Space-efficient and exact system representations for the nonlocal protein electrostatics problem

Dissertation
zur Erlangung des Grades
„Doktor der Naturwissenschaften”
am Fachbereich Physik, Mathematik und Informatik
der Johannes Gutenberg-Universität
in Mainz

Thomas Kemmer

geb. in Wittlich
Mainz, den 14. Oktober 2020

1. Berichterstatter: Prof. Dr. Andreas Hildebrandt
2. Berichterstatter: Prof. Dr. Bertil Schmidt
Tag des Kolloquiums: 09.03.2021

D77

Abstract

The study of proteins and protein interactions, which represent the main constituents of biological functions, plays an important role in modern biology. Application settings such as the development of pharmaceutical drugs and therapies rely on an accurate description of the electrostatics in biomolecular systems. This particularly includes nonlocal electrostatic contributions of the water-based solvents in the cell, which have a significant impact on the long-range visibility of immersed proteins. Existing mathematical models for the nonlocal protein electrostatics problem can be approached with numerical standard techniques, including boundary element methods (BEM). However, the typically large, dense, and asymmetric matrices involved in discretized BEM formulations previously prevented the application of the method to realistically-sized biomolecular systems. Here, we overcome this obstacle by developing implicit yet exact representations for such BEM matrices, capturing trillions of floating-point values in only a few megabytes of memory and without loss of information. We present generalized reference implementations for our implicit matrix types alongside specialized matrix operations for the Julia and Impala programming languages and demonstrate how they can be utilized with existing linear system solvers. On top of these implementations, we develop full-fledged CPU- and CUDA-based BEM solvers for the nonlocal protein electrostatics problem and make them available to the scientific community. In this context, we show that our solvers can perform dozens of matrix-vector products for the previously inaccessible BEM systems within a few seconds or minutes and thus allow, for the first time, to solve the employed BEM formulations with exact system matrices in the same time frame.

German summary

Proteine und ihre Interaktionen mit anderen Biomolekülen in der Zelle bilden eine wichtige Grundlage biologischer Funktionen. Studien im Bereich dieser Interaktionen verwenden, je nach Problemstellung, unterschiedliche mathematische Modelle zur Beschreibung der beteiligten Energien. Im Bereich der Medikamentenentwicklung ist dabei beispielsweise eine möglichst präzise Modellierung der Elektrostatik des Systems entscheidend, da diese maßgeblich am Findungsprozess potenzieller Bindungspartner beteiligt ist. Die strukturelle Komplexität der wasserbasierten Lösungsmittel in der Zelle stellt hier eine besondere Herausforderung dar, denn Dipolmomente sowie der Drang zur Ausbildung von Wasserstoffbrückenbindungen schränken die Bewegungsfreiheit der Wassermoleküle in der unmittelbaren Umgebung gelöster Proteine stark ein. Diese Effekte können z. B. durch das theoretische Rahmenwerk der *nichtlokalen Elektrostatik* beschrieben und die resultierenden Gleichungssysteme mit Standardverfahren der Numerik gelöst werden. Hier betrachten wir eine existierende Formulierung des Proteinelektrostatikproblems unter Verwendung einer Randelementmethode (engl. boundary element method, BEM), deren Beitrag zur genaueren Berechnung elektrostatischer Potentiale in der Vergangenheit bereits für kleine Biomolekularsysteme gezeigt werden konnte, deren praktische Anwendbarkeit auf größere Systeme jedoch bislang durch ihre immensen Speicheranforderungen eingeschränkt war.

Im Rahmen dieser Dissertation entwickeln wir implizite Repräsentationen für die üblicherweise dicht besetzten und nicht symmetrischen Systemmatrizen der in den numerischen Lösungsprozess involvierten linearen Gleichungssysteme, die der Grund für die oben genannten Einschränkungen sind. Unsere impliziten Repräsentationen reduzieren den ursprünglich quadratischen Speicherbedarf auf einen linearen, ohne dabei Informationen zu verlieren, so dass Matrizen mit mehreren Milliarden Elementen problemlos mit wenigen Megabytes im Speicher dargestellt werden können. Wir präsentieren außerdem Referenzimplementierungen für unsere impliziten Matrixtypen sowie Operationen für selbige in den Programmiersprachen Julia und Impala und beschreiben, wie diese in vorhandenen Lösern für lineare Gleichungssysteme zur Anwendung kommen oder die Repräsentationen auf andere Probleme übertragen werden können.

Auf dieser Grundlage entwickeln wir schließlich vollwertige Randelementlöser für das nichtlokale Proteinelektrostatikproblem, die zum ersten Mal in der Lage sind, praxisnahe Eingabegrößen verarbeiten und Lösungen innerhalb weniger Sekunden oder Minuten bereitstellen zu können. Dazu implementieren wir verschiedene Varianten der allgemeinen Matrix-Vektor-Multiplikation für CPUs und GPUs (unter Verwendung von NVIDIAs bekannter CUDA-Plattform), deren Performanz wir hier auf unterschiedlichen Rechnern gegenüberstellen. Ein besonderes Augenmerk legen wir dabei auf unsere Julia-Implementierungen, die durch das geschickte Ausnutzen von Spracheigenschaften eine Manipulation von eigentlich seriellen und rein CPU-basierten Gleichungssystemlösern erlauben, die effektiv zu einer (wahlweise CPU- oder GPU-basierten) Parallelisierung der Löser führen. Ein weiteres Augenmerk legen wir auf die domänenspezifische Implementierung unserer Impala-Lösungen, die für die gewählte Zielplattform spezialisierte und optimierte Programme aus einer weitgehend plattformunabhängigen Codebasis generiert, was einerseits den Wartungsaufwand der Implementierung drastisch verringert und andererseits das einfache Hinzufügen oder Austauschen von Plattformen ermöglicht. Alle im Rahmen dieser Arbeit entwickelten Softwarepakete sind quelloffen und frei verfügbar, so dass die hier präsentierten Ergebnisse leicht nachvollzogen und unsere Randelementlöser insbesondere für wissenschaftliche Zwecke eingesetzt werden können.

Acknowledgments

First and foremost, I want to express my gratitude to my advisor, Professor Dr. Andreas Hildebrandt, who introduced me into the world of nonlocal protein electrostatics, guided me through the early stages of this project and provided invaluable feedback ever since. I am also indebted to my second reviewer, Professor Dr. Bertil Schmidt, for several constructive discussions on the high-performance capabilities of the system solving process.

Furthermore, I want to thank my dear friends who assisted me with proofreading, for every bit of nitpicking, all fruitful discussions, and for accompanying me on my never-ending quest for working through unspeakable amounts of coffee and sushi. I am also deeply grateful to my significant other for her endless support and encouragement over the course of the writing process, for proofreading, and for company during roughly one or two all-nighters.

I would like to thank my friends and colleagues at the Institute of Computer Science, the Scientific Computing and Bioinformatics group, and the Integrated Research Training Group for a familiar working atmosphere and numerous discussions on diverse topics.

Last but not least, I am indebted to my former supervisor, who introduced me into a more structured approach to scientific writing during my Master studies. Although she was not involved in this manuscript or the projects leading to it, her invaluable feedback to my Master's thesis a long time ago sharpened my awareness of stylistic consistency permanently and thus helped me, once again, through the writing process.

Abbreviations

AoS	Array of structs
BEM	Boundary element method
DSL	Domain-specific language
FDM	Finite difference method
FEM	Finite element method
GMRES	Generalized minimal residual method
GPGPU	General-purpose computing on GPUs
IR	Intermediate representation
JIT	Just in time
PE	Partial evaluation
REPL	Read-eval-print loop
SIMD	Single instruction, multiple data
SIMT	Single instruction, multiple threads
SM	Streaming multiprocessor
SoA	Struct of arrays

Contents

1 Introduction

Proteins primarily exert biological functions through a highly complex network of interactions with other biomolecules at cellular level. While the vast variety of possible interaction partners enables a range of chemical reactions fit to power and maintain an organism as complex as a human being, the same mechanism provides a broad attack vector for diseases, implemented as alterations of the interaction network. A deeper understanding of the network's nodes and connections allows for a targeted manipulation of pathogenic protein interactions through drugs and therapies while minimizing the risk of unwanted side effects. In the modeling of protein-protein and protein-ligand interactions, electrostatic contributions of the binding partners and the cell environment are of particular importance for the long-range visibility of the molecules. More specifically, the electrostatic profile of the protein actively influences the binding process by guiding potential interaction partners through the cell and thus substantially increases the odds of two binding partners meeting as compared to pure random thermal motion.

Many commonly used interaction models oversimplify electrostatic contributions of the proteins, their binding partners, or the cell environment. As a consequence, they are only of limited use in the accurate prediction of protein interactions in a long-range setting. Two common simplifications are the negligence of molecular boundaries in the system and the underestimation of polarization effects from the water-based solvents in the cells. These issues can be addressed, for example, with the framework of *nonlocal electrostatics*, which respects solvent molecule interdependencies and captures their amplifying or dampening effects on the electrostatic profiles of immersed proteins. In direct comparison to simpler models, those derived from nonlocal electrostatics are comparatively expensive to compute. In this manuscript, we focus on a particular numerical solution scheme to the problem, namely, a *boundary element method* (BEM) for the nonlocal electrostatics of biomolecules in structured solvents.

BEM formulations reduce the problem entirely to the molecular boundary and, as a result, only require a two-dimensional mesh of the protein surface rather than a three-dimensional representation of the whole system domain. Also, the BEM approach does not require initial values for the outer boundaries of the system,

in contrast to other solution methods, such as finite element methods or finite difference methods. A major challenge of implementing the method arises from the discrete linear systems involved in the solving process: The systems usually contain dense and asymmetric system matrices. What is more, their dimensions scale quadratically with respect to the area and resolution of the approximated protein surfaces. Represented explicitly in this form, most real-world biomolecular systems would exceed several terabytes worth of data, with almost all of them being contained in the system matrices alone.

Here, we develop space-efficient and exact implicit representations for such *BEM system matrices* that can be used with standard iterative solving techniques for linear systems. We start with preexisting BEM formulations for the protein electrostatics problem and identify recurring patterns that serve as a basis for the implicit representations. Most of the generalizations outlined here originate in the discretization process of the continuous BEM formulation and are not specific to the protein electrostatics problem, thus conceptionally enabling their application to other BEM problems as well. We provide multiple different implicit representations for the presented systems alongside reference implementations in two programming languages: Firstly, we use the versatile *Julia* language [1] as a representative for a high-performance yet broadly accessible programming language that is well-suited for the needs of scientific computing. Secondly, we utilize the *Impala* language, which is part of the *AnyDSL* framework [2] for the shallow embedding of domain-specific languages by means of partial code evaluation.

Based on our implicit system matrix representations, we further develop specialized versions of the operations used in the iterative solving process. In particular, we focus on generic matrix-vector products as the typically dominant contributor to the solving time and provide parallelized implementations of these product functions as CPU-based or GPU-accelerated variants. For the latter variants, we utilize NVIDIA's famous *CUDA* platform [3], which is supported through both Julia and Impala. Our efforts are realized in a total of four open-source software packages that we make available publicly and free of charge[1]:

- **ImplicitArrays.jl**: General-purpose implementations for various implicit vector and matrix types for Julia,

- **NESSie.jl**: Fully functional CPU-based BEM solvers and post-processors for the nonlocal protein electrostatics problem in the Julia language,

[1] `https://github.com/tkemmer`

- **CuNESSie.jl**: CUDA-accelerated variants of the solvers and post-processors provided by NESSie.jl,
- **AnyBEM**: Prototypical implementations of the implicit system matrix representations as well as CPU- and GPU-based parallel matrix operations in Impala.

In the Julia context, we show how to exploit Julia language features to "inject" our specialized operations into the solving workflow of existing iterative solvers, allowing us to effectively turn originally serial solvers into parallelized ones. In the Impala context, we show how the protein electrostatics problem in conjunction with our implicit system matrix representations can be used to generate dedicated solver flavors (e.g., CPU- or GPU-based solvers for single- or double-precision floating-point values) from a common code base.

Using the above packages, we demonstrate how our implicit representations are able to reduce the initially quadratic memory footprint of the BEM systems for several small proteins from up to a few terabytes to only a few megabytes. To the best of our knowledge, this reduction to a linear memory footprint (while still representing the original matrices exactly) allows, for the first time, to solve such systems within realistic time and memory constraints. For example, the computation time of nonlocal system matrices can be roughly reduced by two orders of magnitude through GPU acceleration, as compared to a serial CPU-based implementation. This demotes the computation of individual matrix-vector products for system matrices with hundreds of millions of elements to a matter of seconds and results in solving times (involving a few hundred matrix-vector products in the process) on the order of minutes for the same systems.

This manuscript is structured as follows: Chapter 2 provides an introduction into the matter of protein electrostatics and its role in protein interactions. In this context, we outline different approaches to a more accurate treatment of the present solvents and outline related work with regard to numerical formulations and solvers for the nonlocal protein electrostatics problem. In Chapter 3, we analyze a specific set of continuous and discrete BEM formulations for the protein electrostatics problem in order to extract common patterns. Our findings then serve as a basis for both our implicit system matrix representations and specialized matrix operations for the BEM solvers later on in this manuscript. Chapter 4 presents the programming platforms selected for the reference implementations of the implicit system representations, specialized matrix operations, and BEM solvers. Here, we discuss how the protein electrostatics problem can benefit from specific features of the respective platforms. Chapter 5 then lays the foundation for the implicit system representations and reference implementations. After a short introduction of iterative solvers for

linear systems of equations, we develop the theoretical constructs for an exact yet implicit representation of the BEM systems at hand and, possibly, other BEM matrices employing the same discretization schemes. More specifically, we present *interaction matrices* as a common building block of the linear systems, provide reference implementations for both Julia and Impala, and discuss how these representations can be used with iterative solvers. In Chapters 6, 7, and 8, we provide in-depth descriptions of the aforementioned BEM solver packages NESSie.jl, CuNESSie.jl, and AnyBEM, with an emphasis on considerations specific to the underlying platforms. Chapter 9 then gives insights into the performance of the reference implementations. For this, we first introduce the experimental setup and datasets and evaluate our BEM solvers for analytically solved models, before we present the runtime performance results for the CPU- and GPU-based matrix-vector products and the BEM solvers themselves. In Chapter 10, we finally conclude with a summary of the topics presented in this manuscript and our contributions to the nonlocal protein electrostatics problem in particular. In addition, we given an outline of future directions for our system representations and solvers.

Nonlocal protein electrostatics

2

In this chapter, we give a short introduction into the biological and physical backgrounds of nonlocal protein electrostatics. We particularly focus on protein interactions as an implementation of biological functions, which is the fundamental basis for the development of drugs and therapies. For this, we introduce the protein structure hierarchy alongside common protein interaction models. In the second part of this chapter, we try to establish a basic understanding for the contributions of electrostatic terms to a more accurate representation of protein interactions and outline the differences between local and nonlocal protein electrostatics with regard to the cell environment. Finally, we give an overview over related work in the context of numerical solvers for the nonlocal protein electrostatics problem.

2.1 Protein interactions

At the cellular level, biological function is mainly implemented through the interaction of molecular entities such as proteins, rendering the study of proteins and their interactions with other biomolecules a cornerstone of modern biology. In the literature, proteins are described as “the workhorses of the cell” [4] or as “the structural and enzymatic elements of an organism – they are the working parts as well as the building material” [5]. Generally, proteins do not only interact with other proteins but also with DNA, RNA, and various other molecules. The exact characteristics of the interactions depend on the associated biomolecular task, ranging from a broad variety of possible interaction partners (e.g., for protein digestion) to highly specific interactions (e.g., to catalyze reactions under certain environmental conditions).

The set of biomolecular interactions corresponding to a given cell or organism is commonly referred to as the *interactome*, although the term’s definition differs between publications[1]. Alterations of the interactome can lead to a plethora of diseases in an organism (e.g., cancer [8]). At the same time, such alterations serve as a basis for the development of therapies and can be induced intentionally

[1] Some authors loosely define the interactome as “the complete repertoire of interactions potentially encoded in [an organism’s] genome” [6], while others use the same term to specifically refer to protein-protein interactions (e.g., [7, 8]).

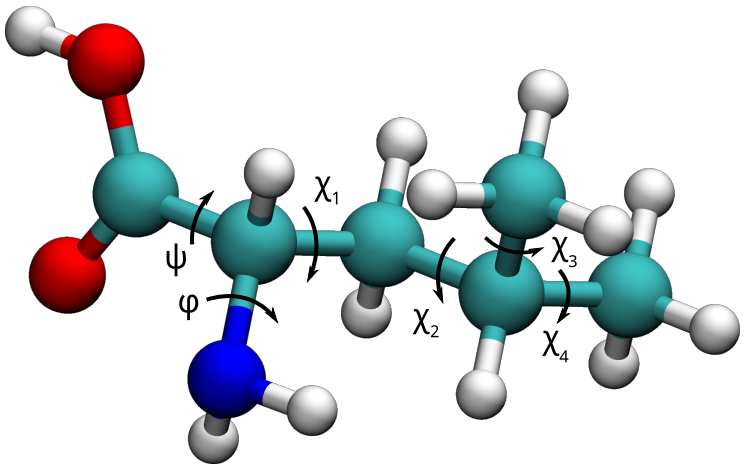


Figure 2.1. – Leucine molecule shown with backbone torsion angles φ and ψ and side chain torsion angles χ_i as an example for amino acid flexibility

(e.g., via drugs) to combat said diseases. A classic yet effective approach to this matter is the regulation of a single protein involved in a harmful interaction rather than a regulation of the whole interaction network. This is often implemented by introducing alternative binding partners into the organism that either compete with or bind to the original binding partner of the protein and thus interfere with the interaction. For a more detailed review of this topic, including alternative approaches, the interested reader is referred to the corresponding book series edited by Thomas Lengauer [9].

2.1.1 Protein structure hierarchy

Amino acids serve as the central building blocks of all proteins. While carrying different side chains, the protein-building (i.e., *proteinogenic*) amino acids share a common molecular structure that allows them to be linked together linearly into polypeptides, which constitute the respective proteins. Each proteinogenic amino acid is assigned a unique letter of the alphabet, such that each protein can be simply described as a sequence of characters. This *protein sequence* is then used to represent proteins in databases such as the *Universal Protein Resource (UniProt)* [10]. While the peptide bond (i.e., the connection point) between linked amino acids is mostly rigid, the protein *backbone* contains up to two degrees of rotational freedom per amino acid, in addition to the flexibility of their side chains (see Figure 2.1 for an example). The resulting flexibility allows proteins to fold into complex structures. The folding space is naturally restricted by factors such as steric collisions and is further influenced by other factors like the physicochemical properties of

the amino acids. Early experiments suggested that a protein's structure is often directly determined by its sequence [11, 12] and efforts have been made to predict protein function directly from the sequence [13]. However, many proteins can be observed in different conformations, heavily influenced by the composition of the cellular environment or by binding partners. For example, many proteins undergo conformational changes during the binding process (cf. Section 2.1.3) or are known to possess no canonical structure (or intrinsically unstructured regions) in the first place [14, 15]. Other reasons include alterations of the protein after its synthesis in the cell, commonly referred to as *post-translational modifications* (PTM), which can change the activity and properties of the protein.

The different characteristics of protein structure are usually captured through a hierarchical classification system: On the first level, each protein is described by its peptide sequence, referred to as its *primary structure*. The *secondary structure* comprises local structures commonly found across many proteins, including helices or β-sheets, i.e., hydrogen bond-induced zigzag patterns of mostly flat amino acid strands. The 3D structure of a protein, that is, the coordinates of all constituent atoms, is captured as its *tertiary structure*, while complexes composed of multiple proteins (or *subunits*) are referred to as *quarternary structures*.

Different approaches exist to classify proteins or associate them to specific biological functions based on common sequence or structural patterns. Notable classification systems include SCOP2 (*Structural Classification of Proteins*) [16, 17] and Pfam [18]. Today, thousands of protein structures are available through databases such as the *Protein Data Bank* (PDB) [19], most of which having been collected by means of *X-ray crystallography* (XRC), *nuclear magnetic resonance (NMR) spectroscopy*, or *3-dimensional electron microscopy* (3DEM). Notwithstanding the comparative abundance of available protein structures these days, the prediction of structural conformations and protein function still remains a challenging task.

2.1.2 The human proteome

According to the central dogma of molecular biology, proteins are synthesized from information encoded in the DNA. More specifically, protein-coding regions of the DNA are transcribed into mRNA (messenger RNA) molecules, which are further translated into proteins, effectively converting triplets of nucleotides (so-called *codons*) into amino acids [20, 21]. Such DNA regions are commonly referred to as *genes*, although the same term also refers to DNA regions encoding for functional RNA molecules, which are not further translated into proteins.

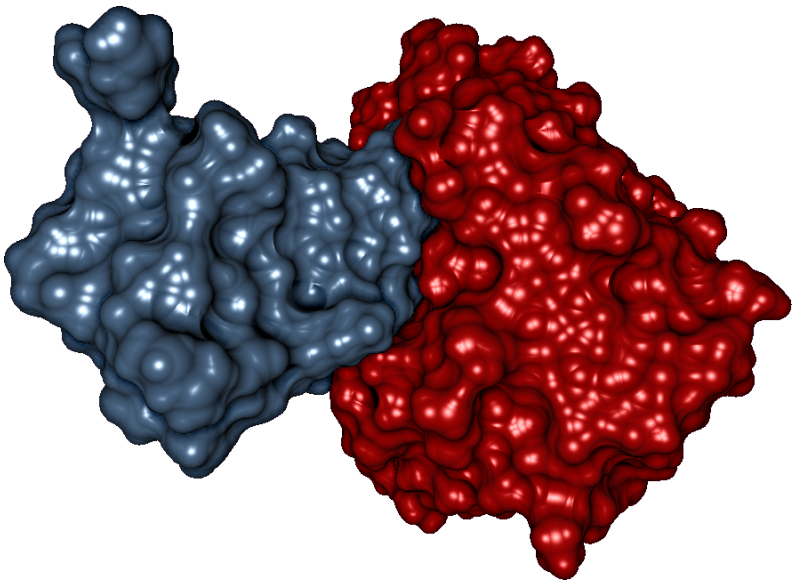


Figure 2.2. – SES (solvent-excluded surface) representation of a protein complex between bovine pancreatic trypsin inhibitor (BPTI, blue/left) and trypsin (red/right), generated from PDB ID 4Y0Y using the BALLView application [30]

Counting the number of human genes and proteins or, in other words, determining the size of the human *genome* and *proteome*, has been an ongoing effort over the past decades. Early estimates fell typically in the range between 50,000 and 100,000 genes [22, 23]. With the first publication of the human genome in 2001 [24] in the context of the *Human Genome Project (HGP)*, several studies suggested new estimates between 19,000 and 35,000 genes [24, 25, 26, 27, 28], with current numbers usually falling in the range between 20,000 to 21,000 distinct protein-coding genes [22, 23]. At the time of writing, UniProt lists 20,359 entries corresponding to the human proteome (UP000005640), accounting for one representative protein per protein-coding gene [29]. Due to a process called *alternative splicing*, the same gene can be synthesized into different proteins (*isoforms*). UniProt currently attributes roughly 22,000 additional protein sequences to this process in the human genome [29], totaling in more than 40,000 different sequences in the human proteome that fold into three-dimensional structures and eventually take part in protein interactions.

2.1.3 Protein docking

As described earlier, protein function is primarily implemented through protein interactions. This specifically includes complex building of proteins with other proteins or smaller ligands and is of particular importance in the context of rational drug design. Being able to predict how two such biomolecules bind together is

known as the *docking problem* and dates back to the late 1970s [31]. The docking problem accounts not only for the relative orientation of the binding partners (also called the *binding mode*, see Fig. 2.2 for an example) but also for the stability of the complex, i.e., the resulting binding free energy (or *binding affinity*) [32].

In order to form a stable complex, the binding partners require a strong geometric compatibility in the vicinity of the binding site (or *binding pocket*) as well as sufficiently compatible physicochemical properties. Historically, the geometric compatibility of the binding partners was assumed to be a hard prerequisite of the binding process. However, this so-called *lock-and-key principle* [33] was further developed into an *induced fit* model [34] more than half a century later, which accounted for conformational changes of the molecules during the binding process. Over the course of the past decades, a plethora of different approaches to the docking problem have been developed, with focuses ranging from a fast database screening for potential binding partners (*virtual screening*) to predictions and scorings that are as accurate as possible. Depending on rigidity of the protein's binding partner (e.g., a rigid protein vs. a highly flexible ligand), researchers can adopt suitable docking techniques, such as rigid body docking (e.g., via the Katchalski-Katzir algorithm [35]) or those accounting for ligand flexibility. The latter category of docking tools plays a key role in the discovery of small drug molecules, with prominent tools including DOCK [36], AutoDock Vina [37], Glide [38, 39], and GOLD [40].

2.1.4 Force field models

Protein interactions are based on electromagnetism, including self-interactions (to form stable conformations) as well as interactions with potential binding partners and the cell environment. Overall, the interactions within a biological system are implemented through many (comparatively small) energy contributions from various sources. Unfortunately, quantum-level descriptions of most systems are too expensive to be efficiently included in numerical applications like the prediction of protein structures or scoring of docking results. While such descriptions can still be used for refinement purposes downstream [41], less complex system descriptions are often employed in practice. In particular, simple models from classical mechanics (e.g., ball-and-spring models) are commonly used as a proxy for the original biological systems. These simpler models are generated on the basis of experimental observations or quantum-chemical expectations, with interactions being described through *force field* models. The latter commonly employ discrete *energy functions* over terms representing different bonded and non-bonded interactions between the atoms of the system. Well-known force fields for biomolecules include AMBER [42], GROMOS

[43], and CHARMM [44]. While the exact approaches to modeling interactions vary between force fields, most of the following components are usually included in the corresponding energy functions: angles between and lengths of covalent bonds (e.g., modeled as harmonic springs), bond rotations, induced dipoles, Van der Waals interactions, electrostatic interactions, and hydrophobic effects. Generally, these energy functions are constructed in a way that is comparatively easy to compute, especially in application settings where they need to be evaluated frequently, e.g., during protein docking or molecular dynamics simulations.

2.2 Protein electrostatics

Among the force field components, the electrostatic contributions take on a special role: Not only are they one of a few terms to model non-bonded interactions in the system but they also represent the only long-ranged interaction term. In other words, for most of the other components to take effect in the binding process, the binding partners already need to be in close proximity to each other. Hence, the long-range visibility of a protein or, more specifically, its electrostatic profile, is the key contributor to guided interactions with other biomolecules in the cell. While the random thermal motion of cell particles would bring binding partners close together every once in a while, the electrostatic profile of the protein helps increasing these chances, e.g., by acting as a funnel to guide potential binding partners [45, 46].

As we will see in a moment, the electrostatic profile of proteins is often heavily influenced by the solvents they are immersed in, especially if they happen to be as structured as water. As a result, the long-range visibility of these proteins can be partly amplified or dampened by the solvent. This is why an accurate description of the true electrostatic profile of the protein is crucial when the binding partners cannot be forced into contact through other means but have to meet naturally. This is particularly important, e.g., for drug molecules that need to find their way through the cell and into the vicinity of the protein they are supposed to bind to.

2.2.1 Local cavity model

In many force fields, the electrostatic contributions to the energy functions are based on simplistic models that can be computed efficiently. For this, the (continuous) charge distributions in the system are discretized by assigning (full or partial) charges to discrete locations of the Euclidean space, e.g., at the atom centers. This

so-called *point charge model* can be easily represented for N charge locations as a single function

$$\rho(\boldsymbol{r}) = e_c \sum_{i=1}^{N} \zeta_i \delta(\boldsymbol{r} - \boldsymbol{r_i}), \tag{2.1}$$

where e_c is the elementary charge and ζ_i and $\boldsymbol{r}_i$ are the dimensionless charge and position of the i-th point charge, respectively. δ denotes the *Dirac delta function*, which takes on a value of $\delta(x - a) = 0$ for all $x \neq a$ and $\int \delta(x - a)\mathrm{d}x = 1$ if a is included in the domain of integration (or 0, otherwise).

For such a point charge representation, the electrostatic energy of a given system can be computed from Coulomb's law as

$$\frac{e_c^2}{4\pi\varepsilon_0\varepsilon} \sum_{i=1}^{N} \sum_{\substack{j=1 \\ i<j}}^{N} \frac{\zeta_i \zeta_j}{||\boldsymbol{r}_i - \boldsymbol{r}_j||_2},$$

with ε_0 being the *vacuum permittivity* and ε being a scalar constant. This model assumes that the whole Euclidean space is either a vacuum ($\varepsilon = 1$) or filled with a homogeneous medium that acts as a "polarizable vacuum" [4] with a dielectric permittivity ε relative to ε_0.

Although models of this form can be implemented efficiently, they are based on multiple simplifications that impact the predicted electrostatic profiles of proteins in the cell, especially from a long-range perspective. Biomolecules are typically immersed in water-based solvents and surrounded by mobile ions. However, the above model does not account for the electrostatic contributions arising from solvent molecule interdependencies, possibly leading to highly inaccurate energy estimates. While the missing contributions could be explicitly modeled in the Coulomb term, the solvent molecules alone vastly outnumber the biomolecule atoms in most cases, leading to huge and costly sums[2]. What is more, the above model does not contain molecular boundaries, that is, the medium is assumed to permeate even the biomolecule.

A solution for the latter problem can be obtained by treating biomolecule and solvent as different domains of the Euclidean space and model electrostatic interactions for these domains separately. Figure 2.3 shows a schematic domain decomposition for this so-called *cavity model*, where the biomolecule is represented by a domain $\Omega \subset \mathbb{R}^3$ and the solvent is represented by a domain $\Sigma := \mathbb{R}^3 \setminus \bar{\Omega}$, both of which are linked through an interface domain $\Gamma := \partial\Omega$. Through the remainder of this manuscript, we will usually refer to Ω, Σ, and Γ as *protein domain*, *solvent domain*,

[2] Since electrostatic interactions are not limited in range, a highly accurate description would need to account for all solvent molecules in the whole Euclidean space. But even in a reasonably sized sub-domain, the difference in numbers exceeds several orders of magnitude.

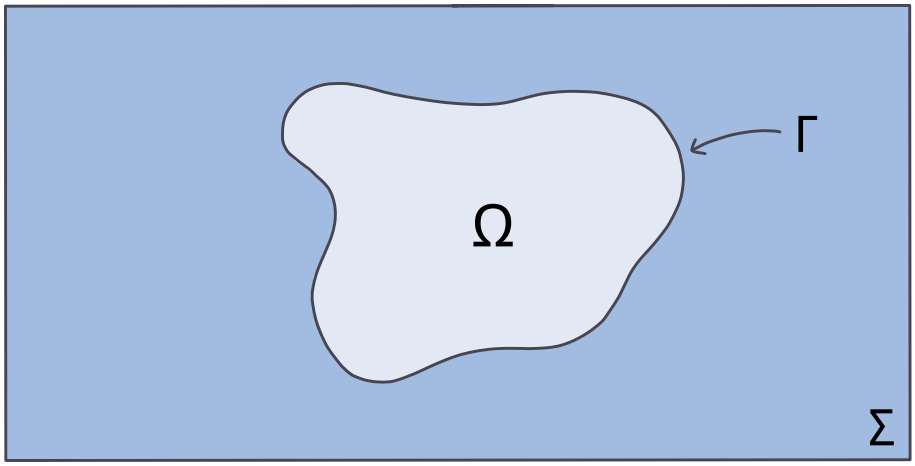


Figure 2.3. – Schematic domain decomposition for the cavity model, with molecule domain $\Omega \subset \mathbb{R}^3$, solvent domain $\Sigma := \mathbb{R}^3 \setminus \bar{\Omega}$, and molecular boundary $\Gamma := \partial\Omega$

and *surface domain*, respectively. It should be noted here that such a sharp molecular boundary is in itself of course only a simplified model. While different definitions for boundaries of this kind exist, e.g., based on the Van der Waals radii associated with the constituent atoms, we do not restrict our work to any specific definition as long as it can be sufficiently represented by Γ.

The cavity model can then be used in conjunction with the well-known Maxwell equations [47, 48], which, for vacuum electrostatics, assume the following form:

$$\boldsymbol{\nabla} \cdot \boldsymbol{E} = \frac{\rho}{\varepsilon_0} \qquad\qquad \boldsymbol{\nabla} \times \boldsymbol{E} = 0$$

or, equivalently,

$$-\Delta\varphi = \frac{\rho}{\varepsilon_0},$$

with $\boldsymbol{E}$ being the electric field, ρ being a charge distribution, $\Delta := \boldsymbol{\nabla} \cdot \boldsymbol{\nabla}$ being the Laplace operator, and φ being the *electrostatic potential*.

Boundary conditions for the electric field can be derived from these equations for the special case of a sufficiently smooth domain-separating boundary surface like Γ being present [49]. More specifically, assuming that ρ is well-behaved and continuous on the surface, then the boundary conditions for any point $\boldsymbol{r}$ on Γ are given by

$$(\boldsymbol{E}(\boldsymbol{r}^+) - \boldsymbol{E}(\boldsymbol{r}^-)) \cdot \hat{\boldsymbol{n}}_{\boldsymbol{r}} = 0$$

and

$$(\boldsymbol{E}(\boldsymbol{r}^+) - \boldsymbol{E}(\boldsymbol{r}^-)) \times \hat{\boldsymbol{n}}_{\boldsymbol{r}} = 0,$$

where $\hat{\boldsymbol{n}}_{\boldsymbol{r}}$ represents the normal vector on Γ corresponding to $\boldsymbol{r}$ and where $\boldsymbol{r}^+ \in \Omega$ and $\boldsymbol{r}^- \in \Sigma$ are two points on the line through $\hat{\boldsymbol{n}}_{\boldsymbol{r}}$, infinitesimally close to $\boldsymbol{r}$, and approaching $\boldsymbol{r}$ from both directions.

The Maxwell equations are then used in their macroscopic form along with suitable material equations to capture polarization effects of the medium. In the case of linear material equations as used for the formulations in Chapter 3, the macroscopic equations assume the following form:

$$-\varepsilon_0 \varepsilon(\boldsymbol{r}) \Delta\varphi(\boldsymbol{r}) = \rho(\boldsymbol{r}). \tag{2.2}$$

Here, $\varepsilon(\boldsymbol{r})$ denotes the *dielectric response function* of the medium, where the polarization effects of the latter under the influence of the biomolecule are modeled through a spatial dependency. This model further assumes that the dielectric response is independent of direction (*isotropy assumption*) and that the solvent molecules move independently from each other (*locality assumption*).

Applying Eq. (2.2) to the separate domains of the cavity model, coupled through the above boundary conditions, finally yields a *local cavity model* for the protein electrostatics problem:

$$\left.\begin{aligned} -\varepsilon_0 \varepsilon_\Omega \Delta\varphi_\Omega(\boldsymbol{r}) &= \rho(\boldsymbol{r}) && \boldsymbol{r} \in \Omega \\ \boldsymbol{\nabla} \cdot (\varepsilon_\Sigma(\boldsymbol{r}) \boldsymbol{\nabla}\varphi_\Sigma(\boldsymbol{r})) &= 0 && \boldsymbol{r} \in \Sigma \\ \hat{\boldsymbol{n}}_{\boldsymbol{r}} \cdot [\varepsilon_\Omega \boldsymbol{\nabla}\varphi_\Omega(\boldsymbol{r}) - \varepsilon_\Sigma(\boldsymbol{r}) \boldsymbol{\nabla}\varphi_\Sigma(\boldsymbol{r})] &= 0 && \boldsymbol{r} \in \Gamma \\ [\varphi_\Omega(\boldsymbol{r}) - \varphi_\Sigma(\boldsymbol{r})] &= 0 && \boldsymbol{r} \in \Gamma \end{aligned}\right\} \tag{2.3}$$

In this form, the potential and dielectric response functions are replaced by their domain-specific variants φ_Ω, φ_Σ, ε_Ω, and ε_Σ. Moreover, the dielectric response of the protein is assumed to be constant and all charges need to be contained in the protein domain. It is important to note here that the latter restriction is biologically unrealistic. As stated above, the water-based solvent in the cell contains mobile ions, let alone other biomolecules. The solvent domain-bound equation in Eq. (2.3) is thus commonly replaced by a suitable Poisson-Boltzmann or Debye-Hückel equation.

2.2.2 Nonlocal cavity model

Although the local cavity model is a big improvement in terms of accurately modeling biomolecular systems, as compared to the simplistic Coulomb model, the locality assumption poses still a huge caveat for structured solvents such as water. In fact, the

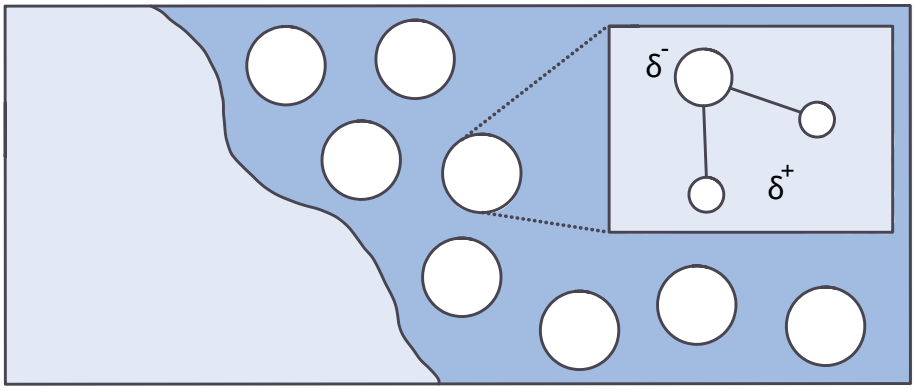


Figure 2.4. – Schematic representation of water molecules around the molecular surface (not to scale). Due to dipole moments and hydrogen bonds, the molecules cannot move independently.

locality assumption does not hold here, as the dipole moments of and the hydrogen bond network formed by the water molecules lead to solvent interactions everywhere in space. When left undisturbed, the water molecules arrange themselves in a way that favors as many hydrogen bonds as possible. Each orientational change of any water molecule invariably threatens to break hydrogen bonds unless surrounding water molecules (and their neighbors) reorient themselves accordingly as well.

We can observe the same effect when introducing a charge distribution to the system, e.g., in the form of a protein (see Fig. 2.4). In this case, the water molecules closest to the molecular boundary (commonly referred to as *first solvation shell*) experience the strongest influence of the external electric field, imposing a pressure to align along the field (while maintaining the hydrogen bond network). The resulting changes in solvent molecule orientation amplify the dielectric shielding effect of the solvent in the presence of the protein and thus change its electrostatic profile and long-range visibility.

This effect is addressed in the framework of *nonlocal electrostatics of solvation* [50, 51], which can be implemented through an extension of the dielectric response function. In contrast to the local model, where ε_Σ only has one spatial dependency, namely, on the observation point, its nonlocal counterpart additionally accounts for solvent molecule interdependencies. For this, the previous $\varepsilon_\Sigma(\boldsymbol{r})$ is basically replaced by

$$\int_{\mathbb{R}^3} \varepsilon_\Sigma(\boldsymbol{r}, \boldsymbol{\xi}) \mathrm{d}\boldsymbol{\xi}$$

or a similar integral. In many application scenarios, the dielectric response is for example modeled to depend on a length scale λ, representing the order on which the solvent interaction strengths decay.

All of these considerations lead to a modified version of the local cavity model[3], i.e., the *nonlocal cavity model* for the protein electrostatics problem:

$$\left.\begin{aligned} -\varepsilon_0\varepsilon_\Omega\Delta\varphi_\Omega(\boldsymbol{r}) &= \rho(\boldsymbol{r}) && \boldsymbol{r}\in\Omega \\ \boldsymbol{\nabla}_{\boldsymbol{r}}\cdot\psi(\boldsymbol{r}) &= 0 && \boldsymbol{r}\in\Sigma \\ \hat{\boldsymbol{n}}_{\boldsymbol{r}}\cdot[\varepsilon_\Omega\boldsymbol{\nabla}_{\boldsymbol{r}}\varphi_\Omega(\boldsymbol{r})-\psi(\boldsymbol{r})] &= 0 && \boldsymbol{r}\in\Gamma \\ [\varphi_\Omega(\boldsymbol{r})-\varphi_\Sigma(\boldsymbol{r})] &= 0 && \boldsymbol{r}\in\Gamma \\ \int_\Sigma \varepsilon_\Sigma(\boldsymbol{r},\boldsymbol{\xi},\lambda)\boldsymbol{\nabla}_{\boldsymbol{\xi}}\varphi_\Sigma(\boldsymbol{\xi})\mathrm{d}\boldsymbol{\xi} &=: \psi(\boldsymbol{r}) && \end{aligned}\right\} \tag{2.4}$$

A concrete solvent response function is then chosen with regard to the problem at hand. For instance, water-based systems are often modeled using the so-called *Lorentzian model*, including the nonlocal formulations presented in this manuscript. More specifically, said formulations employ the following dielectric response, as taken from [49, Theorem 3.4.1]:

$$\varepsilon_\Sigma(\boldsymbol{r},\boldsymbol{\xi},\lambda) := \varepsilon_\infty\delta(\boldsymbol{r}-\boldsymbol{\xi}) + \frac{\varepsilon_\Sigma-\varepsilon_\infty}{4\pi\lambda^2}\frac{\mathrm{e}^{-\frac{|\boldsymbol{r}-\boldsymbol{\xi}|}{\lambda}}}{|\boldsymbol{r}-\boldsymbol{\xi}|}. \tag{2.5}$$

This model introduces two additional constants ε_Σ and ε_∞, which represent the macroscopic dielectric constant of the solvent and the limit of the dielectric solvent response as $\lambda\to\infty$, respectively.

2.3 Numerical solvers

As we have established above, simplistic models such as those directly based on Coulomb's law often fail to describe the electrostatics in biomolecular systems accurately. Thus, more complex models, such as local formulations in the form of Poisson-Boltzmann or Debye-Hückel equations have been in use for decades, solved with standard techniques from numerical analysis, including *finite difference methods* (FDM), *finite element methods* (FEM), and *boundary element methods* (BEM) [52, 53, 54]. However, an extension of these approaches to the nonlocal protein electrostatics problem was hampered due to its formulation as a system of partial integro-differential equations (cf. Eq. (2.4)), in contrast to the pure partial differential formulation of the local problem (e.g., Eq. (2.3)), which are generally easier to solve numerically.

