CUDA-based Scientific Computing
Tools
and
Selected Applications

Dissertation
zur Erlangung des mathematisch-naturwissenschaftlichen
Doktogrades
"Doctor rerum naturalium"
der Georg-August-Universität Göttingen

im Promotionsprogramm *Grundprogramm Mathematik*
der Georg-August-University School of Science (GAUSS)

vorgelegt von
Stephan C. Kramer

aus Mülheim a. d. Ruhr
Göttingen, 2012

Betreuungsausschuss

Herr Prof. Dr. G. Lube,	Institut für Numerische und Angewandte Mathematik
Herr Prof. Dr. R. Kree,	Institut für Theoretische Physik

Mitglieder der Prüfungskommission

Referent :	Herr Prof. Dr. G. Lube,	Institut für Numerische und Angewandte Mathematik
Korreferent :	Herr Prof. Dr. R. Kree,	Institut für Theoretische Physik

Weitere Mitglieder der Prüfungskommission

Frau Prof. Dr. D. Bahns,	Mathematisches Institut
Herr Prof. Dr. W. Kurth,	Abteilung Ökoinformatik, Biometrie und Waldwachstum
Herr Prof. Dr. L. Luke,	Institut für Numerische und Angewandte Mathematik
Herr Prof. Dr. A. Tilgner,	Institut für Geophysik

Tag der mündlichen Prüfung:

22.11.2012

... the North Pole itself - had attracted me from childhood, and here I was at the South Pole.

(Roald Amundsen in his diary)

Contents

Table of Abbreviations

2DEG	two-dimensional electron gas
BLAS	basic linear algebra subroutines
SciPAL	scientific and parallel algorithms library
OpenGL	open graphics language
ABF	Arnoldi-Bratwurst-Faber
ADR	advection-diffusion-reaction
AlGaAs	Aluminum-Gallium Arsenide
API	application programmer's interface
BDF	backward differentiation formula
BEM	boundary element method
BIE	boundary integral equation
CCO	composition closure objects
CPU	central processing unit
CSR	compressed storage row
CUDA	compute unified device architecture
DC	direct current
DoF	degree of freedom
DR	dielectric relaxation
DRS	dielectric relaxation spectroscopy
DtN map	Dirichlet-to-Neumann map
EMF	exterior mapping function
FEM	finite element method

FFT	fast Fourier Transform
FFTW	fastest Fourier Transform in the West
GaAs	Gallium Arsenide
GLSL	GL shading language
GMRES	Generalized Minimal Residual
GPGPU	general purpose GPU computing
GPU	graphics processing unit
HSE	Hardy space infinite elements
ILU factorization	incomplete LU factorization
M	mole/ℓ
MAP	Method of Alternating Projections
MinRes	Minimal Residual
MOSFET	metal-oxide-semiconductor field-effect transistor
MPI	message passing interface
MV product	matrix-vector product
NIPALS	Nonlinear Iterative Partial Least Squares
NMR	nuclear magnetic resonance
NUMA	non-uniform memory access
nvcc	NVidia *C* compiler
PAAL	parallel architecture abstraction layer
PCA	principal component analysis
PDE	partial differential equation
PNS	Parallel Navier-Stokes Solver
PPc	polynomial preconditioning
PSPG	Pressure-Stabilized Petrov/Galerkin
QPC	quantum point contact
RAAR	Relaxed averaged alternating reflections
RANS equations	Reynolds-averaged Navier-Stokes equations

SHM	shared memory
SIMD	single instruction multiple data
SLM	spatial light modulator
SMP	shared multiprocessors
SpAI	sparse approximate inverse
SpMV product	Sparse Matrix-Vector product
SUPG	Streamline-Upwind/Petrov-Galerkin
SVD	singular value decomposition
TBC	transparent boundary conditions

A Note on Notation

Small bold face letters will be used to denote vectors being elements of some finite-dimensional vector space. Especially, $\mathbf{x}$ will usually denote the solution of some set of linear algebraic equations and $\mathbf{b}$ its right-hand side. Matrices will be denoted by capital bold face letters and $\mathbf{A}$ mostly represents the coefficient matrix of a set of linear algebraic equations.

In quantum-mechanical problems the solution of the Schrödinger equation, i.e. the wave function, will be denoted by Ψ. In finite element problems the solution of the fully discretized problem gets an index h to denote the mesh width.

Overview

The first goal of this thesis is the design of *SciPAL* (Scientific and Parallel Algorithms Library), a *C*++-based and operating-system-independent library. The core of *SciPAL* is a domain-specific embedded language [9] for dense and sparse linear algebra which is presented in chapter 1. Chapter 2 shows that by using *SciPAL*, algorithms can be stated in a mathematically intuitive, unified way in terms of matrix and vector operations. The resulting rapid prototyping capabilities are discussed by using LU factorization and principal component analysis as example. *SciPAL* ports the most frequently used linear algebra classes of the widely used finite element library *deal*.II to CUDA (NVidia's extension of the programming language *C* for programming their GPUs). Thereby, simulation frameworks based on *deal*.II can easily be adapted to GPU-based computing. Besides adding a user-friendly API to any BLAS *SciPAL* particularly aims at simplifying the usage of NVidia's CUBLAS, especially the issues of data transfer arising from CUDA's distributed memory programming model. Chapter 3 closes the first part with a brief discussion of the necessary steps to solve sparse, unstructured linear systems.

The second part of this thesis is a collection of various examples which started as projects based on *deal*.II and at some point profit from being moved to the GPU. They are drawn from the field of neuroscience, enginering of indoor airflow, quantum transport in semiconductor heterostructures and structure-function interaction of proteins.

Chapter 4 contributes to the interactive stimulation of light-sensitive neurons by evaluating the efficiency of existing algorithms and their speedup by CUDA.

The accurate numerical prediction of indoor airflows for building configurations of practical relevance is of paramount importance for the energy-efficient design of modern buildings. Chapter 5 discusses CUDA-based preconditioning for indoor airflow and the technical implications for existing codes written for the numerical solution of the underlying Reynolds-averaged Navier-Stokes equations.

Chapter 6 focuses on the main issues of designing proper boundary conditions for finite element simulations of quantum transport in the Landauer-Büttiker picture. The simulation of a two-dimensional electron gas for a given set of model parameters is not a big deal. The difficult part is an accurate formulation of transparent boundary conditions needed to truncate the source and drain leads to a finite length.

The formulation of a GPU-based framework for quantum wave-packet dynamics in chapter 7 is straightforward if formulated in terms of matrix-vector products and is an example for how to combine the topics of this thesis to solve a new problem class.

The accurate simulation of single-molecule impedance spectroscopy of globular proteins must be done in 3D. Current theoretical models ignore boundary conditions and the electro-chemical properties of ions completely. Chapter 8 introduces and improved model based on the Poisson-Nernst-Plack equations. The physical properties are dominated by boundary layers and the discontinuity of the dielectric permittivity at the protein-solvent interface. The goal of the simulation is the determination of the electro-diffusive fluxes in the solvent. To avoid the inclusion of the protein interior in the simulation the Poisson problem in the protein interior is replaced by a boundary-value problem leading to a FEM-BEM coupling. Since it is basically an elliptic problem multigrid methods are particularly well-suited for its efficient solution.

Part I

Tools

Chapter 1

SciPAL: A library for GPU Computing

The first goal of this thesis is the design of SciPAL, the **Sci**entific and **P**arallel **A**lgorithms **L**ibrary. *Its design concepts and most important implementation details are discussed in this chapter.*

Numerical linear algebra is the backbone of most fields of scientific computing. The BLAS standard [45] defines a procedural programming interface for the most frequently recurring basic linear algebra subroutines. *The core of SciPAL is set of C++-classes for matrices and vectors supporting an operator-based programming interface with the purpose to encapsulate the error-prone usage of the bare BLAS programming interface. This is realized by a domain-specific embedded language [9] for dense and sparse linear algebra abstracting from the particular BLAS implementation.*

A current trend in high performance computing is to use the graphics processing unit (GPU) as general-purpose co-processor to a computer's central processing unit (CPU). This trend was triggered by the development of CUDA, NVidia's extension of the programming language C for programming their GPUs. CUDA considerably simplifies using a GPU for non-graphic purposes. The CUBLAS library, the CUDA-based BLAS implementation, allows users to almost immediately take advantage of the high computing power of a GPU in an existing application. By SciPAL's hardware abstraction capabilities, algorithms can be stated in a mathematically intuitive, unified way even for CUBLAS-based computations. A standalone matrix-vector library is not of great value. Hence SciPAL is designed to be compatible with the widely used finite element library deal.II right from the start.

1.1 Introduction

Computers evolve more and more into heterogeneous compute environments in which, especially non-graphics related, *data-parallel* work is delegated from the central processing unit (CPU) to the graphics processing unit (GPU). Exploiting the GPU for other than their traditional task of 3D graphics has quickly become a subject of its own and is by now known as *general purpose GPU* (GPGPU) *computing*. At the beginning GPGPU computing relied on reinterpreting OpenGL textures as vectors and matrices, respectively, and to use the means provided by GLSL (the GL shading language) to implement linear algebra operations [28]. One of the first dedicated high-level languages for GPGPU computing, *Brook for GPU* [31], was developed at the Stanford University Graphics Lab. This eventually evolved into AMD's *Brook+* which became part of AMD's *ATI Stream* computing environment and the nowadays popular CUDA (*compute unified device architecture*) by NVidia which is used in this work.

CUDA extends the *C* programming language by the means to execute *C* functions in parallel on the GPU and concurrent to the CPU. Anyone proficient in *C* knows CUDA as well, at least in principal, which keeps the learning barrier rather low. Programming with CUDA shares many similarities to MPI [106] in that the GPU consists of several multicore processing units with private, dedicated memory equivalent to a node of a cluster. Data exchange happens via the global GPU memory which is the analog to the network connections in a cluster. Yet, CUDA is conceived for a different miniaturization level and rather complements MPI by potentially speeding up what in the MPI context has been defined as job for an individual node of a cluster.

Because of its versatility in domain-specific types and focus on type correctness *C++* has emerged to be the object-oriented (OO) programming language of choice for scientific computing. The modularity of OO software leads to a much higher level of code reuse and thus a more rapid software development. To extensively reuse already well-tested pieces of code is one of the paradigms of OO programming. The disadvantage is that excessive use of OO features like virtual functions may lead to poor performance. An economic approach is to merge the best of both worlds: Use *C++*, especially its operator-overloading capabilities, for an intuitive, user-friendly application programmer's interface (API) close to the mathematical abstraction and offload the work to a BLAS function. User-friendly APIs based on *C++* and high-performance computing are not a contradiction. A good example for a modern software library of high performance following object-oriented design principles is the widely used finite element library *deal*.II [20, 21].

The first step towards an operator-based API is to hide the precision dependence of the names of BLAS functions in wrapper functions with unified names by *C++*'s polymorphism. The standard problem in *C++* with overloaded operators modeling linear algebra operations is their greedy evaluation. With no further action intermediate results are evaluated immediately which in most cases is redundant. The essential ingredient of an efficient operator-based interface are *expression templates* (ET), [130, 131, 132], [9, Chapter 10] and *composition closure objects* (CCO), as Stroustroup calls them [123, 122, Section 22.4 and earlier editions]. Their purpose exactly is to eliminate the redundant evaluation of temporary results in composite expressions by deferring the execution of individual operations until the full expression is known. The difficulty in this *lazy evaluation* is to map expressions to BLAS functions without creating large amounts of redundant source code. Using *C++*'s template capabilities *SciPAL* generates the BLAS expressions from a few building blocks and provides an integration of the large amount of existing, highly optimized BLAS libraries in a hardware-independent manner. Therefore, the goal of encapsulating the bare CUBLAS API to leverage its usage and which is as cumbersome as any other BLAS API immediately leads to a much more general solution.

The modular structure of *SciPAL* allows to compare the performance of different BLAS implementations and hence different hardware and parallelization concepts under otherwise identical conditions. A by-product of *C++*'s type correctness is that a lot of errors can be detected by the compiler already before run-time in contrast to *C* or FORTRAN. For small software projects this issue might be of minor importance but for larger simulations or analyses, possibly running for weeks or months, this is crucial.

1.1.1 BLAS

BLAS itself is a proposal for a standard API for basic linear algebra operations dating back to 1979. It groups operations according to their run-time complexity into three levels (vector-vector, matrix-vector and matrix-matrix operations). It requires a separate implementation for each precision, e.g. for matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ and scalars α, β the generalized matrix-matrix product

$$\mathbf{C} \quad = \quad \alpha \mathbf{A} \cdot \mathbf{B} + \beta \mathbf{C} \tag{1.1}$$

is implemented four-fold in the functions `sgemm`, `dgemm`, `cgemm`, `zgemm`, which encode the single- and double-precision real and complex version, respectively. Often the function names are prepended by identifiers for the specific BLAS implementation. In contrast to modern C++-based programming function names (and their argument lists) are neither really self-explanatory nor self-documenting as is illustrated by the generalized matrix-matrix (GEMM) multiplication in listing 1.1. Maybe, in this case the name is decipherable, at first sight the arguments certainly are not. Nor is this the case for functions like `cublasStrsm`.

Listing 1.1: Declaration of the CUBLAS variant of the single-precision real GEMM operation.

```
void
cublasSgemm (char transa, char transb, int m, int n,
             int k, float alpha, const float *A, int lda,
             const float *B, int ldb, float beta,
             float *C, int ldc)
```

BLAS contains the most ubiquitous time-critical routines encountered in industrial and scientific applications. Therefore, a lot of effort has been spent on producing optimal implementations exploiting the peak floating point performance and memory bandwidth for a broad range of problem sizes. Each of these implementations is tuned for a specific hardware, e.g. Intel's MKL, AMDs ACML with the ACML-GPU extension for GPU-computing on AMD Radeon graphics cards or CUBLAS as its reincarnation on NVidia GPUs. Despite the API the high level hardware-specific optimizations of the various BLAS libraries make them an invaluable tool for performance-critical programs and thus are worthwhile to add what is beyond the scope of BLAS: A user-friendly API.

1.1.2 Related Projects

The idea of adding a *C*++-interface to BLAS or LAPACK is not new, there are several projects (uBLAS [2], CPPLAPACK [3], LAPACK++ [4], TNT [7]) with varying states of completeness and activity. Particularly interesting is MTL4 [55] which rebuilds BLAS and LAPACK from scratch in *C*++. It uses all the advanced template metaprogramming strategies. MTL4 has a mathematically inspired API yet said to reach a similar performance as BLAS or LAPACK.

Mangor et al. have developed an expression-template-based library for vector algebra operations [141]. By design they are limited to CUDA for parallelization and BLAS-level-1 operations, i.e. *saxpy*-like operations. Another attempt to combine GPU computing with expression templates appeared in the context of rigid body dynamics [121] but had its focus on simplifying complex vector expressions inside of CUDA kernels where vectors itself were elements of a large array with a size proportional to the number of rigid bodies (i.e. hard spheres). Smart expression templates with a scope beyond BLAS-level-1 operations are provided by *Blaze* [65, 63, 64] which ignores GPGPU computing by purpose according to the authors.

1.2 API Design and Implementation

The goal of *SciPAL* is to formulate linear algebra operations in the shortest possible way. For instance, when multiplying a vector **x** by a matrix **A** with vector **b** as result we want to type:

```
    b = A * x;
```

This requires several steps. First of all, the notion of matrices and vectors has to be converted into proper *C*++ classes. Secondly, we have to implement an assignment = and a multiplication operator `*` with the appropriate associativity. The techniques to reach this are *lazy evaluation* of operator expressions, *composition closure objects* and *expression templates*.

The backbone of *SciPAL* is the class for arrays encapsulating the memory management in Sec. 1.4.2, the class for dense matrices given in Sec. 1.4.3 and a class for vectors, cf. Sec. 1.4.4. On top of these are convenience classes which provide views on subsets of matrices or vectors. These classes abstract from the type `T` of the matrix or vector elements and from the BLAS implementation given by the template parameter `BW`. Their purpose is to incorporate BLAS as high-performance implementation of the linear algebra operations performed on matrices and vectors and to form the bridge to the operator-based API for these operations.

The naming convention defined by BLAS causes a four-fold maintenance effort for a single routine. The first step in designing an abstraction layer for decoupling BLAS-based programs from the underlying particular BLAS-library is to make use of *C*++'s polymorphism, i.e. the name of a function is a combination of its bare name and the types of its arguments. The compiler distinguishes functions of the same name by differences in their types of arguments or argument lists. This still leaves the option to provide precision-dependent optimizations in the function's body. Therefore, to unify names of the BLAS functions we need a structure like the one in Sec. 1.4.1 providing wrapper functions with idential names for the different precisions.

Auxiliary functions for e.g. memory allocation are unified by templatizing their wrapper function with respect to the number type so that they have to be implemented only once. A nice feature of existing BLAS libraries is that they keep track of execution errors such that one can check for errors after each function call. To enable the error checking in an all-or-nothing fashion the query of the BLAS error state is incorporated into the wrapper functions so that run-time errors can be accurately tracked down by the debugger as discussed in Sec. 1.4.1. In order to avoid any performance penalty due to the wrapper functions they are declared as `inline`. Then, the compiler can replace them at their point of invocation by their body which effectively saves one function call.

Unlike in *C*++, the BLAS function names (and their argument lists) are neither really self-explanatory nor self-documenting as the `sgemm` example in listing 1.1 shows. One of the paradigms of linear algebra is to formulate as much of an algorithm in terms of matrix-matrix products as possible. Hence, a direct use of the BLAS API does not really lead to code which is easy to maintain.

In order to ensure compatibility to *deal*.II, matrix and vector classes will be template classes conforming to the interface implicitly defined by the way the Krylov solver classes of *deal*.II use their template arguments internally. For matrices this boils down to implement a member function `vmult` such that $\mathbf{b} = \mathbf{A} \cdot \mathbf{x}$ can be written as:

```
    A.vmult(b, x);
```

The function `vmult` is to be declared `inline` and hides the details of the corresponding BLAS functions, e.g. `sgemv` if `A` is a single precision, dense matrix of unknown symmetry. Compared to *C* or the call to the corresponding BLAS function with 10 arguments, this is already more expressive. Yet, for algorithms rich in matrix-vector products like *conjugate gradients* or *GMRES* [112] this still can degrade code readability. Similar examples of frequent occurrence are dot products and norms.

1.2.1 Lazy Evaluation

It is well-known that the standard way of overloading binary non-member operators in *C++* requires temporary objects for storing and returning the result of the operation. It is obvious that this may lead to remarkable performance losses and especially in the case of large objects may become a question of memory availability. The key to avoiding unnecessary temporary objects is lazy evaluation, i.e. to defer the evaluation of individual expressions until the whole expression has been completely parsed and processed. This is of particular importance in an expression formed by a chain of elementary linear algebra operations. The basic concept of lazy evaluation is not restricted to making the computation of an algebraic expression more efficient. It has a much broader scope and may also be successfully used in designing frameworks for building parsers like the *C++* template library *spirit* [5] which is part of the *boost* project [1]. In fact *spirit* is a domain specific embedded language for context-free languages, i.e. those generated by type-2 grammars of the Chomsky-hierarchy [6].

Composition Closure Objects

The basic ingredient of lazy evaluation of linear algebraic operations is to store the compound mathematical expression in a tree with operands as leafs and operators as nodes. From the tree structure the compiler can deduce the final expression and replace the chain of operations by a single call to a function which provides an optimal implementation for the compound expression. For instance, for the example in Eq. (1.1) the compiler should replace the operator-based expression by the equivalent call to `sgemm`, `dgemm`, `cgemm` or `zgemm` depending on the number type and selected precision.

Deferring the computation of the expression is achieved by overloading, e.g. the multiplication operator `*`, such that merely a small object is returned. The only purpose of this object is to hold references to the operands such that the operands can be retrieved at a later time. Such objects are called *composition closure objects* (CCO) by Stroustroup [123, 122, Section 22.4 and earlier editions]. For a matrix-vector product they look like:

```
struct MVmult
{
    const Matrix & A;
    const Vector & x;

    MVmult(const Matrix & l, const Vector & r)
    :
    A(l), x(r) {}
};
```

An alternative designation as *smart expression templates* has been coined recently by Iglberger et al. [64]. An overloaded operator for this operation is essentially given by:

```
inline MVmult
operator * (const Matrix & A,
            const Vector & x)
{
    return MVmult(A, x);
}
```

By declaring `operator *` as `inline` the compiler may optimize away the temporary object for the expression and replace it by the evaluation of the assignment operator. The final step is to equip the `Vector` class with a constructor which takes the expression as argument:

```
Vector (const MVmult & mv)
{
    this->reinit(mv.A.n_rows());
    mv.A.vmult(*this, mv.x);
}
```

For objects already instantiated we need an assignment operator performing equivalent operations. The actual implementation of composition closure objects is more involved as we rather code them as template classes with the types of the left- and right-hand side operands and of the mathematical operation as template arguments.

BLAS offers several routines providing operations optimized for matrices of special structure. This can be managed by a class `MatrixTraits` which contains flags for symmetry, upper or lower triangularity or bandedness as static constants so that they can be evaluated at compile-time. For instance, in the `vmult` member function the compiler then can select the proper branch from the different cases of a `switch` statement and remove all other. This is similar to the way how *deal*.II works out the dimension-handling. For the purpose of a simple presentation we do not include the traits in the code listings.

Transposition of Objects

To indicate transposition of a matrix or vector *SciPAL* provides a unary expression structure (templatized with respect to the object's type) which simply stores a reference to the object we want to use the transpose of. The transpose of a vector is interpreted as a row vector if the original one is a column vector and vice versa. Thus, `x * transpose<Vector>(x)` indicates the outer product of a vector `x` with itself which can be easily mapped to one of the BLAS `ger`-functions which exactly compute such a product. Given a matrix `A` the temporary object `transpose<Matrix>(A)` exists long enough so that the reference to the original matrix can be extracted. The lengthy typename could be abbreviated using a suitable `typedef`. If the transpose is frequently needed one can instantiate a permanent object:

```
    transpose<Matrix> A_T(A);

    b = A_T * x;
```

The most general expression offered by BLAS is the generalized matrix-matrix multiplication or the corresponding generalized matrix-vector multiplication. We exploit this considerable simplification and restrict the capabilities of our ET engine to the set of BLAS operations.

1.2.2 Porting to Other Platforms

With API and BLAS implementation separated from each other it becomes very easy to port an application to any hardware platform for which a BLAS library and a *C*++ compiler exist. If at all, only the BLAS wrapper class has to be reimplemented. Due to the widespread distribution of Linux this requirement is met on virtually any modern hardware, even on embedded devices like smartphones or tablet computers. For instance, Apple even requires that apps for their mobile devices are implemented in either *C*, *C*++ or *Objective*-*C*.

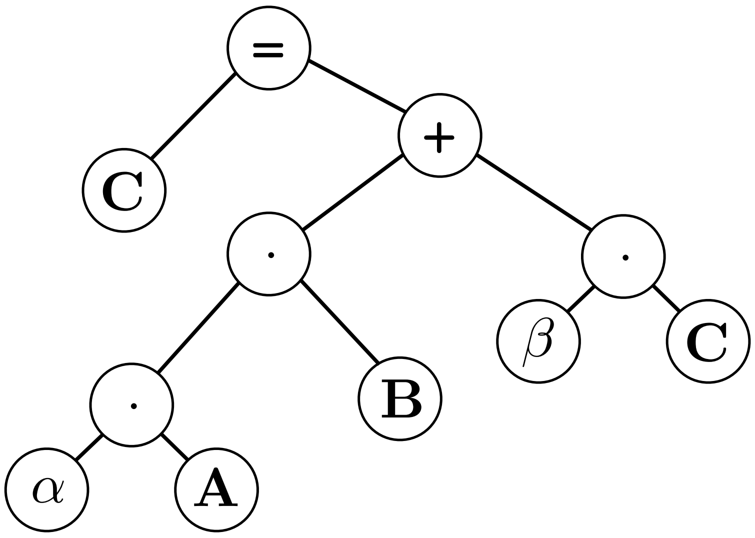


Figure 1.1: Expression tree for GEMM. The tree for GEMV would look the same except for **B** and **C** being vectors then. The left-to-right associativity of operator * assures that $\alpha\mathbf{A}$ is considered as the terminal subexpression and not $\mathbf{A} \cdot \mathbf{B}$.

1.3 Example: GEMM tree

To give an idea of what an expression tree might finally look like we sketch the case of the generalized matrix-matrix multiplication. For each final expression `Expr` the result of which we want to assign to a vector or matrix we need a constructor like:

```
Vector(const Expr & e) { e.apply(*this); }
```

To avoid writing *n* constructors for *n* different expressions we could extend the constructor to be a template function leaving the redundant code duplication to the compiler:

```
template<typename X>
Vector(const X & x) { x.apply(*this); }
```

Unfortunately, this is too generic as, for instance, this pattern would cover as well:

```
Vector v(5);
```

Inevitably this will fail. The compiler would interpret `5` as some sort of `int`, i.e. as a built-in primitive data type which does not offer an `apply()` member function. The solution to this problem is the introduction of a templatized expression base class [56]:

```
template<typename E> struct Expr : public E {

    const E & operator~() const
    {
        return static_cast<const E&>(*this);
    }
};
```

and the corresponding, less ambiguous, constructor:

```
template<typename X> Vector(const Expr<X> & x)
{
    ~x.apply(*this);
}
```

The ~ operator provides a type-safe downcast to the dynamic type of the expression and builds on the Barton-Nackman trick [9, Sec. 9.8].

Before we proceed we should think about what type of compound operations BLAS offers, what elementary expressions are needed for them and in which cases we explicitly have to take into account the notion, or better: concept, of a matrix or a vector. As long as no equation solving is involved we can ignore the division operator as it would only appear in scaling operations and thus can be mapped to multiplication with a reciprocal. Negative signs in front of matrices or vectors can be swallowed up by unary expressions. It remains the need for the concept of a product and a sum where sums can only be applied to operands of the same type. It does not make sense to add a row of a matrix to a column vector. Products are more flexible as, for instance, they allow to create matrices from the product of a column with a row vector. Distinguishing between matrix-vector and matrix-matrix products is also necessary. At least in the final expression at the point where the decision is made about which BLAS function has to be used. Further difficulties arise from the need to have the choice between matrices or vectors or subsets of them as it frequently is the case in all standard factorization methods (LU, QR, SVD). Therefore, the next building block is a template for binary expressions:

```
template<typename _L, Operator _o, typename _R>
struct BinaryX : public Expr<BinaryX<_L, _o, _R> > {

    typedef _L L;
    typedef _R R;

    const L& l;      const R& r;

    static const Operator op = _o;

    BinaryX(const L& l, const R & r) : l(l), r(r) {}
};

enum Operator {times, divide, plus, minus };
```

It encapsulates the storage of the references to the operands, the operator flag and provides abbreviations for the operands' types. Subclassing refines the concept to sums and products:

```
template<typename Number, typename Matrix>
struct Sum : public BinaryX<Number, plus, Matrix> {

    typedef BinaryX<Number, plus, Matrix> B;

    typedef typename  B::L L;
    typedef typename  B::R R;

    Sum(const L & l, const R & r) : Base(l, r) {}
};
```

```
template<typename Number, typename Matrix>
struct Prod : public BinaryX<Number, times, Matrix> {

    typedef BinaryX<Number, times, Matrix> B;

    typedef typename  B::L L;
    typedef typename  B::R R;

    Prod(const L & l, const R & r) : Base(l, r) {}
};
```

Scaled matrices and scaling of matrix-matrix products and the like is then obtained from a set of typedefs which will serve for processing the expression tree in the GEMM operation:

```
    typedef Prod<Number, Matrix> scaledM;

    typedef Prod<Matrix, Matrix> MM;

    typedef Prod<scaledM, Matrix> scaledMM;
```

The final GEMM expression could be built by subclassing a specialization of the `Sum` template:

```
typedef Sum<scaledMM, scaledM> GEMMBase;

struct GEMM : public GEMMBase
{
    typedef typename  GEMMBase::L L;
    typedef typename  GEMMBase::R R;

    GEMM(const L & l, const R & r) : GEMMBase(l, r) {}

    void apply(Matrix& result)
    {
        // unroll tree of scaledMM subexpression
        T alpha = l.l.l;
        const Matrix & A = l.l.r;
        const Matrix & B = l.r;

        // unroll tree of scaledM subexpression
        T beta = r.l;

        // Adapt name of destination to BLAS
        Matrix & C = result;

        // the details of the call to gemm are hidden
        // in a member function of the matrix type.
        // whether A or B is transposed is figured out there.
        C.scaled_mmult_add_scaled(alpha, A, B, beta);
    }
};
```

The code snippets in this section are intended to highlight the principal problems when designing a system of CCOs based on expression templates. They are not supposed to work right after copy-paste-compile. Especially the unconditional genericity of templates may sometimes provide more combinations of types than actually wanted.

1.4 API Classes

We close this chapter by a brief discussion of the main classes of the *SciPAL* library. To this end, we follow the fate of CUBLAS' `dot`-function. As a level-1-function `dot` does not have too many arguments and thus the number of uninteresting details is still at an acceptable level.

1.4.1 `struct cublas`

The basis for creating an efficent *C*++ interface to CUBLAS is to use *C*++'s function overloading mechanism to define number type independent classes for matrices, vectors and views on, i.e. subsets of, matrices and vectors. At the bottom is a structure `cublas` providing an abstraction from the precision dependence of the names of the CUBLAS-functions:

```
struct cublas
{
    template<typename T>
    static void Alloc(T *&dev_ptr, size_t n) {

        cublasStatus status = cublasAlloc(n, sizeof(T), (void**)&dev_ptr);
        check_status(status);
    }

    static
    float dot(int n, const float *x, int incx, const float *y, int incy);

    static
    double dot(int n, const double *x, int incx, const double *y, int incy);
    // ...
};
```

These wrapper functions additionally provide an error-checking mechanism using CUBLAS' status tracking. As they are static we never have to instantiate an object of this class. A simple example for the encapsulation of the type-dependence consider the real-valued single precision version of the `dot`-function which computes the scalar product of two linear arrays:

```
float cublasSdot (int n,
                  const float *x, int incx,
                  const float *y, int incy) { ... }
    /* double and complex variants follow */
```

Either one looks it up in the CUBLAS reference [102] or one makes an educated guess about the meaning of the parameters. Important for us are the pointers `*x` and `*y` which point to the arrays to be multiplied in this function. The key to unified names is function overloading. We still have to stick with the same number of functions, but now the compiler picks the proper CUBLAS function according to the type information associated with the function arguments:

```
inline float
cublas::dot (int n, const float *x, int incx, const float *y, int incy)
{
    float result = cublasSdot(n, x, incx, y, incy);
    // check error status
    return result;
}
```

Declaring the wrapper functions as `inline` lets the compiler eliminate the additional function call. Making the wrapper functions static members eliminates the need to actually instantiate an object of type `cublas`. We can use the wrapper functions for encapsulating the error checking that is provided by CUBLAS. This can be done either unconditionally or only in `DEBUG` mode during the development phase of a project. The version for double precision exemplifies conditional compilation of the error-checking:

```
inline double
cublas::dot (int n, const double *x, int incx, const double *y, int incy)
{
    double result = cublasDdot(n, x, incx, y, incy);

#ifdef DEBUG
    cublasStatus status = cublasGetError();
    check_status(status);
#endif

    return result;
}
/* complex variants follow */
```

To realize a precision-independent scalar product, free of long argument lists, we overload the multiplication operator $*$ as member of the Vector class:

```
template<typename T, typename BW> class Vector {
        // constructors, etc ...

public:
    T operator * (const Vector<T> & other) const
    {
       BW::dot(this->__n_elements,
               this->__values, 1,
               other.__values, 1);
    }

private:
    T *    __values;       // pointer to first element of this vector
    int    __n_elements;   // length of this vector
```

We have not yet justified, why `cublas` has to be a structure and why it does not suffice to use a namespace or why we do not use some clever macro-based mechanism to achieve this type independence. Let's rule out the latter first. Often one finds macro-based function name re-definitions, like, e.g. in the code of SuperLU [82] (cf. file `sluCnames.h` in the sources):

```
#if (F77_CALL_C == ADD__)
#define sdot_  sdot__
...
#endif


#if (F77_CALL_C == UPCASE)
#define sdot_  SDOT
...
#endif
```

In our case we would use something like:

```
#ifdef USE_FLOAT
#define Number float
#define dot cublasSdot
...
#endif

#ifdef USE_DOUBLE
#define Number double
#define dot cublasDdot
...
#endif
```

Compilation would then be started with `-DUSE_FLOAT` or `-DUSE_DOUBLE` (g++-style preprocessor flags) so that the preprocessor would convert lines like:

```
    Number result = dot(n, x, incx, y, incy);
```

into the proper calls to `cublasSdot` or `cublasDdot`, respectively. Only in rare cases such constructs are easy to debug. Although the preprocessor replaces the function names at compile-time the debugger does not so at run-time. Thus it is not able to jump through the actual function call stack to the point where an error occured. This is one of the reasons why extensive use of preprocessor macros is discouraged in Google's C++ styleguide. On the other hand, in the wrapper-function approach the names in the actual function call stack correspond to what the debugger actually finds in the source code and thus tracking down bugs is easy. Especially, if they are of algorithmic type and not just simple segmentation faults.

Up to now we have merely discussed the generic limitations of the (CU)BLAS API. We have not used any specific CUBLAS feature so far, hence it should be clear that this approach works for other BLAS implementations as well. Be it the publicly available ATLAS library or Intel's *math kernel library* (MKL).

If, for example, we define our matrix class to have as template argument not only the number type but also an additional argument `BW` for the BLAS-wrapper structure we reach a much higher abstraction level. Neither matrix or vector class depend on a specific BLAS implementation nor does any algorithm which is exclusively based on these classes. This enables us to execute *exactly* the same algorithm on various types of hardware. This in turn yields much more realistic performance comparisons. Once an algorithm has been formulated, e.g. LU factorization, all we have to do is to define another structure `atlas` for wrapping up the ATLAS library, for instance. Then, if we have a class `LUFactorization<T>` which has already a template argument for the number type `T` we just add a second template argument for the BLAS wrapper and redefine the matrix type to be used:

```
template<typename T, typename BW> class LUFactorization {

public:
    typedef Matrix<T, BW> M_t;

    void factorize(M_t & A);
    // attributes for L, U, etc ...
};
```

1.4.2 `class Array<T>`

As basis for matrices and vectors we use a template class `Array` which encapsulates the memory management into a unified front end. Its sole purpose is to (re-)allocate `n` elements of type `T` and to provide access to the raw memory location in a controllable way through the `val` function. Access to the raw memory is required by the bare BLAS functions.

```
template <typename T, typename BW>
class Array
    :
    protected BW::template Data<T> {

    typedef typename BW::template Data<T> Base;

public:
    Array();

    Array(int n);

    void reinit(int n);

    T * val();

    const T * val() const;

    int n_elements() const { return __n; }

    Array<T, BW> & operator = (const Array<T, BW>& other);

protected:
    int __n;

        // avoid automatic generation of copy constructor
    Array(const Array<T, BW>& other) {}
};
```

1.4.3 `class Matrix<T, BW>`

SciPAL's matrix class is designed to conform to the interface of the `FullMatrix` class offered by *deal*.II. The memory management is delegated to the `Array` class. A very useful feature of *deal*.II is its `SmartPointer` class which provides reference-counting pointers based on a subscription mechanism. In order to let a smart pointer point to an object, the class the object is instantiated from must be derived from the `dealii::Subscriptor` class. The constructors, typedefs for template meta-programming purposes, assignment and incremental operators, matrix-vector and matrix-matrix multiplications behave as expected by *deal*.II-based programs. In addition, the `Matrix` class offers the initialization from an expression or a `dealii::IdentityMatrix` either via constructor or assignment operator. This includes the necessary memory operations. The class definition is given in following listing.

```
template<typename T, typename BW>
class Matrix
    :
    protected bw_types::Array<T, BW>, public dealii::Subscriptor {

public:
    typedef T Number;
    typedef T value_type;
    typedef BW blas_wrapper_type;

    Matrix();

    Matrix(int n_rows, int n_cols);

    Matrix(const Matrix<T, BW> & other);

    template<typename X> Matrix(const Expr<X> & AB);

    Matrix(const dealii::IdentityMatrix & Id);

    void reinit(int n_rows, int n_cols);

    template<typename X>
    Matrix<T, BW> & operator = (const Expr<X> & AB);
        // omitted: operator = Matrix
        //           operator = dealii::IdentityMatrix

        // omitted: operator +=, -= Matrix
        //           operator *= Number

    template<typename VECTOR1, typename VECTOR2>
    void vmult(VECTOR1& dst, const VECTOR2& src) const;
        // omitted: Tvmult

    void scaled_vmult(T beta, Vector<T, BW>& dst,
                      T alpha, const Vector<T, BW>& src) const;

        // dst = this * src
    void mmult(Matrix<T, BW>& dst, const Matrix<T, BW>& src) const;
    void mmult(SubMatrixView<T, BW>& dst, const Matrix<T, BW>& src) const;
        // omitted: other matrix-matrix multiplications

    int n_rows() const { return __n_rows; }
    int n_cols() const { return __n_cols; }

    T operator ()(const unsigned int i, const unsigned int j) const;

    T l2_norm() const;

    inline const bw_types::Array<T, BW> & array() const { return *this; }

private:
    int __n_rows;
    int __n_cols;
};
```

1.4.4 `class Vector<T, BW>`

Dense vectors which are compatible to the Krylov solver suite of *deal*.IIhave to provide a few BLAS1 functions but otherwise there are only little requirements on the class interface. To simplify data transfer between CPU and GPU there is an assignment operator which allows to copy either an STL `vector` or a `dealii::Vector` from the CPU side to the GPU. The reverse direction is provided by the `push_to` member functions. In order to copy directly into an STL `vector` or a `dealii::Vector` would require to change their interfaces.

```
template<typename T, typename BW>
class Vector {

    void reinit (const Vector&, bool leave_elements_uninitialized = false);

// members required by deal.II's Krylov solvers
    double operator * (const Vector &v) const;

    void add (const Vector &x);

    void add (const double a, const Vector &x);

    void sadd (const double a, const double b, const Vector &x);

    void equ (const double a, const Vector &x);

    Vector & operator *= (const double a);

    double l2_norm () const;

// stuff for CUDA
    inline Array<T, BW> & array() { return *this; }

    Vector<T, BW> & operator = (const std::vector<T> & other);

    Vector<T, BW> & operator = (const dealii::Vector<T> & other);

    template<typename T2>
    void push_to(std::vector<T2> & dst) const;

    template<typename T2>
    void push_to(dealii::Vector<T2> & dst) const;

    void set(int k,const T value);
}
```

1.4.5 `class SubMatrixView<T, BW>`

`SubMatrixView` primarily serves to modify only subsets of a matrix. For instance in the QR-factorization of a matrix $\mathbf{A}$ only the elements starting at the kth row and kth column have to be modified by the Householder transformation which eliminates the subdiagonal part of the kth column. To construct a view of a matrix the constructor of the view must get passed the matrix together with the indices of row and column ranges of the view in a $[begin, past\text{-}the\text{-}end)$-style as for iterators in the STL, see the following listing. In all other respects `SubMatrixView` offers the same operations as `Matrix`. In addition, it provides member functions for changing the range of rows and columns of the view. This class is extensively used in the operator-based formulation of the LU factorization in the next chapter.

```
template<typename T, typename BW> class SubMatrixView {

public:
    SubMatrixView(Matrix<T, BW> & src, int r_begin, int r_end,
                  int c_begin, int c_end);

    template<typename X>
    SubMatrixView<T, BW> & operator = (const Expr<X> & AB);

        // omitted: += and -=
        // omitted: access to raw data

    template<typename VECTOR1, typename VECTOR2>
    void vmult(VECTOR1& dst, const VECTOR2& src) const;

    template<typename VECTOR1, typename VECTOR2>
    void Tvmult(VECTOR1& dst, const VECTOR2& src) const;

private:
        // pointer to actual matrix
    dealii::SmartPointer<Matrix<T, BW> >__src;

        // omitted: default and copy constructor; copy-assignment
        // omitted: attributes for storing the index ranges passed to the
            constructor

public:
        // omitted: functions to read out dimensions of a view

        // move mask to another part of the matrix
    void shift(int m_r, int m_c);

    void reset(int new_r_begin, int new_r_end,
               int new_c_begin, int new_c_end);
};
```

1.4.6 Vector Views

The remaining core classes all follow the same logic as the the `SubMatrixView` class:

- `VectorView<T, M_or_V>` which is a base class for subvectors. It deduces the type of the blas wrapper from the template argument `M_or_V` which can be either the `Matrix` or the `Vector` class. The number type is passed as `T`.
- `ColVectorView<T, M_or_V>` implements a subvector of a column vector or a matrix column. The template arguments have the same meaning as for `VectorView<T, M_or_V>`.
- `RowVectorView<T, M_or_V>` provides a subvector of a row vector or a matrix row. The template arguments have the same meaning as for `VectorView<T, M_or_V>`.

Chapter 2

CUDA by Example: Inversion and Analysis of Dense Matrices

To outline the benefits of the DSEL concept forming the basis of SciPAL introduced in the preceding chapter, in particular the operator-based programming interface, we discuss the implementations of a few factorization methods for dense matrices.

The LU factorization demonstrates that an operator-based API can not only be utilized to encode evaluation but also to solve linear systems. It serves as a quick introduction to CUDA-programming and the necessary changes to the programming paradigms. Nonlinear Iterative Partial Least Squares *(NIPALS) is a popular algorithm for principal component analysis (PCA) [40]. Since an independent procedural CUBLAS implementation of NIPALS [12] exists, it was chosen as test case to verify that the operator-based interface does not incur any run-time overhead.*

2.1 LU Factorization

A standard operation in numerical linear algebra is solving a linear system of n equations with a dense coefficient matrix $\mathbf{A} \in \mathbb{K}^{n\times n}$. For simplicity we assume $\mathbb{K} = \mathbb{R}$. Provided $\mathbf{A}$ is nonsingular the common approach is not to invert $\mathbf{A}$ directly but rather to factorize $\mathbf{A}$ into two triangular matrices, an upper one $\mathbf{U}$ with unit diagonal and a lower one $\mathbf{L}$. Then, the solution is computable from solving two auxiliary linear systems of triangular shape by forward and backward substitution. The constitutive relation for the entries L_{rc} and U_{rc} of $\mathbf{L}$ and $\mathbf{U}$ is

$$\sum_{k=1}^{n} L_{rk}U_{kc} \quad = \quad A_{rc} \tag{2.1}$$

which can be made unique by requiring $U_{ii} = 1$. As integral part of standard textbook knowledge [53, 39, 60] LU factorization has a solid mathematical foundation and we can limit our focus to the CUDA-specific details.

In essentially all modern computer architectures memory accesses are much slower than the execution of floating operations. Therefore, performance is memory bound in most cases. As shown in detail in [39] this can be remedied by trying to formulate an algorithm in terms of matrix-matrix (i.e. BLAS3) operations as far as possible. BLAS3 operations have a better floating point operations over memory access ratio than BLAS2 or BLAS1. This increases performance by shadowing memory accesses by floating point computations.

In the following it suffices to know that partitioning $\mathbf{A}$, and thus $\mathbf{L}$ and $\mathbf{U}$, in 2×2 blocks the LU factorization can be written as

$$\mathbf{A} \equiv \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{1c} \\ \mathbf{A}_{r1} & \mathbf{A}_{rc} \end{pmatrix} = \begin{pmatrix} \mathbf{L}_{11} & \\ \mathbf{L}_{r1} & \mathbf{I}_{rc} \end{pmatrix} \begin{pmatrix} \mathbf{I}_{11} & \\ & \tilde{\mathbf{A}}_{rc} \end{pmatrix} \begin{pmatrix} \mathbf{U}_{11} & \mathbf{U}_{1c} \\ & \mathbf{I}_{rc} \end{pmatrix} \tag{2.2}$$

where $\tilde{\mathbf{A}}_{rc}$ is the Schur complement of $\mathbf{A}_{rc}$ and $\mathbf{I}_{11}$ and $\mathbf{I}_{rc}$ are identity matrices of suitable size. The skeleton of one iteration of a block-wise LU factorization has basically three steps:

1. Factorize upper left diagonal block $\mathbf{A}_{11}$. Find matrices $\mathbf{L}_{11}$ and $\mathbf{U}_{11}$ such that

$$\mathbf{L}_{11}\mathbf{U}_{11} = \mathbf{A}_{11} \tag{2.3}$$

   where $\mathbf{A}_{11} \in \mathbb{K}^{b\times b}$ is of blocksize $b \leq n$. The matrices $\mathbf{L}_{11}$ and $\mathbf{U}_{11}$ are of the same dimension and lower or upper triangular. The optimal choice of b depends on the hardware. Since this is the only step requiring a manual implementation we resolve it down to the element level:

$$for\ 1 \leq k \leq b:$$
$$\begin{aligned} L_{kk} &= A_{kk}/U_{kk} = A_{kk}, & & (2.4)\\ U_{kc} &= \frac{1}{L_{kk}} A_{kc}, & \forall\, k+1 \leq c \leq b & \quad (2.5)\\ L_{rk} &= A_{rk}/U_{kk} = A_{rk}, & \forall\, k+1 \leq r \leq b & \quad (2.6)\\ A_{rc} &= A_{rc} - L_{rk}U_{kc} & \forall\, k+1 \leq r,c \leq b & \quad (2.7) \end{aligned}$$

2. Solve triangular systems to compute the off-diagonal blocks

$$\mathbf{U}_{1c} = \mathbf{L}_{11}^{-1}\mathbf{A}_{1c}, \qquad \mathbf{L}_{r1} = \mathbf{A}_{r1}\mathbf{U}_{11}^{-1}. \tag{2.8}$$

   The equivalent systems $\mathbf{L}_{11}\mathbf{U}_{1c} = \mathbf{A}_{1c}$ and $\mathbf{L}_{r1}\mathbf{U}_{11} = \mathbf{A}_{r1}$ can be mapped to one of the `⋆trsm`- functions (the wildcard $\star$ represents the prefix for the particular BLAS implementation and the precision) which solve triangular systems with multiple right-hand sides. This will serve as an example how CCOs cannot only indicate the evaluation of an expression but also trigger the solution of a linear algebraic system.

3. Once the off-diagonal blocks $\mathbf{L}_{r1}$ and $\mathbf{U}_{1c}$ have been computed the part of $\mathbf{A}$ not yet factorized has to be modified for the next iteration by forming the Schur complement

$$\tilde{\mathbf{A}}_{rc} \;=\; \mathbf{A}_{rc} - \mathbf{L}_{r1}\mathbf{U}_{1c}. \tag{2.9}$$

   The Schur complement corresponds to a rank-b update of $\mathbf{A}_{rc}$ and can be realized by a generalized matrix-matrix product as provided by the `⋆gemm` function.

These steps have to be repeated for the modified $\mathbf{A}_{rc}$ until the number $n-r$ of remaining rows and columns is smaller than or equal to the blocksize b. In that case set $b = n-r$ and execute step-1 to finish the factorization as there is no lower right block left.

The goal is to implement these steps with overloaded operators as shown in listing 2.1. We assume that the overall algorithm is implemented as a member function of a class as in the complete listing 2.8 at the end of this section.

Listing 2.1: Body of the blocked *LU* factorization.

```
       SubMatrix L_11, U_11, A_11, L_r1, U_1c, ...
 2
       for (...)// loop over blocks on diagonal
 4     {
           L_11 * U_11 = A_11;
 6
           L_11 * U_1c = A_1c;
 8
           L_r1 * U_11 = A_r1;
10
           A_rest -= L_r1 * U_1c;
12     }
```

Provided the transpose of a matrix $\mathbf{A}$ is diagonally dominant numerical stability of the LU factorization is sufficient and pivoting can be avoided [53, Sec. 3.4.10].

2.1.1 Details for off-diagonal blocks

Before we discuss the computation of $\mathbf{L}_{11}$ and $\mathbf{U}_{11}$ to introduce CUDA programming by example we show how to define the CCO for solving triangular systems. The code is given in listing 2.3 which is written for clarity - not for the most general applicability. Provided properly named objects are in place we want that solving $\mathbf{L}_{11}\mathbf{U}_{1c} = \mathbf{A}_{1c}$ for $\mathbf{U}_{1c}$ is triggered by

```
L_11 * U_1c = A_1c;
```

and $\mathbf{L}_{r1}\mathbf{U}_{11} = \mathbf{A}_{r1}$ by the obvious counterpart (lines 7 and 9 in listing 2.1). The corresponding expressions must be made aware of several features of the operands: i) which operand represents the solution, ii) is the matrix upper or lower triangular, iii) is the diagonal filled with 1s. The first is easy: Just use `const` properly (line 5 in the listing below).

Listing 2.2: Operator for $\mathbf{L}_{r1}$.

```
  template <typename T>
2 inline
  LeftMSolve<T>
4 operator * (SubMatrix<T>&l,
              const SubMatrix<T>&r)
6 {
      return LeftMSolve<T>(l,r);
8 }
```

Listing 2.2 shows the operator needed for solving $\mathbf{L}_{r1}\mathbf{U}_{11} = \mathbf{A}_{r1}$. Tied to this operator is a structure `LeftMSolve` for holding references to the operands and encoding the particular operation. It should be obvious that with a proper `Traits` structure `LeftMSolve` can be made to model triangular systems of the type $AX = B$ as well. For convenience it is also possible to add the cases $XA^T = B$ and $A^T X = B$. Going away from BLAS this structure could also be used to implement the application of an incomplete LU-factorization.

Since communication with BLAS only happens via its wrapper class, `LeftMSolve` works for CPU-based implementations like ATLAS in exactly the same way as for GPU-based libraries like CUBLAS rendering CPU and GPU indistinguishable from a user's perspective.

Listing 2.3: Composition closure object for solving for the off-diagonal block $\mathbf{L}_{r1}$.

```
struct LeftMSolve
{  // XA = B
    SubMatrix<T> &l; const SubMatrix<T> &r;

    LeftMSolve(SubMatrix<T>& _l, const SubMatrix<T>& _r) : l(_l), r(_r) {}

    LeftMSolve<T> & operator = (const SubMatrix<T> & rhs)
    {
        l = *(rhs.__src); // Copy rhs for in-place solving

        char side   = 'R', uplo = 'U',        // Set up blas arguments.
             transa = 'N', diag = 'U';        // This is just for the sake
        int m = rhs.r_end() - rhs.r_begin(); // of  readability.
        int n = rhs.c_end() - rhs.c_begin();
        T alpha = 1.0;
        const T * const A = r.val();
        int lda           = r.leading_dim();
        T * B   = l.val();
        int ldb = l.leading_dim();

        BW::trsm(side, uplo, transa, diag,
                 m, n, alpha,
                 A, lda,
                 B, ldb);
        return *this;
    }
};
```

2.1.2 Factorization of Diagonal Block and Introduction to CUDA

The discussion of the computation of $\mathbf{L}_{11}$ and $\mathbf{U}_{11}$ primarily gives a brief introduction to programming with CUDA. Since CUDA is an extension of the *C* programming language writing code is straightforward after having got acquainted with the fact that the GPU memory is distinct from a computer's ordinary memory which implies explicit copy operations. Managing parallelism requires a special syntax for starting the kernels and a few new keywords and intrinsics/directives. The most important ones are `__global__`, `__shared__` and `__syncthreads()`.

CUDA in a Nutshell

If BLAS is used the factorization of the upper left diagonal block is the only step which requires some manual implementation. Basically, CUDA extends standard *C* by a few keywords and the concept of *single instruction multiple thread* execution in order to enable the programming of graphics cards. For successfully programming with CUDA one has to have a closer look at the hardware and to understand how to max out its capabilities. Otherwise there is no gain compared to an ordinary *C* program but the cost for development is considerably higher.

The most prominent feature of the hierarchical nature of the hardware is that not only the memory consists of several levels but also the compute resources as well. Computation is performed by compute cores pooled to so-called shared multiprocessors (SMP) of fixed size. In case of Tesla cards based on the Fermi architecture each SMP has 32 cores.

On the current Kepler architecture (GK104) one SMP has 384 cores. Due to its design for gaming purposes the performance for double precision (roughly $\leq 5\%$ of the single precision floating point performance) is rather low. Therefore, in the following we only consider Fermi. A Fermi-based GPU consists of up to 16 SMPs or 512 compute cores in total. The execution model is denoted rather as *single-instruction multiple threads* (SIMT) than *single instruction multiple data* (SIMD) as is common practice for multicore CPUs.

To run a function on the GPU it must be coded as a *kernel*. This is a function which can be called from the CPU but its body is executed on the GPU. Parallelism is achieved by starting several instances of a kernel at once. Each instance of a kernel is executed by a dedicated thread which is pinned to one compute core and destroyed when kernel execution finishes.

To map the large numbers of threads to the hardware thread execution is organized in a hierarchical manner as well. The logical unit for memory accesses and instruction execution is the *warp* which is formed by 32 threads. Up to 32 warps form a *thread block*. During its lifetime a thread block is assigned to a particular SMP and cannot migrate from one SMP to another. To hide memory latencies several thread blocks are executed on an SMP at once. How many, depends mainly on the consumption of registers and shared memory in the kernel. The precise number of threads per block for optimal performance is primarily a matter of the hardware parameters. Due to the access latencies at the various memory levels as a rule of thumb 256 is a good choice. Finally, thread blocks are organized in a grid whose dimension and shape are chosen to reflect the problem size. Blocks and grids can be 1-, 2- or 3-dimensional. The built-in variables `ThreadIdx` and `BlockIdx` enable a thread to figure out its position within a block of size `BlockDim` and and a grid of size `GridDim`.

The important parts of the memory hierarchy are the on-chip (i.e. on the single compute core) *registers*, on-chip (rather on-SMP) *shared memory* (SHM), respectively *L1-cache*, a shared off-chip *L2-cache* and finally the global memory accessible from both the GPU and via the PCIe-Bus from the CPU. Yet another level is obtained from using page-locking in the CPU-side memory and mapping it into the address space of the GPU.

Shared memory and L1-cache are both part of a fast on-SMP memory of 64 Kb which can be configured as either 48 Kb SHM + 16 Kb cache or 16 Kb SHM + 48 Kb cache. This can be tuned for each kernel invocation separately. The shared memory of an SMP is organized in 32 *banks* which can be simultaneously, i.e. within one cycle, read or written by the threads of a warp. For double precision access is not by warp but by half-warp because in general at most 128 Bytes can be read or written at a time.

The off-SMP L2-cache of a Fermi GPU consists of 768 Kb which is accessible from all SMPs. L1- and L2-cache buffer the accesses to the global memory. There is no direct access to the global memory anymore as in previous architecture generations. The caches are subdivided into *cachelines* of 128 Bytes each. To maximize performance all entries needed by the threads of a warp should be located in the same cacheline. Further details about the hardware can be found in the Fermi whitepaper [8] and the CUDA programming and best practices guides.

CUDA in Action

Triggering the kernel execution is shown in listing 2.4. After defining the dimensions of the grid of thread blocks (line 4) and of the thread block (line 5) the kernel is started like a *C* function. The sizes of the grid and thread blocks are passed by the `<<< ... >>>` syntax after the kernel's name. Here, we need a grid with one two-dimensional thread block as large as the matrix block.

