10) such that its internal storage format gets converted into CRS format. Subsequently, linear algebra operations can be performed (line 15) or other numerical methods applied like iterative or direct solvers.

```
1  int n = 8;
2  SparseSyMatrix<CRS<double> > A(n, 3);
3
4  for (int i=1; i<=n; ++i) {
5      if (i>1) A(i, i-1) = -1;
6      A(i,i) = 2;
7      if (i<n) A(i, i+1) = -1;
8  }
9
10 A.finalize();
11
12 DenseVector<Array<double> >  x(n), b(n), r(n);
13 // ... init b, x ...
14
15 r = b - A*x;
```

A simple implementation of the matrix-vector product is illustrated in the next subsection.

8.5.3 Iterative Solvers

In order to utilize the cg-method introduced in Chapter 5 an appropriate matrix-vector product has to be implemented. The implementation for sparse matrices with CRS format shown in Listing 8.11 closely follows the algorithm given in [4].

Function `crs_gemv` is capable of computing linear algebra operation of the form

$$y \leftarrow \alpha \mathrm{op}(A)x + \beta y$$

where $\mathrm{op}(A)$ can be either A or A^T and $\beta \in \{0, 1\}$. Most of the parameters of the function are specified similar to those of the BLAS function introduced in Chapter 3. The sparse matrix A gets passed via parameters `a` (pointer to the values), `ia` (pointer to the row indices) and `ja` (pointer to the column indices). In lines 9-13 pointers are shifted to enable the later use of the bracket operator.

```
1  void
2  crs_gemv(Transpose trans, int m, int n, double alpha,
3           const double *a, const int *ia, const int *ja,
4           const double *x, double beta, double *y,
         int indexBase)
{
    assert((beta==double(0)) || (beta==double(1)));

    // shift to index base
    a  = a  - indexBase;
    ia = ia - indexBase;
    ja = ja - indexBase;
    x  = x  - indexBase;
    y  = y  - indexBase;

    const bool init = (beta==double(0));

    if (trans==NoTrans) {
        for (int i=1; i<=m; ++i) {
            if (init) {
                y[i] = 0;
            }
            for (int k=ia[i]; k<ia[i+1]; ++k) {
                y[i] += alpha*a[k]*x[ja[k]];
            }
        }
    } else {
        if (init) {
            for (int i=1; i<=n; ++i) {
                y[i] = 0;
            }
        }
        for (int i=1; i<=m; ++i) {
            for (int k=ia[i]; k<ia[i+1]; ++k) {
                y[ja[k]] += alpha*a[k]*x[i];
            }
        }
    }
}
```

Listing 8.11: Implementation of the matrix-vector product for sparse matrices in CRS format.

The implementation of the matrix-vector product can be imbedded into FLENS as usual:

```
template <typename T, CRS_Storage Storage, typename VX, typename VY>
void
mv(T alpha, const SparseSyMatrix<CRS<T, Storage> > &A,
   const DenseVector<VX> &x,
   typename DenseVector<VY>::T beta, DenseVector<VY> &y)
{
    // ... check assertions ...

    if (Storage==CRS_General) {
        // call crs_gemv
    }
}
```

The cg-method is now ready to be utilized as solver in the `FEM` class:

```
void
FEM::solve()
{
    int M = mesh.numInteriorNodes();
    DenseVector<ArrayView<double> > u_ = u(_(1,M)); // interior nodes
    cg(A, u_, b);
}
```

Listing 8.12: Using the PARDISO solver in the FEM.

8.5.4 Direct Solvers

In the following an interface for the direct solver PARDISO (PArallel DIrect Solver) is shown. Alternatively, direct solvers from the packages SuperLU or UMFPACK can be supported similarly[4].