[3] With the same restrictions regarding charge terms in the solvent

This is why nonlocal electrostatics essentially remained an infeasible application for real-world biological systems until the mid-2000s, when it was shown that the nonlocal cavity model in combination with the Lorentzian dielectric response model (Eqs. 2.4 and 2.5) can be reduced to a system of partial differential equations as well [55]. Since then, the same standard techniques have been also applied to nonlocal formulations and confirmed the gains in accuracy as compared to the local approach [56, 57, 58, 59, 60, 61].

All of the presented methods naturally come with their own pros and cons. For example, the FDM replaces the differential equations by finite difference equations on a spatial grid. In order to represent the molecular boundary in sufficient detail, high-resolution grids are usually employed, which also need to account for a large patch of the solvent domain around the protein, leading to large grids. In addition, initial values for the boundary of the grid need to be provided, leaving another margin for errors. Both BEM and FEM are based on the *method of weighted residuals* and do not work on a spatial grid but rather utilize a less regular domain discretization by means of simple geometric objects.

As so-called *Galerkin methods*, both BEM and FEM rely on a weak formulation of the original equations and translate them into a discrete problem[4]. The FEM requires initial values for the outer boundary (like the FDM) and needs a mesh for both solvent and protein domains (e.g., using three-dimensional simplices such as tetrahedra). The BEM, on the other hand, exploits the fundamental solution of the involved differential operator to eliminate volume terms from the weak formulation. As a result, the original system can be represented through the remaining boundary integrals, which require only a discretization of the surface domain (e.g., as a union of two-dimensional simplices).

Of the presented methods, the BEM is particularly attractive for the protein electrostatics problem, because potentials (and derived quantities) can be easily computed for any point in space. While the method's hard requirement on a fundamental solution, which is not necessarily known, is a severe restriction for general problems, BEM formulations for the protein electrostatics problem already exist. Unfortunately, one obstacle for a feasible application of the BEM still remains and lays the foundation for this manuscript: Unlike FDM and FEM system matrices, which are generally sparse and can thus be represented in a space-efficient way, the ones involved in the BEM context are dense and asymmetric. What is more, the dimensions of the matrices directly depend on the number of simplices used for the corresponding surface mesh, resulting in matrices that often represent several terabytes of data.

[4] See, e.g., [62] for a more detailed introduction to both methods.

Different approaches have been suggested in the past to approximate the system matrix in some way or another, e.g., by means of Wavelets or hierarchical cluster decompositions (cf. [49] for a brief discussion). Here, we instead aim at an *exact* representation of the system matrices that is still efficient enough to support a fast solving of the nonlocal BEM systems.

3 BEM system analysis

Before we can start implementing efficient BEM solvers for the nonlocal protein electrostatics problem, we need to analyze the corresponding linear systems of equations. Notwithstanding the inherent specificity of these analyses in this first step, we believe that many of the recurring patterns and components in the concrete BEM systems outlined in this chapter also appear in more general BEM formulations. If nothing else, BEM systems usually share the problem of large, asymmetric, and fully-populated matrices. Owing to their two-dimensional nature, these matrices require a potentially large amount of memory when represented explicitly, possibly exceeding available resources by multiple orders of magnitude. The memory footprint of BEM matrices can be substantially reduced by means of implicit matrix representations, where the information contained in the matrix is stored in a reduced form such that the explicit representation can be fully or partially recomputed on demand. Since not all matrices allow for suitable implicit representations, a thorough analysis of the BEM systems at hand is required in order to establish a general understanding of their structure. This will enable us to develop said representations in a later chapter and exploit the synergy effects from the integration of optimized solvers.

In this chapter, we first introduce the operators used in the continuous formulations of the presented BEM systems. We mostly work with their discretized counterparts for the derivation of implicit representations and the solver implementations but we still present both continuous and discrete versions in order to rearrange and transform the original formulations. This will be of particular importance for the representation formulas that allow us to compute potentials other than on the molecular surface, or any other derived quantities. However, we limit the level of detail to the required minimum and refer the interested reader to the corresponding original sources for the full derivations and definitions.

3.1 Background: Operators

The operators used in the continuous formulations for the local and nonlocal cavity models can be roughly divided into two categories: trace operators and boundary integral operators. In this section, we give a brief and mostly hand-waving

introduction to each of these operator types to facilitate a basic understanding of the presented systems. A more detailed introduction into the trace and boundary integral operators in the general context of boundary element methods or for a close link to the BEM systems presented here can be found in [63] or [49], respectively.

3.1.1 Trace operators

The formulations in this chapter involve different functions defined inside the protein domain Ω or solvent domain Σ. The trace operators restrict these functions to the boundary Γ, such that they can be incorporated into the boundary integral equations. Depending on the situation, the operators take the form of the *Dirichlet trace operators* γ_0^{int} and γ_0^{ext} or the *Neumann trace operators* γ_1^{int} and γ_1^{ext}, briefly introduced in the following paragraphs. The superscripts *int* and *ext* indicate here whether the *internal* or *external* variant of the respective operator is referred to.

The internal Dirichlet trace operator γ_0^{int} represents the limit of a function $f(\boldsymbol{\xi})$ with $\boldsymbol{\xi} \in \Gamma$, evaluated at a position $\tilde{\boldsymbol{\xi}}$ in the protein domain close to the boundary as $\tilde{\boldsymbol{\xi}} \to \boldsymbol{\xi}$. The internal Neumann trace operator γ_1^{int} takes the same limit of the directional derivative of f along $\hat{\boldsymbol{n}}_{\boldsymbol{\xi}}$, the outward-facing unit normal vector with respect to $\boldsymbol{\xi}$:

$$\gamma_0^{\text{int}} f(\boldsymbol{\xi}) := \lim_{\Omega \ni \tilde{\boldsymbol{\xi}} \to \boldsymbol{\xi} \in \Gamma} f(\tilde{\boldsymbol{\xi}}), \qquad \boldsymbol{\xi} \in \Gamma$$

$$\gamma_1^{\text{int}} f(\boldsymbol{\xi}) := \lim_{\Omega \ni \tilde{\boldsymbol{\xi}} \to \boldsymbol{\xi} \in \Gamma} \frac{\partial}{\partial \hat{\boldsymbol{n}}_{\boldsymbol{\xi}}} f(\tilde{\boldsymbol{\xi}}). \qquad \boldsymbol{\xi} \in \Gamma$$

Analogously, the external Dirichlet and Neumann trace operators γ_0^{ext} and γ_1^{ext} represent the limit of a function $f(\boldsymbol{\xi})$ or its directional derivative along $\hat{\boldsymbol{n}}_{\boldsymbol{\xi}}$, respectively, evaluated at a position $\tilde{\boldsymbol{\xi}}$ in the solvent domain close to the boundary as $\tilde{\boldsymbol{\xi}} \to \boldsymbol{\xi}$:

$$\gamma_0^{\text{ext}} f(\boldsymbol{\xi}) := \lim_{\Sigma \ni \tilde{\boldsymbol{\xi}} \to \boldsymbol{\xi} \in \Gamma} f(\tilde{\boldsymbol{\xi}}), \qquad \boldsymbol{\xi} \in \Gamma$$

$$\gamma_1^{\text{ext}} f(\boldsymbol{\xi}) := \lim_{\Sigma \ni \tilde{\boldsymbol{\xi}} \to \boldsymbol{\xi} \in \Gamma} \frac{\partial}{\partial \hat{\boldsymbol{n}}_{\boldsymbol{\xi}}} f(\tilde{\boldsymbol{\xi}}). \qquad \boldsymbol{\xi} \in \Gamma$$

A visual illustration of the internal and external trace operators is given in Figure 3.1. While the trace operator notation is required for a correct transformation of the *continuous* BEM formulations, it will be handled implicitly in the *discretized* formulations and, for all intents and purposes, disappear.

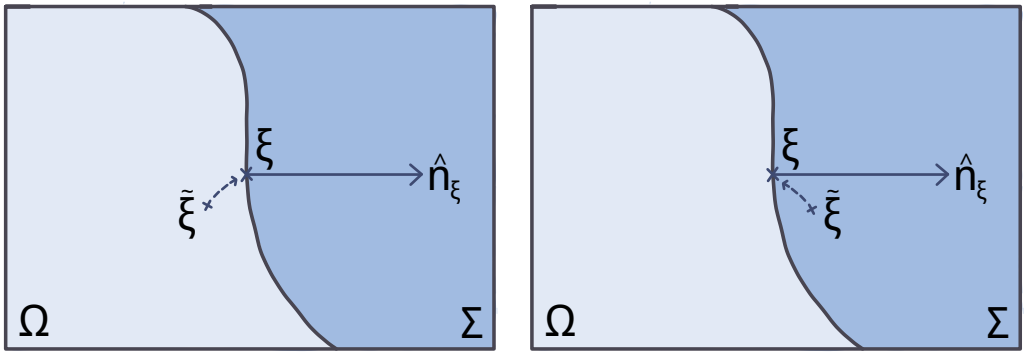


Figure 3.1. – Schematic illustration of the internal (left) and external (right) trace operators for a point $\boldsymbol{\xi}$ at the molecular boundary Γ, its outward-facing unit normal vector $\hat{\boldsymbol{n}}_{\boldsymbol{\xi}}$, and a point $\tilde{\boldsymbol{\xi}}$ in the protein domain Ω or solvent domain Σ, respectively, approaching $\boldsymbol{\xi}$.

3.1.2 Boundary integral operators

The second category of operators we introduce here are based on boundary integrals. As the name "boundary element method" suggests, these integrals are the main components of the BEM systems. Most operators here represent integrals of the general form

$$\oint_\Gamma \mathcal{F}(\boldsymbol{r},\boldsymbol{\xi}) f(\boldsymbol{r}) \mathrm{d}\Gamma_r$$

for some functions $\mathcal{F}$ and f and where $\mathrm{d}\Gamma_r$ is an infinitesimally small surface element of Γ at position $\boldsymbol{r}$.

The two most prominent boundary integral operators in the derivation of the BEM systems are the *single layer potential operator* $\tilde{V}$ and the *double layer potential operator* W for the Laplace operator Δ:

$$[\tilde{V} f](\boldsymbol{\xi}) := -\oint_\Gamma \left(\gamma^{\mathrm{int}}_{0,\boldsymbol{r}} \mathcal{G}^L(\boldsymbol{r},\boldsymbol{\xi})\right) f(\boldsymbol{r}) \mathrm{d}\Gamma_r, \tag{3.1}$$

$$[W f](\boldsymbol{\xi}) := -\oint_\Gamma \left(\gamma^{\mathrm{int}}_{1,\boldsymbol{r}} \mathcal{G}^L(\boldsymbol{r},\boldsymbol{\xi})\right) f(\boldsymbol{r}) \mathrm{d}\Gamma_r, \tag{3.2}$$

with $\boldsymbol{\xi} \in \Omega \cup \Sigma \equiv \mathbb{R}^3 \setminus \Gamma$. The second subscript in the trace operators denotes that the trace is taken with respect to $\boldsymbol{r}$. Here, $\mathcal{F}$ is replaced by the respective traces of the Laplace operator's fundamental solution

$$\mathcal{G}^L(\boldsymbol{r},\boldsymbol{\xi}) := -\frac{1}{4\pi}\frac{1}{|\boldsymbol{r}-\boldsymbol{\xi}|}.$$

The nonlocal formulations introduce two additional variants of $\tilde{V}$ and W, namely, $\tilde{V}^Y$ and W^Y, where $\mathcal{G}^L$ is replaced by the fundamental solution

$$-\mathcal{G}^Y(\boldsymbol{r},\boldsymbol{\xi}) := -\frac{1}{4\pi}\frac{\mathrm{e}^{-\frac{|\boldsymbol{r}-\boldsymbol{\xi}|}{\lambda}}}{|\boldsymbol{r}-\boldsymbol{\xi}|}$$

of the Yukawa operator

$$\mathcal{L}_\lambda := \Delta - \frac{1}{\lambda^2}.$$

With the boundary integral operations presented above, the observation point $\boldsymbol{\xi}$ needs to reside inside the domains Ω or Σ. While this is sufficient for the representation formulas of the BEM systems (cf. Section 3.4), we also need to account for cases where $\boldsymbol{\xi}$ is located at the boundary Γ. For this purpose, the local and nonlocal BEM formulations further introduce boundary integral operators where $\mathcal{F}$ is fully restricted to the boundary:

$$\begin{aligned}[Vf](\boldsymbol{\xi}) &= \left[\left(\gamma_0^{\text{int}}\tilde{V}\right) f\right](\boldsymbol{\xi}),\\ [Kf](\boldsymbol{\xi}) &= \left[\left(\gamma_0^{\text{int}}W\right) f\right](\boldsymbol{\xi}) + (\sigma(\boldsymbol{\xi})-1)f(\boldsymbol{\xi}),\end{aligned}$$

with $\boldsymbol{\xi} \in \Gamma$ and analogously defined variants V^Y for $\tilde{V}^Y$ and K^Y for W^Y. The definition of the geometric quantity $\sigma(\boldsymbol{\xi})$ can be found in [49, Eqs. (5.43) and (5.44)] and [63] and will henceforth be treated as a constant function:

$$\sigma(\boldsymbol{\xi}) = \frac{1}{2}.$$

Generally, this is valid for almost all $\boldsymbol{\xi} \in \Gamma$ as long as the protein surface is differentiable in the vicinity of $\boldsymbol{\xi}$. For our implementations, we delegate this requirement to the generation of the surface models serving as input for our solvers and assume that the surfaces are always sufficiently smooth.

3.2 Local cavity model

We begin our analysis with the boundary integral equations of the local protein electrostatics problem. Primarily, this enables us to implement both local and nonlocal versions of the solvers later on, with the former one serving as a reference. On top of that, we will see that the local case can be considered a blueprint for optimization ideas that can be transferred to the more complex nonlinear case.

3.2.1 Continuous formulation for the local cavity model

The boundary integral equations for the local cavity model introduced in Eq. (2.3) take the form originally presented in [49, Theorem 5.3.1], hereafter shown slightly rearranged and with explicit trace operators and function arguments for improved notational clarity:

$$\left(1 + \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\right) \sigma(\boldsymbol{\xi}) \left[\gamma_0^{\text{int}} \varphi^*\right](\boldsymbol{\xi}) + \left(\frac{\varepsilon_\Omega}{\varepsilon_\Sigma} - 1\right) \left[K\left(\gamma_0^{\text{int}} \varphi^*\right)\right](\boldsymbol{\xi})$$

$$= \left[K\left(\gamma_0^{\text{int}} \varphi_{\text{mol}}\right)\right](\boldsymbol{\xi}) - \sigma(\boldsymbol{\xi}) \left[\gamma_0^{\text{int}} \varphi_{\text{mol}}\right](\boldsymbol{\xi}) - \frac{\varepsilon_\Omega}{\varepsilon_\Sigma} \left[V\left(\gamma_1^{\text{int}} \varphi_{\text{mol}}\right)\right](\boldsymbol{\xi}), \quad (3.3)$$

$$\left[V\left(\gamma_1^{\text{int}} \varphi^*\right)\right](\boldsymbol{\xi}) = \sigma(\boldsymbol{\xi}) \left[\gamma_0^{\text{int}} \varphi^*\right](\boldsymbol{\xi}) + \left[K\left(\gamma_0^{\text{int}} \varphi^*\right)\right](\boldsymbol{\xi}), \quad (3.4)$$

with $\boldsymbol{\xi} \in \Gamma$. The local dielectric response of the solvent $\varepsilon_\Sigma(\boldsymbol{\xi})$ has been modeled as a spatially-independent constant

$$\varepsilon_\Sigma(\boldsymbol{\xi}) := \varepsilon_\Sigma$$

similar to the dielectric response of the protein ε_Ω for the sake of simplicity. A more complex treatment of the solvent response will be provided for the nonlocal formulation in the next section.

Furthermore, this and the upcoming formulations assume a decomposition of the electrostatic potential φ into a *molecular potential* φ_{mol} and a *reaction field potential* φ^* (cf. [49, Eq. (2.78)]):

$$\varphi(\boldsymbol{\xi}) = \varphi_{\text{mol}}(\boldsymbol{\xi}) + \varphi^*(\boldsymbol{\xi}).$$

The traces of the molecular potentials $\gamma_0^{\text{int}} \varphi_{\text{mol}}$ and $\gamma_1^{\text{int}} \varphi_{\text{mol}}$ on the surface can be computed analytically for the point charge model introduced in Section 2.2 and will be given in their discretized form in the next section, along with the discretized boundary integral operators V and K.

The general approach to the nonlocal problem would be to first solve Eq. (3.3) for $\gamma_0^{\text{int}} \varphi^*$ and then use the result to determine $\gamma_1^{\text{int}} \varphi^*$ via Eq.(3.4). The actual potentials and further derived quantities can finally be computed using the representation formulas from Section 3.4.

3.2.2 Discrete formulation for the local cavity model

While the continuous formulation represents a theoretical framework for the local protein electrostatics problem that can be adapted for different approximations of the boundary operators, we will follow the original publication and base our further analysis on the same discretized systems as first presented in [49, Theorem 5.7.1]:

$$\left\{\left(1+\frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\right)\boldsymbol{\sigma}+\left(\frac{\varepsilon_\Omega}{\varepsilon_\Sigma}-1\right)\boldsymbol{K}\right\}\hat{\boldsymbol{u}}=(\boldsymbol{K}-\boldsymbol{\sigma})\hat{\boldsymbol{u}}_{\text{mol}}-\frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\boldsymbol{V}\hat{\boldsymbol{q}}_{\text{mol}} \tag{3.5}$$

$$\boldsymbol{V}\hat{\boldsymbol{q}}=(\boldsymbol{\sigma}+\boldsymbol{K})\hat{\boldsymbol{u}} \tag{3.6}$$

This formulation assumes a triangulation of the molecular surface, that is, an approximation of the original surface domain Γ

$$\bar{\Gamma}\approx\bigcup_{i=1}^{n}\bar{\tau_i}$$

as an area-covering decomposition into n triangles τ_i.

Further, the formulation has been derived in terms of a standard point collocation method. Using piecewise constant basis functions

$$\varphi_i^0(\boldsymbol{\xi})=\begin{cases}1 & \boldsymbol{\xi}\in\tau_i\\ 0 & else\end{cases} \tag{3.7}$$

for all n surface triangles[1], the Dirichlet and Neumann traces of the reaction field are approximated as

$$\gamma_0^{\text{int}}\varphi^*(\boldsymbol{\xi})\approx\sum_{i=1}^{n}\hat{u}_i\varphi_i^0(\boldsymbol{\xi}), \tag{3.8}$$

$$\gamma_1^{\text{int}}\varphi^*(\boldsymbol{\xi})\approx\sum_{i=1}^{n}\hat{q}_i\varphi_i^0(\boldsymbol{\xi}), \tag{3.9}$$

where $\hat{u}_i$ and $\hat{q}_i$ are the approximation coefficients $\hat{\boldsymbol{u}}:=[\hat{u}_1,\hat{u}_2,\ldots,\hat{u}_n]^T$ and $\hat{\boldsymbol{q}}:=[\hat{q}_1,\hat{q}_2,\ldots,\hat{q}_n]^T$, respectively. Then, the matrix

$$\boldsymbol{\sigma}:=\frac{1}{2}\boldsymbol{I} \tag{3.10}$$

[1] The original formulations support any degree of polynomial approximation. While this changes the actual size of the involved matrices considerably, the asymptotic complexity remains still quadratic.

is a uniformly scaled identity matrix $\boldsymbol{I}$ of size $n \times n$. The approximations of the boundary integral operators V and K take the form of $n \times n$ matrices $\boldsymbol{V}$ and $\boldsymbol{K}$, respectively, with their elements being defined as

$$(\boldsymbol{V})_{ij} := \frac{1}{4\pi} \int_{\tau_j} \frac{1}{|\boldsymbol{r} - \boldsymbol{\xi_i}|} \mathrm{d}\Gamma_r, \tag{3.11}$$

$$(\boldsymbol{K})_{ij} := -\frac{1}{4\pi} \int_{\tau_j} \frac{(\boldsymbol{r} - \boldsymbol{\xi_i}) \cdot \hat{\boldsymbol{n}}_{\boldsymbol{j}}}{|\boldsymbol{r} - \boldsymbol{\xi_i}|^3} \mathrm{d}\Gamma_r. \tag{3.12}$$

Each matrix element represents a pair of triangles (τ_i, τ_j), with $\boldsymbol{\xi_i}$ being the centroid of τ_i and $\hat{\boldsymbol{n}}_{\boldsymbol{j}}$ being the outward-facing unit normal vector of τ_j. These integrals can then be solved analytically by using the triangle decomposition technique described in [64]. In this manuscript, we will usually refer to $\boldsymbol{V}$ and $\boldsymbol{K}$ as *potential matrices*.

Using the same point charge model as in Section 2.2, the discretized traces of the molecular potentials can be computed as vectors $\hat{\boldsymbol{u}}_{\text{mol}}$ and $\hat{\boldsymbol{q}}_{\text{mol}}$ with elements

$$(\hat{\boldsymbol{u}}_{\text{mol}})_j := \frac{e_c}{4\pi\varepsilon_0\varepsilon_\Omega} \sum_{i=1}^{N} \frac{\zeta_i}{|\boldsymbol{\xi_j} - \boldsymbol{r_i}|}, \tag{3.13}$$

$$(\hat{\boldsymbol{q}}_{\text{mol}})_j := -\frac{e_c}{4\pi\varepsilon_0\varepsilon_\Omega} \sum_{i=1}^{N} \zeta_i \frac{(\boldsymbol{\xi_j} - \boldsymbol{r_i}) \cdot \hat{\boldsymbol{n}}_{\boldsymbol{j}}}{|\boldsymbol{\xi_j} - \boldsymbol{r_i}|^3}, \tag{3.14}$$

where $\boldsymbol{\xi_j}$ and $\hat{\boldsymbol{n}}_{\boldsymbol{j}}$ are defined as above. It should be noted that the N point charges are not allowed to be located at the molecular surface, i.e., $\boldsymbol{\xi_j} \neq \boldsymbol{r_i} \forall i, j$. In practice, however, special care should be taken to ensure that this requirement also holds for the triangulated surface[2].

The basic idea to solving the local protein electrostatics problem from the discretized systems remains the same as for its continuous counterpart: solve Eq. (3.5) for the approximation coefficients $\hat{\boldsymbol{u}}$, use the result to solve Eq. (3.6) for $\hat{\boldsymbol{q}}$, and then use $\hat{\boldsymbol{u}}$ and $\hat{\boldsymbol{q}}$ with the representation formulas (Section 3.4) to compute the quantities of interest (e.g., potentials everywhere in space).

Looking back at Eqs. (3.5) and (3.6), we can immediately identify the potential matrices $\boldsymbol{V}$ and $\boldsymbol{K}$ as the central building blocks of both systems and as the main cause of the quadratic memory requirements. Both matrices are indeed fully populated and asymmetric but at the same time each element of the same matrix could, in principle, be computed from the same formula, given the corresponding two surface triangles. Moreover, all information required to fully construct $\boldsymbol{V}$ and $\boldsymbol{K}$ is contained in the list of surface elements plus Eqs. (3.11) and (3.12). Since there is no data

[2] More precisely, all charges must strictly reside inside the triangulated surface.

dependency between individual matrix elements whatsoever, all of them can be computed independently. These two crucial observations are potential foundations for both a memory-efficient, implicit matrix representation and the utilization of hardware-accelerated stream processing techniques.

For now, we assume that such an implicit representation for $\boldsymbol{K}$ and $\boldsymbol{V}$ exists and focus on another observation, namely, that these matrices are primarily involved in matrix-vector products. Of course, the occurrence of such products is by no means surprising for linear systems, but when we rearrange Eqs. (3.5) and (3.6) to

$$\left(1 + \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\right) \boldsymbol{\sigma}\hat{\boldsymbol{u}} + \left(\frac{\varepsilon_\Omega}{\varepsilon_\Sigma} - 1\right) \boldsymbol{K}\hat{\boldsymbol{u}} = \boldsymbol{K}\hat{\boldsymbol{u}}_{\text{mol}} - \boldsymbol{\sigma}\hat{\boldsymbol{u}}_{\text{mol}} - \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\boldsymbol{V}\hat{\boldsymbol{q}}_{\text{mol}},$$
$$\boldsymbol{V}\hat{\boldsymbol{q}} = \boldsymbol{\sigma}\hat{\boldsymbol{u}} + \boldsymbol{K}\hat{\boldsymbol{u}}$$

in order to single out $\boldsymbol{V}$ and $\boldsymbol{K}$ and then further use Eq. (3.10), the definition of $\boldsymbol{\sigma}$, to yield

$$\frac{1}{2}\left(1 + \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\right) \hat{\boldsymbol{u}} + \left(\frac{\varepsilon_\Omega}{\varepsilon_\Sigma} - 1\right) \boldsymbol{K}\hat{\boldsymbol{u}} = \boldsymbol{K}\hat{\boldsymbol{u}}_{\text{mol}} - \frac{1}{2}\hat{\boldsymbol{u}}_{\text{mol}} - \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\boldsymbol{V}\hat{\boldsymbol{q}}_{\text{mol}},$$
$$\boldsymbol{V}\hat{\boldsymbol{q}} = \frac{1}{2}\hat{\boldsymbol{u}} + \boldsymbol{K}\hat{\boldsymbol{u}},$$

we can bring all matrix-vector products in the local systems into the forms $\boldsymbol{V}\boldsymbol{x}$ or $\boldsymbol{K}\boldsymbol{x}$ for some vectors $\boldsymbol{x}$. As a consequence, an efficient matrix-vector product implementation alongside a suitable implicit representation for both $\boldsymbol{V}$ and $\boldsymbol{K}$ might be the key to solving these systems in significantly less than quadratic memory while retaining the full system matrix information.

Any local BEM solver for which this proposition holds has to fulfill at least two requirements: first, it obviously is not allowed to store the system matrix explicitly or modify it at any point in time, and secondly, it must solve the system by means of matrix-vector products. Fortunately, these requirements loosely describe a whole class of well-known solvers, henceforth referred to as *matrix-free* solvers, a few of which we will present in Chapter 5.

Many established implementations of such solvers exist for a variety of programming languages. Hence, it would be highly convenient if we were able to use (and eventually compare) them with the implicit system representations tailored to our systems rather than implementing our own custom solvers. For this purpose, we henceforth assume that all potential linear system solvers comply with the following common interface:

Definition 3.2.1. Minimal interface for linear system solvers

A solver for a linear system of the general form

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

with coefficient matrix (or *system matrix*) $\boldsymbol{A}$, input vector (or *system inputs*) $\boldsymbol{x}$, and output vector (or *system outputs*) $\boldsymbol{b}$ expects both $\boldsymbol{A}$ and $\boldsymbol{b}$ as arguments and computes (or approximates) $\boldsymbol{x}$.

In accordance with the above interface, we try to keep all linear systems in the same general form $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, and, in particular, we try to represent the system matrix as a *single* matrix for as long as possible, in order to facilitate its later use as an argument for the solver. Hence, we demand an implicit single-matrix representation for each system in spite of the previous attempt to categorically single out $\boldsymbol{V}$ and $\boldsymbol{K}$. We consider the system outputs to be a generous exception to this rule. For one, we usually refrain from creating implicit representations for vectors in the context of this work. Thus, we need to compute $\boldsymbol{b}$ before calling the solver either way. In the end, this still enables – and even motivates – the exploitation of the matrix-vector products for $\boldsymbol{V}$ and $\boldsymbol{K}$ that we have described earlier, reusing their respective implicit representation nevertheless.

Taking all of these considerations into account, the rearranged discrete formulation for the local cavity model presents itself in the following form:

$$\left\{\frac{1}{2}\left(1+\frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\right)\boldsymbol{I}+\left(\frac{\varepsilon_\Omega}{\varepsilon_\Sigma}-1\right)\boldsymbol{K}\right\}\hat{\boldsymbol{u}} = \boldsymbol{K}\hat{\boldsymbol{u}}_{\text{mol}} - \frac{1}{2}\hat{\boldsymbol{u}}_{\text{mol}} - \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\boldsymbol{V}\hat{\boldsymbol{q}}_{\text{mol}}, \tag{3.15}$$

$$\boldsymbol{V}\hat{\boldsymbol{q}} = \frac{1}{2}\hat{\boldsymbol{u}} + \boldsymbol{K}\hat{\boldsymbol{u}}. \tag{3.16}$$

3.3 Nonlocal cavity model

The nonlocal BEM system shares multiple similarities with its local counterpart, albeit being considerably more complex. In a first step, we highlight the similarities and introduce the differences for the continuous formulation, before analyzing structural patterns in a second step.

3.3.1 Continuous formulation for the nonlocal cavity model

The continuous BEM formulation for the nonlocal cavity model introduced in Eq. (2.4) and the Lorentzian water model from Eq. (2.5) takes the form of the following boundary integral equations, as originally presented in [1, Theorem 5.4.1]:

$$\begin{bmatrix} 1-\sigma-K^Y & \frac{\varepsilon_\Omega}{\varepsilon_\infty}V^Y - \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}(V^Y - V) & \frac{\varepsilon_\infty}{\varepsilon_\Sigma}(K^Y - K) \\ \sigma + K & -V & 0 \\ 0 & \frac{\varepsilon_\Omega}{\varepsilon_\infty}V & 1-\sigma-K \end{bmatrix} \begin{bmatrix} \gamma_0^{\text{int}}\varphi^* \\ \gamma_1^{\text{int}}\varphi^* \\ \gamma_0^{\text{ext}}\Psi \end{bmatrix} = \begin{bmatrix} \beta \\ 0 \\ 0 \end{bmatrix} \tag{3.17}$$

with β being defined as

$$\begin{aligned} \beta := &- \left(1 - \sigma - K^Y + \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}(K^Y - K)\right)\left[\gamma_0^{\text{int}}\varphi_{\text{mol}}\right] \\ &- \left(\frac{\varepsilon_\Omega}{\varepsilon_\infty}V^Y - \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}(V^Y - V)\right)\left[\gamma_1^{\text{int}}\varphi_{\text{mol}}\right] \end{aligned}$$

and with the dielectric constants of the protein and solvent, ε_Ω and ε_Σ, having the same meanings as before. Analogously, the quantity σ, the traces $\gamma_0^{\text{int}}\varphi^*$, $\gamma_1^{\text{int}}\varphi^*$, $\gamma_0^{\text{int}}\varphi_{\text{mol}}$, and $\gamma_1^{\text{int}}\varphi_{\text{mol}}$ as well as the boundary integral operators V and K remain the same as in the local formulations. Please note that the system is shown here without function arguments $\boldsymbol{\xi} \in \Gamma$.

The Lorentzian model for the nonlocal dielectric response of water further introduces the constant ε_∞ alongside the boundary integral operators V^Y and K^Y. The dielectric constant ε_∞ represents the limit of the dielectric solvent response as the correlation length λ approaches ∞ (cf. Section 2.2.2). After solving Eq. (3.17) for the internal traces $\gamma_0^{\text{int}}\varphi^*$ and $\gamma_1^{\text{int}}\varphi^*$ as well as the newly introduced external trace $\gamma_0^{\text{ext}}\Psi$, we can use the representation formulas from Section 3.4 to compute derived quantities such as further potentials and energies.

3.3.2 Discrete formulation for the nonlocal cavity model

As originally presented in [49, Theorem 6.7.2], the continuous system for the nonlocal cavity model takes the following discretized form:

$$\begin{bmatrix} \boldsymbol{I} - \boldsymbol{\sigma} - \boldsymbol{K^Y} & \frac{\varepsilon_\Omega}{\varepsilon_\infty}\boldsymbol{V^Y} - \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}(\boldsymbol{V^Y} - \boldsymbol{V}) & \frac{\varepsilon_\infty}{\varepsilon_\Sigma}(\boldsymbol{K^Y} - \boldsymbol{K}) \\ \boldsymbol{\sigma} + \boldsymbol{K} & -\boldsymbol{V} & \boldsymbol{0} \\ \boldsymbol{0} & \frac{\varepsilon_\Omega}{\varepsilon_\infty}\boldsymbol{V} & \boldsymbol{I} - \boldsymbol{\sigma} - \boldsymbol{K} \end{bmatrix} \begin{bmatrix} \hat{\boldsymbol{u}} \\ \hat{\boldsymbol{q}} \\ \hat{\boldsymbol{w}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\beta} \\ \boldsymbol{0} \\ \boldsymbol{0} \end{bmatrix} \tag{3.18}$$

where $\boldsymbol{\beta}$ is defined as

$$\boldsymbol{\beta} := -\left(\boldsymbol{I} - \boldsymbol{\sigma} - \boldsymbol{K^Y} + \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}(\boldsymbol{K^Y} - \boldsymbol{K})\right)\hat{\boldsymbol{u}}_{\text{mol}} - \left(\frac{\varepsilon_\Omega}{\varepsilon_\infty}\boldsymbol{V^Y} - \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}(\boldsymbol{V^Y} - \boldsymbol{V})\right)\hat{\boldsymbol{q}}_{\text{mol}}.$$

This formulation assumes the same surface domain decomposition into n triangles τ_i and uses the same point collocation method with constant basis functions as in the local case. On that premise, the $n \times n$ matrices $\boldsymbol{V}$, $\boldsymbol{K}$, and $\boldsymbol{\sigma}$, with their definitions being given in Eqs. (3.11), (3.12), and (3.10), respectively, as well as the discretized traces of the molecular potentials $\hat{\boldsymbol{u}}_{\text{mol}}$ and $\hat{\boldsymbol{q}}_{\text{mol}}$, with their respective definitions being given in Eqs. (3.13) and (3.14), remain the same.

The third input trace $\gamma_0^{\text{ext}}\Psi$ is approximated analogously to $\gamma_0^{\text{int}}\varphi^*$ and $\gamma_1^{\text{int}}\varphi^*$ as

$$\gamma_0^{\text{ext}}\Psi(\boldsymbol{\xi}) \approx \sum_{i=1}^{n} \hat{w}_i \varphi_i^0(\boldsymbol{\xi}),$$

with approximation coefficients $\hat{\boldsymbol{w}} = [\hat{w}_1, \hat{w}_2, \ldots, \hat{w}_n]^T$. The approximations of the new boundary integral operators V^Y and K^Y take the forms of $n \times n$ matrices as well, henceforth referred to as potential matrices $\boldsymbol{V^Y}$ and $\boldsymbol{K^Y}$, respectively, with their elements being defined as

$$\left(\boldsymbol{V}^Y\right)_{ij} := -\frac{1}{4\pi}\int_{\tau_j} \frac{\mathrm{e}^{-\frac{|\boldsymbol{r}-\boldsymbol{\xi_i}|}{\lambda}}}{|\boldsymbol{r}-\boldsymbol{\xi_i}|}\mathrm{d}\Gamma_r \tag{3.19}$$

and

$$\left(\boldsymbol{K}^Y\right)_{ij} := \frac{1}{4\pi}\int_{\tau_i} \frac{\lambda + |\boldsymbol{r}-\boldsymbol{\xi_j}|}{\lambda|\boldsymbol{r}-\boldsymbol{\xi_i}|^2}\mathrm{e}^{-\frac{|\boldsymbol{r}-\boldsymbol{\xi}|}{\lambda}}\frac{(\boldsymbol{r}-\boldsymbol{\xi_i})\cdot\hat{\boldsymbol{n}}_{\boldsymbol{j}}}{|\boldsymbol{r}-\boldsymbol{\xi_i}|}\mathrm{d}\Gamma_r. \tag{3.20}$$

Overall, $\boldsymbol{V^Y}$ and $\boldsymbol{K^Y}$ share many fundamental properties with $\boldsymbol{V}$ and $\boldsymbol{K}$: both matrices are also fully populated and asymmetric and each element represents the triangle pair (τ_i, τ_j) without data dependencies between any two elements whatsoever. Also, the matrices can be fully constructed from the list of surface elements and the respective formula from Eqs. (3.19) and (3.20). For these reasons, we expect to find a similar implicit representation for all potential matrices and efficient implementation options for their matrix-vector products.

In contrast to the potential matrices $\boldsymbol{V}$ and $\boldsymbol{K}$ (cf. Section 3.2.2), we do not implement analytic solutions for the integrals of $\boldsymbol{V^Y}$ and $\boldsymbol{K^Y}$. While they can still be approximated by numerical quadrature, special care should be taken with regard to the singularities at $|\boldsymbol{r}-\boldsymbol{\xi}| = 0$. In order to avoid numeric instabilities, we refrain

from computing the elements of $\boldsymbol{V}^{\boldsymbol{Y}}$ and $\boldsymbol{K}^{\boldsymbol{Y}}$ directly. Instead, we work with the matrices $(\boldsymbol{V}^{\boldsymbol{Y}} - \boldsymbol{V})$ and $(\boldsymbol{K}^{\boldsymbol{Y}} - \boldsymbol{K})$, respectively, as they possess finite limits as $|\boldsymbol{r} - \boldsymbol{\xi}| \to 0$. In addition to that, we use alternating series representations of the integrands when approximating the integrals for small $|\boldsymbol{r} - \boldsymbol{\xi}|$ to avoid cancellations. The matrix elements are then computed in all our solvers by a seven-point quadrature with quadrature points and weights taken from [65].

To reflect these changes in the nonlocal BEM system, we define two new potential matrices $\boldsymbol{V}^{\boldsymbol{R}}$ and $\boldsymbol{K}^{\boldsymbol{R}}$ as

$$\begin{aligned} \boldsymbol{V}^{\boldsymbol{R}} &:= \boldsymbol{V}^{\boldsymbol{Y}} - \boldsymbol{V}, \\ \boldsymbol{K}^{\boldsymbol{R}} &:= \boldsymbol{K}^{\boldsymbol{Y}} - \boldsymbol{K} \end{aligned}$$

with elements

$$\left(\boldsymbol{V}^{\boldsymbol{R}}\right)_{ij} := \boldsymbol{V}^{\boldsymbol{Y}}_{ij} - \boldsymbol{V}_{ij}, \tag{3.21}$$

$$\left(\boldsymbol{K}^{\boldsymbol{R}}\right)_{ij} := \boldsymbol{K}^{\boldsymbol{Y}}_{ij} - \boldsymbol{K}_{ij} \tag{3.22}$$

and reformulate the system in the same way as its local counterpart. More specifically, we single out the matrix-vector products of the potential matrices $\boldsymbol{V}^{\boldsymbol{R}}$, $\boldsymbol{V}$, $\boldsymbol{K}^{\boldsymbol{R}}$, and $\boldsymbol{K}$ and use the definition of $\boldsymbol{\sigma}$ from Eq. (3.10) to rearrange Eq. (3.18) to

$$\begin{bmatrix} \frac{1}{2}\boldsymbol{I} - \boldsymbol{K}^{\boldsymbol{R}} - \boldsymbol{K} & \left(\frac{\varepsilon_\Omega}{\varepsilon_\infty} - \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\right)\boldsymbol{V}^{\boldsymbol{R}} + \frac{\varepsilon_\Omega}{\varepsilon_\infty}\boldsymbol{V} & \frac{\varepsilon_\infty}{\varepsilon_\Sigma}\boldsymbol{K}^{\boldsymbol{R}} \\ \frac{1}{2}\boldsymbol{I} + \boldsymbol{K} & -\boldsymbol{V} & \boldsymbol{0} \\ \boldsymbol{0} & \frac{\varepsilon_\Omega}{\varepsilon_\infty}\boldsymbol{V} & \frac{1}{2}\boldsymbol{I} - \boldsymbol{K} \end{bmatrix} \begin{bmatrix} \hat{\boldsymbol{u}} \\ \hat{\boldsymbol{q}} \\ \hat{\boldsymbol{w}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\beta} \\ \boldsymbol{0} \\ \boldsymbol{0} \end{bmatrix} \tag{3.23}$$

$$\begin{aligned} \boldsymbol{\beta} :=& \boldsymbol{K}\hat{\boldsymbol{u}}_{\text{mol}} + \left(1 - \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\right)\boldsymbol{K}^{\boldsymbol{R}}\hat{\boldsymbol{u}}_{\text{mol}} - \frac{1}{2}\hat{\boldsymbol{u}}_{\text{mol}} \\ & - \frac{\varepsilon_\Omega}{\varepsilon_\infty}\boldsymbol{V}\hat{\boldsymbol{q}}_{\text{mol}} + \left(\frac{\varepsilon_\Omega}{\varepsilon_\Sigma} - \frac{\varepsilon_\Omega}{\varepsilon_\infty}\right)\boldsymbol{V}^{\boldsymbol{R}}\hat{\boldsymbol{q}}_{\text{mol}}. \end{aligned} \tag{3.24}$$

3.4 Derived quantities

The presented systems provide the approximation coefficients $\hat{\boldsymbol{u}}$ and $\hat{\boldsymbol{q}}$ for the reaction field potentials at the molecular boundary, $\gamma_0^{\text{int}}\varphi^*$ and $\gamma_1^{\text{int}}\varphi^*$, respectively. In the nonlocal setting, we additionally obtain coefficients $\hat{\boldsymbol{w}}$ for the function $\gamma_0^{\text{ext}}\Psi$.

These coefficients can be used with the representation formulas to compute the potentials everywhere in space and, consequently, other derived quantities. Here, we will use the computation of reaction field energies for both settings as an example for such derived quantities.