Listing 2.4: Starting a CUDA kernel for factorizing the diagonal block.

```
template<typename T>
void LUKernels<T>::diag_block_lu(T *a_d, int block_offset,
                                 int n_remaining_blocks, int n_rows_padded){
    dim3 grid(1);
    dim3 threads(TILE_SIZE, TILE_SIZE);
    __diag_block_lu<<<1,threads>>>(a_d ,block_offset, n_rows_padded);
    cudaThreadSynchronize();
}
```

Listing 2.5: CUDA-based factorization of diagonal block.

```
    #DEFINE TILE_SIZE 16
 2  template <typename T> __global__
    void __diag_block_lu(const T *A, T *_LU, int block_id, int n_rows)
 4  {
        //  Prepare buffering A in shared memory.
 6      // Add a column to avoid bank conflicts
        // in column-wise access.
 8      __shared__ T LU[TILE_SIZE][TILE_SIZE + 1];

10      // position within thread block
        int row = threadIdx.y, col = threadIdx.x;
12
        // global matrix indices
14      int r = block_id * TILE_SIZE + row;
        int c = block_id * TILE_SIZE + col;
16
        // column-wise lexicographic indexing
18      int idx = n_rows * c + r;

20      // load matrix entries into shared memory
        LU[row][col]= A[idx];
22
        for(int k = 0; k < TILE_SIZE; ++k)
24      {
            if (row == k && col > k)
26              LU[row][col]/= LU[k][k];
            __syncthreads();
28
            if (row > k && col > k)
30              LU[row][col] -= LU[row][k] * LU[k][col];
            __syncthreads();
32      }
        _LU[idx] = LU[row][col];
34  }
```

Listing 2.5 shows the CUDA kernel for computing $\mathbf{L}_{11}$ and $\mathbf{U}_{11}$. Line 2 shows that CUDA supports template functions. The keyword `__global__` in line 2 indicates that it is a kernel.

As arguments we pass a pointer `const T *A` to the global memory position containing the matrix to be factorized, a pointer `const T *_LU` where the factors are to be stored, the id of the block on the diagonal which is to be factorized and the number of rows of $\mathbf{A}$ so that we can convert from 2D matrix indexing to a 1D lexicographical row-wise indexing.

Basically, each thread works on one element of $\mathbf{A}_{11}$, $\mathbf{L}_{11}$ and $\mathbf{U}_{11}$. Therefore, they will have to exchange data, that is those entries of $\mathbf{L}_{11}$ and $\mathbf{U}_{11}$ which have already been computed. This is only possible by storing them in shared memory which eventually sets a limit on the size of a matrix block. The factorization of the block is the task of a single thread block and therefore the size of the block is limited by the resources one thread block is able to allocate.

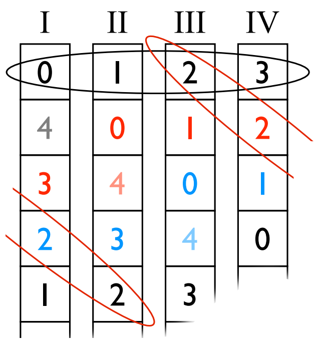


Figure 2.1: Odd-numbers of columns resolve conflicts in shared memory accesses.

Lines 11 to 18 serve figuring out which element of $\mathbf{A}$ the thread is responsible for.

Line 8 declares an array in shared memory which will be used to store intermediate results. The `TILE_SIZE` defined in line 1 indicates the number of rows and columns of the block and is a preprocessor macro because it must be known at compile-time. Passing it as template parameter did not work over several major CUDA revisions. There are basically two kinds of shared memory access which do not lead to *bank conflicts* which force the *thread scheduler* to serialize the individual accesses. For maximum throughput either no two threads of a warp read mutually distinct elements from one bank or parts of a warp try to read one element from one bank (broadcast read). Figure 2.1 illustrates how to avoid bank conflicts when distributing a 2D array with 4 columns over 4 banks which are numbered from I to IV. To fulfill that at a given time no two threads read different elements from the same bank the array must be padded by an extra column so that rows do not start in the same bank anymore. The same technique is used in line 8. The colored indices denote the entries within a row. The black and red ellipses indicate simultaneous access by row and column, respectively. The odd number of columns assures that each thread reads from a different bank even when reading a column is necessary.

Line 21: Each thread loads one entry of $\mathbf{A}$ from global into shared memory. Local coordinates in the block are `row` and `col`, `idx` is the lexicographic value of the global coordinates. Memory accesses coalesce and all of a warp's memory transactions are done in one cycle.

Line 23-32 performs the LU-factorization. Only the rows of $\mathbf{U}_{11}$ need to be explicitly computed since the columns of $\mathbf{L}_{11}$ are an implicit result of the Schur complement.

Let's walk through the `for`-loop: `k=0`: Work on row 0 and all of its off-diagonal elements. By def. $U_{11} = 1$ (in the single-element sense). Similar to the macroscopic off-diagonal blocks in Eq. (2.8) we compute the row $\mathbf{U}_{1c}$ by dividing by $L_{11} = A_{11}$ which is stored in `_LU[0][0]`. By construction $L_{r1} = A_{r1}$, thus there is nothing to do for the column $\mathbf{L}_{r1}$. The `x`-component of the built-in variable `threadIdx` runs fastest, hence the `if`-statement diverges only within one warp which limits the performance penalty due to the branch divergence to a minimum.

Line 27: The statement `__syncthreads()` synchronizes all threads within one thread block and makes sure that all threads have written their results back to shared memory before the next line of the kernel is executed by any other thread in the block. The synchronization is necessary because we need the updated values in the computation of the Schur complement in lines 29-30. Line 33 transfers the elements of $\mathbf{L}_{11}$ and $\mathbf{U}_{11}$ back to global memory.

Starting the Kernel

After going through the details of computing the entries of $\mathbf{L}_{11}$ and $\mathbf{U}_{11}$ in parallel we show how to encapsulate the CUDA code so that the overall program flow is not littered with CUDA-specific code snippets hampering portability. Instead the kernel is called indirectly via a wrapper function `diag_block_lu` which is a member of an interface class `LUKernels`.

The purpose of the interface class is to collect and hide the CUDA-specific details of the numerical back end in one place. A sketch of its structure is shown in listing 2.6. The host part of a program only needs to know the declarations of the wrapper functions and not their definitions. Calling the wrapper function from an overloaded operator is analogous to how the `trsm` BLAS-function is called in listing 2.3.

Listing 2.6: Interface to CUDA-based implementation.

```
template<typename T>
struct LUKernels<T>
{
    static void diag_block_lu (T *a_d,
                               int block_offset,
                               int n_remaining_blocks,
                               int n_rows_padded);

    // ... and other functions
};
```

Integration into Larger Projects

Providing full template specializations [129] of the various wrapper functions makes the NVidia *C*-compiler (nvcc) generate code for the given number types so that code generated by g++ can link against it. To be able to do this basically in one line, cf. listing 2.7, is yet another reason why all wrapper functions are collected in one class. Otherwise, each template wrapper function would require a specialization for each value its template argument may have. Compared to the indirect specialization by means of a specialized class template this leads to a lot of redundant code. Especially if kernels have more than one template argument and all reasonable combinations of template arguments have to be explicitly given.

Listing 2.7: Full template specialization of the interface class.

```
template class LUKernels<float>;

template class LUKernels<double>;
```

The next issue is the integration of a kernel into a larger program in a modular way so that switching back and forth between a CUDA and a non-CUDA implementation of a function is easy. A natural extension of this idea is to modularize the ”CUDA backend” of a program such that one can even switch between different parallelization techniques at runtime. How to do this in a modular manner to minimize impact on the implementation of algorithms is a subject of chapter 4 about real-time computation of phase holograms.

Listing 2.8: Body of the blocked LU factorization.

```
template <typename T>
void
LUDecomposer<T>::factorize(typename cublas<T>::Matrix &A)
{
    typedef  cublas::SubMatrixView<T> SMV;

    // Initialize L und U
    // as identity matrices.
    dealii::IdentityMatrix I(A.n_rows());

    L = I;
    U = I;

    const int bs = TILE_SIZE;
    const int n_blocks = (A.n_rows()+blocksize-1)/blocksize;

    for(int b=0; b< A.n_rows(); )
    {
        const SMV L_11 = L (b, b + bs, b, b + bs);
        const SMV U_11 = U (b, b + bs, b, b + bs);
        const SMV A_11 = A (b, b + bs, b, b + bs);

        // step 1: diagonal block
        // We must be able to
        // distinguish writable from
        // read-only submatrices.
        const_cast<SMV&>(L_11) * const_cast<SMV&>(U_11)  = A_11;


        if( b< n_blocks-1)
        {
            const SMV L_r1 = L (b + bs, A.n_rows(), b, b + bs);
            const SMV A_r1 = A (b + bs, A.n_rows(), b, b + bs);

            const SMV U_1c = U (b, b + bs, b + bs, A.n_cols());
            const SMV A_1c = A (b, b + bs, b + bs, A.n_cols());

            SMV A_rc = A (b + bs, A.n_rows(), b +  bs, A.n_cols());

            // step 2: off-diagonal blocks
            // Syntax highlighting makes it clear
            // which factor will contain the solution
            // of the triangular systems.
            L_11                   * const_cast<SMV&>(U_1c)  = A_1c;

            const_cast<SMV&>(L_r1) *                    U_11   = A_r1;

             // step 3: Schur complement
            A_rc -= L_r1 * U_1c;
        }

        b += std::min(bs, A.n_rows() - b);
    }
}
```

2.2 Iterative Principal Component Analysis

Principal component analysis (PCA) [107, 40] is a standard technique in multivariate data analysis to structure large data sets and hence to extract useful information from it. It is also known as Karhunen Loeve decomposition [70, 85] or empirical orthogonal functions [137].

The essence of a PCA is to compute a partial singular value decomposition (SVD) in order to obtain the k largest and dominating singular values and the associated singular vectors. The SVD of a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ is given by

$$\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T = \sum_{i=1}^{n} \sigma_i \mathbf{u}_i \mathbf{v}_i^T$$

where $\mathbf{U} \in \mathbb{R}^{m\times n}$ and $\mathbf{V} \in \mathbb{R}^{n\times n}$ orthogonal matrices and $\Sigma \in \mathbb{R}^{n\times n}$ is a diagonal matrix containing the singular values σ_i sorted according to their size. The columns $\mathbf{u}_i$ of $\mathbf{U}$ and $\mathbf{v}_i$ of $\mathbf{V}$, respectively, are the left- and right-singular vectors for the corresponding singular values. In the context of PCA the right-singular vectors are called *principal component directions* [57] and the left ones *normalized principal components*. The standard or direct approach to computing an SVD is a two-stage process. At the beginning the matrix is converted to bidiagonal form by using Householder transformations. The second, and much cheaper, step diagonalizes the intermediate bidiagonal matrix and returns the vector of singular values and two orthogonal matrices which multiplied with the orthogonal matrices from step one yield the left- and right-hand side singular vectors.

To avoid the expensive bi-diagonalization step necessary in direct SVD methods, fast PCA algorithms, especially for very large data sets are formulated as iterative methods. An example of everyday importance for a very large data set is the Google matrix for website connectivity with billions of rows and columns [99]. A simple but effective implementation is the *Nonlinear Iterative Partial Least Squares* (NIPALS) algorithm [143] which can be considered as a truncated SVD which computes the leading singular values and vectors by applying the power method for eigenvalue computation to the covariance matrix. In particular it can be completely formulated in terms of BLAS functions and there are investigations of its performance on different hardware architectures [12, 13].

2.2.1 NIPALS Algorithm

After organizing the raw data as a matrix $\mathbf{X} \in \mathbb{R}^{m\times n}$ where each row contains the result of one run of an experiment, e.g. a time series, the goal is to represent the data as a useful part $\hat{\mathbf{X}} \in \mathbb{R}^{m\times n}$ and a residual matrix $\mathbf{R} = \mathbf{X} - \hat{\mathbf{X}}$. The columns of $\mathbf{X}$ must be mean centered, that is to compute the average of the entries of a column and to subsequently subtract that average from the entries of the column.

The NIPALS algorithm [143, 76] iteratively computes a truncated SVD of the mean-centered data matrix $\mathbf{X}$. The pseudo code and a *SciPAL*-based implementation are given in Fig 2.2. For a comparison of implementations an implementation which makes direct use of CUBLAS in double precision is given in listing 2.9. In the field of data analysis the columns of the matrix $\mathbf{U}\Sigma$ are usually called *scores* and the columns of $\mathbf{V}$ *loadings*. We stick with this notation by defining $\mathbf{S}$ as the subset of the K first columns of $\mathbf{U}\Sigma$ and $\mathbf{L}$ as the K first columns of $\mathbf{V}$. For individual columns or rows of those quantities we actually compute we use a MATLAB like syntax, e.g. $\mathbf{L}(:,k)$ denotes the k-th column of matrix $\mathbf{L}$.

The NIPALS algorithm is given in pseudo code form in Fig. 2.2 and works as follows. The residual matrix $\mathbf{R}$ is initialized by the data $\mathbf{X}$ (steps 1). The first K principal components are extracted one at a time (step 2). The largest eigenvalue of the covariance matrix $\mathbf{X}\mathbf{X}^T$ and the largest singular value of $\mathbf{X}$ can be characterized by the Rayleigh quotient

$$\lambda = \sigma_1^2 = \max_{\mathbf{v}\in\mathbb{R}^m, \mathbf{v}\neq\mathbf{0}} \frac{\mathbf{v}^T\mathbf{X}\mathbf{X}^T\mathbf{v}}{\mathbf{v}^T\mathbf{v}}.$$

Therefore, using the power method for computing the largest eigenvalue and its associated eigenvector of $\mathbf{X}\mathbf{X}^T$ gives σ_1 and $\mathbf{u}_1$. This is, what steps 8-10 in the pseudo code in Fig. 2.2 do.

Interleaved with the computation of $\sigma_k\mathbf{u}_k \equiv \mathbf{S}(:,k)$ is the determination of $\mathbf{v}_k \equiv \mathbf{L}(:,k)$ in step 7. Note that due to step 9 we have $\mathbf{R}^T\mathbf{S}(:,k) = \mathbf{R}^T\mathbf{R}\mathbf{L}(:,k)$ which corresponds to an application of the power method to $\mathbf{X}^T\mathbf{X}$. When $\mathbf{S}(:,k)$ is close to $\sigma_k\mathbf{u}_k$ we have $\mathbf{R}^T\mathbf{S}(:,k) \approx \mathbf{V}\Sigma\mathbf{U}^T\sigma_k\mathbf{u}_k = \mathbf{v}_k\sigma_k^2$.

Once the difference $|\lambda - \lambda'|$ of two successive approximations of an eigenvalue of the covariance matrix $\mathbf{X}\mathbf{X}^T$ has become sufficiently small the data matrix is deflated (step 12) which removes any contribution due the principal component from the data so that the next principal component will be orthogonal to all previously computed ones (at least in exact arithmetic). After $K \leq n$ iterations, the decomposition of $\mathbf{X}$ is

$$\mathbf{X} = \sum_{i=1}^{K} \sigma_i\mathbf{u}_i\mathbf{v}_i^T + \mathbf{R}.$$

The sample code in Fig. 2.2 demonstrates the major advantages of *SciPAL*, i.e. mathematically intuitive source code due to expression templates. In order to verify an implementation of the NIPALS-PCA is correct we compute test data from orthogonal matrices $\mathbf{U}$ and $\mathbf{V}$ as described in Sec. 2.2.2 and given singular values. A test of NVidia's CUBLAS against ATLAS in both single and double precision by a PCA of the data `A` can be as dense and clear as in the following code. The details for timing measurements have been omitted.

```
void test_nipals()
{
    PCA<float, cublas> pca_f_cuda;
    pca_f_cuda.nipals(A);

    PCA<double, atlas> pca_d_atlas;
    pca_d_atlas.nipals(A);
}
```

With our template-based approach, we have to implement NIPALS only once. By specializing the `PCA` class we can generate different implementations for varying floating point types and BLAS implementations within a single line of code as it is exemplified in the `test_nipals` function above. Therefore, we achieve the same results with less than a quarter of code in a much more intuitive notation.

```
1.  $\mathbf{R} \leftarrow \mathbf{X}$
2.  for $(k = 0, ..., K-1)$ do {
3.      $\lambda = 0, \lambda' = 2\varepsilon$
4.      $\mathbf{S}(:,k) = \mathbf{R}(:,k)$
5.      while $(|\lambda - \lambda'| > \varepsilon)$ do {
6.        $\lambda \leftarrow \lambda'$
7.        $\mathbf{L}(:,k) \leftarrow \mathbf{R}^T\mathbf{S}(:,k)$
8.        $\mathbf{L}(:,k) \leftarrow \mathbf{L}(:,k) \,/ \|\mathbf{L}(:,k)\|$
9.        $\mathbf{S}(:,k) \leftarrow \mathbf{R}\mathbf{L}(:,k)$
10.       $\lambda' \leftarrow \|\mathbf{S}(:,k)\|$
11.     }
12.     $\mathbf{R} \leftarrow \mathbf{R} - \mathbf{S}(:,k)(\mathbf{L}(:,k))^T$
13. }
```

```
template <typename T, typename blas>
void
PCA<T, blas>::PCA::nipals(const
    HostMatrix& X)
{
    R = X;

    transpose<Matrix> R_T(R);

    for (int k = 0; k < n_components;
        k++)
    {
        T lambda = 0.,
        lambda_old = 0.;

        bool converged = false;

        MatrixSubCol t_k(T, 0, k);
        MatrixSubCol p_k(P, 0, k);
        MatrixSubCol r_k(R, 0, k);

        t_k = r_k;

        for (int j = 0;
             j < max_iter &&
                 !converged; j++)
        {
            p_k = R_T * t_k;
            p_k /= p_k.l2_norm();

            t_k = R * p_k;
            lambda = t_k.l2_norm();

            converged =
              std::fabs(lambda -
                  lambda_old)
                <= num_zero;

            lambda_old = lambda;
        }
        R -= t_k *
        transpose<MatrixSubCol>(p_k);
    }
}
```

Figure 2.2: Left: NIPALS Pseudo code. For individual columns or rows we use a MATLAB like syntax, e.g. $\mathbf{L}(:,k)$ denotes the k-th column of matrix $\mathbf{L}$. Right: Implementation of NIPALS using *SciPAL*. In particular, error checking is completely hidden in *SciPAL* and thus executed automatically. In contrast to the bare CUBLAS implementation *SciPAL* takes into account error reports of CUBLAS functions other than those for memory allocation. Note the light-weight views on the k-th columns of the load and score matrix.

Listing 2.9: NIPALS implementation taken from [12]. Most of the code is devoted to error checking which is partially replaced by ”...” due to space constraints.

```
int nipals_cublas(int M, int N, int K, double *T, double *P, double *R)
{
cublasStatus status;
int J = 10000;
double er = 1.0e-7;
int k, n, j;

double *dR = 0;
status = cublasAlloc(M*N, sizeof(dR[0]), (void**)&dR);
if(status != CUBLAS_STATUS_SUCCESS)
    ...
status = cublasSetMatrix(M, N, sizeof(R[0]), R, M, dR, M);
if(status != CUBLAS_STATUS_SUCCESS)
    ...
double *dT = 0;
status = cublasAlloc(M*K, sizeof(dT[0]), (void**)&dT);
if(status != CUBLAS_STATUS_SUCCESS)
    ...
double *dP = 0;
status = cublasAlloc(N*K, sizeof(dP[0]), (void**)&dP);
if(status != CUBLAS_STATUS_SUCCESS)
    ...
double *dU = 0;
status = cublasAlloc(M, sizeof(dU[0]), (void**)&dU);
if(status != CUBLAS_STATUS_SUCCESS)
    ...
cublasDcopy(M, &dR[0], 1, dU, 1);
for(n=1; n<N; n++) cublasDaxpy(M, 1.0, &dR[n*M], 1, dU, 1);
for(n=0; n<N; n++) cublasDaxpy(M, -1.0/N, dU, 1, &dR[n*M], 1);

double a, b;
for(k=0; k<K; k++) {
    cublasDcopy(M, &dR[k*M], 1, &dT[k*M], 1);
    a = 0.0;
    for(j=0; j<J; j++) {
        cublasDgemv( t , M, N, 1.0, dR, M, &dT[k*M], 1, 0.0, &dP[k*N], 1);
        cublasDscal(N, 1.0/cublasDnrm2(N, &dP[k*N], 1), &dP[k*N], 1);
        cublasDgemv( n , M, N, 1.0, dR, M, &dP[k*N], 1, 0.0, &dT[k*M], 1);
        b = cublasDnrm2(M, &dT[k*M], 1);
        if(fabs(a - b) < er*b) break;
        a = b;
    }
    cublasDger(M, N, -1.0, &dT[k*M], 1, &dP[k*N], 1, dR, M);
}
cublasGetMatrix(M, K, sizeof(dT[0]), dT, M, T, M);
cublasGetMatrix(N, K, sizeof(dP[0]), dP, N, P, N);
cublasGetMatrix(M, N, sizeof(dR[0]), dR, M, R, M);

status = cublasFree(dP);
status = cublasFree(dT);
status = cublasFree(dR);
return EXIT_SUCCESS;
}
```

2.2.2 Designing Matrices for Factorization Tests

Validation of factorization methods requires well-defined test matrices which are to be recovered by factorizing their product. Constructing a matrix from a product of two given matrices is simple. However, most factorization methods have as result one or more orthogonal matrices. For instance, given an orthogonal matrix $\mathbf{Q} \in \mathbb{R}^{m\times m}$ and an upper triangular matrix $\mathbf{R} \in \mathbb{R}^{m\times n}$ a *QR*-factorization of $\mathbf{A} = \mathbf{QR}$, see e.g. [53], should restore $\mathbf{Q}$ and $\mathbf{R}$ up to numerical accuracy. Singular value decomposition, see again [53], and NIPALS both produce two orthogonal matrices. Especially in the presence of finite precision the construction of orthogonal matrices is difficult. A simple way to construct orthogonal matrices by Sylvester [125] dates back to 1867. Starting from the 1-by-1 matrix $\mathbf{H}_0 = (1)$ so-called Hadamard matrices with sizes of power of two can be constructed recursively

$$\forall k \geq 1 \; : \quad \mathbf{H}_k := \begin{pmatrix} \mathbf{H}_{k-1} & \mathbf{H}_{k-1} \\ \mathbf{H}_{k-1} & -\mathbf{H}_{k-1} \end{pmatrix} . \tag{2.10}$$

Their entries are either $+1$ or -1. They are, up to a factor k, orthogonal and symmetric which leads to the, for testing purposes, important property

$$\mathbf{H}_k\mathbf{H}_k^T = \mathbf{H}_k^T\mathbf{H}_k = \mathbf{H}_k^T\mathbf{H}_k^T = k\mathbf{I}.$$

We generate matrices of sizes which are not a power of 2 as block-diagonal matrices where the blocks are filled with the lowest number of normalized Hadamard matrices. Residual matrices are generated as $\mathbf{R}_{ij} = 1 + \sin(i \cdot n_{rows} + j)$. This is far from optimal from the point of view of pseudo random number generation but easy to remember and easy to test.

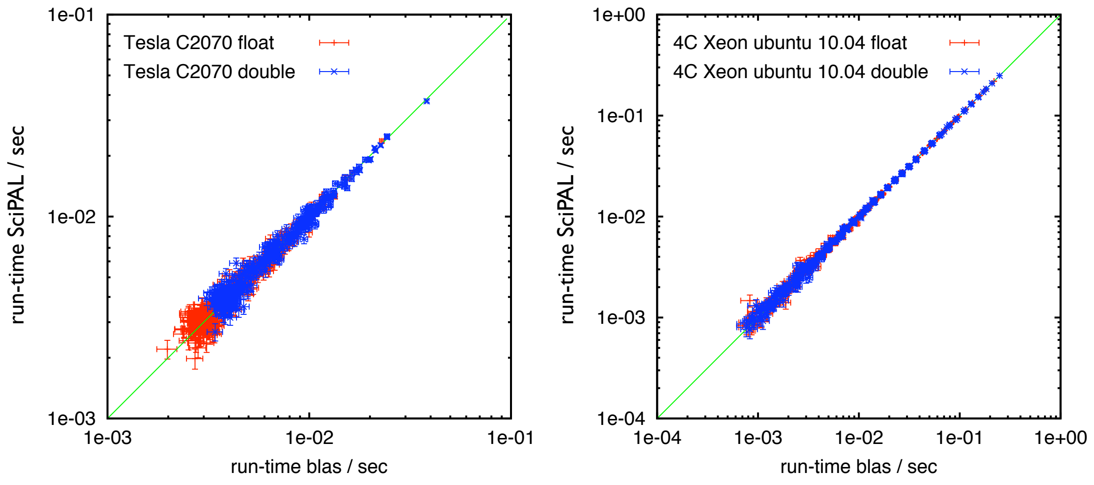


Figure 2.3: CUBLAS test on Tesla C2070 (left) and ATLAS test on Nehalem Xeon (right). T_{DSEL} is on the ordinate and T_{proc} on the abscissa. Problem size ranged from 64 to 2048. For each matrix size 313 runs were executed.

2.2.3 Performance Comparisons

To check that *SciPAL* comes without any additional computational costs, we test the plain BLAS implementation [13] of NIPALS by Andrecut against the implementation using *SciPAL*

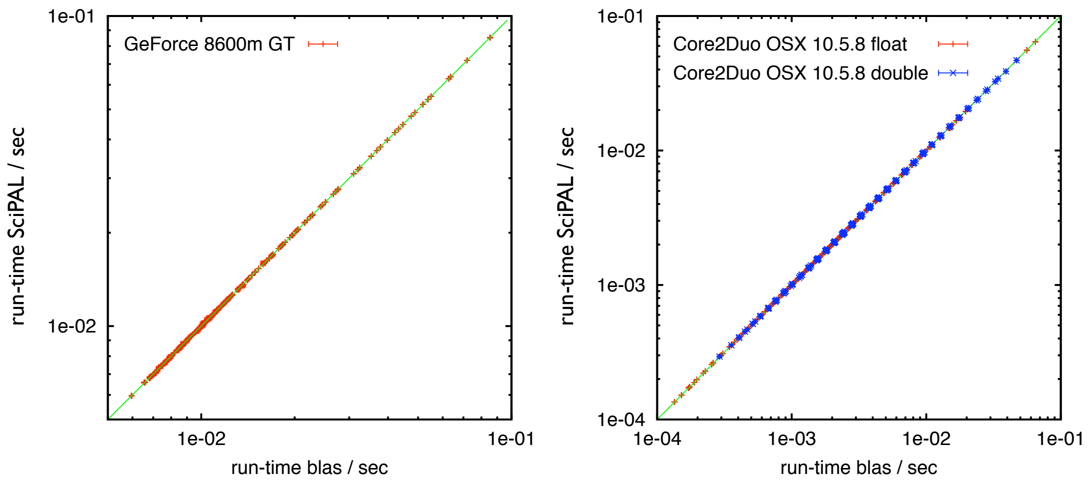


Figure 2.4: CUBLAS test (left), ATLAS test (right) on the MacBook Pro. T_{DSEL} is on the ordinate and T_{proc} on the abscissa. Problem size ranged from 64 to 882 rows. For each number of rows the number of columns increased by 17%. For each matrix 313 repeated runs were executed. On this platform the GPU lacks support for double precision.

for both the CUBLAS and ATLAS library. We run each instance several times for every matrix size to compute average run-times T and their standard deviations ΔT, which are non-vanishing due to mechanisms like temperature control and possibly concurring tasks on the same machine. The runtime for the plain BLAS implementation is denoted as T_{proc} and the one due to *SciPAL* as T_{DSEL}. From our measurements it is simple to determine for which problem sizes computation on graphics cards is superior to a run on the CPU and whether our *C*++-wrapper is free of any run-time overhead on both architectures.

The tests were performed on two different machines with two different graphics cards. The first test site is a MacBook Pro with a Core 2Duo T9300, NVidia 8600m GT, Mac OS X 10.5.8 and Apple's g++ 4.0.3 as compiler. The 8600m GT GPU does not provide double precision arithmetic and therefore tests are limited to the single precision case. The second site is a workstation equipped with a quad-core Nehalem Xeon E5520 as CPU and a NVidia Tesla C2070 based on NVidia's Fermi architecture as GPU. The operating system is ubuntu 10.04 with g++ 4.4.3. The optimization switches for the compiler were identical on both machines: `-finline-limit=2000 -Os -arch i386`. When plotting T_{DSEL} vs. T_{proc} as in Figs. 2.4 and 2.3 the result should be a straight line of points with slope 1. This is indeed the case indicating that both implementations were equally fast. This proves our claim of zero run-time overhead. We consider run-times as equal on average, i.e. neither implementation is faster and especially our object-oriented one is not slower, if both intervals $[T_{DSEL} - \Delta T_{DSEL}, T_{DSEL} + \Delta T_{DSEL}]$ and $[T_{proc} - \Delta T_{proc}, T_{proc} + \Delta T_{proc}]$ intersect each other and both contain the diagonal. For large problems (upper right corner in Fig. 2.3) the high parallelism of the GPU architecture starts to pay off and runtimes are shorter than on the CPU by an order of magnitude, which appears as different scaling of the axes. Note the dependency of GPU-runtimes on computation precision, i.e. faster computation with single precision than with double precision, which is not present in the CPU-runtimes of the Xeon system.

Chapter 3

CUDA-based Sparse Linear Algebra

Parallel numerical linear algebra is not complete without the sparse case. This chapter summarizes the integration of CUDA into the linear algebra modules of an existing PDE library and of an established Navier-Stokes simulation code.

Integration into deal.II is easy: write a class for sparse matrices which copies the data of a `dealii::SparseMatrix` *on the GPU and offers a* `vmult` *member function for the sparse matrix-vector product. The other crucial ingredient, a suitable vector class, has already been introduced in chapter 1. The Krylov solvers of deal.II are templatized with respect to the matrix and vector class. Thus, for them there is nothing to do.*

As example for an existing PDE code we chose PNS [105, 73], the Parallel Navier-Stokes *solver for indoor airflow used at the* Institut für Energietechnik *at the* TU Dresden *and developed at the* Institut für Numerische und Angewandte Mathematik *at the University of Göttingen. The PNS code is written in plain* C. *Therefore, the integration of CUDA proved challenging. As a result, this chapter presents only a high-level description of the necessary changes limiting code snippets to the body of a CUDA implementation of a sparse matrix-vector product.*

In a very terse form Secs. 3.2 and 3.3 have been published previously as part of [77].

3.1 Krylov solvers

The last chapter introduced a small set of tools to solve linear systems which have a dense coefficient matrix. If the system is uniquely solvable one uses LU otherwise QR factorization. The principal component analysis can be considered as a least squares solver in disguise [41, Sec. 7.2.3]. In contrast to direct methods it solves a problem by iteratively computing the dominant part of the spectrum and the associated eigensubspace using the power method. If one were to solve the least squares problem hidden in the PCA one would then seek the solution in the eigensubspace. Since eigenvectors are mutually orthogonal this procedure can be considered as a projection method in which the approximate solution is orthogonal to the residual and equals the projection of the true solution into the subspace spanned by the dominant eigenvectors.

This idea carries over to solving linear algebraic systems with a large and sparse coefficient matrix $\mathbf{A} \in \mathbb{K}^{n\times n}$, $\mathbb{K} = \mathbb{R}$ or $\mathbb{K} = \mathbb{C}$, by Krylov methods. In contrast to the iteration of the power method used in the NIPALS-PCA in Sec. 2.2 the history of all intermediate vectors is kept. The subspace in which the solution is sought is denoted as the Krylov subspace

$$\mathscr{K}_m(\mathbf{A},\mathbf{v}) \quad := \quad \operatorname{span}\{\mathbf{v},\mathbf{A}\mathbf{v},\ldots,\mathbf{A}^{m-1}\mathbf{v}\}\,. \tag{3.1}$$

For general non-hermitian matrices the Arnoldi procedure, introduced in 1951, provides an orthogonal projection onto the subspace $\mathscr{K}_m(\mathbf{A},\mathbf{v}) \subset \mathbb{K}^{n\times n}$. For symmetric matrices this reduces to the Lanczos tridiagonalization. In contrast to the truncated SVD the subspace depends not only on the matrix $\mathbf{A}$ but also on the initial residual $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$ and thus on the right-hand side $\mathbf{b}$ and initial solution $\mathbf{x} \in \mathbb{K}^n$. From this a set of orthogonal vectors $\mathbf{v}_j$ and a Hessenberg matrix $\mathbf{H}$ is computed which maps the $\mathbf{v}_j$s to the non-orthogonal $\mathbf{A}^k\mathbf{r}_0$:

$$\mathbf{r}_0 \quad = \ (\mathbf{b} - \mathbf{A}\mathbf{x}_0)\,, \quad \beta = \|\mathbf{v}_0\|_2\,, \quad \mathbf{v}_1 \ = \ \mathbf{r}_0/\beta \tag{3.2}$$

$$\mathbf{v}_m \quad := [\mathbf{v}_1,\ldots,\mathbf{v}_m]\ \mathbf{H}_m := \{h_{ij}\}_{i,j} \in \mathbb{K}^{(m+1)\times m} \tag{3.3}$$

$$\textbf{for} \quad j = 1,\ldots,m: \tag{3.4}$$

$$\mathbf{w} := \mathbf{A}\mathbf{v}_j \tag{3.5}$$

$$\textbf{for } i = 1,\ldots,j: \tag{3.6}$$

$$h_{ij} \ = \ \mathbf{w}^T\mathbf{v}_i \tag{3.7}$$

$$\mathbf{w} \quad = \mathbf{w} - h_{ij}\mathbf{v}_i \tag{3.8}$$

$$h_{j+1,j} \ = \ \|\mathbf{w}\|_2 \tag{3.9}$$

$$\mathbf{v}_{j+1} \ = \ \mathbf{w}/h_{j+1,j} \tag{3.10}$$

The Hessenberg matrix describes a least squares problem

$$\mathbf{y}_m \quad = \operatorname{argmin}_y \|\beta\mathbf{e}_1 - \mathbf{H}_m\mathbf{y}\| \tag{3.11}$$

which helps to solve the original set of equations

$$\mathbf{x}_m \quad = \mathbf{x}_0 + \mathbf{V}_m\mathbf{y}_m\,, \tag{3.12}$$

$$\mathbf{r}_m \quad = \mathbf{M}^{-1}(\mathbf{b} - \mathbf{A}\mathbf{x}_m)\,. \tag{3.13}$$

This extension of the Arnoldi method is the essence of the *Generalized Minimual Residual* (GMRES) method [113]. If $\mathbf{A}$ is symmetric the Hessenberg matrix becomes a symmetric tridiagonal matrix. In that case Arnoldi becomes the popular *Lanczos* iteration with a tree-term recurrence for the h_{ij}. It is the basis for the computation of eigenvalues of symmetric matrices and the *conjugate gradient* algorithm to solve systems of linear equations with a sparse, symmetric coefficient matrix. A comprehensive list of Krylov methods and a detailed analysis of their theoretical properties can be found in [112].

From a practical point of view Krylov methods are easy to implement since all they need are matrix-vector multiplications and a few BLAS-1 operations of which the scalar product often suffices. These properties make Krylov methods robust black-box solvers for a large class of problems. The linear algebra classes of *deal*.II are a fairly complete set of current implementations of these methods.

Due to their template nature *deal*.II's Krylov solvers can be ported to CUDA easily: All one has to do is to provide a class for vectors and one for sparse matrices which internally have their data on the GPU and use CUDA for the arithmetic. As class for the vectors the one presented in Sec. 1.4.4 can be used. Like the class for dense matrices, cf. Sec. 3.3, the class for sparse matrices only has to offer the `vmult` and `Tvmult` member functions for the products $\mathbf{A}\cdot\mathbf{x}$ and $\mathbf{A}^T\cdot\mathbf{x}$, respectively. If the latter is not referenced by a Krylov solver we do not have to implement it either and get away with even less work.

3.2 Preconditioning

Compared to the dimension of the original problem Krylov methods generally need only a small number of basis vectors to converge. However, for badly conditioned matrices convergence quickly deteriorates. In general, the condition number is the ratio $\|\mathbf{A}\|/\|\mathbf{A}^{-1}\|$ for some matrix norm $\|\cdot\|$. For symmetric matrices the condition number can be defined as the ratio of the absolute values of the extremal eigenvalue and the smallest eigenvalue. Especially matrices representing discretized partial differential equations have a condition number which rapidly grows with spatial resolution as the smallest eigenvalue typically tends to zero with diminishing mesh width. The convergence properties of Krylov methods can be restored and improved by preconditioning the matrix, i.e. to get the condition number somehow close to unity.

From the plethora of preconditioning methods we consider four in this work. In the computation of indoor airflow, cf. chapter 5, the sparse approximate inverse (SpAI) method and polynomial preconditioning (PPc) with Faber polynomials is benchmarked against an existing incomplete LU factorization (ILU) from the BLANC package [108]. The general problem with factorization-based preconditioners is that the forward and backward solve necessary in each step of the Krylov method is a priori difficult to parallelize. The other two methods are Block-Jacobi and geometric multigrid. Strictly speaking this is only one method as we use Block-Jacobi as smoother in the multigrid cycle and not as standalone preconditioner.

Particularly suitable are sparse approximate inverses as they only need sparse matrix-vector products for application. For their initialization merely approximate sparse matrix-sparse matrix products are required which boil down to inner products of sparse vectors. It seems that there has been published only little about CUDA-based parallel preconditioners for the non-symmetric case and even less for matrices arising from multiphysics applications like non-isothermal air-flow. For factorization-based preconditioners there exist a parallel implementation of block-diagonal ILU [136], ILU [59] and a biorthogonalization-based SpAI [144] which is the work most closely related to ours. However, [144] only measures the speedup of a CUDA-based implementation of SpAI over an OpenMP-based one for different sparsification strategies. We compare the performance of a serial ILU-implementation with an unfactored SpAI. Sparse approximate inverses can tackle indefinite matrices as well and thus have a broader scope of applicability than ILU.

3.2.1 Concept

We now take a closer look at the parallel preconditioning techniques applied in the iterative solution of the discretized indoor airflow problems Eqs. (5.15) and (5.16) by Krylov-type methods [112]. Given a linear system

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

with $\mathbf{A} \in \mathbb{R}^{n\times n}$ and $\mathbf{x}, \mathbf{b} \in \mathbb{R}^n$ where $\mathbf{x}$ is the sought solution, the essence of preconditioning is to find a matrix $\mathbf{M} \in \mathbb{R}^{n\times n}$ of which the inverse $\mathbf{M}^{-1}$ is easy to compute and yet approximates the inverse $\mathbf{A}^{-1}$ of $\mathbf{A}$ well. Once a suitable $\mathbf{M}$ has been found the solution of $\mathbf{A}\mathbf{x} = \mathbf{b}$ is obtained either by solving the right-preconditioned system

$$\mathbf{A}\mathbf{M}^{-1}\mathbf{y} = \mathbf{b}, \tag{3.14}$$

$$\mathbf{x} = \mathbf{M}^{-1}\mathbf{y} \tag{3.15}$$

or by solving the left-preconditioned one

$$\mathbf{M}^{-1}\mathbf{A}\mathbf{x} = \mathbf{M}^{-1}\mathbf{b}. \tag{3.16}$$

Depending on the preconditioning strategy computing the inverse is meant either literally or in the sense of solving auxiliary linear systems. The SpAI and PPc strategy belong to the former and ILU to the latter.

In the SpAI case an approximate inverse $\mathbf{M}^{-1} \approx \mathbf{A}^{-1}$ is computed so that applying $\mathbf{M}^{-1}$ reduces to a single SpMV operation. Based on information about the spectrum $\sigma(\mathbf{A}) \subset \mathbb{C}$ of $\mathbf{A}$ the PPc strategy computes a polynomial representation of $\mathbf{M}^{-1}$ from the shape of an inclusion set $D \supset \sigma(\mathbf{A})$, $D \subset \mathbb{C}$. In this case preconditioning means to perform a number of SpMV operations proportional to the polynomial degree.

3.2.2 Block-Jacobi

The simplest preconditioning strategy is to use the inverse of the diagonal of $\mathbf{A}$. Compared to modern methods like ILU and geometric or algebraic multigrid methods the plain Jacobi method is only of historical interest. A more effective way is to use a block-diagonal matrix where each block is the inverse of the corresponding block in $\mathbf{A}$. If the blocks are considered as dense submatrices the LU-Kernel from Sec. 2.1.2 can be used to compute them in parallel. The subsequent application of the preconditioner amounts to a simple matrix-vector multiplication which we anyway have to parallelize for implementing a Krylov method. This Block-Jacobi preconditioner can be used as smoother in the geometric multigrid methods employed in the simulation of dielectric relaxation spectroscopy in the last chapter of this work.

3.2.3 Sparse Approximate Inverse

To compute a sparse $\mathbf{M}^{-1} \approx \mathbf{A}^{-1}$ we solve the n independent minimization problems for the columns $\mathbf{m}_j$ of $\mathbf{M}^{-1}$ where $\mathbf{e}_j$ is the jth column unit vector

$$\mathbf{M}^{-1} := \operatorname*{arg\,min}_{S\in\mathbb{R}^{n\times n}} \|\mathbf{I} - \mathbf{A}\mathbf{S}\|_F\,, \qquad \mathbf{m}_j := \operatorname*{arg\,min}_{s\in\mathbb{R}^{n}} \|\mathbf{e}_j - \mathbf{A}\mathbf{s}\|_2\,. \tag{3.17}$$

Eqs. (3.17) are solved iteratively by the minimal residual (MinRes) algorithm [112, Sections 5.3 and 10.5] with initial guess $\mathbf{M}_0^{-1} = \mathbf{A}^T$. We compute all $\mathbf{m}_j$ in parallel by

$$\mathbf{r}_j = \mathbf{e}_j - \mathbf{A}\mathbf{m}_j \tag{3.18}$$

$$\mathbf{p}_j = \mathbf{A}\mathbf{r}_j \tag{3.19}$$

$$\alpha_j = \frac{(\mathbf{r}_j, \mathbf{p}_j)}{(\mathbf{p}_j, \mathbf{p}_j)} \tag{3.20}$$

$$\mathbf{m}_j = \mathbf{m}_j + \alpha_j \mathbf{r}_j \tag{3.21}$$

until convergence. The SpMV products $\mathbf{A}\mathbf{r}_j$ are computed only once per step. The residuals $\mathbf{r}_j$ and search directions $\mathbf{p}_j$ can be computed simultaneously by the corresponding sparse-matrix-sparse-matrix products. Similarly, all α_j can be computed in parallel easily, as there are no dependencies among the scalar products nor among the different α_j. Even though $\mathbf{A}$ and $\mathbf{M}^{-1}$ are sparse, the residual matrix $R \equiv (\mathbf{r}_0, \ldots, \mathbf{r}_{n-1})$ might be not. The same holds for the matrix of search directions $\mathbf{P} \equiv (\mathbf{p}_0, \ldots, \mathbf{p}_{n-1}) = \mathbf{A}\mathbf{R}$. To avoid a huge fill-in we fix the sparsity patterns of both R and P beforehand to be the one of $\mathbf{A}^T$.

3.2.4 Polynomial Preconditioning

Like Krylov methods PPc is based on the fact that for a linear system $\mathbf{Ax} = \mathbf{b}$ with initial solution $\mathbf{x}_0$ and initial residual $\mathbf{r}_0 = \mathbf{b} - \mathbf{Ax}_0$ the residual in the nth iteration step is

$$\mathbf{r}_n \;=\; p_n(\mathbf{A})\mathbf{r}_0 \;=\; [\mathbf{I} - \mathbf{A}q_n(\mathbf{A})]\mathbf{r}_0 \tag{3.22}$$

where $q_n(z)$ is the iteration polynomial of the chosen Krylov method and is at most of degree $n-1$. The identity matrix is denoted by $\mathbf{I}$. By construction the error in the nth iteration step is

$$\mathbf{e}_n \;=\; \mathbf{A}^{-1}\mathbf{b} - \mathbf{x}_n \;=\; p_n(\mathbf{A})\mathbf{e}_0$$

which requires $\|p_n(\mathbf{A})\| < 1$ to achieve convergence. After diagonalization this is equivalent to

$$\|p_n(z)\| < 1, \qquad \forall z \in \sigma(\mathbf{A}),$$

which follows from

$$q_n \;:=\; \operatorname*{arg\,min}_{s \in \mathbb{P}_{n-1}} \max_{\lambda \in \sigma(\mathbf{A})} |1 - \lambda s(\lambda)| \tag{3.23}$$

where $\mathbb{P}_n$ is the space of polynomials of degree up to n. In practice, p_n is rather computed for a compact inclusion set $D \subset \mathbb{C}$ containing $\sigma(\mathbf{A})$ since this does not require an explicit knowledge of $\sigma(\mathbf{A})$. Due to $p_n(0) = 1$ we must exclude 0 from D. The minimization problem (3.23) is equivalent to best approximation on a compact set in $\mathbb{C}$ which is solved by Faber polynomials [134, 135] which are completely determined by the shape of the boundary ∂D of D.

Our PPc algorithm is based on the Arnoldi-Bratwurst-Faber (ABF) method [83, 75] of which we give a detailed outline in chapter 7. The main advantage of ABF-like methods is that ∂D is given as the image of an analytically known conformal map f, the *exterior mapping function* (EMF), yet their shape is not necessarily convex which simplifies to fulfill the restriction $0 \notin D$. Faber polynomials are only defined implicitly, cf. Eq. (7.8), and their computation has to be done by a n-term recursion, cf. Eq. (7.9). If D is an ellipse, the Faber polynomials coincide with the Chebyshev polynomials and can be computed efficiently from a three-term recurrence. This carries over to a non-convex D constructed from the image of an ellipse under a Moebius transformation. The EMF can be cast into the form of a generalized Joukowski map

$$f : \mathbb{C} \to \mathbb{C}, \qquad f(w) \;:=\; \frac{w^2 + \mu_1 w + \mu_0}{\nu_1 w + \nu_0} \tag{3.24}$$

which, following [84], leads to three-term recurrences for the residual polynomial p_n, Eq. (7.30), and the iteration polynomial

$$q_n(z) = (1 - p_n(z))/z\,. \tag{3.25}$$

The derivation of the latter is similar to the one of the former in chapter 7 and hence is skipped. The iteration polynomial is needed for computing the solution of the linear system

$$\mathbf{x}_n \;=\; \mathbf{x}_0 + q_n(z)\mathbf{r}_0 \tag{3.26}$$

in the nth iteration step. Convergence of the method is completely determined by the parameterization of the enumerator of f and is measured by the asymptotic convergence factor

$$R(D) \;=\; \frac{1}{|f^{-1}(0)|} \;=\; \frac{1}{\left|-\frac{\mu_1}{2} - \frac{1}{2}\sqrt{\mu_1^2 - 4\mu_0}\right|}\,. \tag{3.27}$$

Therefore, the crucial step is to compute the four complex constants μ_0, ν_0, μ_1, ν_1 from an estimated shape of the spectrum.

Algorithm - Pseudo code

Here, we collect all relevant equations and give a memory-efficient formulation in pseudo code taken from [83]. In the following $\vec{r}_0$, $\vec{r}_n$, $\vec{x}$, $\vec{x}_0$, $\vec{v}_n$, $\vec{F}_{n-1}$, $\vec{F}_n$, $\vec{G}_{n-1}$, $\vec{G}_n$ indicate arrays of real floating point numbers that have to be stored and f_{n-1}, f_n, ρ_{n-2}, ρ_{n-1}, ρ_n, ν_0, ν_1, μ_0, μ_1 and S are all floating point numbers. The coefficient matrix is A.

- Compute $\sigma(A)$, a small set of dominating eigenvalues from GMRES' Hessenberg matrix
- given $\sigma(A)$, determine ν_0, ν_1, μ_0, μ_1 as described in [83, Secs 3.2 and 3.3].
- Due to the three-term recurrence the initialization phase has two stages. The actual recurrence is given by Eqs. (7.29)-(7.34) and the final solution follows from

```
Stage 0:
    $\rho_{n-2} = 2$
    $S = -\nu_0/\nu_1$
    $\vec{r}_0 = \vec{b} - A\vec{x}_0$
    $f_{n-1} = 2$
    $\vec{F}_{n-1} = 2\vec{r}_0$
    $\vec{G}_{n-1} = \vec{0}$
Stage 1:
    $f_n = -\mu_1$
    $\rho_{n-1} = f_n - S$
    $\vec{F}_n = -\mu_1\vec{r}_0 + \nu_1 A\vec{r}_0$
    $\vec{G}_n = -\nu_1\vec{r}_0$
    $\vec{r}_n = (1/\rho_1)\vec{F}_n - (S/\rho_1)\vec{r}_0$
Recurrence:
    $while\ \|\vec{r}_n\|_2 \geq Tol:$
        $f_n = -\mu_1 f_n - \mu_0 f_{n-1}$
        $\rho_n = f_n - S^n$
        $\vec{v}_n = \nu_1\vec{F}_n + \nu_0\vec{F}_{n-1}$
        $\vec{F}_n = = A\vec{v}_n - \mu_1\vec{F}_n - \mu_0\vec{F}_{n-1}$
        $\vec{G}_n = \frac{-1}{\rho_n}\left(\vec{v}_n + \mu_1\rho_{n-1}\vec{G}_n + \mu_0\rho_{n-2}\vec{G}_{n-1}\right)$
        $\vec{r}_n = (1/\rho_n)\vec{F}_n - (S^n/\rho_n)\vec{r}_0$
        $\rho_{n-1} = \rho_n$
        for $* \in \{f, \rho, \vec{F}, \vec{G}\} : *_{n-1} = *_n$
    $\vec{x} = \vec{x}_0 + \vec{G}_n$
```

In case PPc is used for enhancing SpAI all occurrences of A must be replaced by AM^{-1}.

3.3 Implementation Issues

The SpMV is the most frequently recurring pattern in the numerical linear algebra for PDE solvers and therefore deserves a detailed discussion of how it can be implemented efficiently with CUDA. A general feature of finite-element matrices on arbitrary unstructured meshes is a lack of structure. The common storage formats for sparse matrices like CSR (compressed storage row) induce an indirect and in general random access to the elements of the source vector of the SpMV. For highly data-parallel and throughput oriented architectures like CUDA this is far from optimal as it destroys cache locality and ruins efficient usage of coalesced memory transfers which are crucial for hiding the latency times. In case of PNS this is partly overcome by ordering the solution components per grid point so that in our vector-valued subproblems (Oseen and k-ε) some local structure is induced and matrix elements can be stored as small dense matrices and the vector entries associated with one matrix element are ordered tuples within a single cache line (provided the whole vector is properly aligned).

In case of PNS iterative solvers require only a small set of basic operations like linear combinations of vectors or matrix-vector products thus porting them to CUDA is fairly easy. How to do this follows the same lines of reasoning as the design of the matrix and vector class in chapter 1 and is not further discussed. As already outlined at the beginning, for *deal*.II there is virtually nothing to do as far as the solvers are concerned.

3.3.1 Sparse Matrix-Vector Product

The circumstance that multiplying a matrix element with the corresponding subvector is in fact a product of a small dense matrix and a small dense vector regularizes the memory access pattern and increases data re-usage so that it is useful to store the vector entries in shared memory. Due to this special structure we implemented our own SpMV product which we have based on ideas found in [25, 26] and reach roughly 50-70% of the possible memory bandwidth on a Tesla C2070 which is consistent with literature [25, 26] when solving the Oseen problem, cf. Eq. (5.15). The Oseen problem is a typical representative for PDEs leading to a block-sparse matrix where matrix elements are not just single numbers but rather small dense matrices.

In the following we discuss how to max out CUDA for the Oseen problem which represents the bulk of the work to be done in one time step in the simulation of indoor airflow. To start with, recall the way CUDA performs memory accesses (in the following we assume double precision as this is what counts for solving PDEs):

Within one cycle one half warp can load up to 128 Bytes either from global, shared or local (if it is not too much these are on-chip registers) memory, i.e. 16 `doubles`. Alignment and shared memory bank conflicts decide about the ”up to”. For the latter we recall Sec. 2.1.2.

The difficulty for the SpMV lies in loading the source $\mathbf{x}$ in $\mathbf{y} = \mathbf{A}\mathbf{x}$ as sparsity of $\mathbf{A}$ implies random access to $\mathbf{x}$. On pre-Fermi cards this was partially resolved by using the texture memory which is optimized for random accesses in that sense, that it offers some sort of caching. On the Fermi architecture access to global memory is cached and access to the cache is fast (1 cycle). Listings 3.1-3.4 show the complete and fully functional code of the CUDA implementation of the SpMV. Using the fact that $\mathbf{x}$ is stored in chunks of 4 doubles due to the node-wise ordering and each matrix is a dense 4×4 matrix, i.e. it has 16 entries which exactly fit into one cache line we do the following: Each half warp computes all element MV products for a row of $\mathbf{A}$. For a blocksize of 256 we have 16 half warps. Hence one block processes 16 consecutive rows.

Listing 3.1: SpMV kernel: Declaration

```
template <typename T>
__global__ void
__blanc_spmv_4(const int nRows,
               const int * const rowPtr, const int * const colIndices,
               const T * const nonZeroEntries, const int eltdim,
               const T * const x, T * y)
{
```

We have to buffer the x and the partial sums. This is done by the shared memory arrays:

Listing 3.2: SpMV kernel: Work arrays

```
    // We need two shared arrays. One for the partial sums of y = A*x ...
    __shared__ T y_ps_shm[256 /*BLOCKSIZE*/];
    // and one for the values of the source vector x
    __shared__ T x_shm[64 /*half-warp size * eltdim*/ ];

    // Initialize the shared arrays
    y_ps_shm[threadIdx.x] = 0.;

    if (threadIdx.x < 64)
        x_shm[threadIdx.x] = 0.;
    __syncthreads();
```

For the navigation within a matrix element we need several local variables. With these we determine which entries a thread has to load and process:

Listing 3.3: SpMV kernel: Local variables

```
    // 16 threads, i.e. each half-warp processes a row of 4x4-Matrices
    const int n_entries_per_element = eltdim*eltdim;

    int n_rows_per_block = blockDim.x/n_entries_per_element; // 256/16 = 16

    int local_row   = threadIdx.x/16 /*n_entries_per_element*/; // thread
        0-15 : 0th row, 16-31 : 1st row, 32-47 : 2nd row, ...

    int entry_idx   = threadIdx.x%16 /*n_entries_per_element*/; // local,
        row-wise indexing of entries of a matrix element; range: 0-15

    int x_component     = entry_idx%eltdim;
    int local_entry_row = entry_idx/eltdim;

    // block row
    int row = blockIdx.x * n_rows_per_block + local_row;
    // does row index exceed bounds?
    if(row >= nRows ) return;
    // global row index for entry in y
    int entry_row = eltdim * row + local_entry_row;

    int rowlen = rowPtr[row+1] - rowPtr[row];

    const T* A_i   = &nonZeroEntries[rowPtr[row]*eltdim*eltdim];
```

```
24      int y_ps_pos = threadIdx.x;
```

Listing 3.4: SpMV kernel: The bare algorithm

```
         volatile T * y_ps = y_ps_shm +y_ps_pos ;
 2
         // loop over elements of a row
 4       for(int j = 0; j < rowlen; j++)
         {
 6           int c = colIndices[rowPtr [row]+j ] ;    // global column index

 8           int c_entry = eltdim * c + x_component;    // global Index  of an
                 entry in source vector

10           int c_x_shm = local_row*eltdim + x_component;    // local row

12           // 1. load entries of source vector
             if (entry_idx < eltdim)
14               x_shm[c_x_shm] = x[c_entry];

16           // 2. load entries of element
             T A_ij = A_i[j*n_entries_per_element + entry_idx];
18
             // 3. compute partial sum
20           (*y_ps) += A_ij * x_shm[c_x_shm];
         }
22
         // accumulate parial sums; assumption: eltdim == 4 !!!
24       if (x_component < 2) (*y_ps) += *(y_ps +2);

26       if (x_component ==0) (*y_ps) += *(y_ps +1);

28       if (x_component ==0) y[entry_row] = (*y_ps); // final result written to
              destination vector
     }
```

The keyword `volatile` in line 1 enforces the compiler to execute each read operation for a variable and avoids over-optimization. Within a warp execution is synchronous and within each thread the kernel code is processed strictly in a serial manner. Therefore, we do not have to synchronize after loading the entries of x by the first 4 threads of a warp. Only after they have been loaded the entries of the matrix element are loaded.