A detailed documentation of the PARDISO solver can be found in [65]. The implementation of a FLENS wrapper for the PARDISO solver merely requires a good understanding of its parameters:

```
template <typename E1, typename E2>
void
pardiso(DenseVector<E1> &x,
        SparseSymmetricMatrix<CRS<double> > &A, DenseVector<E2> &b)
{
    int pt[2*64*10];
    memset(pt, 0, 2*sizeof(int)*64*10);

    int maxfct = 1;
    int mnum = 1;
    int mtype = 2;
    int phase = 13;
    int n = A.numRows();
    double *a = A.engine().values.data();
    int *ia = A.engine().rows.data();
    int *ja = A.engine().columns.data();
    int perm[n];
    int nrhs = 1;
    int iparm[64];
    iparm[0] = 0;
    iparm[2] = 1;
    int msglvl = 1;
    double *y = b.data();
    double *sol = x.data();
    int error;

    pardiso_(pt, &maxfct, &mnum, &mtype, &phase, &n, a, ia, ja, perm, &nrhs,
             iparm, &msglvl, y, sol, &error);
}
```

The PARDISO solver can now be utilized in the FEM class as follows:

```
void
FEM::solve()
{
    int M = mesh.numInteriorNodes();
    DenseVector<ArrayView<double> > u_ = u(_(1,M)); // interior nodes
    pardiso(u_, A, b);
}
```

Listing 8.13: Using the PARDISO solver in the FEM.

8.6 Summary of the Chapter

This chapter demonstrated how FLENS can be utilized for the implementation of the Finite Element Method. The simplicity of the problem as well as the implementation makes the presented program examples attractive for teaching. Furthermore, the Finite Element Method was used to motivate the implementation of sparse matrix types. These types were realized following the same concepts used in Chapters 3 and 4. Linear algebra operations and interaction with external libraries could be supported easily and efficiently.

[4]All these packages are briefly introduced in Section 2.1.4

9 Summary and Conclusions

The development of scientific software requires the collaboration of specialists from different disciplines. For an effective collaboration it is required that these specialists are able to contribute to the software development while working solely on their own area of expertise. In an ideal collaboration a numerical analyst might contribute a first implementation of a new numerical method. However, this prototype is not necessarily optimized for peak performance. A specialist in high-performance computing then tunes this implementation to exploit modern hardware. Subsequently, researchers from disciplines like science, engineering, finance or medicine can utilize this numerical method in their software applications to simulate or optimize domain specific problems. Such an ideal form of scientific software development is only possible by using sophisticated tools that ease and simplify the collaboration.

In order to cope with these demands, this work describes our FLENS library. It shows how FLENS can be utilized as a flexible building block for the effective development of efficient scientific software. FLENS extends the C++ programming language for features that are relevant for scientific computing. Many techniques used in the design and implementation of FLENS can also be found in other libraries in this field. However, the way these techniques are combined and customized makes FLENS distinct.

One crucial feature of FLENS is the possibility of extending C++ for matrix and vector types. The *Curiously Recurring Template Pattern (CRTP)* is used and adapted to realize multilevel class hierarchies for matrix and vector types. One advantage of applying the CRTP is the avoidance of virtual functions which can lead to a considerable runtime overhead. Actually, many libraries apply the CRTP for this sole reason. In FLENS however, the CRTP also plays an essential role within its high-level abstraction for linear algebra operations. FLENS allows the usage of overloaded operators in order to provide an expressive notation. As this work shows, it is not trivial to achieve this without sacrificing efficiency. Performance issues in this respect are mainly due to the unnecessary and unintended creation of temporary objects. Two existing approaches are introduced for solving this issue. In the first approach, the concept of *closures* known from functional programming gets adapted to C++. While this approach allows the utilization of external high-performance libraries for the computation of linear algebra operations, it turns out that it is far from being a feasible method for practical applications. The second approach is based on the *expression template* technique. While this offers a much more feasible method for the realization of a linear algebra package, the resulting code is hard to maintain and to extend. Further, this approach is not capable of exploiting external high-performance libraries for the computation of linear algebra operations. In FLENS the ideas of these two approaches are combined. This results in a mechanism that makes it possible to exploit external libraries for high-performance computing without introducing any runtime overhead. At the same time this mechanism remains easy to extend, maintain and use. Essential for this accomplishment is the CRTP based design of the class hierarchies.