3.4.1 Interior potentials

The electrostatic potential $\varphi_\Omega(\boldsymbol{\xi})$ for any point $\boldsymbol{\xi}$ inside the protein domain can be computed in both local and nonlocal settings through the representation formula for the reaction field [49, Theorem 5.3.1]

$$\varphi^*(\boldsymbol{\xi}) = \left[\tilde{V}\left(\gamma_1^{\text{int}}\varphi^*\right)\right](\boldsymbol{\xi}) - \left[W\left(\gamma_0^{\text{int}}\varphi^*\right)\right](\boldsymbol{\xi}) \qquad \boldsymbol{\xi} \in \Omega$$

and the decomposition

$$\varphi_\Omega(\boldsymbol{\xi}) = \varphi^*(\boldsymbol{\xi}) + \varphi_{\text{mol}}(\boldsymbol{\xi}). \qquad \boldsymbol{\xi} \in \Omega$$

Remembering the approximations for $\gamma_0^{\text{int}}\varphi^*$ and $\gamma_1^{\text{int}}\varphi^*$ in Eqs. (3.8) and (3.9), we can insert them into the definitions of $\tilde{V}$ and W in Eqs. (3.1) and (3.2). For $\tilde{V}$, for example, this would lead to

$$\begin{aligned}\left[\tilde{V}\left(\gamma_1^{\text{int}}\varphi^*\right)\right](\boldsymbol{\xi}) &\approx \left[\tilde{V}\left(\sum_{i=1}^{n}\hat{q}_i\varphi_i^0\right)\right](\boldsymbol{\xi}) \\ &= -\oint_\Gamma \left(\gamma_{0,\boldsymbol{r}}^{\text{int}}\mathcal{G}^L(\boldsymbol{r},\boldsymbol{\xi})\right)\left(\sum_{i=1}^{n}\hat{q}_i\varphi_i^0(\boldsymbol{r})\right)\mathrm{d}\Gamma_r.\end{aligned}$$

Further decomposing the boundary integral into the sum of all individual surface triangles τ_j yields

$$\left[\tilde{V}\left(\gamma_1^{\text{int}}\varphi^*\right)\right](\boldsymbol{\xi}) \approx -\sum_{j=1}^{n}\int_{\tau_j}\left(\gamma_{0,\boldsymbol{r}}^{\text{int}}\mathcal{G}^L(\boldsymbol{r},\boldsymbol{\xi})\right)\underbrace{\left(\sum_{i=1}^{n}\hat{q}_i\varphi_i^0(\boldsymbol{r})\right)}_{=\hat{q}_j}\mathrm{d}\Gamma_r.$$

The inner sum provides a nonzero value if and only if $i = j$ owing to the definition of the basis function φ_i^0 in Eq. (3.7). More specifically, the sum reduces to $\hat{q}_j$, thus resulting in

$$\left[\tilde{V}\left(\gamma_1^{\text{int}}\varphi^*\right)\right](\boldsymbol{\xi}) \approx -\sum_{j=1}^{n}\left[\hat{q}_j\int_{\tau_j}\gamma_{0,\boldsymbol{r}}^{\text{int}}\mathcal{G}^L(\boldsymbol{r},\boldsymbol{\xi})\mathrm{d}\Gamma_r\right],$$

which can be rewritten in terms of vector products. For this, we define a row vector $\tilde{\boldsymbol{V}}_\xi$ whose elements represent the surface element integrals

$$\left(\tilde{\boldsymbol{V}}_\xi\right)_j := -\int_{\tau_j} \gamma_{0,\boldsymbol{r}}^{\text{int}} \mathcal{G}^L(\boldsymbol{r},\boldsymbol{\xi}) \mathrm{d}\Gamma_r$$

that can be computed the same way as $\boldsymbol{V}_{ij}$ in Eq. (3.11), except that the load point $\boldsymbol{\xi}$ is now located in the protein domain Ω rather than on its surface.

Performing the same procedure for W and defining a corresponding row vector $\boldsymbol{W}_\xi$ with elements

$$\left(\boldsymbol{W}_\xi\right)_j := -\int_{\tau_j} \gamma_{1,\boldsymbol{r}}^{\text{int}} \mathcal{G}^L(\boldsymbol{r},\boldsymbol{\xi}) \mathrm{d}\Gamma_r$$

that can be computed analogously to $\boldsymbol{K}_{ij}$ in Eq. (3.12), we can finally express the representation formulas for the interior potentials as

$$\begin{aligned} &\varphi^*(\boldsymbol{\xi}) \approx \tilde{\boldsymbol{V}}_\xi \hat{\boldsymbol{q}} - \boldsymbol{W}_\xi \hat{\boldsymbol{u}}, && \boldsymbol{\xi} \in \Omega \\ &\varphi_\Omega(\boldsymbol{\xi}) = \varphi^*(\boldsymbol{\xi}) + \varphi_{\text{mol}}(\boldsymbol{\xi}). && \boldsymbol{\xi} \in \Omega \end{aligned}$$

3.4.2 Exterior potentials

The electrostatic potential $\varphi_\Sigma(\boldsymbol{\xi})$ for a given point $\boldsymbol{\xi}$ in the solvent domain can be computed from separate representation formulas

$$\varphi_\Sigma^{\text{local}}(\boldsymbol{\xi}) = \left[W\left(\gamma_0^{\text{int}}(\varphi^* + \varphi_{\text{mol}})\right)\right](\boldsymbol{\xi}) - \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\left[\tilde{V}\left(\gamma_1^{\text{int}}(\varphi^* + \varphi_{\text{mol}})\right)\right](\boldsymbol{\xi}) \qquad \boldsymbol{\xi} \in \Sigma$$

for the local version, as originally presented in [49, Theorem 5.3.1], and

$$\begin{aligned} \varphi_\Sigma^{\text{nonlocal}}(\boldsymbol{\xi}) =& \frac{\varepsilon_\infty}{\varepsilon_\Sigma}\left(\tilde{V}^Y - \tilde{V}\right)\left[\gamma_1^{\text{ext}}\left(\Psi + \frac{\varepsilon_\Omega}{\varepsilon_\infty}\varphi_{\text{mol}}\right)\right](\boldsymbol{\xi}) \\ &- \frac{\varepsilon_\infty}{\varepsilon_\Sigma}\left(W^Y - W\right)\left[\gamma_0^{\text{ext}}\left(\Psi + \frac{\varepsilon_\Omega}{\varepsilon_\infty}\varphi_{\text{mol}}\right)\right](\boldsymbol{\xi}) \\ &- \tilde{V}^Y\left[\gamma_1^{\text{ext}}\varphi_\Sigma\right](\boldsymbol{\xi}) + W^Y\left[\gamma_0^{\text{ext}}\varphi_\Sigma\right](\boldsymbol{\xi}) \qquad \boldsymbol{\xi} \in \Sigma \end{aligned} \tag{3.25}$$

for the nonlocal version, originally presented in [49, Eq. (5.160)].

Remembering the definitions of $\hat{\boldsymbol{u}}_{\text{mol}}$ and $\hat{\boldsymbol{q}}_{\text{mol}}$ from Eqs. (3.13) and (3.14) and using the same $\tilde{\boldsymbol{V}}_\xi$ and $\boldsymbol{W}_\xi$ as in the last section, we can approximate the local electrostatic potential for a given $\boldsymbol{\xi}$ from the approximation coefficients $\hat{\boldsymbol{u}}$ and $\hat{\boldsymbol{q}}$ via

$$\varphi_\Sigma^{\text{local}}(\boldsymbol{\xi}) \approx \boldsymbol{W}_\xi(\hat{\boldsymbol{u}} + \hat{\boldsymbol{u}}_{\text{mol}}) - \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\tilde{\boldsymbol{V}}_\xi(\hat{\boldsymbol{q}} + \hat{\boldsymbol{q}}_{\text{mol}}). \qquad \boldsymbol{\xi} \in \Sigma$$

Unfortunately, in the nonlocal case we still need to express the representation formula in terms of the quantities $\gamma_0^{\text{int}}\varphi^*$, $\gamma_1^{\text{int}}\varphi^*$, and $\gamma_0^{\text{ext}}\Psi$ we have obtained from solving Eq. (3.17). For this, we first remember the boundary conditions of the nonlocal formulation [49, Eqs. (5.174) and (5.175)]

$$\gamma_0^{\text{ext}}\varphi_\Sigma = \gamma_0^{\text{int}}(\varphi^* + \varphi_{\text{mol}}) \tag{3.26}$$

$$\gamma_1^{\text{ext}}\psi_\Sigma = \varepsilon_0\varepsilon_\Omega\gamma_1^{\text{int}}(\varphi^* + \varphi_{\text{mol}}) \tag{3.27}$$

as well as the approximation for the boundary conditions in [49, Eq. (5.181)]

$$\begin{aligned}\gamma_1^{\text{ext}}\psi_\Sigma &= \varepsilon_0(\varepsilon_\infty\gamma_1^{\text{ext}}\Psi + \varepsilon_\Omega\gamma_1^{\text{ext}}\varphi_{\text{mol}}) \\ &\approx \varepsilon_0\varepsilon_\infty\gamma_1^{\text{ext}}\varphi_\Sigma.\end{aligned} \tag{3.28}$$

Inserting Eq. (3.28) into Eq. (3.27), we find our new (approximate) second boundary conditions as

$$\gamma_1^{\text{ext}}\varphi_\Sigma \approx \frac{\varepsilon_\Omega}{\varepsilon_\infty}\gamma_1^{\text{int}}(\varphi^* + \varphi_{\text{mol}}). \tag{3.29}$$

Further assuming smooth boundary conditions for the molecular potential, that is, $\gamma_0^{\text{ext}}\varphi_{\text{mol}} = \gamma_0^{\text{int}}\varphi_{\text{mol}}$, we can now reformulate Eq. (3.25) – here again shown without function arguments $\boldsymbol{\xi}$ – as suggested above:

$$\begin{aligned}\varphi_\Sigma^{\text{nonlocal}}(\boldsymbol{\xi}) &\overset{(3.26)}{=} \frac{1}{\varepsilon_\Sigma}\left(\tilde{V}^Y - \tilde{V}\right)\left[\varepsilon_\infty\gamma_1^{\text{ext}}\Psi + \varepsilon_\Omega\gamma_1^{\text{ext}}\varphi_{\text{mol}}\right] - \tilde{V}^Y\left[\gamma_1^{\text{ext}}\varphi_\Sigma\right] \\ &\quad - \frac{1}{\varepsilon_\Sigma}\left(W^Y - W\right)\left[\varepsilon_\infty\gamma_0^{\text{ext}}\Psi + \varepsilon_\Omega\gamma_0^{\text{int}}\varphi_{\text{mol}}\right] + W^Y\left[\gamma_0^{\text{int}}(\varphi^* + \varphi_{\text{mol}})\right] \\ &\overset{(3.28)}{\approx} \frac{\varepsilon_\infty}{\varepsilon_\Sigma}\left(\tilde{V}^Y - \tilde{V}\right)\left[\gamma_1^{\text{ext}}\varphi_\Sigma\right] - \tilde{V}^Y\left[\gamma_1^{\text{ext}}\varphi_\Sigma\right] \\ &\quad - \frac{1}{\varepsilon_\Sigma}\left(W^Y - W\right)\left[\varepsilon_\infty\gamma_0^{\text{ext}}\Psi + \varepsilon_\Omega\gamma_0^{\text{int}}\varphi_{\text{mol}}\right] + W^Y\left[\gamma_0^{\text{int}}(\varphi^* + \varphi_{\text{mol}})\right] \\ &\overset{(3.29)}{\approx} \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\left(\tilde{V}^Y - \tilde{V}\right)\left[\gamma_1^{\text{int}}(\varphi^* + \varphi_{\text{mol}})\right] - \frac{\varepsilon_\Omega}{\varepsilon_\infty}\tilde{V}^Y\left[\gamma_1^{\text{int}}(\varphi^* + \varphi_{\text{mol}})\right] \\ &\quad - \frac{1}{\varepsilon_\Sigma}\left(W^Y - W\right)\left[\varepsilon_\infty\gamma_0^{\text{ext}}\Psi + \varepsilon_\Omega\gamma_0^{\text{int}}\varphi_{\text{mol}}\right] + W^Y\left[\gamma_0^{\text{int}}(\varphi^* + \varphi_{\text{mol}})\right]\end{aligned}$$

Using $\hat{\boldsymbol{u}}$, $\hat{\boldsymbol{q}}$, and $\hat{\boldsymbol{w}}$, the discretized molecular potentials $\hat{\boldsymbol{u}}_{\text{mol}}$ and $\hat{\boldsymbol{q}}_{\text{mol}}$, and further defining new row vectors $\tilde{\boldsymbol{V}}_\xi^{\boldsymbol{Y}}$ and $\boldsymbol{W}_\xi^{\boldsymbol{Y}}$ analogously to $\tilde{\boldsymbol{V}}_\xi$ and $\boldsymbol{W}_\xi$, that is,

$$\left(\tilde{\boldsymbol{V}}_\xi^{\boldsymbol{Y}}\right)_j := -\int_{\tau_j} \gamma_{0,\boldsymbol{r}}^{\text{int}} \mathcal{G}^Y(\boldsymbol{r},\boldsymbol{\xi}) \mathrm{d}\Gamma_r,$$
$$\left(\boldsymbol{W}_\xi^{\boldsymbol{Y}}\right)_j := -\int_{\tau_j} \gamma_{1,\boldsymbol{r}}^{\text{int}} \mathcal{G}^Y(\boldsymbol{r},\boldsymbol{\xi}) \mathrm{d}\Gamma_r,$$

which can be computed the same way as $\boldsymbol{V}_{ij}^{\boldsymbol{Y}}$ and $\boldsymbol{K}_{ij}^{\boldsymbol{Y}}$ with new load point $\boldsymbol{\xi}$, we can represent the nonlocal representation formula for the exterior electrostatic potential by the following approximation

$$\begin{aligned}\varphi_\Sigma^{\text{nonlocal}}(\boldsymbol{\xi}) \approx & \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\left(\tilde{\boldsymbol{V}}_\xi^{\boldsymbol{Y}} - \tilde{\boldsymbol{V}}_\xi\right)(\hat{\boldsymbol{q}} + \hat{\boldsymbol{q}}_{\text{mol}}) - \frac{\varepsilon_\Omega}{\varepsilon_\infty}\tilde{\boldsymbol{V}}_\xi^{\boldsymbol{Y}}(\hat{\boldsymbol{q}} + \hat{\boldsymbol{q}}_{\text{mol}}) \\ & - \frac{1}{\varepsilon_\Sigma}\left(\boldsymbol{W}_\xi^{\boldsymbol{Y}} - \boldsymbol{W}_\xi\right)(\varepsilon_\infty \hat{\boldsymbol{w}} + \varepsilon_\Omega \hat{\boldsymbol{u}}_{\text{mol}}) + \boldsymbol{W}_\xi^{\boldsymbol{Y}}(\hat{\boldsymbol{u}} + \hat{\boldsymbol{u}}_{\text{mol}}). \quad \boldsymbol{\xi} \in \Sigma\end{aligned}$$

In order to take the discussion on improved numerical stability of $\boldsymbol{V}^{\boldsymbol{Y}}$ and $\boldsymbol{K}^{\boldsymbol{Y}}$ in Section 3.3.2 into account, we define

$$\tilde{\boldsymbol{V}}_\xi^{\boldsymbol{R}} := \tilde{\boldsymbol{V}}_\xi^{\boldsymbol{Y}} - \tilde{\boldsymbol{V}}_\xi,$$
$$\boldsymbol{W}_\xi^{\boldsymbol{R}} := \boldsymbol{W}_\xi^{\boldsymbol{Y}} - \boldsymbol{W}_\xi$$

and rearrange our approximation of the nonlocal representation formula in such a way that we only need to compute $\tilde{\boldsymbol{V}}_\xi^{\boldsymbol{R}}$ and $\boldsymbol{W}_\xi^{\boldsymbol{R}}$ instead of $\tilde{\boldsymbol{V}}_\xi^{\boldsymbol{Y}}$ and $\boldsymbol{W}_\xi^{\boldsymbol{Y}}$. Then, the final approximations of the local and nonlocal representation formulas for the exterior potential take the following form:

$$\begin{aligned}\varphi_\Sigma^{\text{local}}(\boldsymbol{\xi}) \approx & \boldsymbol{W}_\xi(\hat{\boldsymbol{u}} + \hat{\boldsymbol{u}}_{\text{mol}}) - \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\tilde{\boldsymbol{V}}_\xi(\hat{\boldsymbol{q}} + \hat{\boldsymbol{q}}_{\text{mol}}), \quad \boldsymbol{\xi} \in \Sigma \\ \varphi_\Sigma^{\text{nonlocal}}(\boldsymbol{\xi}) \approx & \left(\frac{\varepsilon_\Omega}{\varepsilon_\Sigma} - \frac{\varepsilon_\Omega}{\varepsilon_\infty}\right)\tilde{\boldsymbol{V}}_\xi^{\boldsymbol{R}}(\hat{\boldsymbol{q}} + \hat{\boldsymbol{q}}_{\text{mol}}) - \frac{\varepsilon_\Omega}{\varepsilon_\infty}\tilde{\boldsymbol{V}}_\xi(\hat{\boldsymbol{q}} + \hat{\boldsymbol{q}}_{\text{mol}}) \\ & + \boldsymbol{W}_\xi^{\boldsymbol{R}}\left(\hat{\boldsymbol{u}} - \frac{\varepsilon_\infty}{\varepsilon_\Sigma}\hat{\boldsymbol{w}} + \left(1 - \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\right)\hat{\boldsymbol{u}}_{\text{mol}}\right) + \boldsymbol{W}_\xi(\hat{\boldsymbol{u}} + \hat{\boldsymbol{u}}_{\text{mol}}). \quad \boldsymbol{\xi} \in \Sigma\end{aligned}$$

3.4.3 Reaction field energies

The local or nonlocal reaction field energy for the partial point charge model (cf. Section 2.2) can be computed via [49, Theorem 5.3.1]:

$$W_{\text{ppc}}^{\text{rf}} = \sum_{i=1}^{N} \zeta_i \varphi^*(\boldsymbol{r}_i),$$

where φ^* is the reaction field potential

$$\varphi^*(\boldsymbol{\xi}) = \left[\tilde{V}\left(\gamma_1^{\text{int}}\varphi^*\right)\right](\boldsymbol{\xi}) - \left[W\left(\gamma_0^{\text{int}}\varphi^*\right)\right](\boldsymbol{\xi}).$$

Using the previous row vector definitions of $\tilde{\boldsymbol{V}}_\xi$ and $\boldsymbol{W}_\xi$ and replacing $\boldsymbol{\xi}$ by the position of the individual point charge $\boldsymbol{r}_i$, the reaction field energy can easily be approximated as

$$W_{\text{ppc}}^{\text{rf}} \approx \sum_{i=1}^{N} \zeta_i \left\{\tilde{\boldsymbol{V}}_{r_i}\hat{\boldsymbol{q}} - \boldsymbol{W}_{r_i}\hat{\boldsymbol{u}}\right\}.$$

4 Platform selection

In the previous chapter, we analyzed the discrete formulations of the local and nonlocal BEM systems for the protein electrostatics problem and identified multiple potential targets for optimization. We have seen that it ultimately comes down to solving large linear systems, where the involved matrices are composed of a distinct set of other matrices that follow a common pattern: Their elements can be computed from the same functions or, from a programmer's point of view, from the same routines with varying parameters. Since the described systems typically contain billions of nonzero matrix elements, their solving process can be considered highly time-consuming even under the best circumstances. As a result, we require any programming language or platform for our BEM solvers to enable high-performance code. The embarrassingly parallel nature of the aforementioned matrix element computation and, to some extend, matrix-vector products for the same matrices, present a promising basis for hardware-accelerated solutions. GPUs are known to handle such embarrassingly parallel computation tasks very well, so a suitable candidate platform is also required to support GPU acceleration.

While an efficient computation of the matrix elements is absolutely essential for the viability of our BEM solvers, the latter should remain accessible to the bioinformatics community. Many people in the field do not have a programming background and should still be able to use them. Apart from this, the currently available bioinformatics software is at least equally heterogeneous and our solvers should be easy to incorporate into existing pipelines and workflows. In the following, we present the platforms we have chosen as a basis for the work presented in this manuscript and elaborate on to what extent they comply with the above requirements.

4.1 The Julia language

The first programming language we consider as a suitable platform for the implementation of protein electrostatics solvers in this manuscript is Julia [1]. In addition to being open source and cross-platform, the Julia language represents a perfect

symbiosis of a high-performance computing language and a highly accessible programming framework. In terms of computational efficiency, Julia has been shown to be comparable to high-performance languages such as C and Fortran as well as LuaJIT, Rust, and Go [1, 66]. At the same time, Julia provides the "look and feel" of a scripting language like Python, R, or MATLAB, which often trade computational efficiency for a positive user experience (e.g., in the form of high-level machine language abstractions). These features alongside a plethora of third-party libraries and a broad language interoperability render Julia a well-suited programming language for the heterogeneous tasks and audience in the field of scientific computing. As such, the language is used in a growing number of studies, with research areas including RNA [67, 68] and protein bioinformatics [69, 70], metabolic networks [71, 72], and fluid dynamics [73, 74]. In this manuscript, we use Julia to implement numerical solvers not only as a prototypical proof of concept but rather as an efficient and intuitive software package for the nonlocal protein electrostatics problem.

In the following, we briefly summarize core features of the language that contributed towards the decision of choosing Julia for our project. A more detailed elaboration on this matter, including a discussion on how to transfer the presented concepts to other scientific software, can be found in our previous publication [75].

4.1.1 Scripting language characteristics

Julia's scripting language character is rooted in a combination of two outstanding language features, namely, a dynamic type system and *just-in-time* (JIT) code compilation. The type system is complemented with fully optional type annotations and a powerful type inference algorithm [76, 77, 78]. Together, it allows for static type checks and code optimizations where needed and a maximum of notational brevity otherwise. These checks and optimizations are performed by Julia's JIT compiler, which is automatically invoked on code execution and, in a sense, makes Julia a compiled language in the guise of an interpreted language. In particular, the on-demand compilation approach enables users to provide source code either directly through the language's interactive console (called *read-eval-print loop* or *REPL*) or in the form of script files, without prior compilation.

Julia ships with its own package management system, which grants access to thousands of third-party packages[1] that extend the language's standard library and can dynamically be included in user code. Functions in Julia can be easily overloaded for custom types [79, 80] due to the implementation of *symmetric multiple dispatch* [81],

[1] `https://juliahub.com`

Listing 4.1 – Code example for the computation of nonlocal reaction field energies using the NESSie.jl and CuNESSie.jl packages in Julia

```
# 0. Import required packages
using NESSie
using NESSie.BEM   # or CuNESSie
using NESSie.Format: readoff, readpqr

# 1. Create a system model
model           = readoff("data/born/na.off")
model.charges   = readpqr("data/born/na.pqr")
model.params.εΩ = 1   # dielectric constant for vacuum model
model.params.εΣ = 78  # dielectric constant for water

# 2. Invoke a BEM solver for the model
bem = solve(NonlocalES, model)

# 3. Apply a post-processor
println("Reaction field energy: ", rfenergy(bem), " kJ/mol")
```

that is, method lookup is based on the runtime types of all function arguments[2]. In conjunction with Julia's native Unicode support, multiple dispatch encourages clean and descriptive function interfaces, further promoting Julia's role as a modern language for rapid prototyping and full-fledged software development alike.

All language features described above can be found in the few lines of code shown in Listing 4.1, representing a fully functional usage example for NESSie.jl and CuNESSie.jl, the Julia-based solvers developed throughout Chapters 5, 6, and 7. The code can be provided either as a script file or fed line-wise to the Julia REPL. Without going into too much detail just yet, the code computes and prints the reaction field energy (cf. Section 3.4.3) for the nonlocal cavity model (cf. Section 3.3) associated with the biomolecular system represented by the `model` variable. For this, the `NESSie` package is imported, providing the functionality required to represent the system model and perform the necessary computations. The solvers are based on other external libraries, e.g., to enable hardware acceleration (cf. Section 4.3.2). System constants (`model.params`) are closely named after their original counterparts using Greek letters, courtesy of Julia's Unicode and LaTeX math symbol support.

While the shown code does not contain any type annotation whatsoever, all involved types can be unambiguously inferred by the JIT compiler. This is because the required type annotations are here fully contained in the function interface of our packages[3], which allow their end-users to omit explicit annotations in most cases and to use the packages in a scripting-style manner without loss of performance. Finally, multiple dispatch is used to choose a suitable solver for the given locality assumption (i.e., `NonlocalES` or `LocalES` for the nonlocal or local cavity models, respectively) and

[2] "Symmetric" here means that all function arguments contribute equally to the lookup process.

Listing 4.2 – Differences in runtime and memory allocations for repeated same-argument function calls in Julia. The compilation overhead is visible only for the first call to the function, while subsequent calls reuse the compiled code from cache.

```
julia> @time bem = solve(NonlocalES, model);
  2.677756 seconds (11.63 M allocations: 889.775 MiB, 6.65% gc time)

julia> @time bem = solve(NonlocalES, model);
  1.096125 seconds (6.29 M allocations: 630.100 MiB, 1.86% gc time)

julia> @time bem = solve(NonlocalES, model);
  1.073325 seconds (6.29 M allocations: 630.100 MiB, 1.16% gc time)
```

system model, which might generally be represented using either single-precision or double-precision floating-point values. As a bonus, GPU acceleration for the chosen solver and post-processors can be optionally enabled by importing `CuNESSie` instead of `NESSie.BEM` in Listing 4.1, as both packages provide the same function interface for these purposes.

4.1.2 High-performance capabilities

The key to Julia's high-performance capabilities is a combination of function specialization and inlining as well as type inference and object unboxing. Generally, most optimizations are primarily enabled through static code analyses, which is a challenging task for dynamic programming languages. Julia addresses this challenge by a multi-step JIT compilation approach [79], where code is translated into a Julia-specific *intermediate representation* (IR), before these optimizations are applied. Afterwards, the representation is translated into LLVM IR [82], where it benefits from further low-level optimizations. Julia's JIT compiler is invoked at runtime every time a function is called for a new set of argument types. Subsequent calls then reuse the optimized code from cache, resulting in different runtimes (and memory allocation behavior) for the first call vs. subsequent calls to the same function with the same arguments. This behavior can, for example, be observed in Listing 4.2, where the BEM solver from Listing 4.1 is applied multiple times to the same model.

Since Julia's JIT compiler is invoked at runtime, it can fully exploit runtime type information to aggressively specialize code. During function specialization, function calls are replaced by the most specific overloads for the given arguments. Specificity

[3] To be more precise, the type annotations in the function argument lists are only one part of the requirements. In addition, the functions in NESSie and CuNESSie are type stable with respect to their arguments, such that the compiler can infer the return types of the function calls based on the argument types. As a consequence, all types in the given code example ultimately depend on the arguments to the `readoff` function.

Listing 4.3 – Using Julia's `@Threads.threads` macro to distribute loop iterations uniformly over a number of threads (here: five, specified when starting the Julia session, thus not shown here). Only the iterations assigned to the same thread are executed sequentially.

```
julia> @Threads.threads for i in 1:10
           println("Thread ", Threads.threadid(), ", iteration ", i)
       end
Thread 5, iteration 9
Thread 3, iteration 5
Thread 4, iteration 7
Thread 5, iteration 10
Thread 3, iteration 6
Thread 2, iteration 3
Thread 4, iteration 8
Thread 1, iteration 1
Thread 2, iteration 4
Thread 1, iteration 2
```

is here determined with respect to Julia's unique subtyping relation [80], which can be used to define nominal type hierarchies. Besides explicit type annotations, the multiple dispatch implementation serves as one of the compiler's main sources for runtime type information. Given the available information, the compiler assigns suitable types to all expressions and variables in the code. As long as code is type stable, the compiler makes extensive use of function inlining to benefit from its synergies with specialization and type inference. Inlining provides additional context for inference and can even be used to eliminate unreachable code branches in specialized functions [79]. Conversely, type instability is infectious, as it introduces type uncertainty to dependent code and thus impairs optimization.

Apart from the above language feature combination, Julia also supports more general components of high-performance computing, namely, concurrent and parallel computing. Using Julia's standard library, users can choose from a variety of different options, including coroutines for the definition and scheduling of asynchronous tasks, a custom message passing implementation for distributed computing, and multithreading macros. The latter resemble optional code annotations with `parallel` pragmas from OpenMP [83] or OpenACC [84] and direct Julia to automatically split and synchronize the given `for` loop uniformly over multiple threads of execution in a minimally invasive way (see Listing 4.3 for an example). Furthermore, third-party packages can serve as an alternative for or as an extension of these standard library features. Prominent examples include the packages provided by the *JuliaParallel* project[4], providing MPI [85] wrappers and distributed data containers, among other things. Another example are packages for GPU-enabled computing, which we will revisit later in Section 4.3.2.

[4] `https://github.com/JuliaParallel`

4.2 AnyDSL

The second candidate platform for our protein electrostatics solvers is *AnyDSL* [2], a programming framework for high-performance libraries by means of *partial evaluation* (PE). AnyDSL is built around a minimalistic IR called *Thorin* [86] and provides a powerful online partial evaluator as well as the *Impala* language as its front end. Thorin is constructed in a way that allows shallow embedding of *domain-specific languages* (DSL) [87], which enables users to write efficient code in a unified programming framework. The code is tailored to a selected target platform during compilation, including different CPU and GPU back ends. While the AnyDSL community appears rather small compared to "big players" like C and C++, the framework's high-performance capabilities were already demonstrated for multiple application domains, e.g., image processing, ray tracing, or genome sequence analysis [88, 89, 90, 91].

In this manuscript, we use the AnyDSL framework – and Impala in particular – to implement domain-specific operations and system representations for the local and nonlocal protein electrostatics problem (cf. Chapters 5 and 8). While the framework is significantly less accessible than Julia, it bears the potential for a low-maintenance code base in spite of supporting multiple platforms. We consider the native domain specificity aspect of Impala to be a huge advantage for the development of our solvers, as compared, for example, to Julia or a C/C++-based approach. It allows us to easily switch between different CPU or GPU back ends and evaluate native and specialized solvers for the supported platforms. Due to Impala's close interconnection to C and C++, we can opt for a hybrid library implementation, consisting of a C++ layer for common and less performance-critical tasks (like the input data processing) and an Impala-based solver optimized through PE. In this way, we can first try to identify suitable platforms for the protein electrostatics problem. If need be (e.g., in the case of a single most beneficial platform), we can subsequently implement a native C++ counterpart of the best-performing solvers while fully reusing the existing C++ part of the code base.

4.2.1 Compilation flow

The AnyDSL framework heavily relies on partial evaluation, that is, a given program is partially evaluated with respect to all inputs known statically (i.e., at compile time) before being fully evaluated for all dynamic inputs during execution [92, 93]. This approach allows for highly specialized programs computed from a common

Listing 4.4. – Comparison between direct style (top) and continuation-passing style (bottom) for functions in Impala. Both formulations define the same function to compute the smallest integer larger than or equal to `x/y`.

```
// Direct style
fn ceil_div(x: i32, y: i32) -> i32 {
    if(x % y == 0) {
        return(x/y)
    }
    x / y + 1
}
```

```
// Continuation-passing style
fn ceil_div(x: i32, y: i32, return: fn(i32) -> !) -> ! {
    if(x % y == 0) {
        return(x/y)
    }
    return(x / y + 1)
}
```

code base for given sets of static inputs. AnyDSL ships with a partial evaluator implemented in C++ and built on top of Thorin, which specializes code on the fly, without prior analysis (this is called *online* PE). In order to facilitate PE and render PE results more predictable to the developer, Thorin is minimalistic in design and much less complex than other well-known intermediate representations [2] such as LLVM IR [82].

The general idea behind efficient program code in AnyDSL is to hide platform-specific parts in higher-order functions that are used by the platform-independent code base. When treated as static input, the target-specific mapping is then performed during compilation as part of the PE, resulting in programs specialized directly for hardware accelerators through CUDA (cf. Section 4.3) or OpenCL [94]. Alternatively, the programs can be specialized for different LLVM back ends, including x86, ARM, AMDGPU, and NVVM. For CPU vector code generation, LLVM's *Region Vectorizer* (RV) [95] can be optionally included in the toolchain. For our BEM solvers, we aim at specializations for local vs. nonlocal cavity models and single- vs. double-precision system models. We will see in Chapter 8 how to obtain specialized programs for any combination of choices for these options from a common code base.

4.2.2 Impala

The Impala language serves as a front end to the AnyDSL framework. While programs could as well be written directly in Thorin, Impala adds syntactic sugar to the latter in order to facilitate its use. Most notably, Thorin enforces a *continuation-passing style* (CPS) for its code. In contrast to the classic *direct style*, where functions

return their computed values to the respective caller, functions in CPS pass the same value to a function (called *continuation*) taken as an additional argument. In Thorin, control flow is fully expressed through such continuations. Impala adapts the same CPS but additionally allows classic direct-style functions and loops[5], which are then automatically translated into CPS. Listing 4.4 shows a direct comparison between the same function written using direct style and CPS. It should be noted that `return` is not a keyword of the language but rather the (default) name of the function's continuation parameter, hidden in syntactic sugar when using direct style.

As described in the last section, implementations using the AnyDSL framework are usually split into a generic code base (describing the general algorithm) and implementation details provided through different layers of indirection, e.g., higher-order functions. Generally, this approach enables a programming language to provide certain *algorithmic skeletons* with parallelized implementations and without prior knowledge of the concrete application [96]. Impala offers such skeletons in the form of primitives like `parallel`[6] and `vectorize` or hardware accelerator abstractions (cf. Sections 4.3.3 and 8.3.1).

4.3 CUDA

The last platform we present in this chapter is NVIDIA's *Compute Unified Device Architecture* (CUDA) [3], which we will use for the GPU-accelerated versions of our protein electrostatics solvers. GPUs are an integral part of most modern workstations and have been subjected to massive improvements over the course of the last few decades. As the name suggests, the initial application domain of GPUs (or *graphics processing units*) was that of a coprocessor for manipulating graphics data, with data usually encoded as textures and manipulation tasks written in a shader language. This is in contrast to CPUs, which are specialized for the fast execution of heterogeneous sequences of operations. For this, CPUs typically employ sophisticated data caching and flow control facilities, spread over a moderate number of CPU cores with complex instruction sets. GPUs, on the other hand, offer only restricted instruction sets and flow control but are equipped with hundreds or thousands of threads that allow for a highly parallelized execution of homogeneous data processing tasks.

[5] With corresponding continuations for `break` and `continue`

[6] The `parallel` primitive triggers pragma-like parallel execution of the associated loop over multiple C++ threads or via *Intel Threading Building Blocks* (TBB) [97], while `vectorize` invokes the aforementioned LLVM RV. In contrast to pragmas, `parallel` and `vectorize` are first-class citizens of the language.

Nowadays, GPUs are no longer restricted to their original purpose but rather widely utilized for more general computations, especially in the scientific community. This application is commonly referred to as *general-purpose computation on graphics processing units* (GPGPU) and CUDA facilitates GPGPU by bridging the gap between classic CPU programming techniques and GPUs. Most notably, CUDA provides language extensions for C and C++ along with extensive software libraries and a compiler suite to translate C/C++-like code into GPU-compatible programs.

GPGPU is of particular interest for embarrassingly parallel tasks, which can often fully exploit the highly abundant compute units of GPUs, such as the computation of the potential matrices in the local and nonlocal BEM formulations (cf. Chapter 3). It should be noted that CUDA is only available for NVIDIA GPUs. While vendor-independent GPU programming platforms such as OpenCL [94] or OpenACC [84] exist, we decided to use CUDA in this manuscript due to the broad availability of compatible devices and the excellent CUDA support in both Julia and Impala, which will be described shortly. Before that, we give a brief summary of CUDA's general programming model, which will serve as a basis for the GPU-accelerated implementations later in Chapters 7 and 8. For more detailed introductions into the CUDA platform and GPGPU in general, the reader is referred to [98, 99, 100].

4.3.1 Programming model

The GPU (or *device*) acts as a coprocessor to the CPU (or *host*) rather than a fully autonomous piece of hardware. Hence, a CUDA-enabled program does not run exclusively on the device but on a combination of both host and device. More precisely, the GPU-based parts of the program are invoked explicitly in the form of one or more *kernel* function calls by the driver program residing on the host. Since host and device are associated with physically separated memory, data transfers and memory management have to be performed through the driver program before (and possibly after) the kernel calls. What is more, host and device generally interact asynchronously and need to be synchronized manually.

Aside from suitable synchronization mechanisms, CUDA ships with two key abstractions that need to be taken into account when writing CUDA-enabled code, namely, the hierarchical organization of GPU threads and memory. Each GPU contains a number of *streaming multiprocessors* (SM), which handle the kernel execution. Each SM is a composition of different processing units and schedulers, including dedicated multiprocessor cores for arithmetic operations on integers (INT32) and

single- (FP32) or double-precision (FP64) floating-point values. The concrete numbers of individual processing units vary between devices and strongly influence the high-performance capabilities of the respective devices, especially with regard to a comparison between single- and double-precision versions of the same applications (cf. Sections 9.1, 9.4.3, and 9.5.2).

Each kernel function represents a task to be performed by a single thread of execution, where the total number of threads needs to be specified alongside the kernel call in the driver program. Additionally, threads need to be organized into equally-sized thread *blocks*, which are then distributed automatically across the available SMs. While the kernel is eventually executed for each thread at some point in time, only a portion of them is actually executed in parallel. For this, mutually exclusive groups of 32 threads within the same block follow a mode of operation that is similar to SIMD (*single instruction, multiple data*) [101]. Traditionally, these so-called *warps* share a common program counter and perform the same operation at the same time. Multiple warps are then scheduled to run concurrently, depending on available resources.

In contrast to the original SIMD pattern, CUDA allows warp threads to access memory in a non-contiguous way. Also, CUDA allows instruction masking to enable branching of control flow. This modified pattern is commonly referred to as SIMT (*single instruction, multiple threads*) in the CUDA context. With the release of the Turing device architecture, NVIDIA introduced *Independent Thread Scheduling*, which allows "full concurrency between threads, regardless of warps" [98]. It should be noted that the CUDA-enabled implementations presented in this manuscript support both technologies and can thus be used with pre-Turing devices.

The partitioning of threads into equally-sized blocks can be instructed by the programmer within certain restrictions imposed by the CUDA platform and the chosen device. However, one aspect to be considered for an efficient partitioning scheme is the utilization of memory. Each thread generally has access to its own *local memory* as well as to the *global memory* shared by all threads of the application (even across multiple kernel calls). Apart from that, all threads of the same block can exchange data through a *shared memory*[7]. All of these memory types come with different access and size restrictions, which partly depend on the chosen partitioning. The exact way of which, when, and how memory is accessed by individual threads during kernel execution is one of the main adjusting screws for performance optimization in CUDA and will be the topic of Section 7.3.2.

[7] Each CUDA device additionally provides two read-only memory types (*texture* and *constant* memory) with similar properties as the global memory, which are optimized for application scenarios outside the scope of this manuscript.

4.3.2 CUDA support in Julia

Although CUDA is not directly supported as part of the Julia language, support has been added in the form of Julia packages through efforts of the *JuliaGPU* project [102]. The project aims at maintaining packages for Julia-based GPU programming in general, but CUDA is the only production-ready platform at the time of writing. CUDA support is implemented through a combination of the packages *CUDAnative.jl*[8] and *CUDAdrv.jl*[9] [103, 104], providing a complete infrastructure for compiling and executing Julia kernels on CUDA-enabled devices in addition to providing access to GPU library functions and the low-level CUDA driver API[10].

CUDA kernels are written directly in Julia code, with kernel calls being marked with a special `@cuda` macro. This macro invokes the CUDAnative.jl package's GPU compiler, which reuses the original Julia compiler infrastructure (as described in Section 4.1.2) with GPU-specific optimization options. Native device code is then generated from the LLVM IR using the LLVM [82] NVPTX back end. Since the GPU compiler is invoked at runtime on each first kernel call, it acts as a JIT compiler for GPU code generation. This approach was shown to result in runtimes comparable to statically compiled CUDA C code [104], e.g., for the Rodinia benchmark suite [105]. Julia kernel functions are subject to certain restrictions, which will be discussed later for our GPU-accelerated BEM solvers in Section 7.2.

4.3.3 CUDA support in Impala

As described earlier in Section 4.2, CUDA support in AnyDSL is provided as part of the standard compilation flow through Thorin. Transitively, the full supported feature set can also be used in Impala, such that CUDA-enabled program code can be generated from the same code base as corresponding CPU-based programs. In order to facilitate the implementation of such a generic code base, AnyDSL ships with different platform abstractions through the *AnyDSL runtime library*. These abstractions provide access to platform-specific functions (e.g., arithmetic operations) and hardware devices (e.g., different GPUs) by means of a unified interface. We will come back to these abstractions later in more detail (cf. Section 8.3.1).

[8] `https://github.com/JuliaGPU/CUDAnative.jl`
[9] `https://github.com/JuliaGPU/CUDAdrv.jl`
[10] At the time of writing, CUDAnative.jl and CUDAdrv.jl are in the process of being merged into a single package named *CUDA.jl* (`https://github.com/JuliaGPU/CUDA.jl`). In this manuscript, we still refer to the separate packages.

5 Solving implicit linear systems

In Chapter 3, we have outlined a minimal interface for linear system solvers (Definition 3.2.1) and additionally assumed that the system matrix is never updated in the solving process. We have further assumed that the matrix is only accessed by means of matrix-vector products, coining the term *matrix-free* solvers. In this chapter, we first give a short introduction into common iterative solvers – the basis for most matrix-free solver implementations and gold standard for solving large linear systems numerically. Many of the presented solvers were originally developed with sparse system matrices in mind but can also be applied to the dense and non-symmetric systems that are targeted in this manuscript. For this purpose, we further develop a unified implicit model representation for our local and nonlocal BEM systems in the second part of this chapter and show how to generalize these ideas for similar tasks. In the remaining sections, we provide implementation concepts of the implicit system representations for both Julia and Impala and show how they can be used with the aforementioned solvers.

5.1 Background: Iterative solvers

The history of iterative solvers for linear systems began with a number of different relaxation schemes for an approximation of the solution, which is updated in each iteration while reducing the residual error. These schemes include still well-known variants such as Jacobi and Gauss-Seidel iterations. These variants are nowadays rarely used individually and rather encountered as parts of more complex iterative methods or used as preconditioners for the systems instead. A broad introduction into iterative solvers for large linear systems, including most of the examples presented in this section, can be found in [106].

5.1.1 Deterministic iterative solvers

Let us consider a linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, where $\boldsymbol{A} \in \mathbb{R}^{m\times n}$, $\boldsymbol{x} \in \mathbb{R}^n$, and $\boldsymbol{b} \in \mathbb{R}^m$. Most modern iterative solving techniques are so-called *projection methods* and project

$\boldsymbol{x}$ into a subspace of the solution space $\mathcal{K} \subset \mathbb{R}^n$, referred to as *search subspace*. In each iteration, an approximation $\tilde{\boldsymbol{x}}$ of $\boldsymbol{x}$ is extracted from $\mathcal{K}$ and subjected to a number of orthogonality constraints. More precisely, the residual vector $\boldsymbol{b} - \boldsymbol{A}\tilde{\boldsymbol{x}}$ needs to be orthogonal to another subspace of the solution space $\mathcal{L} \subset \mathbb{R}^n$, called *subspace of constraints*.