3.3.2 Sparse Matrix-Sparse Matrix Residual

The essential operation for assembling a SpAI is the inner product of two sparse vectors whereas its application only requires the standard SpMV product. Due to the ordering of the solution components for vector-valued problems multiplying two vector elements amounts to multiplying small dense matrices. Especially in the 4×4 case this allows for an efficient hardware utilization by assigning each inner product to a half-warp. Then, global memory accesses in loading source and destination elements fully coalesce as they take place as multiples of 128 Byte which is an exact match for the size of the cache lines [8]. Our inner product resembles the one given in [144] except that we have to multiply and add small dense matrices.

Part II

Applications

Chapter 4

Real-Time Stimulation Patterns for Interactive Photostimulation

One of the most successful applications based on the library introduced in the first part is the computation of phase-only holograms. Initially aimed at investigating the feasibility of computing phase masks in real-time with respect to the timescale of the dynamics of neural activity, the project evolved into a GPGPU framework for phase retrieval problems.

A fair comparison of different parallelization techniques requires a unified algorithm execution. This requires a separation of algorithmic details from the details of parallelization. This separation led to the introduction of the concept of a parallel architecture abstraction layer, PAAL.

4.1 Introduction

The development of light-sensitive neurons has been a milestone in optogenetics [38]. The ultimate goal is a non-destructive and fast, yet accurate photo-stimulation of individual sites in networks of living neurons. With respect to energy efficiency and spatial resolution holographic methods are considered to be the most suitable [52] for this task. Holograms, i.e. computer generated phase masks, displayed on a spatial light modulator (SLM) realize pixel-wise phase retardations of a coherent laser beam. Upon illumination the intensity of the Fourier transform of the phase mask yields a high-resolution optical stimulation pattern at the specimen. For a sketch of the experimental setup see Fig. 4.1. The optical stimulation pattern follows from the subset of neurons selected for stimulation. By targeting specific neurons the neural activity in the network and thus its collective behavior can be influenced.

The origin of the neural activity is the generation of spikes in the membrane potential at the axon hillock depending on the synaptic input. The spikes travel along the axon to the synaptic connections to other nerve cells. In genetically altered neurons light-sensitive ion channels are expressed in the cell membrane. If lit with the correct frequency the ion channels may open and thus change the membrane potential which eventually either inhibits or enhances spike generation. After transmission to the next neuron the spike adds to the synaptic input which may result in another spike. For interactive modification of the spiking behavior the optical stimulus must be generated within the time a spike needs to travel from one neuron to the other. Interspike intervals of adjacent neurons are in the range of 10-20 ms and set the time-scale for computing the unknown phase mask. Due to this severe time constraint multi-site stimulation thus had to use precomputed phase masks, up to now. For interactive network control phase

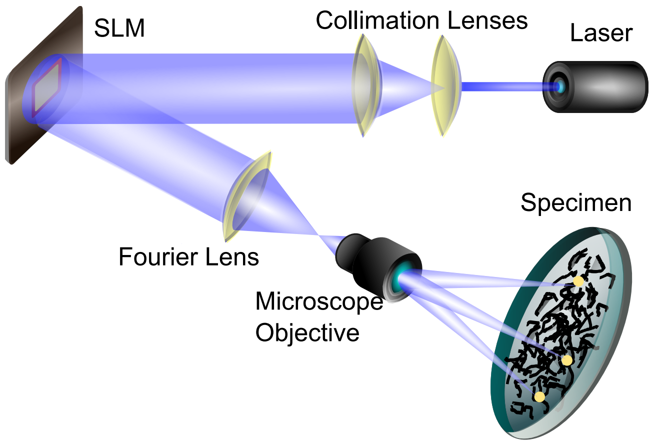


Figure 4.1: Holographic illumination of a network of opto-genetically altered neurons.

masks must be computed online which for frame rates in the required range of 0.1 to 1 kHz poses a substantial challenge. On nowadays many- and multi-core processing units this can only be achieved by extensive parallelization.

Mathematically, computing a phase mask for a given optical stimulation pattern constitutes an inverse problem equivalent to wavefront reconstruction (see [89] and references therein) and is an instance of the phase retrieval problem in diffraction imaging [124]. Numerical approaches to the phase problem abound, but convergence results and global solutions are limited to special cases [58, 23] that do not necessarily apply to the case discussed in this chapter. An arbitrary optical stimulation pattern is unlikely to have a phase-only Fourier transform. Thus our phase retrieval problem is fundamentally *inconsistent* as defined in [90]. To account for the mathematical structure, a careful analysis of the performance of the parallelization techniques available and a strong focus on long-term software reusability distinguishes this work from others, e.g. [93, 119, 139]. Useful approximations of a phase mask for a given optical stimulation pattern can be achieved by iterative algorithms like the widely used *Method of Alternating Projections* [133], also known as Gerchberg-Saxton algorithm [50].

On the next pages we will combine parallel computation on graphics cards with C++-based generic programming and a sound mathematical theory. Only this combination of techniques allows to generate phase masks within 10 ms, matching the dynamics of neural activity.

4.2 Method of Alternating Projections

The wavefront is to be altered by a phase shift at the finite number of pixels of the SLM. The entire system is modeled on a finite dimensional vector space. Let $L_x, L_y > 0$ be the dimensions of the SLM and n_x, n_y the respective number of pixels. We seek a signal $u \in \mathbb{C}^N$ for $N \equiv n_x \times n_y$. The intensity distribution of the laser beam sets the amplitude of the wavefront u on the SLM. Assuming a constant intensity over one pixel, we discretize the intensity distribution by the nonnegative $p \in \mathbb{R}^N$. Wavefronts u emanating from the SLM are given by the set of vectors with fixed amplitudes

$$S \quad \equiv \quad \{u \in \mathbb{C}^N \; : \; |u_{jk}| = p_{jk}, \quad j = 1, 2, \ldots, n_x, \; k = 1, 2, \ldots, n_y\}. \tag{4.1}$$

Propagation of the light through the lens system is modeled by Fraunhofer diffraction [54]. The light at the SLM is related to the observed optical stimulation pattern at the specimen by the Fourier transform F. Waves with modulus matching the amplitude distribution $m \in \mathbb{R}^N$ of the optical stimulation pattern form the set

$$M \equiv \{u \in \mathbb{C}^N \ : \ |(Fu)_{jk}| = m_{jk}, \quad j \leq n_x,\ k \leq n_y\}. \tag{4.2}$$

On the one hand our wavefront must fulfill the amplitude constraint Eq. (4.1), on the other hand the amplitudes of its Fourier transform are fixed by the intensity distribution of the stimulation pattern, Eq. (4.2). Altogether, the mathematical problem we address is to

$$\text{Find } u \in S \cap M. \tag{4.3}$$

For a nonempty intersection the problem is defined to be *consistent*; otherwise it is said to be *inconsistent* or *ill-posed*. A common algorithm for problems of this type is the method of alternating projections [133, 50]. For a review of this and other projection-based approaches for the phase retrieval problem see [23]. Algorithms of this kind are built on *projection operators* onto the constraint sets S and M. By a *projection* of a point u in a space X onto a subset C of that space, we mean the mapping of that point to the set of nearest points in C with respect to the norm induced by the real inner product on X. For general phase retrieval problems, it was proved in [89] that

$$P_S u \ = \ \left\{ v \ : \ v_{jk} = \begin{cases} p_{jk} \frac{u_{jk}}{|u_{jk}|}, & \text{if } u_{jk} \neq 0; \\ p_{jk} \exp(i\theta), & \text{for } \theta \in [0, 2\pi) \end{cases} \right\}, \tag{4.4a}$$

$$P_M u \ = \ \left\{ F^{-1}\widehat{u} \ : \ \widehat{u} \in \widehat{M}(u) \right\} \tag{4.4b}$$

are projections onto the sets S and M, respectively, where

$$\widehat{M}(u) \equiv \left\{ \hat{v} \ : \ \hat{v}_{jk} = \begin{cases} m_{jk} \frac{(Fu)_{jk}}{|(Fu)_{jk}|}, & \text{if } (Fu)_{jk} \neq 0 \\ m_{jk} \exp(i\theta), & \text{for } \theta \in [0, 2\pi) \end{cases} \right\}. \tag{4.4c}$$

For given $u^0 \in \mathbb{C}^N$ the method of alternating projections computes the iterates u^ν via

$$u^{\nu+1} \in P_S P_M u^\nu, \quad \nu = 0, 1, 2, \ldots \tag{4.5}$$

The multi-valuedness of Eq. (4.4) makes Eq. (4.3) a *non-convex* feasibility problem [89]. Hence Eq. (4.5) must be understood as a selection from set-valued mappings. Due to nonconvexity, except in special cases [58], global convergence of Eq. (4.5) cannot be guaranteed in general. For consistent phase retrieval problems local convergence results are available [90]. Yet, it is more the exception than the rule that our phase retrieval problem will be consistent: a set of fixed amplitudes cannot produce an arbitrary optical stimulation pattern. Our tests indicate that our phase retrieval problems were indeed inconsistent as measured by the magnitude of the *gap*

$$G \ \equiv \ \|P_S u^\nu - P_M u^\nu\|_2 \tag{4.6}$$

between accumulation points in M and their projections onto S. The gap is measured in the standard Euclidean norm $\|\cdot\|_2$. This systematic inconsistency is a major difference between optogenetic photo-stimulation and phase retrieval problems due to imaging experiments. In the latter the diffraction pattern comprising the set M is *causal*, that is, comes from diffraction by a physical object, for instance a protein crystal. Assuming that Eq. (4.3) is inconsistent, we content ourselves with finding *best approximation pairs* $(u^*, v^*) \in \mathbb{C}^N \times \mathbb{C}^N$ with $u^* \in S$, $v^* \in M$, $P_M u^* = v^*$ and $P_S v^* = u^*$.

To account for the particularities of optogenetic photo-stimulation, we define the physical error as the sum of the pixel-wise relative violation of deviation tolerances between target and reconstructed optical stimulation pattern [116]. We allow for a relative deviation $t_\ell = 0.1$ for non-zero target pixels and an absolute deviation of $t_\text{d} = 3 \cdot 10^{-4}$ for non-lit pixels. Violations are summed in multiples of t_ℓ and t_d. With $a \equiv u^v_{jk}$ being the intensity from the current iteration step, $A \equiv m_{jk}$ the intensity in the target optical stimulation pattern and $\Theta(\cdot)$ the Heaviside step function, the total error is the sum of

$$E_\ell = \sum_{pixels \in light} \left(\frac{t_\text{d}|A-a|}{t_\ell A} - t_\text{d} \right) \cdot \Theta \left(\frac{|A-a|}{A} - t_\ell \right), \tag{4.7a}$$

$$E_\text{d} = \sum_{pixels \in dark} (a - t_\text{d}) \cdot \Theta (a - t_\text{d}) \, . \tag{4.7b}$$

4.3 Unified Implementation

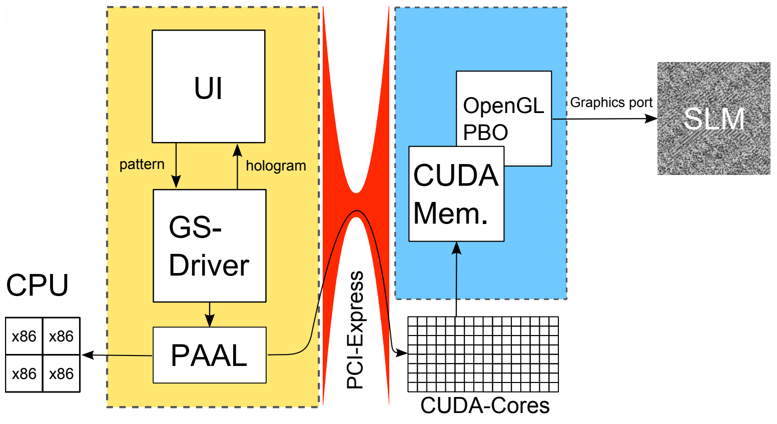


Figure 4.2: Software structure and its association with hardware components.

The compound system of CPU and GPU with dedicated memories represents a non-uniform memory access (NUMA) architecture with a very heterogeneous distribution of processing capabilities and internal transfer rates. Figure 4.2 shows a schematic of the class structure and its distribution over the compound system of CPU and GPU. In general, the major bottleneck is the PCIe-BUS which according to the PCIe v2.0 specification has a maximal transfer rate of 8 GByte/sec although in practive one rather gets 4 to 5 GByte/sec. This will rise to 16 GByte/sec with the forthcoming PCIe v3.0, yet memory transfer rates within the GPU is of the order of 100 GByte/sec. On CPUs with integrated memory controllers the memory transfer rates are in the range of 20 to 30 GByte/sec.

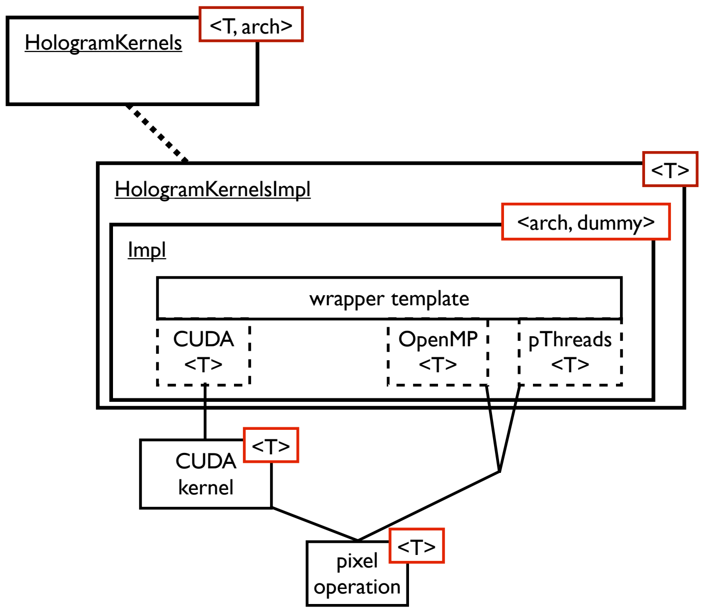


Figure 4.3: Class diagram of the implementation of the PAAL concept. Class names are underlined. Template arguments are given in the red boxes. Dashed boxes indicate the partial template specializations.

To anticipate the rapid evolution of hardware and parallelization techniques we spent considerable effort on modularizing the program using C++'s templating capabilities. CUDA extends *C* for programming NVidia GPUs. OpenMP mainly supports multithreaded parallelization on multi-core CPUs. Depending on the parallelization paradigm the program works on different sides of the PCIe-BUS. To achieve hardware-specific optimizations at the low-level, e.g. for the details of Eq. (4.4), yet keeping the implementation of algorithms generic we introduced the concept of a parallel architecture abstraction layer (PAAL). Since most of our computational tasks are data-parallel they are perfect candidates for abstraction with respect to floating-point precision and parallelization technique. This is easily achieved by a suitable set of template parameters leaving the generation of the hardware-specific part of the code to the compiler. As FFTs in the implementation of the projector onto the constraint due to the optical stimulation pattern, Eq. (4.4b), we use either FFTW [47] or NVidia's cuFFT.

The aim of the PAAL concept is a quick recombination of algorithms and parallelization strategies by explicit template specializations. The front end comprises the user-interface (UI) and manages the execution of the phase retrieval algorithms for which separate driver classes exist, e.g. GS-DRIVER for the method of alternating projections. The final phase mask is transferred to an OpenGL framebuffer object for display on the SLM, cf. Fig. 4.2.

The PAAL concept is explained best by walking through the essential code snippets. This is done roughly in a top down approach, i.e. from host to device and how things build on each other. A sketch of the class structure is given in Fig. 4.3. At the top is the interface to the GS-Driver which is formed by the class `HologramKernels`, given in listing 4.1. It takes two template arguments: `T` for the precision and `arch` for the architecture the algorithm is to run on. At the bottom of the hierarchy is the operation one has to do on a particular pixel, cf. listing 4.5.

PAAL Header File

To express that the class `HologramKernels` is *implemented with* `HologramKernelsImpl` inheritance is private [96] (indicted by the dashed line in Fig. 4.3) and listing 4.1.

Listing 4.1: PAAL header: front end class: HologramKernels

```
template<typename T,
         ParallelArch arch>
struct HologramKernels
        :
        HologramKernelsImpl<T>::template Impl<arch, T>
{};
```

The class `HologramKernels` needs partial specializations (given in listing 4.2) for the different architectures because for the CUDA kernels the wrapper functions behave differently with respect to the architecture. On a NVidia GPU they have to call a CUDA kernel. On a CPU they have to either use OpenMP or pthreads for parallelization. The parallel execution of the per-pixel operation `__ps_element()` (cf. listing 4.5) via OpenMP or pthreads can be done directly in the specialization of the wrapper function, cf. listing 4.8. In the following we omit the pthreads specialization as it is structurally very similar to OpenMP.

Listing 4.2: PAAL header: specialized front end classes.

```
template<typename T>
struct HologramKernels<T, gpu_cuda>
        :
        HologramKernelsImpl<T>::template Impl<gpu_cuda, T>
{
    HologramKernels(int /*num_omp_threads*/) {}
};


template<typename T>
struct HologramKernels<T,cpu>
        :
        HologramKernelsImpl<T>::template Impl<cpu, T>
{
    HologramKernels(int num_omp_threads)
    {
        omp_set_num_threads(num_omp_threads);
    }
};

#endif // CUDA_KERNEL_STEP_16_CU_H
```

The wrapper functions are implemented by the internal class `Impl` of `HologramKernelsImpl`. The reason for this awkward design is that the *C*++ standard does not allow to define partial specializations of (a subset of) the member functions of a class. This issue can be circumvented by introducing an internal class with a dummy template parameter and to partially specialize its members. Within the class `Impl` the particular type of real and complex numbers is deduced from the template parameter `T` by means of the `PrecisionTraits` structure. This is a typical example of template metaprogramming [9]. See the following listing.

Listing 4.3: PAAL header: class: HologramKernelsImpl

```
template<typename T>
struct HologramKernelsImpl {

template <ParallelArch arch, typename dummy > class Impl;

template <typename dummy> class Impl<gpu_cuda, dummy> {
public:
    typedef typename PrecisionTraits<T, gpu_cuda>::CudaComplex Complex;
    typedef typename PrecisionTraits<T, gpu_cuda>::NumberType  Number;

    void ps(Complex *d_devPtr, const Complex *d_original, int size);
    // omitted: rescaling, RAAR, randomization, error estimation
};

template <typename dummy> class Impl<cpu, dummy> {
    public:
        // omitted: typedefs for Complex and Number

        // same functions as in the CUDA-specific version of this class
        void ps(...);
    };
};
```

PAAL Source File

The backend, i.e. the details of the implementation, are stored in a separate source file to keep g++ away from CUDA-specific code. Within the evaluation of Eqs. (4.4) and (4.4c) we need precision-dependent tolerances for what is considered as zero. To this end we localize the inevitable magic numbers in a structure `__eps` and a function `__is_zero`. In case of being compiled into a CUDA kernel the `__device__` keyword is put into effect indicating that the function can only be executed on the device, i.e. the GPU. Prepending `__device__` by `__host__` signals the compiler (i.e. nvcc) to compile two versions of a function or operator. One for the execution on the GPU and one for the CPU. At the binary level these are distinct functions.

Listing 4.4: Parallel architecture abstraction layer - Backend (Kernels, etc ...)

```
template <typename T> struct __eps { T operator()(); };

template<> __host__ __device__
double __eps<double>::operator()() { return 1e-8; }

template<> __host__ __device__
float __eps<float>::operator()() { return 1e-4; }


template <typename T>
__host__ __device__ bool __is_zero(T x);

__host__ __device__ bool __is_zero(double x) { return abs(x) < 1e-16; }

__host__ __device__ bool __is_zero(float x) { return abs(x) < 1e-8; }
```

The actual work is done by an architecture-independent function `__ps_element`, cf. listing 4.5. Its arguments are a pointer `d_devPtr*` to the beginning of the array of pixels of the iterated image Fu^{ν}, a pointer `d_original*` to the beginning of the array of pixels of the original image and the lexicographic index of the pixel `x`.

Listing 4.5: PAAL: Architecture-independent per-pixel operation of P_S, Eq. (4.4a).

```
template <typename T, ParallelArch arch>
__host__
__device__
void
__ps_element(T *d_devPtr, const T *d_original, int x)
{
    typedef typename PrecisionTraits<T, arch>::NumberType Number;
    __eps<Number> eps;
    Number eps2 = eps()*eps();

    //Copy pixel values from global memory to local memory
    // to avoid multiple reads of the same value.
    Number real = d_devPtr[x].x;
    Number imag = d_devPtr[x].y;

    //Calculate modulus of pixel (complex)value
    Number abs = sqrt(real*real+imag*imag);
    Number abs2 = abs*abs;
    //original value
    Number original = d_original[x].x;

    //if original image is zero, then reconstructed image is zero, too
    if(original < 2*eps())
    {
        d_devPtr[x].x = 0;
        d_devPtr[x].y = 0;
    }
    else
    {
        //rescaling of the complex values
        if( !__is_zero(abs)) {
            real = (real/abs)*original;
            imag = (imag/abs)*original;
        }

        //write result back to global memory.
        d_devPtr[x].x = real;
        d_devPtr[x].y = imag;
    }
}
```

The CUDA kernel basically has the same arguments as the element function. The only difference is, that the kernel gets passed the size of the image as third argument in order to avoid operating on non-existent pixels. The kernel computes the position `x` of its pixel from its `threadIdx` and `BlockIdx` (line 5 in listing 4.6) introduced in Sec. 2.1.2. Both arrays are one-dimensional reflecting the storage of the images as linear arrays of pixels. Given the pixel position the element function is invoked.

Listing 4.6: PAAL: CUDA kernel for amplitude adaption

```
template <typename T>
__global__ void __ps(T *d_devPtr, const T *d_original, int size)
{
    //Calculate the thread ID. The thread ID determines the pixel.
    int x = blockDim.x*blockIdx.x+threadIdx.x;

    //Prevents kernel to calculate something outside the image vector.
    if(x<size)
     __ps_element<T,gpu_cuda>(d_devPtr,d_original,x);
}
```

The missing link between back end and driver class is provided by the specializations of the wrapper functions for the kernels. The GPU version (listing 4.7) starts as many threads as there are pixels for the kernel `__ps`.

The CPU-OpenMP version (listing 4.8) has a for-loop over all pixels in the image which is parallelized by an OpenMP preprocessor directive. Since we defined the `__ps_element` to be a `__host__ __device__` function, we can call the same function from the CPU as from the GPU but without the intermediate kernel layer. In this way we have the actual computation implemented only once. With the individual specialized classes wrapped around this implementation we can choose our computing precision and hardware.

Listing 4.7: PAAL: GPU specialization of wrapper function

```
template<typename T>
template< typename dummy>
void
HologramKernelsImpl<T>::Impl<gpu_cuda, dummy>::ps
(Complex *d_devPtr, const Complex *d_original, int size)
{
#if __CUDA_ARCH__ < 200
    int threads_per_block = 512;
#else
    int threads_per_block = 1024;
#endif
    int blocks = (size + threads_per_block - 1) / threads_per_block;

    __ps<T><<<blocks,threads_per_block>>>(d_devPtr, d_original, size);
    cudaThreadSynchronize();
}
```

Listing 4.8: PAAL: CPU specialization of wrapper function

```
template<typename T>
template< typename dummy>
void
HologramKernelsImpl<T>::Impl<cpu, dummy>::ps
(Complex *d_devPtr, const Complex *d_original, int size)
{
#pragma omp parallel for private(i)
for(int i = 0;i < size;i++)
    __ps_element<T, cpu>(d_devPtr, d_original, i);
}
```

Finally, we have to provide full template specializations of all the combinations of precision and architecture template parameters we want to work with. This must be at the end of file as all functions have to be declared and their bodies defined before the class can be explicitly instantiated by the compiler [129]. The explicit specializations are necessary, at least for CUDA, since we compile the back end with a different compiler than the front end of the program.

Listing 4.9: PAAL: Explicit template specializations

```
template class HologramKernels<float2, gpu_cuda>;
template class HologramKernelsImpl<float2>::Impl<gpu_cuda, float2>;

template class HologramKernels<float2, cpu_openmp>;
template class HologramKernelsImpl<float2>::Impl<cpu_openmp, float2>;

template class HologramKernels<float2, cpu_pthreads>;
template class HologramKernelsImpl<float2>::Impl<cpu_pthreads, float2>;

    // and the same again for double2
```

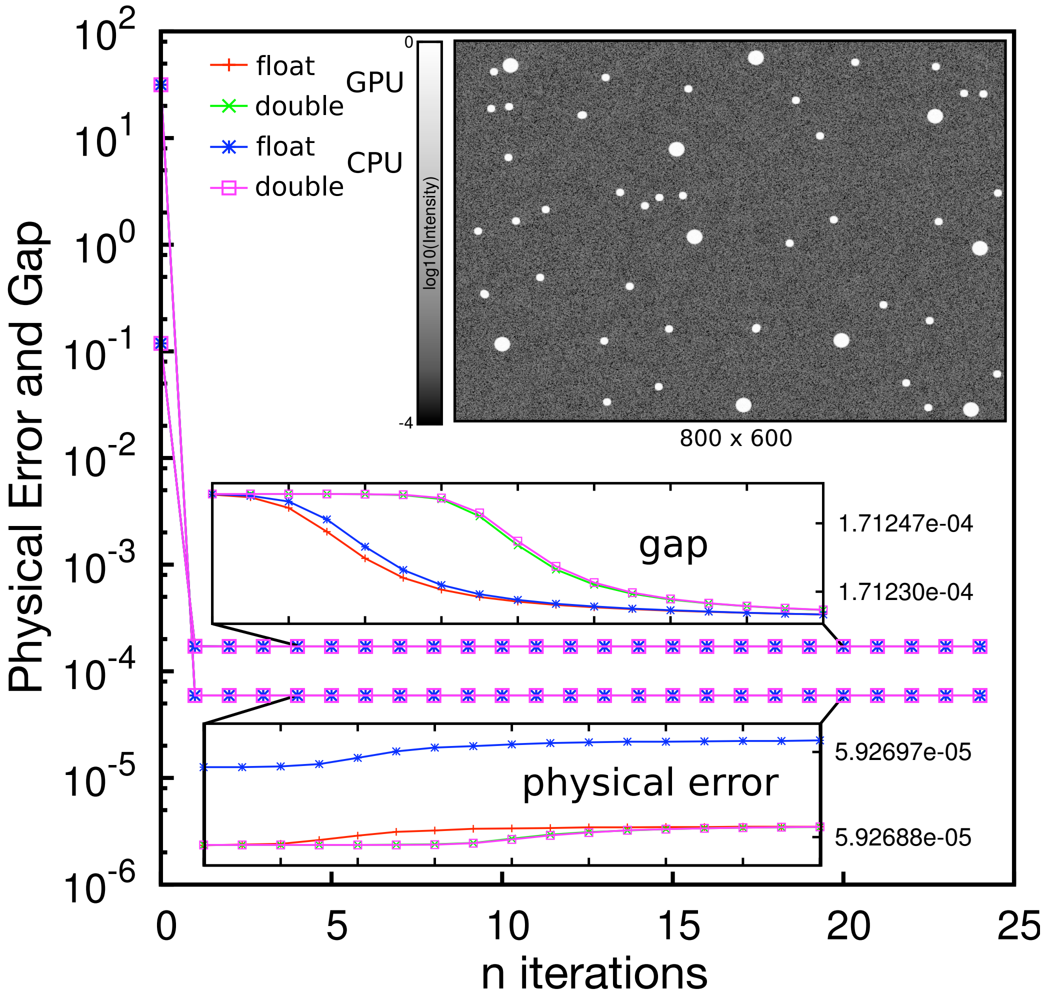


Figure 4.4: Convergence for a spot pattern of the physical error and the constraint gap.

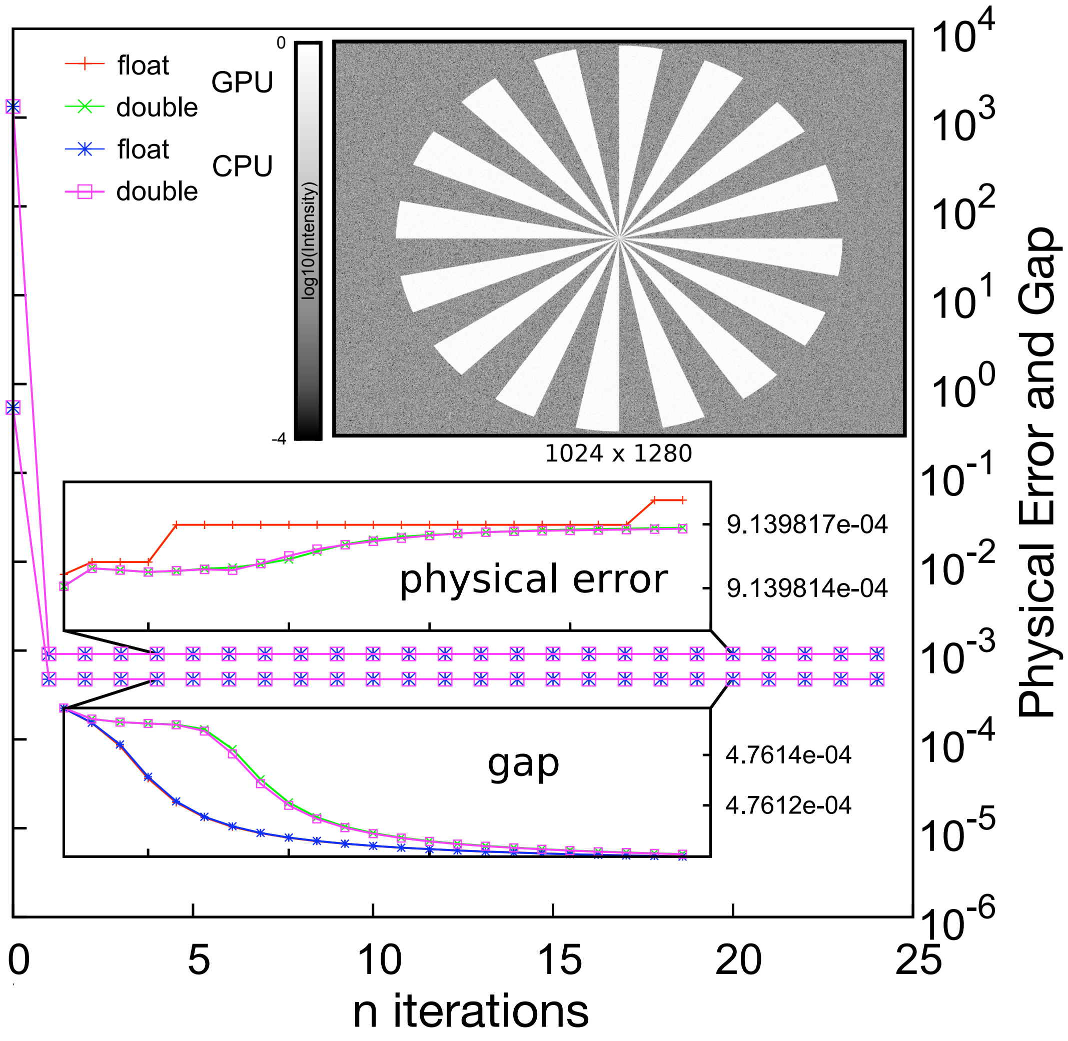


Figure 4.5: Siemens star: Convergence for single and double precision.

4.4 Results and Conclusion

The optical stimulation pattern for benchmarking the computation of the phase masks for photo-stimulation are bright spots on a dark background (Fig. 4.4). The limits of spatial resolution in the reconstruction is tested on the Siemens star (Fig. 4.5).

In both cases we use $P_M \hat{u}^0$ as initial phase mask where u^0 is the 1-bit target optical stimulation pattern. Thus, our initial condition is computed by taking the Fourier transform of the targeted pattern, adapting the Fourier coefficients to the amplitude constraints on the SLM and transforming back into real space again.

4.4.1 Benchmarking

For interactive holographic photostimulation the physical error, Eq. (4.7), must reach a subthreshold level within interspike intervals, i.e. 10 to 20 ms. Hence, the first issue is which parallelization technique provides sufficient performance to meet this requirement. The GPU-based computations were done on a Tesla C2070. The CPU-based ones using OpenMP or pthreads were run on a two-socket system with X5650 Xeons. The CPUs have six cores each and support hyperthreading which not in all cases pays off for numerical computations. Therefore we decided to use 12 threads, i.e. as much as there are physical cores.

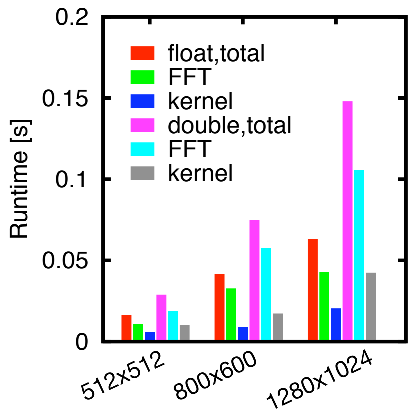


Figure 4.6: GPU runtimes for various image sizes and 25 iterations.

Computing a single phase mask on the CPU requires several seconds no matter whether OpenMP or pthreads are used. When using CUDA and thus moving to the GPU the total runtimes basically match the interspike interval constraint. Fig. 4.6 shows GPU runtimes in total and broken down into the contributions due to FFT (green and cyan bars) and enforcement of amplitude constraints (blue and grey bars) for different image sizes and 25 iterations of Eq. (4.5). The runtimes are further subdivided into the results for single and double precision indicated by the red and magenta bars, respectively.

For the resolution of our SLM (800×600) computing 25 iterations in single precision takes 45 ms including transfer of the given targeted optical stimulation pattern to the GPU (1 ms). The iteration essentially converges after one step (Fig. 4.4) so that a reasonable optical stimulation pattern is available already after less than 10 ms. The precise number depends on the problem size and how many iteration steps are actually done. The proportion of work to be done in the FFT increases with probem size as the FFT is of log-linear complexity whereas enforcing the amplitude constraints is linear in the problem size as each pixel is visited only once per iteration and exchange of information between different pixels is not required.

Figure 4.7 shows the speedups of the CUDA implementation over its OpenMP and pthreads counterparts. On average CUDA is 50 times faster per iteration than the 12-thread CPU variants. The performance gain per iteration solely depends on the size of an image and not on its content. For the Fermi architecture used in the Tesla C2070 the floating point performance in double precision is half of the one for single precision. Basically, this can be attributed to the fact that a `double` is twice as large as a `float` and thus requires twice as much memory bandwidth to reach the same performance. On CPUs this is less of an issue due to the different architecture which is reflected by the fact that for double precision the speed-up roughly is only half of the one for single precision. Yet, this is still sufficient to compute optical stimulation patterns even in double precision within the time limits set by the interspike intervals.

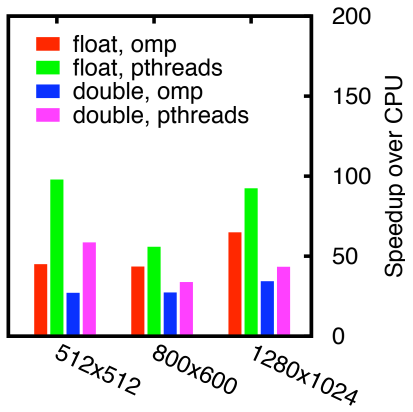


Figure 4.7: Speedup per iteration for various image sizes.

4.4.2 Precision and Convergence

The second issue is the influence of the floating point precision ε on convergence and performance. Figure 4.4 shows the convergence and reconstruction results for a sample spot pattern as it would be used in a photostimulation experiment. Figure 4.5 summarizes the results for the well-known Siemens star which is a standard test image for the spatial resolution achieved by reconstruction algorithms.

The reconstructed images are given as inset with a logarithmic gray scale for intensity. The convergence curves represent the behavior of the physical error, Eq. (4.7), and the gap, Eq. (4.6), with respect to the number of iterated steps in Eq. (4.5). We are interested in the influence of the hardware architecture and the precision. Hence the convergence history is given for single and double precision on GPU and CPU. The details of the curves for the gap and for the physical error in the insets reveal that the behavior primarily depends on whether the computation is run in single or double precision but not on the architecture. This is a subtle effect on the order of the single precision accuracy as illustrated by the scaling of the ordinates in the insets.

Both figures show that convergence of the phase mask in single is as good as in double precision as each quantity has a unique limit value independent of the precision.

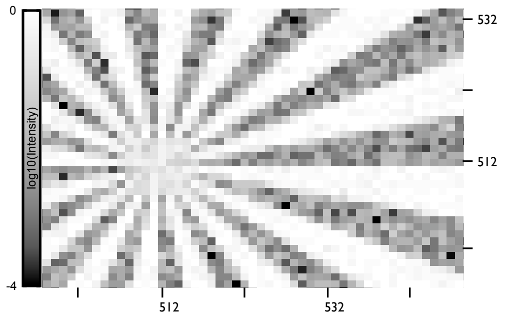


Figure 4.8: Resolution limits revealed by Siemens star.

The intensity plots of the reconstructed patterns indicate that the contrast between lit and unlit areas is approximately 3 orders of magnitude. The close-up view on the center of the Siemens star in Fig. 4.8 points out that structures down to a few pixels can be resolved. The insets of both figures 4.4 and 4.5 demonstrate that the gap, as defined in Eq. (4.6), and the physically motivated error, Eq. (4.7), saturate within one iteration indicating the inconsistency of our phase retrieval problem. All later changes are $\mathcal{O}(\varepsilon)$. For practical purposes convergence does not depend on hardware as the limit values of error and gap are several orders of magnitude larger than any precision. Depending on ε we expect the following resolution limits for G. Our number of pixels is of the order of 10^6. Assuming statistical independence for the errors of u^{ν}_{jk} we get as theoretical limit $G_{th} \propto 10^3\varepsilon$, i.e. 10^{-5} for single precision ($\varepsilon = 10^{-8}$) and 10^{-13} for double precision ($\varepsilon = 10^{-16}$). An interesting phenomenon reflecting the difference between exact and finite precision arithmetic is revealed by comparing the convergence behavior as function of ε. Single precision (blue and red curves) cannot resolve the inconsistency of the phase retrieval problem, i.e. whether or not $S \cap M = \emptyset$, as $G \approx G_{th}$. According to [90] this should improve convergence. The downside is, that while the phase retrieval problem appears to be consistent from a numerical point of view, larger ε means worse approximation of the projection operators. For double precision (green and magenta curves) we get more accurate projection operators but at the same time the gap is resolved (since for both precisions G is of similar magnitude). This renders the phase retrieval problem inconsistent again, justifying our assumption of inconsistency. Our results also indicate that, despite a rather large G the method of alternating projections does not suffer from stagnation at bad local minima which otherwise would call for more sophisticated algorithms like RAAR [88].

Chapter 5

Preconditioning for Indoor Airflow

In the following the results concerning the acceleration of PNS (the Parallel Navier-Stokes Solver) *[105, 73] by porting its linear algebra to CUDA are presented. PNS solves the Reynolds-averaged Navier-Stokes equations for simulating indoor airflow and is coupled to the TRNSYS package [104] for building simulations. The accurate numerical prediction of indoor-air flows for building configurations of practical relevance [87] is of paramount importance for the energy-efficient design of modern buildings.*

Matrices from such a real-life application should be perfect for testing whether NVidias marketing division has made promisses which at the end are hard to keep.

The results for the sparse approximate inverse preconditioner have been published as part of [77].

5.1 Bouyancy Driven Fluid Flow

A possible basis of indoor airflow simulations are the non-dimensional incompressible, non-isothermal Reynolds-Averaged Navier-Stokes (RANS) equations coupled to the k-ε turbulence model [98]. A detailed account of the RANS model used in PNS can be found in [74, 87] which also covers more general boundary conditions and the details of the wall function method.

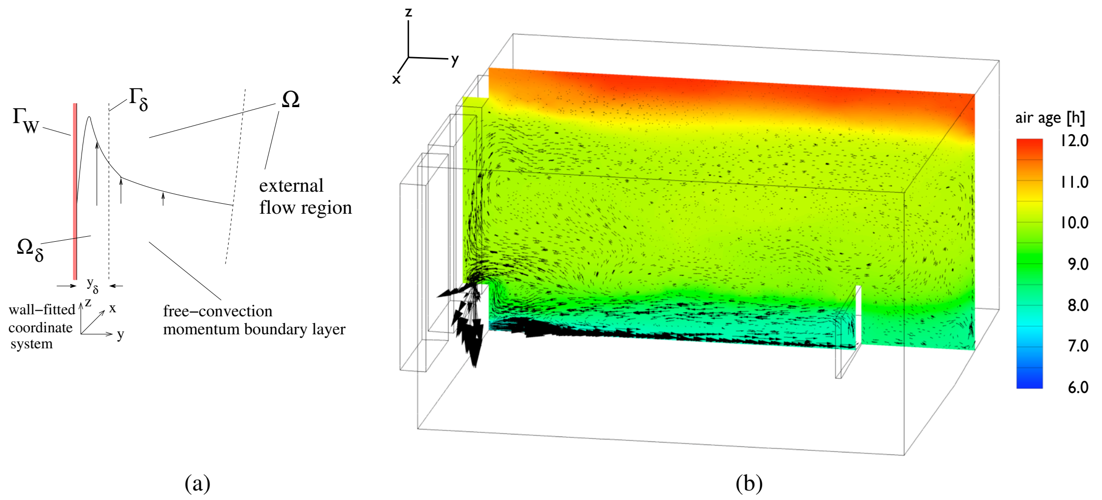


Figure 5.1: (a) Separate treatment of boundary layer Ω_δ. (b) A snapshot of the flow field and distribution of air age (with permission by R. Gritzki). The three boxes at the left are windows.

A decent parallelization of a RANS simulation tool box capable of meeting the challenges of real-life engineering problems is a topic of its own, see [86] and T. Knopp et al. in [91] for an overview. In this chapter we contend ourselves with using this existing RANS implementation as a block-box generator for unstructured, large sparse matrices.

5.1.1 Navier-Stokes Equations

To model the bare flow in a bounded domain $\Omega \subset \mathbb{R}^3$ with boundary $\partial\Omega$ one seeks velocity $\mathbf{u}$, pressure p and temperature θ solving

$$\begin{aligned} \partial_t \mathbf{u} - \nabla \cdot (2\nu_e \mathbb{S}(\mathbf{u})) + (\mathbf{u}\cdot\nabla)\mathbf{u} + \nabla p &= -\beta\theta\mathbf{g}, \\ \nabla\cdot\mathbf{u} &= 0, \\ \partial_t\theta + (\mathbf{u}\cdot\nabla)\theta - \nabla\cdot(a_e\nabla\theta) &= c_p^{-1}\dot{q}^V \end{aligned} \tag{5.1}$$

with the rate of strain tensor

$$\mathbb{S}(\mathbf{u}) := (\nabla\mathbf{u} + \nabla\mathbf{u}^T)/2,$$

isobaric volume expansion coefficient β, gravitational acceleration $\mathbf{g}$, volumetric heat source $\dot{q}^V$ and isobaric specific heat capacity c_p. Buoyancy forces are modeled by the Boussinesq approximation. The turbulence model requires the introduction of the effective viscosities

$$\begin{aligned} \nu_e &= \nu + \nu_t, \\ a_e &= a + a_t \end{aligned}$$

with kinematic viscosity ν, turbulent viscosity ν_t, thermal diffusivity

$$a = \nu/Pr$$

and turbulent thermal diffusivity

$$a_t = \nu_t/Pr_t$$

with Prandtl numbers $Pr = 0.7$ and $Pr_t = 0.9$ for air. The non-constant ν_t and a_t reflect turbulent effects and depend on the turbulence model. The sign of $\mathbf{u}\cdot\mathbf{n}$, $\mathbf{n}$ being the outer normal, rules the division of $\partial\Omega$ into wall zones Γ_W, inlet zones Γ_- and outlet zones Γ_+ where we impose

$$\sigma(\mathbf{u},p)\mathbf{n} \equiv 2\nu_e\mathbb{S}(\mathbf{u}) - p\mathbb{I} = \tau_n\mathbf{n} \quad \text{on } \Gamma_- \cup \Gamma_+, \tag{5.2}$$

$$\mathbf{u} = \mathbf{0} \quad \text{on } \Gamma_W, \tag{5.3}$$

with $\sigma(\mathbf{u},p) = 2\nu_e\mathbb{S}(\mathbf{u}) - p\mathbb{I}$. For θ we require

$$\theta = \theta_{in} \quad \text{on } \Gamma_-, \tag{5.4}$$

$$a_e\nabla\theta\cdot\mathbf{n} = 0 \quad \text{on } \Gamma_+, \tag{5.5}$$

$$\theta = \theta_w \quad \text{on } \Gamma_W. \tag{5.6}$$

The in- and outflow conditions in Eqs. (5.2) and (5.4) are suitable for natural ventilation. Near Γ_W, $\mathbf{u}$ and θ exhibit strong gradients. Fig. 5.1a shows a typical near-wall profile for the streamwise component of $\mathbf{u}$ for the flow along a heated vertical wall.

5.1.2 Discretization in Time

For simplicity, time discretization is performed with the BDF(1) scheme. This leads to a sequence of coupled nonlinear problems within each time step. The global problem in Ω reads

$$\begin{aligned}
-\nabla\cdot(2\nu_e\mathbb{S}(\mathbf{u}))+(\mathbf{u}\cdot\nabla)\mathbf{u}+\frac{1}{\delta t}\mathbf{u}+\nabla p &= -\beta\theta\mathbf{g}+\frac{1}{\delta t}\mathbf{u}_{old}\,, &(5.7)\\
\nabla\cdot\mathbf{u} &= 0 &(5.8)\\
-\nabla\cdot(a_e\nabla\theta)+(\mathbf{u}\cdot\nabla)\theta+\frac{1}{\delta t}\theta &= c_p^{-1}\dot{q}^V+\frac{1}{\delta t}\theta_{old} &(5.9)
\end{aligned}$$

with modified boundary conditions on Γ_W

$$\begin{aligned}
\mathbf{u}\cdot\mathbf{n} &= 0\,, &(5.10)\\
(\mathbb{I}-\mathbf{n}\otimes\mathbf{n})\sigma(\mathbf{u},p)\mathbf{n} &= \tau_t(\mathbf{u},\mathbf{u}^L,\theta^L)\,, &(5.11)\\
a_e\nabla\theta\cdot\mathbf{n} &= c_p^{-1}\dot{q}(\mathbf{u}^L,\theta^L). &(5.12)
\end{aligned}$$

Boundary data τ_t, $\dot{q}$ at Γ_W are taken from the solution $(\mathbf{u}^L,p^L,\theta^L)$ of Eqs. (5.7) in the boundary layer $\Omega_\delta := \{x\in\Omega : \mathrm{dist}(x,\Gamma_W)<y_\delta\}$ with a modified wall-function approach, thus avoiding locally fine meshes and expensive anisotropic grid refinement in Ω_δ, cf. Fig. 5.1a.

5.1.3 Turbulence Model

As global turbulence model in Ω a standard one- or two-equation model suffices like the k-ε turbulence model [98]. A reasonable parametrization for indoor airflow is

$$\begin{aligned}
\nu_t &= c_\mu k^2/\varepsilon\,,\\
c_\mu &= 0.09\,.
\end{aligned}$$

The turbulent kinetic energy k and dissipation ε are semidiscrete solutions of

$$\begin{aligned}
-\nabla\cdot(\nu_k\nabla k)+(\mathbf{u}\cdot\nabla)k+\frac{1}{\delta t}k &= P_k+G-\varepsilon+\frac{1}{\delta t}k_{old}\,, &(5.13)\\
-\nabla\cdot(\nu_\varepsilon\nabla\varepsilon)+(\mathbf{u}\cdot\nabla)\varepsilon+\frac{1}{\delta t}\varepsilon+C_2\frac{\varepsilon^2}{k} &= C_1\frac{\varepsilon^2}{k}(P_k+G)+\frac{1}{\delta t}\varepsilon_{old} &(5.14)
\end{aligned}$$

with effective viscosities

$$\begin{aligned}
\nu_k &= \nu+\nu_t/Pr_k\,,\\
\nu_\varepsilon &= \nu+\nu_t/Pr_\varepsilon\,,
\end{aligned}$$

production and buoyancy terms

$$\begin{aligned}
P_k &= 2\nu_t|\mathbb{S}(\mathbf{u})|^2\,,\\
G &= \beta a_t\mathbf{g}\cdot\nabla\theta
\end{aligned}$$

and empirical constants

$$\begin{aligned}
C_1 &= 1.44\,, \qquad Pr_k = 1.0\,,\\
C_2 &= 1.92\,, \qquad Pr_\varepsilon = 1.3.
\end{aligned}$$

The k-ε Eqs. (5.13), (5.14) are solved in $\Omega\backslash\Omega_\delta$ with suitable boundary conditions for k,ε on Γ_δ.

5.1.4 Full Discretization

For the full discretization the model is decoupled and linearized within each time step. Two basic problems are to be solved:

i) An Oseen problem with variable viscosity ν and positive reaction term

$$\begin{aligned} -\nabla\cdot(2\nu\mathbb{S}(\mathbf{u}))+(\mathbf{a}\cdot\nabla)\mathbf{u}+c\mathbf{u}+\nabla p &= \mathbf{f} && \text{in } \Omega, \\ \nabla\cdot\mathbf{u} &= 0 && \text{in } \Omega \end{aligned} \tag{5.15}$$

with boundary conditions

$$\begin{aligned} \sigma(\mathbf{u},p)\mathbf{n} &= \tau_n\mathbf{n} \text{ on } \Gamma_-\cup\Gamma_+, \\ (\mathbb{I}-\mathbf{n}\otimes\mathbf{n})\sigma(\mathbf{u},p)\mathbf{n} &= \tau_t, \\ \mathbf{u}\cdot\mathbf{n} &= 0 \text{ on } \Gamma_W. \end{aligned}$$

ii) Advection-diffusion-reaction (ADR) problems for θ, k and ε with variable viscosity

$$-\nabla\cdot(\nu\nabla u)+(\mathbf{a}\cdot\nabla)u+cu = f \qquad \text{in } \tilde{\Omega} \tag{5.16}$$

where $\tilde{\Omega}=\Omega$ or $\tilde{\Omega}=\Omega\setminus\Omega_\delta$ with Dirichlet boundary $\tilde{\Gamma}_D$ and von Neumann boundary $\tilde{\Gamma}_N$. The boundary conditions are

$$\begin{aligned} u &= g \text{ on } \tilde{\Gamma}_D, \\ \nu\nabla u\cdot\mathbf{n} &= h \text{ on } \tilde{\Gamma}_N. \end{aligned}$$

The testcase considered in Sec 5.2 requires an additional equation like (5.16) for the air age.

For the finite element discretization of (5.15)-(5.16) an admissible triangulation of Ω is assumed. The discrete subspaces are defined to consist of globally continuous and piecewise linear ansatz and test functions. The standard Galerkin FEM for the Oseen problem (5.15) with an equal-order ansatz for velocity and pressure does not pass the discrete inf-sup condition and must be stabilized [51]. For reasons of numerical stability pressure stabilization [68] (PSPG) together with divergence and SUPG [30] stabilization must be applied. The Galerkin-FEM for the ADR-problem (5.16) needs stabilization and is SUPG-stabilized, too.

The resulting linear systems are highly non-symmetric and in general of non-normal type. The discretized Oseen problem has a saddle-point structure. Figure 5.2 shows the dominating eigenvalues of the discretized Oseen problem for the testcase to be considered in Sec. 5.2. Preconditioning strategies for Eqs. (5.15) and (5.16) aim at a better clustering of the spectra.

5.2 Numerical Results

There seems to be only little which has been published about CUDA-based parallel preconditioners for nonsymmetric matrices and even less for multiphysics applications like non-isothermal air-flow including radiative heat transfer. In contrast to [144] we compare the performance of a serial ILU-implementation with an unfactored SpAI. The Block-Jacobi preconditioner parallels the approach pursued in [136]. However in the tests presented here, the blocksize is restricted to the number of DoFs per node (the 'dim' column in Table 5.1) and for the matrix element on the diagonal the full rather than the incomplete LU factorization is computed.

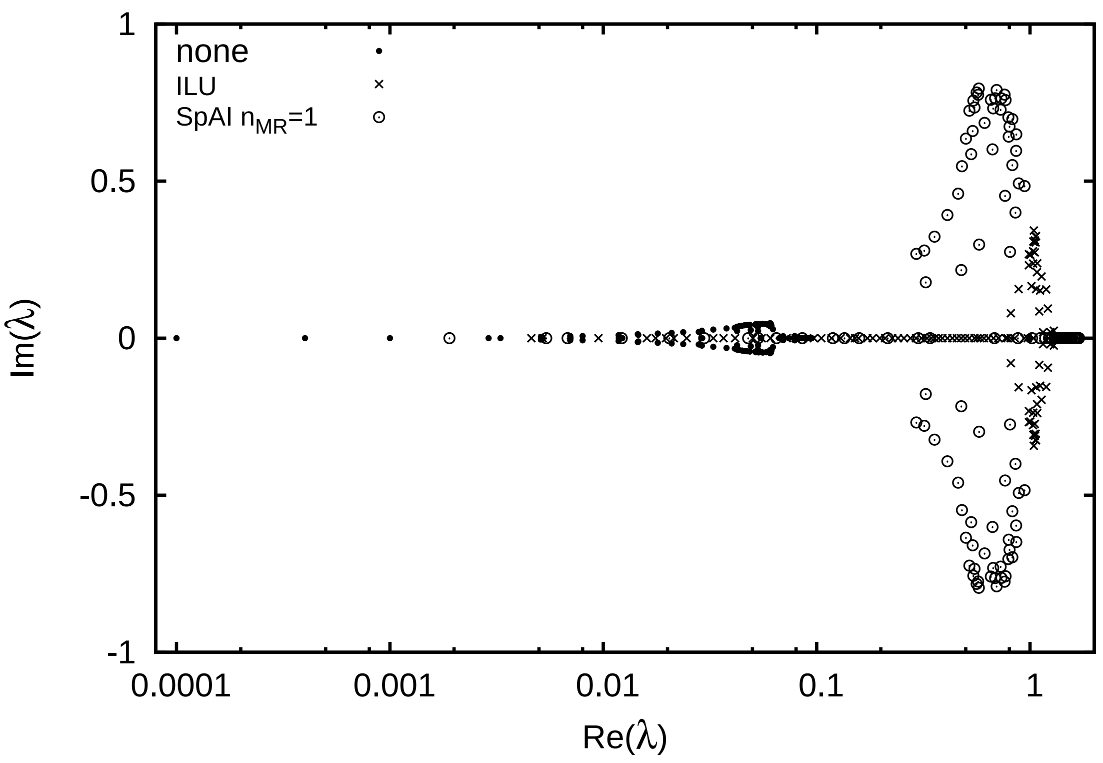
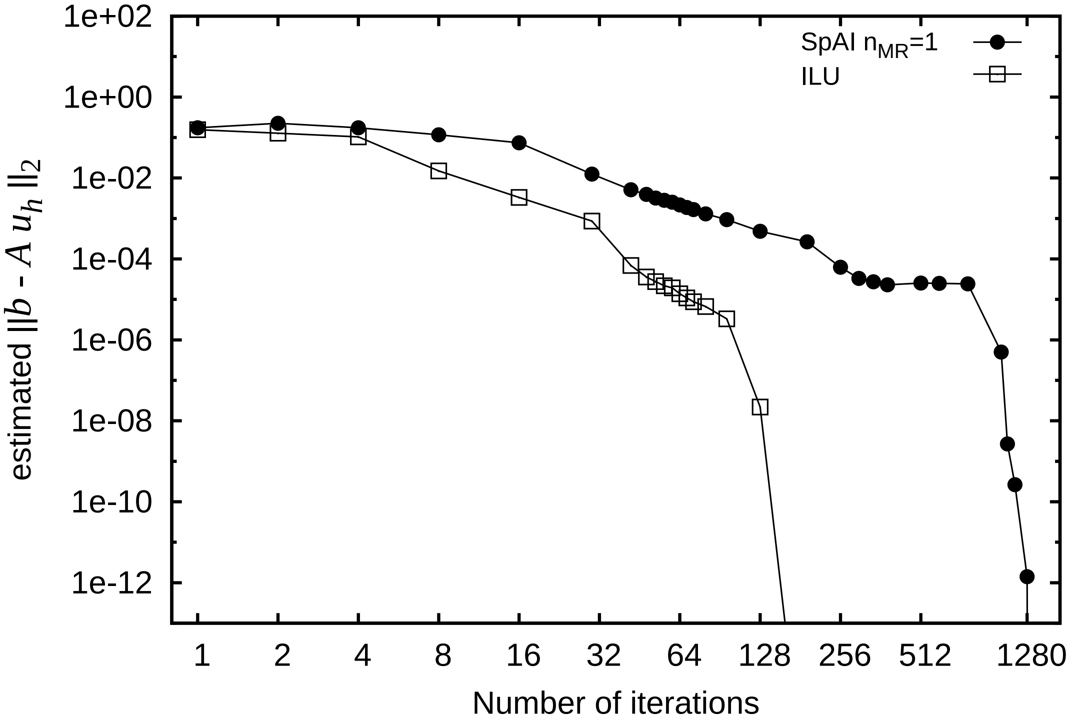


Figure 5.2: The dominating 128 eigenvalues λ of the unpreconditioned Oseen matrix ($\cdot$), right-preconditioned by ILU ($\times$) and by SpAI ($\circ$). The x-axis is in logarithmic scale to highlight the order of magnitude of the real parts. Eigenvalues were computed from the Hessenberg matrix obtained from the Arnoldi process using MATLAB's `eig()` function.