At first, this work illustrates the seamless integration of BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra PACKage) into FLENS. BLAS and LAPACK are well established standards in high-performance computing. Various BLAS and LAPACK implementations which are capable of exploiting modern hardware architectures are available. However, these libraries exhibit a rather low level of abstraction and their usage requires the knowledge of many technical details. This work shows how FLENS can be used to provide a high-level interface for these libraries. Overloaded operators provide a natural and expressive notation for linear algebra operations. Due to the techniques embedded in the architecture of FLENS there is no runtime overhead involved. Developers using FLENS receive the full power and functionality of the underlying BLAS and LAPCK implementation in a very convenient way. However, the ease of use and expressiveness provided by FLENS is more than just 'syntactic sugar', instead it is the key to enable numerical analysis to realize modern numerical methods in a form which is close to the mathematical notation. At the same time specialists in high-performance computing can tune the underlying implementation for performance without modifying these numerical methods.

In order to endorse and enhance the implementation of new state of the art algorithms, the sole integration of BLAS and LAPACK is not sufficient. As illustrated in this work, the numerical discretization of partial differential equations often leads to matrices that exhibit very special structures. Standard libraries like BLAS and LAPACK are not always capable of exploiting these structures. The crucial feature in this respect is the flexible extensibility of FLENS. Several examples are given to motivate and illustrate how FLENS can be extended for new matrix or vector types. Extending the set of matrix and vector types also requires the extension of the set of supported linear algebra operations. As this work shows, FLENS provides a simple and transparent mechanism to achieve both.

Several example are given in order to illustrate how an implementation of iterative methods can be reused for new user-defined matrix and vector types. Hereby it is not required to modify the existing implementation. As mentioned above, this gets first exploited to take advantage of some special matrix structures that occur in numerical applications. Furthermore, this concept is also applied to attain the parallelization of numerical methods on computer clusters. This was in particular illustrated for the multigrid method by utilizing the Dirichlet-Poisson problem as a toy example. The same implementation can be reused to solve the problem in different space dimensions and even parallelization does not require any modifications. Subsequently, the multigrid method is used as a component in a simple solver for the Navier-Stokes equations and again, parallelization of this solver immediately follows.

Despite the particular strengths of FLENS there are a few limitations. For the implementation of FLENS, various template techniques were extensively applied. Unfortunately, template instantiation can have a major impact on compilation time. Hence it can take a long time to compile large applications that use FLENS as a building block. In order to cope with this issue, techniques like *explicit instantiation* or *precompiled headers* [78] could be utilized to speed up the build time. A further issue resulting from using C++ templates is the error messages that C++ compilers typically produce. Even a slight error can result in many lines of cryptic error messages. How to decrypt and manage such error messages is covered in [78]. In the upcoming C++ standard the *concepts extension* [68] will further enhance the support for generic programming. Using this extension the issue can be resolved completely. Another issue is related to the design goals of FLENS. Consider the expressive notation that FLENS provides for linear algebra operations. As shown in Section 4.3.3 several efforts were realized in FLENS to prevent obscure side-effects like the unintended creation of temporary objects. For programmers who are inexperienced in scientific computing this feature of FLENS might seam to be inconvenient at first. However, it is a crucial requirement for the serious development of scientific software.

Currently the FLENS interface for BLAS and LAPACK supports only a subset of the functionality provided by these libraries. Since its public release in July 2007 the support has been increased. Also the number of numerical methods and matrix/vector types in FLENS could be extended. This progress is mainly due to various contributions from the steadily growing community of FLENS developers. In order to further encourage this process, the FLENS community needs to be supported through tutorials, documentation as well as example implementations for various numerical applications. We hope that FLENS becomes the building block of choice for the development of scientific software.