In each iteration, an initial guess $\boldsymbol{x}_0$ to the solution is then used to find an approximation $\tilde{\boldsymbol{x}} = \boldsymbol{x}_0 + \boldsymbol{\delta}$ with $\boldsymbol{\delta} \in \mathcal{K}$ such that $\boldsymbol{b} - \boldsymbol{A}\tilde{\boldsymbol{x}} \perp \mathcal{L}$. This process is repeated as long as the convergence criteria are not met, usually with new subspaces $\mathcal{K}$ and $\mathcal{L}$ and with the current approximation $\tilde{\boldsymbol{x}}$ serving as new initial guess $\boldsymbol{x}_0$.

While there exist multiple valid choices for $\mathcal{K}$ and $\mathcal{L}$, one of the most important classes of projection methods utilizes a *Krylov subspace* for $\mathcal{K}$, which is defined as

$$\mathcal{K}_p(\boldsymbol{A}, \boldsymbol{r}_0) := \operatorname{span}\{\boldsymbol{r}_0, \boldsymbol{A}\boldsymbol{r}_0, \boldsymbol{A}^2\boldsymbol{r}_0, \ldots, \boldsymbol{A}^{p-1}\boldsymbol{r}_0\},$$

where $\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$ is the residual vector of the initial guess $\boldsymbol{x}_0$. We cover a few examples of these so-called *Krylov subspace methods* in the following paragraphs and refer the interested reader to [106, 107, 108] for a more detailed coverage.

The popular *conjugate gradient* (CG) algorithm [109] uses the same Krylov subspace for both $\mathcal{K}$ and $\mathcal{L}$ and can be utilized to solve systems with a symmetric and positive-definite matrix $\boldsymbol{A}$. While this constraint cannot be satisfied by the BEM systems in the protein electrostatics problem, there exist less restrictive variants of the original CG formulation capable of handling such systems, e.g., the *bi-orthogonal conjugate gradient* (BiCG) method [110, 111] or its transpose-free variant *BiCGStab* [112].

Another well-known Krylov subspace method is the *generalized minimum residual method* (GMRES) [113] in which the approximation $\tilde{\boldsymbol{x}}$ of the solution minimizes the residual norm $||\boldsymbol{b} - \boldsymbol{A}\tilde{\boldsymbol{x}}||_2$ in the current Krylov subspace. In order to reduce the memory footprint and computational cost of GMRES, the solver can be restarted after a fixed number of iterations. Hence, this so-called *restarted GMRES* variant is often preferred in practice, although it might stagnate when the system matrix is not positive definite [106]. GMRES and its restarted variant are usually used in combination with suitable preconditioners (cf. Section 5.1.3).

To circumvent the restriction to symmetric positive-definite system matrices of many solvers, a non-symmetric linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ is sometimes expressed in terms of its normal equations, that is, the system is brought to the equivalent form

$$\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{x} = \boldsymbol{A}^T\boldsymbol{b}, \tag{5.1}$$

which is symmetric and positive definite by construction. An alternative formulation in terms of normal equations often used is

$$\boldsymbol{A}\boldsymbol{A}^T\boldsymbol{u} = \boldsymbol{b}, \quad \boldsymbol{x} = \boldsymbol{A}^T\boldsymbol{u}. \tag{5.2}$$

Using either of these two representations, a non-symmetric system can now be used with the CG method or any other solver with the same restrictions. In particular, applying CG to Eq. (5.1) is known as the *CGNR* method, while applying it to Eq. (5.2) instead is known as the *CGNE* method [106]. It should be noted that the system matrices in Eqs. (5.1) and (5.2) are generally much worse conditioned than $\boldsymbol{A}$ itself, potentially resulting in slower convergence rates and other numerical problems.

Many iterative solvers based on the normal equations make use of the fact that the products $\boldsymbol{A}^T\boldsymbol{A}$ and $\boldsymbol{A}\boldsymbol{A}^T$ do not need to be computed explicitly. These methods are commonly referred to as *row-projection methods*, as the involved relaxation steps require only single rows of the system matrix to be present at the same time to form the necessary product components. Row-projection methods are considered to be among the first iterative solvers for large non-symmetric systems and date back at least to the first half of the 20th century. Prominent representatives of this solver class include the famous *Kaczmarz* method [114] and *Cimmino's method* [115].

Apart from the examples presented above, many *hybrid* methods exist, aiming at synergies of the individual components. Examples include the *BiCGStab(l)* method [116], which combines CG with l GMRES iterations, and *CGMN* [117, 118], a hybrid of CG and the Kaczmarz method.

5.1.2 Probabilistic and randomized iterative solvers

Over the past decades, different extensions and paradigms were developed for iterative solvers. Recent studies suggest a probabilistic view on some of these solvers, embedding them in a Bayesian framework to output posterior distributions with their mean representing the classical solution. Thus, these so-called *probabilistic linear solvers* provide an alternative interpretation of established solvers such as projection methods and GMRES in particular [119].

Another branch of solver development focuses on breaking the deterministic character of the classic approaches. More specifically, these *randomized iterative methods* introduce randomization to some aspects of the base solver, e.g., by performing

random rather than sequential row projections, proving beneficial in certain situations [120]. Examples include *randomized Kaczmarz* and *randomized block Kaczmarz* methods [121, 122, 123], *randomized coordinate descent* methods [124, 125], or *randomized Newton methods* [126]. Many of these randomized solvers can be generalized into the framework of *stochastic dual ascent* [120].

5.1.3 Preconditioning

A large number of iterative solvers support the use of preconditioners for the linear system. Generally, preconditioners transform the problem $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ into a more suitable form by utilizing a nonsingular matrix that approximates $\boldsymbol{A}$ in some sense. Linear systems can be preconditioned from the left:

$$\left(\boldsymbol{P}_l^{-1}\boldsymbol{A}\right)\boldsymbol{x} = \boldsymbol{P}_l^{-1}\boldsymbol{b}$$

or from the right:

$$\left(\boldsymbol{A}\boldsymbol{P}_r^{-1}\right)\boldsymbol{y} = \boldsymbol{b}, \quad \boldsymbol{x} = \boldsymbol{P}_r^{-1}\boldsymbol{y},$$

where $\boldsymbol{P}_l$ and $\boldsymbol{P}_r$ are the *left* and *right preconditioners*, respectively. Using both at the same time results in the so-called *split preconditioning*:

$$\left(\boldsymbol{P}_l^{-1}\boldsymbol{A}\boldsymbol{P}_r^{-1}\right)\boldsymbol{y} = \boldsymbol{P}_l^{-1}\boldsymbol{b}, \quad \boldsymbol{x} = \boldsymbol{P}_r^{-1}\boldsymbol{y}.$$

In preconditioned form, the system still has the same solution as before, but the solving process might be easier. A survey on preconditioning techniques for large linear systems can, for example, be found in [127, 106].

Which type of preconditioning and preconditioner should be used usually depends both on the solver and the system. Iterative methods based on the normal equations (cf. Section 5.1.1), for example, tend to be more challenging targets [106]. There exist many different types of preconditioners, some of them developed for a particular solution scheme, e.g., boundary element methods [128]. In this manuscript, we primarily outline how custom preconditioners can be supported with the system representations developed in this chapter. However, we will still find that even the simple Jacobi preconditioner, that is, the diagonal of the system matrix, can achieve significant performance gains in certain cases (cf. Chapter 9).

5.1.4 Parallelization schemes

Solving large linear systems is a challenging task, both in terms of memory and computational constraints. While we primarily attend to the first part of the challenge in this chapter, namely, finding efficient representations for linear systems, we must not neglect the computational demands. One important criterion for the selection of a high-performant solving method is whether the method itself or parts of it are suitable for parallelization. In the context of numerical solvers, and especially in the case of iterative methods, typical gateways to parallel computation are matrix and vector operations as well as data-independent loop iterations. Often there is a considerable overlap between these two working points, as many of the latter iterations can be formulated as matrix or vector operations and vice versa. For many of the iterative solvers presented above, there exist equivalent formulations that operate on multiple vectors at the same time. These so-called *block methods* facilitate concurrent solver implementations and improve the utilization of available resources. Block generalizations for Krylov subspace methods, e.g., GMRES, can be found in [106], while recent studies also covered block versions of row-projection methods, such as the Kaczmarz method and its variants [129, 123].

A more widely applicable parallelization approach with respect to the actual solver *implementations*, especially for those already existing, is a closer look at the individual operations. In the particular case of large, *dense* system matrices, the main contributor to the memory and computational demands is usually the very same matrix, rendering operations on it the primary targets of optimization. The most basic operations performed by iterative solvers on the system matrices are matrix-vector products. In addition to products of the form $\boldsymbol{A}\boldsymbol{x}$, the forms $\boldsymbol{A}^T\boldsymbol{x}$, $\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{x}$, or $\boldsymbol{A}\boldsymbol{A}^T\boldsymbol{x}$ are common alternatives and might be either handled implicitly in the formulation (e.g., in typical row-projection methods) or need to be computed explicitly (e.g., in the CGNR method). Although an exclusive optimization focus on particular operations might not generally lead to the most efficient solver implementations, this method allows for the most flexibility when it comes to existing solvers. As we discuss later in this chapter, it even enables us to inject parallelized code into solvers that are completely sequential implementations otherwise.

5.2 Interaction matrices

In Chapter 3, we have pointed out the embarrassingly parallel nature of the potential matrices $\boldsymbol{V}$, $\boldsymbol{K}$, $\boldsymbol{V}^R$, and $\boldsymbol{K}^R$, that is, all matrix elements can be computed

independently. At the same time, these matrices might occupy multiple terabytes of memory when represented explicitly (cf. Section 9.4.1). It should be noted that such (quadratic) memory demands are not specific to the protein electrostatics computations but rather a property of most system matrices in the BEM context, as they tend to be dense and nonsymmetric.

For the potential matrices at hand, every piece of information required to reconstruct them is contained in the list of surface elements. In particular, each row i of the matrices represents the centroid of the i-th surface element whereas each row j represents the whole j-th surface element. The elements of the potential matrices $\boldsymbol{V}$, $\boldsymbol{K}$, $\boldsymbol{V}^R$, and $\boldsymbol{K}^R$ can then be computed for the corresponding pair of triangles from Eqs. (3.11), (3.12), (3.21), and (3.22), respectively.

Definition 5.2.1. Interaction matrix representation

Let $r = \langle r_1, r_2, \ldots, r_m \rangle$ be a sequence of objects $r_i \in R$ and $c = \langle c_1, c_2, \ldots, c_n \rangle$ be a sequence of objects $c_j \in C$. Given a function $f : R \times C \to \mathbb{R}$, we call the triplet $\mathcal{M} := (r, c, f)$ an *interaction matrix representation* of matrix $\boldsymbol{M} \in \mathbb{R}^{m \times n}$ if and only if $(\boldsymbol{M})_{ij} = f(r_i, c_j)$ for all $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$.

As a shorthand notation, we call $\mathcal{M}$ *interaction matrix* for $\boldsymbol{M}$, with elements $(\mathcal{M})_{ij} := (\boldsymbol{M})_{ij}$. We further call r the *row elements*, c the *column elements*, and f the *interaction function* of $\mathcal{M}$, respectively. Similarly, we call R the *row element set* and C the *column element set* of $\mathcal{M}$.

Remark. Generally, the interaction matrix representation by itself does not necessarily reduce the memory footprint as compared to the original matrix. In fact, we can trivially construct an interaction matrix $\mathcal{M} = (r, c, f)$ for any given matrix $\boldsymbol{M} \in \mathbb{R}^{m \times n}$ by choosing $r = \langle 1, 2, \ldots, m \rangle$, $c = \langle 1, 2, \ldots, n \rangle$, and

$$f(i, j) = \boldsymbol{M}_{ij} = \begin{cases} \boldsymbol{M}_{11} & i = 1, j = 1 \\ \vdots & \\ \boldsymbol{M}_{mn} & i = m, j = n. \end{cases}$$

An interaction matrix constructed this way captures all of the represented information explicitly in its interaction function and thus has the same (asymptotic) memory footprint as the original matrix $\boldsymbol{M}$ (i.e., $\mathcal{O}(mn)$). However, in the cases where the interaction function has a sublinear memory requirement, the overall requirement of the interaction matrix representation is reduced to $\mathcal{O}(m + n)$, that is, the combined size of the row and column elements.

Following the description in the beginning of this section, the potential matrices $\boldsymbol{V}$, $\boldsymbol{K}$, $\boldsymbol{V}^R$, and $\boldsymbol{K}^R$ can now be represented as interaction matrices $\mathcal{V}$, $\mathcal{K}$, $\mathcal{V}^R$, and $\mathcal{K}^R$ with Eqs. (3.11), (3.12), (3.21), and (3.22) as respective interaction functions. When implemented as suggested in Sections 3.2.2 and 3.3.2, these functions do not depend on the system size and require only a constant amount of memory. What is more, all four interaction matrices share the exact same row elements, i.e., the position vectors of n triangle centroids, and column elements, i.e., n surface triangles. Assuming that the latter information can be captured in four position vectors per triangle (that is, three triangle nodes and one normal vector), the row and column elements of all potential matrices can be represented using a grand total of five position vectors (or, equivalently, 15 floating-point numbers) per triangle.

While the potential memory savings effect of the interaction matrix representation vanishes for small m and n[1], this is of little concern to us in the context of this work since we mainly work with large square matrices. Furthermore, the implicit representation results involuntarily in a computational overhead when accessing (i.e., recomputing) the matrix elements. Notwithstanding this concern, we will see in Chapter 9 that the reduced memory footprint alone already justifies this approach for the BEM systems and that the computational cost of individual elements can be reduced substantially on hardware accelerators.

5.2.1 Matrix-free system for the local cavity model

In the local case, the discretized linear systems take the form of Eqs. (3.15) and (3.16):

$$\left\{\frac{1}{2}\left(1+\frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\right)\boldsymbol{I}+\left(\frac{\varepsilon_\Omega}{\varepsilon_\Sigma}-1\right)\boldsymbol{K}\right\}\hat{\boldsymbol{u}}=\boldsymbol{K}\hat{\boldsymbol{u}}_{\text{mol}}-\frac{1}{2}\hat{\boldsymbol{u}}_{\text{mol}}-\frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\boldsymbol{V}\hat{\boldsymbol{q}}_{\text{mol}} \tag{5.3}$$

$$\boldsymbol{V}\hat{\boldsymbol{q}}=\frac{1}{2}\hat{\boldsymbol{u}}+\boldsymbol{K}\hat{\boldsymbol{u}} \tag{5.4}$$

Inspecting the right-hand sides of both systems, we can identify the matrix-vector products for $\boldsymbol{V}$ and $\boldsymbol{K}$ as base components. Furthermore, the system matrix in Eq. (5.4) is actually the potential matrix $\boldsymbol{V}$. Therefore, we only need interaction matrices alongside matrix-vector product implementations for $\boldsymbol{V}$ and $\boldsymbol{K}$ to be able to solve the second system in linear space without further ado. For the first linear system we find that the system matrix is a linear combination of $\boldsymbol{K}$ and the identity matrix $\boldsymbol{I}$.

[1] So far, we have also gracefully ignored the impact of binary representation sizes for elements of the sets R, C, and $\mathbb{R}$.

We can easily construct a linear-space interaction matrix for the $n \times n$ identity matrix $\boldsymbol{I}$ by choosing $r = c = \langle 1, 2, \ldots, n \rangle$ as row and column elements and the Kronecker delta δ_{ij} as interaction function

$$f(i,j) = \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & else. \end{cases}$$

However, in order to represent the whole first system matrix as a single interaction matrix, we need means to scale and combine interaction matrices first.

Theorem 5.2.1. Scaling of interaction matrices

Let $\mathcal{M} = (r, c, f)$ be an interaction matrix for $\boldsymbol{M} \in \mathbb{R}^{m \times n}$ and some row and column element sets R and C, respectively. Then, we can define a new interaction function $\tilde{f}_\alpha : R \times C \to \mathbb{R}$ as

$$\tilde{f}_\alpha(r, c) = \alpha f(r, c)$$

for any choice of $\alpha \in \mathbb{R}$ and represent

$$\alpha \boldsymbol{M} =: \widetilde{\boldsymbol{M}}_\alpha$$

as interaction matrix $\widetilde{\mathcal{M}}_\alpha = (r, c, \tilde{f}_\alpha)$.

Proof. Let $r = \langle r_1, r_2, \ldots, r_m \rangle$ and $c = \langle c_1, c_2, \ldots, c_n \rangle$. Each element of $\alpha \boldsymbol{M}$ can then be reconstructed from $\widetilde{\mathcal{M}}_\alpha$ via

$$\left(\widetilde{\mathcal{M}}_\alpha\right)_{ij} = \tilde{f}_\alpha(r_i, c_j) = \alpha f(r_i, c_j) = \alpha \cdot (\boldsymbol{M})_{ij}$$

for all $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$. □

Theorem 5.2.2. Linear combinations of interaction matrices

Let

$$\widetilde{\boldsymbol{M}} = \sum_{k=1}^{l} \alpha_k \boldsymbol{M}_k$$

be a linear combination of matrices $\boldsymbol{M}_k \in \mathbb{R}^{m \times n}$ with interaction matrix representations $\mathcal{M}_k = (r, c, f_k)$ for some row and column element sets R and C, respectively, and coefficients $\alpha_k \in \mathbb{R}$.

Then we can define a new function $\tilde{f} : R \times C \to \mathbb{R}$ as

$$\tilde{f}(r, c) = \sum_{k=1}^{l} \alpha_k f_k(r, c)$$

such that $\widetilde{\boldsymbol{M}}$ can be represented as a single interaction matrix $\widetilde{\mathcal{M}} = (r, c, \tilde{f})$.

Proof. Let $r = \langle r_1, r_2, \ldots, r_m \rangle$ and $c = \langle c_1, c_2, \ldots, c_n \rangle$. Each element of the linear combination can then be reconstructed from $\widetilde{\mathcal{M}}$ as

$$\begin{aligned}
(\widetilde{\mathcal{M}})_{ij} &= \tilde{f}(r_i, c_j) \\
&= \sum_{k=1}^{l} \alpha_k f_k(r_i, c_j) \\
&= \sum_{k=1}^{l} \alpha_k (\boldsymbol{M}_k)_{ij} \\
&= (\widetilde{\boldsymbol{M}})_{ij}
\end{aligned}$$

for all $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$. □

Nevertheless, Theorem 5.2.2 only allows us to express the system matrix of Eq. (5.3) as a linear combination of two interaction matrices if both of them share the same row and column elements. However, we could extend these elements by additional information. For that, let us assume that $\boldsymbol{K}$ is implicitly represented by the interaction matrix

$$\mathcal{K} = (\langle r_1, r_2, \ldots, r_m \rangle, \langle c_1, c_2, \ldots, c_n \rangle, f_K)$$

for some row and column element sets R and C, respectively. Then an alternative but nonetheless valid interaction matrix for $\boldsymbol{I}$ can be found by choosing

$$\begin{aligned}
\tilde{r} &= \langle (1, r_1), (2, r_2), \ldots, (m, r_m) \rangle, \\
\tilde{c} &= \langle (1, c_1), (2, c_2), \ldots, (n, c_n) \rangle
\end{aligned}$$

as row and column elements over the sets $(\mathbb{Z} \times R)$ and $(\mathbb{Z} \times C)$, respectively, and

$$\tilde{f}_I((i, r), (j, c)) = \delta_{ij}$$

as interaction function, with $\tilde{f}_I : (\mathbb{Z} \times R) \times (\mathbb{Z} \times C) \to \mathbb{R}$.

At first glance, we have simply added information to the row and column elements that is completely ignored by the interaction function. But if we now repeat the same process for $\boldsymbol{K}$ and choose the same $\tilde{r}$ and $\tilde{c}$ as above in addition to a new interaction function

$$\tilde{f}_K((i,r),(j,c)) = f_K(r,c),$$

we end up with two interaction matrices $\mathcal{I} = (\tilde{r}, \tilde{c}, \tilde{f}_I)$ for $\boldsymbol{I}$ and $\mathcal{K} = (\tilde{r}, \tilde{c}, \tilde{f}_K)$ for $\boldsymbol{K}$ that can be linearly combined according to Theorem 5.2.2.

Theorem 5.2.3. Harmonization of interaction matrices

Let $\mathcal{A}$ and $\mathcal{B}$ be interaction matrix representations for matrices $\boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{m\times n}$ and

$$\mathcal{A} = \left(\left\langle r_1^A, r_2^A, \ldots, r_m^A \right\rangle, \left\langle c_1^A, c_2^A, \ldots, c_n^A \right\rangle, f_A\right),$$
$$\mathcal{B} = \left(\left\langle r_1^B, r_2^B, \ldots, r_m^B \right\rangle, \left\langle c_1^B, c_2^B, \ldots, c_n^B \right\rangle, f_B\right)$$

for row elements $r_i^A \in R_A$ and $r_i^B \in R_B$ for all $i = 1, 2, \ldots, m$, column elements $c_j^A \in C_A$ and $c_j^B \in C_B$ for all $j = 1, 2, \ldots, n$, and interaction functions $f_A : R_A \times C_A \to \mathbb{R}$ and $f_B : R_B \times C_B \to \mathbb{R}$ for some row element sets R_A and R_B and column element sets C_A, and C_B.

Then we can find common row and column elements for $\boldsymbol{A}$ and $\boldsymbol{B}$ by choosing

$$\tilde{r} = \left\langle (r_1^A, r_1^B), (r_2^A, r_2^B), \ldots, (r_m^A, r_m^B) \right\rangle,$$
$$\tilde{c} = \left\langle (c_1^A, c_1^B), (c_2^A, c_2^B), \ldots, (c_n^A, c_n^B) \right\rangle,$$

and subsequently replace $\mathcal{A}$ and $\mathcal{B}$ by their *harmonized* versions

$$\widetilde{\mathcal{A}} = (\tilde{r}, \tilde{c}, \tilde{f}_A),$$
$$\widetilde{\mathcal{B}} = (\tilde{r}, \tilde{c}, \tilde{f}_B),$$

with $\tilde{f}_A, \tilde{f}_B : (R_A \times R_B) \times (C_A \times C_B) \to \mathbb{R}$ and

$$\tilde{f}_A((r^A, r^B), (c^A, c^B)) = f_A(r^A, c^A),$$
$$\tilde{f}_B((r^A, r^B), (c^A, c^B)) = f_B(r^B, c^B).$$

Proof. Using $\widetilde{\mathcal{A}}$ and $\widetilde{\mathcal{B}}$ we can reconstruct the elements of $\boldsymbol{A}$ and $\boldsymbol{B}$ as

$$(\widetilde{\mathcal{A}})_{ij} = \tilde{f}_A(\tilde{r}_i, \tilde{c}_j) = \tilde{f}_A\left((r_i^A, r_i^B), (c_j^A, c_j^B)\right) = f_A(r_i^A, c_j^A) = (\boldsymbol{A})_{ij},$$
$$(\widetilde{\mathcal{B}})_{ij} = \tilde{f}_B(\tilde{r}_i, \tilde{c}_j) = \tilde{f}_B\left((r_i^A, r_i^B), (c_j^A, c_j^B)\right) = f_B(r_i^B, c_j^B) = (\boldsymbol{B})_{ij}$$

for all $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$. □

Now we can finally represent each system matrix of the local BEM formulation as single interaction matrix and we could start implementing operations on said matrices. Moreover, we can use the presented methods for any linear system where the system matrix can be represented as a linear combination of interaction matrices, as long as their dimensions match. The harmonization process could, in concept, be applied recursively to harmonize an arbitrary number of interaction matrices of the same size. However, every additional matrix adds another set of row and column elements to the representation. While this does not affect the asymptotic memory requirement, it further diminishes the memory savings effect of the implicit representation in practice.

At this point it is worthwhile to discuss whether or not a system matrix representation in the form of a single interaction matrix is generally beneficial. After all, we could attempt to provide custom implicit matrices for each individual linear system. This would of course yield the most flexible optimization options as each representation can be tailored to the particular system and available resources. The interaction matrix approach, on the other hand, represents a unified framework for this kind of system matrices, requires less boilerplate code, and thus implies less error-prone implementations and facilitated maintainability.

Providing a fixed representation for all system matrices might also not be an optimal solution with regard to computational efficiency. This heavily depends on the structure of the systems and operations of interest. In the case of iterative solvers for the BEM systems of the protein electrostatics problem, we can indeed target matrix-vector products for performance optimizations. Let us assume the existence of an abstraction layer for the system matrix in Eq. (5.3), which is not necessarily an interaction matrix but still a *single* matrix. Then we can once again exploit the distributive property of the matrix-vector product and reformulate

$$\left\{\frac{1}{2}\left(1 + \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\right)\boldsymbol{I} + \left(\frac{\varepsilon_\Omega}{\varepsilon_\Sigma} - 1\right)\boldsymbol{K}\right\}\boldsymbol{x} \tag{5.5}$$

as

$$\frac{1}{2}\left(1 + \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\right)\boldsymbol{x} + \left(\frac{\varepsilon_\Omega}{\varepsilon_\Sigma} - 1\right)\boldsymbol{K}\boldsymbol{x} \tag{5.6}$$

for some vector $\boldsymbol{x}$ without losing the desired property of a single system matrix. More specifically, the abstraction layer would encapsulate Eq. (5.6) and present itself as a single matrix. This would eliminate the identity matrix completely in the process. Also, we no longer require additional space for the row and column elements of the identity matrix in this representation and can use the original interaction matrix $\mathcal{K}$ for $\boldsymbol{K}$ without harmonization. Now, we have simplified the previous matrix-vector product such that we can solve the local problem with implicit matrices by providing an implementation for $\boldsymbol{V}\boldsymbol{x}$ and $\boldsymbol{K}\boldsymbol{x}$ – or, more precisely, $\mathcal{V}\boldsymbol{x}$ and $\mathcal{K}\boldsymbol{x}$ – as well as a custom abstraction layer for the decomposition shown in Eq. (5.6). As we will see later in this chapter, this can be elegantly achieved in both Julia and Impala.

Theorem 5.2.4. Matrix-free representation of the discretized local BEM systems

Let $\mathcal{V}\boldsymbol{x}$ and $\mathcal{K}\boldsymbol{x}$ denote the matrix-free versions of the matrix-vector products $\boldsymbol{V}\boldsymbol{x}$ and $\boldsymbol{K}\boldsymbol{x}$ for a given vector $\boldsymbol{x}$, the potential matrices $\boldsymbol{V}$ and $\boldsymbol{K}$ of the discretized local BEM systems in Eqs. (5.3) and (5.4), and their interaction matrix representations $\mathcal{V}$ and $\mathcal{K}$. Then we can express the systems as their matrix-free counterparts in terms of matrix-free products:

$$\frac{1}{2}\left(1+\frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\right)\hat{\boldsymbol{u}}+\left(\frac{\varepsilon_\Omega}{\varepsilon_\Sigma}-1\right)\mathcal{K}\hat{\boldsymbol{u}}=\mathcal{K}\hat{\boldsymbol{u}}_{\text{mol}}-\frac{1}{2}\hat{\boldsymbol{u}}_{\text{mol}}-\frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\mathcal{V}\hat{\boldsymbol{q}}_{\text{mol}},$$
$$\mathcal{V}\hat{\boldsymbol{q}}=\frac{1}{2}\hat{\boldsymbol{u}}+\mathcal{K}\hat{\boldsymbol{u}}.$$

Although we ultimately settle on a representation that is not purely based on interaction matrices as a unifying framework, we still consider it a good compromise of both extremes of representation approaches discussed above. Providing a minimal abstraction layer for the system matrix of Eq. (5.3) allows us both to maintain the general form $\boldsymbol{A}\boldsymbol{x}=\boldsymbol{b}$ of the system and to reduce its matrix-free representation to two distinct and reusable matrix-vector products.

5.2.2 Matrix-free system for the nonlocal cavity model

Let us now have a look at the discretized nonlocal BEM system previously presented in Eqs. (3.23) and (3.24):

$$\begin{bmatrix} \frac{1}{2}\boldsymbol{I}-\boldsymbol{K}^{\boldsymbol{R}}-\boldsymbol{K} & \left(\frac{\varepsilon_\Omega}{\varepsilon_\infty}-\frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\right)\boldsymbol{V}^{\boldsymbol{R}}+\frac{\varepsilon_\Omega}{\varepsilon_\infty}\boldsymbol{V} & \frac{\varepsilon_\infty}{\varepsilon_\Sigma}\boldsymbol{K}^{\boldsymbol{R}} \\ \frac{1}{2}\boldsymbol{I}+\boldsymbol{K} & -\boldsymbol{V} & \boldsymbol{0} \\ \boldsymbol{0} & \frac{\varepsilon_\Omega}{\varepsilon_\infty}\boldsymbol{V} & \frac{1}{2}\boldsymbol{I}-\boldsymbol{K} \end{bmatrix}\begin{bmatrix} \hat{\boldsymbol{u}} \\ \hat{\boldsymbol{q}} \\ \hat{\boldsymbol{w}} \end{bmatrix}=\begin{bmatrix} \boldsymbol{\beta} \\ \boldsymbol{0} \\ \boldsymbol{0} \end{bmatrix} \tag{5.7}$$

where $\boldsymbol{\beta}$ is defined as

$$\boldsymbol{\beta} := \boldsymbol{K}\hat{\boldsymbol{u}}_{\text{mol}} + \left(1 - \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\right)\boldsymbol{K}^{\boldsymbol{R}}\hat{\boldsymbol{u}}_{\text{mol}} - \frac{1}{2}\hat{\boldsymbol{u}}_{\text{mol}} - \frac{\varepsilon_\Omega}{\varepsilon_\infty}\boldsymbol{V}\hat{\boldsymbol{q}}_{\text{mol}} + \left(\frac{\varepsilon_\Omega}{\varepsilon_\Sigma} - \frac{\varepsilon_\Omega}{\varepsilon_\infty}\right)\boldsymbol{V}^{\boldsymbol{R}}\hat{\boldsymbol{q}}_{\text{mol}} \quad (5.8)$$

There is one apparent difference to the local system albeit being composed of similar components: The system matrix is no longer a simple linear combination of interaction matrices but rather a 3×3 block matrix where each component more or less appears to be a linear combination of interaction matrices[2]. By using the techniques from the last section, we can indeed confirm this suspicion for all but two components, namely, the null matrices $\mathbf{0}$ in the second and third row of the matrix. For these null matrices, we can choose a constant interaction function $f(r, c) = 0$ as well as row and column elements of the appropriate dimensions – but apart from that with arbitrary row and column element sets – to obtain corresponding interaction matrices. What is more, using the harmonization technique from Theorem 5.2.3, we can represent each block matrix component with the same row and column elements, motivating an attempt at creating an interaction matrix representation for block matrices.

Theorem 5.2.5. Interaction block matrices

Let

$$\widetilde{\boldsymbol{M}} = \begin{bmatrix} \boldsymbol{M_{11}} & \cdots & \boldsymbol{M_{1q}} \\ \vdots & \ddots & \vdots \\ \boldsymbol{M_{p1}} & \cdots & \boldsymbol{M_{pq}} \end{bmatrix}$$

be a $p \times q$ block matrix with components $\boldsymbol{M_{kl}} \in \mathbb{R}^{m \times n}$, corresponding interaction matrix representations

$$\mathcal{M}_{kl} = (\langle r_1, r_2, \ldots, r_m \rangle, \langle c_1, c_2, \ldots, c_n \rangle, f_{kl}),$$

and row and column element sets R and C, respectively. Then we can define new row elements as a series of length mp by duplicating the original row elements p times and enumerating them as

$$\tilde{r} = \langle (1, r_1), (2, r_2), \ldots, (m, r_m), (m+1, r_1), \ldots, (mp, r_m) \rangle .$$

Similarly, we can define new column elements as a series of length nq by duplicating the original column elements q times and enumerating them as

$$\tilde{c} = \langle (1, c_1), (2, c_2), \ldots, (n, c_n), (n+1, c_1), \ldots, (nq, c_n) \rangle .$$

[2]The local systems could naturally also be represented by a single block matrix with similar properties.

Finally, we define a new interaction function $\tilde{f} : (\mathbb{Z} \times R) \times (\mathbb{Z} \times C) \to \mathbb{R}$ as

$$\tilde{f}((i,r),(j,c)) = \sum_{k=1}^{p} \sum_{l=1}^{q} \delta((k-1)q + l, g_{mnq}(i,j)) f_{kl}(r,c)$$

with $\delta(i,j) = \delta_{ij}$ and

$$g_{mnq} = \left(\left\lceil \frac{i}{n} \right\rceil - 1 \right) q + \left\lceil \frac{j}{m} \right\rceil$$

to represent $\widetilde{\boldsymbol{M}}$ as interaction matrix $\widetilde{\mathcal{M}} = (\tilde{r}, \tilde{c}, \tilde{f})$.

Proof. First, it should be noted that

$$\mathcal{B} = \left(\langle 1, 2, \ldots, mp \rangle, \langle 1, 2, \ldots, nq \rangle, g_{mnq} \right)$$

represents an interaction matrix for a matrix $\boldsymbol{B}$ with the same dimensions as $\widetilde{\boldsymbol{M}}$ and its elements correspond to the linear indices of the individual blocks of $\widetilde{\boldsymbol{M}}$ in row-major order. For $m = n = p = q = 2$, for example, we obtain

$$\widetilde{\boldsymbol{M}} = \begin{bmatrix} \boldsymbol{M_1} & \boldsymbol{M_2} \\ \boldsymbol{M_3} & \boldsymbol{M_4} \end{bmatrix} \qquad \text{and} \qquad \boldsymbol{B} = \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{bmatrix},$$

where the block matrix components have also been indexed linearly to make the connection more apparent. Second, owing to the replication of the original row and column elements, a direct correspondence between the new and the original elements is given via

$$\begin{aligned} \tilde{r}_i &= (i, r_{((i-1) \bmod m)+1}), \\ \tilde{c}_j &= (j, c_{((j-1) \bmod n)+1}) \end{aligned}$$

for all $i = 1, 2, \ldots, mp$ and $j = 1, 2, \ldots, nq$. For the sake of notational simplicity, we also use i and j to refer to the original row elements in this proof:

$$\begin{aligned} r_i &:= r_{((i-1) \bmod m)+1}, \\ c_j &:= c_{((j-1) \bmod n)+1}. \end{aligned}$$

The interaction function $\tilde{f}$ is the sum of all individual interaction functions f_{kl} times a coefficient determined by the Kronecker delta function (i.e., 0 or 1). The first argument to the delta function computes the linear component index from k and l,

while the second argument computes the same index from i and j. By design, there is exactly one original interaction function for which the delta function results in a value of one, namely, the interaction function corresponding to the matrix block (k, l) where the index pair (i, j) is located.

We can confirm this by having a closer look at the interaction function $\tilde{f}$. For any given block matrix $\widetilde{\boldsymbol{M}}$ with the abovementioned dimensions and components $\boldsymbol{M_{kl}}$, the corresponding component for any element $(\widetilde{\boldsymbol{M}})_{ij}$ can be retrieved via

$$k = \left\lceil \frac{i}{n} \right\rceil \quad \wedge \quad l = \left\lceil \frac{j}{m} \right\rceil .$$

When inserting these equations into the definition of g_{mnq}, we find that the sum in $\tilde{f}$ (when ignoring f_{kl} for a moment) reduces to a value of one. Indeed,

$$\begin{aligned}
&\sum_{\tilde{k}=1}^{p} \sum_{\tilde{l}=1}^{q} \delta((\tilde{k}-1)q + \tilde{l}, g_{mnq}(i,j)) \\
&= \sum_{\tilde{k}=1}^{p} \sum_{\tilde{l}=1}^{q} \delta\left((\tilde{k}-1)q + \tilde{l}, \left(\left\lceil \frac{i}{n} \right\rceil - 1\right) q + \left\lceil \frac{j}{m} \right\rceil\right) \\
&= \sum_{\tilde{k}=1}^{p} \sum_{\tilde{l}=1}^{q} \delta\left((\tilde{k}-1)q + \tilde{l}, (k-1)\, q + l\right) \\
&= 1 + \underbrace{\sum_{\substack{\tilde{k}=1 \\ \tilde{k}\neq k}}^{p} \sum_{\substack{\tilde{l}=1 \\ \tilde{l}\neq l}}^{q} \delta\left((\tilde{k}-1)q + \tilde{l}, (k-1)\, q + l\right)}_{=0}
\end{aligned}$$

for any valid index pair (i, j), confirming that only one delta function contributes to the sum. As expected, this is the delta function corresponding to the interaction function f_{kl} of component $\boldsymbol{M}_{kl}$, i.e., the location of index pair (i, j).

As a consequence, we can reconstruct each element of $\widetilde{\boldsymbol{M}}$ from its interaction matrix $\widetilde{\mathcal{M}}$ as

$$(\widetilde{\mathcal{M}})_{ij} = \tilde{f}(\tilde{r}_i, \tilde{c}_j) = f_{kl}(r_i, c_j)$$

for the corresponding f_{kl} and all $i = 1, 2, \ldots, mp$ and $j = 1, 2, \ldots, nq$, which is the element of $\boldsymbol{M}_{kl}$ in row $((i-1) \bmod m) + 1$ and column $((j-1) \bmod n) + 1$. □

The same considerations regarding a restriction on single *interaction matrices* (rather than more generic representations) as in the local case can also be applied here.

Assuming once more that we only require an implementation of a matrix-vector product to solve the nonlocal BEM system, we can write the product

$$\begin{bmatrix} \frac{1}{2}\boldsymbol{I} - \boldsymbol{K}^{\boldsymbol{R}} - \boldsymbol{K} & \left(\frac{\varepsilon_\Omega}{\varepsilon_\infty} - \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\right)\boldsymbol{V}^{\boldsymbol{R}} + \frac{\varepsilon_\Omega}{\varepsilon_\infty}\boldsymbol{V} & \frac{\varepsilon_\infty}{\varepsilon_\Sigma}\boldsymbol{K}^{\boldsymbol{R}} \\ \frac{1}{2}\boldsymbol{I} + \boldsymbol{K} & -\boldsymbol{V} & \boldsymbol{0} \\ \boldsymbol{0} & \frac{\varepsilon_\Omega}{\varepsilon_\infty}\boldsymbol{V} & \frac{1}{2}\boldsymbol{I} - \boldsymbol{K} \end{bmatrix} \begin{bmatrix} \boldsymbol{x_1} \\ \boldsymbol{x_2} \\ \boldsymbol{x_3} \end{bmatrix}$$

for a given block vector $[\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_3}]^T$ as

$$\begin{bmatrix} \boldsymbol{K}^{\boldsymbol{R}}\left(\frac{\varepsilon_\infty}{\varepsilon_\Sigma}\boldsymbol{x}_3 - \boldsymbol{x}_1\right) - \boldsymbol{K}\boldsymbol{x}_1 + \left(\frac{\varepsilon_\Omega}{\varepsilon_\infty} - \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\right)\boldsymbol{V}^{\boldsymbol{R}}\boldsymbol{x}_2 + \frac{\varepsilon_\Omega}{\varepsilon_\infty}\boldsymbol{V}\boldsymbol{x}_2 + \frac{1}{2}\boldsymbol{x}_1 \\ \boldsymbol{K}\boldsymbol{x}_1 - \boldsymbol{V}\boldsymbol{x}_2 + \frac{1}{2}\boldsymbol{x}_1 \\ \frac{\varepsilon_\Omega}{\varepsilon_\infty}\boldsymbol{V}\boldsymbol{x}_2 - \boldsymbol{K}\boldsymbol{x}_3 + \frac{1}{2}\boldsymbol{x}_3 \end{bmatrix}.$$

This representation has two valuable advantages for this particular system: For one, we can eliminate the null matrices and identity matrices from the product altogether, meaning that they neither have to be represented in memory nor taken into account when performing the actual computation. On platforms where the multiplication $\boldsymbol{0x}$ cannot be optimized away, the latter is a huge benefit. Secondly, we can see that some matrix-vector products are used multiple times in different components of the result vector. Here we also benefit from treating the products individually and reusing them whenever possible, especially when the interaction functions are expensive to compute.

Theorem 5.2.6. Matrix-free representation of the discretized nonlocal BEM system

Let $\mathcal{V}\boldsymbol{x}$, $\mathcal{K}\boldsymbol{x}$, $\mathcal{V}^R\boldsymbol{x}$, and $\mathcal{K}^R\boldsymbol{x}$ denote the matrix-free versions of the matrix-vector products $\boldsymbol{V}\boldsymbol{x}$, $\boldsymbol{K}\boldsymbol{x}$, $\boldsymbol{V}^R\boldsymbol{x}$, and $\boldsymbol{K}^R\boldsymbol{x}$ for a given vector $\boldsymbol{x}$, the potential matrices $\boldsymbol{V}$, $\boldsymbol{K}$, $\boldsymbol{V}^R$, and $\boldsymbol{K}^R$ of the discretized nonlocal BEM systems in Eqs. (5.7) and (5.8), and their interaction matrix representations $\mathcal{V}$, $\mathcal{K}$, $\mathcal{V}^R$, and $\mathcal{K}^R$. Then we can express the system in terms of matrix-free products:

$$\begin{bmatrix} \mathcal{K}^R\left(\frac{\varepsilon_\infty}{\varepsilon_\Sigma}\hat{\boldsymbol{w}} - \hat{\boldsymbol{u}}\right) - \mathcal{K}\hat{\boldsymbol{u}} + \left(\frac{\varepsilon_\Omega}{\varepsilon_\infty} - \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\right)\mathcal{V}^R\hat{\boldsymbol{q}} + \frac{\varepsilon_\Omega}{\varepsilon_\infty}\mathcal{V}\hat{\boldsymbol{q}} + \frac{1}{2}\hat{\boldsymbol{u}} \\ \mathcal{K}\hat{\boldsymbol{u}} - \mathcal{V}\hat{\boldsymbol{q}} + \frac{1}{2}\hat{\boldsymbol{u}} \\ \frac{\varepsilon_\Omega}{\varepsilon_\infty}\mathcal{V}\hat{\boldsymbol{q}} - \mathcal{K}\hat{\boldsymbol{w}} + \frac{1}{2}\hat{\boldsymbol{w}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\beta} \\ \boldsymbol{0} \\ \boldsymbol{0} \end{bmatrix},$$

where

$$\boldsymbol{\beta} := \mathcal{K}\hat{\boldsymbol{u}}_{\text{mol}} + \left(1 - \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\right)\mathcal{K}^R\hat{\boldsymbol{u}}_{\text{mol}} - \frac{1}{2}\hat{\boldsymbol{u}}_{\text{mol}} - \frac{\varepsilon_\Omega}{\varepsilon_\infty}\mathcal{V}\hat{\boldsymbol{q}}_{\text{mol}} + \left(\frac{\varepsilon_\Omega}{\varepsilon_\Sigma} - \frac{\varepsilon_\Omega}{\varepsilon_\infty}\right)\mathcal{V}^R\hat{\boldsymbol{q}}_{\text{mol}}.$$

With this optimized representation, we can solve the nonlocal problem by providing matrix-vector product implementations for $\mathcal{V}\boldsymbol{x}$, $\mathcal{K}\boldsymbol{x}$, $\mathcal{V}^R\boldsymbol{x}$, and $\mathcal{K}^R\boldsymbol{x}$, requiring a minimum of five different products computed in total (two for $\mathcal{K}\hat{\boldsymbol{u}}$ and $\mathcal{K}\hat{\boldsymbol{w}}$ and one for each of the other matrices). This requirement can be further reduced by concatenating $\hat{\boldsymbol{u}}$ and $\hat{\boldsymbol{w}}$ horizontally and providing a regular matrix product implementation for $\mathcal{K}$.