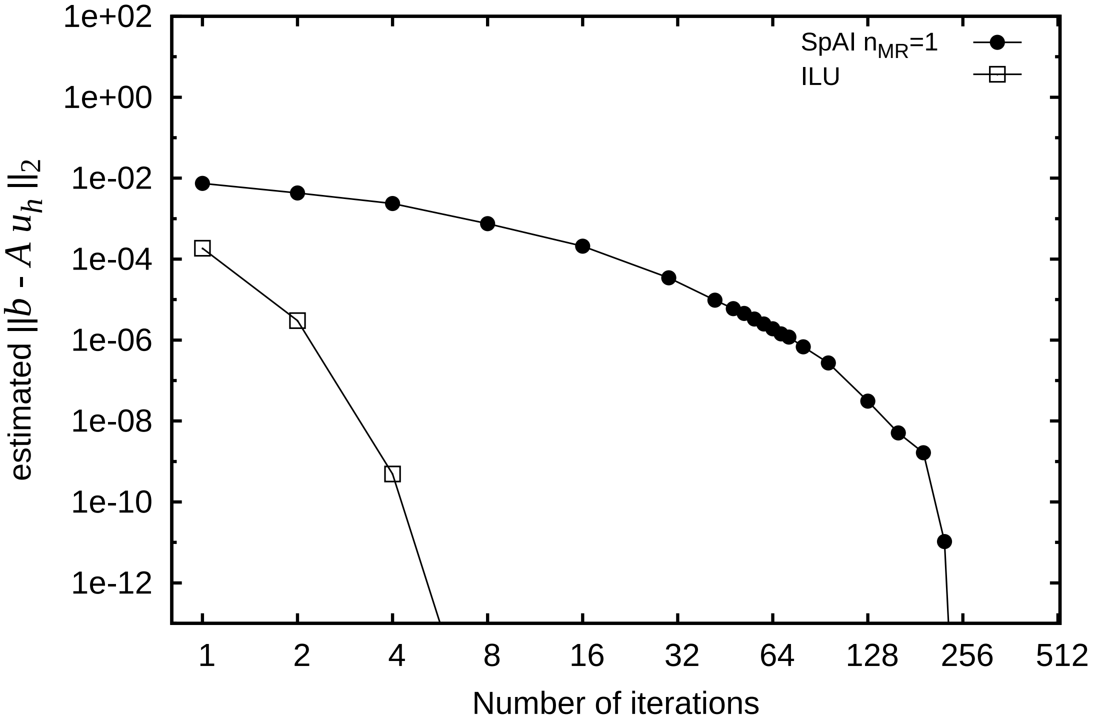
Our tests were done on a Dell T7500 workstation equipped with two Intel Xeon X5650, 96 GB RAM, and two NVIDIA Tesla C2070, running under Ubuntu 10.04 and CUDA 4.0. As integrated development environment we use QtCreator. The test programs are compiled in its predefined release mode. Total runtimes and times per iteration are summarized in Table 5.1.

Our test matrices stem from a case study of indoor air flow in a room subject to energy-focused building refurbishment. The goal was to measure the impact on air exchange by decentralized air-conditioning attached to the windows. The room was discretized by a tetrahedral mesh with 80621 nodes resulting in 644968 degrees of freedom (DoFs) in total. For details see Table 5.1. A snapshot of the flow field and distribution of air age is displayed in Figure 5.1b. We solve Eqs. (5.15) and (5.16) by GMRES [112] and QMRCGSTAB [33]. Especially the latter has turned out to be the best choice for solving a broad range of problems and on average needs 30% less iterations to converge. Therefore, GMRES is not considered in the following. As reference preconditioner on the CPU we use ILU(0), i.e. the sparsity pattern of the L and U factors are restricted to the one of the original matrix.

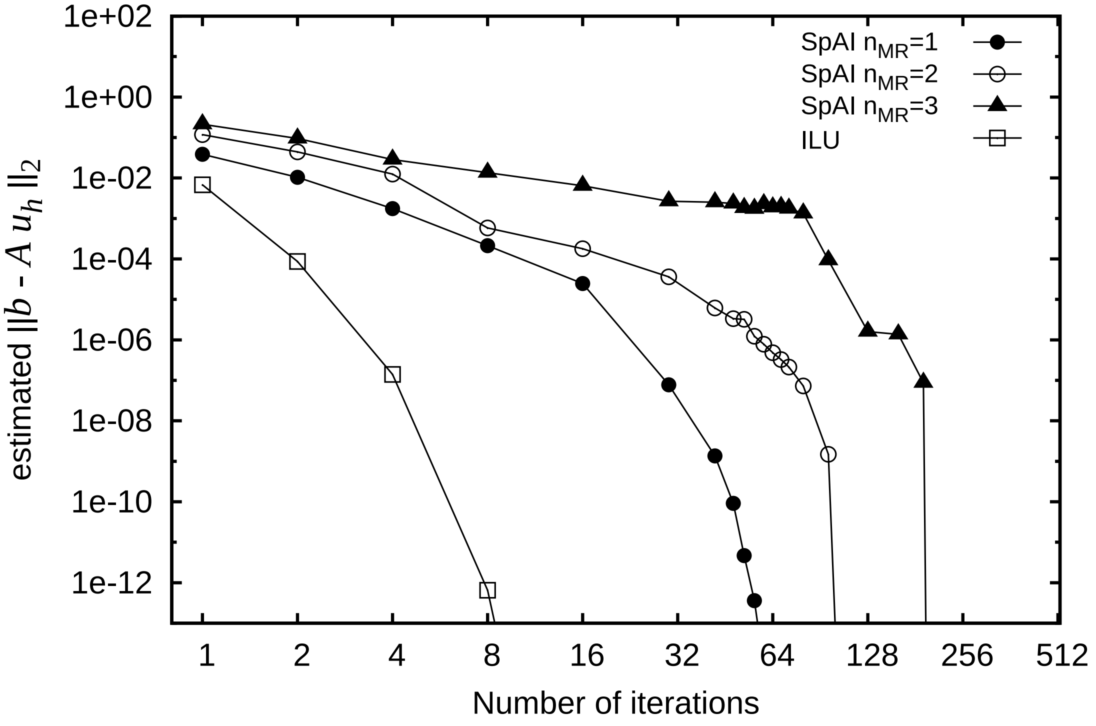
For illustrating the performance of the preconditioners we considered the clustering of the spectra of the preconditioned linear systems and the convergence history. For good preconditioning eigenvalues should cluster in the vicinity of the point $(1,0)$ in the complex plane, cf. Fig. 5.2 for a comparison of SpAI with ILU.



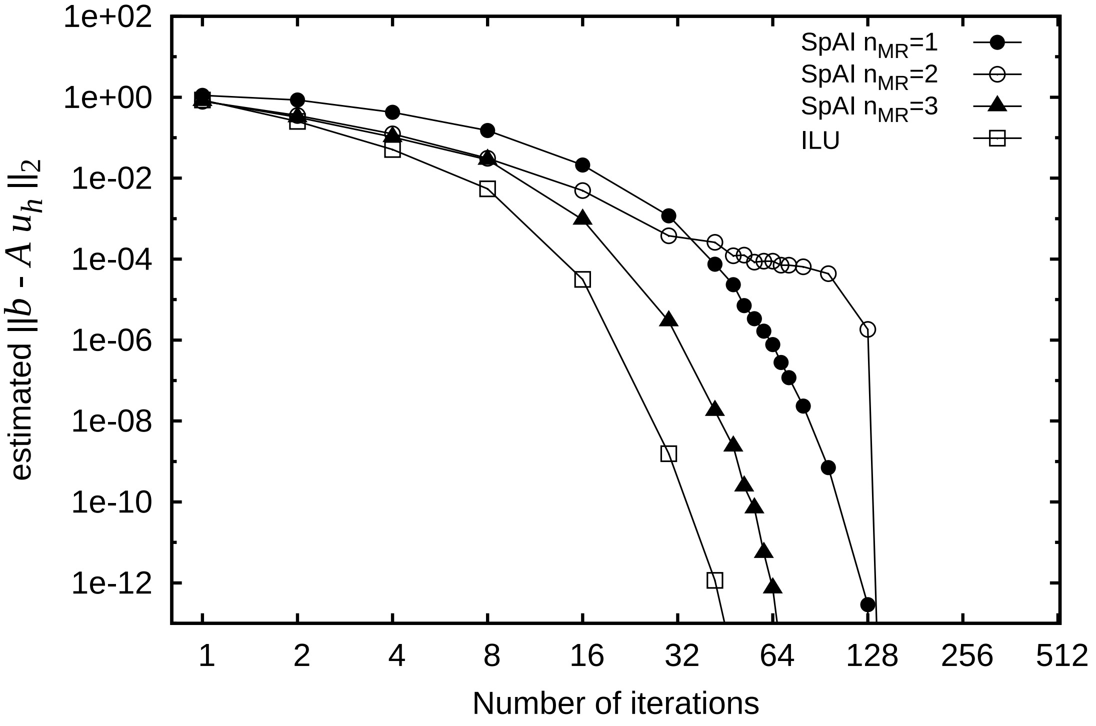
(a) Oseen



(b) k-ε.

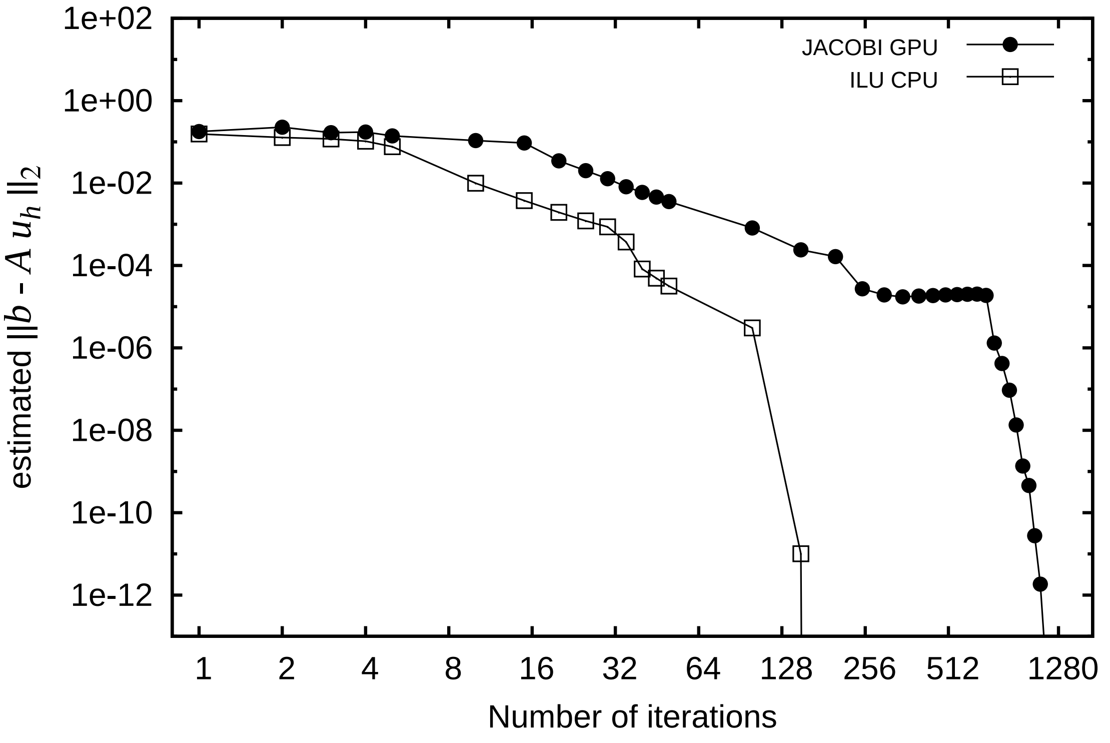


(c) Fourier

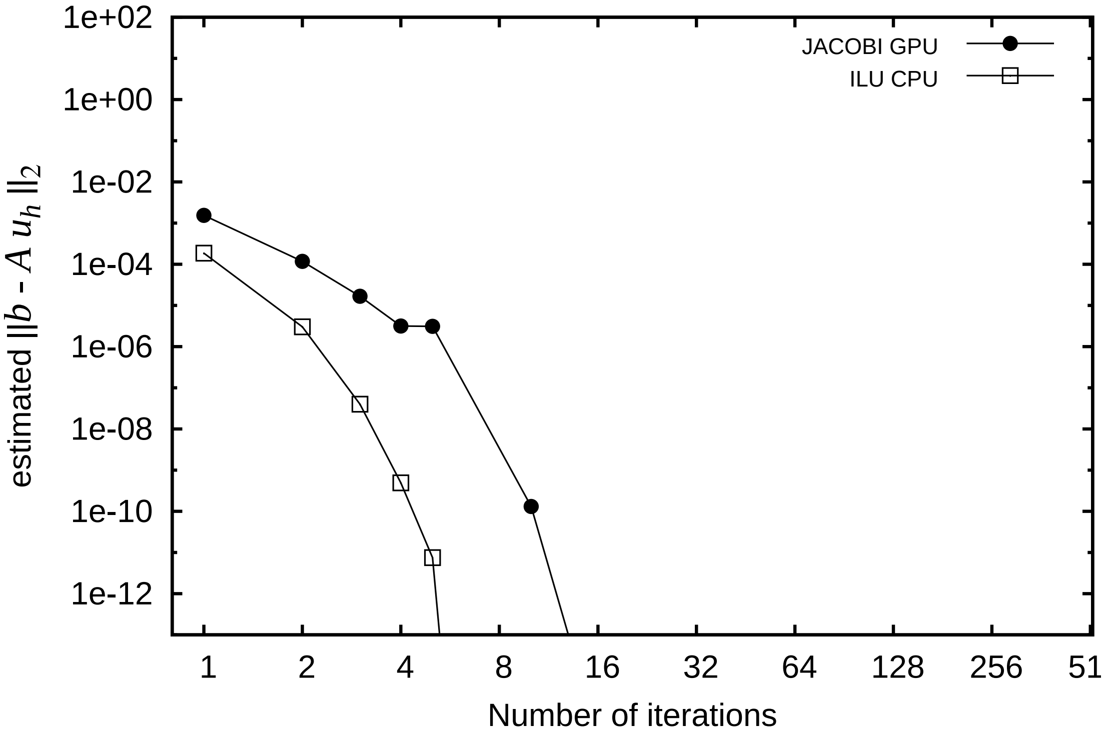


(d) Air age

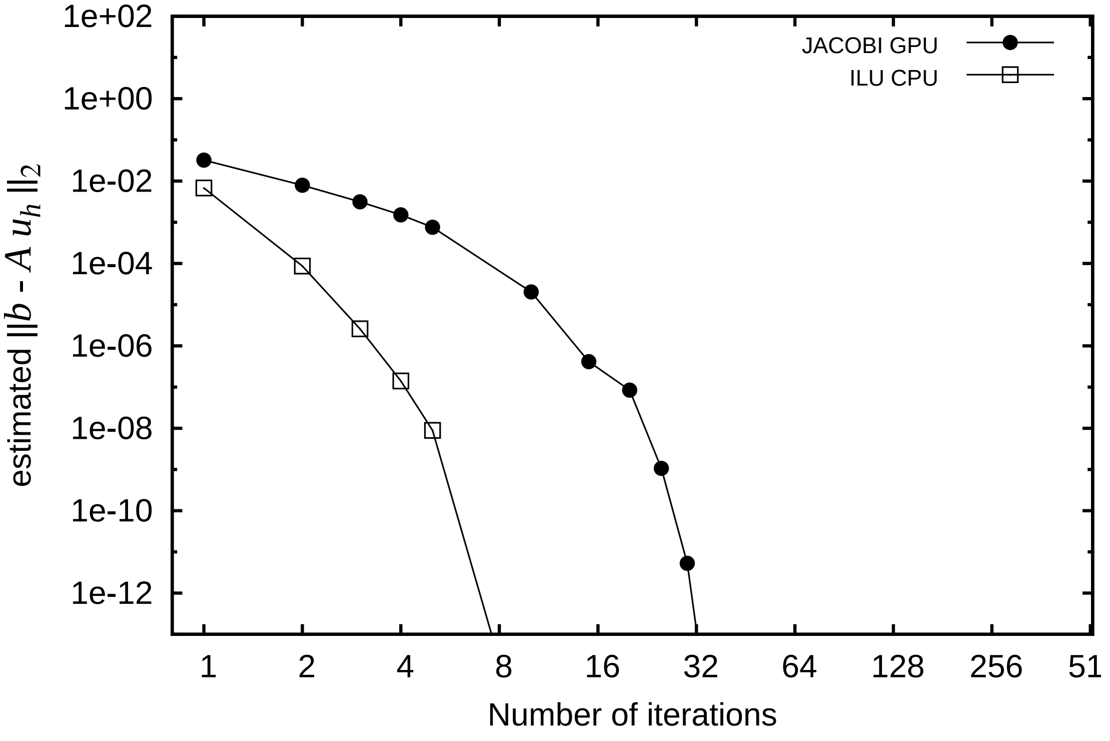
Figure 5.3: SpAI results for QMRCGSTAB [33].



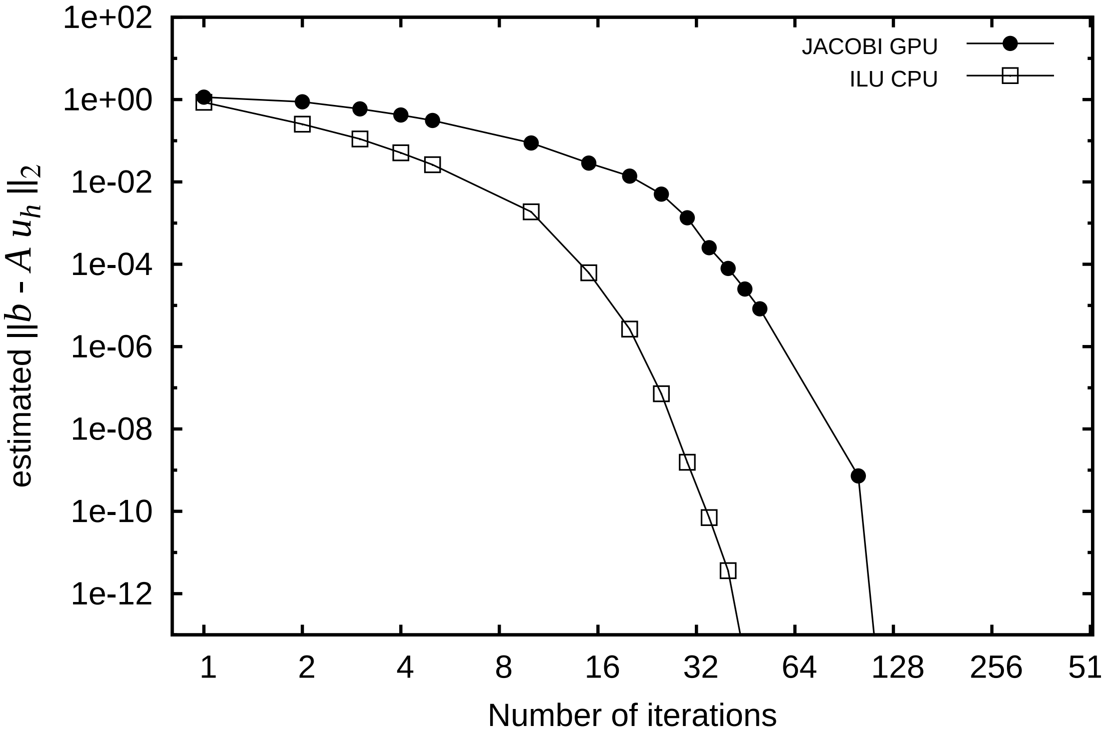
(a) Oseen



(b) k-ε.



(c) Fourier



(d) Air age

Figure 5.4: Block-Jacobi results for QMRCGSTAB [33].

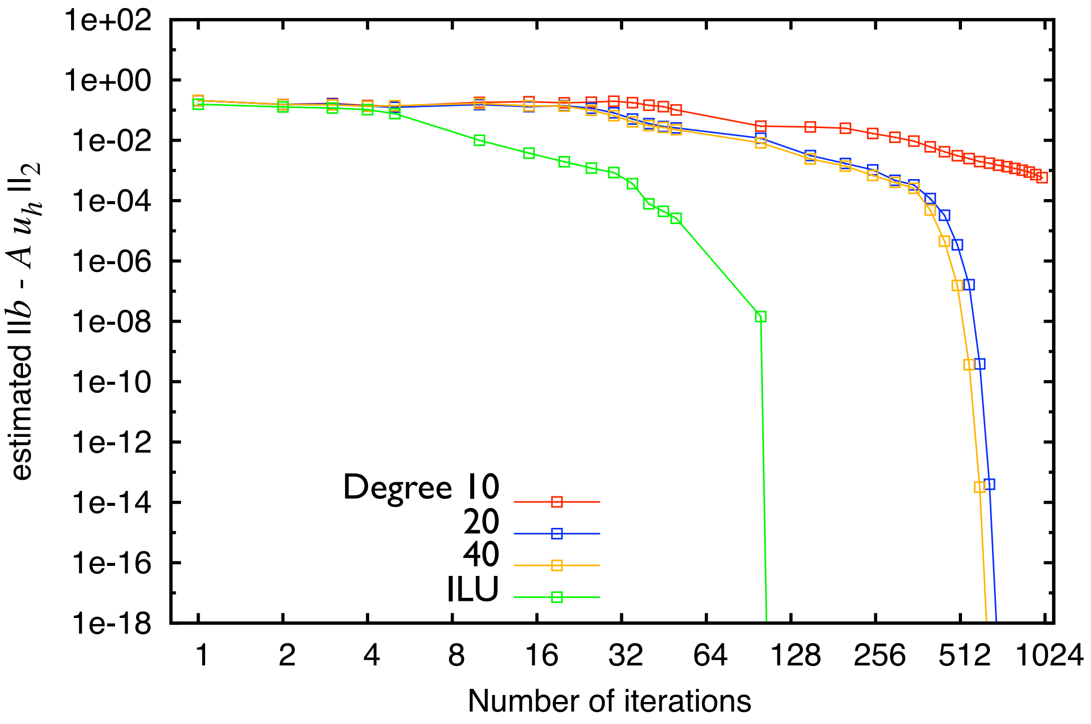


Figure 5.5: PPc results for the Oseen problem solved by QMRCGSTAB.

5.2.1 Polynomial Preconditioning

Polynomial preconditioning as described in Sec. 3.2.4 was slower than the CPU-based ILU. The high polynomial degree needed for convergence and the increased iteration count more than compensated the gain in speed obtained from the parallelization of the SpMV. For sake of completeness the results for PPc are listed as well but without detailed discussion. The convergence history for the Oseen problem when solved with QMRCGSTAB preconditioned for different polynomial degrees is shown in Fig. 5.5. The inclusion set and the dominating part of the spectrum of the unpreconditioned matrix is given in Fig. 5.6.

5.2.2 Sparse Approximate Inverse

The convergence histories for different numbers n_{MR} of MinRes steps are summarized in Figs. 5.3. Our tests show that already $n_{MR} = 1$ suffices to obtain a preconditioner with reasonable performance, cf. Fig. 5.3. SpAI takes more steps to converge than ILU but this is more than compensated by the efficient implementation of the SpMV which is needed to apply the preconditioner. In most of the cases ILU is outperformed by SpAI.

Table 5.1 shows that for the Oseen subproblem, which is the most expensive one, we measured an average speed up of ca. 40 for the individual iteration step but have to pay with a considerably higher iteration count until convergence. Thus, our CUDA-based SpAI is roughly five times faster than the ILU preconditioner used so far. As the increased iteration count is due

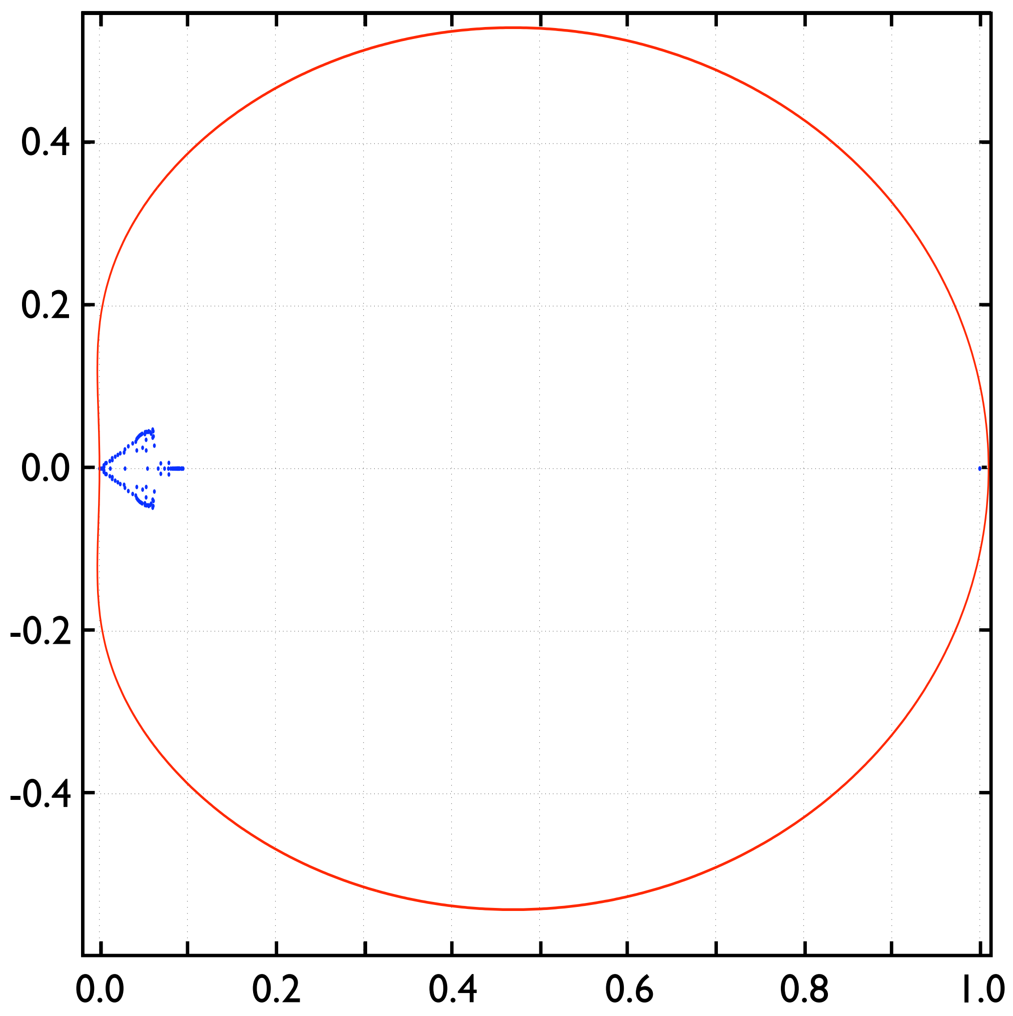


Figure 5.6: Spectrum (blue dots) and inclusion set (red) for the Oseen problem.

to the change of the preconditioner from ILU to SpAI the same behavior should occur if the SpAI was parallelized on the CPU using OpenMP for instance. For that case we would expect a speedup of the individual iteration step which would only compensate the increased amount of iterations due to the lower memory bandwidth of current CPUs compared to GPUs so that there would be no net gain. For instance, a Westmere Xeon has roughly one-fifth of the bandwidth of a Tesla card. Hence, to get a speed up due to massive parallelization a high-bandwidth architecture like CUDA is mandatory.

With respect to the number of iterations SpAI is not as effective as ILU. Yet, ILU is outperformed because the individual SpAI-preconditioned Krylov iteration step is almost two orders of magnitudes faster, especially for the Oseen problem. Provided the decision has been made to stick with the SpAI approach from Sec. 3.2.3, Eq. (3.14) shows that further improvements must tackle the problem of finding a cheap inverse of $\mathbf{AM}^{-1}$. A simple and cheap improvement would be to use a Block-Jacobi preconditioner based on the diagonal elements $\mathbf{D}_i := (\mathbf{AM}^{-1})_{ii}$ and to precondition Eq. (3.14) with the matrix $\mathbf{J}^{-1} := (\mathbf{D}_i^{-1})_{i=0}^{N}$ from the left. As the corresponding kernels are already at our disposal, cf. Sec. 2.1.2, we could also base $\mathbf{J}^{-1}$ on the

LU-factorizations of 16×16 or 32×32 diagonal blocks and either compute their exact inverse in parallel on the GPU or stick to the traditional approach to solve small linear systems. The structure of $\mathbf{J}^{-1}$ should simplify coalesced memory accesses in the SpMV thus exploiting the theoretical memory bandwidth.

5.2.3 Block-Jacobi

The results on preconditioning with the Block-Jacobi preconditioner are summarized in Fig. 5.4. The timing results are given in Table 5.1. Of all tested parallel preconditioners Block-Jacobi was the most satisfactory alternative to ILU. The setup is cheap, its application only requires one SpMV and because of the block-diagonal structure of the preconditioning matrix the access to the source vector is not random. However, like for the SpAI case the number of iterations to reach convergence is higher than for ILU. For the Oseen equation and the equation for the air age SpAI and Block-Jacobi have a similar convergence behavior. Both need roughly the same number of iterations The big difference is the behavior for the k-ε model and the temperature equation. For k-ε SpAI is slower than ILU whereas Block-Jacobi only needs roughly twice as much iterations as ILU which suffices to be faster than ILU by almost an order of magnitude. For the temperature equation Block-Jacobi is faster than SpAI and ILU whereas for the equation for air age SpAI is faster than ILU and Block-Jacobi.

Table 5.1: Typical speedups for the different parallel preconditioning strategies. To optimize usage of CPUs and GPU move the Oseen problem to the GPU and solve the other problems on the CPU. Numbers are approximate as in practice the precise figures differ from one time-step to another, e.g. due to changes in machine load or variations in the matrix properties.

case	nnz	dim	DoFs	iterations to convergence		time per iteration / sec	Wall time / sec
Oseen	18275344	4	322484	ILU	180	0.6	110
				SpAI	1280	0.014	18
				Faber (20)	700	0.3	210
				Block-Jacobi	1100	0.015	16
k-ε	4568836	2	161242	ILU	5	0.4	2
				SpAI	240	0.01	2.4
				Faber (20)	256	0.2	51
				Block-Jacobi	11	0.02	0.2
Fourier	1142209	1	80621	ILU	8	0.05	0.4
				SpAI	60	0.01	0.6
				Faber (20)	256	0.1	26
				Block-Jacobi	31	0.01	0.3
air-age	1142209	1	80621	ILU	40	0.05	2
				SpAI	64	0.01	0.6
				Faber (20)	230	0.1	23
				Block-Jacobi	120	0.01	1.2

Chapter 6

Quantum Simulation of Single-Electron Transport

…and now for something completely different[1]*: Boundary conditions.*

High electron mobility GaAs/AlGaAs heterostructures can accommodate two-dimensional electron gases of almost macroscopic size. Recent experiments [126, 127, 11, 78] on ballistic transport and magneto-conductance have shown that even in high purity samples the remaining weak disorder potential due to the donor layer causes a branching of electron flow in the two-dimensional electron gas. The wave nature of the current carrying, electronic scattering states can only be revealed by directly solving the Schrödinger equation for the entire system including the semi-infinite leads. In an experiment the transport properties of a heterostructure are often governed by a small part of the device. The truncation of the parts of the experimental domain irrelevant for the scattering can only be achieved by artificial, transparent boundaries. For wave-like phenomena restricting the computation to a finite region is prone to spurious reflections.

To exploit the flexibility concerning complex geometries and the high quality of current conservation provided by higher order finite elements we need exact transparent boundary conditions for the case that the scattered wave function is a superposition of several modes and is modeled by finite elements. The Hardy space infinite element method [61] provides a rigorous way of setting up taylor-made, exact transparent boundary conditions for finite element simulations of the quantum magneto-conductance in the Landauer-Büttiker picture.

In this chapter we describe a deal.II-based, finite element simulation framework for the investigation of branching of electron flow due to weak disorder potentials in the presence of a magnetic field in heterostructures of macroscopic size. For the sake of brevity, the scope of this chapter is limited to the discussion of discretizing a Hamiltonian on a domain with transparent boundaries and the validation of the methods using a non-trivial physical model problem. The discussion of parallelization is resumed in chapter 7 where a framework for time-dependent transport is developed.

6.1 Introduction

During the past three decades, low-temperature quantum transport phenomena in mesoscopic electron devices have been intensively studied. Their in-depth study in electron inversion layers in heterostructures, metallic films and MOSFETs has been initiated by the discovery of signatures of quantum interference in the electric current through mesoscopic devices. Especially sub-micron Aharonov-Bohm (AB) like geometries [117, 17], quantization of the conductance

[1]Little homage to Monty Python's Flying Circus

of point contacts, in Hall geometries [71, 128, 140] and by advances in low-temperature physics without experiments close to absolute zero temperature would not have been possible. Other phenomena of interest are, for instance, reproducible (universal) conductance fluctuations [46] or signatures of quantum chaos in electron transport through two dimensional cavities [92]. Recently, spatially resolved recordings of electron densities using scanning tunneling microscopy techniques have revealed caustic phenomena and current branching in absence [126, 127, 78] and in presence of magnetic fields [11, 95].

The theoretical understanding of quantum transport has been pioneered by R. Landauer [80] and Büttiker [32]. Quantitative calculations of the phenomena have been carried out by using the Landauer relation between zero-temperature direct current electron conductance and quantum transmission [37].

In practice, this implies that the calculation of the quantum coherent transport effects requires knowledge of the quantum transmission amplitudes [66]. These can be determined without explicitly knowing the scattering states. However, a central result of recent scanning tunneling experiments [126, 127, 11] is a spatially resolved picture of the electron density of the two-dimensional (2D) electron gas (2DEG) under the condition of stationary current flow. For a detailed comparison of theory, simulation and experiment the computation of the scattering wave functions is mandatory. It is the aim of this thesis to outline a tool for the accurate prediction of the wave functions of scattering states in semi-infinite systems with complex boundaries.

In the ballistic regime, i.e. on length scales less than the mean free path between two scattering events, electrons move almost freely. By lowering the temperature the inelastic mean free path due to interaction effects can be made arbitrarily long leaving the mean free path due to impurities, i.e. a material parameter, to set the length scale for the dimensions of a ballistic 2DEG device. In specimen smaller than the mean free path scattering occurs only at the walls and in the contacts.

At low temperatures and for high electron-mobility materials like GaAs/AlGaAs heterostructures the mean free path of an electron can be of the order of hundreds of micrometers. Hence, transport in micron-size semiconductor devices as commonly used in experiments can be considered as essentially ballistic and the quantum mechanical properties follow from the single-electron Schrödinger equation using an appropriate effective mass for the electron.

Modern fabrication technologies allow for preparation of almost ideal 2DEGs with large electron mean free paths. It nevertheless seems necessary to take into account some randomness in terms of a weak potential in order to explain certain experimental findings like the branching of electron flow mentioned above. A commonly accepted source for disorder is the weak potential created by charge fluctuations in the donor layer beneath the 2DEG. The influence of magnetic fields on quasi-ballistic electron transport in 2DEGs plays also an important role in many experiments [37].

The main contribution of this thesis is to adapt a recently developed type of transparent boundary conditions (TBC) for finite element methods for acoustic scattering [101] to quantum transport computations in semi-open electron systems. The presentation of the TBC is independent of the spatial dimension and thus applies to the three-dimensional case as well. Due to the widespread interest gained by imaging the current carrying electron states in magnetic focussing [11, 95] it is shown how to extend the TBC to account for the experimentally important case of a constant magnetic field perpendicular to the 2DEG.

The benefits of the method are demonstrated by computing the scattering states and their spatial electron density distributions for the specimen with chamfered corner discussed in the thesis by Metzger[95]. The purpose of the TBC is to truncate the specimen to the physically relevant part such that the computational domain is confined to the subregion where scattering actually occurs. This substantially lowers the computational cost and enables us to do a full quantum mechanical calculation of the stationary scattering states taking into account several incident and transmitted transport channels. In contrast to standard recursive Green's function approaches based on tight-binding methods we use higher order finite elements and work on irregular, problem-adapted meshes which cannot be obtained from cartesian grids by means of a single coordinate transformation [111]. The numerical accuracy of the method is demonstrated by studying the quality of the current conservation in a quantum wire.

6.2 Modeling Ballistic Electron Transport

The scattering domain is modeled as some finite area Ω, cf. Fig. 6.1. Electrons can enter and leave the scattering domain only via perfect leads which by definition have constant width and infinite length. Any additional transverse potential in a lead remains constant along its length. Sections of the lead where the potential varies in the longitudinal direction must be included in the scattering domain.

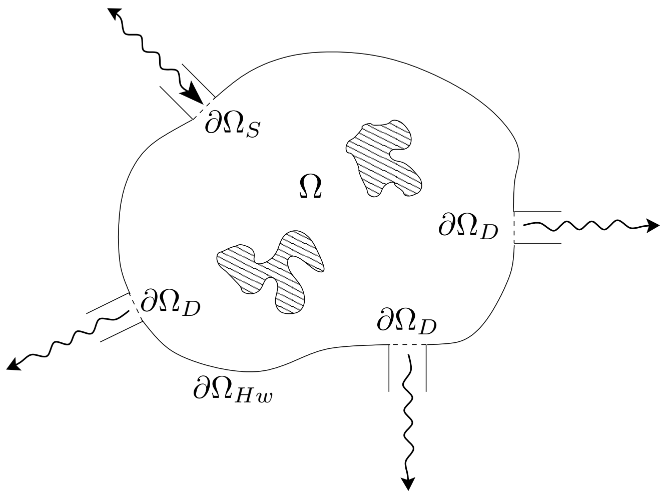


Figure 6.1: Sketch of experiment. Hatched subdomains denote regions containing scatterers.

The walls of the leads and the impenetrable part of the boundary $\partial\Omega$ of the scattering domain are denoted as $\partial\Omega_{Hw}$. The contacts between the leads and the scattering domain are referred to as either $\partial\Omega_S$, for the source lead, or as $\partial\Omega_D$ for drain leads. The quantum point contacts (QPCs) employed in experiments for connecting the leads to the specimen are modeled as short narrow constrictions of the leads close to Ω cf. Fig. 6.7.

In the following we consider geometries with only two leads. One acts as source and the other as drain. We could deal with the general multi-terminal case from Fig. 6.1 as well but in the context of this work the focus is rather on replacing sections of the hard walls by transparent boundaries. Within the leads the coordinate x denotes the longitudinal coordinate and points away from Ω. The transverse coordinate is y. The coordinates (x,y) in the leads are local ones, i.e. $x=0$ denotes the interface to the scattering domain. We will use this convention throughout this work as the meaning of x and y is usually obvious from the context unless otherwise stated. The wave function Ψ_p in the drain is a superposition of the incident plane waves ψ_q from the source after scattering. Their individual contributions are measured by scattering amplitudes t_{pq}.

At absolute zero of the temperature, in the absence of any interaction, electron transport is quantum mechanically coherent and only the electrons at the Fermi level with energy E_F and wave length λ_F contribute to the current. For these electrons the wave function in a drain lead far away from the scattering domain can be written as

$$\Psi_p = \sum_q t_{pq}\psi_q\,, \tag{6.1}$$

$$\psi_q = g_q(y)e^{ik_q x} \tag{6.2}$$

where q counts the active states in the drain. We assume that the states in the source and drain lead with respect to their local coordinate systems are given by the same set of orthogonal functions $\{\psi_p\}_{p\in\mathbb{N}}$. The details of the transverse potential in the drain determine the shape of the transverse profile $g_q(y)$ of the wave function and the distribution of the energy levels

$$\varepsilon_q = E_F - k_q^2$$

due to the transverse confinement. Current is carried only by those modes for which longitudinal wave numbers $k_q = \sqrt{E_F - \varepsilon_q}$ are real.

The functions $g_q(y)$ solve the transverse eigenproblem and are mutually orthogonal. Units are chosen such that the effective mass of the electron, its charge and Planck's constant are one, see Sec. 6.2.2. Normalization of the functions g_q is chosen such that

$$\int_{\partial\Omega_X} |g_q|^2 = 1, \quad X \in \{S,D\}\,. \tag{6.3}$$

6.2.1 Quantum Conductance

Transmission and reflection probabilities are computed from the quantum mechanical probability current density.

In a lead of finite width L the net current density for an arbitrary wave function Ψ is given by the normal component which can be obtained by computing the contribution flowing in the direction of the outbound normal n of the channel cross section

$$\mathbf{j}(y) = \frac{\hbar}{m^*}\mathrm{Im}(\Psi\partial_n\Psi^*) \tag{6.4}$$

where $\partial_n = n\cdot\nabla$ is the normal derivative and m^* is the effective mass. Integrating over the lead's cross section gives the current

$$\mathbf{J} = \mathrm{Im}\left(\int_0^L dy\Psi(x_{out},y)\partial_n\Psi^*(x_{out},y)\right)\,, \tag{6.5}$$

where x_{out} is the longitudinal coordinate denoting the position where the lead is cut off, see Fig. 6.2. The wave function and thus the probability current in Eq. (6.5) depends on the incoming mode p. Normalizing the current with respect to the current of the incoming mode of wave number $k_p = p\pi$, i.e. to divide the current by k_p, gives the transmission probability

$$T_p(\Psi_p) = \mathbf{J}_p/k_p\,. \tag{6.6}$$

The total transmission T is given by summing over p, i.e. the contributions due to all open input modes. In our units it is identical to the total conductance

$$T = \sum_p T_p(\Psi_p) = \sum_p \mathbf{J}_p / k_p . \tag{6.7}$$

The unknown amplitudes $t_{pq} \in \mathbb{C}$ are determined by the details of the scattering mechanism and the particular input mode ψ_p in the source lead. Provided both leads have the same shape and mode p is used as input, utilizing the decomposition given in Eq. (6.1) the conductance between the source and one drain lead is given by the well-known Landauer-Büttiker formula

$$G = \frac{e_0^2}{h} T = \frac{e_0^2}{h} \sum_p \sum_q \frac{k_q}{k_p} |t_{pq}|^2 \tag{6.8}$$

where e_0 is the elementary charge.

6.2.2 Single Electron Description

The wave function Ψ describing our scattering state on Ω is determined by the stationary Schrödinger equation

$$\frac{1}{2m^*} \left[\left(\frac{\hbar}{i} \nabla - q\mathbf{A}(\mathbf{x}) \right)^2 + V(\mathbf{x}) - E \right] \Psi = 0 \tag{6.9}$$

of a single electron with effective mass m^*, charge q and total energy E in the presence of a vector potential $\mathbf{A}(\mathbf{x})$ giving rise to a static magnetic field. The model is completed by a suitable set of boundary conditions to describe hard walls and the semi-infiniteness of the leads.

To describe a constant magnetic field within the scattering domain Ω we use $\mathbf{A}(\mathbf{x}) = B(-y, 0, 0)^T$, i.e. the Landau gauge. Throughout this work, B is considered as the primary control parameter. Depending on the spatial scale a the dimensionless quantities $\frac{2m^* a^2}{\hbar^2} E \rightarrow E$, $\frac{qa}{\hbar} \mathbf{A} \rightarrow \mathbf{A}$, $a\nabla \rightarrow \nabla$, $\frac{2m^* a^2}{\hbar^2} qV \rightarrow V$ lead to the final dimensionless Schrödinger equation

$$\left[-\nabla^2 + 2i\mathbf{A} \cdot \nabla + |\mathbf{A}|^2 + V - E \right] \Psi = 0 . \tag{6.10}$$

The spatial scale a (corresponds to the lattice parameter in tight-binding calculations) is taken to be in the range .1 to 1μm. As effective mass we can use the one of GaAs, i.e. $m^* = 0.067 m_0$ at 0 K when measured in units of the mass of a free electron m_0.

At those parts $\partial\Omega_{Hw}$ of the boundary $\partial\Omega$ where no leads are attached to the scattering domain Ω the wave function is subject to perfectly hard walls, i.e. we have to use homogeneous Dirichlet conditions by setting the value of the wave function to zero

$$\Psi \Big|_{\partial\Omega_{Hw}} = 0 . \tag{6.11}$$

Far away from the scattering domain the scattered part of the electron's wave function has to match the free-particle behavior of the wave function in the leads, i.e. it has to fulfill Sommerfeld's radiation condition

$$\lim_{x \to \infty} x^{\frac{1}{2}} (\partial_x - ik) \Psi \Big|_{\partial\Omega_D} = 0 . \tag{6.12}$$

6.2.3 Variational Formulation

We subdivide the domain Ω into a finite interior part Ω_{int} consisting of the scattering domain and the stubs, cf. Fig. 6.2, and an exterior part Ω_{ext} containing the unbounded exterior parts of the leads. The edge interfacing Ω_{int} and Ω_{ext} at the cut off will be denoted by Γ. By construction finite elements are only applicable to bounded domains which obviously contradicts leads of infinite length. The semi-infinite extent of the leads has to be modeled by suitable surrogate

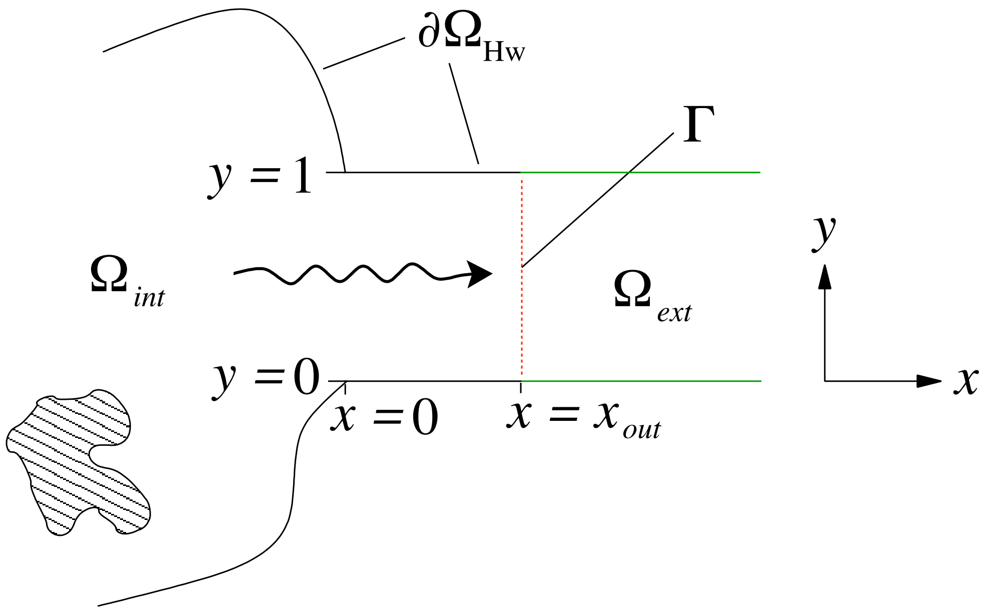


Figure 6.2: Sketch of outlet interface to exterior Hardy domain. The dotted line separates the finite domain Ω of the interior problem from the infinite exterior domain Ω_{ext}.

boundary conditions which allow an outgoing wave to pass without any artificial reflections. Such a condition is essentially given by Sommerfeld's radiation condition, Eq. (6.12), but for numerical purposes its direct application is impractical. Instead, we construct the boundary conditions from Hardy-space infinite elements [61] which basically are a transformation of Sommerfeld's radiation condition, cf. Eq. (6.12), to the unit circle. The zero-Dirichlet boundary conditions modeling the hard walls are directly built into the function space in which the solution is sought. The weak form is obtained from multiplying both sides of Eq. (6.10) with a test function Φ and integrating over Ω

$$\begin{aligned} -\int_\Omega \Phi\nabla^2\Psi + 2i\int_\Omega \Phi\mathbf{A}\cdot\nabla\Psi & \\ + \int_\Omega \Phi\left[|\mathbf{A}|^2 + V - E\right]\Psi &= 0. \end{aligned} \tag{6.13}$$

For scalar products we introduce the short-hand notation

$$(u,v)_D := \int_D uv \tag{6.14}$$

where D is the domain of integration and u and v are some functions defined on D. If D denotes a subset of the boundary u and v denote the restrictions to it.

The weak form is obtained by integrating by parts which removes one derivative of the solution. To this end, we explicitly write $(\cdot,\cdot)_\Omega = (\cdot,\cdot)_{\Omega_{int}} + (\cdot,\cdot)_{\Omega_{ext}}$ and integrate on both subdomains independently. In the arising boundary terms, $(\cdot,\cdot)_{\partial\Omega_{int}}$ and $(\cdot,\cdot)_{\partial\Omega_{ext}}$, we denote by n_{int} the outer normal of the interior domain Ω_{int} and by n_{ext} the outer normal of the exterior domain Ω_{ext}. At the interfaces Γ where Ω_{int} and Ω_{ext} have a common edge both normals are opposite to each other, $n_{int}|_\Gamma = -n_{ext}|_\Gamma$. The test functions Φ are chosen such that they vanish on $\partial\Omega_{Hw}$ so that only the boundary terms at the interfaces Γ remain. To emphasize that we are going to treat interior and exterior differently, the integrals on the exterior part will be given explicitly. Then, the weak form formally reads

$$\begin{aligned}(\nabla\Phi,\nabla\Psi)_{\Omega_{int}} + \left(\Phi,\left[|\mathbf{A}|^2+V-E\right]\Psi\right)_{\Omega_{int}} + (\Phi, 2i\mathbf{A}\cdot\nabla\Psi)_{\Omega_{int}} - (\Phi,\partial_{n_{int}}\Psi)_\Gamma & \\ + \int_0^\infty \int_\Gamma \nabla\Phi\cdot\nabla\Psi + 2i\Phi\mathbf{A}\cdot\nabla\Psi + \Phi\left[|\mathbf{A}|^2+V-E\right]\Psi - \int_\Gamma \Phi\partial_{n_{ext}}\Psi &= 0\,. \end{aligned} \tag{6.15}$$

For notational simplicity we define the bilinear forms $a(\cdot,\cdot)$ for the interior problem

$$\begin{aligned} a(\Phi,\Psi) \;:=\; & (\nabla\Phi,\nabla\Psi)_{\Omega_{int}} \\ & + \left(\Phi,\left[|\mathbf{A}|^2+V-E\right]\Psi\right)_{\Omega_{int}} \\ & + (\Phi,2i\mathbf{A}\cdot\nabla\Psi)_{\Omega_{int}} \end{aligned} \tag{6.16}$$

and $b(\cdot,\cdot)$ for the sum of the exterior problems in the truncated part of the leads which eventually will give us the required transparent boundary conditions

$$\begin{aligned} b(\Phi,\Psi) \;:=\; & \sum_c (\nabla\Phi,\nabla\Psi)_{\Gamma_c\times\mathbb{R}^+} \\ & + \left(\Phi,\left[|\mathbf{A}|^2+V-E\right]\Psi\right)_{\Gamma_c\times\mathbb{R}^+} \\ & + (\Phi,2i\mathbf{A}\cdot\nabla\Psi)_{\Gamma_c\times\mathbb{R}^+}\,. \end{aligned} \tag{6.17}$$

For the drain lead the surface integrals in Eq. (6.15) cancel each other. In the source lead the wave function consists of an incoming plain wave part f and a contribution due to the backscattering at the contact. As Ψ describes the scattered part of the wave function the surface integrals do not cancel and the surface integral over f remains. This yields the right-hand side

$$f(\Phi) \;:=\; (\Phi,\partial_{n_{ext}}f)_{\Gamma_{in}} \tag{6.18}$$

where Γ_{in} is the interface to the source lead. The variational problem is:

find $\Psi\in X\subset H^1(\Omega_{int})$ *such that* $\forall\Phi\in X$:

$$a(\Phi,\Psi)+b(\Phi,\Psi) \;=\; f(\Phi)\,. \tag{6.19}$$

The terms due to the interior problem can be treated by standard finite element methods. For the details of how to setup the solution space X we refer the interested reader to the thorough disscussion in [100]. Repeating the construction of X is not the purpose of this thesis.

6.3 Hardy Space Infinite Elements

For the exterior problem we have to convert the integral over the longitudinal coordinate into an integral over a finite domain to make the integral exist.

6.3.1 Converting $\int_0^\infty$ to $\int_0^{2\pi}$

The inapplicability of finite element methods to infinite domains forces us to model the leads as finite stubs by truncation at x_{out}, see Fig. 6.2. To recover the infiniteness of the domain we employ so-called Hardy space infinite elements (HSE) as developed in the thesis by Nannen [100]. In contrast to other methods Hardy space infinite elements are formally exact and lead to a rather sparse matrix for the discretized boundary conditions.

HSE map the longitudinal coordinate x of the unimportant exterior subdomain onto the unit circle so that $\int_0^\infty$ is transformed into $\int_0^{2\pi}$. This is achieved by applying a Möbius transformation $\mathscr{M}_{\kappa_0}$ to the Laplace transform $\hat{u}(s)$ of the wave function $u(x)$ with respect to the longitudinal coordinate

$$u(x) \quad \overset{\mathscr{L}}{\longmapsto} \quad \hat{u}(s) \overset{\mathscr{M}_{\kappa_0}}{\longmapsto} \hat{U}(z)\,. \tag{6.20}$$

The Laplace transform $\mathscr{L}$ maps the real axis to another straight line through the origin in the complex plane and $\mathscr{M}_{\kappa_0}$ maps the line onto the boundary of the unit disk. If there is no ambiguity we drop the index κ_0 of $\mathscr{M}$. Given a function $u(x):\mathbb{R}^+ \to \mathbb{C}$ we will denote its $\mathscr{M}\mathscr{L}$-transform as $\hat{U}(z)$. Applied to a univariate plane wave with wave number k and amplitude $u_0 \in \mathbb{C}$ this chain of transformations looks as follows

$$u_0 e^{ikx} \quad \overset{\mathscr{L}}{\longmapsto} \quad \frac{u_0}{s-ik} \quad \overset{\mathscr{M}_{\kappa_0}}{\longmapsto} \quad \frac{-iu_0}{\kappa_0(z+1)-k(z-1)}\,. \tag{6.21}$$

This results in a finite element formulation [61] which for the interior domain of interest keeps the usual notion of approximating the wave function as a function of position in space whereas on those parts of the boundary representing the cut-off cross sections of the leads a different set of tensor-product polynomials is used. For the longitudinal direction they may be thought of as modeling the behavior in a generalized frequency domain while keeping the spatial dependency for the transverse direction. An important feature of HSEs is the ability to correctly take into account constant magnetic fields.