A FLENS Storage Schemes for BLAS

A.1 Full Storage Interface

The interface for classes `FullStorage`, `FullStorageView` and `ConstFullStorageView` can be grouped into the categories initialization, access to internal data structures, creation of views, constructors/assignment operators and public typedefs:

1. Initialization, element access and changing size/index base:

`int`
`firstRow() const;`
	First valid row index.

`int`
`firstCol() const;`
	First valid column index.

`int`
`lastRow() const;`
	Last valid row index.

`int`
`lastCol() const;`
	Last valid column index.

`const T &`
`operator()(int row, int col) const;`
	Element access.
	Debug mode: checks whether indices are valid.

`T &`
`operator()(int row, int col);`	

`void`
`resize(int numRows, int numCols,`
`       int firstRow=1, int firstCol=1);`
	Changes size or index base. Previously stored data will not be copied!

2. Access to internal data structures: in particular these methods are suited for calling BLAS and LAPACK functions (see Section 3.3 and 3.4).

`int`
`numRows() const;`
	Number of rows.

`int`
`numCols() const;`
	Number of columns.

`int`
`leadingDimension() const;`
	Leading dimension.

`int`
`strideRow() const;`
	Stride in memory between row elements.

`int`
`strideCol() const;`
	Stride in memory between column elements.

`const T *`
`data() const;`
	Pointer to the first element.

`T *`
`data();`	

3. Creation of views: FLENS provides several methods for the creation of views referencing parts of a full storage scheme. These methods reflects what kind of views are frequently needed in numerical applications and can roughly be grouped in three subcategories described in the following.

   (a) Creation of views referencing sub-matrix blocks:

`ConstFullStorageView<T, Order>`
`view(int fromRow, int fromCol,`
`     int toRow, int toCol,`
`     int firstViewRow=1,`
`     int firstViewCol=1) const;`
	Creates a view object referencing a sub-matrix block. The referenced block has row indices `fromRow,..,toRow` and column indices `fromCol,..,toCol` in the original matrix.
	Index bases for the view can be specified by `firstViewRow` and `firstViewCol`.

`FullStorageView<T, Order>`
`view(int fromRow, int fromCol,`
`     int toRow, int toCol,`
`     int firstViewRow=1,`
`     int firstViewCol=1);`	

   (b) Views referencing complete rows or columns (hence these results in array views):

`ConstArrayView<T>`
`viewRow(int row, int firstViewIndex=1) const;`
	Creates an array view of a complete row.

`ArrayView<T>`
`viewRow(int row, int firstViewIndex=1);`	

`ConstArrayView<T>`
`viewCol(int col, int firstViewIndex=1) const;`
	Creates an array view of a complete column.

`ArrayView<T>`
`viewCol(int col, int firstViewIndex=1);`	

   (c) Views referencing parts of rows or columns:

`ConstArrayView<T>`
`viewRow(int row, int fromCol, int toCol,`
`        int firstViewIndex=1) const;`
	Creates an array view of a partial row.

`ArrayView<T>`
`viewRow(int row, int fromCol, int toCol,`
`        int firstViewIndex=1);`	

`ConstArrayView<T>`
`viewCol(int col, int fromRow, int toRow,`
`        int firstViewIndex=1) const;`
	Creates an array view of a partial column.

`ArrayView<T>`
`viewCol(int col, int fromRow, int toRow,`
`        int firstViewIndex=1);`	

4. Constructors and assignment operators are implemented to provide functionality analogous to that provided for arrays (see Section 3.2.3).

`FullStorage(int numRows, int numCols, int firstRow=1, int firstCol=1);`	Creates a full storage storage scheme.