5.2.3 Post-processing with implicit representations

In the discretized version of the representation formulas presented in Section 3.4, we have encountered the single and double layer potentials $\tilde{\boldsymbol{V}}_\xi$, $\boldsymbol{W}_\xi$, $\tilde{\boldsymbol{V}}_\xi^R$, and $\boldsymbol{W}_\xi^R$ for a given load point $\boldsymbol{\xi}$:

$$
\begin{aligned}
\varphi_\Omega(\boldsymbol{\xi}) \approx& \tilde{\boldsymbol{V}}_\xi\hat{\boldsymbol{q}} - \boldsymbol{W}_\xi\hat{\boldsymbol{u}} + \varphi_{\text{mol}}(\boldsymbol{\xi}), && \boldsymbol{\xi} \in \Omega \\
\varphi_\Sigma^{\text{local}}(\boldsymbol{\xi}) \approx& \boldsymbol{W}_\xi(\hat{\boldsymbol{u}} + \hat{\boldsymbol{u}}_{\text{mol}}) - \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\tilde{\boldsymbol{V}}_\xi(\hat{\boldsymbol{q}} + \hat{\boldsymbol{q}}_{\text{mol}}), && \boldsymbol{\xi} \in \Sigma \\
\varphi_\Sigma^{\text{nonlocal}}(\boldsymbol{\xi}) \approx& \left(\frac{\varepsilon_\Omega}{\varepsilon_\Sigma} - \frac{\varepsilon_\Omega}{\varepsilon_\infty}\right)\tilde{\boldsymbol{V}}_\xi^{\boldsymbol{R}}(\hat{\boldsymbol{q}} + \hat{\boldsymbol{q}}_{\text{mol}}) - \frac{\varepsilon_\Omega}{\varepsilon_\infty}\tilde{\boldsymbol{V}}_\xi(\hat{\boldsymbol{q}} + \hat{\boldsymbol{q}}_{\text{mol}}) \\
&+ \boldsymbol{W}_\xi^{\boldsymbol{R}}\left(\hat{\boldsymbol{u}} - \frac{\varepsilon_\infty}{\varepsilon_\Sigma}\hat{\boldsymbol{w}} + \left(1 - \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\right)\hat{\boldsymbol{u}}_{\text{mol}}\right) + \boldsymbol{W}_\xi(\hat{\boldsymbol{u}} + \hat{\boldsymbol{u}}_{\text{mol}}). && \boldsymbol{\xi} \in \Sigma
\end{aligned}
$$

For these post-processors, $\boldsymbol{\xi}$ represents the position in space for which the corresponding potential should be computed and is either located in the protein domain Ω or the surrounding solvent Σ. Thus, for any given $\boldsymbol{\xi}$, the entities $\tilde{\boldsymbol{V}}_\xi$, $\boldsymbol{W}_\xi$, $\tilde{\boldsymbol{V}}_\xi^R$, and $\boldsymbol{W}_\xi^R$ take the form of row vectors or, equivalently, matrices of size $1 \times n$, which are not the main target of our work and can be represented explicitly in most cases. Alternatively, we could as well use our interaction matrices to achieve the same goal. There would be no considerable benefit in terms of memory savings but since we need each vector element at most once (and have to recompute the potentials for every change in $\boldsymbol{\xi}$ in any case), there would be no inherent computational overhead in doing so either.

In fact, $\mathcal{V}$, $\mathcal{K}$, $\mathcal{V}^R$, and $\mathcal{K}^R$ from the last section already provide everything we need to construct interaction matrices for $\tilde{\boldsymbol{V}}$, $\boldsymbol{W}$, $\tilde{\boldsymbol{V}}^R$, and $\boldsymbol{W}^R$ except for the new load point $\boldsymbol{\xi}$. Hence, we can reuse the same column elements and interaction functions of these matrices for the post-processors. What is more, the interaction matrix representation still holds if we decide to compute the same interior or exterior potentials for different load points at the same time. In this case, we can interpret

the list of load points as the new row elements of the interaction matrices and use the same post-processors to compute all potentials of interest at the same time, supported by the matrix-vector product implementation we have provided for solving the corresponding BEM systems.

We can find this exact use case among the post-processors in the form of the reaction field energies, where the reaction field potentials φ^* are computed at the positions of the partial point charges in the system:

$$W_{\text{ppc}}^{\text{rf}} \approx \sum_{i=1}^{N} \zeta_i \left\{ \tilde{\boldsymbol{V}}_{r_i} \hat{\boldsymbol{q}} - \boldsymbol{W}_{r_i} \hat{\boldsymbol{u}} \right\}.$$

Constructing matrices out of all individual row vectors $\tilde{\boldsymbol{V}}_{r_i}$ and $\boldsymbol{W}_{r_i}$ as outlined above, we can compute the reaction field energy of a given system as a single dot product:

$$W_{\text{ppc}}^{\text{rf}} \approx \underbrace{\begin{bmatrix} \zeta_1 \\ \vdots \\ \zeta_N \end{bmatrix}}_{=:\boldsymbol{\zeta}} \cdot \left(\underbrace{\begin{bmatrix} \tilde{\boldsymbol{V}}_{r_1} \\ \vdots \\ \tilde{\boldsymbol{V}}_{r_N} \end{bmatrix}}_{=:\tilde{\boldsymbol{V}}_\zeta} \hat{\boldsymbol{q}} - \underbrace{\begin{bmatrix} \boldsymbol{W}_{r_1} \\ \vdots \\ \boldsymbol{W}_{r_N} \end{bmatrix}}_{=:\boldsymbol{W}_\zeta} \hat{\boldsymbol{u}} \right),$$

where $\tilde{\boldsymbol{V}}_\zeta$ and $\boldsymbol{W}_\zeta$ are matrices of size $N \times n$ and can be represented as interaction matrices $\tilde{\mathcal{V}}_\zeta$ and $\mathcal{W}_\zeta$ with the same column elements and interaction functions as $\mathcal{V}$ and $\mathcal{K}$, respectively. Using $\tilde{\mathcal{V}}_\zeta$ and $\mathcal{W}_\zeta$, we can bring the approximation of the reaction field energy into matrix-free form:

$$W_{\text{ppc}}^{\text{rf}} \approx \boldsymbol{\zeta} \cdot \left(\tilde{\mathcal{V}}_\zeta \hat{\boldsymbol{q}} - \mathcal{W}_\zeta \hat{\boldsymbol{u}} \right).$$

5.3 Implicit arrays for Julia

Working with arrays of any dimension in Julia boils down to the abstract type `AbstractArray`. While abstract types in Julia cannot be instantiated or have their own member variables and functions, they serve as nodes in the type hierarchy graph. As such, `AbstractArray` is meant to be used as direct or indirect base type of any array-like type in the language. In its full form, `AbstractArray{T,N}` carries two type variables `T` and `N` for the type of its elements and the number of array dimensions, respectively, so that functions and other types can be specialized for

arrays of specific dimensions, element types, or both. As a matter of fact, most array-based functions in the Julia standard library are defined generically, enabling them to be used easily with different specialized array types. This is a very strong feature of the language, because it allows functionality and data types to be developed independently, and we exploit this option for our implicit matrix representations regularly throughout this section.

The Julia language already ships with a number of different array types, and many implicit ones in particular, including sparse arrays, reshaped arrays, and adjoint matrices. Many more examples are available from third parties through Julia's package system and there is virtually no limitation on the number of additional array types that could be added to the language. While there is no strict requirement for a new array type to be declared a subtype of `AbstractArray`, this detail opens the way to using generically defined standard and third-party array functions without considerable caveats. The Julia documentation[3] lists a set of functions as a common array interface that should be provided for subtypes of `AbstractArray`. Again, there is no real way of enforcing this requirement but many existing array operations rely on this and will fail eventually if these functions are missing.

In this section, we show the basic principles of creating new Julia array types. We start with arrays of fixed values as a simple introductory example and then focus on interaction matrices and other array types that we have encountered throughout this chapter. It should be noted that, albeit being valid, all of the code presented here has been significantly simplified to emphasize the key properties of the respective implementation. The main features not shown include bounds checks, convenience constructor overloads, setter functions, and conventional error handling. We provide the full implementations for most of the array types presented in this section in the form of *ImplicitArrays.jl*, a free and open-source Julia package available online[4].

5.3.1 Fixed-value arrays

One of the most simple array types to be represented implicitly is one that provides the exact same value for each of its elements. We have encountered a particular specimen of this category in the form of null matrices in the nonlocal BEM formulation already (cf. Section 5.2.2). The only information we need to store for such a matrix type is its dimensions. The represented value would then be provided by the getter function of our type.

[3] `https://docs.julialang.org/en/v1/manual/interfaces`
[4] `https://github.com/tkemmer/implicitarrays.jl`

Listing 5.1 – Minimal working example for an implicit null matrix in Julia

```
# Declare a type NullMatrix as two-dimensional integer array
struct NullMatrix <: AbstractArray{Int, 2}
    dims::NTuple{2, Int}
end

# Implement the standard size function and getter
Base.size(A::NullMatrix) = A.dims
Base.getindex(::NullMatrix, ::Int, ::Int) = 0
```

Listing 5.2 – Using the null matrix type with standard functions in Julia

```
julia> A = NullMatrix((2, 20))
2×20 NullMatrix:
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0

julia> A * ones(Int, 20, 5)
2×5 Array{Int64,2}:
0  0  0  0  0
0  0  0  0  0

julia> length(NullMatrix((1_000_000_000, 1_000_000_000)))
1000000000000000000

julia> NullMatrix((1_000_000_000, 1)) * ones(Int, 1, 1_000_000_000)
ERROR: OutOfMemoryError()
```

A minimal working example of such a `NullMatrix` type is shown in Listing 5.1 where we first declare our type to be a subtype of a two-dimensional `AbstractArray` with `Int` elements. `NullMatrix` has a single member variable `dims` to store the matrix dimensions as a 2-tuple of integers. At this point we would already be able to create `NullMatrix` objects that would be recognized as some sort of array by other functions. However, our type still lacks implementations for crucial interface functions. Firstly, we need to provide an overload for the `size` function of Julia's `Base` module, which is the common way to determine the dimensions of any given array. Secondly, we need to give access to the matrix elements. So far, there is no notion of the value represented by the `NullMatrix` type in the code other than its name. Getters are generally defined through the `getindex` function of the same `Base` module. The third interface function that is considered mandatory is a corresponding setter (via `Base.setindex!`). Since the null matrix is by definition a read-only type, we do not provide such an implementation[5].

Now we are finally able to construct `NullMatrix` objects of different dimensions and use them just like any other matrix in Julia. The language provides a default constructor for each type that expects one object per member variable. In the case of `NullMatrix`, this would consequently be a single 2-tuple for the dimensions.

[5] In practice, we would make this function throw an error with a suitable message.

Listing 5.3 – Custom product operator for null matrices in Julia

```
function Base.:*(A::NullMatrix, B::NullMatrix)
    @assert size(A, 2) == size(B, 1) "dimension mismatch!"
    NullMatrix((size(A, 1), size(B, 2)))
end

Base.:*(A::NullMatrix, B::AbstractArray{Int, 2}) = A * NullMatrix(size(B))
Base.:*(A::AbstractArray{Int, 2}, B::NullMatrix) = NullMatrix(size(A)) * B
```

Once instantiated, our null matrices can be used with most standard operations designed for array types. A few usage examples are shown in Listing 5.2: The second example represents a matrix product of a 2×20 null matrix and a 20×1 matrix of ones. The third usage example computes the length of the given null matrix by internally calling our custom `size` implementation for the `NullMatrix` type. As evident from the same example, we are now able to represent huge null matrices, with their dimensions solely being limited by the choice of data type (e.g., signed 64-bit integers in this case). An explicit representation of the same $10^9 \times 10^9$ null matrix using regular Julia arrays would require around 7 exabytes of memory while the implicit representation only requires a constant 16 bytes[6].

One important thing to consider when designing implicit representations is that most matrix operators in Julia return regular arrays. This can be seen in the second usage example, where the result is small enough to fit easily into memory in its explicit representation. However, this might not always be the case and result in `OutOfMemoryErrors` like in the fourth usage example. What is more, since the matrix product of a null matrix can be trivially represented by yet another null matrix (with appropriate dimensions), we might want to provide a custom implementation for this operator. As shown in Listing 5.3, we first define a matrix product for two null matrices[7]. In a second step, we provide overloads for the matrix product where only one argument is a null matrix. By means of these custom implementations, the previous matrix products can now be performed efficiently and represented in constant memory (Listing 5.4).

Of course, the `NullMatrix` type is not optimal from a software design perspective as we have fixed the represented value along with its type and the number of array dimensions. For any change in these properties, we would require a new type definition with only a few potential differences in their respective interface implementations. Hence, we could try to generalize our `NullMatrix` implementa-

[6] This requirement accounts for the tuple of integers representing the matrix dimensions and is thus completely independent of the actual matrix size.

[7] Although the binary infix multiplication operator is called `*` in Julia, it has to be referred to by `:*` in its fully-qualified name to prevent syntax ambiguities with its vectorized version, the `.*` operator.

Listing 5.4 – Using the null matrix product in Julia

```
julia> NullMatrix((2, 20)) * ones(Int, 20, 5)
2×5 NullMatrix:
 0  0  0  0  0
 0  0  0  0  0

julia> ones(Int, 2, 20) * NullMatrix((20, 5))
2×5 NullMatrix:
 0  0  0  0  0
 0  0  0  0  0

julia> NullMatrix((1_000_000_000, 1)) * ones(Int, 1, 1_000_000_000)
1000000000×1000000000 NullMatrix:
[...]
```

Listing 5.5 – Minimal working example for a fixed-value array type in Julia

```
# The array can now hold a value of any type and have any dimensions
struct FixedValueArray{T, N} <: AbstractArray{T, N}
    val::T
    dims::NTuple{N, Int}
end

Base.size(A::FixedValueArray{T, N}) where {T, N} = A.dims
Base.getindex(A::FixedValueArray{T, N}, ::Vararg{Int,N}) where {T, N} = A.↩
    val
```

tion: First, we could allow the matrix to hold any value other than zero, without restricting the element type[8]. Second, we allow any number of array dimensions. For this new, more generic type, we can make use of the same abstract base type, but this time we do not fix its type variables but keep them to be set by the user. Another minimal working example for such a `FixedValueArray` type is shown in Listing 5.5. The main difference to our previous implementation is that we now have to store the represented value (with generic type `T`) alongside the array dimensions and use a generic N-tuple instead of a 2-tuple for the latter. The interface functions also look similar to before, although we now have make them generic functions with declared type variables `T` and `N`. A few usage examples with different element types for `FixedValueArray` can be found in Listing 5.6.

5.3.2 Interaction matrices

The basic approach to providing an interaction matrix type in Julia is again very similar to the null matrices and fixed-value arrays from the last section, since we are implementing the same abstract array interface. At the same time, the

[8] This generalization would be at the cost of our custom null matrix products. If such a kind of operator optimization is considered beneficial in the particular use case, the following generalizations could always be provided in addition to a suitable "null array" type.

Listing 5.6 – Examples for fixed-value arrays with different element types in Julia

```
julia> FixedValueArray(0, (2, 5))
2×5 FixedValueArray{Int64,2}:
 0  0  0  0  0
 0  0  0  0  0

julia> FixedValueArray("(^_^) Hi!", (1, 2))
1×2 FixedValueArray{String,2}:
 "(^_^) Hi!"  "(^_^) Hi!"

julia> FixedValueArray(FixedValueArray(5.0, (1, 2)), (1, 3))
1×3 FixedValueArray{FixedValueArray{Float64,2},2}:
 [5.0 5.0]  [5.0 5.0]  [5.0 5.0]
```

Listing 5.7 – Minimal working example for an interaction matrix type in Julia

```
struct InteractionMatrix{T, R, C, F <: Function} <: AbstractArray{T, 2}
    rowelems::Vector{R}
    colelems::Vector{C}
    interact::F

    # constructor with a single type variable
    InteractionMatrix{T}(
        rowelems::Vector{R},
        colelems::Vector{C},
        interact::F
    ) where {T, R, C, F} = new{T, R, C, F}(rowelems, colelems, interact)
end

function Base.size(A::InteractionMatrix{T}) where T
    (length(A.rowelems), length(A.colelems))
end

function Base.getindex(A::InteractionMatrix{T}, i::Int, j::Int) where T
    A.interact(A.rowelems[i], A.colelems[j])::T
end
```

implementation should bear a close resemblance to the implicit interaction matrices representation we have specified in Definition 5.2.1. This implies that our `InteractionMatrix` type should somehow capture the row and column elements as well as the interaction function. The former two can be considered one-dimensional arrays of (potentially) different element types and lengths and can programmatically be represented using Julia's standard `Vector` type. Fortunately, functions are first-class citizens in Julia, thus the interaction function can also be treated like an object and captured as a member.

These observations lead to a rather intuitive translation to Julia code, as shown in Listing 5.7 (when ignoring the constructor for a moment). `InteractionMatrix` has four type variables to represent its element type `T`, the row element type `R`, the column element type `C` and the interaction function type `F` (which is constrained to be a subtype of `Function`). It should be noted that we could have used `Function` as the type of the `interact` member directly and, as a consequence, made the type

Listing 5.8 – Examples for different interaction matrices in Julia

```
julia> InteractionMatrix{Int}([1, 2, 3], [1, 2, 3], +)
3×3 InteractionMatrix{Int64,Int64,Int64,typeof(+)}:
 2  3  4
 3  4  5
 4  5  6

julia> InteractionMatrix{Int}([1, 2, 3], [1, 2, 3], (r, c) -> Int(r == c))
3×3 InteractionMatrix{Int64,Int64,Int64,var"#3#4"}:
 1  0  0
 0  1  0
 0  0  1
```

variable `F` obsolete. However, this would result in interaction matrices with the same arguments for `T`, `R`, and `C` to be considered of the same concrete type, thus restricting the compiler's ability to specialize the matrices and, more importantly, any operations on interaction matrices, for different interaction functions. Without this static type information, many decisions regarding the handling of the matrices would need to be made at runtime rather than at compile time, impairing the theoretically achievable performance. This is the main reason why abstract member variables should be generally avoided in Julia.

Although not strictly necessary, we have equipped `InteractionMatrix` with a custom constructor. In most cases, the type arguments are automatically inferred from the constructor arguments and can be omitted from the constructor call. Unfortunately, this is not possible here since the type variable `T` does not appear in any member variable whatsoever, so the compiler cannot know from the constructor arguments alone what concrete type `T` is supposed to be. Our custom constructor solves this problem by allowing us to specify `T` explicitly when called. Previously, we would have needed to provide all four type arguments explicitly, causing another problem as we do not necessarily know the concrete type of the provided interaction function. As can be seen in the usage examples in Listing 5.8, we can pass pre-defined operators or lambda expressions to the constructor, such as `+` or an anonymous function representing the Kronecker delta, with two parameters `r` and `c`. This usually results in cryptic types inferred by the compiler (e.g., `typeof(+)` and `var"#3#4"`). Specifying `F` as `Function` would work around this issue but result in the same situation as described above, restricting specialization.

Another peculiarity introduced in Listing 5.7 is an explicit type annotation `::T` on the value returned by the getter, i.e., the result of the interaction function evaluated for a specific pair of row and column elements. This is part of a larger problem of our current type design. More specifically, the requirements on the interaction functions are solely imposed by its only consumer, that is, the getter

Listing 5.9 – Type-safe function objects in Julia

```
abstract type InteractionFunction{R, C, T} <: Function end

struct ConstInterFun{R, C, T} <: InteractionFunction{R, C, T}
    val::T
end
(f::ConstInterFun{R, C, T})(::R, ::C) where {R, C, T} = f.val
```

Listing 5.10 – Interaction matrices with type-safe interaction functions in Julia

```
julia> f = ConstInterFun{Int, Int, Int}(0)
(::ConstInterFun{Int64,Int64,Int64}) (generic function with 1 method)

julia> f(1, 2)
0

julia> InteractionMatrix{Int}([1, 2, 3], [1, 2, 3], f)
3×3 InteractionMatrix{Int64,Int64,Int64,ConstInterFun{Int64,Int64,Int64}}:
 0  0  0
 0  0  0
 0  0  0
```

implementation: The function needs to accept two arguments of types `R` and `C`. Without the explicit annotation, the compiler would not know the return type of the interaction function and could not use this knowledge to specialize depending code. Even more problematic would be the consequential type inconsistency in the case where the interaction function actually returns an object of a different type. This could even happen intermittently as functions are generally not required to exclusively return objects of the same type. Consumers of our `InteractionMatrix` type would rightfully expect that the elements obtained through the getter would be of type `T` (as reported by the type definition) while they might actually be of any other type. In the best case, this could lead to errors resulting in program termination or, in the worst case, lead to faulty results without reported errors, especially when implicit type conversations are involved.

In an ideal world, we would specify all requirements on an interaction function directly in the `InteractionMatrix` type definition (i.e., the function should represent a map $R \times C \to T$) and these requirements should be enforced by the compiler. Luckily, such a degree of type safety can be achieved in Julia by defining custom function-like types. A concrete example for our use case can be found in Listing 5.9. We start by defining a new abstract type `InteractionFunction` as a subtype of `Function`[9]. Each concrete interaction function can then be defined as a new subtype of `InteractionFunction` and optionally contain additional member variables. The latter property allows us to pass information to the functions that is otherwise not

[9] Specifying `Function` as a base type does *not* make a type behave like a function. It only allows us here to use `InteractionFunction` objects with our previous `InteractionMatrix` implementation.

directly included in the row and column elements, such as the system constants ε_Ω and ε_Σ in the protein electrostatics setting. In the example shown above, we define a constant interaction function `ConstInterFun` that uses a member variable to represent the constant value similar to the `FixedValueArray` in Section 5.3.1. In order to make our function objects behave like functions, that is, make them callable, we also have to add a special method to the type. This method uses an object instead of an alphanumeric name as an identifier but otherwise works the same way as a regular function. For `ConstInterFun` we require two arguments of types `R` and `C`, respectively, and return the constant value of type `T` contained in the function object. Now, we can create objects of our new type and use them like any other function (cf. Listing 5.10).

For the real implementation of `InteractionMatrix`, we would now replace `Function` in Listing 5.7 by `InteractionFunction{R,C,T}` in order to restrict interaction functions to their type-safe variants. This way, we do not need the custom constructor any longer, since all type arguments can be inferred from the constructor parameters. Also, we could remove the explicit type annotation from the getter implementation as the type-safe interaction functions would allow the compiler to infer the return type as well.

5.3.3 Block matrices

In this section, we present a dedicated block matrix type in Julia rather than its interaction matrix equivalent briefly discussed in Section 5.2.2. While the latter could already be realized using our `InteractionMatrix` type, this might involve tedious interaction matrix harmonizations and duplications of basically the same row and column elements, depending on the given system. We have seen the implications for the particular case of the nonlocal system matrix in the section referenced above. A dedicated block matrix type shall serve as an alternative approach to the single-matrix problem. In the following, we discuss its properties and limitations with respect to computational overhead.

Listing 5.11 shows a simple working example for our block matrix type. Its basic idea is to represent the matrix as an aggregation of several equally-sized matrices. The block dimensions are stored separately for convenience. In that sense, `BlockMatrix` is just a wrapper for a matrix of matrices that hides the additional matrix layer, performs all index conversions, and maintains invariants of the blocks. The restriction of the block dimensions is mainly used to simplify the implementation, but it also facilitates custom implementations for certain matrix operations. Examples for such

Listing 5.11 – Minimal working example for a generic block matrix in Julia

```
struct BlockMatrix{T} <: AbstractArray{T, 2}
    bdims::NTuple{2, Int}
    blocks::Matrix{AbstractArray{T, 2}} # <- caution!
end

Base.size(M::BlockMatrix{T}) where T = M.bdims .* size(M.blocks)

function Base.getindex(M::BlockMatrix{T}, i::Int, j::Int) where T
    Base.getindex(
        M.blocks[cld(i, M.bdims[1]), cld(j, M.bdims[2])],
        (i - 1) % M.bdims[1] + 1,
        (j - 1) % M.bdims[2] + 1
    )
end
```

operations include a matrix-vector product for a block vector with the same dimensions such that matrix and vector blocks can be multiplied and added separately (a technique which we have generously exploited for the nonlocal BEM system in Section 5.2.2). Other than that, the block matrix could also be implemented for differently sized blocks as long as they eventually align into a rectangular shape.

A key decision is required when designing a block matrix type for Julia regarding whether and how to restrict the type of the matrix blocks. As it turns out, this is not a trivial question. The solution shown in Listing 5.11 does not have such a type restriction owing to a matrix of abstract arrays as a member. We have seen in the last section that there generally is a considerable performance overhead implied when using member variables of abstract types, because it deprives the compiler of its ability to specialize functions for our type[10]. Restricting the matrix blocks to interaction matrices, would simplify the individual interaction functions and remove the need to harmonize the matrix blocks. At the same time, aiming at maximal function specialization implies a restriction to the same exact type for all elements. In the case of our `InteractionMatrix{T,R,C,F}` definition, this requires fixing *all* of the type arguments as well, including the interaction function. At the end of the day, we would need to solve the same problems we have already encountered when developing an interaction matrix representation for block matrices.

On the other hand, not imposing any restrictions on the matrix blocks (in addition to a fixed size) would allow us to choose the best implicit (or even explicit) representation for each matrix block. A simple example for a block matrix of different component matrix types is shown in Listing 5.12. In this case, there is no need to represent a null matrix as an interaction matrix with artificial row and column

[10] Technically, there is also a general problem with containers of abstract elements as the elements can only be stored by reference in memory, requiring dereferencing on access and thus causing performance issues in certain situations.

Listing 5.12 – Example for a block matrix with elements of different types in Julia

```
julia> blocks = Matrix{AbstractArray{Int, 2}}(undef, 1, 3)
1×3 Array{AbstractArray{Int64,2},2}:
 #undef  #undef  #undef

julia> blocks[1] = [1 2; 3 4]; blocks[2] = NullMatrix((2, 2));

julia> blocks[3] = InteractionMatrix{Int}([2, 3], [1, 2], *);

julia> BlockMatrix((2, 2), blocks)
2×6 BlockMatrix{Int64}:
 1  2  0  0  2  4
 3  4  0  0  3  6
```

Listing 5.13 – Minimal working example for a row projection matrix in Julia

```
struct RowProjectionMatrix{T} <: AbstractArray{T, 2}
    base::AbstractArray{T, 2}
    rows::Vector{Int}
end

function Base.size(M::RowProjectionMatrix{T}) where T
    (length(M.rows), size(M.base, 2))
end

function Base.getindex(M::RowProjectionMatrix{T}, i::Int, j::Int) where T
    Base.getindex(M.base, M.rows[i], j)
end
```

elements only constructed for the sake of a uniform representation. However, there is no practical way to provide a fully generic yet highly performant block matrix type that does not impose any type restrictions on its blocks in Julia. Overall, the best solution for block matrices does not seem to be a single generic data type but rather a dedicated matrix type tailored to each of the specific systems of interest, based on the techniques presented and discussions held so far. In particular, we will see an optimized representation and operations for the nonlocal system matrix later in Section 5.3.5.

5.3.4 Matrix projections

The last implicit matrix type we briefly present in this chapter is not directly derived from the local and nonlocal BEM systems but rather motivated by the row-projection methods introduced in Section 5.1.1. We do not necessarily want to store the row projections explicitly in each iteration. Especially in the cases where the number of projected rows differs in each iteration (e.g., some randomized block Kaczmarz variants), it would be challenging to reuse preallocated memory efficiently.

Listing 5.14 – Example for a row projection matrix in Julia

```
julia> RowProjectionMatrix([1 2 3; 4 5 6; 7 8 9], [3, 1])
2×3 RowProjectionMatrix{Int64}:
 7  8  9
 1  2  3
```

Listings 5.13 and 5.14 show examples for the implementation and usage of a `RowProjectionMatrix` type in Julia. Similar to the block matrices, this type is mostly a wrapper for a given matrix and keeps track of the projected rows through an additional vector of row indices (relative to the base matrix). In particular, `RowProjectionMatrix` facilitates the handling of row projections in external code by mimicking a matrix of appropriate dimensions, which would otherwise require code modifications or duplication of the affected matrix rows.

5.3.5 Matrix-free systems for the cavity models

With a number of different implicit matrix implementations and concepts at hand, we can head back to the discretized local and nonlocal BEM systems and finalize our requirement analysis. To summarize, we have first defined a minimal interface for matrix-free solvers, requiring the given linear system to be represented in terms of a single system matrix and a single vector representing its outputs. The solver then computes the system input vector by means of matrix-vector products as sole operation required for the system matrix. In the following, we reify the presented ideas and implicit matrix types for the final protein electrostatics framework in Julia. However, we once again refrain from showing the exact implementation in the context of this manuscript and choose minimal code examples to focus on the underlying concepts and refer to chapters 6 and 7 for platform-specific considerations and the code repositories for the full implementations and documentations.

We have successfully identified the potential matrices $\boldsymbol{V}$, $\boldsymbol{K}$, $\boldsymbol{V}^R$, and $\boldsymbol{K}^R$ as central building blocks of the systems. In total, they can be represented as interaction matrices utilizing a single series of row elements (i.e., the centroids of all surface triangles), a single series of column elements (i.e., the surface triangles), and four interaction functions representing Eqs. (3.11), (3.12), (3.21), and (3.22). Although we have presented interaction matrices as a theoretical, unified framework to represent the whole system matrices alongside their components, we have also discussed how to provide more efficient solutions for the BEM systems in the form of matrix abstraction layers.

Listing 5.15 – Minimal concept draft for interaction matrix $\mathcal{K}$ in Julia

```
struct Triangle{T <: AbstractFloat}
    v1    ::Vector{T}   # position of the first node
    v2    ::Vector{T}   # position of the second node
    v3    ::Vector{T}   # position of the third node
    center::Vector{T}   # centroid of the triangle
    normal::Vector{T}   # normal vector of the triangle
end

struct Kfun{T} <: InteractionFunction{Vector{T}, Triangle{T}, T} end
# (f::Kfun{T})(ξ::Vector{T}, τ::Triangle{T}) where T = ...

const Kmat{T} = InteractionMatrix{T, Vector{T}, Triangle{T}, Kfun{T}}

function Base.:*(K::Kmat{T}, x::AbstractArray{T, 1}) where T
#    ... (specialized matrix–vector product Kx)
end
```

The potential matrices can be easily represented using the `InteractionMatrix` type with dedicated interaction functions. Since the latter functions are an integral part of `InteractionMatrix`, we can provide type aliases for all potential matrices and specialize any operation on interaction matrices for these individual aliases. A code example for an interaction matrix representation of $\boldsymbol{K}$ is shown in Listing 5.15, where we first define a surface triangle type that we can then use as the column elements of the potential matrices. In order to provide solutions for both single- and double-precision floating-point representations `Float32` or `Float64`, respectively, we keep this information generic via a type variable `T`. Thus, the functions are not only specialized for the different potential matrices but also for the precision level of the represented values. The interaction function `Kfun` evaluates Eq. (3.12), i.e.,

$$(\boldsymbol{K})_{ij} := -\frac{1}{4\pi} \int_{\tau_j} \frac{(\boldsymbol{r} - \boldsymbol{\xi_i}) \cdot \hat{\boldsymbol{n}}_j}{|\boldsymbol{r} - \boldsymbol{\xi_i}|^3} \mathrm{d}\Gamma_r$$

for a given load point ξ and surface triangle τ (code omitted). The concrete type for interaction matrix $\mathcal{K}$, here aliased by `Kmat` for simplicity, can then be used for specialized function overloads, such as the matrix-vector product $\mathcal{K}\boldsymbol{x}$ for some vector $\boldsymbol{x}$. Specific implementations for the remaining potential matrices are provided analogously.

With such implementations for both potential matrices $\boldsymbol{V}$ and $\boldsymbol{K}$, we can represent most parts of the matrix-free systems for the local cavity model from Theorem 5.2.4:

$$\frac{1}{2}\left(1 + \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\right)\hat{\boldsymbol{u}} + \left(\frac{\varepsilon_\Omega}{\varepsilon_\Sigma} - 1\right)\mathcal{K}\hat{\boldsymbol{u}} = \mathcal{K}\hat{\boldsymbol{u}}_{\text{mol}} - \frac{1}{2}\hat{\boldsymbol{u}}_{\text{mol}} - \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\mathcal{V}\hat{\boldsymbol{q}}_{\text{mol}},$$
$$\mathcal{V}\hat{\boldsymbol{q}} = \frac{1}{2}\hat{\boldsymbol{u}} + \mathcal{K}\hat{\boldsymbol{u}}.$$

Listing 5.16 – Minimal concept draft for the first local BEM system matrix in Julia

```
struct LocalSystemMatrix{T} <: AbstractArray{T, 2}
    K ::Kmat{T}
    εΩ::T
    εΣ::T
end

Base.size(A::LocalSystemMatrix{T}) where T = size(A.K)

function Base.:*(A::LocalSystemMatrix{T}, x::AbstractArray{T, 1}) where T
    ((1 + A.εΩ / A.εΣ) / 2) .* x .+ ((A.εΩ / A.εΣ - 1) .* (A.K * x))
end
```

Indeed, we can compute the outputs of both systems and represent the second system matrix as a single matrix object. What remains is an abstraction layer for the first system matrix. For this, we can store its components in terms of its matrix-vector product separately, i.e., the matrix $\mathcal{K}$ and the constants ε_Ω and ε_Σ, and provide a suitable overload for the matrix-vector product. A working draft for such an abstraction layer in Julia is shown in Listing 5.16. `LocalSystemMatrix{T}` presents itself as a matrix of element type `T`, with its size being the same as potential matrix $\boldsymbol{K}$ and with its matrix-vector product representing

$$\frac{1}{2}\left(1 + \frac{\varepsilon_\Omega}{\varepsilon_\Sigma}\right)\boldsymbol{x} + \left(\frac{\varepsilon_\Omega}{\varepsilon_\Sigma} - 1\right)\mathcal{K}\boldsymbol{x}$$

for a given $\boldsymbol{x}$, where `A.K * x` in the code refers to the specialized matrix-vector product of `Kmat` from Listing 5.15. It is worth noting that we currently do not provide getters or setters for `LocalSystemMatrix`, implying that we can only access its elements through the matrix-vector product.

We can implement the abstraction layer for the nonlocal BEM system in a similar way as the local case. This only requires storing the system constants in addition to the row and column elements, allowing us to construct the potential matrices on demand, e.g., in the matrix-vector product implementation. Since the nonlocal system is represented as a 3×3 block matrix, the size function needs to be implemented accordingly. A minimal concept draft for such an abstraction layer is shown in Listing 5.17, where the matrix-vector product implementation takes care of the matrix-free representation of the system as shown in Theorem 5.2.6. Furthermore, the single and double layer potential matrices appearing in the representation formulas (cf. Section 3.4) can be constructed just like the corresponding interaction matrices shown above by replacing the row elements accordingly. As a consequence, we do not need additional abstraction layers for the post-processors.

Listing 5.17 – Minimal concept draft for the nonlocal BEM system matrix in Julia

```
struct SystemParams{T <: AbstractFloat}
    εΩ::T
    εΣ::T
    ε∞::T
    λ ::T
end

struct NonlocalSystemMatrix{T} <: AbstractArray{T, 2}
    rowelems::Vector{Vector{T}}
    colelems::Vector{Triangle{T}}
    params  ::SystemParams{T}
end

function Base.size(A::NonlocalSystemMatrix{T}) where T
    (3 * length(A.rowelems), 3 * length(A.colelems))
end

function Base.:*(A::NonlocalSystemMatrix{T}, x::AbstractArray{T, 1}) where T
    K = Kmat{T}(A.rowelems, A.colelems, Kfun{T}())
#   V = ...
#   ...
end
```

The presented implementations for the potential matrices and the abstraction layers for the local and nonlocal system matrices could easily be extended to support other operations, e.g., the matrix product interface from Julia's `LinearAlgebra` module (cf. Section 5.4.1) or operations required by suitable preconditioners (cf. Section 5.4.2).

5.4 Solving implicit systems in Julia

We have seen in the last section that we can implement almost arbitrary matrix-like types in the Julia language, which allows us to bring linear systems into a form complying with our minimal solver interface from Definition 3.2.1. In this section, we direct our focus toward existing solver implementations, facilitating the comparison of different solution schemes and enabling us to choose the best-fitting solver for our linear systems.

Julia already ships with multiple linear system solvers in its `LinearAlgebra` module. More precisely, the standard solver is implemented as a polyalgorithm that calls the most suitable solver depending on the system matrix structure, possibly after invoking a suitable matrix factorization first. For dense matrices, the linear system is usually solved using BLAS [130] and LAPACK [131] routines, while for sparse matrices, additional routines from SuiteSparse [132] are used. Owing to the `AbstractArrays` interface, most standard solvers can also be used with our implicit

matrices. In fact, we can even solve the local and nonlocal protein electrostatics problem in this way. However, the system matrix is always copied during the standard solving procedure, rendering this option even worse than the original approach with an explicit representation of the system.

Third-party packages in Julia often provide high-level function interfaces only impose a few requirements on their arguments. In turn, they rely on the generic function interfaces initially provided by Julia's standard library and extended by other packages, especially in the case of math or array routines. This is in contrast to many other high-performance languages, such as C and C++, where operations frequently exploit low-level knowledge of the operands (e.g., their memory layout) for optimization. In Julia, the conventionally generic function overloads play particularly well in conjunction with Julia's dynamic type system and its symmetric multiple dispatch mechanism, as functions can be reliably dispatched even when all operands have varying dynamic types. For example, wrapping given array-like types in a custom array type (just as we did multiple times throughout the last section) allows us to provide specialized overloads that are automatically dispatched from within a third-party function supporting our custom type.

Using the same technique, we cannot only make our implicit array types compatible with existing iterative solvers, we can naturally also control how and where these operations are executed (e.g., on a GPU). Obtaining such solvers is comparatively simple: many third-party packages can be obtained directly through Julia's package repositories. For the electrostatics problem, we focus on matrix-free solvers (as discussed before). Once a candidate package is chosen, the operations used on the system matrix have to be identified in order to provide specialized overloads. This can usually be achieved in a quick way by consulting the package documentation, inspecting the code, or trial and error. Unfortunately, there is currently no build-in mechanism to automate this procedure in Julia and it has to be done manually.

5.4.1 IterativeSolvers.jl

The *IterativeSolvers.jl* package[11], maintained as part of the language's *Julia Math* project, provides a collection of CPU-based iterative solvers for linear systems, eigenproblems, and singular value problems, most of which can be used matrix-free. For nonsymmetric system matrices, the package provides essentially three options:

[11] `https://github.com/JuliaMath/IterativeSolvers.jl`

- **Restarted GMRES**: A variant of the original *general minimized residual* (GMRES) algorithm [113] that is restarted after a given number of times to reduce memory constraints,
- **IDR(s)**: A variant of the *induced dimension reduction* (IDR) method [133] that generates residuals in nested subspaces of shrinking dimensions [134, 135],
- **BiCGStab(l)**: A combination of the *bi-conjugate gradient* (BiCG) method [110, 111] with l GMRES iterations [116].

All of these solvers only require a matrix-vector product implementation by means of the `mul!` function provided through Julia's `LinearAlgebra` module. This function is mostly an extension of the standard multiplication operator that allows the computation to be performed in-place by expecting the destination vector as an additional argument. Hence, by providing a suitable `mul!` implementation for our implicit matrix types (in addition or instead of `Base.:*`), we can use all of the abovementioned solvers for the local and nonlocal protein electrostatics problem. What is more, since the matrix-vector product is potentially the single most dominant contributor to the above solvers' overall runtimes, we can even expect substantial performance improvements by providing parallel implementations for the operation (in contrast to the otherwise serial solvers).

5.4.2 Preconditioners.jl

Many iterative solvers – including GMRES and BiCGStab(l) from *IterativeSolvers.jl* – support the use of preconditioners, which can speed up the computation considerably. Preconditioner support in the package is implemented through split preconditioning

$$\left(\boldsymbol{P}_l^{-1}\boldsymbol{A}\boldsymbol{P}_r^{-1}\right)\boldsymbol{y} = \boldsymbol{P}_l^{-1}\boldsymbol{b}, \quad \boldsymbol{x} = \boldsymbol{P}_r^{-1}\boldsymbol{y},$$

where $\boldsymbol{P}_l$ and $\boldsymbol{P}_r$ can be defined separately. By using an identity matrix $\boldsymbol{I}$ for any subset of $\{\boldsymbol{P}_l, \boldsymbol{P}_r\}$, the split preconditioning can consequently be reduced to left preconditioning ($\boldsymbol{P}_r = \boldsymbol{I}$), right preconditioning ($\boldsymbol{P}_l = \boldsymbol{I}$), or to no preconditioning at all ($\boldsymbol{P}_l = \boldsymbol{P}_r = \boldsymbol{I}$).

The package can be used with any custom preconditioner $\boldsymbol{P}$ as long as there is a suitable implementation for the `ldiv!` function, i.e., the operation representing $\boldsymbol{P}^{-1}$ times a given matrix or vector. Alternatively, there exist a plethora of third-party packages for popular preconditioners, including incomplete LU decompositions[12]

[12] `https://github.com/haampie/IncompleteLU.jl`

and algebraic multigrid preconditioners[13], although many of them strictly expect sparse systems and are thus not suitable for our systems.