The most important tool for converting the integrals over the semi-infinite domain of the leads into integrals over some finite domain is the following identity [100] based on Cauchy's integral theorem

$$\begin{aligned}(u,v)_{\mathbb{R}^+} &= \int_0^\infty u(r)v(r)dr = \frac{1}{2\pi i}\int_{\kappa_0\mathbb{R}} \hat{u}(s)\hat{v}(s)ds \\ &= \frac{\kappa_0}{i\pi}\int_0^{2\pi} \hat{U}\left(e^{-i\theta}\right)\hat{V}\left(e^{i\theta}\right)d\theta = \frac{\kappa_0}{i\pi}\left(\hat{U},\hat{V}\right)_{S^1}\,.\end{aligned} \tag{6.22}$$

Mapping the tilted real axis to the unit circle S^1 is achieved by the Möbius transformation

$$\begin{aligned} \varphi(z) &:= i\kappa_0 \frac{z+1}{z-1}\,, \qquad z = e^{-i\theta}\,, \Re\kappa_0 > 0\,, \\ (\mathscr{M}f)(z) &:= \frac{1}{z-1} f(\varphi(z))\,, \quad f : \kappa_0\mathbb{R} \to \mathbb{C}\,. \end{aligned} \tag{6.23}$$

Due to the symmetry properties of the exterior domain $\Omega_{ext} = \mathbb{R}^+ \times \partial\Omega_c = \mathbb{R}^+ \times [0,1]$ we can make a separation ansatz factoring out the longitudinal direction x

$$\Psi(x,y) = \psi(y)u(x)\,, \tag{6.24}$$

$$\Phi(x,y) = \phi(y)v(x) \tag{6.25}$$

and apply the $\mathscr{M}\mathscr{L}$-transform to the longitudinal direction

$$\mathscr{M}\mathscr{L}\Psi(x,y) = \psi(y)(\mathscr{M}\mathscr{L}u)(z)\,, \tag{6.26}$$

$$\mathscr{M}\mathscr{L}\Phi(x,y) = \phi(y)(\mathscr{M}\mathscr{L}v)(z)\,. \tag{6.27}$$

Let us first apply the HSE method to a simplified case in which there is no vector potential in the leads (independent of the issue of how to realize this in an experiment). Due to the separation ansatz and the fact that on Ω_{ext} the potential only depends on the local transverse coordinate y, cf. Fig. 6.2, the scalar products factorize

$$\begin{aligned} (\nabla\Phi, \nabla\Psi)_{\Omega_{ext}} + (\Phi, [V-E]\,\Psi)_{\Omega_{ext}} &= (\phi_y, \psi_y)_\Gamma (u,v)_{\mathbb{R}^+} \\ &\quad + (\phi, \psi)_\Gamma (u_x, v_x)_{\mathbb{R}^+} + (\phi, [V-E]\,\psi)_\Gamma (u,v)_{\mathbb{R}^+}\,. \end{aligned} \tag{6.28}$$

Derivatives with respect to a coordinate are indicated by indices. By applying Eqs. (6.26) and (6.27) we get as weak formulation of the transparent boundary condition

$$\frac{i\pi}{\kappa_0} b(\Phi,\Psi) = \sum_{c\in\{\text{contacts}\}} \left(\left\{ (\phi_y, \psi_y)_{\Gamma_c} + (\phi, [V-E]\,\psi)_{\Gamma_c} \right\} \left(\hat{U}, \hat{V}\right)_{S^1} + (\phi, \psi)_{\Gamma_c} \left(\hat{U}_x, \hat{V}_x\right)_{S^1} \right)\,. \tag{6.29}$$

6.3.2 Incorporation of Magnetic Fields

In contrast to [101] we additionally have to incorporate the magnetic field into the boundary conditions. The Landau gauge $\mathbf{A} = (-y,0,0)$ facilitates this task because: (i) On boundaries perpendicular to the vector potential only the terms due to the transverse coordinate have to be modified. (ii) in this gauge the vector potential can always be regauged transparently [22] so that it is oriented parallel to the lead's direction and thus perpendicular to the interface, i.e. in the lead's coordinate system it is again of the form $\mathbf{A}(\mathbf{x}) = B(-y,0,0)^T$. Then, the boundary term $b(\cdot,\cdot)$ has to be augmented by

$$\frac{2B\kappa_0}{\pi} \sum_{c\in\{\text{contacts}\}} \left(\hat{U}_x, \hat{V}\right)_{S^1} (\phi, -y\psi)_{\Gamma_c}\,. \tag{6.30}$$

In order to use the standard FEM approach we still need to define a finite dimensional subspace of the functions on the unit circle. This is accomplished by using the span of a finite set of complex trigonometric monomials restricted to the unit circle.

$$\{z^0, z^1, \dots, z^{n_H}\}, \quad z = e^{i\theta}, \theta \in [0, 2\pi)\,. \tag{6.31}$$

For details see [61, 101].

6.3.3 Finite Element Discretization

As shown in [100] a $\mathscr{ML}$-transformed function $\hat{F}$ can still be decomposed into a solely interface-dependent part $f_0 := f(x_{out})$ and a pure volume part $F \in H^+(S^1)$ describing the behavior in the open domain $\Omega_{ext} \backslash \Gamma$. As it plays a role whether or not we transform a function f or its radial derivative f' we have two rules how to represent this decomposition

$$\hat{F}(z) = \frac{1}{2i\kappa_0}[f_0 + (z-1)F(z)]\,, \tag{6.32}$$

$$\hat{F}'(z) = \frac{1}{2}[f_0 + (z+1)F(z)]\,, \tag{6.33}$$

$$F(z) := \frac{2i\kappa_0 \hat{F}(z) - f_0}{z-1}\,. \tag{6.34}$$

Because of the limit theorem for Laplace transformations a $\mathscr{ML}$-transformed function $\hat{F}$ has the important property [100, Lemma 4.13]

$$f_0 = f(x_{out}) = 2i\kappa_0 \hat{F}(1)\,, \tag{6.35}$$

$$f_0' = f'(x_{out}) = -4i\kappa_0 \frac{d\hat{F}}{dz}(1) - 2i\kappa_0 \hat{F}(1)\,. \tag{6.36}$$

As cell T for our finite element we use the unit circle S^1 in the complex plane extruded in the direction of an adjacent edge $\mathbf{e} \subset \Gamma$ on the interface Γ to the interior domain Ω. In the following we will call this a *Hardy cylinder*. The polynomial space P_T defined on this cell T is a tensor product formed by the polynomial space $V_h^{\mathbf{e}}$ containing the traces of the interior shape functions restricted to the edge $\mathbf{e} \subset \Gamma$ with dimension m, the number of degrees of freedom (DoFs) on an edge, and the discretized Hardy space for the radial direction which itself is a tensor product of $\mathbb{C}$ and the space Π_n of complex polynomials up to degree n. Its DoFs are the polynomial degrees and the function value of the wave function at the interface

$$T = \mathbf{e} \times S^1\,, \tag{6.37}$$

$$P_T = (\mathbb{C} \times \Pi_n) \times V_h^{\mathbf{e}}\,, \tag{6.38}$$

$$dim\, P_T = (1 + n + 1)m\,. \tag{6.39}$$

The shape functions on a Hardy cylinder are given by discretizing the separation ansatz

$$b_{jk} : P_T \to \mathbb{C}\,, \qquad b_{jk}(y,z) = \varphi(y)_j \hat{b}_k(z)\,. \tag{6.40}$$

Recalling the basic properties of Hardy spaces a natural choice of a finite-dimensional subspace of $H^+(S^1)$ in which an approximation F_h of F might be sought is given by using the span of the first $N+1$ trigonometric polynomials. Thus, we can expand $\hat{F}_h$ and $\hat{F}_h'$ as

$$i\kappa_0 \hat{F}_h(z) = \alpha_{-1}\hat{b}_{-1}(z) + (z-1)\sum_{k=0}^{N}\alpha_k z^k = \sum_{k=-1}^{N}\alpha_k \hat{b}_k(z)\,, \tag{6.41}$$

$$\hat{F}_h'(z) = \alpha_{-1}\hat{b}_{-1}'(z) + (z+1)\sum_{k=0}^{N}\alpha_k z^k = \sum_{k=-1}^{N}\alpha_k \hat{b}_k'(z) \tag{6.42}$$

in order to approximate their exact values as given in Eqs. (6.32) and (6.33). This leads us to a non-orthogonal set of shape functions $\hat{b}_k$ for the function values and $\hat{b}_k'(z)$ for the derivative values, respectively. Their explicit expressions are

$$\hat{b}_{-1}(z) = \frac{1}{2}\,, \tag{6.43}$$

$$\hat{b}_k(z) = \frac{1}{2}z^k(z-1) \quad 0 \le k \le n\,, \tag{6.44}$$

$$\hat{b}_{-1}'(z) = \frac{1}{2}\,, \tag{6.45}$$

$$\hat{b}_k'(z) = \frac{1}{2}z^k(z+1) \quad 0 \le k \le n\,. \tag{6.46}$$

Denoting the first degree of freedom by a negative index may be a unusual, but becomes evident when looking at the resulting matrices from this finite element method. The first DoF is the one on the interface and by convention is counted as an interior DoF. As, on the other hand, it is also addressable from the exterior it gets as index -1 to indicate that one grabs into the last column of the matrix block representing the discretization of the interior.

Having defined the shape functions we can compute the matrices by making the following ansatz for the discretized wave function in the exterior

$$\hat{\Psi}_h(y,z) = \sum_T \sum_{j,k\in T} \hat{\Psi}_{jk}^T b_{jk} = \sum_T \sum_{j,k\in T} \hat{\Psi}_{jk}^T \varphi(y)_j \hat{b}_k(z)\,. \tag{6.47}$$

Note that the indices i and j represent the local enumeration of the shape functions on a Hardy cylinder T. Plugging this ansatz into the exterior part of the variational form, Eq. (6.29), and testing with b_{il} gives the entry of the global system matrix

$$\mathbf{H}_{il,jk} = -2i\kappa_0 \sum_T \sum_{j,k\in T} \left(\nabla b_{il}, \hat{\Psi}_{jk}^T \nabla b_{jk}\right)_T + 2i\kappa_0 E \sum_T \sum_{j,k\in T} \left(b_{il}, \hat{\Psi}_{jk}^T b_{jk}\right)_T \tag{6.48}$$

$$\begin{aligned} &= -2i\kappa_0 \sum_T \sum_{j,k\in T} (\varphi_i,\varphi_j)_\Gamma \left(\hat{b}_l', \hat{b}_k'\right)_{S^1} \hat{\Psi}_{jk}^T \\ &\quad -2i\kappa_0 \sum_T \sum_{j,k} \left(\varphi_i', \varphi_j'\right)_\Gamma \left(\hat{b}_l, \hat{b}_k\right)_{S^1} \hat{\Psi}_{jk}^T \\ &\quad +2i\kappa_0 E \sum_T \sum_{j,k} (\varphi_i,\varphi_j)_\Gamma \left(\hat{b}_l, \hat{b}_k\right)_{S^1} \hat{\Psi}_{jk}\,. \end{aligned} \tag{6.49}$$

In these expressions the individual matrix entries from the interface contributions are readily identified. They are formed by the traces of the interior finite elements on the interface Γ between the finite and infinite domain and will be denoted by

$$\mathbf{M}_\Gamma = (\varphi_i, \varphi_j)_\Gamma , \tag{6.50}$$

$$\mathbf{A}_\Gamma = (\varphi_i', \varphi_j')_\Gamma . \tag{6.51}$$

For the radial Hardy space contributions we analogously denote the entries of the mass and stiffness matrices as

$$\hat{\mathbf{M}} = (\hat{b}_l, \hat{b}_k)_{S^1} , \tag{6.52}$$

$$\hat{\mathbf{A}} = (\hat{b}_l', \hat{b}_k')_{S^1} = \frac{1}{2\pi}\int_{S^1} \hat{b}_l'(z)\hat{b}_k'(\bar{z})\ |dz| . \tag{6.53}$$

where we have suppressed the indices at the entries of the matrices **M** and **A**. Going back to Eq. (6.48) and using tensor notation we get for the exterior part of the global system matrix

$$\mathbf{H}_{ext} = -2i\kappa_0 \left[\mathbf{M}_\Gamma \otimes \hat{\mathbf{A}} + \mathbf{A}_\Gamma \otimes \hat{\mathbf{M}} - E\mathbf{M}_\Gamma \otimes \hat{\mathbf{M}}\right] \hat{\boldsymbol{\Psi}} . \tag{6.54}$$

6.3.4 Computation of the Matrix Entries

The entries of $\mathbf{H}_{ext}$ are obtained from analytical evaluation of Eqs. (6.52) and (6.53) in which we parameterize the unit circle by $z = e^{i\theta}$. For coupling on the interface we get

$$\hat{\mathbf{M}}_{-1,-1} = \frac{1}{4} = \hat{\mathbf{A}}_{-1,-1} . \tag{6.55}$$

The interactions of interface DoF and volume shape functions $\hat{b}_k(z)$, $k \geq 0$ are

$$\begin{aligned}\hat{\mathbf{M}}_{-1,k} &= \frac{1}{2\pi}\int_{S^1} \frac{1}{4}\left(e^{i(0-(k+1))\theta} - e^{i(0-k)\theta}\right)\, d\theta \\ &= \frac{1}{4}(\delta_{0,k+1} - \delta_{0,k}) = -\frac{1}{4}\delta_{0,k} \quad k \geq 0 \end{aligned}\tag{6.56}$$

$$\hat{\mathbf{A}}_{-1,k} = \frac{1}{4}(\delta_{0,k+1} + \delta_{0,k}) = +\frac{1}{4}\delta_{0,k} \quad k \geq 0 \tag{6.57}$$

For the truncated part of the leads we have to compute the interactions among the volume shape functions $\hat{b}_k(z)$, $j,k \geq 0$, which amount to

$$\begin{aligned}\hat{\mathbf{M}}_{j,k} &= \frac{1}{2\pi}\int_{S^1} \frac{1}{4}\left(e^{i(j+1)\theta} - e^{ij\theta}\right)\left(e^{-i(k+1)\theta} - e^{-ik\theta}\right)\, d\theta \\ &= \frac{1}{2\pi}\int_{S^1} \frac{1}{4}\left(e^{i(j-k)\theta} - e^{-i(j-(k-1))\theta} e^{-i(j-(k+1))\theta} - e^{-i(j-k)\theta}\right)\, d\theta \\ &= \frac{1}{4}(-\delta_{j,k-1} + 2\delta_{j,k} - \delta_{k,j+1}) ,\end{aligned}\tag{6.58}$$

$$\hat{\mathbf{A}}_{j,k} = \frac{1}{4}(\ \delta_{j,k-1} + 2\delta_{j,k} + \delta_{k,j+1}) . \tag{6.59}$$

In summary, the matrix representation of the radial part of the hardy space boundary conditions is given by the matrices

$$\hat{\mathbf{M}} = \frac{1}{4}\begin{pmatrix} +1 & -1 & & & & & & \\ -1 & +2 & -1 & & & & & \\ & -1 & +2 & -1 & & & & \\ & & -1 & +2 & -1 & & & \\ & & & & \cdots & & & \\ & & & & -1 & +2 & -1 \\ & & & & & -1 & +2 \end{pmatrix} \tag{6.60}$$

$$\hat{\mathbf{A}} = \frac{1}{4}\begin{pmatrix} 1 & 1 & & & & & \\ 1 & 2 & 1 & & & & \\ & 1 & 2 & 1 & & & \\ & & 1 & 2 & 1 & & \\ & & & & \cdots & & \\ & & & & 1 & 2 & 1 \\ & & & & & 1 & 2 \end{pmatrix} \tag{6.61}$$

6.4 Software Validation on the Quantum Wire

For validating our methods we consider the simple quantum wire and study the different aspects of the quality of the numerical solution individually. All computations have been based on the finite element library *deal*.II [2]. To show the usefulness of the new method for unbounded domains we simulate the quantum transport through a semi-open corner device as used in magnetic focussing experiments and which is discussed in [95]. In the simulation the domain of the device is truncated to roughly $5 \times 5\mu$m which contains the experimentally interesting part. This is possible only by imposing transparent boundary conditions.

Dependence on HSE parameters

We did not find any significant effect of the HSE parameters, i.e. κ_0 and n_H, on the current conservation, provided the method is correctly employed, i.e. κ_0 defines a line that does not intersect with the spectrum of the Hamiltonian. Therefore, in the following we rather focus on the effects on the numerical error induced by using different finite elements and mesh widths.

Perfect Lead

Figure 6.3 shows a contour plot of the electron density $|\Psi|^2$ in an empty channel for two different polynomial degrees of the finite elements. Since the channel is empty we have as analytic solution for a wave incident from the left $\Psi(x,y) \propto sin(\pi y)e^{ikx}$ and $|\Psi|^2 \propto sin(\pi y)$, i.e. the electron density does not have any explicit dependence on the longitudinal coordinate x as it is to be expected. We chose $L_x = 6.9$ as length in x-direction and $L_y = 1$ as width. For linear elements we chose mesh widths $h_x = .121$ and $h_y = .125$. For the quartic elements we used $h_x = .484$ and $h_y = .5$. Thus, in both cases we had 9 degrees of freedom (DoFs) on the width of the channel and 1026 DoFs in total. As Fig. 6.3 shows, at this resolution the linear elements

[2]www.dealii.org

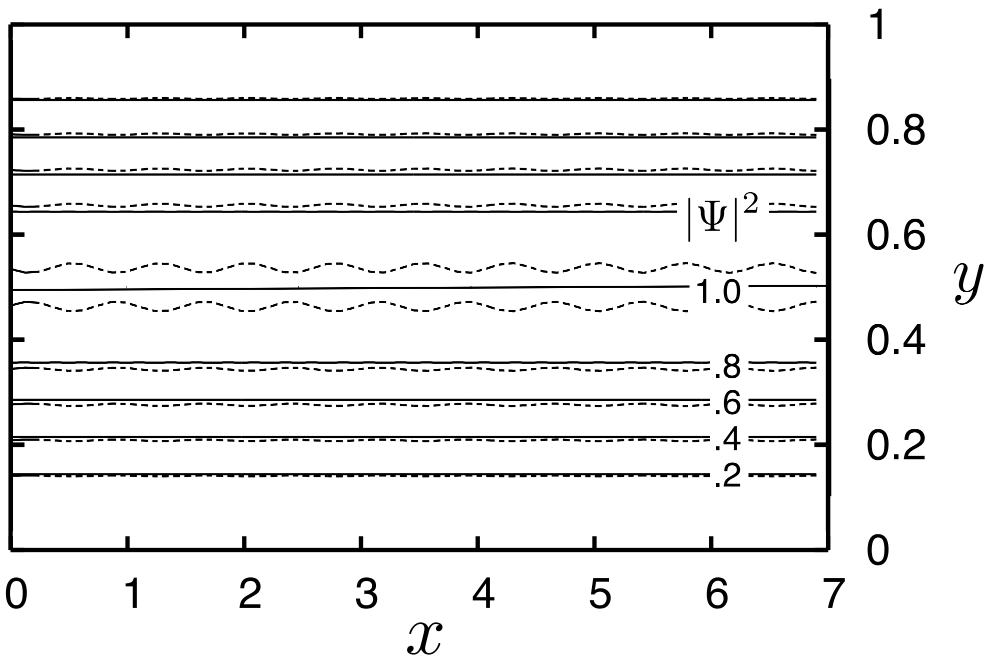


Figure 6.3: Higher order FEM solution (solid) vs. linear elements (dashed). Contour lines represent electron density $|\Psi|^2$, $E_F = 28$, $L_x = 6.9$, $n_{dofs} = 1026$.

still produce some wiggling structure in x-direction whereas the quartic elements produce a constant profile, i.e. real and imaginary part of the wave function have the correct phase shift of $\pi/2$. This could be further quantified by a more thorough convergence analysis but the picture norm reveals the physical consequences of the effect better.

Perfect Lead with Potential Barrier

The quality of the current conservation in the presence of a scatterer is studied next. The results are shown in Figs. 6.4 and 6.5. We use a simple step barrier of height $U = 20$ and length $L = 2.25$ in a section of an infinite wire with domain $\Omega_{int} = [0, x_{out}] \times [0, 1]$ with $x_{out} = 6$, see inset of Fig. 6.4. The conductance is computed from Eq. (6.7) and depicted in Fig. 6.4 together with the corresponding error in the current conservation for $R_0 + T_0$ as function of the Fermi energy and shows the expected quantization and oscillations near its jumps as it can be found in standard textbooks. The steps in the conductance reflect the number of open modes whereas the oscillatory behavior is due to the barrier length.

To compare the quality of the current conservation three combinations of finite elements and mesh refinement which all lead to the same number of DoFs have been used. The underlying coarse grid consists of six cells of size 1×1. Since *deal*.II for two-dimensional computations only provides quadriliteral cells all finite elements are constructed from tensor products of finite elements of intervals. For linear elements (Q1) the mesh was refined five times so that the final mesh width was $h = 1/32$. For quadratic elements (Q2) which (away from the domain boundaries) have two degrees of freedom per coordinate direction per cell the meshwidth was $h = 1/16$. Similarly, quartic elements (Q4) have 4 degrees of freedom per coordinate direction

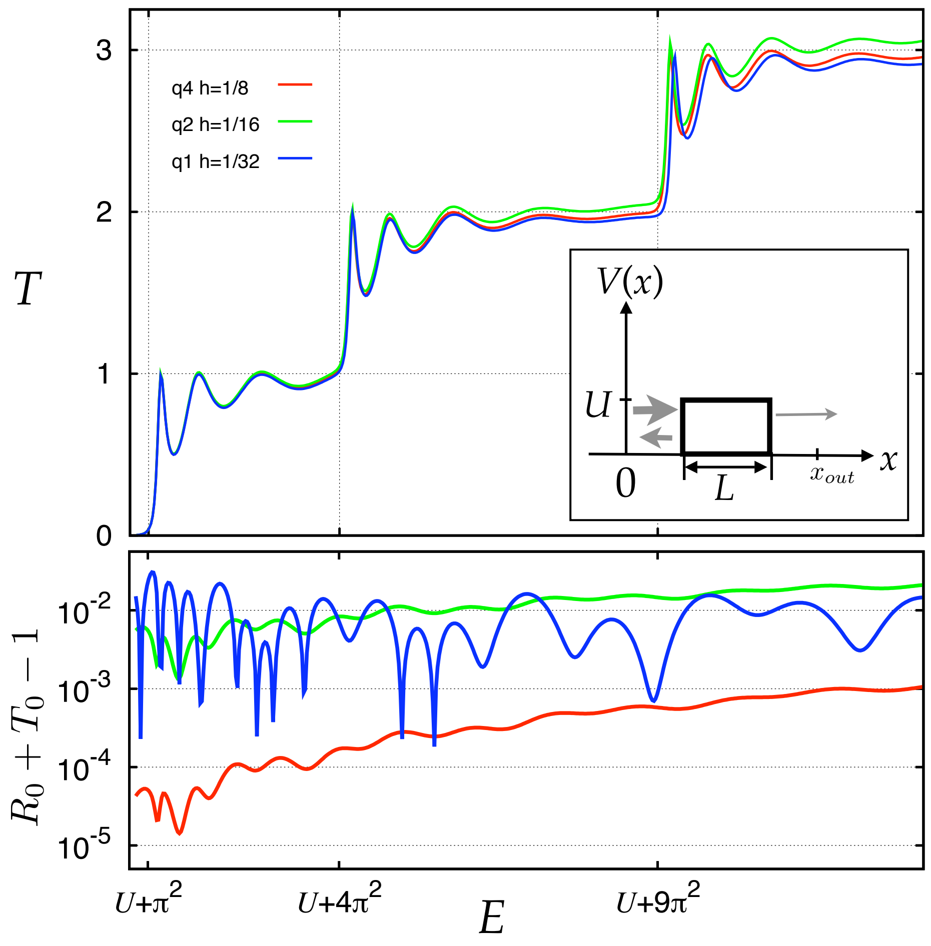


Figure 6.4: Top: Transmission T vs. energy E for various mesh widths h and finite elements for a fixed total number of DoFs. Inset: Sketch of system. Bottom: Current conservation for the corrsponding ground state mode.

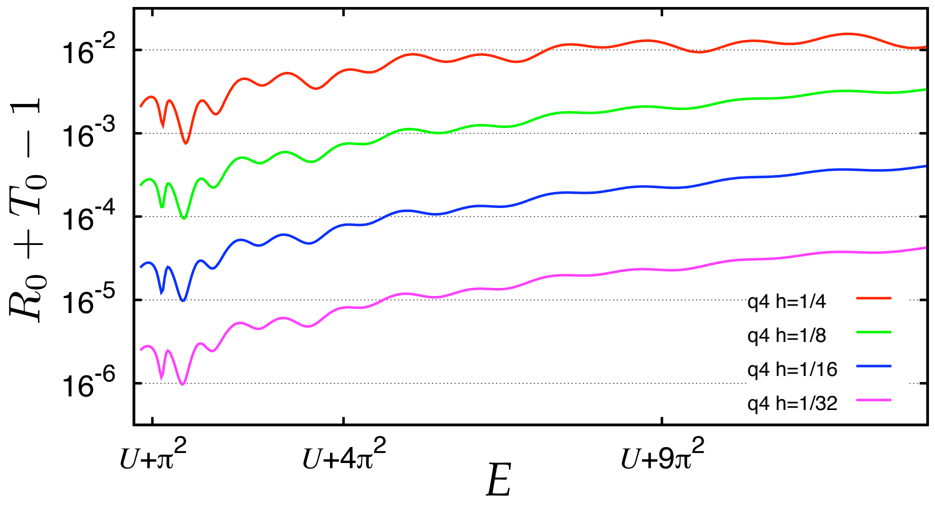


Figure 6.5: Consistent with finite element theory the error of the current conservation scales like h^4 for quartic elements.

per cell and require only $h = 1/8$ to obtain the same number of DoFs.

The lower part of Fig. 6.4 clearly shows that higher order elements pay off by two effects. On the one hand, the error behavior for elements of even order is much smoother than for elements of odd order, especially linear ones. On the other hand, although quartic elements were employed on the coarsest grid the error in the current conservation is by almost two orders of magnitude smaller than for quadratic elements. This can be understood by looking at the trace theorems from finite element theory [81] which predict that the error in Eq. 6.5 induced by replacing the exact wave function by the numerically computed one should scale like h^q where q is the polynomial order of the element. Hence, in case of quartic elements a global refinement of the mesh by a factor of 2 should reduce the error by a factor of 16. This is highlighted in Fig. 6.5 by choosing a $\log_{16}$ scale for the abscissa.

6.5 Magnetic Focussing in 2DEGs

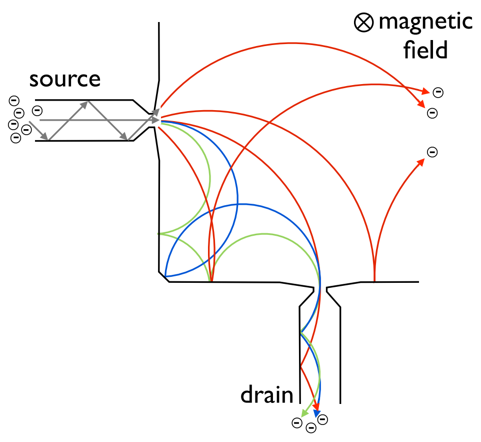


Figure 6.6: Classical electron trajectories in the magnetic focusing device studied by Metzger [95]. Colors indicate different strengths of the magnetic field.

Electrons moving in a magnetic field $\mathbf{B}$ with velocity $\mathbf{v}$ are subject to the Lorentz force

$$\mathbf{F}_L = q(\mathbf{v} \times \mathbf{B}) . \qquad (6.62)$$

For electrons the charge q is negative. Hence, electrons moving from left to right in Fig. 6.6 curve downward if the magnetic field points into the plane of the figure. From the balance of centripetal and Lorentz force we can compute the classical cyclotron radius

$$R_c = \frac{m|\mathbf{v}|}{q|\mathbf{B}|} . \qquad (6.63)$$

In the absence of a potential electrons of energy $E = m\mathbf{v}^2/2$ hit the drain if $R_c = R$, cf. Fig. 6.7. For this to happen the magnetic field strength must be

$$B_0 = \frac{\sqrt{E}}{R} . \qquad (6.64)$$

As units for B_0 we choose the dimensionless ones introduced in Sec. 6.2.2. In a quantum mechanical description of a moving electron subject to a constant magnetic field perpendicular to the plane of motion the classical behavior should be recoverable from the wave function Ψ. When plotting its modulus as in Fig. 6.11 the classical trajectories should appear as local maximums in areas where no self-interference occurs. In areas where the wave function interferes with itself due to reflections at nearby hard walls the wave trains should be perpendicular to the local direction of the trajectories. Similar to the classical setting only for selected values of the magnetic field strength the electronic wave function should be able to penetrate from the bulk into the drain. For geometric reasons this should happen mainly at integer multiples of B_0. Measuring the quantum mechanical current, Eq. (6.5) at the terminal cross section of the drain and plotting its normalized form, Eq. (6.6) against the magnetic field measured in units of B_0 gives the magneto-conductance curves in Figs. 6.8-6.14. Peaks indicate that a transmission of electrons from source to drain has taken place at the corresponding value of the magnetic field. When scaling the magnetic field with B_0 the transmission becomes a universal feature which does not depend on the absolute size of the device anymore. Due to the fact that electrons can leave the source under different angles there is a multitude of classical trajectories leaving the source and the peak height is a measure for how many classical trajectories lead from source to drain. This allows to construct a probabilistic theory for the transmission based on the classical trajectories which then can be compared to the quantum theory. A representative selection of results for the magneto-conductance and associated electron densities are shown in Figs. 6.8-6.14 for the system sketched in Fig. 6.7. It models a magnetic focusing device previously studied by Metzger [95] who computed the conductance from an ensemble of classical single-electron trajectories.

Before we can simulate the quantum mechanical properties of the device sketched in Fig. 6.7 by solving the Schrödinger equation (6.10) for fixed energies E to get the scattering states Ψ_p for the various input modes we have to check whether we can trust the Hardy space transparent boundary conditions, i.e. physical results do not depend on the method's parameters.

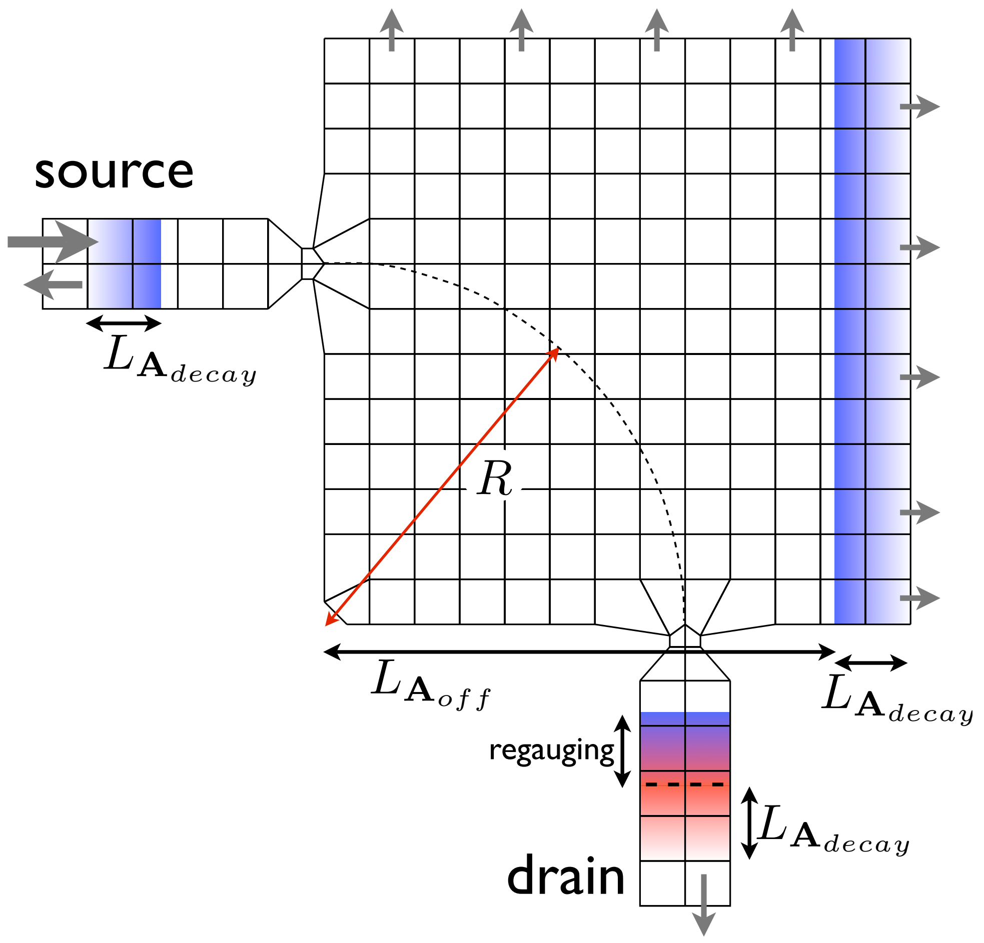


Figure 6.7: Coarse grid of the focusing device. Grey arrows indicate in- and outbound waves on the transparent boundaries. Color gradients denote the areas where the vector potential is turned off. In the outlet this must be preceeded by a regauging step.

6.5.1 Choice of Model parameters

Because of the size of the device used in experiments we only consider the physically interesting part of it, i.e. the lower left corner with dimensions of about 5x5μm. Much of the experimental effort is spent on connecting the leads to the bulk of the device by quantum point contacts (QPC) so that electrons entering the device from the exterior reservoirs have a well-defined electronic state and a very narrow distribution of their direction of motion. The QPCs connecting the wide leads to the bulk are modeled by narrowing the leads to a third of their width where they are attached to the bulk. In order to directly compute the wave function several modifications are necessary:

(i) Close to boundaries parallel to the vector field we turn off the vector potential smoothly in a strip of width $L_{\mathbf{A}_{decay}}$ (indicated by the color gradient ranging from blue or red to white in Fig. 6.7) such that no artificial reflections occur. This allows us to re-use the transparent boundary conditions from the Helmholtz case where the leads have been cut off. The vector potential merely constitutes an additional phase factor but does not change the energy of an electron which justifies this truncation. Using the continuous regauging method from [22] we can locally change the direction of the vector potential in the leads such that we subsequently may turn it off in a reflectionless manner (indicated by the color gradient ranging from blue to red in Fig. 6.7). This does not change the quantum mechanic flux through the leads' artificial cut off boundaries where we measure. Hence, we have restored a situation in which boundary conditions for the simpler Helmholtz case suffice.

(ii) To model the point-like contacts between the 2DEG and the wires the boundary is shaped like a shallow funnel on the side of the 2DEG.

(iii) The electron density n_{e^-} used in the magnetic focussing devices studied by Metzger in his thesis [95] is in the range of 2.2 to 2.5$\cdot 10^{11}$cm^{-2}. The electron density can be computed from the volume occupied by one state in k-space. In 2D this is given by $4\pi^2/L^2$ where L^2 is the area covered by the 2DEG. The volume of the Fermi sphere in 2D is given by πk_F^2. Including spin degeneracy the number of states in the Fermi sphere is given by $N_{k_F} = 2\pi k_F^2/(4\pi^2/L^2)$. Their density is obtained by dividing by the sample size again which yields $n_{e^-} = N_{k_F}/L^2 = k_F^2/2\pi = 2\pi/\lambda_F^2$. From this we can calculate the Fermi wavelength as a function of the electron density in cm $\lambda_F = \sqrt{2\pi/n_{e^-}}$. In the experiments the distance from the lower left corner to the centers of the QPCs, i.e. the cyclotron radius R, is 3μm. To minimize the computational effort it turned out that in computational units $R = 6.5a$ which corresponds to 13 cells in the coarse grid is an economic choice. Thus, 1cm $= 10^4 R/3$ where a is our length scale which emerged from the dimensional analysis in Sec. 6.2. Therefore, up to $\pm 10\%$ the Fermi energy in our dimensionless units is $E_F = k_F^2 = 2\pi n_{e^-}(3\cdot 10^{-4}/R_c)^2 \approx 3000$ and the Fermi wave length is $\lambda_F = 2\pi/\sqrt{E_F} \approx .12a$ or 4 wavelengths per coarse grid cell. The computations presented here have been performed for $E_F = 715.5$, i.e. $n_{e^-} \approx 6\cdot 10^{10}$cm^{-2}, which lies half-way between the energy needed for opening the 8th and 9th mode in the reservoir part of the leads, respectively.

Additionally, we provide the possibility to add a "door sill" potential V_{ds} in the constrictions to let only pass electrons with kinetic energy in the direction of the lead (long straight arrow in the source lead in Fig. 6.6). In the local coordinates of the leads the shape of V_{ds} is $V_{ds} = \alpha E_F \cos^2(\pi x/2L_{ds})$, $\alpha \in [0,1)$ and L_{ds} is a parameter measuring the width of the door sill. For $|x| > L_{ds}$ we set $V_{ds} \equiv 0$. The parameter L_{ds} is of the order of half of the size of a coarse grid cell. Electrons with a finite amount of kinetic energy for a movement transverse to the direction of a lead (zig-zagging arrows in Fig. 6.6) are reflected.

6.5.2 Independence of Method parameters

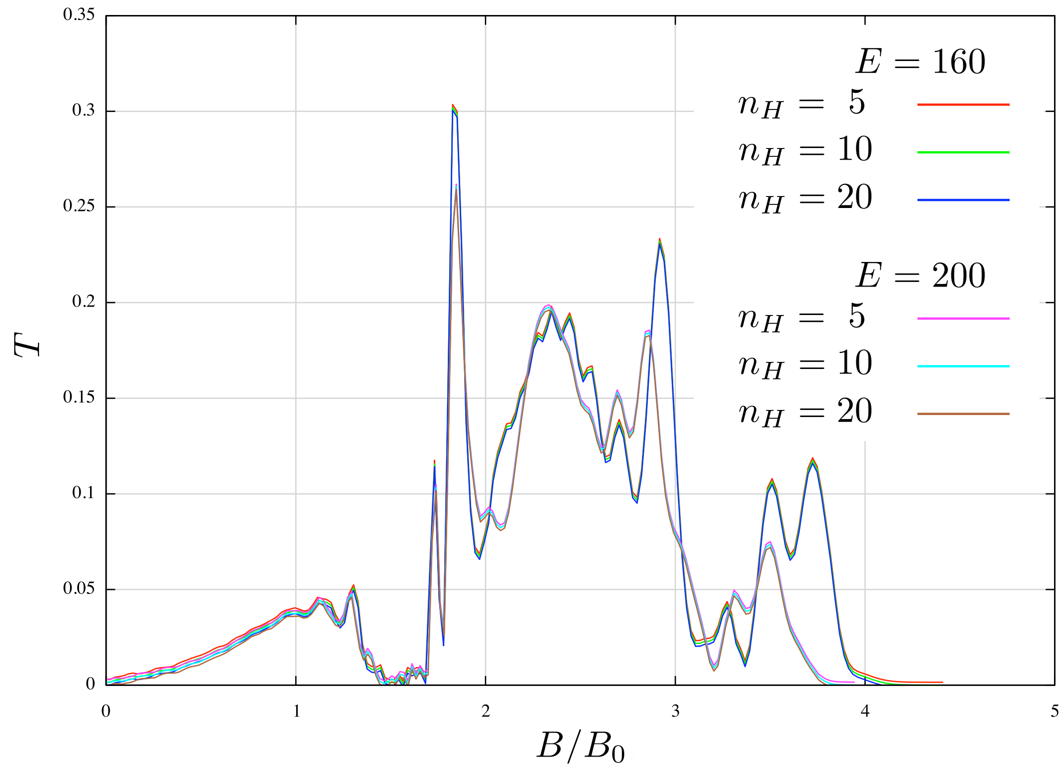


Figure 6.8: The number of DoFs of the Hardy space elements does not change the transmission.

Before assessing the physical properties of the device in more detail the dependence on the parameterization of the boundary conditions has to be tested. As in the case of a quantum wire the Möbius parameter κ_0 does not play a significant role as long as the corresponding axis in the complex plane does not intersect the spectrum of the Hamiltonian. The independence of the conductance of the number of Hardy-space DoFs n_H is demonstrated in Fig. 6.8. The Fermi energy, here denoted as E, and the n_H used in these tests is given in the legend of the figure. The overall system size is $L = 8$ and geometric parameters for the decay of the vector potential are $L_{\mathbf{A}_{off}} = 6.5$ and $L_{\mathbf{A}_{decay}} = .75$. To emphasize that for practical purposes the curves are identical the transmission values have been arbitrarily shifted by 0.0015 with respect to each other. At the northern boundary the cut off of the vector potential can be made sharp. For finite values of the magnetic flux the wave function almost vanishes close to boundary.

Varying the position of the subdomain in which the vector potential decays does not influence the transmission. The width $L_{\mathbf{A}_{decay}}$ slightly changes peak heights for values of the magnetic flux which are large compared to what the computational mesh can resolve, see Fig. 6.9. Nevertheless, the positions of the peaks do not change.

The dependence of transmission on energy is shown in Fig. 6.10. All simulations were done with the same spatial resolution. Rescaling of the magnetic field with B_0 reveals the universality of the peaks positions. For larger magnetic fields ($B/B_0 \gtrsim 3$) the mesh is not able to properly resolve the features of the wave function. Therefore, the last resolved peak of a transmission curve is considered to be dominated by numerical artifacts.

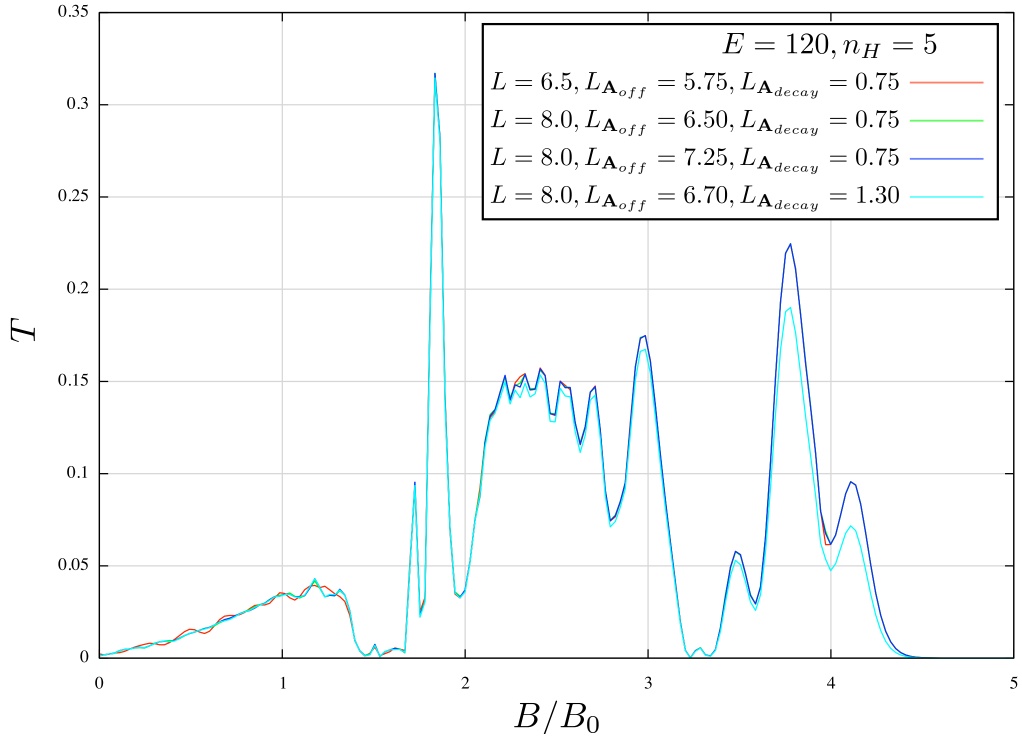


Figure 6.9: Transmission for different system sizes and cut off parameters $L_{\mathbf{A}_{off}}$ and $L_{\mathbf{A}_{decay}}$.

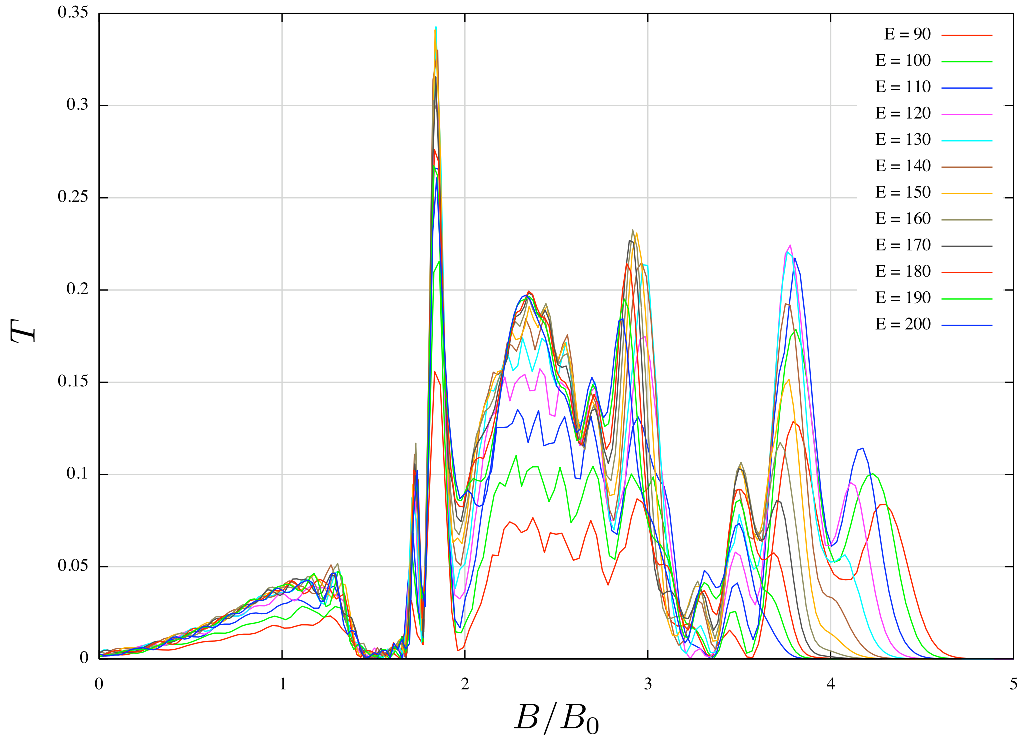


Figure 6.10: Energy dependence of the transmission as a function of magnetic flux. The bulk is of size 6.5x6.5. The strip in which the vector potential decays starts at $x = 5.75$ and ends at 6.5.

6.5.3 Electron Densities and Conductance

Figure 6.11 is the quantum mechanical complement to Fig. 6.6. It shows the modulus of the wave function Ψ_0 for four different values of the magnetic field strength. The modulus is the square root of the electron density and simpler to plot in a meaningful way. In the classical picture electrons enter through the lead on the left and, in case of transmission, leave through the lead on the right. The top left subfigure shows a situation which corresponds to the blue trajectory in Fig. 6.6. The bottom left figure is equivalent to the green trajectory. The top right represents a non-conducting state, indicated by the negligible modulus of the wave function in the source drain. In the subfigure on the lower right-hand side the magnetic field strength is such that the classical trajectory would make three contacts with the walls of the device before hitting the drain. This case also shows why a fixed mesh width leads to insufficiently resolved transmission peaks for high magnetic fields. High magnetic fields focus the wave function such that at some point the extent of the local maximums of the modulus perpendicular to the classical direction of motion is smaller than the diameter of a grid cell.

Transmission curves for a device with and without chamfered corner are shown in Fig 6.12. The chamfered corner moves the peak at $B/B_0 \approx 2$ to higher values of the magnetic field. This can be understood by considering the classical trajectories. The chamfered corner hinders the trajectories for $B/B_0 \lessapprox 2$ to hit the drain.

A purely quantum mechanical feature of the transmission curves are all the small wiggles between the main peaks which are located around the integer values of B/B_0. The wiggles are caused by the small local maximums in the electron density close to the hard walls which by increasing magnetic field strength get pushed into the drain one after another.

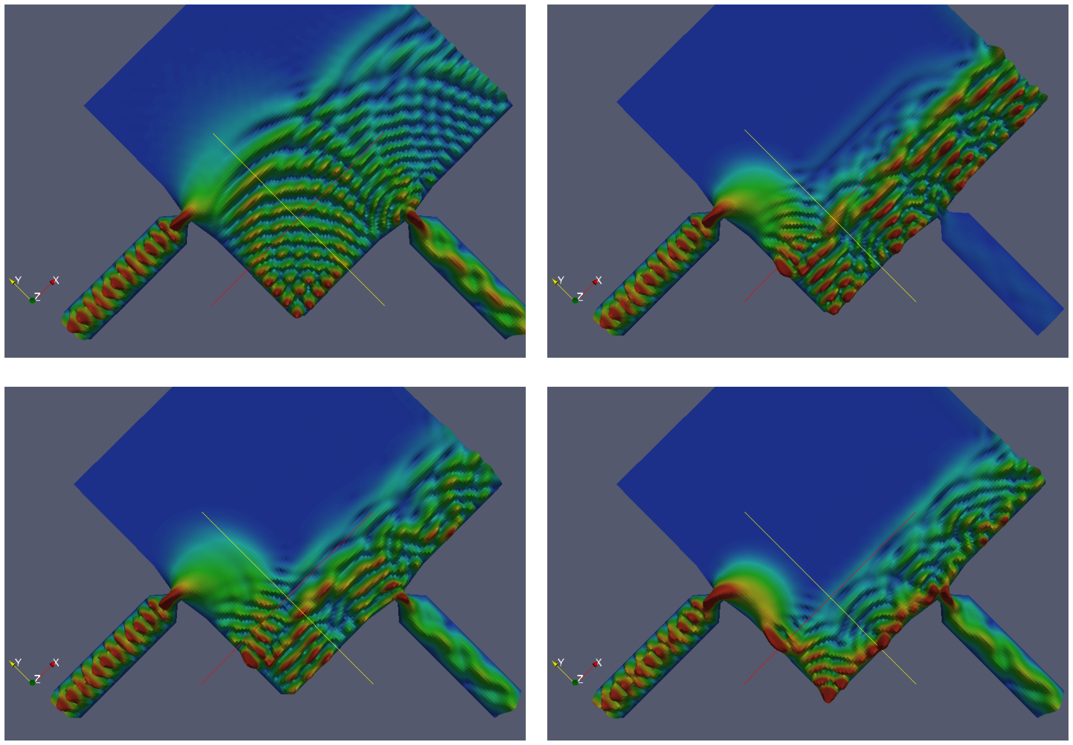
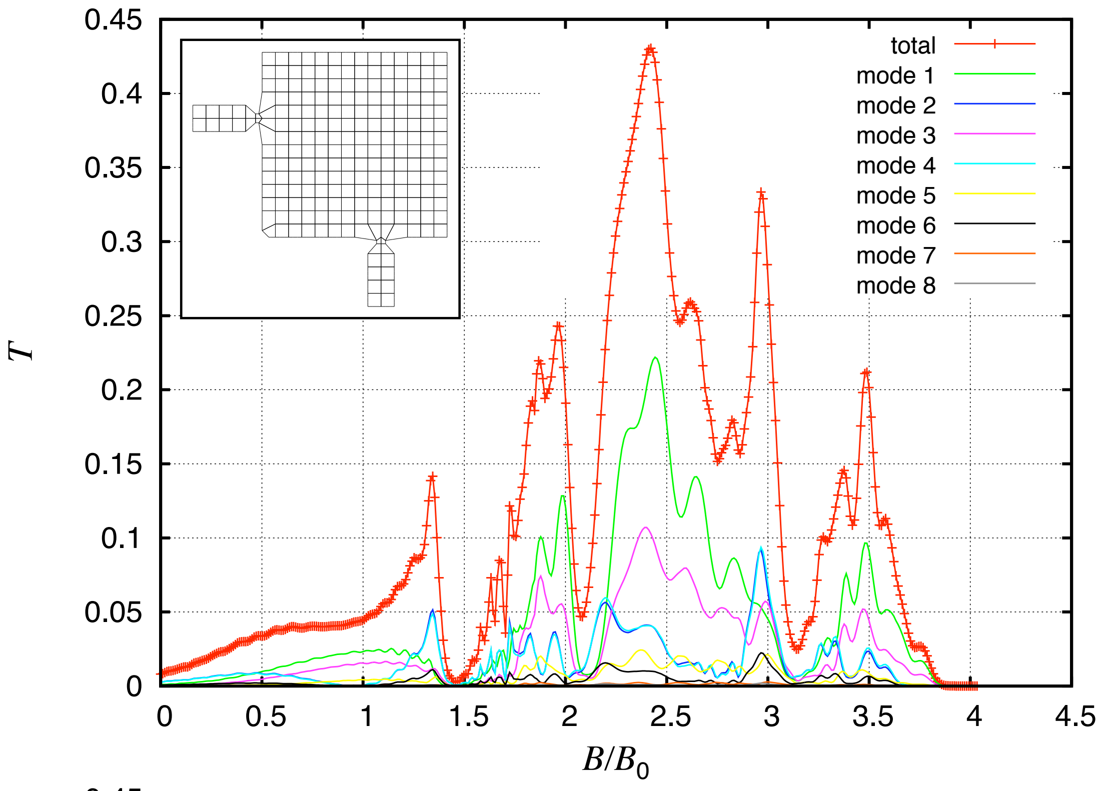


Figure 6.11: Modulus of Ψ_0. Top: Second magnetic focussing peak (left) and non-transmitting state at $B/B_0 \approx 3.5$ (right). Bottom: Third (left) and fourth (right) focussing peak.



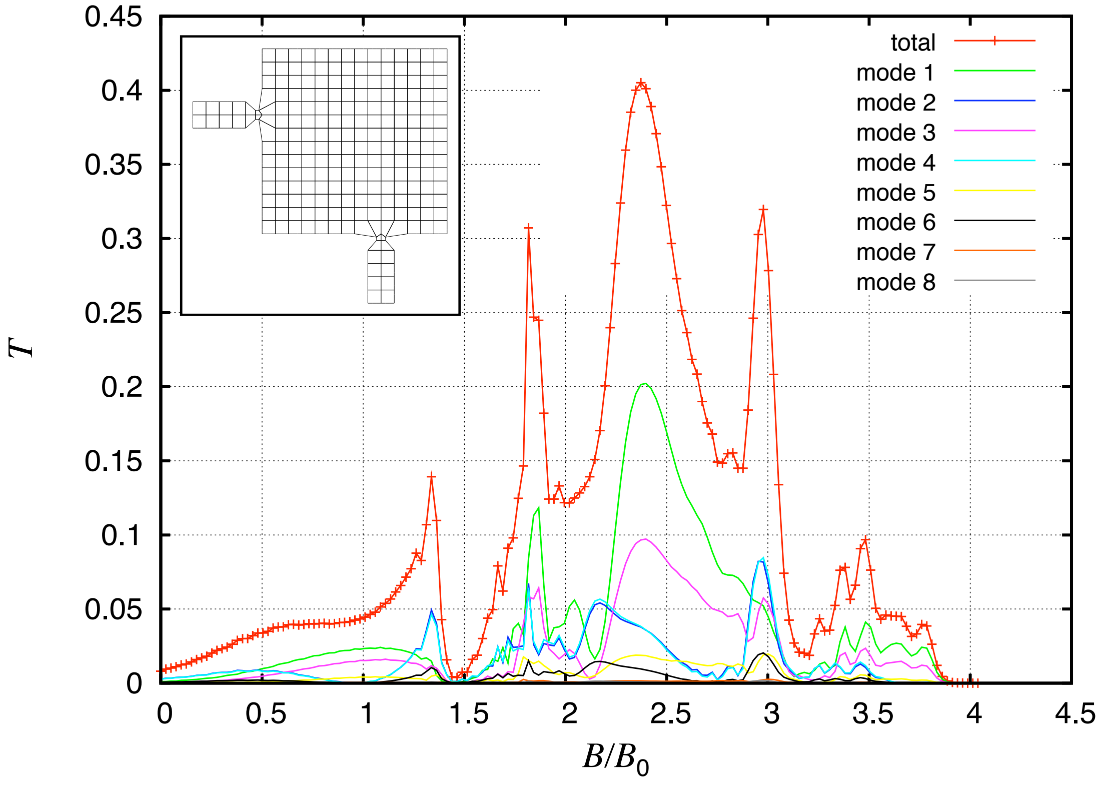


Figure 6.12: Transmission curves for all open input modes at $E = 715.5$, $L = 7$, in the clean sample and without barriers in contacts. Coarse meshes are shown in insets and differ only by the shape of the lower left corner.