5. Public typedefs:

`ElementType`	Element type.
`NoView`	Type of a full storage scheme, which is no view.
`ConstView`	Types returned by the `view` methods.
`View`	
`ConstArrayView`	Types returned by `viewRow` and `viewCol` methods.
`ArrayView`	

The storage order of a full storage scheme can be retrieved using the `StorageInfo` trait:

```
template <typename FS>
void
dummy(const FS &fs)
{
    StorageOrder order = StorageInfo<FS>::order;
    // ...
}
```

A.2 Band Storage Interface

The interface for classes `BandStorage`, `BandStorageView` and `ConstBandStorageView` can be grouped into the categories initialization, access to internal data structures, creation of views, constructors/assignment operators and public typedefs:

1. Initialization, element access and changing size/index base:

`int firstIndex() const;`	First valid index for rows and columns
`int lastIndex() const;`	Last valid index for rows and columns
`int numSubDiags() const;`	Number of sub-diagonals
`int numSuperDiags() const;`	Number of super-diagonals
`const T & operator()(int row, int col) const;`	Element access. *Debug mode*: checks whether indices are valid.
`T & operator()(int row, int col);`	
`void resize(int numRows, int numCols, int numSubDiags, int numSuperDiags, int indexBase=1);`	Change size or index base. Previously stored data will not be copied.

2. Access to internal data structures: together with `numSubDiags` and `numSuperDiags` (from above) these methods are typically needed for calling BLAS and LAPACK functions dealing with band storage schemes (see Section 3.3 and 3.4).

`int`
`numRows() const;`
	Number of rows.

`int`
`numCols() const;`
	Number of columns.

`int`
`leadingDimension() const;`
	Leading dimension.

`const T *`
`data() const;`
	Pointer to first element.

`T *`
`data();`	

3. Views referencing parts of rows or columns:

`ConstArrayView<T>`
`viewDiag(int diag, int viewIndexBase=1) const;`
	Creates an array view referencing the *diag*-th diagonal.

`ArrayView<T>`
`viewDiag(int diag, int viewIndexBase=1);`	

`ConstBandStorageView<T, Order>`
`view(int fromDiag, int toDiag,`
`     int viewIndexBase=1) const;`
	Creates a band storage view referencing diagonals `fromDiag,...,toDiag`.

`BandStorageView<T, Order>`
`view(int fromDiag, int toDiag,`
`     int viewIndexBase=1);`	

4. Constructors and assignment operators are implemented to provide functionality analogous to that provided for arrays (see Section 3.2.3).

`BandStorage(int numRows, int numCols,`
`            int numSubDiags,`
`            int numSuperDiags,`
`            int indexBase=1);`
	Creates a storage scheme for a banded matrix.

5. Public typedefs and traits:

`ElementType`	Element type.
`NoView`	Type of a band storage scheme, which is no view.
`ConstView`	Types returned by the *view methods*.
`View`	
`ConstArrayView`	Types returned by `viewDiag` and `viewDiag` methods.
`ArrayView`	

The storage order can be retrieved using the `StorageInfo` trait:

```
template <typename BS>
void
dummy(const BS &bs) {
    StorageOrder order = StorageInfo<BS>::order;
    // ...
}
```

A.3 Packed Storage

The interface for class `PackedStorage` can be grouped into the categories initialization, access to internal data structures, constructors/assignment operators and public typedefs. Views are not supported:

1. Initialization, element access and changing size/index base:

`int`
`firstIndex() const;`
	First valid index for rows and columns

`int`
`lastIndex() const;`
	Last valid index for rows and columns

`const T &`
`operator()(int row, int col) const;`
	Element access.
	Debug mode: checks whether indices are valid.

`T &`
`operator()(int row, int col);`	

`void`
`resize(int dim, int indexBase);`
	Change size or index base.

2. Access to internal data structures: these methods are suited for calling BLAS and LAPACK functions dealing with packed storage schemes (see Section 3.3).

`int`
`dim() const;`
	Number of rows/columns.

`const T *`
`data() const;`
	Pointer to first element.

`T *`
`data();`	

3. Constructors and assignment operators are implemented to provide functionality analogously to that provided for arrays (see Section 3.2.3).

`PackedStorage(int dim, int indexBase=1);`	Creates a packed storage scheme.

4. Public typedefs and traits:

`ElementType`	Element type.