In our solvers, we rely on the *Preconditioners.jl* package[14] package, which generally provides a selection of preconditioners. However, the only usable option for our dense and non-symmetric systems is the simple Jacobi preconditioner (provided as `DiagonalPreconditioner`). It is implemented in terms of the `diag` function of Julia's `LinearAlgebra` module, returning the diagonal of the given matrix as a vector and can be specialized for our implicit matrix representations exactly the same way as `mul!`.

5.5 Implicit arrays for Impala

Developing implicit array representations for Impala is considerably different and usually less convenient than for Julia. For one, the language is in an early stage of its development and so far only provides a very sparse standard library. Unsurprisingly, there also exists just a few third-party packages that would be of use here. Another reason for the scarceness of features is the language's close connection to C. The build-in array type can best be compared to static C arrays, representing a patch of consecutive memory of fixed size and fixed element type. After all, the array types of Impala and C are byte-compatible and can be used for inter-language calls as long as the element types are byte-compatible as well (such as scalars, trivial aggregate types, or pointers to the same).

A second aspect to be considered for custom array types in Impala is that the memory location of objects always depends on the program context. It is in the nature of domain-specific languages that the same array of objects can reside in memory of different devices and platforms (e.g., host or GPU memory). Thus, our custom array types should allow the exact memory location to be transparent at least at some layers of the implementation. Similar considerations can naturally also be made for our implicit Julia arrays – and we will see in Chapter 7 how this is done as an optional extension – but here the domain-specific location is an integral part of the language and should be modeled accordingly.

In this section, we develop concepts to represent explicit vectors and implicit matrices in Impala. Due to the current lack of linear algebra libraries and solvers for the language, we restrict ourselves to approaches that are only partly specific to the

[13] `https://github.com/JuliaLinearAlgebra/AlgebraicMultigrid.jl`
[14] `https://github.com/mohamed82008/Preconditioners.jl`

Listing 5.18 – Built-in `Buffer` type for generic memory management in Impala

```
struct Buffer {
    data:   &[i8],
    size:   i64,
    device: i32
}
```

Listing 5.19 – Minimal working example for a real-valued vector in Impala

```
type Real = f64;

struct RealVector {
    size: fn()          -> i64,
    get:  fn(i64)       -> Real,
    set:  fn(i64, Real) -> ()
}

fn as_real_vector(buf: Buffer) -> RealVector {
    RealVector {
        size: ||          buf.size / sizeof[Real]() as i64,
        get:  |idx|       bitcast[&[Real]](buf.data)(idx),
        set:  |idx, val| bitcast[&mut[Real]](buf.data)(idx) = val
    }
}
```

protein electrostatics problem and can be extended to be used with future libraries. Once again, our presentation includes code examples with a limited amount of details. In particular, we usually do not show explicit annotations for the partial evaluation mechanism of Impala and refer the interested reader to Chapter 8 along with the corresponding code repository. In order to present valid Impala code in spite of the detail limit, incomplete definitions are commented out and supplemented with additional comments.

5.5.1 Buffers and vectors

The Impala runtime library provides a `Buffer` type to manage memory of domain-specific locations in a generic way (Listing 5.18). `Buffer` objects, henceforth simply referred to as *buffers*, store a pointer to a byte array of indeterminate length (`data`), the number of bytes it occupies in memory (`size`), and a numeric identifier for its location (`device`). Additionally, the runtime library provides convenience functions to allocate, free, and copy memory on supported platforms through buffers.

At the time of writing, the byte representation of data is the only way to provide generic containers in Impala. However, due to the actual element type not being associated with a buffer, such byte containers would need to be specialized for each given element type, with conversions being handled by means of static type casts.

Listing 5.19 shows such a minimal working example for a vector of floating-point values. First, we define a type alias `Real` for double floating-point precision that serves both as element type of the `RealVector` and as a proxy to later change the precision level at compile time.

It should be noted that the `RealVector` type defines a vector interface purely in terms of member functions. Hence, it does not expose any information on where or how its elements are stored. All implementation details are hidden in the interface functions, which need to be defined only when a `RealVector` object is constructed. This is a common pattern in Impala and other domain-specific languages as it allows to introduce layers of indirection to the code, i.e., abstraction layers that hide away domain-specific implementation details, such as the location or structure of the represented values. We henceforth refer to such function-based types as *interface types*. For instance, such types allow `RealVector` to wrap any sequential container of floating-point values in a fully transparent way, as long as there exists a meaningful implementation of the interface functions.

The most restrictive feature of `RealVector` in that sense is the `set` function because it requires the underlying data structure to be writable on a per-element basis. For buffer-based instantiations of `RealVector`, this restriction does usually not pose a considerable challenge as Impala allows both read and write access to its buffers. An example constructor (`as_real_vector`) for such a buffer-based `RealVector` can be found in Listing 5.19. Many other base representations, on the other hand, do not possess a meaningful write semantics (e.g., implicit representations). As a result, we use buffer-based vectors only for explicitly represented entities in the BEM formulations, i.e., system inputs and outputs as well as row and column elements of the interaction matrices.

In addition to real-valued vectors, we often encounter more semantically structured containers in the electrostatics problem, such as the row and column elements of the interaction matrices. In the former case, the containers represent a series of position vectors in $\mathbb{R}^3$ (usually the centroids of the surface triangles). Instead of implementing the row elements through a `RealVector` of length $3n$ for n surface triangles, we could also provide a dedicated `PositionVector` type where each element represents a position in space. While this distinction does not necessarily change the memory requirements of the representation, it enables us to develop more expressive interfaces, which are hopefully harder to use incorrectly. A working example for such a `PositionVector` type is given in Listing 5.20[15]. Since its elements are aggregates of simple floating-point objects, the implementation

Listing 5.20 – Examples for positions and vectors of positions in Impala

```
struct Position {
    x: Real,
    y: Real,
    z: Real
}

struct PositionVector {
    size: fn()              -> i64,
    get:  fn(i64)           -> Position,
    set:  fn(i64, Position) -> ()
}

fn as_position_vector(buf: Buffer) -> PositionVector {
    PositionVector {
        size: ||          buf.size / sizeof[Position]() as i64,
        get:  |idx|       bitcast[&[Position]](buf.data)(idx),
        set:  |idx, val| bitcast[&mut[Position]](buf.data)(idx) = val
    }
}
```

of the `as_position_vector` function to represent a buffer as `PositionVector` is completely analog to `as_real_vector`.

The `Position` type looks innocent enough to hide a crucial design decision having been made here: While the simple aggregate type serves as a good example for semantically structured data containers, we could instead have opted for a representation in terms of interface functions `x()`, `y()`, and `z()`, returning the respective `Real` value. In fact, this variant should generally be preferred if the represented data is already available in a different format or needs to be rearranged for optimized memory access patterns. However, this alternative representation has implications for the vector constructors as we would no longer be able to perform static type casts to the function-based `Position` type in order to access the vector elements. We analyze the effects for a more complex example: a vector type for surface triangles, i.e., the column elements for our potential matrices.

For a vector of triangle objects, we repeat the same procedure as above and create a dedicated `Triangle` type to represent surface triangles as well as a corresponding `TriangleVector` type (Listing 5.21). This time, we model the element type in terms of interface functions. In addition to the concerns regarding memory access patterns raised above, the function-based interface also allows for a minimal representation of a triangle in terms of its nodes while the other quantities `center` and `normal` could be computed on demand. This would also leave us an option to add further quantities, such as the triangle area or its distance.

[15] `PositionVector` actually represents a vector of position vectors. In order to avoid these nomenclature issues, position vectors in the mathematical sense are here referred to as *positions* and the container types as *vectors*.

Listing 5.21 – Examples for triangles and triangle vectors in Impala

```
struct Triangle {
    v1:     fn() -> Position,   // first node
    v2:     fn() -> Position,   // second node
    v3:     fn() -> Position,   // third node
    center: fn() -> Position,   // centroid
    normal: fn() -> Position,   // unit normal vector
}

struct TriangleVector {
    size: fn()              -> i64,
    get:  fn(i64)           -> Triangle,
    set:  fn(i64, Triangle) -> ()
}
```

5.5.2 Context managers for data containers

One consequence of Impala's close connection to C is the need of manual memory management. Because this is and has been a bottomless source of intermittent errors and vulnerabilities in the history of C software, we present a simple method to mitigate these problems on a small scale before continuing to talk about container types for matrices.

The Impala language provides a special syntax for a construct that is somewhat similar to `with`-statements in Python, commonly referred to as *context managers*. The basic idea behind such context managers is to wrap a series of instructions on a certain resource (usually a file or database connection) in such a way that the resource is automatically released once the instructions are processed. In contrast to Python, the `with`-constructs in Impala are modeled as expressions rather than statements and turn out to be syntactic sugar for a simple function call. More specifically, a function

$$f : A_1 \times \cdots \times A_n \times (B_1 \times \cdots \times B_m \to C) \to R$$

can be called as

```
with b_1, ..., b_m in f(a_1, ..., a_n) {
    // block expression returning a value of type C
}
```

where each a_i is an object of type A_i, each b_j is an object of type B_j, and the whole expression returns an object of type R, that could, e.g., be bound to a variable or returned as part of an outer expression.

Listing 5.22 – Host-specific context manager for real-valued vectors in Impala

```
fn new_real_vector(size: i32, body: fn(RealVector) -> ()) -> () {
    let buf = alloc_cpu(size * sizeof[Real]());
    body(as_real_vector(buf));
    release(buf);
}

fn do_something() -> () {
    with vec in new_real_vector(10) {
        vec.set(0_i64, 42 as Real);
        vec.set(1_i64, vec.get(0_i64) + 1295 as Real)
    } // <- vec is automatically destroyed
}
```

Context managers have proven a very powerful and versatile tool in Impala. Besides the classic resource management purpose, they can for example be used to capture error states. We have built a testing framework on this idea[16], where tests can be grouped hierarchically by using nested `with`-expressions.

Here, we use context managers in a more classic setting to implement automatic memory management for our vector types. For this, we provide a function f of the kind shown above that handles the allocation and release of the underlying buffer. Listing 5.22 shows an example implementation for the `RealVector` type. The context manager then tracks the object lifetime and invokes the release of claimed memory as soon as the `with`-expression is left.

The same mechanism can also be exploited to transfer memory between different platforms. A typical workflow when using coprocessors is to first copy local memory to the device, perform the computation there, and then copy the results back to local memory. The transfer part of this workflow can be automated through a context manager and buffers in a similar way as shown above. What is more, with a domain-specific implementation of the context manager, the distinction between a computation on the local CPU or the coprocessor can be made completely transparent. We will see more of this in Chapter 8.

5.5.3 Matrices and matrix operations

Generic matrix types can be defined in the same way as their vector counterparts (for a given element type). This time, we refrain from including a setter declaration as part of the interface to facilitate the implementation of implicit matrices where a meaningful per-element write semantics does not exist.

[16] `https://github.com/tkemmer/badhron`

Listing 5.23 – Minimal working example for a real-valued matrix in Impala

```
struct RealMatrix {
    size: fn()         -> (i64, i64),
    get:  fn(i64, i64) -> Real
}

fn new_fixed_value_matrix(rows: i64, cols: i64, val: Real) -> RealMatrix {
    RealMatrix {
        size: ||    (rows, cols),
        get:  |i,j| val
    }
}

fn new_null_matrix(rows: i64, cols: i64) -> RealMatrix {
    new_fixed_value_matrix(rows, cols, 0 as Real)
}

fn new_identity_matrix(size: i64) -> RealMatrix {
    RealMatrix {
        size: ||    (size, size),
        get:  |i,j| if i == j { 1 as Real } else { 0 as Real }
    }
}

/* generic matrix-vector product */
// fn mv_real(A: RealMatrix, x: RealVector) -> RealVector { ... }
```

Listing 5.24 – Concept draft for potential matrix $\boldsymbol{K}$ as `RealMatrix` in Impala

```
fn compute_Kij(xi: Position, elm: Triangle) -> Real {
    let mut Kij = 0 as Real;
    // ... compute Kij
    Kij
}

fn new_matrix_K(rows: PositionVector, cols: TriangleVector) -> RealMatrix {
    RealMatrix {
        size: ||    (rows.size(), cols.size()),
        get:  |i,j| compute_Kij(rows.get(i), cols.get(j))
    }
}
```

Listing 5.23 shows the reduced floating-point matrix interface as well as multiple exemplary matrix constructors. The `RealMatrix` type can still be used to provide read-only access to buffer data. Instead, we can now also exploit the lack of a setter and implement fixed-value and null matrices based on the Julia models presented in Section 5.3.1, or arbitrarily sized identity matrices. We can even implement interaction matrices as `RealMatrix` objects by defining a suitable constructor, as shown for potential matrix $\boldsymbol{K}$ in Listing 5.24. Since all of these matrices would be implemented through different constructors using the same `RealMatrix` type, they could benefit from a common set of functions, such as a generic matrix-vector product (indicated as `mv_real` function in Listing 5.23).

Listing 5.25 – Concept draft for a dedicated potential matrix type in Impala

```
struct PotentialMatrix {
    rowelems: fn()         -> PositionVector,
    colelems: fn()         -> TriangleVector,
    size:     fn()         -> (i64, i64),
    get:      fn(i64, i64) -> Real
}

// Conversion PotentialMatrix -> RealMatrix
fn as_real_matrix(mat: PotentialMatrix) -> RealMatrix {
    RealMatrix {
        size: mat.size,
        get:  mat.get
    }
}

/* specialized matrix-vector product */
// fn mv_pot(A: PotentialMatrix, x: RealVector) -> RealVector {...}
```

When defining custom matrix operations in Impala, we have to keep two of the language's properties in mind: Firstly, there is no type inheritance system in place, which implies that we cannot provide functions with arguments defined in terms of base types, e.g., functions for abstract floating-point numbers (such as `AbstractFloat` for `Float64` and `Float32` in Julia). Secondly, functions cannot be overloaded or overridden in Impala. Hence, we cannot provide interface functions meant to be overloaded for custom data types (such as Julia's `AbstractArray` interface functions we have seen in Section 5.3). As a consequence, we need to define separate, uniquely-named operations for each element type of our vectors and matrices and for each additional argument. All these restrictions make it considerably harder to develop generic data types that can be used with third-party software, including linear algebra libraries and solvers alike.

With `RealMatrix`, we can at least provide a minimal interface for all real-valued matrices and use it for common operations. We can utilize the type even as a wrapper for other specialized matrices with the same element type and thus mimic a weak kind of type compatibility by means of explicit conversion functions. Moreover, the conversion can then be done overhead-free in many cases due to Impala's partial evaluation mechanism, rendering this kind of representation highly suitable for cross-library interfaces (see Section 5.6).

Let us for a moment reconsider the previous potential matrix design for Impala. Instead of the generic `RealMatrix` type, we might want to provide a dedicated `PotentialMatrix` type that additionally exposes the row and column elements, namely, the corresponding `PositionVector` and `TriangleVector` objects (cf. Listing 5.25). The implementation of a suitable constructor would be the same as before, except for two new member functions returning the row and column elements

obtained as constructor arguments. Apart from that, the new `PotentialMatrix` type behaves like the previous `RealMatrix` implementation. For the compiler, however, there is one crucial difference: `PotentialMatrix` and `RealMatrix`, in spite of their similar behavior, represent two distinct and incompatible types. This allows us to provide "specialized" operations for potential matrices, e.g., matrix-vector products (here: `mv_pot`) in addition to those for more general `RealMatrix` instances. This specialization would be common to all instances of `PotentialMatrix`, including the potential matrices encountered so far and any real-valued interaction matrix with the same row and column elements. This method can subsequently be repeated for any matrix type in need of specialization, e.g., when a separate specialization is required for each concrete potential matrix or, more precisely, for each concrete interaction function.

Supplying a suitable conversion function from `PotentialMatrix` to `RealMatrix` is trivial since the full `RealMatrix` interface is already included in `PotentialMatrix` and can be forwarded accordingly (see `as_real_matrix` in Listing 5.25). This is generally true for all alternative matrix types either satisfying the `RealMatrix` interface directly (by definition) or providing corresponding data members instead. Through the conversion function, `PotentialMatrix` objects can be optionally called with operators expecting either type. In particular, a matrix-vector product for potential matrices can be computed using the generic or specialized implementation.

Different implementations for operations, e.g., the matrix-vector product, could alternatively be realized without a dedicated matrix type by demanding a `RealMatrix` argument for the `mv_pot` function in Listing 5.25[17], since each function requires a unique name in any case. However, this way we can prohibit `RealMatrix` objects not representing potential matrices from calling the specialized matrix-vector product variant, thus making the interface harder to use incorrectly.

5.5.4 Matrix-free systems for the cavity models

With the techniques presented above, we are finally able to represent implicit matrices – and potential matrices in particular – as well as to provide vector and matrix operations that can be specialized at least for some types. For the local and nonlocal BEM systems, we only need additional abstraction layers for the respective system matrices, just like we have seen multiple times throughout this chapter.

[17] Assuming that the latter does not use the additional interface functions `rowelems` and `colelems`.

Listing 5.26 – Minimal concept draft for the local or nonlocal BEM matrix in Impala

```
struct SystemParams {
    epsOmega: Real,
    epsSigma: Real,
    epsInf:   Real,
    lambda:   Real
}

struct LocalSystemMatrix {
    K:      fn() -> PotentialMatrix,
    params: fn() -> SystemParams,
    size:   fn() -> (i64, i64)
}

/* Matrix-vector product for the local matrix */
// fn mv_local(A: LocalSystemMatrix, x: RealVector) -> RealVector {...}

struct NonlocalSystemMatrix {
    rowelems: fn() -> PositionVector,
    colelems: fn() -> TriangleVector,
    params:   fn() -> SystemParams,
    size:     fn() -> (i64, i64)
}

/* Matrix-vector product for the nonlocal matrix */
// fn mv_nonloc(A: NonlocalSystemMatrix, x: RealVector) -> RealVector {...}
```

In concept, we could reuse the `RealMatrix` interface for the system matrices. However, taking into account the discussion from the last sections, we rather opt for custom types with dedicated matrix-vector products for each of them. This way we can also spare ourselves a getter implementation and express all system matrices in terms of their matrix-vector product only.

Listing 5.26 shows two possible definitions for the system matrix abstraction layers that are very close to their Julia counterparts from Section 5.3.5. Many other variants are equally possible as long as they provide all information required by the matrix-vector product implementation or any other operation defined for the types (e.g., for preconditioning).

5.6 Solving implicit systems in Impala

To the best of our knowledge, there currently exist no native linear algebra libraries or iterative solvers for Impala that we could utilize for the protein electrostatics problem. One workaround for this problem would be to use suitable C or C++ libraries wherever a missing Impala counterpart does not exist. Since this language crossing comes at the cost of Impala's partial evaluation capabilities, this would challenge the decision to work with Impala in the first place. Thus, this method is

Listing 5.27 – Concept draft for a linear algebra library *LAOps* in Impala

```
struct LAVectorF64 {
    size: fn()    -> i64,
    get:  fn(i64) -> f64
}

struct LAMatrixF64 {
    size: fn()         -> (i64, i64),
    get:  fn(i64, i64) -> f64
}

/* Vector sum */
// fn vector_add_f64(u: LAVectorF64, v: LAVectorF64) -> LAVectorF64 { ... }

/* Dot product */
// fn vector_dot_f64(u: LAVectorF64, v: LAVectorF64) -> f64 { ... }

// ...
```

not considered a goal of this work. Instead, we outline in the following paragraphs how to approach cross-library data transfer and function specialization in spite of Impala's type and function restrictions. This way, we hope to lay a foundation for the future development of libraries.

We have seen in Section 5.5.3 that function specialization in Impala is a general problem, especially when it comes to third-party library calls. Since all types are categorically incompatible to each other, we cannot provide functions for common base types. We have also seen how to implement type conversion functions to mimic a kind of type compatibility and how this approach can be combined with interface types to create functions that are generic in some sense. The same idea can now be used for Impala libraries where the latter ships with interface types for data representations (e.g., vectors and matrices). Any functionality provided by the library (e.g., vector operations and matrix-vector products) is then defined in terms of these interface types.

Let us assume the existence of two hypothetical Impala libraries *LAOps* and *LASolver* for vector/matrix operations and linear algebra solvers, respectively, both of which we want to utilize for the protein electrostatics problem. Let us further assume that we cannot modify these libraries for a particular use case (e.g., to directly include our specialized matrix-vector products for potential matrices) and have to work with the interfaces as is.

The *LAOps* library is meant as a general-purpose linear algebra library, shipping with vector and matrix types for all built-in Impala types. In accordance with the above discussion, they are exclusively defined in terms of interface types. Listing 5.27 shows two such definitions, `LAVectorF64` and `LAMatrixF64`, for double-precision

Listing 5.28 – Concept draft for a matrix-free solver library *LASolver* in Impala

```
struct LASystemMatrixF64 {
    /* All operations required on system matrix */
    times_vec: fn(LAVectorF64) -> LAVectorF64
}

/* Iterative solver for Ax = b */
// fn solve(A: LASystemMatrixF64, b: LAVectorF64) -> LAVectorF64 { ... }
```

floating point vectors and matrices, respectively. In the given form, `LAVectorF64` is a read-only variant of `RealVector` while `LAMatrixF64` is semantically equivalent to `RealMatrix`, albeit being of different and thus incompatible types. In addition to these interface types, the library provides a variety of functions, including vector sums, dot products, and generalized matrix-vector products.

The `LASolver` library, on the other hand, is build upon `LAOps` and extends the library by functions to solve linear systems, including matrix-free solvers. In order to comply with our minimal solver interface from Definition 3.2.1, the actual solver function (`solve`) expects the system matrix (in some format) and outputs (as `LAVectorF64` object), computes the system inputs and returns them as `LAVectorF64` object. The system matrix is modeled here by yet another dedicated matrix type `LASystemMatrixF64`. This special treatment is necessary as the more generic `LAMatrixF64` type would impede the solver from calling a specialized matrix-vector product and instead enforce the corresponding implementation from `LAOps` (the example in Section 5.5.3 is analog to this case). As a consequence, `LASystemMatrixF64` is not a mere wrapper for `LAMatrixF64` but rather a representation of the *operations* required on the system matrix, such that specialized implementations for these operations can be fed into the solver.

Listing 5.28 shows a concept draft for a matrix-free solver in `LASolver`, where the required operation is of course a matrix-vector product for the system matrix. In a sense, this model is again very similar to the implicit representation of the BEM matrices in Julia, where the matrices are also defined only in terms of their matrix-vector product (cf. Section 5.3.5).

Now, our own protein electrostatics solver would also be based on `LAOps` types, either by using `LAVectorF64` and `LAMatrixF64` directly or providing similar types (e.g., `RealVector` and `RealMatrix`) alongside suitable bidirectional conversion functions[18]. In particular, the local and nonlocal BEM solvers can create their

[18] While such implementations would be trivial for `RealMatrix`, special care would need to be taken for `RealVector`. Since there is no matching setter in `LAVectorF64`, a conversion function from the latter to `RealVector` cannot be implemented in a fully operational way.

Listing 5.29 – Example usage of linear algebra libraries in Impala

```
/* new matrix-vector product with LAops types */
//fn mv_nonloc(A: NonlocalSystemMatrix, x: LAVectorF64) -> LAVectorF64 {...}

/* nonlocal BEM solver using LAops and LAsolver
fn solve_bem() -> LAVectorF64 {
    // Create nonlocal system
    let mat = new_nonlocal_system_matrix(...);
    let A = LASystemMatrixF64 {
        times_vec: |vec| matvec_nonlocal_real(mat, from_lavectorf64(vec))
    };
    let b = ...

    // solve system
    solve(A, b)
}
*/
```

corresponding systems the same way as before, convert them to `LAOps` types if necessary, and call a matrix-free solver from `LAsolver` (Listing 5.29).

Until suitable linear algebra facilities are available in Impala, real-world examples of the cross-library communication techniques presented here can be found in the *Badhron* text framework[19]. In Badhron, interface types enable users, among other things, to extend the framework by custom assertion functions tailored to their particular needs. The test framework's application in other Impala software can then, for example, be reviewed in the repository of our Impala-based protein electrostatics solver (Chapter 8).

[19] `https://github.com/tkemmer/badhron`

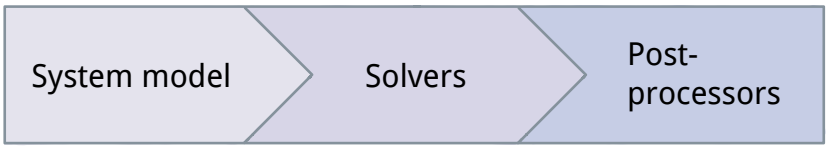
6 NESSie.jl

In this chapter, we present *NESSie.jl*, or *NESSie* for short, our own Julia package providing efficient and intuitive BEM solvers for the local and nonlocal protein electrostatics problems[1]. An earlier version of this package, only providing solvers for the explicit BEM system representations, has been published before [75] with a focus on an extensible package design while highlighting features of the Julia language auxiliary in the general context of scientific computing. NESSie is loosely based on a prototypical reference program for the BEM formulations we use. It is written in C and was originally published as [61]. While the prototype delegates most linear algebra operations to the highly optimized ATLAS library [136], it is restricted to explicit representations for the linear systems, performs all computations sequentially, and lacks a comprehensible user interface, which renders the prototype unsuitable for general use.

Throughout this chapter, we outline the organization of the NESSie package and design decisions resulting in its sophisticated usage workflow. We further present how the basic ideas of the reference program were extended for the implicit system representations and potential shortcomings of the method. Finally, we direct our attention to the post-processors and analytically solved test models available through the package.

6.1 Package organization

The common usage workflow of NESSie can be described as a three-step process:



First, the input data is read into a unified system model representation, which is then transferred to a supported BEM solver in the second step. In the third and final

[1] `https://github.com/tkemmer/nessie.jl`

Module	Description
`NESSie`	Common system models, constants and functions
`NESSie.BEM`	Local and nonlocal BEM solvers and postprocessors
`NESSie.Format`	Supported input and output formats
`NESSie.Radon`	Internal quadrature routines
`NESSie.Rjasanow`	Internal solvers for Laplace potentials
`NESSie.TestModel`	Analytically solved test models

Table 6.1. – NESSie.jl modules

step, any number of post-processors can be applied to the results of the solver to yield the desired quantities, including electrostatic potentials and energies.

The types and functions of the package are organized into different submodules, shown in Table 6.1. The main module, `NESSie`, is the starting point for the protein electrostatics problem in Julia and comprises a system model representation (cf. Section 6.2) along with common functions and constants for all submodules. NESSie supports a variety of different data formats for its system models (cf. Section 6.2), with corresponding functions being provided through the `NESSie.Format` module. The local and nonlocal BEM solvers for both explicit and implicit system matrix representations as well as a number of post-processors are available through the `NESSie.BEM` module (cf. Section 6.3). Two additional modules `NESSie.Radon` and `NESSie.Rjasanow` contain functionalities internally used by the solvers, namely, quadrature routines (with weights and quadrature points taken from [65]) and an analytic solver for the Laplace potentials based on techniques presented in [64], respectively. Finally, the `NESSie.TestModel` module provides analytically solved test models for evaluation purposes (cf. Section 6.4).

In addition to its modules, the NESSie package ships with a full documentation[2], code examples, and convenience tools to generate input meshes and perform conversions between supported data formats.

6.2 System models

In order to comply with the three-step workflow shown above, system models in NESSie are designed to comprise all information required for the solving process. At the same time, they are generic enough so that they can be extended and made compatible with future solver alternatives. System models can be created from and

[2]The documentation is also available online: `https://tkemmer.github.io/NESSie.jl/latest`

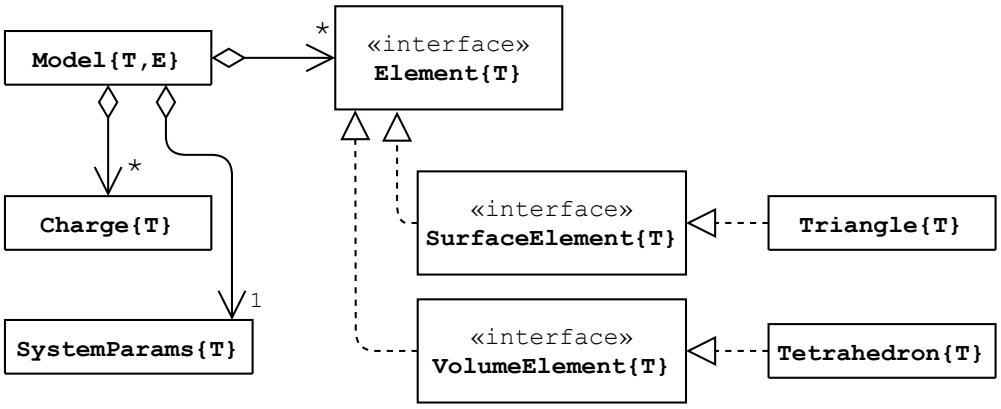


Figure 6.1. – The extensible NESSie system model, where `T` is a subtype of `AbstractFloat` and `E` is a subtype of `Element{T}`, enabling function specializations for each individual choice of `T` and `E`.

exported to a variety of different data formats to facilitate integration of NESSie into existing pipelines.

6.2.1 Unified representation

NESSie uses a common data structure for all system models, consisting of three different components: a suitable discretization of the relevant system domains (e.g., a triangulation of the protein surface as introduced in Section 3.2.2), a charge model, and a set of system constants describing the solute and solvent properties. NESSie currently assumes a point charge model as shown in Eq. (2.1), that is, a finite number of point charges, each represented by a position vector and a dimensionless (partial) charge value. The system constants include the dielectric constants ε_Ω, ε_Σ, and ε_∞ as well as the correlation length λ for the nonlocal dielectric solvent response (cf. Section 3.3.1).

The discretization of the system domains can be represented in various different ways: NESSie distinguishes between surface and volume meshes and further between the simplex type the particular mesh is based on. For a surface mesh, these simplices are usually triangles or quadrilaterals, while tetrahedra or pyramids are common choices for volume meshes. Figure 6.1 shows an abstract view on the NESSie system model (`Model`) with all currently supported simplex types (`Triangle` and `Tetrahedron`). Additional simplices can then easily be added by providing new subtypes for the abstract `SurfaceElement` and `VolumeElement` types.

The main reason for the generic system model design in NESSie is to facilitate the exploitation of Julia's function specialization mechanism. Each instance of

Listing 6.1 – Example code of a function specialized for different NESSie model types. Generally, the function most specific to all given function arguments is dispatched.

```
using NESSie

# general models
f(::Model{T}) where T                                = println("model")

# general surface or volume models
f(::Model{T,E}) where {T, E <: SurfaceElement{T}} = println("surface")
f(::Model{T,E}) where {T, E <: VolumeElement{T}}  = println("volume")

# triangle-based surface models
f(::Model{T,Triangle{T}}) where T                    = println("triangle")
```

Listing 6.2 – Usage examples for functions specialized for different model types. Note the differing output for triangle- and tetrahedron-based models owing to the lack of a more specific function for the latter.

```
julia> f(Model{Float64, Triangle{Float64}}())
triangle

julia> f(Model{Float64, Tetrahedron{Float64}}())
volume
```

`Model` carries two type variables `T` and `E` which represent the precision level of floating-point numbers and the mesh type, respectively. Thus, functions with `Model` parameters can be specialized for each possible combination of `T` and `E` if required. An example for such a function specialization is shown in Listings 6.1 and 6.2, where the invocation of the function `f` with different models results in different outputs. For instance, this mechanism can be used to provide different electrostatics solvers under a common name such that a suitable implementation can be automatically chosen based on the given model. Conversely, specializations for base types can be used to provide common solvers for derived models (e.g., for *all* surface models). Among other things, this mechanism is used in the Julia language for the default linear system solver, where the concrete solver is chosen with respect to the system matrix (cf. Section 5.4).

It should be noted that the BEM solvers presented in this manuscript only support surface triangulations, i.e., surface meshes with triangular simplices. An earlier version of NESSie shipped with a prototypical FEM solver implementation for the nonlocal electrostatics problem using the volume mesh part of the model [75]. However, FEM support has been discontinued indefinitely in favor of the *JuliaFEM* project [137], which was developed at the same time as NESSie, and will be revisited at a later point in time. In the meantime, we will present in Chapter 7 how the system model can be used with Julia packages other than NESSie.

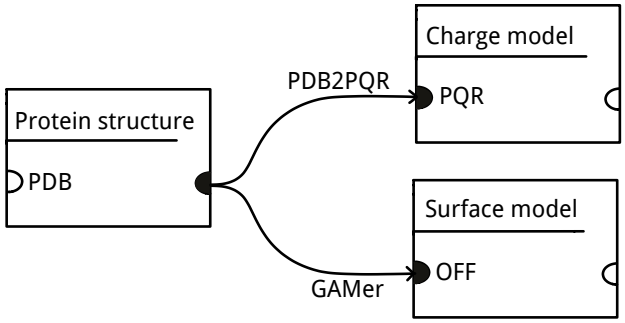


Figure 6.2. – Example workflow for generating a NESSie-compatible system model (surface mesh and charge model) from a single PDB file using PDB2PQR and GAMer.

6.2.2 Input formats

NESSie aims at maintaining an intuitive interface for a heterogeneous target audience. This involuntarily implies supporting commonly used data formats for the generation of system models. Two different points of view are especially important to be considered for this task: In the bioinformatics context, we find a source of protein models as well as applications for the protein electrostatics problem. From a numerical perspective, we also need to consider suitable discretization techniques for the system domains.

In the biology and bioinformatics fields, *Protein Data Bank (PDB)* [19] files are the predominant way of representing the 3D structure of proteins and other biomolecules. Thousands of such files are freely available through the PDB website[3], which can then be used to visualize the corresponding structures or to generate partial point charge models for the same. The latter can be obtained using the *PDB2PQR* application [138], resulting in so-called *PQR* files, which can be fed into NESSie.

Many libraries and applications for the generation of surface and volume meshes for given structures exist, some of which being supported by NESSie (see Table B.1 in Appendix B for a full list of supported formats). In the context of this work, we highlight one particular representative, namely, *GAMer* [139]. GAMer is part of the *Finite Element ToolKit (FETK)*[4] and thus well suited for the generation of both surface and volume meshes for the purpose of numerical treatment. What is more, meshes can be computed directly from PDB files, such that a complete set of input data required for our BEM solvers can be generated from a single PDB file, as shown in Figure 6.2. GAMer is mainly intended to be used as a library in other programs. While it does expose some of its functions directly through a few simple

[3] `https://www.rcsb.org`
[4] `http://fetk.org`

main programs that can be build alongside the library, changes of parameters require active code manipulation. To facilitate the generation of NESSie-compatible meshes without prior programming knowledge, we included two C++ applications in the NESSie repository where parameters can be set through a command-line interface instead. The first application is called *Mesher* and can be used to generate surface meshes for a given PDB file. The second application, *Sphere*, generates surface meshes for spherically symmetric domains of a given radius and resolution (e.g., to be used with NESSie's test models). In any case, the output are meshes in the form of OFF files that can be fed into NESSie. Pointers to the format specifications for both PQR and OFF can be found in Appendix B (Table B.3).

6.2.3 Output formats

Although we are equipped with all tools necessary to generate system models, we still need to consider two important aspects: Firstly, NESSie does not provide any means to visualize or assess the quality of the given systems directly. Secondly, so far there is no cross-platform representation for system models. The second point is not an issue as long as we do not leave the Julia ecosystem, since we can use `Model` objects to transfer system models between packages. In this case, we can generate (and preprocess) input data with NESSie and perform subsequent tasks through other Julia packages that are either compatible with `Model` directly or provide a type interface `Model` objects can be converted to. For software outside the Julia ecosystem, NESSie can be used to export system models into multiple established data formats. A full list of output formats available in NESSie can be found in Appendix B (Table B.2), along with pointers to the concrete format specifications the implementations are based on (Table B.3).

Most of the available export formats primarily target visualization tools, implying a focus on assemblies of geometrical structures rather than physical representations of biomolecular systems. As a consequence, most data formats for this purpose can only represent the surface or volume meshes of our system models. For NESSie, we chose common file formats for structured data with a broad software support as well as a variety of more specialized plain-text formats. In addition to the visualization-focused variants, we implemented another simple plain-text format in NESSie that captures not only the surface or volume mesh information but also the charges of a given system model. It is based on the *HMOFile* implementation of the *Biochemical ALgorithms Library (BALL)* [30] and is also used by the reference program [61] mentioned at the beginning of this chapter. To the best of our knowledge, *HMOFile*

Listing 6.3 – Structure of the HMO+ format for triangulated surfaces in NESSie.jl

```
BEGIN_NODL_DATA
    <number of nodes>
    1   <x>   <y>   <z>
    2   <x>   <y>   <z>
    ...
END_NODL_DATA

BEGIN_ELEM_DATA
    <number of elements> <unused>...
    1   1   103   <index node1>   <index node2>   <index node3>
    2   1   103   <index node1>   <index node2>   <index node3>
    ...
END_ELEM_DATA

BEGIN_CHARGE_DATA
    <number of charges>
    1   <x>   <y>   <z>   <charge>
    2   <x>   <y>   <z>   <charge>
    ...
END_CHARGE_DATA
```

represents a slight modification of the *HMO* format for polygonal meshes and surfaces. According to the documentation of the ROXIE software used at the *European Organization for Nuclear Research (CERN)* [140, 141], the HMO format was used by the HyperMesh [142] and Hermes2D [143] mesh generators in the past. Traces of the original HMO format can still be found in the same documentation[5] but are scarce otherwise.

Listing 6.3 outlines the general structure of the extended HMO format implemented in NESSie, henceforth referred to as *HMO+* to emphasize the modifications. The system model is logically separated into three blocks of enumerated items, representing the mesh nodes (`NODL_DATA`), mesh elements (`ELEM_DATA`), and point charges (`CHARGE_DATA`). The HMO format generally supports multiple different surface and volume elements. HMO+ support in NESSie is currently restricted to triangular meshes (indicated by the number 103 in the `ELEM_DATA` block), but can easily be extended to other element types when needed.

6.2.4 Format conversion

With several input and output formats at hand, we can now use NESSie as common system model provider and preprocessing facility for all of our BEM solvers; either directly via the `Model` type or indirectly via HMO+ as a transfer format. All supported data formats are available through the `NESSie.Format` module, where `Model` serves

[5] `https://espace.cern.ch/roxie/Documentation/`

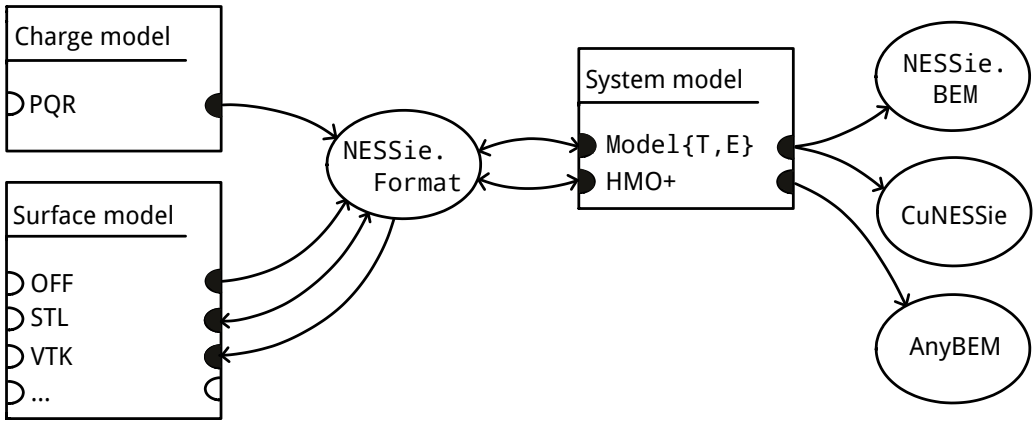


Figure 6.3. – Format conversion options for system models with surface meshes through the `NESSie.Format` module and system model compatibility with our solvers.

as both destination type for all input routines and source type for all output routines. As a result, `NESSie.Format` can be considered a full-fledged conversion unit, which can be utilized independently from the solvers. For this reason, NESSie ships with an additional Julia script *Converter.jl* to provide immediate access to all available format conversion options. Exported surface models can then, for instance, be visualized and assessed with the 3D creation suite *Blender* [144] through the popular *STL* format. Alternatively, surface or volume models can be exported into the *PolyData* or *UnstructuredGrid* formats of the *Visualization Toolkit (VTK)* [145], respectively, and visualized with *ParaView* [146]. Other visualization software known to work with exported system models include applications such as *MeshLab* [147] or *GeomView* [148] as well as libraries such as *XML3D* [149] or *BALL* [30].

As the only cross-platform format for whole system models, HMO+ plays a special role in the workflow of the protein electrostatics problem followed in this manuscript. While system models can be directly fed into both of our Julia solvers, `NESSie.BEM` (Section 6.3) and *CuNESSie* (Chapter 7), HMO+ is used to transfer the preprocessed models to our Impala solver *AnyBEM* (Chapter 8). An example for the role of `NESSie.Format` as conversion unit for system models based on surface meshes is shown in Figure. 6.3.

6.3 Solvers and post-processors

In compliance with the three-step NESSie workflow presented in Section 6.1, the system models can be passed to any compatible solver. Currently, NESSie ships

with different variations of solvers for the local and nonlocal electrostatics problems, all of which expect triangulation-based system models. Two variants of each solver determine whether the system matrix is to be represented explicitly (as a simple floating-point matrix) or implicitly (as a composition of interaction matrices). Depending on the precision level of the system model (i.e., single or double floating-point precision), the specialization of the chosen solver variant for the same precision level is invoked automatically. In any case, the sole input to the solver is a complete system model, which is then used to compute and return the approximation coefficients of the reaction field potentials at the molecular surface. The return value of all solvers is encapsulated in a uniform result type that can then be fed into any available post-processor (and any number thereof).

6.3.1 Solvers for the explicit representations

The local and nonlocal BEM solvers for explicit system representations are closely modeled after the prototypical reference program shortly introduced in the beginning of this chapter. The primary goal of these solvers is their computational efficiency in spite of the tremendous memory constraints imposed by the explicit representations. For this purpose, matrix and vector objects occupy pre-allocated memory that is reused whenever possible. In particular, each potential matrix is computed at most once and most operations are performed in place by means of state-of-the-art BLAS [130] routines, which are available through Julia's `LinearAlgebra` module.