Electron Beam Collimation in the Contacts

Adding local barrier potentials in the QPCs collimates the electron beam and suppresses the caustic peak located at $\sqrt{2}B_0$ and other peaks. By varying the height of the "door sill" potentials it can be figured out which peaks are mostly due to electrons leaving the source under a finite angle with respect to the lead axis. The suppression of the caustic peak agrees with the results from semi-classical methods [95]. Figure 6.13 shows the results for varying the height of the door-sill potentials (given in units of the Fermi energy in the legend) for $L = 9$, $R_c = 6.5$, $E_F = 715.5$ and no chamfered corner.

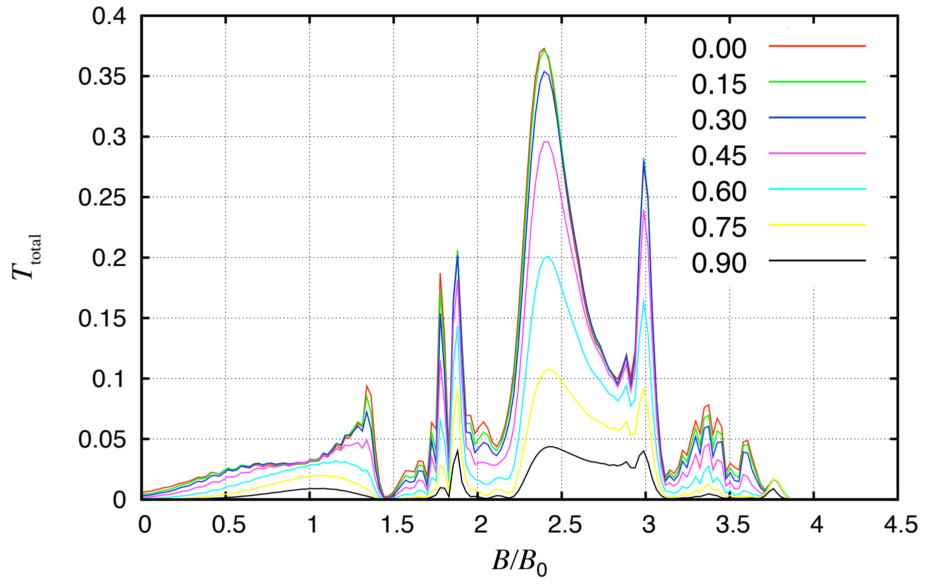


Figure 6.13: Suppression of caustic peaks by door-sill potentials.

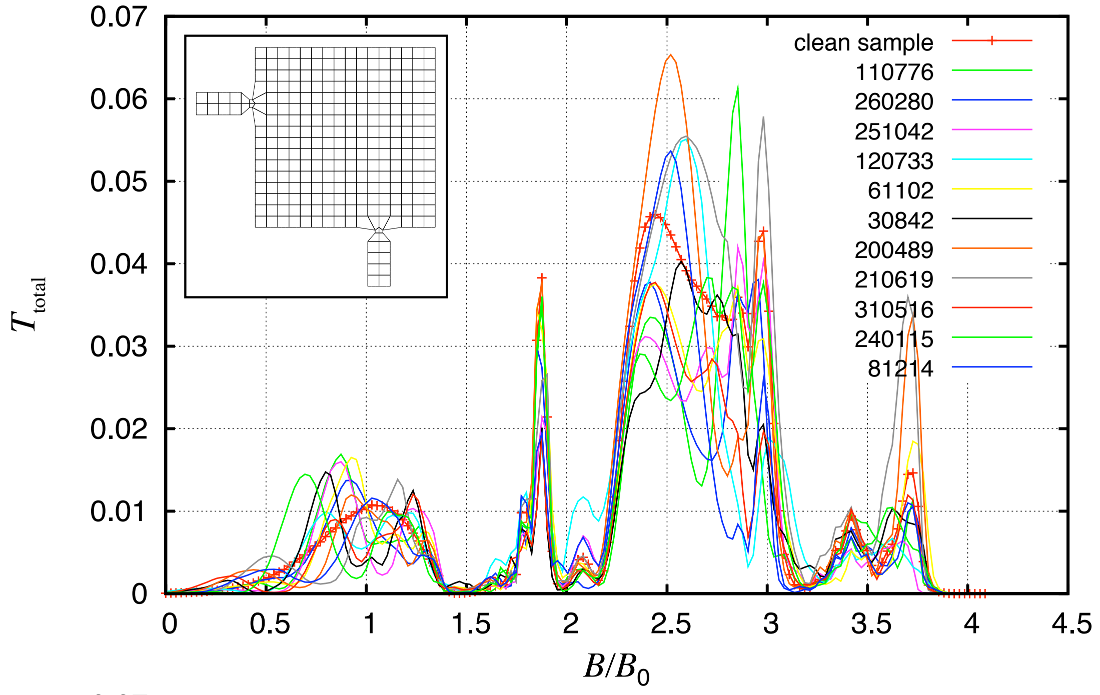
Influence of disorder

We close this chapter by first results on the influence of disorder on the magneto-conductance. A common source for a disorder potential in a semiconductor heterostructure the layer of donor atoms beneath the 2DEG. Its strength is measured in units of the Fermi energy E_F and common values are in the range of a few per cent E_F. A simple model for disorder is a superposition of N plane waves with random phases φ_i, wave numbers k_i, and directions α_i

$$V(x,y) = \frac{\gamma}{N}\sum_{i=1}^{N} \cos\left[k_i \cdot (x \cos\alpha_i + y \sin\alpha_i) + \varphi_i\right]. \tag{6.65}$$

The standard deviation of the potential can be adjusted by choosing the factor γ accordingly. Wave numbers are usually chosen from a narrow gaussian distribution around some mean wave number k_0. The correlation length l_c is proportional to $1/k_0$, cf. [118].

The transmission curves are given in Fig. 6.14 and have been measured for various realizations of a random plane wave potential with 50 waves and correlation length $l_c = .2275$. Each realization is identified by the seed used for the random number generator. The seed is given in the legends of the figure. The fixed parameters are $E = 715.5$, $B_0 = 4.863$ and $L = 8$. The chosen energy corresponds to a Fermi wave length $\lambda_F = .235$. The barrier height in the contacts is $0.9E$. The strength of the disorder potential is $0.02E$. The coarse meshes are given as insets. in agreement with [95] we observe a splitting of the first focusing peak at $B/B_0 = 1$. Our simulations show that the broad peak over the range $B/B_0 \in (2.4, 3)$ is also sensitive to the details of the realization of the disorder. The shape of the double peak at $B/B_0 = 2$ mainly depends on the presence of the chamfered corner but only little on the details of the disorder since around $B/B_0 \approx 2$ the conductance curves are all very close and almost fall on top of each other. What changes, though, is the peak height. Yet, in contrast to the odd-numbered peaks the position does not vary with the disorder.



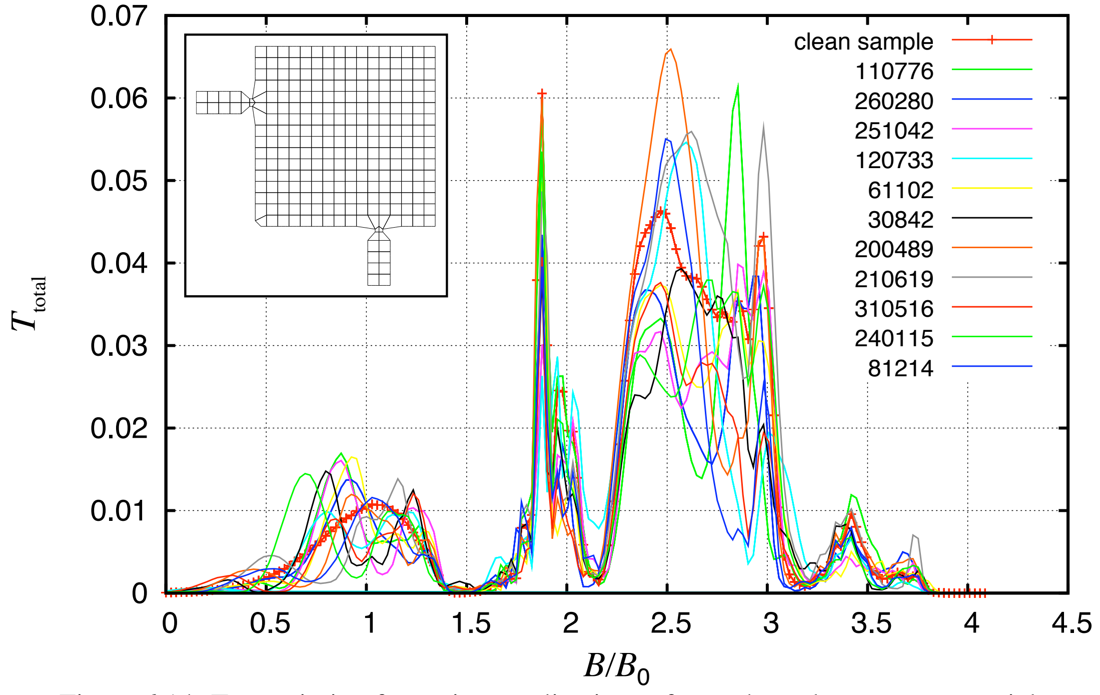


Figure 6.14: Transmission for various realizations of a random plane wave potential.

Chapter 7

Exponential Integrators for Quantum Time Propagation

This chapter is not so much about solving yet another problem from scratch. Instead it provides a blueprint for combining topics of previous chapters into a solver for a new problem class. The basis of quantum time propagation is a fast application of the Hamiltonian to a given state. For a numerical simulation this translates to the need of a fast evaluation of matrix vector products. This has been discussed in Sec. 1.2 and 3.3.1. The evolution of a quantum state over time can be described by a matrix exponential, at least symbolically. In a numerical setting the exponential will become a finite series for which we will put to good use the ideas behind the polynomial preconditioner from Sec. 3.2.4. Time-dependent quantum mechanics are often studied in systems which are much larger than what is numerically feasible. Hence, accurate transparent boundary conditions are needed in order to truncate the physically uninteresting parts of the domain which was the main theme of chapter 6. Last but not least, SciPAL's operator-based API allows for a concise implementation when assembling all the pieces into a simulation framework for time-dependent phenomena in disordered semiconductor heterostructures.

7.1 Quantum Time Propagation in a Nutshell

Quantum wave packet dynamics is another broad class of problems which dramatically profit from parallelization just by using a parallelized matrix-vector product. Well-known representatives of this class are molecular charge and energy transfer processes [94], electrodynamics of passive media, e.g. photonic materials [29], and ab-initio calculation of chemical reaction rates [97, 34]. Especially for the latter exponential integrators are considered as a good choice [142, 109]. Besides single-particle systems like in chapter 6 or in [78], the dynamics of dissipative or open quantum systems [62, 138] are frequently investigated by wave packet methods. From the propagation of the density matrix the time-correlation function can be computed which is the Fourier transform of the spectral density. The latter can be directly compared to experimental measurements of absorption spectra. A seminal application of the theory of open quantum systems [138] is the understanding of photosynthesis at the atomic level in light harvesting systems [36].

The quantum aspects of the dynamics of virtually all of these problems can be described by the time-dependent Schrödinger equation

$$i\partial_t \Psi \quad = \quad H\Psi \,. \tag{7.1}$$

The physical properties of a quantum problem and the numerical artifacts like transparent boundary conditions are completely encoded in the details of the Hamiltonian H. Provided H is time-independent and at some initial time t_0 a state $\Psi(t_0)$ is known, the solution $\Psi(t)$ at all later times $t > t_0$ and thus the dynamical behavior is, at least in a formal sense, described by

$$\Psi(t) \quad = \quad e^{-i(t-t_0)H}\Psi(t_0)\,. \tag{7.2}$$

To compute $\Psi(t)$ from Eq. (7.2), rather than directly solving the PDE (7.1), one has to find a way to evaluate the action of the propagator $e^{-i(t-t_0)H}$ on the initial state $\Psi(t_0)$. In general, to evaluate the matrix exponential for large intervals $t-t_0$ the semigroup property of $e^{-i(t-t_0)H}$ is employed. Instead of one big leap rather N small steps of length $\tau = (t-t_0)/N$ are taken

$$\Psi(t) \quad = \quad \left(e^{-i\tau H}\right)^N \Psi(t_0)\,. \tag{7.3}$$

A traditional method for approximating the incremental propagator $e^{-i\tau H}$ is operator splitting. The Hamiltonian $H = T + V$ is the sum of kinetic, $T = \nabla^2/2m$, and potential energy, V. After discretization the potential energy is a diagonal matrix whereas the kinetic energy leads to some banded matrix. However, in momentum space the kinetic energy is essentially the square of the modulus of the wave vector $\mathbf{k}$ and T becomes a diagonal matrix $\hat{T} = |\mathbf{k}|^2$ (the factor $1/2m$ is absorbed in properly chosen units). This observation led to a symmetric operator-splitting scheme [44] based on the Baker-Campbell-Hausdorff series for the incremental propagator

$$e^{-i\tau H} \quad = \quad e^{-i\tau V/2} e^{-i\tau T} e^{-i\tau V/2} e^{\mathscr{O}(\tau^3)}\,. \tag{7.4}$$

Since T and V do not commute this splitting is not exact. The linear and quadratic error terms are canceled by the symmetric distribution of the $V/2$ terms. Application of the kinetic energy operator to a state Ψ in momentum space is realized by computing the Fourier transform $\mathscr{F}(\Psi)$ of Ψ, multiply by $e^{-i\tau|\mathbf{k}|^2}$ and transforming back. Therefore, the complete propagation scheme for one time step is

$$\Psi(t+\tau) = e^{-i\tau H}\Psi(t) \quad = \quad e^{-i\tau V/2}\mathscr{F}^{-1}\left\{e^{-i\tau T}\mathscr{F}\left(e^{-i\tau V/2}\Psi(t)\right)\right\}\,. \tag{7.5}$$

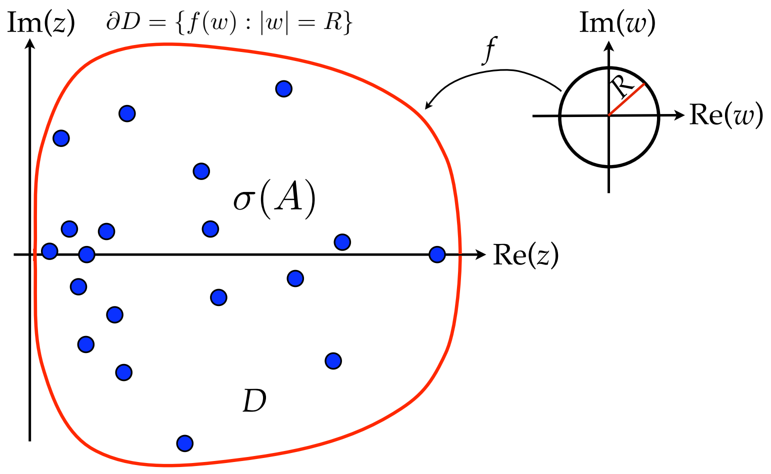


Figure 7.1: The exterior mapping function f constructs the inclusion set D as image of a circle of radius R. Blue dots represent dominant eigenvalues obtained from the Arnoldi process. These are used to fit known shapes of D.

7.2 Propagation Scheme based on Faber Polynomials

The appealing property of Eq. (7.5) is that one does not need to know anything about the spectrum $\sigma(H)$ of H. Yet, if some information about $\sigma(H)$ is available the operator-splitting is not necessary and a more accurate propagation scheme can be devised by expanding the incremental propagator into a Faber series. Faber series are based on Faber polynomials which were introduced to generalize the Taylor expansion from the unit disc to an arbitrary simply connected domain D [43]. They are unique with respect to the shape of the particular domain D.

7.2.1 Faber Polynomials

Provided $D \subset \mathbb{C}$ is compact, $0 \notin D$ and the complement D^c is simply connected in $\overline{\mathbb{C}} := \mathbb{C} \cup \{\infty\}$, Riemann's Mapping Theorem guarantees the existence of a conformal map $f : S^c \to D^c$, known as *exterior mapping function* (EMF) of D, cf. Fig. 7.1. S is the unit disc and S^c its complement. The starting point for determining the coefficients of Faber polynomials is the expansion of the EMF and its inverse into Laurent series

$$f(w) = t\left(w + a_0 + \sum_{k=0}^{\infty} \frac{a_k}{w^k}\right), \tag{7.6}$$

$$f^{-1}(z) = \frac{z}{t} + b_0 + \sum_{k=0}^{\infty} \frac{b_k}{z^k}, \tag{7.7}$$

where $t := f'(\infty)$ is the transfinite diameter of D. Faber polynomials $F_n(z;D)$ for a given domain D are defined as the regular part of $[f^{-1}(z)]^n$, $n \geq 0$. Given the Jordan curve $J_R := \{f(w) : |w| = R > 1\}$ which forms the boundary of the open set J the nth Faber polynomial for $D = \bar{J}$ is implicitly defined by its generating function

$$\frac{f'(w)}{f(w) - z} =: \sum_{n=0}^{\infty} \frac{F_n(z)}{w^{n+1}}, \quad |w| > R, \quad z \in J. \tag{7.8}$$

The sequence starts with $F_0(z) = 1$ and continues for $n \geq 1$ with

$$F_n(z) = \frac{z}{t} F_{n-1}(z) - \sum_{j=0}^{n-2} b_j F_{n-1-j}(z) - n b_{n-1} \tag{7.9}$$

which is of exact degree n with leading term $(z/t)^n$. This general recursion requires to keep all previously computed polynomials which would lead to an uneconomic propagator because of the quadratic complexity with respect to the polynomial degree. According to [84] Eq. (7.9) can be shortened to a three-term recurrence if the EMF is a rational function of the form

$$f(w) \equiv \frac{P(w)}{Q(w)} := \frac{w^2 + \mu_1 w + \mu_0}{\nu_1 w + \nu_0}. \tag{7.10}$$

The parameters ν_0, ν_1, μ_0, μ_1 are completely determined by the shape of D.

7.2.2 Polynomial Approximation of the Exponential Integrator

One of the lessons of chapter 3 was, that although rich in matrix-vector products (MVP), for PDEs polynomial preconditioning is inferior to other preconditioning methods since gains from parallelization are balanced by the large number of MVPs needed to achieve convergence. The reason is that the asymptotic convergence rate R, Eq. (3.27), which measures the reduction of the residual per iteration step, approaches one with decreasing mesh width.

However, in the context of quantum time propagation the situation changes as $R = 1$ would mean a perfect conservation of the norm of the wave function which is one of the fundamental properties of a quantum mechanical system. Although very similar from a technical point of view, propagation of a wave packet requires one additional step compared to preconditioning. For the latter it suffices to recursively compute one Faber polynomial F_n of desired degree n whereas for the former we additionally have to approximate the propagator e^{-itH} by a linear combination of Faber polynomials. Just as for the operator-splitting scheme the semigroup property is employed so that it is the incremental propagator $e^{-i\tau H}$ which has to approximated by polynomial expansion

$$\Psi(t) = \left(e^{\frac{-itH}{N}}\right)^N \Psi(t_0) \approx \left(\sum_{n=0}^{m} c_n F_n(H)\right)^N \Psi(t_0)\,. \tag{7.11}$$

It is known from complex analysis that Faber polynomials [135] provide a near-best approximation. For the practical purposes this means that this expansion is the best one can get. To achieve a good approximation of $e^{-i\tau H}$ we need a good estimate of $\sigma(H)$, the spectrum of H, which can be obtained from the Arnoldi procedure in Sec. 3.1 by diagonalizing the Hessenberg matrix. Given an estimate of $\sigma(H)$ we determine an inclusion set covering as little of the upper half of the complex plane as possible to avoid blow up of the factor $\max_{\partial D} |\exp(-i\tau z)|$, see the discussion of the convergence properties of the scheme in [29]. The expansion coefficients c_n only depend on the incremental timestep τ and the shape of the inclusion set D. The latter is the image of a circle of radius r under the exterior mapping function f, Eq. (7.10). The coefficients are obtained from Cauchy's integral formula for a holomorphic function $g : D \to \mathbb{C}$. For $z \in D$ and by the definition of the Faber polynomials, Eq. (7.8), we have

$$\begin{aligned}
2\pi i\, g(z) &= \oint_{\partial D} \frac{g(u)}{u-z} du && (7.12)\\
&= \oint_{|w|=R} \frac{g(f(w))}{f(w)-z} \frac{df}{dw} dw && (7.13)\\
&= \oint_{|w|=R} g(f(w)) \sum_{n=0}^{\infty} \frac{F_n(z)}{w^{n+1}} dw && (7.14)\\
&= \sum_{n=0}^{\infty} F_n(z) \oint_{|w|=R} g(f(w)) w^{-(n+1)} dw\,. && (7.15)
\end{aligned}$$

Set $g(f(w)) = \exp(-i\tau f(w))$ and the integral in the last expression gives

$$c_n = c_n(\tau) = \frac{1}{2\pi i} \oint_{|w|=r} e^{-i\tau f(w)} w^{-(n+1)} dw\,. \tag{7.16}$$

7.2.3 Choice of Inclusion sets

In quantum chemistry or photonics the inclusion set often is approximated by ellipses if analytical expressions for the coefficients c_n are desired or the spectrum is enclosed in polygons for which the EMF can only be computed numerically.

In case of an ellipse, i.e. $f(w) = a^2 w + b^2/w$, $a, b \in \mathbb{R}$ and noting that on the unit circle $aw/b = e^{ix}$ we have $2ab\cos(x) = a^2 w + b^2/w$, the integral in Eq. (7.16) can be computed analytically. By use of elementary trigonometric identities and the definition of Bessel's first integral [10, Eq. 9.1.21] we get for the coefficients

$$c_n(\tau) \equiv i^{-n} \mathrm{J}_n(2ab\tau) \quad = \quad \frac{1}{2\pi}\left(\frac{a}{b}\right)^n \int_0^{2\pi} e^{-i\tau 2ab\cos(x) - inx} dx. \tag{7.17}$$

The asymptotic behavior of the Bessel function, $J_n(u) = \mathscr{O}((u/2)^k)$, for small arguments, i.e. $u \ll \sqrt{k+1}$, leads to an exponential convergence of the Faber series, Eq. (7.19), provided $u = 2ab\tau < 2$. The disadvantage of ellipses is the possibly large area which has to be covered in the positive half-plane of $\mathbb{C}$ leading to more terms required in the polynomial expansion of $e^{-i\tau H}$.

The great achievement of [83] was the introduction of croissant-shaped, non-convex inclusion sets with a closed form for the EMF as given in Eq. (7.10) and a short recursion for the Faber polynomials. In that case the coefficients are given by

$$c_n(\tau) \quad = \quad \frac{1}{2\pi i} \oint_{|w|=r} e^{-i\tau \frac{w^2 + \mu_1 w + \mu_0}{\nu_1 w + \nu_0}} w^{-(n+1)} dw\,. \tag{7.18}$$

7.2.4 Propagation Polynomials

The propagation from t to $t+\tau$ requires to compute

$$\Psi(t+\tau) \quad = \quad \sum_{n=0}^{m} c_n F_n(H) \Psi(t) \tag{7.19}$$

by sequentially computing

$$\psi_n \quad := \quad F_n(H)\Psi(t) \tag{7.20}$$

and then forming the linear combination

$$\Psi(t+\tau) \quad := \quad \sum_{n=0}^{m} c_n \psi_n\,. \tag{7.21}$$

As Eq. (7.16) shows all the coefficients c_n can be computed in advance and thus are considered as known in the following. The numerical approximation of $\Psi(t+\tau)$ will be denoted as Ψ_m and ψ_n will be used synonymously for the numerical approximation of $F_n(H)\Psi(t)$.

To compute the action of the nth Faber polynomial on the initial state $r_0 = F_0\Psi(t) \equiv \Psi(t)$ the propagation polynomial (it plays the same role as the initial residual in Krylov methods) will be approximated by normalized Faber polynomials $\tilde{F}_n(z)$, i.e. $\tilde{F}_n(0) = 1$ and

$$\tilde{\psi}_n = \tilde{F}_n(H) r_0 \quad = \quad \frac{1}{\rho_n} F_n(H) r_0 \tag{7.22}$$

where the normalization constant is given by

$$\rho_n \quad := \quad F_n(0)\,. \tag{7.23}$$

To apply the theory for non-convex inclusion sets the recursion for the Faber polynomials has to expressed in terms of shifted Faber polynomials $\hat{F}_n$ which are related to the original ones by

$$\hat{F}_n(z) - F_n(z) \quad = \quad \left(\frac{-\nu_0}{\nu_1}\right)^n \; =: \; S^n\,, \quad \nu_0 \neq 0\,, \tag{7.24}$$

$$\hat{F}_n(z) - F_n(z) \quad = \quad 1 \; =: \; S^n\,, \quad \nu_0 = 0\,. \tag{7.25}$$

The sole reason for their introduction is the fact that shifted Faber polynomials can be expressed in terms of the zeroes $w_1(z)$ and $w_2(z)$ of an auxiliary polynomial $P(w) - zQ(w)$

$$(w - w_1(z))(w - w_2(z)) \quad = \quad w^2 + (\mu_1 - \nu_1 z)w + (\mu_0 - \nu_0 z) \tag{7.26}$$

and thus $w_1(z)$ and $w_2(z)$ are given in terms of the parameters of the EMF in Eq. (7.10). The explicit expression for the shifted Faber polynomials is

$$\hat{F}_n(z) \quad = \quad w_1(z)^n + w_2(z)^n, \quad n \geq 1\,. \tag{7.27}$$

As shown in [84] (Thm 3.1) for domains Moebius-equivalent to an ellipse shifted Faber polynomials can be computed from a three-term recurrence with coefficents which are functions of the zeroes $w_1(z)$ and $w_2(z)$. For convenience we introduce some auxiliary functions

$$\begin{aligned} 2W(z) \quad &:= \quad w_1(z) + w_2(z) \; = \; \nu_1 z - \mu_1\,, \\ V(z) \quad &:= \quad w_1(z)w_2(z) \; = \; \mu_0 - \nu_0 z\,. \end{aligned}$$

In terms of these functions the recursion is given by

$$\begin{aligned} \hat{F}_0(z) \quad &= \quad 2 \\ \hat{F}_1(z) \quad &= \quad 2W(z) \\ \hat{F}_n(z) \quad &= \quad 2W(z)\hat{F}_{n-1}(z) - V(z)\hat{F}_{n-2}(z)\,, \quad n \geq 2\,. \end{aligned} \tag{7.28}$$

By grouping terms according to the dependence on z of the coefficients and defining

$$v_n(z) \quad := \quad \nu_1 \hat{F}_{n-1}(z) + \nu_0 \hat{F}_{n-2}(z) \tag{7.29}$$

the final form of the nth polynomial is

$$\hat{F}_n(z) \quad = \quad z v_n(z) - \mu_1 \hat{F}_{n-1}(z) - \mu_0 \hat{F}_{n-2}(z)\,. \tag{7.30}$$

From this recursion we can immediately deduce a recursion for the normalization factor by evaluating Eqs (7.24) and (7.25) at $z = 0$ which gives

$$\hat{F}_n(0) \quad = \quad -\mu_1 \hat{F}_{n-1}(0) - \mu_0 \hat{F}_{n-2}(0)\,, \quad n \geq 2\,, \tag{7.31}$$

$$\rho_n \quad = \quad \hat{F}_n(0) - S^n\,, \quad n \geq 2\,. \tag{7.32}$$

with the special values $\rho_0 = 2$ and $\rho_1 = -\mu_1 - S$ for initialization. In terms of the shifted Faber polynomials the propagation polynomial is

$$p_n(z) = \frac{1}{\rho_n}\left[\hat{F}_n(z) - S^n\right] \tag{7.33}$$

which follows from Eq. (7.22). For the normalized intermediate states we get

$$\tilde{\psi}_n = p_n(z) r_0 . \tag{7.34}$$

For the final state we have

$$\begin{aligned} \Psi_m &= c_m \psi_m + \sum_{n=0}^{m-1} c_n \psi_n \\ &= c_m \psi_m + \Psi_{m-1} = c_m \rho_m \tilde{\psi}_m + \Psi_{m-1} . \end{aligned} \tag{7.35}$$

The polynomial corresponding to the recursive computation of the final state plays the same role as the iteration polynomial in polynomial preconditioning but is simpler to compute. Hence, for implementing the scheme we can copy the MATLAB pseudo code given in Liesen's PhD thesis and use it for quantum time propagation in wave packet methods after the proper modifications for the computation of Ψ_m.

Due to the exponential convergence of the Faber series [42] the point of truncation, i.e. the value of m in Eq. (7.19) is chosen such that $c_m \leq Tol$. The real parameter Tol is a user-prescribed tolerance, e.g. $Tol = 10^{-15}$ like in [29]. Since the c_k can all be computed prior to the first timestep there computation can be utilized to dynamically determine m which is then kept fixed during the whole simulation as the c_k only depend on the spectral properties of H which are time-independent.

7.3 Time-Dependent Magnetic Focussing

Chapter 6 gave a detailed account of the computation of stationary currents due to ballistic electron transport through a two-dimensional electron gas. The particular physical issues addressed were the influence of an external magnetic field and the presence of disorder. A more intuitive picture of electron propagation can be obtained from time-resolved simulations of the quantum mechanical system as recently discussed [78] which uses the operator-splitting scheme together with inexact absorbing boundary conditions. The last task is to design an initial state. The easiest realization is to use a wave packet with a Gaussian distribution of velocities (wave numbers). In real space it is of the form

$$\Psi(\mathbf{x}, t=0) = \frac{\exp\left(-\frac{|\mathbf{x}-\mathbf{x}_0|^2}{2a^2} + \frac{a^2 k_0^2}{4}\right)}{a\sqrt{\pi \mathrm{I}_0(a^2 k_0^2/2)}} \mathrm{J}_0(k_0|\mathbf{x}-\mathbf{x}_0|) . \tag{7.36}$$

It is parameterized by the mean wave number k_0, the initial position $\mathbf{x}_0$ and the half width $1/a$ of the wave number distribution. The functions J_0 and I_0 denote Bessel functions of the first and second kind, respectively. As of writing this thesis there does not seem to be any systematic investigation of time-resolved transport of 2DEG corner device presented in Sec. 6.5.

Chapter 8

Dielectric Relaxation Spectroscopy of Ubiquitin in Aqueous Solution

Recent dielectric spectroscopy studies of ubiquitin in solution have revealed the influence of conformational sampling on the direct current contribution to the dielectric loss spectrum. Initial theoretical studies show that a simple 2-state, ratchet-like model for the conformational dynamics of a protein coupled to a Fokker-Planck model for the mobile ions may explain the experimentally observed sub-β peak in the dielectric loss spectrum.

This chapter discusses the main issues of replacing the stochastic description by a more realistic Poisson-Nernst-Planck model for the ion dynamics and the electrostatic potential:

- The set of partial differential equations modeling dielectric relaxation spectroscopy are derived from the continuity equation and the electro-diffusive fluxes.

- The simulation of the dielectric relaxation spectroscopy experiment on a generic, solvated globular protein needs appropriate boundary conditions for the dielectric relaxation cell and for the protein-solvent interface. The experimental setup introduces solvent-electrode interfaces which give rise to dielectric double layers well-known from the electro-chemistry of surfaces. The excluded volume interaction between protein and ions leads to an integral equation for the electrostatic potential on the protein-solvent interface.

- In the bulk the model is discretized by finite elements which is coupled to a boundary element method for the non-local boundary condition on the protein-solvent interface.

- Last but not least, a hybrid parallelization strategy is outlined using CUDA for the data-parallel cell contributions.

First results on the simplified two-dimensional model indicate the validity of the model.

8.1 Introduction

Structural protein dynamics are a key component for protein function and have been used to explain a large variety of complex processes spanning timescales from picoseconds to seconds or minutes. Nuclear magnetic resonance (NMR) spectroscopy is the most frequently used experimental technique to characterize protein dynamics, yet it is not able to span the full range of timescales in question. Especially, the rates for the intramolecular dynamics in the so-called supra-τ_c time window between the nanosecond rotational correlation time τ_c and 50 μs are usually inaccessible to NMR techniques or can at best be measured indirectly.

However, in the case of free ubiquitin in solution [18] the internal dynamics could be determined directly by dielectric relaxation spectroscopy (DRS). The dielectric spectrum of ubiquitin is known for a long time [72] with a, by now, well established set of peaks of which the most prominent ones are those for tumbling of the protein's static dipole (β-peak near 10 MHz) and for reorientation of bulk water (γ-peak near 10 GHz). Peaks at lower frequencies and of less striking height were either overlooked, for technical reasons not observable or were explained rather generically, e.g. by unidentified protein-water interactions or the movement of polar side groups of the amino acids. Recently published evidence for interaction of protein conformation and direct current (DC) conductivity in dielectric spectroscopy experiments [18] revealed the importance of low frequency peaks as they seem to be intimately connected to the kinetics of the conformational sampling of a protein. In the case of ubiquitin the recently discovered sub-β peak [18] is considered to reflect the ability of a protein to temporarily store various amounts of ions in its dielectric double layer, thus working like an internal capacitor. To explain this on a microscopic level it is currently postulated that, depending on its conformation, a protein molecule stores different amounts of ions in its hydration shell. Due to overall charge conservation this influences the number of mobile ions in solution and thus the direct current in a DRS experiment has to fluctuate with conformation. Therefore, if this mechanism pertains and is attributable to the sub-β peak, then further insight into the mechanisms of molecular recognition and other types of structural protein dynamicscan be obtained from careful investigation of the low-frequency part of the dielectric loss spectrum.

Numerical modeling allows us to study the level of detail required in the physical description of the protein and the DRS experiment needed to explain the sub-β peak. The electrostatics of the aqueous environment of the protein molecule are described by means of the Poisson-Nernst-Planck equation which extends the usual Poisson-Boltzmann description by stationary currents due to the local potential. The protein is considered as globular and as a simply connected zone of excluded volume for the mobile ions. The protein is characterized by the shape of this exclusion zone, its dielectric constant and the distribution of static charges in this zone to represent local inhomogeneities which are caused by the different amino acid residues. The link to the experiment is provided by a ratchet-like stochastic dynamics of switching between a set of conformations with exponentially distributed waiting times which leads to a simulated time series of the DC component of the DR spectrum.

8.2 Master-Fokker-Planck Theory

The simplest model for the interaction between conformational dynamics and the DC current [19] only addresses the dependence of the distribution of bound and mobile ions on the different protein states. To quantify this dependence we need a relation between polarization and electric field that allows to identify the susceptibility, which is measured in the experiment.

The probability to find a mobile ion at position $\mathbf{r}$ at time t while a protein is in state k is $w_k(\mathbf{r},t)$. We define c_k to be the space-averaged charge density of mobile ions if all proteins are in state k. The state-dependent dipole moment of the mobile ions is given by

$$\mathbf{R}_k \quad := \quad \int_{\mathbb{R}^3} d^3r \, \mathbf{r} w_k(\mathbf{r},t) \tag{8.1}$$

and the state-dependent electric polarization vector is

$$\mathbf{P}_k \quad := \quad c_k \mathbf{R}_k \, .$$

A change in polarization induces a current

$$\mathbf{j}_k \equiv \partial_t \mathbf{P}_k .$$

In the simplest setting there are only two protein states with corresponding sub-ensembles for the mobile ions with mutually different mobilities μ_1 and μ_2. A protein molecule switches back and forth between the two states with rate γ. From the underlying Master-Fokker-Planck equation we obtain equations of motion for the dipole moments

$$\begin{aligned}\partial_t \mathbf{R}_1 &= \mu_1 \mathbf{E} + \gamma \mathbf{R}_2 - \gamma \mathbf{R}_1 ,\\ \partial_t \mathbf{R}_2 &= \mu_2 \mathbf{E} + \gamma \mathbf{R}_1 - \gamma \mathbf{R}_2\end{aligned}$$

and polarization vectors

$$\begin{aligned}\partial_t \mathbf{P}_1 &= c_1 \mu_1 \mathbf{E} + c_1 \gamma \mathbf{R}_2 - c_1 \gamma \mathbf{R}_1 ,\\ \partial_t \mathbf{P}_2 &= c_2 \mu_2 \mathbf{E} + c_2 \gamma \mathbf{R}_1 - c_2 \gamma \mathbf{R}_2 .\end{aligned}$$

To get insight into the influence of small changes in the concentration of mobile ions we define the state-averaged polarization

$$\mathbf{P} := \mathbf{P}_1 + \mathbf{P}_2$$

and its fluctuation

$$\mathbf{Q} := \mathbf{P}_1 - \mathbf{P}_2 .$$

The state-averaged conductivitiy and its fluctuation are

$$\begin{aligned}\Sigma &:= c_1 \mu_1 + c_2 \mu_2 ,\\ \sigma &:= c_1 \mu_1 - c_2 \mu_2 .\end{aligned}$$

As measure for changes in average ion concentrations we introduce

$$\delta := \frac{c_1 - c_2}{c_1} .$$

In case of small changes $\delta \ll 1$ we get in linear order

$$\begin{aligned}c_1 &= c_2 (1 + \delta) ,\\ c_2 &\approx c_1 (1 - \delta) ,\end{aligned}$$

and the equations of motion for $\mathbf{P}_k$ lead to the corresponding ones for $\mathbf{P}$ and $\mathbf{Q}$

$$\begin{aligned}\partial_t \mathbf{P} &= \Sigma \mathbf{E} - \gamma \mathbf{P} + c_1 \gamma \mathbf{R}_2 + c_2 \gamma \mathbf{R}_1 \\ &= \Sigma \mathbf{E} - \delta \gamma \mathbf{Q} ,\end{aligned}$$

$$\begin{aligned}\partial_t \mathbf{Q} &= \sigma \mathbf{E} - \gamma \mathbf{Q} - \gamma (c_1 \mathbf{R}_2 - c_2 \mathbf{R}_1) \\ &= \sigma \mathbf{E} - 2\gamma \mathbf{Q} - \gamma \delta \mathbf{P} .\end{aligned}$$

Transforming from time to frequency domain, i.e. $\mathbf{P}(t) = \mathbf{P}_w e^{-iwt}$, $\mathbf{Q}(t) = \mathbf{Q}_w e^{-iwt}$ and $\mathbf{E}(t) = \mathbf{E}_w e^{-iwt}$ we get a system of equations for the amplitudes

$$\begin{aligned} -iw\mathbf{P}_w &= \Sigma\mathbf{E}_w - \delta\gamma\mathbf{Q}_w\,, \\ -iw\mathbf{Q}_w &= \sigma\mathbf{E}_w - 2\gamma\mathbf{Q}_w - \gamma\delta\mathbf{P}_w\,, \\ \mathbf{Q}_w &= \frac{\sigma\mathbf{E}_w - \gamma\delta\mathbf{P}_w}{-iw + 2\gamma}\,. \end{aligned}$$

After inserting the latter into the former

$$\begin{aligned} -iw\mathbf{P}_w &= \Sigma\mathbf{E}_w - \frac{\delta\gamma\sigma\mathbf{E}_w - \gamma^2\delta^2\mathbf{P}_w}{-iw + 2\gamma} \\ \mathbf{P}_w &= \frac{i}{w}\frac{(w^2+4\gamma^2)\Sigma - (iw+2\gamma)\delta\gamma\sigma}{w^2+4\gamma^2}\mathbf{E}_w + O(\delta^2) \\ &= \frac{i}{w}\left(\Sigma - \frac{2\delta\gamma^2\sigma}{w^2+4\gamma^2} - i\frac{w\delta\gamma\sigma}{w^2+4\gamma^2}\right)\mathbf{E}_w \end{aligned}$$

we obtain an expression of the form $\mathbf{P}_w = \chi(w)\mathbf{E}_w$ which allows us to read off the frequency-dependence of the complex susceptibility $\chi(w) = \chi'(w) + i\chi''(w)$ of which the imaginary part represents the dielectric loss spectrum

$$\chi''(w) = \frac{1}{w}\left(\Sigma - \frac{2\delta\gamma^2\sigma}{w^2+4\gamma^2}\right)\,.$$

In this preliminary form there would be no sign of a peak. To reveal the desired sub-β peak we have to remove the pole at $w = 0$, i.e. the low frequency or direct current part

$$w\chi''_{\text{sub-}\beta} = w\chi'' - \lim_{w\to 0} w\chi'' \tag{8.2}$$

$$= \left(\Sigma - \frac{2\delta\gamma^2\sigma}{w^2+4\gamma^2}\right) - \left(\Sigma - \frac{2\delta\gamma^2\sigma}{4\gamma^2}\right) \tag{8.3}$$

which leads to

$$\chi''_{\text{sub-}\beta} = \frac{\delta\sigma}{2}\frac{w}{w^2+4\gamma^2} \tag{8.4}$$

with a peak at $w_{\text{sub-}\beta} = 2\gamma$. The ratio of $\chi''_{\text{sub-}\beta}$ and the DC contribution $\chi_{DC} = w/\Sigma$ at $w_{\text{sub-}\beta}$ is

$$\frac{\chi''_{\text{sub-}\beta}(w_{\text{sub-}\beta})}{\chi_{DC}(w_{\text{sub-}\beta})} \tag{8.5}$$

$$= \frac{\delta\sigma}{4\Sigma} \tag{8.6}$$

$$\approx \frac{\delta}{4}\left(\frac{\delta\mu_1}{\mu_1+\mu_2} + \frac{\mu_1-\mu_2}{\mu_1+\mu_2} - \frac{\delta(\mu_1-\mu_2)}{(\mu_1+\mu_2)^2} + O(\delta^2)\right)\,. \tag{8.7}$$

The concentrations c_k only measure the total densities of mobile ions irrespective of the details of the chemical composition. Especially, if the protein has a state-dependent affinity to a particular species this should affect the overall mobilities, too. We assume that the mobilities vary in a similar fashion as the concentrations w.r.t. the protein state so that $\mu_1 - \mu_2 = O(\delta)$. Thus, the ratio of the susceptibilities should behave like

$$\frac{\chi''_{\text{sub-}\beta}(w_{\text{sub-}\beta})}{\chi_{DC}(w_{\text{sub-}\beta})} = O(\delta^2) \tag{8.8}$$

which can be compared to experiments [19, Fig. S3B]. From the observed ratio we deduce

$$\delta = O(10^{-2}).$$

In a perfect experiment with ionic contributions solely due to the eigen-dissociation of water the average concentration of mobile ions would be 10^{-7} M. In this case $\delta = O(10^{-2})$ corresponds to a relative change in concentration of 10^{-9} M. In the experiment, the proteins themselves constitute a solute and typical concentrations are 10^{-3} M. Thus, it suffices if one out of 10^6 protein particles absorbs one mobile ion for a period of time longer than the average lifetime of the protein states.

8.3 Finite System with Selected Ion Species Resolved

The Fokker-Planck theory covers the salient features of the proposed mechanism. For a more quantitative analysis we have to model the whole experiment including boundary effects in order to get a notion of the interaction of the experimental setup with the modulation of the density of mobile ions. This implies that we have to model the flux of current through the impedance cell and especially how it is measured. We have to compute the distribution of ions and the electrostatic potential. In the electro-diffusive picture this requires us to distinguish at least between anions and cations and their respective densities.

We begin with the geometric properties of our model problem. We consider a globular protein as a bounded region Ω_P with surface Γ in a rectangular box or cylinder Ω of aqueous solution (see Fig. 8.1 for the cylindrical setup). The solution fills the domain $\Omega_S = \Omega \backslash \Omega_P$. The axes of the box coincide with the x, y, z axes of a laboratory cartesian frame. The lower left corner is at $(x, y, z) = (0, 0, 0)$ and the upper right at (L_x, L_y, H). The cylinder is of radius R and height H. Its symmetry axis is aligned with the z-axis while its bottom lies in the x, y-plane. The protein may fluctuate between two conformations (denoted 1 and 2), differing in the distribution of internal charges $\rho_1(\mathbf{r}) \neq \rho_2(\mathbf{r})$, but not in shape. In our simulations the protein is taken to be spherical which is sufficient for globular proteins. The cylindrical geometry with is axial symmetry is designed for single-molecule studies. If several molecules have to simulated (which is not the topic of this work) as in the supplementary material to [18] the box-shaped geometry may be more favorable in case the molecules are arranged in a regular lattice.

The top and bottom boundaries of the cell Ω (denoted by Γ_C and Γ_A in Fig. 8.1) are metal contacts a distance H apart and connected to a battery, which keeps them at constant potentials Φ_C (at the cathode C) and Φ_A (at the anode A). The remaining part of the outer surface of Ω is denoted as Γ_0. In the following $\partial_{\mathbf{n}}$ is the normal derivative at the boundary $\Gamma_S = \Gamma_A \cup \Gamma_C \cup \Gamma_0 \cup \Gamma$ which is not simply connected. If required by the context the normals of the sub-boundaries are denoted as $\mathbf{n}_A$, $\mathbf{n}_C$, $\mathbf{n}_0$ and $\mathbf{n}_{PS}$. The normal derivative of a scalar function f at a subset E of the boundary Γ_S will be $\partial_{\mathbf{n}} f\big|_E$. Ultimately, we want to calculate the total electric current between the electrodes as a function of potential difference $\Phi_A - \Phi_C$ and extract the corresponding conductivity σ.

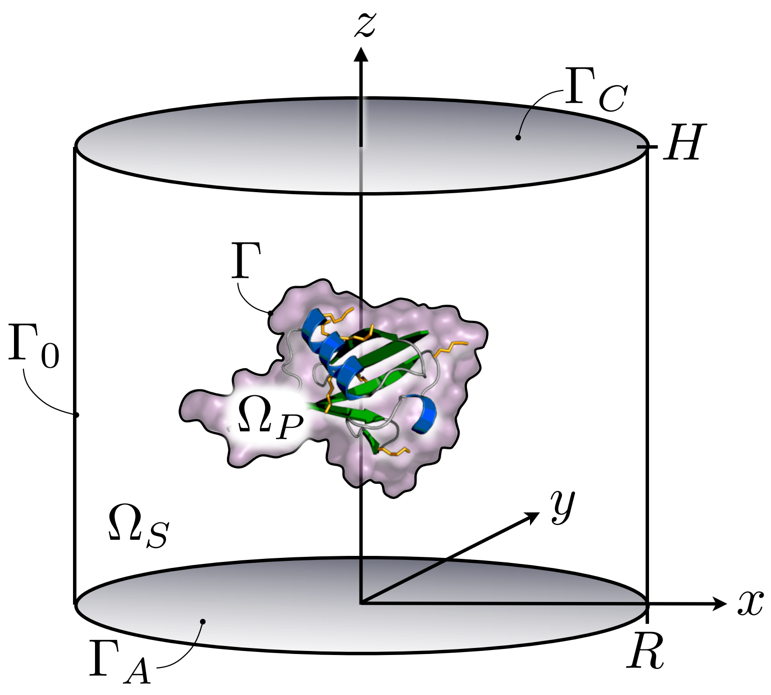


Figure 8.1: Geometry for single-molecule dielectric relaxation spectroscopy. The molecule is Ubiquitin shown as overlay of ribbon-model representation and and molecular surface.

For simplicity, we only consider a single type of cation (symbolically, I^+ with charge $q_+ = -ze$, $e < 0$ denoting the elementary charge quantum) and a symmetric anion with $q_- = ze$. The particle densities of these ions are denoted by $n_\pm$, respectively. At the metal electrodes, the cations may be reduced in the redox reaction

$$I^+ + electron \leftrightarrow N \tag{8.9}$$

and become neutral molecules N which are only subject to diffusion. The N particles may stay in the solution, in which case we have to include their density n_0 ($q_0 = 0$) as a dynamical field in the description. We assume that the electrodes are perfectly blocking for anions and neutral molecules in order to avoid highly nonlinear model equations for the boundary conditions.

Within the exterior domain Ω_S the densities of mobile ions obey continuity equations

$$\partial_t n_i = -\nabla \cdot \mathbf{j}_i \tag{8.10}$$

and the current densities will be taken to be of the Nernst-Planck form (neglecting convective fluxes due to solvent flow)

$$\mathbf{j}_i = -D_i[\nabla n_i + \beta q_i n_i \nabla \Phi] \tag{8.11}$$

where $\beta = 1/k_B T$ and k_B is Boltzmann's constant. The electric potential Φ is obtained from Poisson's equation

$$-\nabla \cdot (\varepsilon_0 \varepsilon_r(\mathbf{r}) \nabla \Phi) = -ze(n_+ - n_-)\chi_{\Omega_S} + \rho_a \tag{8.12}$$

where ε_0 is the vacuum permittivity and ε_r is the relative permittivity. Another implication by this approach is that accounting for cations and anions must be restricted to the solvent subdomain. During the modeling stage this is done by using the characteristic function χ_{Ω_S}. The electrostatic properties of the model protein are assumed to be completely characterized by a conformation-dependent charge density ρ_a and a homogeneous dielectric constant $\varepsilon_P \approx 2$. In the simplest case there are only two conformations, i.e. $a \in \{1,2\}$.

8.3.1 Experimental Objectives

To further simplify the problem we assume that all relaxation processes within the system of ions are fast compared to the rate of change between the two protein conformations. Within this approximation, we just need to calculate the steady state current of both conformations, I_1, I_2. Then we can obtain the time dependence of the current from the simple stochastic two-state dynamics of the conformations, i.e.

$$2I(t) = I_1 + I_2 + a(t)(I_1 - I_2)$$

with $a(t) \in \{-1,+1\}$ representing a two-state telegraph process with an exponential distribution of waiting times τ between transitions

$$p(\tau)d\tau = \gamma e^{-\gamma\tau} d\tau\,. \tag{8.13}$$

From the boundary condition Eq.(8.19) the total current through the cathode can be calculated

$$I = K_R \int_{\Gamma_C} n_+ \, d\Gamma_C. \tag{8.14}$$

Once the currents are computed, their relative deviation

$$\delta_I := \frac{|I_1 - I_2|}{|I_1 + I_2|} \tag{8.15}$$

should explain the amplitude of the sub-β peak.

8.3.2 Boundary Conditions

The mathematical model is not complete without appropriate boundary conditions at the electrodes and at the interface Γ between protein and solvent.

Electrodes

BCs including Thickness of Stern Layer: For the potential the boundary conditions at the electrodes are rather intricate and there exists a considerable amount of literature adressing this question. Usually, the redox reaction rates are described by Butler-Volmer kinetics, including the Frumkin correction due to the presence of a Stern layer [115].

As already pointed out in [24] the boundary conditions for ”ideally polarizable” or ”completely blocking” electrodes can be simplified to a Robin-type condition

$$\Phi\big|_{\Gamma_E} = \Phi_E + \lambda_L \partial_{\mathbf{n}} \Phi\Big|_{\Gamma_E}, \quad E \in \{A,C\}\,. \tag{8.16}$$

The compact part of the double layer of width λ_L, which includes the Stern layer [120] on the surface of the electrodes, can be considered as a very thin parallel plate capacitor Based on this capacitor model it is a reasonable assumption that the electric field is proportional to the difference between the potential Φ_E prescribed on the electrode and the resulting potential at the interface to the diffusive layer $\Phi\big|_{\Gamma_E}$ in solution. When λ_L is much smaller than the smallest length scale which is resolved in a model the Robin-type boundary condition degenerates to a Dirichlet condition. Here we are not interested in details of diffuse-charge effects and neglect the finite thickness of the Stern layer in Eq. (8.16). Furthermore, we adopt the simplified Tafel kinetics [115], which neglects oxidation (reduction) processes at the cathode (anode) completely. With these simplifications the boundary conditions for the electric potential are

$$\Phi\big|_{\Gamma_C} = \Phi_C, \quad \Phi\big|_{\Gamma_A} = \Phi_A\,. \tag{8.17}$$

For the current densities $\mathbf{j}_-$ of anions we have the blocking condition

$$\mathbf{n}\cdot\mathbf{j}_-\big|_{\Gamma_A} = \mathbf{n}\cdot\mathbf{j}_-\big|_{\Gamma_C} = 0\,. \tag{8.18}$$

The redox reaction at the electrodes, Eq. (8.9), implies a balance of inward and outward fluxes which makes the current transferring species obey at the cathode

$$\mathbf{n}\cdot\mathbf{j}_+\big|_{\Gamma_C} = \; K_R\, n_+\big|_{\Gamma_C} = -\mathbf{n}\cdot\mathbf{j}_0\big|_{\Gamma_C}\,. \tag{8.19}$$

Cations are removed from the solvent and are transformed into neutral particles at the cathode at a rate K_R. Simultaneously, neutral particles are generated and enter the system which is reflected by the fact that the corresponding currents are of opposite sign. At the anode neutral particles leave and cations enter:

$$\mathbf{n}\cdot\mathbf{j}_0\big|_{\Gamma_A} = \; K_O\, n_0\big|_{\Gamma_A} = -\mathbf{n}\cdot\mathbf{j}_+\big|_{\Gamma_A}\,. \tag{8.20}$$

The rates K_R and K_O are treated as fixed model parameters, in particular we neglect their dependence on Φ. In the following we consider them as equal.