The storage order and wether the upper or lower part gets stored in the packed storage scheme can be retrieved using the `StorageInfo` trait:

```
template <typename PS>
void
dummy(const PS &ps)
{
    StorageOrder order = StorageInfo<PS>::order;
    StorageUpLo  upLo  = StorageInfo<PS>::upLo;
    // ...
}
```

Bibliography

[1] J. Alberty, C. Carstensen, and S. A. Funken. Remarks around 50 lines of Matlab: short finite element implementation. *Numerical Algorithms*, 20(2-3):117–137, 1999.

[2] G. Alefeld, I. Lenhardt, and H.Obermaier. *Parallele numerische Verfahren.* Springer, 2002.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK Users' Guide. *Society for Industrial and Applied Mathematics, Philadelphia*, 1999.

[4] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, Philadelphia, PA, second edition, 1994.

[5] P. Bastian, K. Birken, S. Lang, K. Johannsen, N. Neuß, H. Rentz-Reichert, and C. Wieners. UG: A flexible software toolbox for solving partial differential equations. *Computing and Visualization in Science*, 1:27–40, 1997.

[6] P. Bastian, M. Blatt, C. Engwer, A. Dedner, R. Klöfkorn, S. P. Kuttanikkad, M. Ohlberger, and O. Sander. The Distributed and Unified Numerics Environment (DUNE). In *In Proceedings of the 19th Symposium on Simulation Technique in Hannover*, September 2006.

[7] J. Bilmes, K. Asanović, C. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proceedings of International Conference on Supercomputing*, July 1997.

[8] J. Bilmes, K. Asanović, C. Chin, and J. Demmel. Using PHiPAC to speed error backpropagation learning. In *Proceedings of ICASSP*, April 1997.

[9] J. Bilmes, K. Asanovic, C. Chin, and J. Demmel. The PHiPAC v1.0 matrix-multiply distribution. Technical report, Computer Science Division (EECS), University of California, Berkeley, 1998.

[10] A. F. Borchert. *Zur Erweiterung von Programmiersprachen durch Bibliotheken: Konzept und Realisierung der Ulmer Oberon-Bibliothek.* PhD thesis, Ulm University, 1994.

[11] D. Braess. *Finite Elemente.* Springer-Verlag, second edition, 1997.

[12] W. L. Briggs, V. E. Henson, and S. F. McCommick. *A Multigrid Tutorial.* SIAM, second edition, 2000.

[13] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance. *Computer Physics Communications*, Volume 97:1–15, 1996.

[14] Intel Corporation. Intel(R) Math Kernel Library Reference Manual. http://www.intel.com/software/products/mkl/docs/mklman.htm.

[15] T. A. Davis. Algorithm 832: UMFPACK — An Unsymmetric-Pattern Multifrontal Method. *ACM Transactions on Mathematical Software*, 30(2):196–199, 2004.

[16] T. A. Davis. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30(2):165–195, 2004.

[17] T. A. Davis and I. S. Duff. An Unsymmetric-Pattern Multifrontal Method for Sparse LU Factorization. *SIAM Journal on Matrix Analysis and Applications*, 18(1):140–158, 1997.

[18] T. A. Davis and I. S. Duff. A Combined Unifrontal/Multifrontal Method for Unsymmetric Sparse Matrices. *ACM Transactions on Mathematical Software*, 25(1):1–20, 1999.

[19] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A Supernodal Approach to Sparse Partial Pivoting. *SIAM Journal Matrix Analysis and Applications*, 20(3):720–755, 1999.

[20] J. W. Demmel, J. R. Gilbert, and X. S. Li. An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination. *SIAM Journal on Matrix Analysis and Applications*, 20(4):915–952, 1999.

[21] P. Deuflhard and F. Bornemann. *Numerische Mathematik I.* Walter de Gruyter & Co., 2002.

[22] J. J. Dongarra. *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide.* SIAM, Philadelphia, 2000.

[23] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–28, 1990.

[24] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988.

[25] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. LINPACK User's Guide. *Society for Industrial and Applied Mathematics, Philadelphia*, 1979.