Overall, the solving procedure with explicit representations is split into two computationally expensive stages: First, the system matrix and system outputs are fully assembled. Then, the system is solved using Julia's standard solver, which internally delegates the task to a LAPACK solver [131]. The reference program generally performs all computations in a single thread of execution, although both stages contain several data-independent operations that can conceptionally be computed in parallel. As we have seen before, this is especially true for the system matrix, or more precisely, the elements of the potential matrices. What is more, some BLAS implementations provide optional multithreading support for their operations (e.g., OpenBLAS[6] and ATLAS [136]). In Julia, this behavior can be controlled through an additional function call, granting a partially concurrent version of the otherwise serial solvers for free. Other than that, multithreading support for embarrassingly parallel tasks (such as the system matrix assembly) can be enabled via Julia macros from Julia's `Threads` module (cf. Section 4.1).

[6] `https://www.openblas.net/`

Listing 6.4 – Simple parallel matrix-vector product in Julia

```
using Base: Threads

function mv_parallel(A::Matrix{Float64}, x::Vector{Float64})
    m, n = size(A)
    dst  = zeros(m)
    Threads.@threads for i in 1:m
        dst[i] = sum(A[i, j] * x[j] for j in 1:n)
    end
    dst
end
```

That being said, we assume typical system matrices to exceed feasible memory limits in the context of this work, rendering the explicit-representation variants of the BEM solvers highly impractical. While coarse-grained triangulations of smaller molecules only occupy a few gigabytes of memory and could as well be represented explicitly, neither the local nor nonlocal systems scale well in this regard. This is why we utilize these solvers only as a tool for comparison against the reference program and do not explicitly provide multithreading support for them.

6.3.2 Solvers for the implicit representations

The local and nonlocal BEM solvers for the implicit system representations are a direct realization of the data types developed in Section 5.3.5 and the solving techniques discussed in Section 5.4. As such, the system matrices are represented by dedicated matrix-like types, accessible only in terms of matrix-vector products (for the iterative solvers) and their diagonals (for the preconditioning). The potential matrices serving as their components are modeled through interaction matrices available from our own Julia package *ImplicitArrays.jl*[7], with specialized matrix-vector product implementations in NESSie. We selected the *restarted GMRES* solver from the *IterativeSolvers.jl* package to be a sensible default for all BEM systems, augmented with a Jacobi preconditioner from the *Preconditioners.jl* package (see Chapter 9 for a short discussion).

All computations in NESSie are exclusively performed on the CPU. Therefore, we implemented basic multithreading support for the solvers through Julia's `@threads` macro. More specifically, the matrix-vector products for the potential matrices[8] are performed row-wise, whereas the matrix rows are equally divided between the available worker threads and computed in parallel. A simplified implementation for

[7] `https://github.com/tkemmer/implicitarrays.jl`

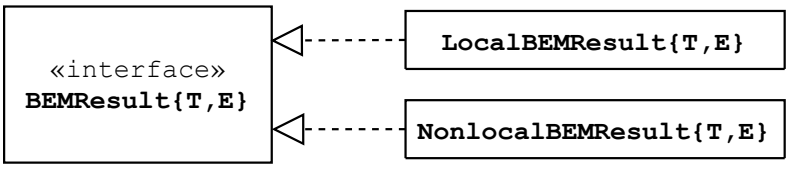


Figure 6.4. – Common BEM solver result type in NESSie, where `T` is a subtype of `AbstractFloat` and `E` is a subtype of `Element{T}`.

such a multithreaded matrix-vector product is shown for double-precision floating-point values in Listing 6.4. The number of worker threads can be controlled through the `JULIA_NUM_THREADS` environment variable. Unfortunately, the multithreading facilities in Julia are still considered an experimental language feature (as of version 1.4) and do not scale particularly well. This is one of the reasons why we focus on GPU-accelerated parallelization schemes in a separate Julia package that will be the main topic of the next chapter. We will get back to another CPU-parallel implementation of our solvers later in Chapter 8.

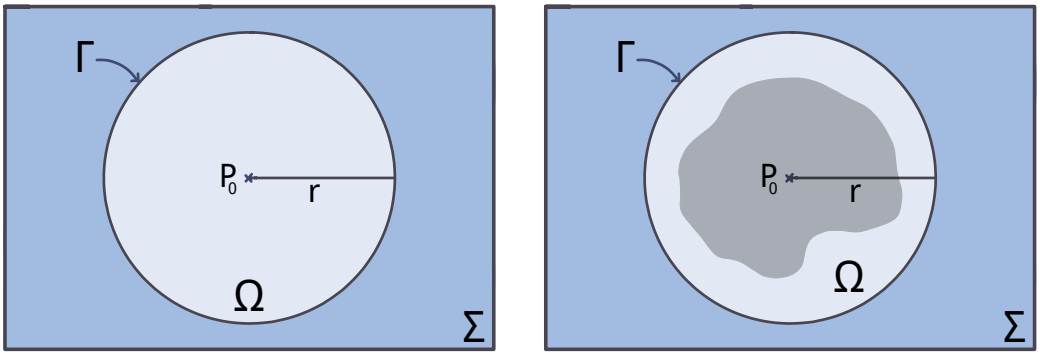
6.3.3 Post-processors

Each BEM solver returns an object representing the given system model and the computed approximation coefficients for the electrostatic potentials on the protein surface. In other words, the resulting objects contain all information needed for the calculation of derived quantities (cf. Section 3.4). NESSie currently ships with post-processors for the interior and exterior electrostatic potentials as well as for the computation of reaction field energies. The objects returned from our solvers are modeled in such a way that they can be used with any one of these post-processors. More specifically, the `NESSie.BEM` module provides an abstract `BEMResult` type, which is used as a base type for all solver results (Figure 6.4). Each post-processor can be addressed by its unique function name. When passed to a post-processor function, a suitable specialization is then automatically determined by the given system model and solver type (e.g., local or nonlocal).

6.4 Test models

While the BEM formulations for the protein electrostatics problem can be solved analytically for simple geometries, surface approximations of actual proteins are

[8] For potential matrix $\boldsymbol{K}$, we actually provide a full matrix product implementation to account for the fact that two products with the matrix have to be computed in each iteration (cf. Section 5.2.2).



(a) Born model **(b)** Poisson test model

Figure 6.5. – Schematic representation of the test models available in NESSie. In all cases, the "protein" boundary Γ is a sphere centered at the origin $P_0 = [0, 0, 0]^T$. Left: a single point charge is located at P_0. Right: Multiple point charges are provided by a rescaled protein (gray shape inside the sphere).

usually far too complex for this purpose. This is why NESSie provides two test models for simple spherically-symmetric systems that can be used for the evaluation of the solver results. We briefly describe these models in the following sections.

6.4.1 Born model

The first model represents a monoatomic ion, modeled as a single point charge at the center of a vacuum-filled sphere (Figure 6.5a). The implementation of this so-called *Born model* in NESSie is based on the derivation in [49, Section 5.3]. We provide implementations for both local and nonlocal problem formulations, with their corresponding potential functions being summarized in Appendix A.

In addition to these functions, NESSie ships with Born models and spherical meshes for a selection of different monovalent and divalent ions, so that the models can be solved analytically or via our BEM solvers. The individual sphere radii for these *Born ions* are physically motivated and have been chosen according to [49, Section 5.3.2], based on simulations from [150]. A list of available Born ions in NESSie is given in Section 9.2.1.

6.4.2 Nonlocal Poisson test model

The second test model can handle multiple charges inside the sphere at once. The implementation of this model in NESSie is based on the first *nonlocal Poisson dielectric test model* presented in [151] and henceforth referred to as *nonlocal Poisson test*

model. In their publication, the authors present two different variants of the Poisson test model for the nonlocal case, in addition to one for the local case. Furthermore, they provide a reference implementation in the form of a free Fortran package. The latter can be used to compute electrostatic and reaction field potentials for a given point charge model (via a PQR file), a given sphere radius, and a pre-defined set of system constants. The charge model is usually computed from a suitable protein, which is then rescaled to fit into the sphere (Figure 6.5b).

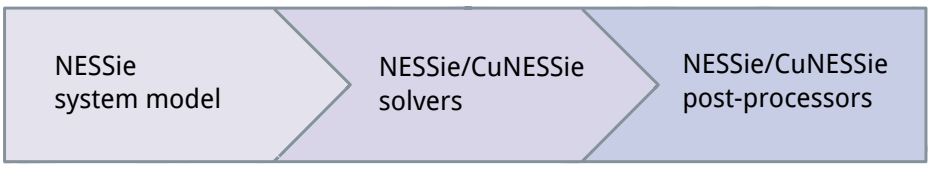
The interior and exterior potential functions used to implement one of the original nonlocal Poisson dielectric test models in NESSie are summarized in Appendix A. It should be noted, however, that the Poisson test model assumes a slightly different behavior of the nonlocal dielectric function of the solvent as compared to the BEM formulations our solvers are based on. While the response function is also expressed in terms of two dielectric constants ε_Σ and ε_∞, it is considered to act on the whole space $\mathbb{R}^3$ instead of being restricted to the solvent domain Σ. This behavior effectively extends the solvent interdependencies into the protein domain Ω and thus leads to slightly different results for the reaction field potentials as compared to our BEM solvers. As as consequence, direct comparisons between the analytical test model and the numerical BEM solvers should always be taken with a grain of salt. We will later see in Section 9.3.1 that the differences primarily occur in close proximity to the sphere and that they become negligible in the full electrostatic potentials.

7 CuNESSie.jl

With the addition of implicit system model representations to NESSie in Chapter 6, the package directly contributes towards the feasibility of the BEM approach for the nonlocal protein electrostatics task. While interaction matrices for the potential matrices reduce the required memory drastically as compared to an explicit representation, this comes at the cost of computational efficiency. In particular, the expensive computations of the matrix elements have to be run repeatedly during the iterative solving process. Unfortunately, high-performance capabilities of CPU parallelism are limited in this regard. Instead, the embarrassingly parallel nature of the matrix element computation can better be exploited using hardware accelerators. For this purpose, we present *CuNESSie.jl*, or *CuNESSie* for short, our own Julia package that extends NESSie with CUDA-accelerated versions of the local and nonlocal BEM solvers for the implicit system representations.

7.1 Interoperability with NESSie.jl

CuNESSie is built upon NESSie data types and can be neatly integrated in its usage workflow: CuNESSie reuses the NESSie system model representation (cf. Section 6.2) and can thus work with models generated or preprocessed through NESSie directly. Similarly, the CuNESSie solvers reuse NESSie's dedicated BEM result types (cf. Section 6.3.3). This is why CuNESSie does not only behave in many ways like its role models, but its solvers and post-processors can even be used as drop-in replacements for all (or some) of their NESSie counterparts. Hence, the adapted usage workflow for the protein electrostatics task in Julia takes the following form:



7.2 GPU abstraction

CuNESSie does not provide an extensive submodule structure but rather exposes its interface functions directly to the users. Internally, however, most functionality is organized either into *host code* or *device code*, depending on whether it is meant to be executed on the CPU or GPU, respectively. The transition between host and device is usually handled automatically in our package and thus fully transparent to the user. Beyond this abstraction layer, the exact context of computation still has to be perpetually taken into account in the implementations, as the different platforms bear certain restrictions with regard to the availability of functions and data types as well as to differing memory locations for the handled objects.

7.2.1 Host vs. device code

As briefly introduced in Section 4.3, device code is initially called through a *kernel function*, which invokes the context switch from host to device. It is usually not obvious at first glance whether a given function is part of the host or device code since both are written in Julia. Conversely, not every piece of Julia code could or should be executed on the GPU. For example, device code cannot perform dynamic function calls, i.e., the function dispatch must always happen at compile time. This restriction already excludes a considerable amount of Julia functions from device code, especially those relying on the dynamic inference of their argument types. For other functions, there might simply be more efficient implementations provided by the CUDA framework. Most notably, this includes mathematical functions from the *CUDA Math API*, which are available through the *CUDAnative.jl* package[1] [103, 104] and which should be preferred over their native Julia counterparts.

Similarly, not every data type can be used in device code. Apart from those requiring dynamic type deductions (e.g., owing to abstract data members), array types as well as aggregate types containing arrays need a special treatment. This is mainly due to the fact that each GPU ships with its own separate memory, which is commonly further split into different purpose-specific memory types and might need to be associated explicitly with objects in the code.

[1] `https://github.com/JuliaGPU/CUDAnative.jl`

Listing 7.1 – Concept draft for a `CuArray`-compatible kernel function in Julia

```
using CuArrays, CUDAnative

#= device code =#
function cuvector_kernel(v::CuDeviceVector{T}) where T
    @cuprintln("Commencing kernel execution...");
    # ...
    return
end
```

Listing 7.2 – Kernel call example with a `CuArray` parameter in Julia

```
julia> v = CuVector([1, 2, 3])
3-element CuArray{Int64,1,Nothing}:
 1
 2
 3

julia> @cuda cuvector_kernel(v)
Commencing kernel execution...
```

7.2.2 CuArrays

A typical data processing workflow with CUDA-enabled GPUs heavily depends on array representations. The input data is usually first prepared on the host before being copied into a suitable part of the device memory. Subsequent results are then, in turn, copied back from the device onto the host.

To facilitate this frequently occurring procedure, one can utilize the dedicated array types of the *CuArrays.jl* package[2] [103, 152]. The general `CuArray` type and its more specific variants `CuVector` and `CuMatrix` (for one- and two-dimensional arrays, respectively) represent data residing in the device memory, with an interface usable in host code. In other words, array data encapsulated in a `CuArray` object is silently transferred to the device, leaving the user with an object that mostly behaves like any other array, except that operations on it might be performed on the device.

The *CuArrays.jl* package provides many overloaded functions for its arrays, including vectorized array operators (e.g., element-wise sums or products) and matrix products. These functions internally use native vendor libraries whenever possible. The underlying array data is automatically transferred back into host memory as soon as the `CuArray` object is converted into a regular Julia array.

Alternatively, `CuArray` objects can be passed manually to a kernel function, where they are converted into into their corresponding device-specific representation, namely, `CuDeviceArray`, `CuDeviceVector`, or `CuDeviceMatrix` (all provided by the

[2] `https://github.com/JuliaGPU/CuArrays.jl`

CUDAnative.jl package). Listings 7.1 and 7.2 show a simple example for such a kernel function that can be invoked with a `CuArray` argument. `CuVector` and `CuDeviceVector` are per se incompatible types and since we are passing an object of the former type to a function that requires the latter, we would expect the kernel call to fail. However, the type conversion is performed as part of the `@cuda` macro before the function is evaluated, causing the kernel to be called with the correct object in the first place. The actual conversion is implemented through an abstract interface provided by the *Adapt.jl*[3] package [103], which we will later extend for custom device array types.

7.3 Potential matrices for CUDA

Vector operations build an essential basis for our BEM solvers, but we still need to translate the (host-based) implicit system representations into a device-compatible form. In this section, we particularly focus on their building blocks, i.e., suitable representations for the potential matrices and efficient implementations for their matrix-vector products.

7.3.1 Semantically-structured device arrays

Choosing interaction matrices as blueprints for the potential matrices implies the need for a device-compatible representation for their row and column elements as well as their interaction functions. The latter can be considered translations of the original functions in terms of device code. And as long as we assume that each matrix element is computed in a single thread of execution, we do not need to develop sophisticated parallelization schemes for the same purpose.

For the row and column elements, we can use `CuArray` or `CuDeviceArray` objects, depending on the context. So far, we have chosen floating-point vectors to represent positions in space. As a consequence, the row elements were implemented as a vector of such vectors (for the triangle centroids) and the column elements were implemented as a vector of aggregated vectors (for the triangle nodes, the centroids, and the normal vectors; see Section 5.3.5).

Unfortunately, working with several short vectors in CUDA can become quite cumbersome because one cannot freely and dynamically allocate arrays in device code, as

[3] `https://github.com/JuliaGPU/Adapt.jl`

Listing 7.3 – Concept draft for device-compatible position vectors in Julia

```
using CUDAnative

#= host/device code =#
const CuPosition{T} = NamedTuple{(:x, :y, :z), NTuple{3, T}}

#= device code =#
struct CuPositionVector{T} <: AbstractArray{T, 1}
    dim::Int
    vec::CuDeviceVector{T, AS.Global}
end

Base.size(v::CuPositionVector{T}) where T = (v.dim,)

function Base.getindex(
    v::CuPositionVector{T},
    i::Int
) where T
    (x = v.vec[i], y = v.vec[v.dim + i], z = v.vec[2 * v.dim + i])
end
```

Listing 7.4 – Example usage of named tuples in Julia

```
julia> v = (x = 1.0, y = 2.0, z = 3.0)
(x = 1.0, y = 2.0, z = 3.0)

julia> v isa CuPosition{Float64}
true

julia> v.x, v.y, v.z
(1.0, 2.0, 3.0)
```

opposed to native Julia host code. Such nested (and possibly mixed) array structures are usually flattened into simpler floating-point vectors before transferring them onto the device, often resulting in a loss of structural information. For example, the row elements for our potential matrices could be brought into the form of a single floating-point vector with the first n values representing the x-coordinates of all n surface triangles, the next n values representing their y-coordinates, and the remaining n values representing their z-coordinates.

While this form of data is usually easier to express in device code, it does no longer contain any information regarding the underlying structure. Consequently, such representations are highly error-prone and generally hard to maintain. In the above example, the same representation could have been achieved by concatenating the coordinates of all n triangles directly instead of separating the axes. In both cases the result is a floating-point vector of length $3n$, but accessing the individual positions requires additional knowledge about the ordering to provide correct results.

One simple way to mitigate this problem is to provide an abstraction layer on top of the flattened single-vector representation that allows to access the row elements

Listing 7.5 – Concept draft for device-compatible triangle vectors in Julia

```
#= host/device code =#
const CuTriangle{T} = NamedTuple{
    (:v1, :v2, :v3, :normal),
    NTuple{4, CuPosition{T}}
}

# device code =#
struct CuTriangleVector{T} <: AbstractArray{T, 1}
    dim::Int
    vec::CuDeviceVector{T, AS.Global}
end

Base.size(v::CuTriangleVector{T}) where T = (v.dim,)

function Base.getindex(
    v::CuTriangleVector{T},
    i::Int
) where T
    # ...
end
```

in a semi-structured manner. More specifically, we aim at a vector representation for positional data that behaves like `CuArray` and allows to read its elements into individual registers when requested. Listing 7.3 shows an example implementation for such an abstraction layer, implemented by means of Julia's own `NamedTuple` type. First, we create a type alias `CuPosition{T}` to represent a single position as a tuple containing three objects of type `T` associated with the names `x`, `y`, and `z`. The individual values can then be accessed through the tuple as if they were member variables of a corresponding aggregate type (see Listing 7.4 for an example).

The `CuPositionVector` type internally uses the same flattened row element representation as described above, separating all coordinates by axis. The base vector is here stored in global device memory (`AS.Global`) and its length – in terms of positions – is stored separately (`dim`). The getter is then responsible for creating named tuples for a given position index, such that each position can be treated and passed around like a single object in the device code, while for the device itself, the coordinates are just individual floating-point values occupying local registers once read from global memory.

A similar representation can be found for the column elements by using a named tuple for each surface triangle, given a suitable flattened base representation (Listing 7.5). The transparent nature of such an additional abstraction layer leaves us the option to change the internal structure of the represented elements, e.g., to implement more beneficial memory layouts (cf. Section 7.3.2), without affecting depending code. For instance, the column elements could be internally stored solely

Listing 7.6 – Automatic host-to-device conversion for position vectors in Julia

```
using Adapt, CuArrays

#= host code =#
function Adapt.adapt_storage(
    a   ::CUDAnative.Adaptor,
    pos::Vector{Vector{T}}
) where {T <: AbstractFloat}
    # rearrange data into desired layout
    vec = CuVector(T[#= ... =#])
    # convert to device type
    CuPositionVector(length(pos), Adapt.adapt_storage(a, vec))
end

#= device code =#
function hello_kernel(vec::CuPositionVector{T}) where T
    @cuprintln("Greetings, human! Your position vector was well received!")
    # ...
    return
end
```

Listing 7.7 – Kernel call example for the automatic host-to-device conversion in Julia

```
julia> v = Vector{Float64}[zeros(3), ones(3)]
2-element Array{Array{Float64,1},1}:
 [0.0, 0.0, 0.0]
 [1.0, 1.0, 1.0]

julia> @cuda hello_kernel(v)
Greetings, human! Your position vector was well received!
```

by the individual triangle nodes while other properties, such as the normal vectors, are computed by the getter function.

In order to enable automatic memory transfer (and host-to-device type conversions) for our new vector types, we can extend the same *Adapt.jl* interface used by the *CuArrays.jl* data types. For this, we first need to decide on how to represent the vectors on the host and then provide an overloaded version of the corresponding conversion function. Listing 7.6 shows an example implementation of such a function (`Adapt.adapt_storage`) for our `CuPositionVector` type. The row elements are initially represented as a vector of floating-point vectors (`pos`), rearranged to match the desired memory layout and subsequently transferred onto the device through a `CuVector` object (`vec`). The data is finally packaged into a `CuPositionVector` object, where the `CuVector` is converted into a corresponding `CuDeviceVector` through its regular *Adapt.jl* conversion function. The row elements can now finally be passed in its host representation (`Vector{Vector{T}}`) to a suitable kernel function with a `CuPositionVector{T}` parameter (`hello_kernel`) through the `@cuda` macro (Listing 7.7).

7.3.2 Parallel matrix-vector products

The techniques presented in the last section allow us to reuse our previous potential matrix representations in the host code and convert the corresponding row and column elements into suitable device representations. But we still need to translate the interaction functions into device code before we can fully use the interaction matrices in kernel functions. To fully utilize the multiprocessing capabilities of CUDA-enabled GPUs, we also need suitable parallelization schemes for these functions. More specifically, we need to decide which parts of the computations are best performed in parallel. We could try to parallelize the individual interaction functions to speed up the computation of individual elements. However, since the matrices are only accessed in terms of matrix operations, e.g., matrix vector products, many if not all elements are required eventually. What is more, by developing parallelization schemes for these operations rather than the elements, we can exploit the fact that the elements of the same potential matrix always share the exact same interaction function, suggesting an underlying *SIMD (single instruction, multiple data)* pattern in the process.

While we focus on matrix-vector products as the main matrix operation designated by iterative solvers, the same ideas are transferable to other operations as well. Corresponding parallelization schemes for the less complex `diag` operation, used for Jacobi preconditioning in our solvers, can be found in the CuNESSie repository. Each operation is represented as a dedicated kernel function in the device code. For these functions to be as efficient as possible, we need to consider a few aspects. Typical performance bottlenecks of CUDA applications involve memory access and data transfers. Thus, we want to keep the number and size of memory transfers between host and device at a level that does not cause a negative impact on the overall performance. The same applies to the number of memory accesses performed on the device side.

All potential matrices for the local and nonlocal BEM solvers share the exact same row and column elements. This observation implies that, in concept, it would suffice to transfer them to the device once, e.g., at the beginning of the execution, and reuse them for all operations, given that there is enough space on the device. Fortunately, this behavior can be easily achieved through the `CuVector`-based row and column element implementations presented in the last section, as long as we pass around references to the same vector objects in the code. Furthermore, this observation is completely independent of any particular matrix operation, as the row and column elements constitute the whole potential matrix (together with a suitable interaction function). Also, since the BEM system matrices are simple compositions of the same

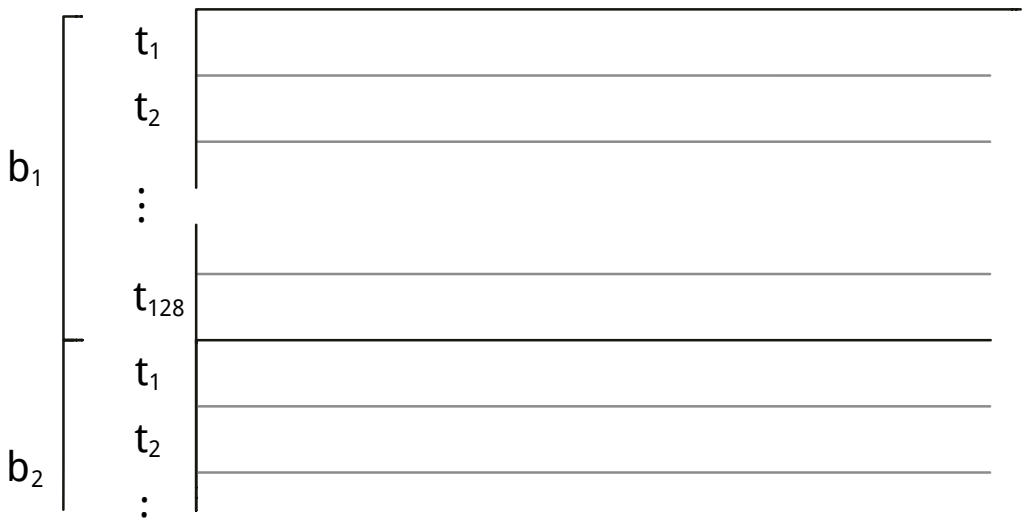


Figure 7.1. – Partition scheme used for the CUDA-accelerated matrix-vector products in CuNESSie.jl. The matrix is split into numbered blocks of 128 rows (b_i) and each row is assigned a dedicated thread (t_j).

potential matrices and some known vectors, the effective memory requirements for the implicit BEM system representations is in $\mathcal{O}(n)$, given a protein surface approximation of n triangles. Owing to this negligible memory footprint, the data required for the full solving process fits easily into a few hundred megabytes of device memory, even for large surface triangulations. This demotes memory transfers to a minor contributor to the overall runtime of our CUDA-accelerated solvers.

A single matrix-vector product requires each potential matrix element to be computed at some point. Since there is no direct data dependency between the matrix elements, we could assign each element to a separate thread of execution, causing each thread to read a single position and a single triangle from the row and column elements residing in the global device memory. Conversely, for a matrix of size n^2 each position and triangle is read by n threads, respectively. Unfortunately, in the context of matrix-vector products, this approach leads to a new data dependency between threads as each element of the result vector represents the product of the corresponding matrix row and the input vector. In other words, in order to compute a single element of the result vector, n matrix elements have to be multiplied to the corresponding input vector component and then summed up. The latter sum could be efficiently implemented through the shared memory of the device.

Alternatively, we could attempt to get rid of the new data dependency altogether and compute each result vector element with the registers of a same thread. In this scenario, each position is only read once for each matrix-vector product, along with all triangles and the full input vector. Figure 7.1 shows a partition scheme for

Listing 7.8 – CUDA-enabled matrix-vector product for interaction matrices in Julia

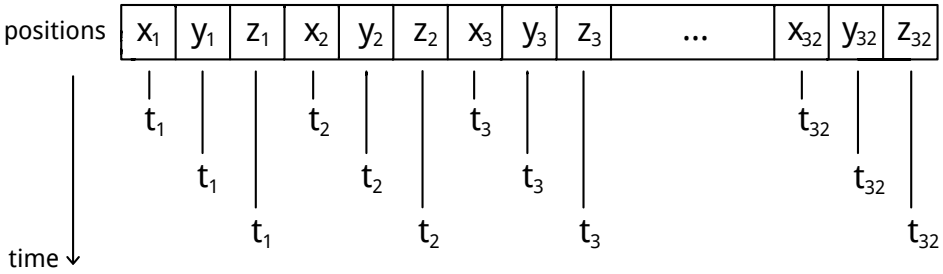
```
#= device code =#
function mv_kernel!(
    dst      ::CuDeviceVector{T},     # destination vector
    rowelems::CuPositionVector{T},    # row elements
    colelems::CuTriangleVector{T},    # column elements
    interact::F,                      # interaction function
    x        ::CuDeviceVector{T}      # vector x
) where {T, F <: Function}
    # compute matrix row
    i = (blockIdx().x - 1) * blockDim().x + threadIdx().x
    i > length(rowelems) && return

    ξ = rowelems[i]
    val = T(0)
    for j in 1:length(colelems)
        val += interact(ξ, elements[j]) * x[j]
    end
    dst[i] = val
    nothing
end
```

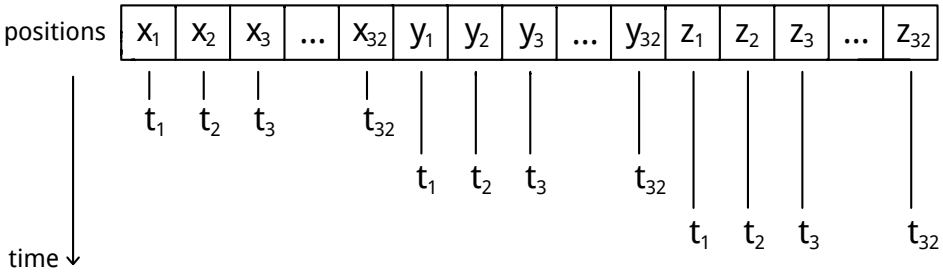
the new thread distribution. In CUDA, threads are grouped into warps and blocks (cf. Section 4.3). While warps usually consist of a fixed number of 32 threads, the number of blocks can be manually set. For reasons that will be discussed later (cf. Section 9.4.3), we assume block sizes of 128 threads, with the possible exception of the last block being smaller in cases where n is not a multiple of 128. Thus, each block houses a total of four warps.

Listing 7.8 shows a simplified implementation of a CUDA-accelerated matrix-vector product for the presented partition scheme. For the sake of simplicity, the potential matrix is here not represented using a dedicated device type for potential matrices but as its individual components (i.e., row and column elements as well as an interaction function). The kernel function represents a single thread of execution, where in the first step we determine the corresponding matrix row `i` from the current thread index, block index, and the block dimensions. Afterwards, the i-th position is read from the row elements residing in the global device memory. The remaining function body carries out the actual computation through an element-wise swipe over the matrix columns[4]. In each iteration, a single triangle and a single input vector element are read from the global memory. It should be noted that these memory accesses happen at the same time for the same objects for all threads of the current warp.

[4] In the actual kernel implementation, the separate multiply and add operations in each iteration are replaced by the *fused multiply-add (FMA)* instruction to reduce the number of involved rounding operations for the floating-point values. FMA is available through the CUDA Math API as `CUDAnative.fma`.



(a) Array-of-structs (AoS) layout



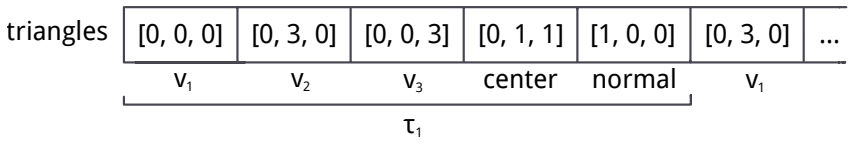
(b) Struct-of-arrays (SoA) layout

Figure 7.2. – Different memory layouts for the internal representation of position vectors

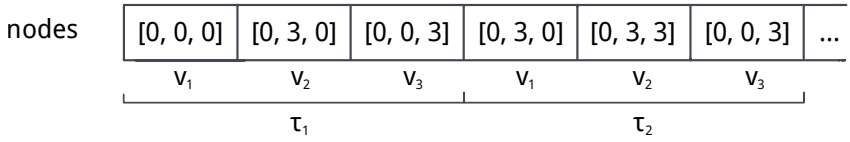
Apart from the number of memory accesses and transfers, the memory access pattern must also be accounted for in efficient CUDA code. Generally, contiguous (or *coalesced*) memory access should be preferred over random access in kernel code as it allows to efficiently reuse cached data with a minimum of memory transfers from the respective memory (e.g., global or shared). For a position vector modeled using named tuples (as shown in the last section), the coordinate values are always read in the order of their appearance in the tuple definition. In other words, reading a `CuPosition` object results in first accessing its x-coordinate, then its y-coordinate, and finally its z-coordinate. Hence, this observation suggests that the coordinates should be stored in the same pattern in the underlying `CuVector`. However, this would only be beneficial when restricting the matrix-vector product computation to a single thread. As evident from Figure 7.2a, all threads of the same warp read their corresponding position at the same time from the memory, leading to 32 x-coordinates being accessed simultaneously. Thus, instead of coalesced memory access, we have a strided memory access pattern, which is not optimal.

The storage pattern described above is commonly referred to as *array of structs (AoS)*, in which the position vector is considered an array of aggregate types (structs) representing its elements[5]. A better solution for the position vectors is to store the elements in the same way presented in the last section, that is, by splitting

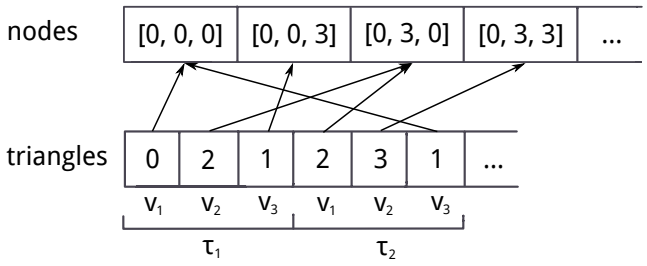
[5]We are using the term AoS loosely here because the elements are modeled as named tuples rather than classic structs. In any case, the basic idea remains the same.



(a) Flat representation



(b) Flat node-based representation



(c) Node-based representation reusing coordinates

Figure 7.3. – Different memory layouts for the internal representation of triangle vectors

the coordinates (cf. Listing 7.3). This storage pattern is commonly referred to as *struct of arrays (SoA)*, where the position vector is an aggregate type of arrays representing the separated element members. Here, the SoA layout allows us to access the position vectors in a coalesced way across the warp threads.

For the column elements, i.e., the triangle vectors, the AoS pattern turns out to be more suitable, since all threads access the same memory locations at the same time. Thus, the triangles are also accessed sequentially to the level of their members. This is mainly due to the fact that we already chose a flattened base storage pattern for triangles in the first place. Different layout options are shown in Figure 7.3, with the first two options representing the flattened representations (where derived triangle members are either stored explicitly or computed on demand).

At the other end of the spectrum, we could have opted for a representation similar to the HMO+ format (cf. Section 6.2.3), where the triangle nodes are stored separately and referred to by their respective index (third option). While this representation is clearly more space-efficient than the flattened representation, it also leads to chaotic access patterns that would significantly impair the performance of the matrix-vector product implementations. Since the memory footprint of the implicit BEM system

representations is in the range of a few megabytes (cf. Chapter 9), the flattened representation scales well enough for the methods presented here. Apart from that, the actual storage pattern is completely hidden in the getter and conversion functions of the position and triangle vectors. As a consequence, changes in the layout do not affect depending code, rendering the abstraction layer for semantically-structured device arrays a valuable means for developing CUDA code in Julia.

7.4 Solvers and post-processors

With device version of vectors and interaction matrices at hand, we can finally implement CUDA-accelerated versions of our BEM solvers and post-processors. As described in Section 7.1, the main goal is to provide drop-in replacements for the `NESSie.BEM` module. For this reason, we only briefly summarize their specific characteristics in this section while the interested reader is kindly referred to Section 6.3 for a more general interface description.

7.4.1 CUDA-accelerated solvers

On the surface, the BEM solvers in CuNESSie behave exactly like their NESSie counterparts: They assemble the linear systems from the same potential matrices and subsequently solve them through the restarted GMRES implementation of the *IterativeSolvers.jl* package, using a Jacobi preconditioner from the *Preconditioners.jl* package, and provide the exact same input and output interface via the NESSie system model and the `BEMResult` types, respectively (cf. Section 6.3.3). The only difference is the fact that the systems are now represented in such a way that all matrix (and some vector) operations are performed on the GPU rather than the CPU. This is due to additional host types for the local and nonlocal system matrices with overloaded functions, which invoke the corresponding CUDA kernels. While the focus on matrix operations implies that the vast majority of instructions from the GMRES implementation are still executed on the CPU (and in a single thread of execution at that), the involved operations are comparatively cheap to compute and, in terms of runtime, fully dominated by the optimized matrix operations. For the latter operations, we use the device types and kernel functions presented in the last sections. In addition, all vector operations within the context of these matrix operations are (implicitly) parallelized through the `CuVector` type we use for floating-point vectors.

7.4.2 CUDA-accelerated post-processors

While NESSie provides solvers for both explicit and implicit system representations, the post-processors exclusively use the explicit representations as a basis. This is usually not an issue since potential matrices in the post-processors are only of size $1 \times n$ for each individual observation point and a surface approximation of n triangles (cf. Section 3.4). Even if the derived quantities are computed for several observation points at once, the corresponding matrices can be split into manageable chunks, allowing almost fine-grained control over the memory footprint of the post-processors. On the other hand, reusing the CUDA-accelerated potential matrices for similarly-optimized post-processors is trivial. By replacing the row elements with the observation points passed to the respective post-processor, the very same potential matrix types can be utilized as for the solvers. For this reason, CuNESSie ships with CUDA versions of all previously presented post-processor functions, namely, those for the computation of electrostatic potentials everywhere in space as well as reaction field energies.

8 AnyBEM

In this chapter, we present the *AnyBEM* project, our alternative implementation for the Julia-based protein electrostatics problem with implicit BEM system representations. AnyBEM exploits the domain-specific properties of the Impala programming language. While the underlying ideas of the implemented data structures and functions remain the same as developed over the course of this manuscript and implemented in NESSie and CuNESSie, we focus on the domain-specific abstraction layers newly introduced to the problem. These layers allow the implementation of solvers in a fully transparent way for different platforms, different degrees of parallelization, and different levels of floating-point precision.

8.1 Package organization

The AnyBEM package can be divided into different components: Its core part is represented by the *libanybem* library[1], which provides data types and operations for the nonlocal protein electrostatics problem in Impala. Built on top of that are tests, to ensure that the library components work as intended, and a command-line interface, implemented as a separate application *anybem* to provide easy access to libanybem functionality. The library can further be split into an Impala part and a C++ part. The former part contains the majority of the program logic, including buffer-based vector types and a minimalistic math library for position vectors (cf. Section 5.5.1) as well as the implicit system representations and domain-specific matrix operations. The C++ part serves as an interface between Impala and client applications and provides basic HMO+ file support (cf. Section 6.2.3). Together, the library comprises the full infrastructure to generate implicit representations of the local and nonlocal BEM systems. Unfortunately, at the time of writing, there are no iterative solver implementations natively available for Impala that could be used to solve the systems. We have, however, discussed in Section 5.6 how to incorporate such libraries in the future.

[1] Although *AnyBEM* technically refers to the package as a whole, we commonly use the name as an alias for libanybem in this manuscript as the other components do not add any new functionality.

Listing 8.1. – Example for cross-language function calls in C++ (left) and Impala (right)

```
extern "C" {
    int times_two(int i) {
        return 2 * i;
    }

    int main(int, char**);
}
```

```
extern "C" {
    fn times_two(i32) -> i32;
}

extern
fn main(argc: i32, &[&[u8]]) -> i32 {
    let number = times_two(argc) + 1;

    print_string("Heya! Have a number: ");
    print_i32(number);
    print_string("\n");
    0 // EXIT_SUCCESS
}
```

8.2 Cross-language features

The remarkable interoperability between C/C++ and Impala is not a characteristic feature of AnyBEM but rather common to most Impala applications. Generally, this strong connection serves two purposes: Firstly, exposing Impala functions through a suitable C/C++ interface facilitates the incorporation of Impala programs into other projects. Once compiled into a C/C++ library, a corresponding Impala program can be treated just like any other C/C++ library and linked against by its consumers[2]. Secondly, the bidirectional nature of the language interconnection allows for the splitting of core functionality, so that each function of the program can be implemented in the language suited best for its purpose. Typical use cases outsourced to C++ include I/O and string manipulations or exception handling.

8.2.1 Cross-language interface

In AnyBEM, almost all solver-related functionality is implemented in Impala rather than C++ to maximally benefit from the language's partial evaluation mechanism, while utility routines (e.g., input data formats) reside in the C++ part of the package. Hence, the AnyBEM cross-language interface mostly comprises native Impala functions intended for use by consumer code. Currently, this includes functions to print the contents of system models read from HMO+ files (after transfer to the Impala part of the package) and the benchmarking facility used for performance assessments (see Section 9.4.3). In the future, we plan to expose the full BEM solver interface in the same way. In addition to these Impala-specific functions, libanybem also exposes the HMO+ input format routines, making the

[2] This restricts the library functionality to a specific domain and to the functions exposed through its public interface as the partial code evaluation is performed when building the library. For maximum flexibility, Impala code can alternatively be incorporated into consuming Impala code directly.

Listing 8.2 – Example output for an Impala application with cross-language function calls

```
1 shell> ./hello_impala
2 Heya! Have a number: 3
3 shell> ./hello_impala nice to meet you
4 Heya! Have a number: 11
```

library's public interface transparent in the sense that it is not visible to the consumer whether a function actually belongs to the Impala or the C++ part of the package.

The communication between C/C++ and Impala usually happens on a per-function basis, with potential function parameters of data types that possess equivalent representations in both languages. The latter mainly describe fundamental types such as Boolean types, signed and unsigned fixed-width integer types (available in C and C++ through the `stdint.h` and `cstdint` headers, respectively), single- and double-precision floating-point numbers as well as pointers and definite or indefinite arrays of these types. Overall, the cross-language interface is effectively a classic C interface on both sides (with the necessary modifications on the Impala side). A minimalistic example application for fluent bidirectional cross-language function calls is shown in Listing 8.1. The main program is declared in the C++ part of the code and defined in the Impala part, printing a number corresponding to one plus twice the number of specified command-line arguments. The multiplication is in turn performed by a function that resides in the C++ part of the code. The resulting program in action can be reviewed in Listing 8.2, where the code from Listing 8.1 has been compiled into an executable named `hello_impala`.

8.2.2 Cross-language system models

In addition to fundamental types, aggregate types can also be used at the cross-language boundary (i.e., C structs or, equivalently, C++ standard-layout types). This allows us to transfer semantically-structured data between the languages, which represent an integral part of the AnyBEM system models. This is due to the HMO+ format being implemented in the C++ part of our package, resulting in the input data always being read in on the C++ side and subsequently being transferred to the Impala side. For this, we provide a dual implementation of the system models in AnyBEM, with equivalent types in both languages that can be easily exchanged by the cross-language interface while maintaining the same underlying data structures.

Each system model consists of a surface model, a charge model, and a set of system constants. The basic structure of these components is the same as in NESSie (cf.