Protein Surface

We assume that the interior of the protein is free of solvent and ions, i.e. we get homogeneous von-Neumann conditions for the electro-diffusive currents at the protein-solvent interface

$$-\,\mathbf{n}_P\cdot\mathbf{j}_i = 0 \quad \text{on}\,\Gamma. \tag{8.21}$$

The electrostatic boundary conditions at Γ are of standard form for dielectric interfaces, i.e. the total potential must be continuous

$$\lim_{a\to 0}\Phi(\mathbf{x}-a\mathbf{n}_P) \;=\; \lim_{a\to 0}\Phi(\mathbf{x}+a\mathbf{n}_P) \quad \forall\mathbf{x}\in\Gamma, \tag{8.22a}$$

and the jump in the dielectric constant induces a jump in the normal derivative of the potential

$$[\varepsilon(\mathbf{x})\mathbf{n}_P\cdot\nabla\Phi] \;=\; 0. \tag{8.22b}$$

The square brackets denote the jump of the bracketed quantity across Γ and $\mathbf{n}_P$ is the normal from Ω_P to Ω_S. To correctly take into account the influence of the intramolecular charge distribution and the difference in dielectric properties on the various subdomains we would have to solve for Φ on the whole domain Ω. The correct jump of the normal component of the electric field at Γ should then be a result of the computation. Yet, to compute the dielectric currents it suffices to know the potential on Ω_S. We have to convert the interface condition, Eq. (8.22b), into a boundary condition either for the values of Φ or its normal derivative at the interface Γ. In the interior of the protein, i.e. $\varepsilon_r = \varepsilon_P$, we only have to solve a Poisson equation

$$-\varepsilon_0\varepsilon_P\nabla^2\Phi \quad = \quad \rho_P \equiv \sum_k q_k\delta(\mathbf{x}-\mathbf{x}_k) \tag{8.23}$$

where we model the intramolecular charge distribution as a collection of point charges q_k at fixed positions $\mathbf{x}_k$. Equation (8.23) is a linear partial differential equation (PDE) with constant coefficients. Therefore, the PDE can be replaced by the corresponding boundary integral equation (BIE). The original interface problem on Ω is transformed into a boundary value problem effectively restricting the computation of Φ to Ω_S. To do this, we follow the discussion of interior the BIE formulation for linear interior Neumann Boundary value and interface problems in [110]. From potential theory [79] we know that for sufficiently smooth surface Γ the intramolecular potential $\Phi_P = \Phi|_{\Omega_P}$ at a point $\mathbf{x} \in \Gamma$ has to fulfill

$$\begin{aligned}\frac{1}{2}\Phi_P(\mathbf{x}) \quad &+ \quad \oint\limits_\Gamma \left[\Phi_P(\mathbf{x}')\frac{\partial G_{\mathbf{x}}}{\partial \mathbf{n}'_P}(\mathbf{x}') - G_{\mathbf{x}}(\mathbf{x}')\frac{\partial \Phi_P}{\partial \mathbf{n}'_P}\right] d\Gamma(\mathbf{x}') \\ &= \quad +\frac{1}{\varepsilon_0\varepsilon_P}\int_{\Omega_P} G_{\mathbf{x}}(\mathbf{x}')\rho_P(\mathbf{x}')\end{aligned} \tag{8.24}$$

where we have defined

$$G_{\mathbf{x}}(\mathbf{y}) := \frac{1}{4\pi|\mathbf{x}-\mathbf{y}|}$$

as the Green's function for the Poisson equation in three dimensions and $\mathbf{n}'$ is the outer normal at $\mathbf{x}' \in \Gamma$ w.r.t. Ω_P. In potential theory the right-hand side is commonly known as Newton potential. It collects the contributions due to the intramolecular charges

$$\phi^C(\mathbf{x}) = \frac{1}{\varepsilon_0\varepsilon_P}\sum_k q_k G_{\mathbf{x}_k}(\mathbf{x}) = \frac{1}{4\pi\varepsilon_0\varepsilon_P}\sum_k \frac{q_k}{|\mathbf{x}-\mathbf{x}_k|}\,. \tag{8.25}$$

For a more general formulation we introduce the normal component of the electric field

$$t_P \quad := \quad \partial_{\mathbf{n}_P}\Phi_P\,, \tag{8.26}$$

as auxiliary unkown. The integral formulation, Eq. (8.24), defines a Dirichlet-to-Neumann (DtN) map which on Γ maps the values of Φ_P to its normal derivative. The DtN map for the normal derivative is

$$t_P \quad = \quad S^P\Phi_P - V^{-1}\phi^C \tag{8.27}$$

where

$$S^P \quad := \quad V^{-1}\left(\frac{1}{2}I + K_P\right) \tag{8.28}$$

is the non-symmetric form of the Steklov-Poincaré operator and I is the identity. The Steklov-Poincaré operator is defined in terms of the single layer boundary integral operator

$$(Vt_P)(\mathbf{x}) \quad = \quad \oint\limits_\Gamma G_{\mathbf{x}}(\mathbf{x}')t_P(\mathbf{x}')d\Gamma(\mathbf{x}') \tag{8.29}$$

and double layer boundary integral operator

$$(K_P\Phi_P)(\mathbf{x}) = \oint_\Gamma \frac{\partial G_\mathbf{x}}{\partial \mathbf{n}'_P}(\mathbf{x}')\Phi_P(\mathbf{x}')d\Gamma(\mathbf{x}') . \tag{8.30}$$

The single layer boundary integral operator $V : H^{-1/2}(\Gamma) \to H^{1/2}(\Gamma)$ is bounded and $H^{-1/2}(\Gamma)$-elliptic and thus invertible. The sign of the double layer operator depends on the direction of the normal. To indicate that we have chosen $\mathbf{n}_P$ for the moment K gets an index P.

The DtN map enables us to incorporate the results from the BIE into the variational form of the Poisson problem for the potential on Ω_S as Neumann boundary data. Insertion of Eq. (8.27) into Eq. (8.22b) gives the boundary condition on Γ which correctly models the dielectric interface. In its final formulation

$$\varepsilon_S \partial_\mathbf{n}\Phi\big|_\Gamma = -\varepsilon_P t_P \equiv -\varepsilon_P S^P \Phi_P + \varepsilon_P V^{-1}\phi^C . \tag{8.31}$$

we have reversed the direction of the normal. We define $\mathbf{n}$ as the inward normal of Ω_P, evidently pointing from Ω_S to Ω_P. This also reverses the sign of the double layer operator which is indicated by the loss of the index P. In practice we rather use the implicit version

$$\left(\frac{1}{2}I - K\right)\Phi\big|_\Gamma + \frac{\varepsilon_S}{\varepsilon_P} V\partial_\mathbf{n}\Phi\big|_\Gamma = \phi^C \tag{8.32}$$

which avoids inverting integral operators and constitutes an additional equation Φ has to fulfill.

8.3.3 Dimensionless PNP-Model

The analysis of the model is simplified by introducing a set of properly rescaled equations. First, let us choose $\Phi_A = 0$, so that Φ has to fulfill homogeneous Dirichlet boundary conditions at the anode. We introduce the dimensionless potential

$$\varphi = \beta q_+ \Phi \tag{8.33}$$

and express densities in units of the fixed average anion density

$$n^* = \frac{1}{\mathrm{Vol}}\int n_- dV, \quad c_i = n_i/n^*, \quad i \in \{+,0,-\}. \tag{8.34}$$

As control parameter we use the potential at the cathode

$$\eta := \varphi\big|_{\Gamma_C} . \tag{8.35}$$

As unit of length we choose the distance between the electrodes, H. Assuming $D_+ = D_- \equiv D$ time is measured in units of H^2/D, i.e. $t = H^2\tau/D$ and τ is the dimensionless time. Then Eqs. (8.10) and (8.11) take on the dimensionless form

$$\partial_\tau c_+ = \nabla\cdot(\nabla c_+ + c_+\nabla\varphi) , \tag{8.36}$$
$$\partial_\tau c_0 = \nabla^2 c_0 , \tag{8.37}$$
$$\partial_\tau c_- = \nabla\cdot(\nabla c_- - c_-\nabla\varphi) . \tag{8.38}$$

We introduce a material-dependent Debye-Hückel length

$$\lambda_r^* = \sqrt{\frac{\varepsilon_0 \varepsilon_r}{\beta q_+^2 n^*}}, \quad r \in \{P, S\} \tag{8.39}$$

which is used to introduce the ratio of the squares of the Debye-Hückel length and inter-electrode distance as dimensionless dielectric permittivity

$$\varepsilon_r := \left(\frac{\lambda_r^*}{H}\right)^2 . \tag{8.40}$$

Using $\rho = \rho/(q_+ n^*)$ as dimensionless protein charge density the dimensionless potential follows from

$$-\nabla \cdot (\varepsilon_r(\mathbf{r}) \nabla \varphi) = \chi_{\Omega_S}(c_+ - c_-) + \rho \,. \tag{8.41}$$

This equation explicitly keeps the spatial dependence of the permittivity ε_r which models the protein implicitly by a local change of the dielectric properties of the bulk. The choice of units implies that current densities are given in units of Dn^*/H, corresponding to the Nernst diffusion limited current. In the boundary conditions for the current densities at the electrodes, Eqs.(8.19) and (8.20), the rates K_R and K_O have to be rescaled as follows:

$$k_f = K_f L/D, \quad f \in R, O. \tag{8.42}$$

After rescaling the boundary conditions and restricting the computational domain to Ω_S the complete system is

$$\begin{aligned}
\partial_t c_+ - \nabla^2 c_+ - \nabla \cdot (c_+ \nabla \varphi) &= 0 && (8.43)\\
\partial_t c_0 - \nabla^2 c_0 &= 0 && (8.44)\\
\partial_t c_- - \nabla^2 c_- + \nabla \cdot (c_- \nabla \varphi) &= 0 && (8.45)\\
-\nabla \cdot (\varepsilon_r \nabla \varphi) - (c_+ - c_-) &= \rho_P \,. && (8.46)
\end{aligned}$$

$$\begin{aligned}
(\partial_{\mathbf{n}} c_j + j c_j \partial_{\mathbf{n}} \varphi)\big|_{\Gamma_0} &= 0 \quad j \in \{+, 0, -\} && (8.47)\\
\varepsilon_r \partial_{\mathbf{n}} \varphi\big|_{\Gamma_0} &= 0 && (8.48)
\end{aligned}$$

$$\begin{aligned}
-\mathbf{n} \cdot \mathbf{j}_+\big|_{\Gamma_A} = (\partial_{\mathbf{n}} c_+ + c_+ \partial_{\mathbf{n}} \varphi)\big|_{\Gamma_A} &= +k_O c_0\big|_{\Gamma_A} && (8.49)\\
\partial_{\mathbf{n}} c_0\big|_{\Gamma_A} &= -k_O c_0\big|_{\Gamma_A} && (8.50)\\
(\partial_{\mathbf{n}} c_- - c_- \partial_{\mathbf{n}} \varphi)\big|_{\Gamma_A} &= 0 && (8.51)\\
\varphi\big|_{\Gamma_A} &= 0 && (8.52)
\end{aligned}$$

$$\begin{aligned}
-\mathbf{n} \cdot \mathbf{j}_+\big|_{\Gamma_C} = (\partial_{\mathbf{n}} c_+ + c_+ \partial_{\mathbf{n}} \varphi)\big|_{\Gamma_C} &= -k_R c_+\big|_{\Gamma_C} && (8.53)\\
\partial_{\mathbf{n}} c_0\big|_{\Gamma_C} &= +k_R c_+\big|_{\Gamma_C} && (8.54)\\
(\partial_{\mathbf{n}} c_- - c_- \partial_{\mathbf{n}} \varphi)\big|_{\Gamma_C} &= 0 && (8.55)\\
\varphi\big|_{\Gamma_C} &= \eta \,. && (8.56)
\end{aligned}$$

The conditions at the protein-solvent boundary depend on the quantity we want to compute. For the various ion densities we need blocking conditions for the fluxes

$$(\partial_{\mathbf{n}} c_j + j c_j \partial_{\mathbf{n}} \varphi)\big|_{\Gamma} \quad = \quad 0 \qquad j \in \{+, 0, -\} \,. \tag{8.57}$$

For the potential we have the non-local condition

$$\varepsilon_S \partial_{\mathbf{n}} \varphi\big|_{\Gamma} \quad = \quad -\varepsilon_P t_P \tag{8.58}$$

which introduces t_P as additional unknown.

8.4 Numerical Solution

The simulations are based on our deal.II-based finite element framework which allows for higher order isoparametric elements in case of curved boundaries and adaptive mesh refinement. Basically, there are two different options for a FEM-based simulation.

Standard continuous Galerkin methods lead to a straightforward implementation but require to explicitly enforce the conservation law for the anions. An additional difficulty is the particularly low degree of structure in the stiffness matrix making an efficient parallel solution of the linear algebraic equations more demanding.

Interior-penalty, discontinuous Galerkin (DG) methods [15, 14, 16] by design lead to a conservative discrete problem. They are particularly well suited to combining local mesh adaption with geometric multigrid methods on meshes with hanging nodes [69] since the transfer operators can be constructed as local operation on the individual cell and do not need to work on patches of cells. Compared to standard globally continuous Galerkin methods their disadvantage is a larger number of DoFs per cell and more complex variational formulations because of the explicit consideration of inter-cell fluxes.

Unlike for the continuous approach the FEM-BEM coupling for DG methods is still a topic of current research [49, 48, 103]. The pioneering work for the continuous case has been done by Johnson and Nédélec [67], Bielak and Macamy [27] and Costabel [35]. Convergence and stability of the Johnson-Nédélec coupling for polygonal surfaces has been proven by Sayas [114].

8.4.1 Weak Formulation

Prior to discretization we have to convert Eqs. (8.43 - 8.58) into the corresponding weak form. We define $(v, w)_D := \int_D vw \, dD$ to be the L^2 scalar product of two functions v, w defined on a domain D, e.g. Ω or its surface Γ and dD is the corresponding measure. All equations are of advection-diffusion-reaction type, i.e.

$$-\nabla \cdot (a(\mathbf{x}) \nabla u) + \nabla \cdot (bu) + cu \quad = \quad f \tag{8.59}$$

where u denotes the scalar solution, $a(\mathbf{x})$ a diffusion coefficient function, $\mathbf{b}$ some prescribed flow field, c is a reaction rate and f some external stimulus. Multiplying with an arbitrary scalar test function v from the left and integration by parts over a domain D with boundary ∂D and normal derivative $\partial_{\mathbf{n}} = \mathbf{n} \cdot \nabla$ gives the standard weak form

$$\begin{aligned} & (\nabla v, a(\mathbf{x}) \nabla u)_D + (\nabla v, \mathbf{b} u)_D + (v, cu)_D \\ & -(v, a(\mathbf{x}) \partial_{\mathbf{n}} u)_{\partial D} - (v, \mathbf{b} \cdot \mathbf{n} \, u)_{\partial D} \quad = \quad (v, f)_D \,. \end{aligned} \tag{8.60}$$

Finite element methods (FEM) require to start with defining a suitable function space for the solution. The Dirichlet boundary conditions for the potential on the electrodes are built into the solution space

$$X := \{(c_+, c_0, c_-, \varphi) \in [H^1(\Omega_S)]^4 : \ \varphi|_{\Gamma_A} = 0, \ \varphi|_{\Gamma_C} = \eta\}$$

so that Dirichlet boundary conditions are already incorporated. For later purposes we consider X as a direct product of a space for the concentrations and a separate one for the potential

$$\begin{aligned} X &= X^c \times X^\varphi , \\ X^c &:= [H^1(\Omega_S)]^3 , \\ X^\varphi &:= \{\varphi \in H^1(\Omega_S) : \ \varphi|_{\Gamma_A} = 0, \ \varphi|_{\Gamma_C} = \eta\} . \end{aligned}$$

For the boundary conditions for φ on Γ we need a separate subspace $Y := H^{-1/2}(\Gamma)$ so that the complete solution space is

$$V := X \times Y . \tag{8.61}$$

For the solution increments in the Newton iteration we need homogenized spaces

$$\begin{aligned} X_0 &:= X^c \times X_0^\varphi , & (8.62)\\ X_0^\varphi &:= \{\varphi \in H^1(\Omega) : \ \varphi|_{\Gamma_A} = 0, \ \varphi|_{\Gamma_C} = 0\} , & (8.63)\\ V_0 &:= X_0 \times Y . & (8.64) \end{aligned}$$

Using a vector-valued test function $(s, u, v, w, \psi) \in V_0$ we get as weak form

$$\begin{aligned} (\nabla s, \nabla c_+)_{\Omega_S} + (\nabla s, c_+ \nabla \varphi)_{\Omega_S} & \\ -(s, \partial_{\mathbf{n}} c_+ + c_+ \partial_{\mathbf{n}} \varphi)_{\Gamma_S} &= 0 & (8.65)\\ (\nabla u, \nabla c_0)_{\Omega_S} - (u, \partial_{\mathbf{n}} c_0)_{\Gamma_S} &= 0 & (8.66)\\ (\nabla v, \nabla c_-)_{\Omega_S} - (\nabla v, c_- \nabla \varphi)_{\Omega_S} & \\ -(v, \partial_{\mathbf{n}} c_- - c_- \partial_{\mathbf{n}} \varphi)_{\Gamma_S} &= 0 & (8.67)\\ (\nabla w, \varepsilon_S \nabla \varphi)_{\Omega_S} - (w, c_+ - c_-)_{\Omega_S} & \\ -(w, \varepsilon_S \partial_{\mathbf{n}} \varphi)_{\Gamma_S} &= 0 . & (8.68) \end{aligned}$$

Inserting the boundary conditions for the fluxes we get

$$\begin{aligned}(\nabla s, \nabla c_+)_{\Omega_S} + (\nabla s, c_+ \nabla \varphi)_{\Omega_S}& \\ -(s, +k_O c_0)_{\Gamma_A} - (s, -k_R c_+)_{\Gamma_C} &= 0\end{aligned} \tag{8.69}$$

$$\begin{aligned}(\nabla u, \nabla c_0)_{\Omega_S}& \\ -(u, -k_O c_0)_{\Gamma_A} - (u, +k_R c_+)_{\Gamma_C} &= 0\end{aligned} \tag{8.70}$$

$$(\nabla v, \nabla c_-)_{\Omega_S} - (\nabla v, c_- \nabla \varphi)_{\Omega_S} = 0 \tag{8.71}$$

$$\begin{aligned}(\nabla w, \varepsilon_S \nabla \varphi)_{\Omega_S} - (w, c_+ - c_-)_{\Omega_S}& \\ +(w, \varepsilon_P t_P)_\Gamma &= 0\end{aligned} \tag{8.72}$$

$$\left(\psi, \left(\frac{1}{2}I - K\right)\varphi\right)_\Gamma + \left(\psi, \frac{\varepsilon_S}{\varepsilon_P} V t_P\right)_\Gamma = \left(\psi, \phi^C\right)_\Gamma . \tag{8.73}$$

Note that in its Galerkin formulation the boundary condition for the electric field on the protein surface introduces double surface integrals. This can be avoided by using collocation methods. We abbreviate the FEM part of the solution as

$$\mathbf{u} := (c_+, c_0, c_-, \varphi)$$

and the corresponding test function as $\mathbf{v} := (s, u, v, w)$. Adding up Equations (8.69)-(8.72) except for the interface term defines a semilinear form

$$a(\cdot;\cdot) : X \times X_0 \to \mathbb{R}$$

which is nonlinear in its first argument. This allows us to write the variational problem in a compact form:

Find $(\mathbf{u}, t_P) \in X \times Y$ *s. t.*

$$\begin{aligned}&\forall \mathbf{v} \in X_0 : \\ &\quad a(\mathbf{u}; \mathbf{v}) + (w, \varepsilon_P t_P)_\Gamma = 0\end{aligned} \tag{8.74}$$

$$\begin{aligned}&\forall \psi \in Y : \\ &\left(\psi, \left(\tfrac{1}{2}I - K\right)\varphi\right)_\Gamma + \left(\psi, \tfrac{\varepsilon_S}{\varepsilon_P} V t_P\right)_\Gamma = \left(\psi, \phi^C\right)_\Gamma .\end{aligned} \tag{8.75}$$

8.4.2 FEM Discretization

In case of the standard continuous Galerkin method with globally continuous Lagrange elements of polynomial order q discretization is straightforward by choosing a finite set of test and ansatz functions $\{(s, u, v, w, \psi)\} \subset V_h$ from a finite-dimensional subspace $V_h \subset V$ of which

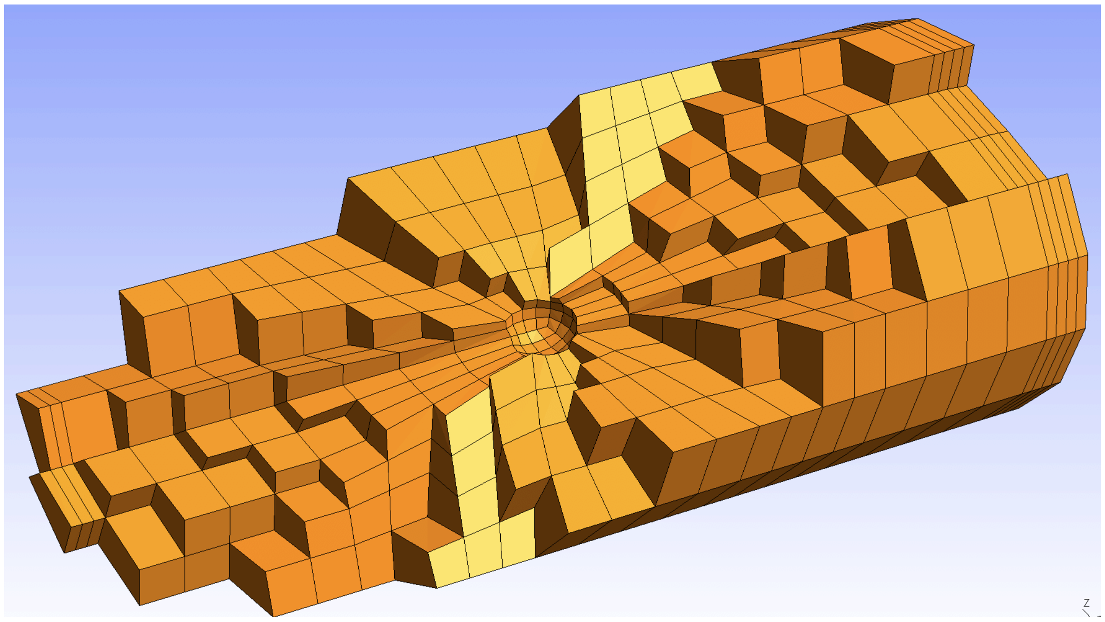


Figure 8.2: Cut through the hexahedral mesh used as coarse mesh.

the dimension $N := 4M + N_P$ depends on the average mesh width h. The number of DoFs per solution component in the FEM part is M and the number of DoFs on the dielectric interface is N_P. To obtain the discretized variational form the solution is considered as a linear combination of the ansatz functions

$$\begin{aligned} c_{\alpha,h} &= \sum_j c_{\alpha,j} s_{\alpha,j}, \quad \alpha \in \{+,0,-\}, && (8.76)\\ \varphi_h &= \sum_j \varphi_j w_j, && (8.77)\\ t_{P,h} &= \sum_j t_{P,j} \psi_j. && (8.78) \end{aligned}$$

For a convenient treatment of the coupling between Ω_P and Ω_S we consider X_h^φ as a direct sum of two subspaces which single out the DoFs of φ_h on Γ (symbolized by white dots in Fig. 8.3):

$$\begin{aligned} X_h^\varphi &= X_h^{\varphi,\Omega_S} \oplus X_h^{\varphi,\Gamma}, \\ X_h^{\varphi,\Omega_S} &:= \operatorname{span}\left\{ w_i \in X_h^\varphi \;:\; w\big|_\Gamma = 0 \right\}_{i=1}^{M_S}, \\ X_h^{\varphi,\Gamma} &:= \operatorname{span}\left\{ w_i \in X_h^\varphi \;:\; w\big|_\Gamma \neq 0 \right\}_{i=M_S+1}^{M}. \end{aligned}$$

The number of DoFs for the potential on the interface is $N_P = M - M_S$. We use an analogous decomposition for $X_{0,h}^\varphi$, i.e. $X_{0,h}^\varphi = X_{0,h}^{\varphi,\Omega_S} \oplus X_{0,h}^{\varphi,\Gamma}$. For the Galerkin discretization of the weak forms of the boundary integral equation (8.73) we use the finite dimensional ansatz space

$$Y_h := \operatorname{span}\{\psi_k\}_{k=1}^{N_P} \subset H^{-1/2}(\Gamma) = Y.$$

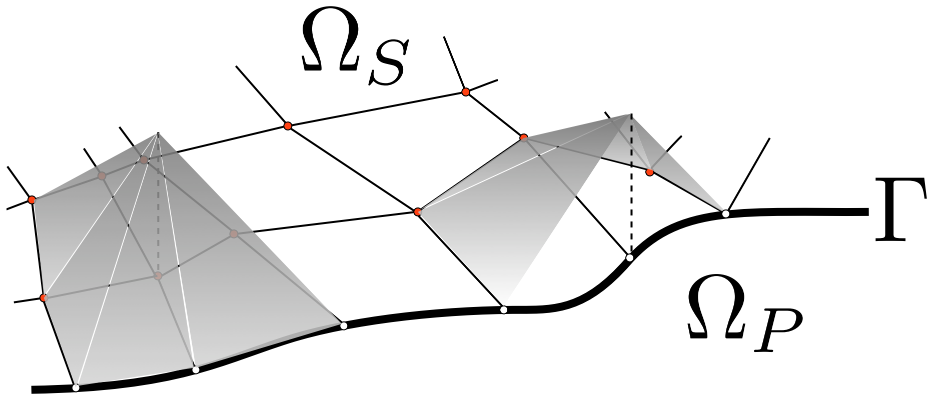


Figure 8.3: Red dots indicate interior degrees of freedom (DoFs) and white points those on the protein-solvent interface Γ. The trial functions associated with interior DoFs are zero on Γ.

Solving the linear algebraic problem amounts to determine the coefficient vectors

$$\begin{aligned}
\underline{c}_\alpha &:= (c_{\alpha,0},\dots,c_{\alpha,M-1}), \quad \alpha\in\{+,0,-\},\\
\underline{\varphi}_S &:= (\varphi_0,\dots,\varphi_{M_S-1})\\
\underline{\varphi}_P &:= (\varphi_{M_S},\dots,\varphi_{M_S+N_P-1}),\\
\underline{t}_P &:= (t_0,\dots,t_{N_P-1}),\\
\underline{c} &:= (\underline{c}_+,\underline{c}_0,\underline{c}_-).
\end{aligned}$$

We split φ_h into φ_S and φ_P. The latter is the link between the FEM and BEM discretization. The various ionic concentrations can be treated by FE-only methods. Their contribution to the overall solution is collectively abbreviated as $\mathbf{c} := (c_{+,h}, c_{0,h}, c_{-,h}) \in X_h^c$ with corresponding test function $\mathbf{d}\in X_h^c$ so that $\mathbf{v} = (\mathbf{d},w)\in X_{0,h}$. For the potential we have a part $\varphi_S \in X_h^{\varphi,\Omega_S}$ treated by FEM and a contribution $\varphi_P \in X_h^{\varphi,\Gamma}$ defined solely on the protein surface. The discrete variational problem is:

Find $(\mathbf{c},\varphi_S,\varphi_P,t_{P,h}) \in X_h^c \times (X_h^{\varphi,\Omega_S}\oplus X_h^{\varphi,\Gamma})\times Y_h$ *s. t.*:

$$\begin{aligned}
&\forall \mathbf{v}\in X_{0,h} \ : \\
&a((\mathbf{c},\varphi_S);(\mathbf{d},w)) + (w,\varepsilon_P t_{P,h})_\Gamma \quad = \ 0
\end{aligned} \tag{8.79}$$

$$\begin{aligned}
&\forall \psi \in Y_h \ : \\
&\left(\psi,\left(\tfrac{1}{2}I-K\right)\varphi_P\right)_\Gamma + \left(\psi,\tfrac{\varepsilon_S}{\varepsilon_P}V t_{P,h}\right)_\Gamma \quad = \ (\psi,\phi^C)_\Gamma\,.
\end{aligned} \tag{8.80}$$

The resulting (non-)linear [1] algebraic system has a 4×4 block structure

$$\begin{pmatrix}
A^c & A^{c,\varphi_S} & A^{c,\varphi_P} & \\
A^{\varphi_S,c} & A^{\varphi_S} & A^{\varphi_S,\varphi_P} & \\
A^{\varphi_P,c} & A^{\varphi_P,\varphi_S} & A^{\varphi_P} & -M_h^T \\
 & & \frac{1}{2}M_h - K_h & \frac{\varepsilon_S}{\varepsilon_P}V_h
\end{pmatrix}
\begin{pmatrix} \underline{c} \\ \underline{\varphi}_S \\ \underline{\varphi}_P \\ \underline{t}_P \end{pmatrix}
=
\begin{pmatrix} \underline{0} \\ \underline{0} \\ \underline{0} \\ \underline{\phi}^C \end{pmatrix}. \tag{8.81}$$

[1] We postpone this issue to the discussion the solution of the equations.

The individual matrix entries are computed as follows. The submatrix A^c contains the entries of the subsystem of the concentrations of ions, i.e. their diffusion and the reactive boundaries:

$$\begin{aligned}
&\forall\, i,j = 0,\dots,M-1: \\
B_{ij} &= \begin{pmatrix} -(s_i,-k_R s_j)_{\Gamma_C} & -(s_i,+k_O u_j)_{\Gamma_A} & 0 \\ -(u_i,+k_R s_j)_{\Gamma_C} & -(u_i,-k_O u_j)_{\Gamma_A} & 0 \\ 0 & 0 & 0 \end{pmatrix} \\
A^c_{ij} &= B_{ij} + \begin{pmatrix} (\nabla s_i,\nabla s_j)_{\Omega_S} & & \\ & (\nabla u_i,\nabla u_j)_{\Omega_S} & \\ & & (\nabla v_i,\nabla v_j)_{\Omega_S} \end{pmatrix}.
\end{aligned}$$

The off-diagonal blocks A^{c,φ_S} and A^{c,φ_P} represent the drift of the ions because of the local gradient of the electrostatic potential in the bulk (DoFs $0,\dots,M_S-1$) and on the dielectric interface (DoFs $M_S,\dots,M-1$):

$$\begin{aligned}
&\forall\, i = 0,\dots,M-1: \\
&\begin{pmatrix} (\nabla s_i, c_{+,h}, \nabla w_j)_{\Omega_S} \\ 0 \\ -(\nabla v_i, c_{-,h}, \nabla w_j)_{\Omega_S} \\ 0 \end{pmatrix} = \begin{cases} A^{c,\varphi_S} & j = 0,\dots,M_S-1 \\ A^{c,\varphi_P} & j = M_S,\dots,M-1 \end{cases}.
\end{aligned}$$

The off-diagonal blocks $A^{\varphi_S,c}$ and $A^{\varphi_P,c}$ represent the interactions of the bulk electrostatic potential (DoFs $0,\dots,M_S-1$) and on the dielectric interface (DoFs $M_S,\dots,M-1$) with the ions:

$$\begin{aligned}
&\forall\, j = 0,\dots,M-1: \\
&\begin{pmatrix} -(w_i,s_j)_{\Omega_S} & 0 & (w_i,v_j)_{\Omega_S} & 0 \end{pmatrix} = \begin{cases} A^{\varphi_S,c} & i = 0,\dots,M_S-1 \\ A^{\varphi_P,c} & i = M_S,\dots,M-1 \end{cases}.
\end{aligned}$$

The FEM part of the Poisson equation on the interface is

$$(\nabla w_i, \varepsilon_S \nabla w_j)_{\Omega_S} = \begin{cases} A^{\varphi_S} & i,j = 0,\dots,M_S-1 \\ A^{\varphi_P} & i,j = M_S,\dots,M-1 \\ A^{\varphi_S,\varphi_P} & i = 0,\dots,M_S-1,\ j = M_S,\dots,M-1 \\ A^{\varphi_P,\varphi_S} & i = M_S,\dots,M-1,\ j = 0,\dots,M_S-1 \end{cases}.$$

The components of the Galerkin version of the BEM part on the dielectric interface are:

$$\begin{aligned}
&\forall\, i,j = M_S,\dots,M-1: \\
-M_h^T &= (w_i,\psi_j)_\Gamma\,, \\
K_h &= \left(\psi_i,\left(\frac{1}{2}-K\right)w_j\right)_\Gamma \\
&= \frac{1}{2}(\psi_i,w_j)_\Gamma - \left(\psi_i,\oint_\Gamma \partial_{\mathbf{n}'}G_{\mathbf{x}}(\mathbf{x}')w_j(\mathbf{x}')d\Gamma(\mathbf{x}')\right)_\Gamma \\
&= \frac{1}{2}(\psi_i,w_j)_\Gamma - (\psi_i,(\partial_{\mathbf{n}'}G_{\mathbf{x}},w_j)_\Gamma)_\Gamma\,, \\
V_h &= (\psi_i,V\psi_j)_\Gamma \\
&= \left(\psi_i,\oint_\Gamma G_{\mathbf{x}}(\mathbf{x}')\psi_j(\mathbf{x}')d\Gamma(\mathbf{x}')\right)_\Gamma = (\psi_i,(G_{\mathbf{x}},\psi_j)_\Gamma)_\Gamma\,.
\end{aligned}$$

For collocation we treat test and trial functions differently and use $(\psi_i,\cdot)_\Gamma = (\delta(\mathbf{x}-\mathbf{x}_i),\cdot)_\Gamma$ in the formulas for V_h and K_h, whereas M_h^T is assembled as non-diagonal boundary mass matrix.

Discretization of Boundary Integral Equation

Using the ansatz for the potential, Eq. (8.77) and the interface DoFs as collocation points we can immediately write down the discretization of the integral operators at the dielectric interface. Let $S_i \subset \Gamma$ denote the support of the trace of ansatz function w_i and E the part of the surface of cell K which is part of the interface.

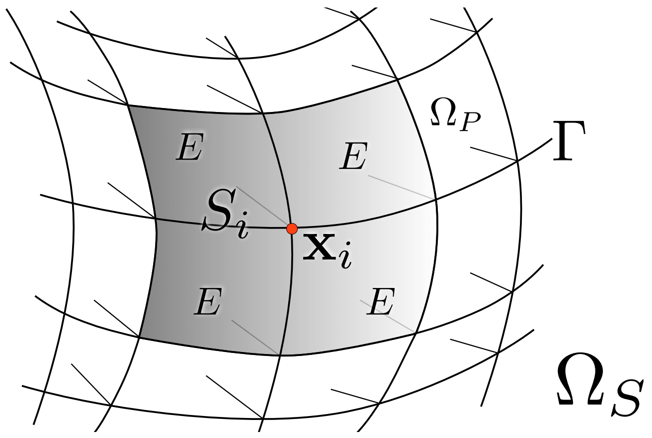


Figure 8.4: The red dot indicates support point $\mathbf{x}_i$ on surface Γ. The support of the trace of a trial function is S_i (shaded area) which is the union of all cell surfaces containing $\mathbf{x}_i$.

For a given support point $\mathbf{x}_i$ of a degree of freedom φ_i we have $S_i = \cup_{\mathbf{x}_i \in E} E$. Using the fact that for continuous Galerkin methods $w_i(\mathbf{x}_j) = \delta_{ij}$ we get for the interface mass matrix

$$M_{h,ij} = -(\delta(\mathbf{x}-\mathbf{x}_i), w_j)_\Gamma = -\delta_{ij}\,. \tag{8.82}$$

The single-layer operator becomes

$$V_{h,ij} = (\delta(\mathbf{x}-\mathbf{x}_i), V\psi_j)_\Gamma = (G_{\mathbf{x}_i}, \psi_j)_\Gamma \tag{8.83}$$

$$= \oint\limits_{S_j \subset \Gamma} G_{\mathbf{x}_i}(\mathbf{x}')\psi_j(\mathbf{x}')d\Gamma(\mathbf{x}')\,. \tag{8.84}$$

Similarly, we get for the double-layer operator

$$K_{h,ij} = \frac{1}{2}\delta_{ij} - \oint\limits_{S_j \subset \Gamma} \partial_{\mathbf{n}'} G_{\mathbf{x}}(\mathbf{x}') w_j(\mathbf{x}') d\Gamma(\mathbf{x}')\,. \tag{8.85}$$

The right-hand side due to the Newton potential is

$$\phi_i^C = \frac{1}{4\pi\varepsilon_P}\sum_k \frac{q_k}{|\mathbf{x}_i - \mathbf{x}_k|}\,. \tag{8.86}$$

The final step is to replace integration by quadrature. In the following $\mathbf{x}'_a$ is the ath quadrature point. We introduce the abbreviations $H_{ia} := \frac{\partial G_{\mathbf{x}_i}}{\partial \mathbf{n}'}(\mathbf{x}'_a)$, $G_{ia} := G_{\mathbf{x}_i}(\mathbf{x}'_a)$ and, following *deal*.II's notation, JxW_a for the product of the Jacobian of the transformation to the reference element and the quadrature weight at $\mathbf{x}'_a$. Due to the curvilinear boundary JxW_a is a function of $\mathbf{x}'_a$. The function values of the ansatz functions at the quadrature points are abbreviated as $\psi_{aj} := \psi_j(\mathbf{x}'_a)$ and $w_{aj} := w_j(\mathbf{x}'_a)$. The contributions of the single- and double-layer operator are then

$$V_{h,ij} = \sum_{E \subset S_j}\sum_{a \in E} G_{ia}\psi_{aj} JxW_a\,, \tag{8.87}$$

$$K_{h,ij} = \frac{1}{2}\delta_{ij} - \sum_{E \subset S_j}\sum_{a \in E} H_{ia} w_{aj} JxW_a\,. \tag{8.88}$$

This shows that the assembly of the cell contributions to the global matrix elements is given by matrix-matrix products which is one of the most optimized routines in computer science.

The factor $1/2$ in Eq. (8.88) is valid only on continuously differentiable boundaries. In case a polynomial approximation of the boundary is used $1/2$ has to be replaced by a position-dependent factor $\alpha(\mathbf{x}_i)$, $\mathbf{x}_i \in \Gamma$ which measures the fraction of the interior solid angle subtended by the cell faces having $\mathbf{x}_i$ as common boundary point.

Without further action BEM matrices are dense and the computational effort for assembly and storage grows quadratically with the number of boundary DoFs. To save memory one could employ the method of adaptive cross approximation with partial pivoting, for an introduction see [110, Chapter 3] and references therein. In cases of practical interest this type of lossy matrix compression allows to save roughly 50-80% of memory without sacrificing accuracy.

8.4.3 Parallelization Strategy

Discretization of the FEM Part

For the assembly of the finite element part of the problem it is better to use a coarse-grained parallelization based on pthreads or MPI (depending the problem size). This is largely due to the fact that FEM matrices on general meshes have very unstructured sparsity pattern and the final step of adding a cell's contribution to the global stiffness matrix means a lot random memory accesses which cannot be coalesced.

Discretization of the BEM Part

The assembly of the boundary integral equation is well-suited for parallelization by CUDA. The fact that all combinations of collocation points and trial functions have to be considered implies regular memory access patterns. To understand where exactly CUDA pays off let us begin with a complexity analysis of the individual steps in the evaluation of Eqs. (8.87) and (8.88) assembling the matrix repesentation of the single- and double layer operator, respectively.

Let n_E be the number of DoFs and $n_{E,q}$ the number of quadrature points per surface element E. In three dimensions typical values are $n_E = 4$ (one DoF per corner; linear elements) and $n_{E,q} = \mathscr{O}(10^s)$, $s \in (1,2)$ for Gauss quadrature rules with four to eight quadrature points per coordinate. To estimate the number of DoFs on the protein-solvent interface N_P recall that in the coarse grid the cavity is just a box with six faces which only becomes a sphere by employing curvilinear boundary approximations. After the rth mesh refinement each coarse grid face is subdivided into 4^r faces. For instance, $r = 6$ gives 6144 surface elements and thus $N_P = \mathscr{O}(10^4)$. For typical orders of finite elements this can also be considered as the number of surface elements on the protein-solvent interface Γ. On each cell the matrices G_{ia} and H_{ia} have to be computed first. This is of order $\mathscr{O}(N_P n_{E,q})$ per cell, i.e. $\mathscr{O}(N_P^2 n_{E,q})$ for the whole interface. The amount of data needed for these interaction terms is only $N_P + n_{E,q}\mathscr{O}(N_P)$ which follows from the number collocation points, cells and quadrature points per cell. Similarly, there are $n_{E,q}\mathscr{O}(N_P)$ function values of the trial functions at the quadrature points of all cells and $n_{E,q}\mathscr{O}(N_P)$ associated Jacobian-weighted quadrature weights. Hence, the amount of data to prepare for the assembly is $\mathscr{O}(N_P)$.

To take advantage of CUDA we have to copy the quadrature points $\mathbf{x}'_a$, the corresponding function values $w_j(\mathbf{x}'_a)$, $\psi_j(\mathbf{x}'_a)$ and the weights JxW_a of each cell to the GPU. These data can be copied in an incremental manner, concurrently to the computation of the contributions of other cells to the global matrix which hides some of the latency of the PCIe-Bus. The collocation points $\mathbf{x}_i$ have to be copied only once. In a practice, on current GPUs this means that this data is precomputed on the CPU by the *deal*.II-based part of the program. For the consequences of this see the discussion in Sec. 4.3 and Fig. 4.2. For a hardware-independent software-design we will end up with a class structure very similar to the one in Fig. 4.3. That is, we encounter the same programming issues although from a physical point of view the computation of phase holograms and the simulation of the Poisson-Nernst-Planck model are totally unrelated.

The evaluation of Eqs. (8.87) and (8.88) is of order $\mathscr{O}(N_P^2 n_{E,q} N_E)$. From an algorithmic point of view the computation of G_{ia} and H_{ia} is an outer product as provided by the BLAS `ger` function. For efficiency it is better to merge the preparation of G_{ia} and H_{ia}, the evaluation of the integrands and their subsequent summation in Eqs. (8.87) and (8.88) in one CUDA-Kernel. Like in the LU factorization in Sec. 2.1 we use the shared memory of an SMP as buffer for the values of G_{ia} and H_{ia} and other re-used intermediate results. Due to its limited size the global matrices V_h and K_h will be tiled where each tile will be computed by a different SMP.

A high-level description of the assembly of V_h is shown in Fig. 8.5. Colored arrows indicate the different stages of the computation which require synchronization in between. For instance, G_{ia} must have been computed before the values of the integrand $G_{ia}w_{ja}JxW_a$ can be summed to get this cell's contribution to $V_{h,ij}$. The ellipses indicate that input data needed for the computation of the values of the integrand $G_{ia}w_{ja}JxW_a$ which can be done simultaneously.

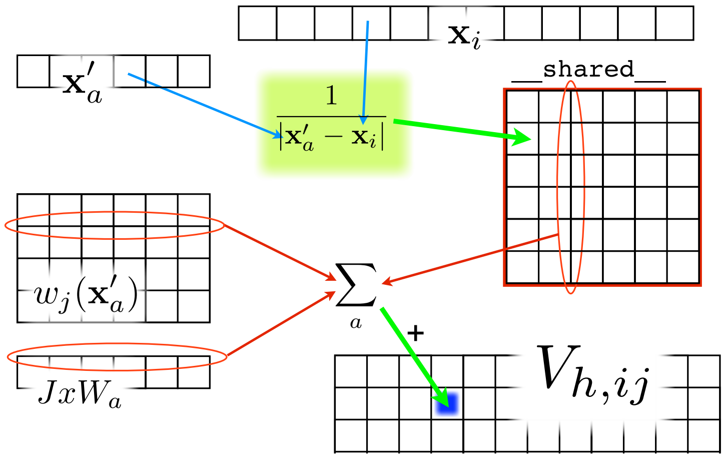


Figure 8.5: Flow chart of assembly of single-layer operator V_h, ij, Eq. (8.87).

Solving the Fully Discretized Equations

Before one can make a decision on the parallel solution of Eq. (8.81) one has to select a linearization strategy. The nonlinearities are given by the off-diagonal blocks A^{c,φ_S}, A^{c,φ_P}, $A^{\varphi_S,c}$ and $A^{\varphi_P,c}$ describing the drift of an ion in the electric field. However, the linearization strategy is not independent of the solution method and of the values of the model parameters, especially the ratio between thermal energy and electrostatic energy, cf. Sec. 8.3.3. For biological systems this ratio is typically of the order of one, i.e. neither the drift nor the diffusion terms dominate, in contrast to flow problems where the dynamics often is dictated by the drift term. Due to the very different conditioning of the FEM and BEM part of Eq. (8.81) a valid approach is to decouple them and to let them iteratively improve each other.

If diffusion is non-negligible multigrid methods are always an attractive and efficient solution method. For nonlinear problems the non-linearity can be built into the smoother (Brandt's full approximation scheme). In most cases the main components of a multigrid scheme are implemented as matrix-vector products of which we know how to parallelize them.

At the current level of algorithmic optimization the BEM part is described by a dense matrix. Therefore, it is reasonable to solve the fully discretized equations either by means of a LU-factorization of V_h or by solving the subproblem

$$V_h \underline{t}_P = \frac{\varepsilon_P}{\varepsilon_S} \underline{\phi}^C - \frac{\varepsilon_P}{\varepsilon_S} \left(\frac{1}{2} M_h - K_h \right) \underline{\phi}_P$$

iteratively by some Krylov method like GMRES, cf. Sec. 3.1. Either option is able to exploit the full parallelization power of CUDA as we have shown in part I of this thesis.

8.5 Results

A thorough analysis of the physical properties of the dielectric relaxation problem is unfortunately beyond the scope of this thesis. The reported results on dependence of the DC current on the protein conformation are preliminary and, for the sake of numerical simplicity, only for a two-dimensional system in which the FEM-BEM coupling is reduced to the Newton potential. Nevertheless, the postulated effect is observable in the simulations. The three-dimensional case is severely hampered by the slow solving of the fully discretized, nonlinear problem and lags behind the level achieved in the modeling process.

The technical part of the results summarizes the correctness of the FEM-BEM coupling with focus on the convergence properties of the boundary element method. A correct implementation of the FEM-BEM coupling has been the most difficult subtask.

8.5.1 Proof of Concept for the Physical Model

Before we started the protein computations we verified that the simulation reproduces the linear, battery-like *I-V* curve one would expect in absence of a protein. As a first test we model the protein as a sphere with a surface potential $\Phi_{PS} = \phi_D|_\Gamma$ given by a dipole, cf. Fig. 8.6, which allows us to restrict all equations to Ω_S and measure the current I_C as a function of Φ_{PS}. In this case the internal protein conformation is synonymous with the value of the surface potential. Using a ubiquitin molecule centered at the origin as example we assume for the dipole moment of the protein 240 D which corresponds to two elementary charges of opposite sign 5 nm apart. The different protein conformations are modeled by changing the distance of the two charges. The charges are placed inside the protein at the positions $\mathbf{p}_0 = (.125s, 0)$ and $\mathbf{p}_1 = (-.125s, 0)$. By changing parameter $s \in [-.5, +.5]$ a continuum of protein conformations is realized. The range of s is sampled in an equidistant manner in order to figure out whether the DC current predictably depends on the protein conformation.

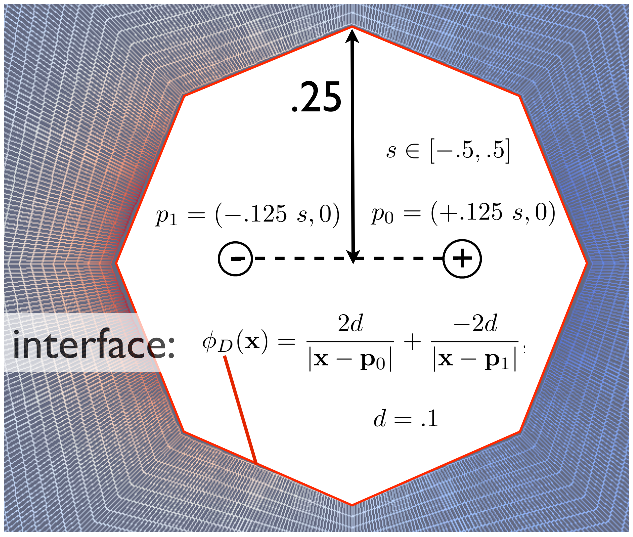
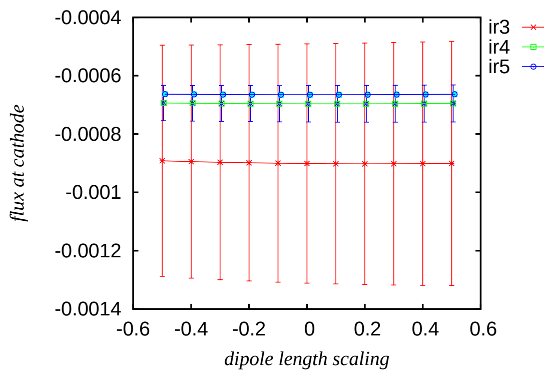

.25
$s \in [-.5, .5]$
$p_1 = (-.125\ s, 0)$
$p_0 = (+.125\ s, 0)$
−
+
interface:
$\phi_D(\mathbf{x}) = \frac{2d}{|\mathbf{x}-\mathbf{p}_0|} + \frac{-2d}{|\mathbf{x}-\mathbf{p}_1|}$
$d = .1$

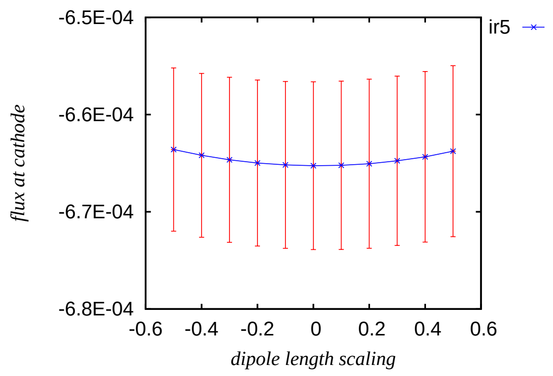
Figure 8.6: Setup of the 2D test case. The mesh is colored according to the values of the electrostatic potential in the surrounding solvent.

The current I_C through the cathode is computed from the relation given in Eq. (8.14). To check global current conservation the current I_A at the anode is computed in an analogous manner. If the current is conserved the difference $I_C - I_A$ should vanish. In a numerical simulation the error should scale like some power of the mesh width with a degree which depends on the approximation order of the chosen finite element. Figure 8.7 shows the behavior of I_C as function of the dipole length scaling s and its dependence on the mesh refinement (Fig. 8.7a) and the reduction rate (Fig. 8.7c). In all simulations we use cubic Lagrange elements, equal rates $k_R = k_O$, the potential difference $\eta = .5$ and an average anion density $n^* = .1$. The behavior of the error in the current conservation behaves as expected, i.e. refining the mesh reduces the error while the current saturates. The dependence of the current rate on the reduction is not a surprise either: Increasing the reduction rate gives larger currents.

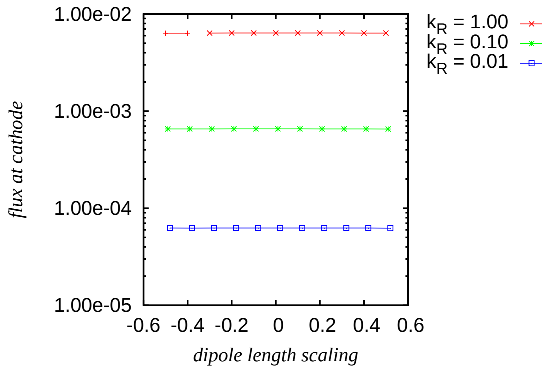
Under the assumption that the error estimate $I_C - I_A$ for the violation of the global current conservation is too pessimistic Fig. 8.7d shows that the bare effect of the protein conformation is of the order of 10^{-6} which corresponds to roughly 1% of the measured current and 10% of the estimated error which follows from Fig. 8.7c. The potential difference between the electrodes induces a symmetry breaking thereby the dipole configurations for s should not be the same as for $-s$ and this should be reflected in the currents. Indeed, Fig. 8.7d shows that the current values for $s = -.5$ is higher than the one for $s = .5$.



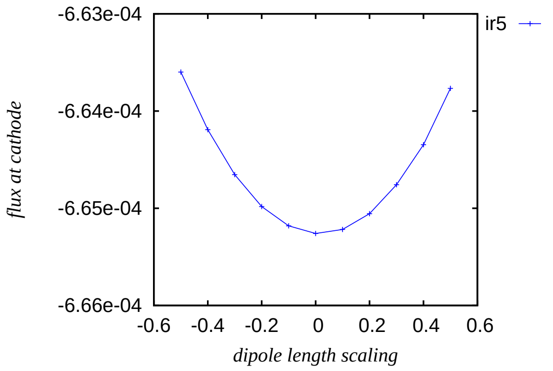
(a) I_C with $I_C - I_A$ as error bars for different mesh widths.



(b) I_C with $I_C - I_A$ as error bars for the finest mesh.



(c) I_C without errors for different k_R.



(d) I_C without errors for $k_R = .1$.

Figure 8.7: Current at the cathode against surface potential of the protein for different mesh refinements and reduction rates k_R. The key ir$d$ indicates that the coarse mesh has been globally refined d times.

The effects of the changing dipole length should also be visible in the distribution of the cations around the protein. The cation distributions for $s = -.5$, $s = 0$ and $s = +.5$ are shown in Fig. 8.8. The electrodes are on the left and right, respectively. Reorienting the dipole swaps the position of the local maximum and minimum of the cation distribution at the protein surface.

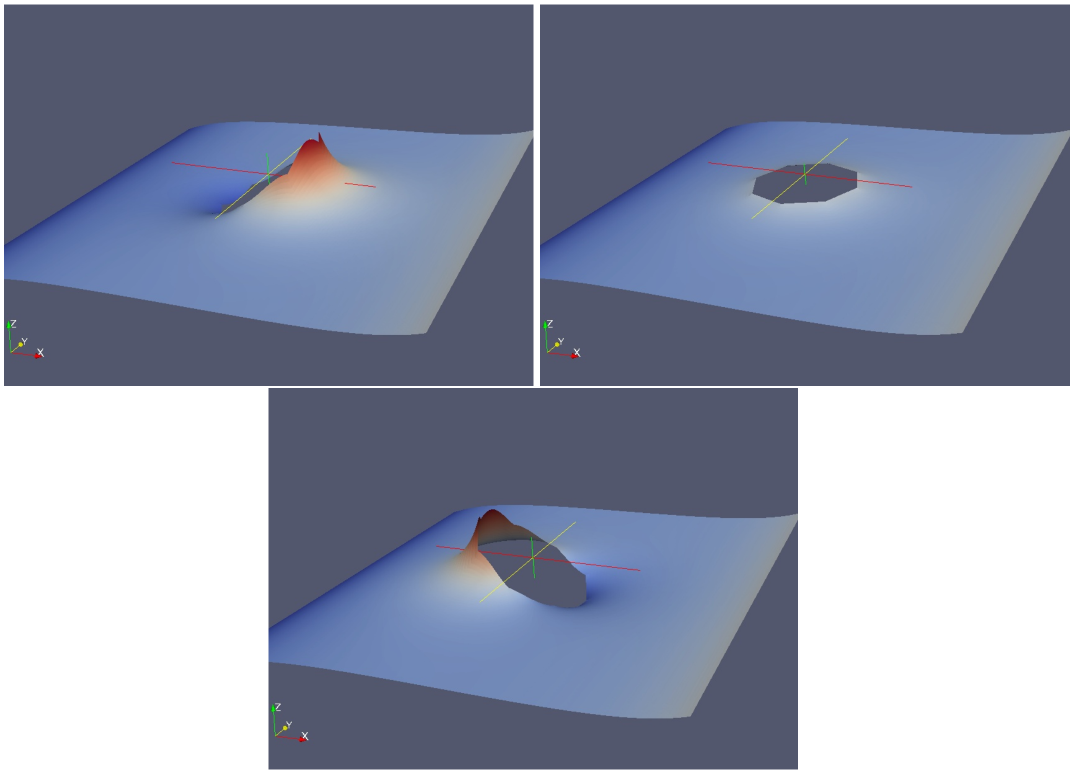


Figure 8.8: Distribution of cations for cubic Lagrange elements, $k_R = k_O = .1$, $\eta = .5$, $n^* = .1$. Top: $s = -.5$, middle: $s = 0$ (uncharged protein), bottom: $s = +.5$ (reversed dipole).

Let us end the discussion of the physical results with a word of caution on the physical valididity of the two-dimensional case. The whole system of model equations, Eqs. (8.43)-(8.58), is basically a set of Poisson equations coupled to each other via the drift terms. To simplify the discussion ignore for a moment that the drift terms cannot be ignored. The crucial point is that in 2D the long-range behavior of the fundamental solution of a Poisson equation is very different from 3D. In 2D the solution decays logarithmically which is even slower than the $1/r$ behavior in 3D which leads to more extended boundary layers at electrochemical interfaces and a less precise notion of the thickness of these layers. This, in turn, blurs the meaning of the different length scales used in the dimension analysis in Sec. 8.3.3.

Nevertheless, the 2D test case hints already at all the physical features one might expect although the setup is oversimplified in some aspects. The current depends on the conformation. The symmetric change in the dipole length leads to an asymmetric dependence of the current. The extrema of the cation distribution at the protein surface follow the surface potential of the dummy molecule and vanish for zero dipole strength which can be considered as change in the ability of the protein to temporarily store mobile ions in its hydration shell.