[26] K. Driesen and U. Hölzle. The Direct Cost of Virtual Function Calls in C++. *ACM SIGPLAN Notices*, 31(10):306–323, 1996.

[27] J. W. Eaton. *GNU Octave Manual.* Network Theory Limited, 2002.

[28] C. A. J. Fletcher. *Computational Galerkin Methods.* Springer, 1984.

[29] C. A. J. Fletcher. *Computational Techniques for Fluid Dynamics*, volume I. Springer, 1991.

[30] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[31] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Mass., 1995.

[32] G. H. Golub and C. F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, MD, USA, third edition, 1996.

[33] N. Gould, Y. Hu, and J. A. Scott. A Numerical Evaluation of Sparse Direct Solvers for the Solution of Large Sparse Symmetric Linear Systems of Equations. Technical Report RAL-TR-2005-005, Council for the Central Laboratory of the Research Councils, 2005.

[34] T. Granlund. The GNU MP Bignum Library. http://gmplib.org/.

[35] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, Cambridge, MA, USA, second edition, 1999.

[36] W. Hackbusch. *Multigrid Methods and Applications.* Springer, Berlin, 1995.

[37] S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley. ScaLAPACK: A Linear Algebra Library for Message-passing Computers. In *Proceedings of the 1997 SIAM Conference on Parallel Processing*, May 1997.

[38] M. Hanke-Bourgeois. *Grundlagen der Numerischen Mathematik und des Wissenschaftlichen Rechnens.* Teubner, 2002.

[39] H. Herwig. *Strömungsmechanik – Eine Einführung in die Physik und die mathematische Modellierung von Strömungen.* Springer, Berlin, 2002.

[40] C. W. Hirt, B. D. Nichols, and N. C. Romero. SOLA: A numerical solution algorithm for transient fluid flows. Technical report, Los Alamos Report LA-5852, jan 1975.

[41] IEEE. *IEEE Standard Glossary of Software Engineering Terminology.* IEEE Standard 729, 1983.

[42] R. Jones and R. Lins. *Garbage collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, 1996.

[43] P. Knabner and L. Angermann. *Numerical Methods for Elliptic and Parabolic Partial Differential Equations*, volume 44. Springer, New York, 2003.

[44] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. Basic Linear Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(1):308–323, 1979.

[45] M. Lehn. FLENS: Flexible Library for Efficient Numerical Solutions. http://flens.sf.net.

[46] X. S. Li and J. W. Demmel. SuperLU DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Transactions on Mathematical Software*, 29(2):110–140, June 2003.

[47] T. Neunhoeffer M. Griebel, T. Dornseifer. *Numerische Simulation in der Strömungslehre. Ein praxisorientierte Einführung.* Vieweg Verlagsgesellschaft, 11 1995.

[48] Maplesoft. Maple. http://www.maplesoft.com/.

[49] The MathWorks. MATLAB. http://www.mathworks.com/products/matlab/.

[50] J. Mayer. *On Quality Improvement of Scientific Software: Theory, Methods, and Application in the GeoStoch Development.* PhD thesis, Ulm University, 2003.

[51] S. McCormick. *Multilevel Adaptive Methods for Partial Differential Equations.* SIAM, 1989.

[52] A. Meister. *Numerik linearer Gleichungssysteme.* Vieweg, 2005.

[53] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall International, second edition, 1997.

[54] S. Meyers. *Effective C++.* Addison-Wesley Professional Computing Series. Addison-Wesley Professional, third edition, 2005.

[55] C. Moler. The Origins of MATLAB. http://www.mathworks.com/company/newsletters/.

[56] N. Myers. Traits: a new and useful template technique. *C++ Report*, 1995.

[57] Netlib. CLAPACK. http://www.netlib.no/netlib/clapack/.

[58] Netlib. Netlib BLAS directory. http://www.netlib.org/blas/index.html.

[59] Netlib. Scalable LAPACK. http://www.netlib.org/scalapack/.

[60] Numerical Algorithms Group. Basic Linear Algebra Subprogramms – A Quick Reference Guide. http://www.netlib.org/blas/.

[61] O. Pauly. Numerische Simulation amerikanischer Optionen. Diploma thesis, Ulm University, 2004.

[62] A. Quarteroni, R. Sacco, and F. Saleri. *Numerical Mathematics*, volume 37 of *Texts in applied mathematics*. Springer, Berlin, 2nd ed edition, 2007.

[63] O. Schenk and K. Gartner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Generation Computer Systems*, 20(3):475–487, 2004.

[64] O. Schenk and K. Gartner. On fast factorization pivoting methods for symmetric indefinite systems. *Electronic Transactions on Numerical Analysis*, 23:158–179, 2006.

[65] O. Schlenk. PARDISO Solver Project. http://www.pardiso-project.org/.

[66] A. Schmidt and K. G. Siebert. *Design of Adaptive Finite Element Software: The Finite Element Toolbox ALBERTA*, volume 42 of *Lecture Notes in Computational Science and Engineering*. Springer, Berlin Heidelberg, 2005.

[67] H. Schwandt. *Parallele Numerik*. Teubner, 2003.

[68] J. Siek, D. Gregor, R. Garcia, J. Willcock, J. Järvi, and A. Lumsdaine. Concepts for C++0x. http://www.open-std.org/.

[69] S. Singer. Numerische Optimierung der Hydromechanik des Voith-Schneider-Propellers. Diploma thesis, Ulm University, 2003.

[70] B. T. Smith et al. *Matrix Eigensystem Routines – EISPACK Guide*, volume 6. Springer, Berlin, 1976.

[71] A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, HP Technical Report HPL-94-34, February 1995.

[72] A. Stippler. LAWA: Library for Adaptive Wavelet Applications. http://lawa.sf.net/.

[73] A. Stippler. Design and Implementation of a Flexible Object-Oriented Library for Adaptive Numerical Methods. Diploma thesis, Ulm University, 2003.

[74] J. Stoer and R. Burlisch. *Numerische Mathematik 2*. Springer, 2005.

[75] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional Wesley, 1997.

[76] A. Tveito and R. Winther. *Introduction to Partial Differential Equations : A Computational Approach*, volume 29. Springer, Berlin, corr. 2nd print edition, 2005.

[77] K. Urban. *Wavelet Methods for Elliptic Partial Differential Equations*. Oxford University Press, To appear in 2008.

[78] D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley Professional, 2003.

[79] T. L. Veldhuizen. Blitz: Object Oriented Scientific Computing. http://www.oonumerics.org/blitz/.

[80] T. L. Veldhuizen. oonumerics.org: Scientific Computing in Object-Oriented LanguagesScientific Computing in Object-Oriented Languages. http://www.oonumerics.org.

[81] T. L. Veldhuizen. Expression Templates. *C++ Report*, 7(5):26–31, June 1995.

[82] T. L. Veldhuizen. Using C++ Template Metaprograms. *C++ Report*, 7(4):36–43, may 1995.

[83] T. L. Veldhuizen. Scientific Computing: C++ versus Fortran. *Dr. Dobb's Journal of Software Tools*, 22(11):34, 36–38, 91, 1997.

[84] T. L. Veldhuizen. Techniques for Scientific C++. Technical report, Indiana University Computer Science Technical Report #542, 2000.

[85] T. L. Veldhuizen and D. Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *roceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*. SIAM, 1998.

[86] T. L. Veldhuizen and M. E. Jernigan. Will C++ be faster than Fortran? In *Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97)*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[87] J. Walter and M. Koch. uBLAS. http://www.boost.org/libs/numeric/.

[88] R. C. Whaley. Automatically Tuned Linear Algebra Software (ATLAS). http://math-atlas.sf.net/.

[89] R. C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998.

[90] S. Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley, second edition, 1991.

[91] K. Zaglauer. Risk-Neutral Valuation of Participating Life Insurance Contracts in a Stochastic Interest Rate Environment. Diploma thesis, Ulm University, 2006.