8.5.2 Validation of the FEM-BEM Coupling

It should be clear by now, the cumbersome part in this project is the FEM-BEM coupling at the dielectric interface. To verify that the implementation is correct we use a sequence of test problems to assess the quality of the BEM part, the FEM part and finally the coupled problem. For testing purposes the goal is to solve the Poisson problem, Eq. (8.23), in its integral form, Eq. (8.24), on a refined version of the mesh from Fig. 8.2. The final results of the FEM-BEM test for the two selected test functions (given below) are shown in Figs. 8.9 and 8.10. The exact solution is depicted by iso-surfaces and the numerical solution as points. Since the points are almost on the iso-surfaces the exact solution has been successfully recovered. Deviations of the points from the iso-surfaces are partly due to further approximation errors in the iso-surface computation during the visualization process.

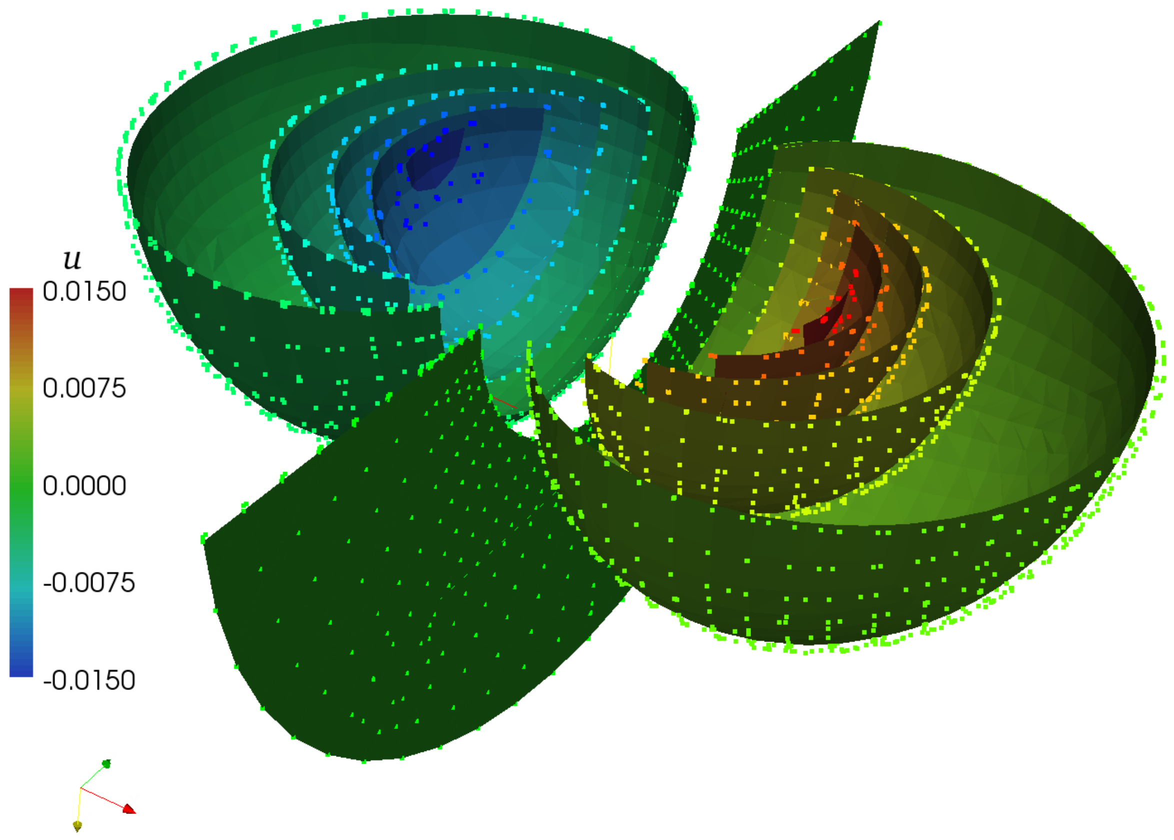


Figure 8.9: Numerical solution of the dipole testcase (points), Eq. (8.89) vs. exact solution u (isosurfaces). The solution is approximated by linear elements. The boundary is approximated by quartic polynomials and matrices are assembled with 8-point Gauss quadrature rules.

Test Cases

As reference functions we choose the bare dipole potential

$$u_D = \frac{1}{4\pi|\mathbf{x}-\mathbf{x}_0|} - \frac{1}{4\pi|\mathbf{x}-\mathbf{x}_1|}, \tag{8.89}$$

and the sum of the dipole potential and a harmonic function

$$u_{HD} = \frac{1}{4\pi|\mathbf{x}-\mathbf{x}_0|} - \frac{1}{4\pi|\mathbf{x}-\mathbf{x}_1|} + \frac{1}{10}(2x+y+z) + \frac{1}{100}xyz. \tag{8.90}$$

In case of the dipole potential reference solution and Newton potential are identical. The parameters for the dielectric constants and charges are chosen such that both u_D and u_{HD} solve

$$\begin{aligned} & \mathit{Find}\ \Phi : \Omega_S \to \mathbb{R}\ s.t.: \\ -\nabla^2\Phi &= \delta(\mathbf{x}-\mathbf{x}_0) - \delta(\mathbf{x}-\mathbf{x}_1), \\ \mathbf{x}_0 &= (+.5,0,0) \\ \mathbf{x}_1 &= (-.5,0,0) \end{aligned} \tag{8.91}$$

$$\Big(\alpha(\mathbf{x})I - K\Big)\Phi\big|_\Gamma + V\partial_\mathbf{n}\Phi\big|_\Gamma = \phi^C \equiv u \tag{8.92}$$

$$\Phi\big|_{\Gamma_A\cup\Gamma_0\cup\Gamma_C} = u\big|_{\Gamma_A\cup\Gamma_0\cup\Gamma_C}. \tag{8.93}$$

BEM

To validate the BEM part we solve for $u \equiv u_D$ and $u \equiv u_{HD}$, respectively, the two subproblems

$$\mathit{Find}\ \Phi\big|_\Gamma : \Gamma \to \mathbb{R}\ s.t.: \qquad \Big(\alpha(\mathbf{x})I - K\Big)\Phi\big|_\Gamma = u - V\partial_\mathbf{n}u\big|_\Gamma, \tag{8.94}$$

$$\mathit{Find}\ t_P\big|_\Gamma : \Gamma \to \mathbb{R}\ s.t.: \qquad Vt_P\big|_\Gamma = u - \Big(\alpha(\mathbf{x})I - K\Big)u\big|_\Gamma. \tag{8.95}$$

The former should recover the Dirichlet values of the reference solution on the dielectric interface Γ and the latter the Neumann values, i.e t_P should, up to discretization errors, coincide with the values of $\partial_\mathbf{n}u$ at Γ. In contrast to the FEM part the BEM part involves possibly singular integrals and therefore its convergence properties are not obvious. To solve the problems we discretize the solution with standard Lagrange finite elements of degree q as provided by the `dealii::FE_Q` class. As numerical surface quadrature we use the tensor Gauss rules provided by the class `dealii::QGauss` with p^2 quadrature points. For the curvilinear approximation of polynomial order m of the dielectric interface Γ we use the `dealii::MappingQ` class. Figs. 8.13(a) - 8.13(f) show that the most important parameters are the degree m of the boundary approximation and the number p of quadrature points per space dimension. Higher order finite elements do not pay off.

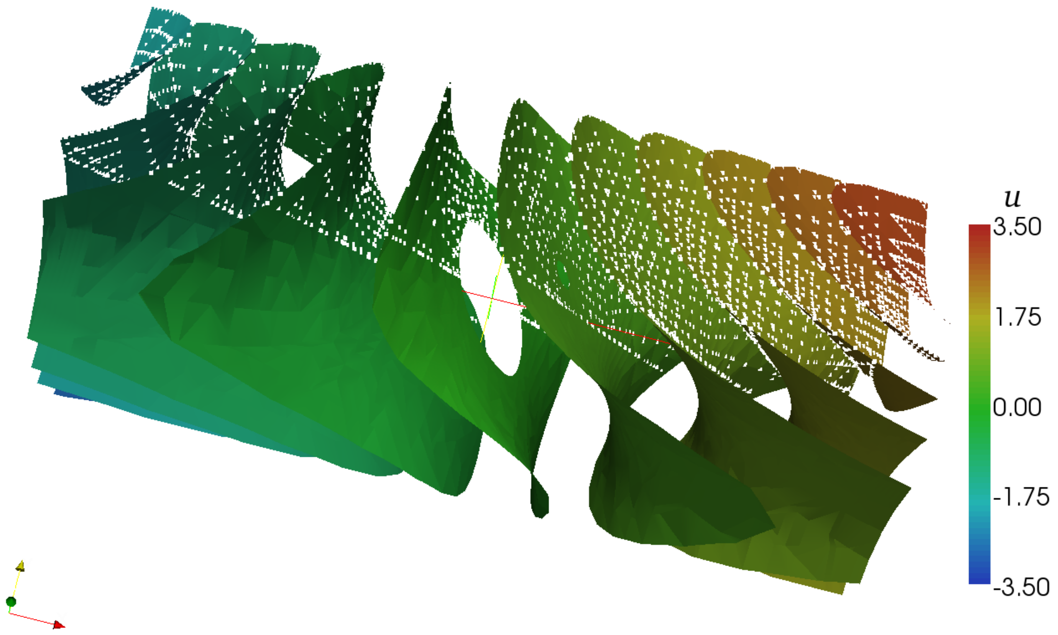


Figure 8.10: Numerical solution of the dipole+harmonic contribution testcase (points), Eq. (8.90) vs. exact solution u (isosurfaces). The solution is approximated by linear elements. The boundary is approximated by quartic polynomials and matrices are assembled with 8-point Gauss quadrature rules.

FEM-BEM

For the full problem in Eq. (8.91) the boundary integral is only a sub-dimensional problem and the issue is whether its convergence properties influence the convergence behavior of the finite element method. The convergence for linear elements and different boundary approximations and quadrature rules is shown in Fig. 8.11. The L^2-error shows the expected quadratic behavior independent of m and p. This can be expected from the error behavior for linear elements, Figs. 8.13(b) - 8.13(d), of the bare boundary integral problem. Depending on the particular test case the convergence for quadratic elements, Fig. 8.12, strongly depends on the quality of the boundary approximation and the of the quadrature rule. More importantly, the convergence is worse than for linear elements. A detailed analysis of the convergence behavior with respect to the various method parameters is not the purpose of this chapter. To investigate the physical problem of the dependence of the DRS DC current on the protein conformation it suffices that the numerical solution converges in a finite time so that the dependence of the solution on the physical parameters can be studied. Nevertheless, with an improved understanding of the physics the questions a simulation has to answer will be more demanding. Hence, at some point we will have to come back to a detailed investigation of the convergence properties of the FEM-BEM problem for higher order finite elements.

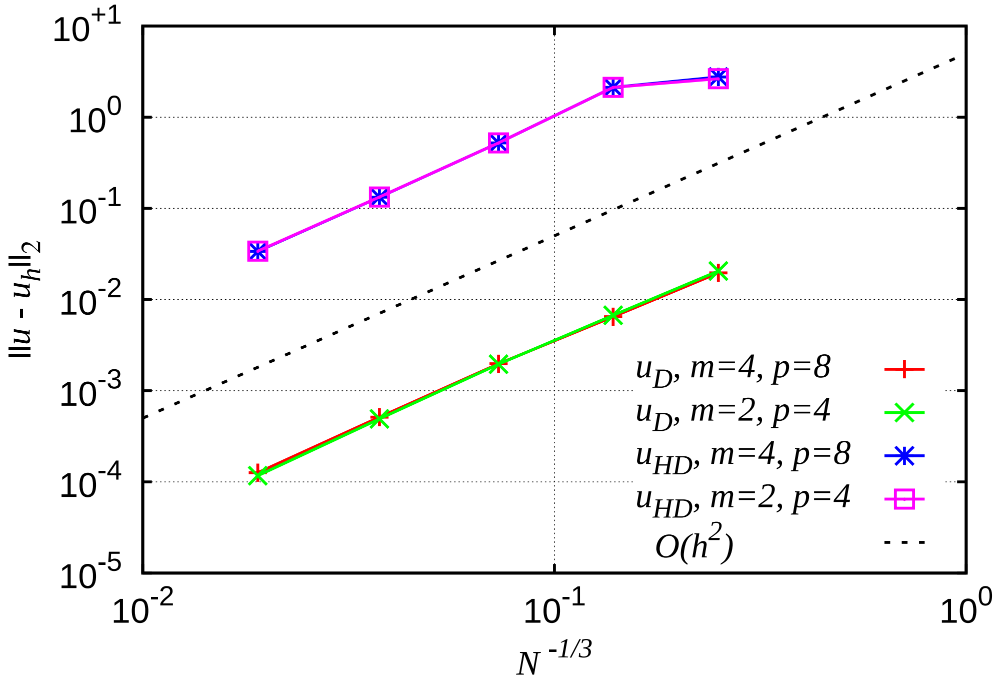


Figure 8.11: Convergence of the coupled FEM-BEM test problem.

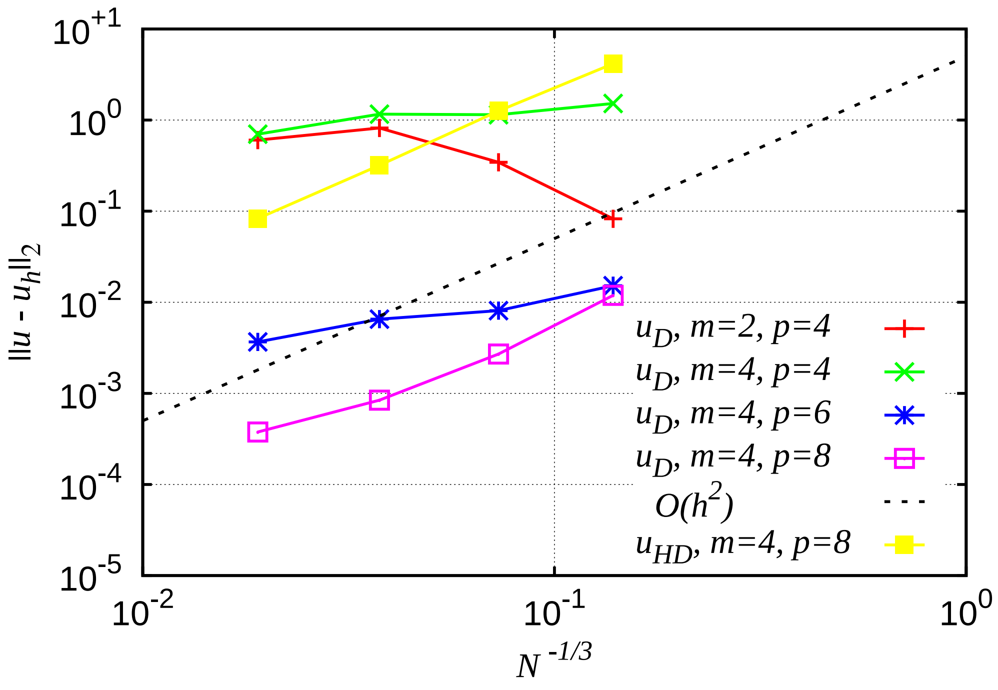
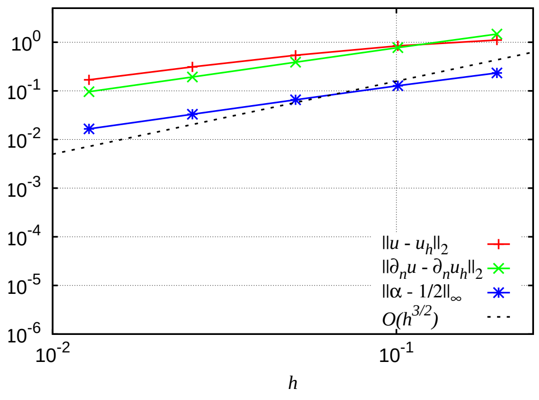
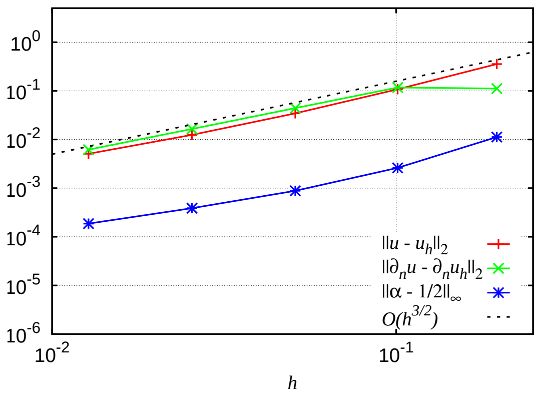


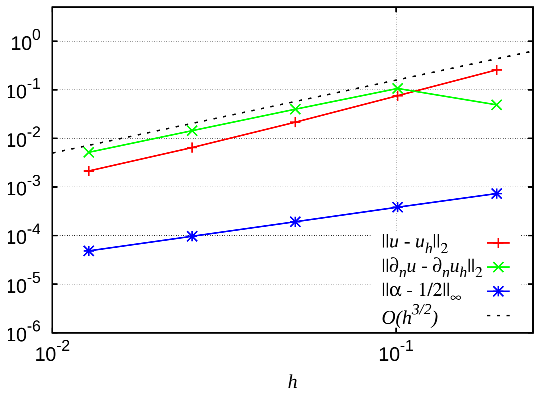
Figure 8.12: Convergence of the coupled FEM-BEM dipole test problem for quadratic elements, quadrature rules of order 4, 6, 8 and boundary approximations of order 2 and 4.



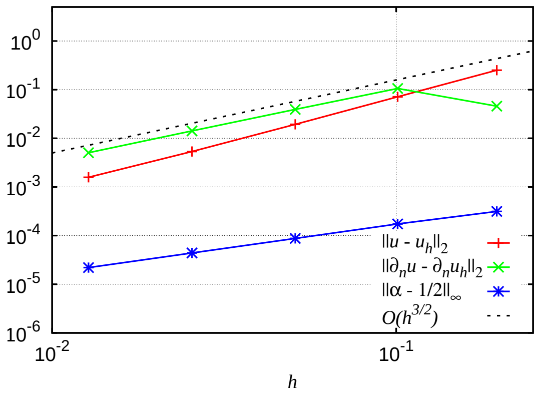
(a) $q = 1$, $m = 1$, $p = 4$



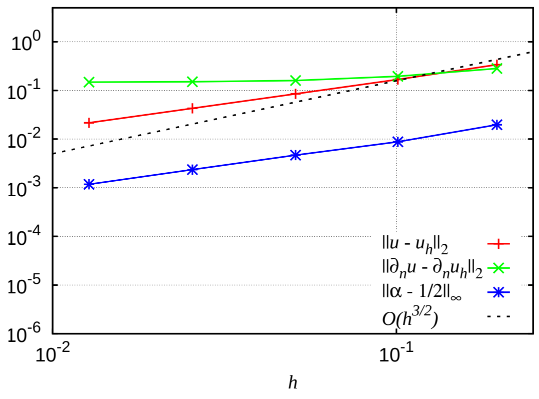
(b) $q = 1$, $m = 2$, $p = 4$



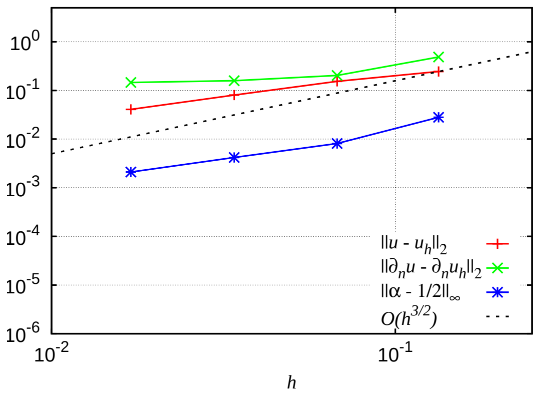
(c) $q = 1$, $m = 4$, $p = 4$



(d) $q = 1$, $m = 4$, $p = 12$



(e) $q = 2$, $m = 4$, $p = 8$



(f) $q = 3$, $m = 3$, $p = 8$

Figure 8.13: Convergence behavior for different finite element degrees q, order of surface quadrature rules p and polynomial orders m of the curvilinear boundary approximation.

Conclusion and Outlook

Part I - Tools

In the first part of this thesis we have introduced *SciPAL*, a new *C*++-class library for rapid integration of GPGPU computing into existing applications. Currently, it mainly comprises classes for linear algebra. *SciPAL* is going to be published under an open-source license. The extension to CUDA-based finite elements and boundary elements is work in progress.

Benefits: The gain from the *SciPAL* library is a domain-specific embedded language for matrix computations for stating the linear algebraic part of a problem in its intuitive mathematical notation. This is realized by providing properly overloaded operators with zero-overhead and type-safe programming. As we have shown, the operator overloading does not incur any run-time overhead and is realized by expression-templates which allow for very compact, self-documenting code. To this end, matrix and vector classes consistent with *deal*.II [20, 21] have been introduced and run-time errors are minimized by extensive compile-time checking due to the template metaprogramming approach. The effort of porting to a new BLAS-flavor is minimized to writing one header file with some inlined wrapper functions.

Software Engineering: *SciPAL* tries to fulfill many of the criteria for “good” software engineering such as reusability, readability and portability, to name just a few. In particular, the library is subject to extensive unit testing and continuous integration to ensure that modifications do not break existing programs or interactions between modifications have unwanted side effects.

Easy integration of GPU-based computations: Following a current trend, *SciPAL* can delegate data-parallel computations to GPUs. In particular, CUBLAS is incorporated in a *deal*.II-compatible way. A tremendous simplification for using CUDA is provided by the fact, that the error-prone transfer between the different memory sections of the heterogeneous system formed by CPU and GPU is hidden in assignment operators. In particular, it is left to the compiler to pick the correct direction depending on the matrix or vector types. The assignment opertors are the key feature for a seamless integration of CUDA-based computations into modern, *C*++-based scientific libraries like *deal*.II.

Application-oriented examples: Last but not least, there is a continuously growing list of example programs demonstrating the features of *SciPAL* and explaining in detail why things have been implemented as they are.

Part II - Applications

Phase Holograms

One of the most successful applications based on *SciPAL* is the computation of phase-only holograms for photo-stimulation of individual sites in networks of living neurons. Mathematically, computing a phase-only holograms to create an optical stimulation pattern which selects predetermined neurons constitutes is equivalent to wavefront reconstruction. Useful approximations of a phase mask for a given optical stimulation pattern can only be achieved by iterative algorithms like the widely used Method of Alternating Projections. The results show that at most 5 iterations suffice to compute a phase mask within less than 10ms, matching the dynamics of neural activity. The generic programming approach led to a flexible framework for phase retrieval problems which offers easy switching between the three parallelization techniques tested: CUDA, OpenMP and pthreads. Only the presented CUDA-based implementation is currently capable of the necessary frame rates for stimulating networks of optogenetically altered neurons on their intrinsic timescale of several ms.

The high modularity of the simulation framework makes it easy to implement other reconstruction algorithms and to apply to other problems of wavefront reconstruction totally unrelated to the presented test case from the field of optogenetics.

Preconditioning for Indoor Airflow Simulations

We considered CUDA-based parallel preconditioning for non-normal matrices as they arise from fluid mechanics problems in engineering indoor airflow. A low-turbulent indoor airflow case study served as testbed.

Our results show that a CUDA-based sparse approximate inverse or Block-Jacobi preconditioner can outperform a CPU-based ILU preconditioner by almost an order of magnitude despite a higher iteration count. Especially the Oseen problem which is the numerically most expensive one can profit from CUDA-based the parallelization. Even in its accelerated version the run-time to solve it is longer than the accumulated run-times for solving the turbulence model, temperature equation and the equation for the air age. Therefore, the algebraic part of turbulent indoor air flow simulations can be accelerated by almost an order of magnitude by employing a hybrid strategy where one host solves the Oseen problem on the GPU and while a second host thread solves the turbulence model, Fourier law and air age on the CPU.

Besides the nice gain in performance this project demonstrates that finding a good preconditioner is not only related to the physical and mathematical properties of a particular problem but also to the details of its implementation. Without fine-grained parallelization sparse approximate inverse or Block-Jacobi preconditioners would probably never outperform ILU.

The investigation and discussion in this thesis of parallel preconditioning strategies for turbulent indoor airflow is certainly not exhaustive. The issues of the influence of combining the various stabilization techniques with different finite element types and orders has not even been mentioned. For a systematic investigation it is certainly worthwhile to apply the presented preconditioning techniques to the long list of example programs provided by *deal*.II. They cover the different fundamental types of partial differential equations (elliptic, advective, hyperbolic) in their scalar or vector-value variants.

Quantum Transport

During the past three decades, low-temperature quantum transport phenomena in mesoscopic electron devices have been intensively studied. This thesis provides a well-tested, *deal*.II-based simulation tool for the accurate prediction of the wave functions of scattering states in semi-infinite systems with complex boundaries. The main contribution is an adaption of a recently developed type of transparent boundary conditions for finite element methods for acoustic scattering to quantum transport computations in semi-open electron systems. This allows a numerical study of caustic phenomena and branched electron flow, only recently discovered by spatially resolved recordings of electron densities in two-dimensional electron gases.

The presented numerical results for the magneto-conductance of a particular device show that subtle details in the shape of the geometry may have drastic effects on the transport properties of a device which definitely need a closer look. There are certainly other experiments worthwhile an in-depth investigation of their transport properties. To do that, all one has to do is to change the geometry of the computational domain.

The next step is to employ the programming techniques presented in the first part to enhance the performance of the simulation framework. On the fully discretized level the Schrödinger equation amounts to solving a system of equations with an indefinite matrix which is the most time consuming part in the magneto-transport computations. In practice, sparse direct solvers are the only ones which satisfactorily work for indefinite problems up to now. Yet their complexity and especially the memory consumption do not scale favorably with problem size The project on parallel preconditioning has shown that because of parallelization by CUDA algorithms of inferior complexity might outperform algorithms previously considered as superior. In this case the direct methods would have to be replaced by Krylov solvers with a preconditioner capable of the indefinite case, for instance sparse approximate inverse techniques of which we know that their application fits the CUDA programming paradigm well.

Quantum Time Propagation

Most systems in which wave-packet dynamics is studied are defined on unbounded domains. The methods developed in Chapter 6 for an exact treatment of transparent boundaries can be used to improve the wave-packet techniques used in quantum chemistry. A common application of wave packet dynamics is the computation of chemical reaction rates.

We have shown that a propagation scheme on truncated Faber series is rather simple to implement. Faber series are chosen because of their exponential convergence with respect to the point of truncation and their near-best approximation property. Provided a matrix representation of the Hamiltonian is given, the only non-trivial operation is a matrix-vector product. Depending on the basis set in which the Hamiltonian is approximated either dense or sparse matrix-vector products are needed. Therefore, the operator-based interface for linear algebra operations provided by *SciPAL* should make implementing a time-propagation scheme based on Faber polynomials particularly easy.

The polynomial preconditioning method presented in this thesis could improve such computations by adding the non-convex inclusion sets known from the Arnoldi-Bratwurst-Faber method by Liesen to the Faber-based time evolution methods already known in quantum chemistry and combine them with a more accurate treatment of transparent boundaries. The advantages of these particular inclusion sets is that the exterior mapping function is known in a closed form, gives rise to a memory-efficient three-term recursion for the Faber polynomials and because of their convexity reduce the number of terms needed in a truncated Faber series to achieve a given error tolerance.

Dielectric Relaxation Spectroscopy

Recent dielectric spectroscopy studies of ubiquitin in solution have revealed the influence of conformational sampling on the direct current contribution to the dielectric loss spectrum. From these experimental findings it is deduced that, depending on its conformation, a ubiquitin molecule may bind a varying number of ions in its hydration shell. This is supposed to influence the density of mobile ions responsible for the direct current component. The simplest model for the interaction between conformational dynamics and the direct current only addresses the dependence of the distribution of bound and mobile ions on the different protein states. Yet it is able to explain the salient features of the sub-β peak although boundary conditions, excluded volume effects and the chemical traits of the ion species are ignored completely.

This thesis describes an improved model based on the Poisson-Nernst-Planck equations for electro-diffusion to achieve a more quantitative description of dielectric relaxation spectroscopy of proteins in solution. First results on a simplified two-dimensional model show similar alterations of the direct current.

The simulation of the three-dimensional case is severely hampered by the solution of the fully discretized equations. The non-negligible ellipticity of the problem makes multigrid techniques an attractive option to speedup the calculations. The solution of the FEM-BEM test case is already based on multigrid methods and could not have been solved without it. Their theoretical complexity is optimal and all components (smoother, interpolation and prolongation) are suitable for CUDA's parallelization paradigm. Parallelization can be further simplified by switching to discontinuous Galerkin methods which would solve the issues of conservation laws for the ion species. The partial differential equations describing the physical model are sorted out by now. This is the main contribution of this thesis to this physical problem. Obviously, during the modeling process programming issues are usually of less importance. However, in order to go to the next stage of optimizing the implementation we definitely need the tools from part one in order to quickly switch between different solution strategies for discretized equations and accelerating the discretization process itself. Without an optimized solver for the stationary case the extension to time-dependence does not even need to be considered. At the same time, the discretization of this problem offers a great testbed for employing CUDA for more than just linear algebra. It simulates physical questions of current interest, yet it is simpler than the indoor airflow problem and issues of parallelization can be studied more easily. Thus, while optimizing the code one gains new physical insights almost for free.

From a physical point of view the simulation of the time-dependent version of dielectric relaxation problem would be very much appreciated. A priori it is not obvious that the life times of the internal states of a protein are sufficiently long for the direct current to reach stationary value before the protein switches into another conformation. The next level of detail in the model would be to take into account the true shape of a protein which eventually can be enhanced by extracting the internal states from a molecular dynamics simulation.

Acknowledgments

This thesis would not have been possible without the the help of several people. First of all, I would like to express my gratitude to my supervisors Gert Lube (GL) and Reiner Kree (RK) for giving me the opportunity to finish this long-term project. Their assistance and motivation can hardly be underestimated. Precisely speaking, I have to thank GL twice. When finishing my math diploma thesis the acknowledgments fell victim to the deadline. Therefore: Thank you for my math diploma.

I thank Dorothea Bahns, Winfried Kurth, Russell Luke and Andreas Tilgner for taking the time to be members of my thesis committee.

The rapid calculation of phase holograms grew on many fruitful discussions with Russell Luke, Johannes Hagemann, Hecke Schrobsdorff and Fred Wolf. Especially Johannes' interest in CUDA, software design and all sorts of physics were a constant source of inspiration. I should also mention Oliver Bendix, Ragnar Fleischmann, Thorsten Hohage, Jakob Metzger and Lothar Nannen. Without their assistance and encouragement I would not have started the magneto-conductance project. Once I had got it going the permission to use the clusters of the Max-Planck Institute for Dynamics and Self-Organisation proved to be an invaluable help. I also gratefully acknowledge the long-term cooperation with Ralf Gritzki, Alf Perschk and Markus Rösler from the TU Dresden. Their interest in faster and more accurate solutions for heating networks, indoor airflow and radiative heat transfer allowed me to play with OpenGL and Qt and to add an engineer's perspective to my repertoire of problem solving strategies.

I would like to thank my family for the support they provided me through my entire life. In particular, I must acknowledge my father's thorough editing assistance and that he never lost the faith in me finishing a PhD thesis. My most beloved Kasia encouraged me to go on although years passed and the finishing of this thesis sometimes seemed to be light years away.

Often I am characterized as resistant to well-intentioned advice. I think this is only partially true. I do listen other people. Maybe just on timescales different from those they expect.

...I have only one eye.

I have a right to be blind sometimes,

and, raising his telescope to his blind eye:

I really do not see the signal.

(Vice-Admiral Horatio Nelson off Copenhagen, 1801)

Bibliography

[1] *BOOST C++ Libraries*. `http://www.boost.org`.

[2] `BOOST uBLAS: a Linear Algebra Library`, *Technical Reference*. http://www.boost.org/libs/numeric/ublas/doc/index.htm.

[3] `CPPLapack Documentation`, *Technical Reference*.

[4] *Lapack++ Documentation, Technical Reference*. http://lapackpp.sourceforge.net/.

[5] *Spirit Parser libraries*. `http://boost-spirit.com`.

[6] Three models for the description of language. *IEEE Transactions on Information Theory*, 2(3):113–124, Sept. 1956.

[7] http://math.nist.gov/tnt/index.html, 2004.

[8] Fermi compute architecture whitepaper. Technical report, Nvidia Corporation, 2009.

[9] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley, 2004.

[10] M. Abramowitz and I. Stegun. *Handbook of Mathematical Functions: With Formulas, Graphs, and Mathematical Tables*. Applied mathematics series. Dover Publications, 1965.

[11] K. E. Aidala, R. E. Parrott, T. Kramer, E. J. Heller, R. M. Westervelt, M. P. Hanson, and A. C. Gossard. Imaging magnetic focusing of coherent electron waves. *Nature Physics*, 3:464–468, July 2007.

[12] M. Andrecut. Parallel gpu implementation of iterative pca algorithms. *xxx.lanl.gov/abs/0811.1081 preprint including source code*, 2008.

[13] M. Andrecut. Parallel gpu implementation of iterative pca algorithms. *Journal of Computational Biology*, 16(11):1593–1599, 2009. PMID: 19772385.

[14] D. N. Arnold, , F. Brezzi, and B. Cockburn. Discontinuous galerkin methods for elliptic problems. *IN DISCONTINUOUS GALERKIN METHODS (NEWPORT, RI, 1999), LECTURE NOTES COMPUTATIONAL SCIENCE ENGINEERING*, page 89, 2000.

[15] D. N. Arnold. An interior penalty finite element method with discontinuous elements. *SIAM J. Numer. Anal.*, 19:742–760, 1982.

[16] D. N. Arnold, F. Brezzi, B. Cockburn, and L. D. Marini. Unified analysis of discontinuous galerkin methods for elliptic problems. *SIAM J. Numer. Anal.*, 39(5):1749–1779, May 2001.

[17] A. Aronov. Magnetic flux effects in disordered conductors. *Rev. Mod. Phys.*, 59(3):755–779, 1987.

[18] D. Ban, M. Funk, R. Gulich, D. Egger, T. M. Sabo, K. F. A. Walter, R. B. Fenwick, K. Giller, F. Pichierri, B. L. deGroot, O. F. Lange, H. Grubmüller, X. Salvatella, M. Wolf, A. Loidl, R. Kree, S. Becker, N.-A. Lakomek, D. Lee, P. Lunkenheimer, and C. Griesinger. Kinetics of conformational sampling in ubiquitin. *Angewandte Chemie International Edition*, 50(48):11437–11440, 2011.

[19] D. Ban, M. Funk, R. Gulich, D. Egger, T. M. Sabo, K. F. A. Walter, R. B. Fenwick, K. Giller, F. Pichierri, B. L. deGroot, O. F. Lange, H. Grubmüller, X. Salvatella, M. Wolf, A. Loidl, R. Kree, S. Becker, N.-A. Lakomek, D. Lee, P. Lunkenheimer, and C. Griesinger. Kinetics of conformational sampling in ubiquitin, supplementary material. *Angewandte Chemie International Edition*, 50(48), 2011.

[20] W. Bangerth, R. Hartmann, and G. Kanschat. `deal.II` *Differential Equations Analysis Library, Technical Reference*.

[21] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – a general purpose object oriented finite element library. *ACM Trans. Math. Softw.*, 33(4):24/1–24/27, 2007.

[22] H. U. Baranger and A. D. Stone. Electrical linear-response theory in an arbitrary magnetic field: A new fermi-surface formation. *Phys. Rev. B*, 40(12):8169–8193, Oct 1989.

[23] H. H. Bauschke, P. L. Combettes, and D. R. Luke. Phase retrieval, error reduction algorithm and Fienup variants: a view from convex feasibility. *J. Opt. Soc. Amer. A.*, 19(7):1334–45, 2002.

[24] M. Z. Bazant, K. Thornton, and A. Ajdari. Diffuse-charge dynamics in electrochemical systems. *Phys. Rev. E*, 70:021506, Aug 2004.

[25] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, 2008.

[26] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 18:1–18:11, New York, NY, USA, 2009. ACM.

[27] J. Bielak and R. MacCamy. An exterior interface problem in two-dimensional elastodynamics. *Quarterly of Applied Mathematics*, 41(1):143–159, 1983.

[28] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, July 2003.

[29] A. G. Borisov and S. V. Shabanov. Wave packet propagation by the Faber polynomial approximation in electrodynamics of passive media. *Journal of Computational Physics*, 216(1):391 – 402, 2006.

[30] A. N. Brooks and T. J. Hughes. Streamline upwind/petrov-galerkin formulations for convection dominated flows with particular emphasis on the incompressible navier-stokes equations. *Computer Methods in Applied Mechanics and Engineering*, 32(1-3):199 – 259, 1982.

[31] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, Aug. 2004.

[32] M. Büttiker. Chemical potential oscillations near a barrier in the presence of transport. *Phys. Rev. B*, 40:3409–3412, Aug 1989.

[33] T. F. Chan, E. Gallopoulos, V. Simoncini, T. Szeto, and C. H. Tong. A quasi-minimal residual variant of the bi-cgstab algorithm for nonsymmetric systems. *SIAM J. Sci. Comput.*, 15:338–347, 1994.

[34] D. Colbert and W. Miller. A novel discrete variable representation for quantum mechanical reactive scattering via the s-matrix kohn method. *The Journal of chemical physics*, 96(3):1982, 1992.

[35] M. Costabel. Symmetric methods for the coupling of finite elements and boundary elements (invited contribution). In *Boundary elements IX, Vol. 1 (Stuttgart, 1987)*, pages 411–420. Comput. Mech., Southampton, 1987.

[36] A. Damjanović. Excitons in a photosynthetic light-harvesting system: A combined molecular dynamics, quantum chemistry, and polaron model study. *Physical Review E*, 65(3):031919, 2002.

[37] S. Datta. *Electronic Transport in Mesoscopic Systems (Cambridge Studies in Semiconductor Physics and Microelectronic Engineering)*. Cambridge University Press, 1997.

[38] K. Deisseroth, G. Feng, A. K. Majewska, G. Miesenböck, A. Ting, and M. J. Schnitzer. Next-generation optical technologies for illuminating genetically targeted brain circuits. *Journal of Neuroscience*, 26(41), October 2006.

[39] J. Demmel. *Applied Numerical Linear Algebra*. Miscellaneous Bks. Society for Industrial and Applied Mathematics, 1997.

[40] G. H. Dunteman. *Principal Component Analysis*. Sage Publications, 1989.

[41] L. Eldén. *Matrix methods in data mining and pattern recognition*. Fundamentals of algorithms. Society for Industrial and Applied Mathematics, 2007.

[42] S. W. Ellacott. Computation of Faber series with application to numerical polynomial approximation in the complex plane. *MATHEMATICS OF COMPUTATION*, 40(162):575–587, 1983.

[43] G. Faber. Über polynomische Entwickelungen. *Mathematische Annalen*, 57:389–408, 1903.

[44] M. Feit, J. F. Jr., and A. Steiger. Solution of the schrödinger equation by a spectral method. *Journal of Computational Physics*, 47(3):412 – 433, 1982.

[45] N. S. Foundation and D. of Energy. BLAS. http://www.netlib.org/blas/, 2010.

[46] A. Fowler. Conductance in restricted-dimensionality accumulation layers. *Phys. Rev. Lett.*, 48(3):196–199, 1982.

[47] M. Frigo and S. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[48] G. N. Gatica, N. Heuer, and F.-J. Sayas. A direct coupling of local discontinuous Galerkin and boundary element methods. *Math. Comp.*, 79(271):1369–1394, 2010.

[49] G. N. Gatica and F.-J. Sayas. An a priori error analysis for the coupling of local discontinuous galerkin and boundary element methods. *Math. Comp.*, 75(256):1675–1696 (electronic), 2006.

[50] R. Gerchberg and W. Saxton. A practical algorithm for the determination of phase from image and diffraction plane pictures. *Optik*, 35(2):237–246, 1972.

[51] V. Girault and P. Raviart. *Finite Element Approximation of the Navier-Stokes Equations*. Springer Verlag Germany, 1986.

[52] L. Golan, I. Reutsky, N. Farah, and S. Shoham. Design and characteristics of holographic neural photo-stimulation systems. *Journal of Neural Engineering*, 6:066004, 2009.

[53] G. Golub and C. Van Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, 1996.

[54] J. W. Goodman. *Introduction to Fourier Optics*. McGraw-Hill, 2nd edition, 1996.

[55] P. Gottschling, D. S. Wise, and A. Joshi. Generic support of algorithmic and structural recursion for scientific computing. *International Journal of Parallel, Emergent and Distributed Systems*, 24(6):479 – 503, 2009.

[56] J. Härdtlein. *Moderne Expression Templates Programmierung – Weiterentwickelte Techniken und deren Einsatz zur Lösung partieller Differentialgleichungen*. PhD thesis, Technischen Fakultät der Universität Erlangen-Nürnberg, 2007.

[57] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition*. Springer Series in Statistics. Springer, 2009.

[58] H. Hauptman. Direct methods and anomalous dispersion – Nobel lecture, 9 December 1985. *Chemica Scripta*, 26(2):277–286, 1986.

[59] V. Heuveline, D. Lukarski, and J.-P. Weiss. Fine-grained Parallel ILU Preconditioners with Fill-ins for Multi-core CPUs and GPUs. In *International Conference On Preconditioning Techniques For Scientific And Industrial Applications 2011*, accepted.

[60] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2002.

[61] T. Hohage and L. Nannen. Hardy space infinite elements for scattering and resonance problems. *SIAM Journal on Numerical Analysis*, 47(2):972–996, 2009.

[62] W. Huisinga, L. Pesce, R. Kosloff, and P. Saalfrank. Faber and Newton polynomial integrators for open-system density matrix propagation. *Journal of Chemical Physics*, 110:5538–5547, Mar. 1999.

[63] K. Iglberger, G. Hager, J. Treibig, and U. Rüde. Expression templates revisited: A performance analysis of current methodologies. *SIAM Journal on Scientific Computing*, 34(2):C42–C69, 2012.

[64] K. Iglberger, G. Hager, J. Treibig, and U. Rüde. High performance smart expression template math libraries. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 367 –373, july 2012.

[65] K. Iglberger and U. Rüde. The math library of the pe physics engine - combining smart expression templates and blas efficiency. Technical report, Institut für Informatik, Universität Erlangen-Nürnberg, 2009.

[66] Y. Imry and R. Landauer. Conductance viewed as transmission. *Rev. Mod. Phys.*, 71:S306–S312, Mar 1999.

[67] C. Johnson and J. Nédélec. On the coupling of boundary integral and finite element methods. *Math. Comp*, 35(152):1063–1079, 1980.

[68] C. Johnson and J. Saranen. Streamline diffusion methods for the incompressible euler and navier-stokes equations. *Math. Comput.*, 47:1–18, July 1986.

[69] G. Kanschat. Multilevel methods for discontinuous galerkin fem on locally refined meshes. *Computers and Structures*, 82(28):2437 – 2445, 2004. Preconditioning methods: algorithms, applications and software environments.

[70] K. Karhunen. Zur Spektraltheorie stochastischer Prozesse. *Ann. Acad. Sci. Fennicae*, 37, 1946.

[71] K. Klitzing. New method for high-accuracy determination of the fine-structure constant based on quantized hall resistance. *Phys. Rev. Lett.*, 45(6):494–497, 1980.

[72] A. Knocks and H. Weingärtner. The dielectric spectrum of ubiquitin in aqueous solution. *The Journal of Physical Chemistry B*, 105(17):3635–3638, 2001.

[73] T. Knopp. *Finite-element simulation of buoyancy-driven turbulent flows*. PhD thesis, Georg-August-Universität Göttingen, Mathematische Fakultät, 2003.

[74] T. Knopp, G. Lube, R. Gritzki, and M. Rösler. A near-wall strategy for buoyancy-affected turbulent flows using stabilized fem with applications to indoor air flow simulation. *Computer Methods in Applied Mechanics and Engineering*, 194(36-38):3797 – 3816, 2005.

[75] T. Koch and J. Liesen. The conformal 'bratwurst' maps and associated Faber polynomials. *Numerische Mathematik*, 86(1):173–191, 2000.

[76] R. Kramer. *Chemometric techniques for quantitative analysis*. CRC, 1998.

[77] S. Kramer, C. Pfaffenbach, and G. Lube. CUDA-based parallel preconditioning for rans simulations of indoor airflow. In A. C. et al., editor, *Numerical Mathematics and Advanced Applications, Proceedings of the ENUMATH 2011*, Berlin, 2012. Springer. to appear.

[78] T. Kramer, E. J. Heller, and R. E. Parrott. An efficient and accurate method to obtain the energy-dependent green function for general potentials. *Journal of Physics: Conference Series*, 99(1):012010, 2008.

[79] R. Kreß. *Linear Integral Equations*. Number Bd. 82 in Applied Mathematical Sciences. Springer, 1999.

[80] R. Landauer. Spatial variation of currents and fields due to localized scatterers in metallic conduction. *IBM Journal of Research and Development*, 1(3):223 –231, july 1957.

[81] S. Larsson and V. Thomee. *Partielle Differentialgleichungen und numerische Methoden*. Springer-Lehrbuch Masterclass. Springer, 2005.

[82] X. S. Li. An overview of superlu: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31(3):302–325, Sept. 2005.

[83] J. Liesen. *Construction and analysis of polynomial iterative methods for non-hermitian systems of linear equations*. PhD thesis, Universität Bielefeld, 1998.

[84] J. Liesen. Faber polynomials corresponding to rational exterior mapping functions. *Constructive Approximation*, 17(2):267–274, 2001.

[85] M. M. Loeve. *Probability Theory*. Number 46 in Graduate Texts in Mathematics. Springer New York, 1978.

[86] G. Lube, T. Knopp, R. Gritzki, M. Rosler, and J. Seifert. Application of domain decomposition methods to indoor air flow simulation. *Int. J. Comput. Math.*, 85(10):1551–1562, Oct. 2008.

[87] G. Lube, T. Knopp, G. Rapin, R. Gritzki, and M. Rösler. Stabilized finite element methods to predict ventilation efficiency and thermal comfort in buildings. *International Journal for Numerical Methods in Fluids*, 57(9):1269–1290, 2008.

[88] D. Luke. Relaxed averaged alternating reflections for diffraction imaging. *Inverse Problems*, 21:37, 2005.

[89] D. Luke, J. Burke, and R. Lyon. Optical wavefront reconstruction: theory and numerical methods. *SIAM review*, 44(2):169–224, 2002.

[90] D. R. Luke. Local linear convergence of approximate projections onto regularized sets. *Nonlinear Anal.*, 75:1531–1546, 2012.

[91] F. Magoules. *Mesh Partitioning Techniques and Domain Decomposition Methods*. Saxe-Coburg Publications, 2008.

[92] C. M. Marcus, A. J. Rimberg, R. M. Westervelt, P. F. Hopkins, and A. C. Gossard. Conductance fluctuations and chaotic scattering in ballistic microstructures. *Phys. Rev. Lett.*, 69:506–509, Jul 1992.

[93] N. Masuda, T. Ito, T. Tanaka, A. Shiraki, and T. Sugie. Computer generated holography using a graphics processing unit. *Opt. Express*, 14(2):603–608, Jan 2006.

[94] V. May and O. Kühn. *Charge and Energy Transfer Dynamics in Molecular Systems*. John Wiley & Sons, 2011.

[95] J. J. Metzger. *Branched Flow and Caustics in Two-Dimensional Random Potentials and Magnetic Fields*. PhD thesis, Max-Planck Institut f. Dynamik und Selbstorganisation, 2010.

[96] S. Meyers. *Effective C++: 55 Specific Ways to Improve your Programs and Designs*. Addison-Wesley Professional Computing Series. Addison-Wesley, 2005.

[97] S. M. Miller and T. C. Jr. Calculation of reaction probabilities using wavepackets. *Chemical Physics Letters*, 267(5-6):417 – 421, 1997.

[98] B. Mohammadi and O. Pironneau. *Analysis of the K-Epsilon Turbulence Model*. 1994.

[99] C. Moler. The worlds largest matrix computation. *Matlab News and Notes*, October 2002.

[100] L. Nannen. *Hardy-Raum Methoden zur numerischen Lösung von Streu- und Resonanzproblemen auf unbeschränkten Gebieten*. PhD thesis, Universität Göttingen, 2008.

[101] L. Nannen and A. Schädle. Hardy space infinite elements for helmholtz-type problems with unbounded inhomogeneities. *Wave Motion*, In Press, Corrected Proof:–, 2010.

[102] NVIDIA Corporation, 2701 San Tomas Expressway Santa Clara, CA 95050. *CUDA CUBLAS Library*, v3.0 edition.

[103] G. Of, G. Rodin, O. Steinbach, and M. Taus. Coupling of discontinuous galerkin finite element and boundary element methods. *SIAM Journal on Scientific Computing*, 34(3):A1659–A1677, 2012.

[104] U. of Wisconsin-Madison. Solar Energy Laboratory and K. S. A. *TRNSYS, a Transient System Simulation Program*. Solar Energy Laboratory, University of Wisconsin–Madison, 1979.

[105] F.-C. Otto. *A non-overlapping domain decomposition method for elliptic problems*. PhD thesis, Georg-August-Universität Göttingen, Mathematische Fakultät, 1999.

[106] P. S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, 1997.

[107] K. Pearson. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine*, 2(6):559–572, 1901.

[108] A. Priesnitz. Untersuchung iterativer lösungsverfahren am beispiel diskretisierter konvektions-diffusions-reaktions-gleichungen. Master's thesis, Institut für Angewandte und Numerische Mathematik, Universität Göttingen, 1996.

[109] A. J. Rasmussen and S. C. Smith. A lanczos-powered implementation of the Faber polynomial quantum time propagator for reaction probabilities. *Chemical Physics Letters*, 387(4-6):277 – 282, 2004.

[110] S. Rjasanow and O. Steinbach. *The Fast Solution of Boundary Integral Equations (Mathematical and Analytical Techniques with Applications to Engineering)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[111] S. Rotter, J.-Z. Tang, L. Wirtz, J. Trost, and J. Burgdörfer. Modular recursive green's function method for ballistic quantum transport. *Phys. Rev. B*, 62(3):1950–1960, Jul 2000.

[112] Y. Saad. *Iterative methods for sparse linear systems*. Second edition, 2003.

[113] Y. Saad and M. H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7:856–869, July 1986.

[114] F. Sayas. The validity of Johnson–Nédélec's BEM–FEM coupling on polygonal interfaces. *SIAM Journal on Numerical Analysis*, 47(5):3451–3463, 2009.

[115] W. Schmickler and E. Santos. *Interfacial Electrochemistry*. Springer, 2010.

[116] H. Schrobsdorff. Private communication. 2012.

[117] D. Y. Sharvin and Y. V. Sharvin. Magnetic-flux quantization in a cylindrical film of a normal metal. *Soviet Journal of Experimental and Theoretical Physics Letters*, 34:272–275, Sept. 1981.

[118] S. E. J. Shaw. *Propagation in smooth random potentials*. PhD thesis, Harvard University, 2002.

[119] T. Shimobaba, T. Ito, N. Masuda, Y. Ichihashi, and N. Takada. Fast calculation of computer-generated-hologram on AMD HD5000 series GPU and OpenCL. *Opt. Express*, 18(10):9955–9960, May 2010.

[120] O. Stern. Zur theorie der elektrolytischen doppelschicht. *Z. Elektrochem. Angew. Phys. Chem.*, 30:508, 1924.

[121] S. Strobl. Gpu-based rigid body dynamics. Master's thesis, Institut für Informatik, Universität Erlangen-Nürnberg, 2009.

[122] B. Stroustrup. *Die C++-Programmiersprache*. Professionelle Programmierung. Addison-Wesley, 2000.

[123] B. Stroustrup. *Die C++-Programmiersprache*. Programmer's Choice. Addison Wesley Verlag, 2010.

[124] J. W. Strutt. On the interference bands of approximately homogeneous light; in a letter to Prof. A. Michelson. *Phil.Mag.*, 34:407–411, 1892.

[125] J. Sylvester. Thoughts on inverse orthogonal matrices, simultaneous sign successions, and tessellated pavements in two or more colours, with applications to Newton's rule, ornamental tile-work, and the theory of numbers. *Philosophical Magazine*, 34:461–475, 1867.

[126] M. A. Topinka, B. J. LeRoy, S. E. J. Shaw, E. J. Heller, R. M. Westervelt, K. D. Maranowski, and A. C. Gossard. Imaging coherent electron flow from a quantum point contact. *Science*, 289:2323–2326, September 2000.

[127] M. A. Topinka, B. J. LeRoy, R. M. Westervelt, S. E. J. Shaw, R. Fleischmann, E. J. Heller, K. D. Maranowski, and A. C. Gossard. Coherent branched flow in a two-dimensional electron gas. *Nature*, 410:183–186, Mar. 2001.

[128] B. van Wees. Quantized conductance of point contacts in a two-dimensional electron gas. *Phys. Rev. Lett.*, 60(9):848–850, 1988.

[129] D. Vandervoorde and N. M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 2003.

[130] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995.

[131] T. L. Veldhuizen. Blitz++: The library that thinks it is a compiler. In H. Langtangen, A. Bruaset, and E. Quak, editors, *Advances in Software Tools for Scientific Computing*, volume 10 of *Lecture Notes in Computational Science and Engineering*, pages 57–87. Springer, 2000.

[132] T. L. Veldhuizen. C++ templates are turing complete. Technical report, Indiana University, 2003.

[133] J. von Neumann. *Functional Operators, Vol II. The geometry of orthogonal spaces*, volume 22 of *Ann. Math Stud.* Princeton University Press, 1950. Reprint of mimeographed lecture notes first distributed in 1933.

[134] J. Walsh. *Approximation by polynomials in the complex domain*. Number v. 73 in Mémorial des sciences mathématiques. Gauthier-Villars, 1935.

[135] J. Walsh. *Interpolation and approximation by rational functions in the complex domain*. Colloquium Publications - American Mathematical Society. American Mathematical Society, 1969.

[136] M. Wang, H. Klie, M. Parashar, and H. Sudan. Solving sparse linear systems on nvidia tesla gpus. In *Proceedings of the 9th International Conference on Computational Science: Part I*, ICCS '09, pages 864–873, Berlin, Heidelberg, 2009. Springer-Verlag.

[137] B. C. Weare and J. S. Nasstrom. Examples of Extended Empirical Orthogonal Analyses. *Monthly Weather Rev.*, 110:481–485, 1982.

[138] U. Weiss. *Quantum Dissipative Systems*. World Scientific Publishing Company Incorporated, 2012.

[139] J. Weng, T. Shimobaba, N. Okada, H. Nakayama, M. Oikawa, N. Masuda, and T. Ito. Generation of real-time large computer generated hologram using wavefront recording method. *Opt. Express*, 20(4):4018–4023, Feb 2012.

[140] D. A. Wharam, T. J. Thornton, R. Newbury, M. Pepper, H. Ahmed, J. E. F. Frost, D. G. Hasko, D. C. Peacock, D. A. Ritchie, and G. A. C. Jones. One-dimensional transport and the quantisation of the ballistic resistance. *Journal of Physics C: Solid State Physics*, 21(8):L209, 1988.

[141] P. Wiemann, S. Wenger, and M. Magnor. CUDA expression templates. In *WSCG Communication Papers Proceedings 2011*, pages 185–192, Jan. 2011. ISBN 978-80-86943-82-4.

[142] F. Woittequand, M. Monnerville, and S. Briquez. Iterative time independent calculation of the cumulative reaction probability within a basis adapted preconditioner. *Chemical Physics Letters*, 417(4-6):492 – 497, 2006.

[143] S. Wold, K. Esbensen, and P. Geladi. Principal component analysis. *Chemometrics and Intelligent Laboratory Systems*, 2(1-3):37 – 52, 1987.

[144] S. Xu, W. Xue, K. Wang, and H. X. Lin. Generating approximate inverse preconditioners for sparse matrices using cuda and gpgpu. *Journal of Algorithms & Computational Technology*, 5(3):475 – 500, 2011.