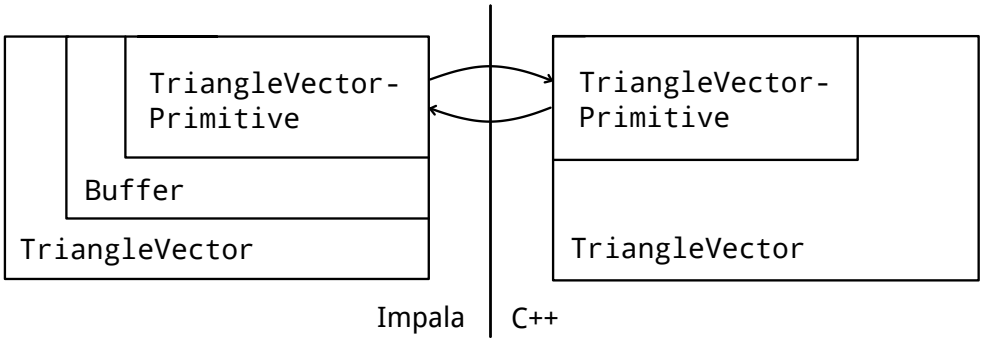


Figure 8.1. – Primitive-based triangle vectors for C++ and Impala

Section 6.2), albeit more restrictive in the sense that only triangulated surfaces are currently supported in AnyBEM. We have seen Impala-compatible data types for the surface models and system constants already in the form of `Triangle`, `TriangleVector`, and `SystemParams` (cf. Sections 5.5.1 and 5.5.4). Respective types could be defined for point charges and charge vectors in the same way. However, while `SystemParams` is modeled as a simple aggregate type that could directly be used as part of the cross-language system models, `Triangle` and `TriangleVector` are defined as interface types. This purely function-based kind of representation bears many advantages (see Section 5.5.1 for a discussion) but it is less suitable for cross-language data types.

Instead, we create new types for the internal representation of triangles, charges, and vectors thereof in terms of simple aggregate types, henceforth referred to as *primitives*. These primitives can then later be wrapped into the existing interface types on the Impala side. We have also seen how this is done to create vector objects from buffers. Hence, by creating simple representations for charge and triangle vectors that are both compatible to buffers and can be used as cross-language types, we can read these model components in once, store them in said representation, and wrap them into buffer objects and charge/triangle vectors as needed without copying or moving data.

Similar primitive-based wrapper types can then be added to the C++ side as well to allow convenient access to the underlying data everywhere. Figure 8.1 shows a schematic representation of this idea for the particular case of triangle vectors. Using buffers here as intermediate representations grants the additional bonus of compatibility with Impala's built-in memory management functions (e.g, to be copied directly into device memory).

Listing 8.3 shows a working example for some system model components, both in the C++ and Impala version. We continue to use a type alias `Real` for the precision

Listing 8.3. – Cross-language system model components for C++ (left) and Impala (right)

```
using Real = double;

struct Position {
    Real x;
    Real y;
    Real z;
};

struct ChargePrimitive {
    Position pos;
    Real     val;
};

struct TrianglePrimitive {
    Position v1;
    Position v2;
    Position v3;
};

struct TriangleVectorPrimitive {
    TrianglePrimitive* data;
    int64_t            size;
};
```

```
type Real = f64;

struct Position {
    x: Real,
    y: Real,
    z: Real
}

struct ChargePrimitive {
    pos: Position,
    val: Real
}

struct TrianglePrimitive {
    v1: Position,
    v2: Position,
    v3: Position
}

struct TriangleVectorPrimitive {
    data: &[TrianglePrimitive],
    size: i64
}
```

level of floating-point values, which can be exchanged later. Furthermore, we reuse our previous `Position` type for position vectors in $\mathbb{R}^3$. The new primitive types are then defined in terms of `Position` and other standard-layout types, where charges carry a position and a charge value and triangles are stored as the positions of their respective vertices. The primitives for vectors take a special place here as they mimic the internal structure of buffers. More specifically, buffers store their data as pointers to indefinite byte arrays (`&[i8]`). By recreating a similar representation for our vectors, these types can be converted by means of static type casts (cf. Section 5.5.1). This procedure is repeated for all system model components and the system model type itself.

8.3 Domain specialization

A key property of domain-specific languages is, as the name suggests, the specialization of code to particular application domains. In Impala, this specificity primarily refers to a particular computational platform (e.g., processors and hardware accelerators) and a corresponding software back end (e.g., CUDA or OpenCL). The domain specificity is typically realized through multiple layers of indirection in the Impala code. More precisely, these abstraction layers make the same functionality available for each supported domain by means of a common function interface while rendering the currently targeted domain fully transparent to its consumer.

Listing 8.4. – Impala's `Intrinsics` type for platform-specific math routines (left) and pre-defined instances for different platforms (right)

```
struct Intrinsics {
    sqrt:  fn(f64)      -> f64,
    sqrtf: fn(f32)      -> f32,
    min:   fn(i32, i32) -> i32,
    fmin:  fn(f64, f64) -> f64,
    fminf: fn(f32, f32) -> f32,
    log:   fn(f64)      -> f64,
    // ...
}
```

```
static cpu_intrinsics = Intrinsics {
    sqrt:  cpu_sqrt,
    sqrtf: cpu_sqrtf,
    // ...
};

static cuda_intrinsics = Intrinsics {
    // ...
};
```

8.3.1 Built-in platform abstractions

The AnyDSL project provides three platform abstractions for Impala through the AnyDSL runtime library: buffers, intrinsics, and accelerators. We have introduced buffers in Section 5.5.1 as Impala's generic data containers and used them numerous times as such. However, when taking a closer look at the definition of the `Buffer` type, it becomes clear that the data is only referenced by a memory address and that the actual memory location is captured in the form of a numeric identifier, representing both the compute platform and the physical device corresponding to said location. Thus, buffers are not only mere generic data containers but rather platform abstractions for containers of generic data, extended by functions to uniformly manage memory and to transfer data between supported platforms.

The second built-in abstraction layer is Impala's `Intrinsics` interface type, which provides uniform access to domain-specific implementations of various math routines. An excerpt from its type definition is shown in Listing 8.4 alongside a few examples of concrete `Intrinsics` instances pre-defined in Impala. Most operations are available for different value types (e.g., `min` for integers or `fmin` and `fminf` for double- and single-precision floating-point numbers, respectively). The different `Intrinsics` instances can then be used to request a specific platform (e.g., `cpu_intrinsics` for the CPU or `cuda_intrinsics` for CUDA) or transparently through a global constant set according to the current application domain.

The third abstraction layer serves as a generic interface for different hardware accelerators. More specifically, the `Accelerator` interface type represents parallel execution units (so-called *work items*) with structured partition schemes. The collective work items need to be organized into groups of equal size, representing a single grid of groups, with each layer supporting up to three dimensions. The `Accelerator` type can then be used to start or synchronize the parallel execution of a given grid or to allocate accelerator-specific memory.

Listing 8.5. – Realization of domain-specific math routines through mutually-exclusive definitions in separate files

```
// file: intrinsics_cpu.impala
static math = cpu_intrinsics;
```

```
// file: intrinsics_cuda.impala
static math = cuda_intrinsics;
```

Listing 8.6 – Example for a function to compute the Euclidean distance between two position vectors using domain-specific math routines

```
fn euclidean(u: Position, v: Position) -> Real {
    let dx = u.x - v.x;
    let dy = u.y - v.y;
    let dz = u.z - v.z;
    math.sqrt(dx * dx + dy * dy + dz * dz) // <- caution!
}
```

8.3.2 Floating-point precision

All of the presented platform abstractions are used to provide support for CPU- and CUDA-based computations in AnyBEM. In practice, most of these abstractions are realized through global-scope constants with domain-specific definitions of the same interface in separate source files. Then, only the source files corresponding to the chosen target domain are included during the building process of the Impala program. Listing 8.5 shows a simple example for such a constant, where `math` is defined either as the CPU- or CUDA-specific `Intrinsics` object. By subsequently including exactly one version of the definition in the building process, the `math` constant always refers to the functions corresponding to the target domain.

This technique enables the implementation of depending functions in the same domain-specific fashion. Listing 8.6 shows an example for such a function to compute the Euclidean distance between two points in $\mathbb{R}^3$ given as `Position` objects. Due to the use of the same `math` constant, the function is also specific to the target domain. It should be noted, however, that the shown implementation strictly requires `Real` to represent double-precision floating-point numbers when combined with the definitions from Listing 8.5.

In order to circumvent this issue while still leaving the option to support both floating-point precision levels, we extended the `Intrinsics` interface in AnyBEM. More specifically, math functions are now provided through a new `RealMath` type, with each function being defined in terms of the `Real` alias instead of `f32` or `f64` directly. The alias can then be set, e.g., via

```
type Real = f32;
```

Listing 8.7. – Domain-specific math routines for different platforms (CPU or CUDA) and floating-point precision levels (single- or double-precision)

```
// file: intrinsics_cpu.impala
static intrinsics = cpu_intrinsics;
```

```
// file: intrinsics_cuda.impala
static intrinsics = cuda_intrinsics;
```

```
// file math_f64.impala
type Real = f64;

struct RealMath {
    sqrt: fn(Real)       -> Real,
    min:  fn(Real, Real) -> Real,
    log:  fn(Real)       -> Real,
    // ...
}

static math = RealMath {
    sqrt: intrinsics.sqrt,
    min:  intrinsics.fmin,
    log:  intrinsics.log,
    // ...
};
```

```
// file math_f32.impala
type Real = f32;

struct RealMath {
    sqrt: fn(Real)       -> Real,
    min:  fn(Real, Real) -> Real,
    log:  fn(Real)       -> Real,
    // ...
}

static math = RealMath {
    sqrt: intrinsics.sqrtf,
    min:  intrinsics.fminf,
    log:  intrinsics.logf,
    // ...
};
```

Listing 8.8 – Modified real-valued vector type for Impala that exposes its underlying buffer

```
struct RealVector {
    size: fn()          -> i32,
    get:  fn(i32)       -> Real,
    set:  fn(i32, Real) -> (),
    buf:  fn()          -> Buffer
}

fn as_real_vector(buf: Buffer) -> RealVector {
    RealVector {
        size: ||          (buf.size as i32) / sizeof[Real](),
        get:  |idx|       bitcast[&[Real]](buf.data)(idx),
        set:  |idx, val|  bitcast[&mut[Real]](buf.data)(idx) = val,
        buf:  ||          buf
    }
}
```

for single precision similar to the `math` constant above. In addition, the domain-specific instantiations of `RealMath` are defined in terms of the original `Intrinsics` objects such that the supported platforms and precision levels can be chosen independently from one another. Listing 8.7 shows the corresponding Impala code for `RealMath`. By defining `math` as `RealMath` object instead of `Intrinsics`, our previous function for Euclidean distances now fully supports CPU- and CUDA based contexts as well as single- and double-precision floating-point values.

8.3.3 Cross-platform data transfers

The second AnyBEM-specific platform abstraction reuses the previously presented context managers (cf. Section 5.5.2) to make the data transfer between platforms

Listing 8.9. – Example for cross-language function calls in C++ (left) and Impala (right)

```
// file: platform_cpu.impala
fn real_vector_on_platform(
    vec:  RealVector,
    body: fn(RealVector) -> ()
) -> () {
    body(vec);
}
```

```
// file: platform_cuda.impala
fn real_vector_on_platform(
    vec:  RealVector,
    body: fn(RealVector) -> ()
) -> () {
    let buf = alloc_cuda(0, vec.size());
    copy(vec.buf(), buf);
    body(as_real_vector(buf));
    copy(buf, vec.buf());
    release(buf);
}
```

transparent. With CPUs and CUDA, AnyBEM supports two structurally different platforms. Typically, input data is initially read into host memory, no matter the actual target platform, where it can be directly accessed and processed in CPU-based contexts. However, when delegating parts of the computation to hardware accelerators, such as CUDA-enabled GPUs, the data needs to be transferred onto the device before it is processed any further. Afterwards, the resulting data is usually copied back from the device into the host memory. Context managers elegantly hide this additional data transfer, but first we need to modify the definitions of our previously presented vector types for semantically-structured data. Since Impala's memory management facilities operate on buffers, the vector types need to expose their underlying data, before they can be copied between platforms. The necessary modifications for our `RealVector` type (formerly presented in Section 5.5.1) are shown in Listing 8.8.

The basic idea behind the data transfer abstraction is to pass a prepared host vector to a special context manager that creates a new vector object representing the same data on the target platform. Listing 8.9 shows example implementations of such context managers for the `RealVector` type on the CPU or CUDA-enabled GPUs. In the former case, the context manager is an identity function and simply executes the body of the `with` block, since no data needs to be copied in this particular case. For CUDA, a new device vector of the same size as the given host vector is created before the data is copied onto the device and the `with` block body is executed. Afterwards, the device vector is automatically copied back into the buffer underlying the original host vector. Consumer code can then use the context managers in the same pattern to ensure that the data at hand is present on the target platform:

```
let host_vec = /* a RealVector in host memory */;
with platform_vec in real_vector_on_platform(host_vec) {
    // some platform-specific code
} // <- platform_vec is copied back into host_vec
```

Listing 8.10. – Domain-specific loop parallelization in Impala

```
// file: eachindex_serial_cpu.impala
fn each_index_on_platform(n: i32, body: fn(i32) -> ()) -> () {
    for i in range(0, n) {
        body(i)
    }
}
```

```
// file: eachindex_parallel_cpu.impala
fn each_index_on_platform(n: i32, body: fn(i32) -> ()) -> () {
    for i in parallel(0, 0, n) {
        body(i)
    }
}
```

```
// file: eachindex_parallel_cuda.impala
fn each_index_on_platform(n: i32, body: fn(i32) -> ()) -> () {
    let block = (128, 1, 1);
    let grid = (cld(n, 128) * 128, 1, 1);
    let acc = cuda_accelerator(0);

    for item in acc.exec(grid, block) {
        let i = item.bidx() * item.bdimx() + item.tidx();
        if i < n {
            body(i)
        }
    }
    acc.sync();
}
```

8.3.4 Parallel `for` loops

The last abstraction layer allows to automatically parallelize the iterations of a `for` loop similar to the `@threads` macro in Julia (cf. Sections 4.1.2 and 6.3). In Impala, `for` loops are modeled as expressions and behave similarly to the `with` expressions we encountered in the last section. In contrast to the latter, `for` expressions usually encapsulate other loops and call the `body` function repeatedly in the process.

Listing 8.10 shows three example implementations for a `for` loop construct that generates indices between zero and a given number `n`. In the first case, the corresponding function is an alias for a call to Impala's `range` function with the same effect, by fixing its first argument (i.e., the starting value) to zero. For example, when used as

```
for i in each_index_on_platform(10) {
    print_i32(i);
}
```

the code will intuitively print all integers between zero and nine. The other implementations mimic the same behavior, parallelized and for different platforms, so that the same code snipped can be used for any supported platform. More specifically, in

Listing 8.11 – Domain-specific matrix-vector product for potential matrices in Impala

```
fn mv_pot(A: PotentialMatrix, x: RealVector, y: RealVector) -> () {
    let (rows, cols) = A.size();
    for i in each_index_on_platform(rows) {
        let mut sum = 0 as Real;
        for j in range(0, cols) {
            sum += A.get(i, j) * x.get(j);
        }
        y.set(i, sum);
    }
}
```

the second implementation, the loop iterations are distributed evenly[3] over all of the CPU's threads of execution by means of the *Intel Threading Building Block (TBB)* library[4]. The library is directly available through Impala's `parallel` function. The third implementation in Listing 8.10 corresponds to a CUDA kernel call, with a fixed partition scheme of 128 threads per block[5]. The `body` function serves as a kernel function and is only called if the computed index is valid.

8.4 Matrix-vector products

Using the same definitions for potential matrices and BEM systems developed earlier in this manuscript (cf. Sections 5.5.3 and 5.5.4), we obtain domain-specific implementations simply by adding further data transfer context managers for position and triangle vectors, that is, the basic components of these matrices and systems. What is more, with the parallel `for` loop constructs from the previous section, we are finally able to provide domain-specific matrix-vector products for the presented matrices. For the potential matrices in particular, we utilize the same approach and partition scheme as in CuNESSie (cf. Section 7.3.2) and compute the products row-wise. The corresponding Impala code for this function is given in Listing 8.11. On the CPU, the rows are once again distributed evenly over all available threads of execution while on CUDA-enabled devices, each thread is assigned exactly one matrix row. The same code is used for any combination of supported platforms and floating-point precision levels due to the abstraction layers implemented in AnyBEM. For the BEM matrices, which are composed of the same potential matrices, we then simply reuse these product implementations for their own domain-specific matrix-vector products.

[3] This is due to the first argument of the `parallel` function being set to zero. The number of threads can alternatively be specified manually through a nonzero value.
[4] `https://software.intel.com/en-us/tbb`
[5] The operation `cld(a,b)` represents the result of $\lceil a/b \rceil$ and is defined separately.

9 Performance analysis

In this chapter, we evaluate the performance of the data types, matrix-vector products, and BEM solvers implemented in NESSie, CuNESSie, and AnyBEM. For this, we give details about the hardware and software environments used for the generation of all results. In addition, we provide an overview of all datasets involved in the analyses and outline the data preparation process. The actual performance assessments are then split into three categories, where we first evaluate the BEM solvers by means of comparison to simple, analytically-solved test models. Afterwards, we evaluate the runtimes of the sequential and parallel matrix-vector product implementations for the potential matrices as well as for the BEM system matrices. Finally, we repeat the assessment for all BEM solvers presented earlier in this manuscript.

9.1 Experimental setup

All performance analyses presented in this chapter are based on data generated on two different workstations. The technical specifications of both machines are summarized in Table 9.1. The first machine represents a typical commodity workstation while the second machine is an above-average workhorse for computing tasks. What is more, we have chosen two state-of-the-art GPUs from the consumer branch and the general-purpose computing branch of the CUDA device market. We briefly summarize the compute features of these two devices in the next paragraphs. In order to reduce data noise and increase the quality of GPU-based performance results, said devices come in addition to a primary GPU in each machine and are used exclusively for the computations presented here.

The *NVIDIA GeForce RTX 2070 SUPER* GPU, henceforth referred to as *RTX 2070 SUPER*, was developed with a focus on computer graphics applications in mind and enhanced with dedicated ray-tracing hardware and tensor cores for NVIDIA's *Deep Learning Super Sampling 2.0* (DLSS 2.0) platform. For this reason, the device does only provide rudimentary support for double-precision floating-point (FP64) operations in contrast to an excellent support for single-precision (FP32) operations. More specifically, the FP64 hardware units are outnumbered by their single-precision

	Workstation 1	Workstation 2
CPU		
◦ Type	Intel Core i7-4790K	2x Intel Xeon E5-2683 v4
◦ Clock speed	4.0 GHz	2.1 GHz
◦ Physical cores	4	32 (2x 16)
◦ Threads	8	64 (2x 32)
Memory		
◦ Type	2x DIMM DDR 3	16x RIMM
◦ Capacity	16 GiB (2x 8 GiB)	256 GiB (16x 16 GiB)
◦ Clock speed	1.6 GHz	2.4 GHz
GPU†		
◦ Type	NVIDIA GeForce RTX 2070 SUPER	NVIDIA TITAN V
◦ Architecture	Turing	Volta
◦ Clock speed	1.605 GHz	1.2 GHz
◦ Memory	8 GB (GDDR 6)	12 GB (HBM2)
◦ Memory bandwidth	448 GB/s	652.8 GB/s
◦ Compute capability	7.5	7.0
◦ CUDA cores (FP32)	2,560	5,120
◦ FP32 vs. FP64 units	32:1	2:1
Software		
◦ Operating system	Gentoo Linux	Ubuntu 16.04 LTS
◦ GCC version	9.3.0	7.3.0
◦ NVIDIA driver version	440.82	440.64.00

Table 9.1. – Technical specifications of the machines used for all performance analyses. Features specific to single-precision (FP32) or double-precision (FP64) floating point operations are marked accordingly. † The listed GPUs are used exclusively for the computational tasks presented here. Each machine is equipped with a separate GPU (not listed) for all other purposes.

Our own packages		Third-party Julia packages		AnyDSL components	
Name	Version	Name	Version	Name	Commit
AnyBEM	0.2.0	CuArrays.jl	2.2.0	AnyDSL	`d9fd666`
CuNESSie.jl	0.2.0	CUDAapi.jl	4.0.0	Impala	`ecb1452`
ImplicitArrays.jl	0.2.0	CUDAdrv.jl	6.3.0	Runtime	`4384228`
NESSie.jl	1.2.0	CUDAnative.jl	3.1.0	Thorin	`4c7b44b`
		IterativeSolvers.jl	0.6.4		
		Preconditioners.jl	0.3.0		

Table 9.2. – List of exact package versions used for the performance analyses. All third-party Julia packages have been installed from Julia's official default package registry (`https://github.com/JuliaRegistries/General`). AnyDSL component versions are specified as short SHA-1 git commit identifiers relating to the `master` branch of the respective repositories (`https://github.com/AnyDSL`).

counterparts by a factor of 32. As a result, the double-precision variants of our linear system solving attempts cannot be considered to belong to the device's target group. While we still can fully utilize the same FP32 units for our single-precision system models as for any other graphics-based task, we do not expect to achieve competitive performance results for the double-precision models with this device.

The *NVIDIA TITAN V* GPU, in the following shortened to *TITAN V*, is based on the slightly older Volta microarchitecture. While the latter might technically be considered a predecessor to the Turing microarchitecture, which the RTX 2070 SUPER is based on, the Volta architecture has a stronger focus on artificial intelligence tasks and is not targeted at the consumer market. Instead, the TITAN V aims at general-purpose computing on GPUs (GPGPU), thus rendering it a compelling choice for our intents. As compared to its competitor, the device does not only ship with twice as many single-precision CUDA cores but also with a remarkable support for double floating-point precision (namely, one FP64 unit per two FP32 units).

Both workstations utilize Julia version 1.4.0 for NESSie and CuNESSie, while AnyBEM was build in Release mode using the respective GNU Compiler Collection (GCC) version specified in Table 9.1. The exact versions of the relevant Julia packages and AnyDSL components are further listed in Table 9.2.

9.2 Experimental data

In the following, we present the datasets used for the performance analyses throughout this chapter. Most datasets are directly available through the NESSie package or the NESSie code repository. Where possible, we give instructions on how to recreate the datasets, including sources as well as the used tools and parameters.

9.2.1 Born ions

For the basic evaluation of the local and nonlocal BEM solvers in terms of the Born test model (cf. Section 6.4.1), we use the full set of Born ions directly implemented in the `NESSie.TestModel` module of our Julia package. Each Born ion is modeled as a point charge ($+1e_c$ or $+2e_c$, with e_c being the elementary charge) at the center of a spherically-symmetric domain located at position $P_0 = (0, 0, 0)$. Table 9.3 lists all available monovalent and divalent Born ions along with their implemented sphere radii. The radii are taken from [49, Table 3.4] and were, in turn, computed based on simulations from [150].

Ion	Charge [e_c]	Radius [Å]	Ion	Charge [e_c]	Radius [Å]
Li^+	$+1$	0.645	Mg^{2+}	$+2$	0.615
Na^+	$+1$	1.005	Ca^{2+}	$+2$	1.015
K^+	$+1$	1.365	Sr^{2+}	$+2$	1.195
Rb^+	$+1$	1.505	Ba^{2+}	$+2$	1.385
Cs^+	$+1$	1.715			

Table 9.3. – List of Born ions built into the `NESSie.TestModel` module, shown here with point charge values (given as multiples of the elementary charge e_c) and sphere radii (taken from [49, 150])

The approximations of the surface domain Γ for each ion were generated via GAMer's `generateBoundingSphere` function [139], or, equivalently, via NESSie's *Sphere* tool (`sphere_quality`=4, cf. Section 6.2.2). This resulted in surface meshes with 512 triangles for each ion. All meshes are available in the NESSie repository.

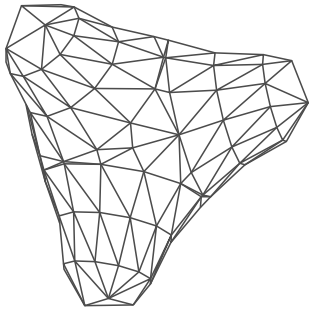
9.2.2 Small objects and molecules

The next dataset category intended for usage with the BEM solvers is a collection of small surface meshes, with a selection thereof being shown in Figure 9.1. The datasets comprise 53 surface meshes of simple geometries and small molecules, including amino acids and alcohols. The mesh sizes range from 132 to 3,540 triangles, with an average size of 946 triangles. Additionally, the data contains four protein-based surface meshes with resolutions between 10,044 and 12,668 triangles.

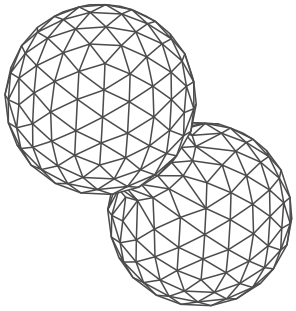
The meshes and corresponding charge models were originally generated in the same HMO+ format described in Section 6.2.3. In this form, they were used for testing purposes of the prototypical reference program [61] a part of NESSie is based on (cf. Chapter 6). Here, we reuse the meshes in their original forms exclusively for the comparison between the reference program and the NESSie BEM solvers for the explicit system representations.

9.2.3 Macromolecules

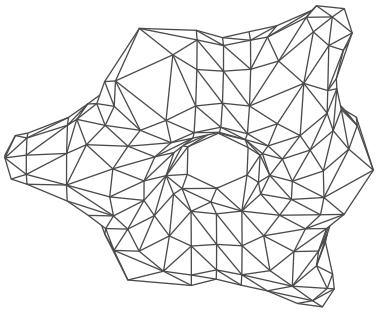
For the performance analyses of our sequential and parallel matrix-vector product implementations and the actual BEM solvers, we use different surface meshes based on proteins and protein complexes from the Protein Data Bank (PDB) [19]. Since the focus of this manuscript is the scalability of the presented methods rather than a



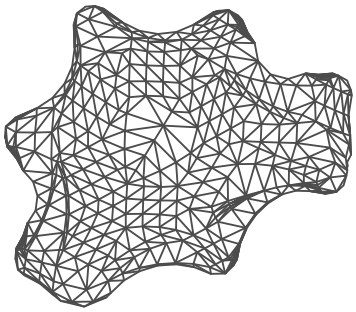
(a) NH_4 (316 triangles)



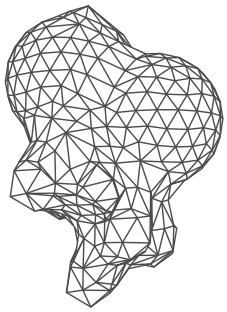
(b) Na_2 (628 triangles)



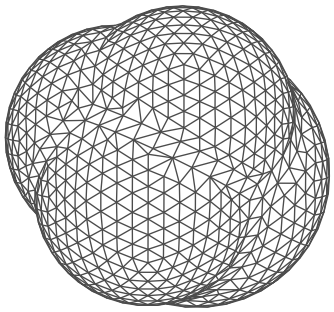
(c) Pyridinium (498 triangles)



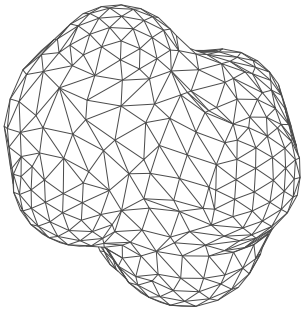
(d) 3-Methylpyridine (1728 triangles)



(e) Glutamine (840 triangles)



(f) Serine (2132 triangles)



(g) Ethanol (1077 triangles)

(h) Hexanol (1478 triangles)

Figure 9.1. – Wireframe representations of small surface meshes used for the evaluation of the BEM solvers for the explicit system representations.

PDB ID	Chains	Proteins	Atoms
2LZX	A	Asteropsin B	488
2PTC	E	Beta-trypsin	1,629
1C4K	A	Ornithine decarboxylase	5,826
6DS5	A-K	Seipin	13,662
2H1L	A-F, M-R	Large T antigen, cellular tumor antigen p53	26,948

Table 9.4. – Prepared PDB structures [19] used as a basis for the surface mesh generators. Shown here are the chains and proteins left after data preparation along with their corresponding atom counts. All data refer to the *first biological assembly* of each PDB structure.

Name	Triangles	`max-iterations`	`iso-value`	`mesh-nodes` v
`2lzx-5k`	4,978	6	2.5	1k $\leq v \leq$ 8k
`2lzx-20k`	19,912	6	2.5	5k $\leq v \leq$ 40k
`2lzx-80k`	79,648	6	2.5	10k $\leq v \leq$ 80k
`2ptc-9k`	9,464	6	2.5	1k $\leq v \leq$ 8k
`2ptc-19k`	18,556	6	2.5	5k $\leq v \leq$ 40k
`2ptc-74k`	74,224	6	2.5	10k $\leq v \leq$ 80k
`1c4k-15k`	15,208	6	1.4	1k $\leq v \leq$ 8k
`1c4k-60k`	60,356	6	1.4	5k $\leq v \leq$ 40k
`6ds5-170k`	170,252	20	1.4	10k $\leq v \leq$ 500k
`2h1l-200k`	200,580	20	1.4	10k $\leq v \leq$ 500k

Table 9.5. – List of protein surface meshes used for the performance analyses along with the parameters used to generate them via the GAMer-based [139] *Mesher* tool shipped with NESSie.jl. Each dataset name refers to the corresponding PDB structure and number of surface triangles.

biological interpretation of the results, we generally prefer structures that have been targeted by other protein electrostatics studies in the past. In order to sufficiently cover the scalability aspect of the evaluation, we further require the collected datasets to contain surface meshes with a varying number of triangles, as these numbers directly influence the size of the corresponding BEM system matrices.

Since commonly targeted proteins tend to be comparatively small in terms of their atom counts, the mesh sizes are mainly influenced by variations of the meshing resolution, that is, the level of details required to be contained in the approximation of the protein surface. For an extensive evaluation, we utilize both meshes for structures of varying size as well as different resolutions for meshes of the same structures. In particular, this allows us to capture the effects of different proteins and different levels of detail on the solving process.

Table 9.4 shows the PDB structures serving as a basis for all protein surface meshes involved in the performance analyses. As indicated above, the table includes small proteins in the range of a few hundred to a few thousand atoms targeted by previous

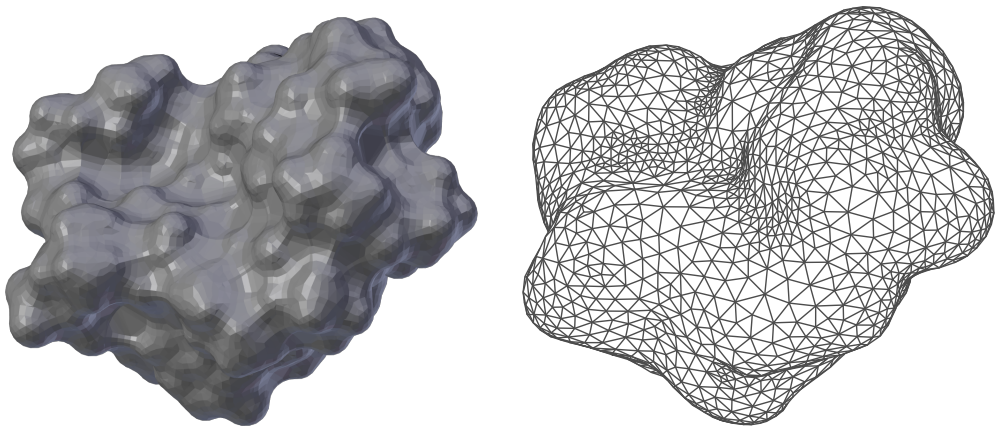
studies, namely, 2LZX, 2PTC, and 1C4K (e.g., in [49, 56, 153]). In addition, the table contains two structures (6DS5 and 2H1L) with a larger number of atoms. Each structure was obtained through the BALLView application [30] and subsequently filtered for the proteins and chains (corresponding to the "first biological assembly" included in the PDB structures) shown in the same table. The processed structures were then exported to new PDB files and fed into the GAMer-based [139] mesh generator shipped with NESSie (cf. Section 6.2.2), using GAMer version 1.5 and the parameters shown in Table 9.5 to generate a total of ten datasets.

Afterwards, each surface mesh was converted to STL format using NESSie and imported into Blender [144] for both a visual inspection and systematic checks from the "3D Printing" menu. These checks were performed to rule out degenerated or intersecting triangles as well as sharp edges that could cause issues with the solvers. Blender was further used to flip the (GAMer-provided) inward-facing unit normal vectors of the meshes to match the prerequisites of NESSie system models. The final meshes were then exported again as STL files.

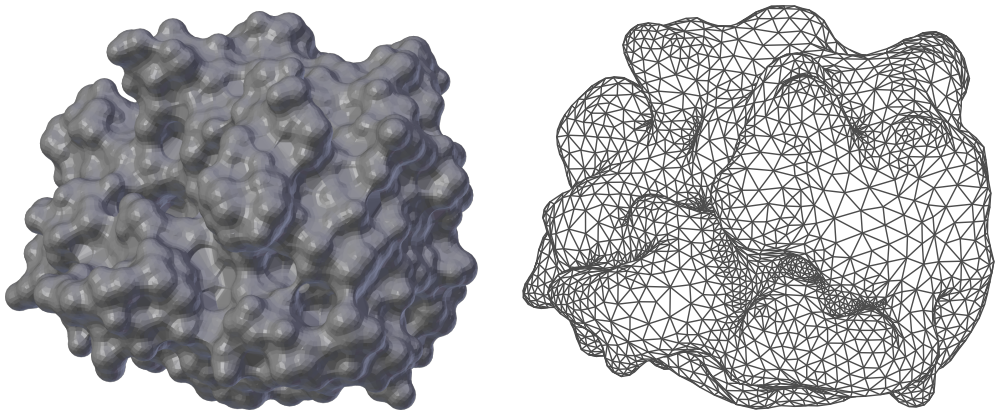
The corresponding charge models were generated from the exported PDB files using the PDB2PQR application [138] (version 1.9.0) with the AMBER [42] force field option and default parameters. For the use with AnyBEM, the final meshes and charge models were read into NESSie system models and exported as HMO+ files (cf. Sections 6.2.3 and 6.2.4). Unless noted otherwise, all datasets are used with NESSie's default system constants, that is, $\varepsilon_\Omega = 2$, $\varepsilon_\Sigma = 78$, $\varepsilon_\infty = 1.8$, and $\lambda = 20$. A selection of the resulting surface meshes can be reviewed in Figures 9.2 and 9.3.

Generating suitable surface meshes for a given protein is by no means a trivial task, even when using a meshing tool dedicated to numerical applications. In particular, we will see a serious impact of varying mesh resolutions on the solving process later in this chapter (Section 9.5.2). Generally, higher resolutions yield more and smaller triangles for the same structure than low resolutions. The small triangle counts of low-resolution meshes lead to smaller system matrices, which are potentially easier to solve, with moderately sized triangles that are usually easier to handle numerically.

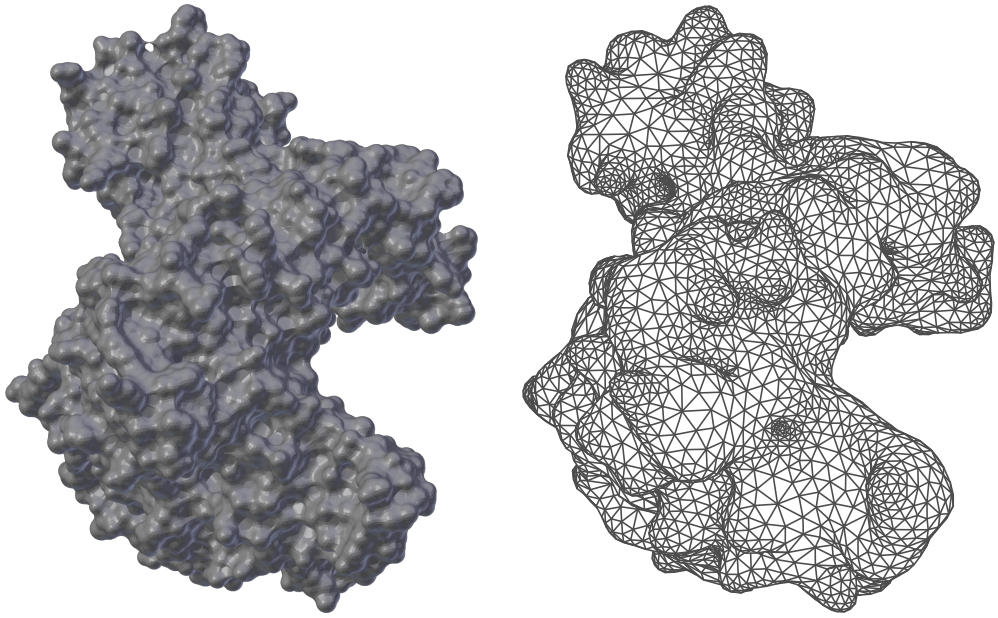
At the same time, these coarse models harbor the risk of no longer representing the protein surface accurately enough to justify the nonlocal protein electrostatics treatment in the first place. An extreme case here being the nonlocal Poisson test models from Section 6.4, where the whole protein is embedded in a spherically-symmetric domain. High-resolution meshes lead to very small triangles and large system matrices, provoking numerical instabilities in multiple steps of the computation, especially when using single-precision floating-point numbers.



(a) 2LZX / `2lzx-5k`

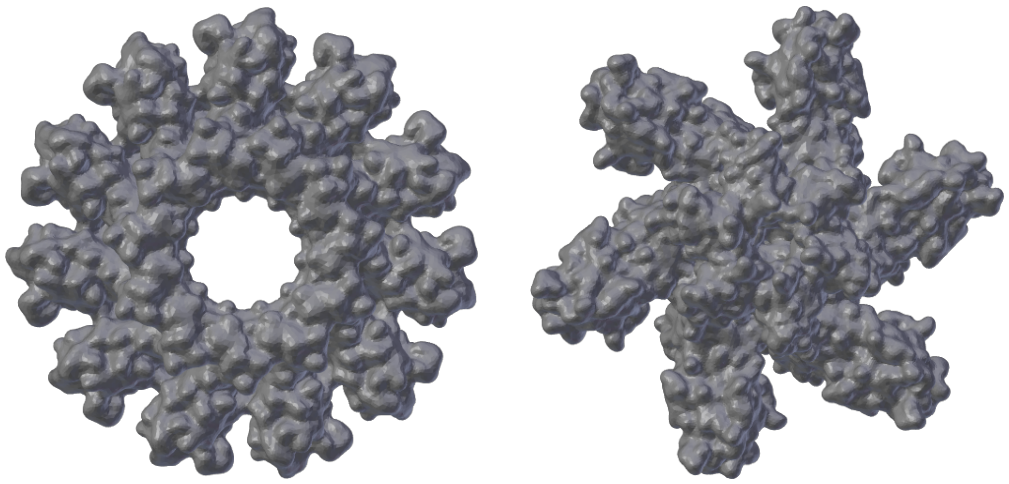


(b) 2PTC / `2ptc-9k`



(c) 1C4K / `1c4k-15k`

Figure 9.2. – Visualization of the three smallest prepared PDB structures and their corresponding surface meshes. Left: SES (solvent-excluded surface) representations of the structures generated with BALL [30]. Right: Wireframe representations of the surface meshes using the lowest resolution for each dataset.



(a) 6ds5-170k **(b)** 2h1l-200k

Figure 9.3. – Visualization of the two largest protein-based surface meshes used in the analyses. Due to the high resolutions, the meshes are shown with a uniform surface material rather than a wireframe representation.

An alternative way to provide large surface meshes while maintaining moderate triangle sizes (to mitigate the abovementioned issues) is the use of larger structures in the first place. This is why we included 6DS5 and 2H1L in the datasets. The more complex geometries of these two large structures introduced additional challenges during the surface mesh generation. Unfortunately, GAMer does not seem to cope too well with large and complex geometries, often resulting in degenerate surface triangles or premature termination of the triangulation process.

In other cases, holes in the surface topologies often lead to intersecting triangles in GAMer's surface coarsening step. While more aggressive coarsening rates intensified this problem, the coarsening process is, among other reasons, an important means to prevent point charges from crossing the molecular boundary, which would violate the base assumptions of the solvers and lead to undefined results.

In order to utilize surface meshes for 6DS5 and 2H1L, we were forced to introduce an additional preprocessing procedure to fix such mesh problems for 6DS5 and 2H1L. In particular, we used Blender's "Cleanup" functions from the "3D Printing" menu with as little modifications to the original meshes as possible. While this procedure might render the meshes less suitable for biological interpretation of the results, it still allows us to study the scalability of our implementations. Hence, we will use these large meshes primarily in the assessment of the matrix-vector product implementations. Further work is planned to investigate alternative mesh generation techniques suitable for both purposes.

9.3 Test models

In order to assure that all our solver implementations work correctly, we start with the computation of electrostatic potentials of simple spherically-symmetric models with known analytical results. The strategy used in this section is based on our previous analyses [75] for an older version of NESSie, which were exclusively performed with the explicit system representations. For this manuscript, we repeated the same analyses for the implicit system representations in both NESSie and CuNESSie.

9.3.1 Born models

In the first step, each solver had to recreate the potentials for all Born ions listed in Section 9.2.1. The analytical results were computed from the model equations implemented in the `NESSie.TestModel` module (cf. Section 6.4.1), based on the formulations summarized in Appendix A (Section A.1). The numerical results were computed by the local and nonlocal BEM solvers from the `NESSie.BEM` module (both for the explicit and implicit system representations) and the CUDA-accelerated solvers from CuNESSie. The parameters for all potentials account for a vacuum-filled solute[1] ($\varepsilon_\Omega = 1$) in water ($\varepsilon_\Sigma = 78$), with a correlation length ($\lambda = 23$) chosen according to [49, Section 3.5.3].

Figure 9.4 shows two example configurations for representative monovalent and divalent Born ions with similar radii. In the first configuration, the exterior potentials are shown for a straight line pointing away from the point charge. In the second configuration, the same potentials were computed on another straight line outside (but close to) the spherically-symmetric domain. Overall, all of our local and nonlocal solvers were able to recreate the analytical results. As expected, the nonlocal electrostatic profiles of the ions extend further into the surrounding space, implying an improved long-range visibility of the solute.

9.3.2 Poisson test models

For the second performance analysis, we utilize the Poisson test models originally published in [151], extending the previous monoatomic systems by additional point

[1] While the dielectric response of the solute is already implicitly included in the analytical formulation of the Born model, it must still be provided for the numerical BEM solvers. Choosing any different value for ε_Ω thus involuntarily leads to results that are no longer comparable to the analytical solutions.

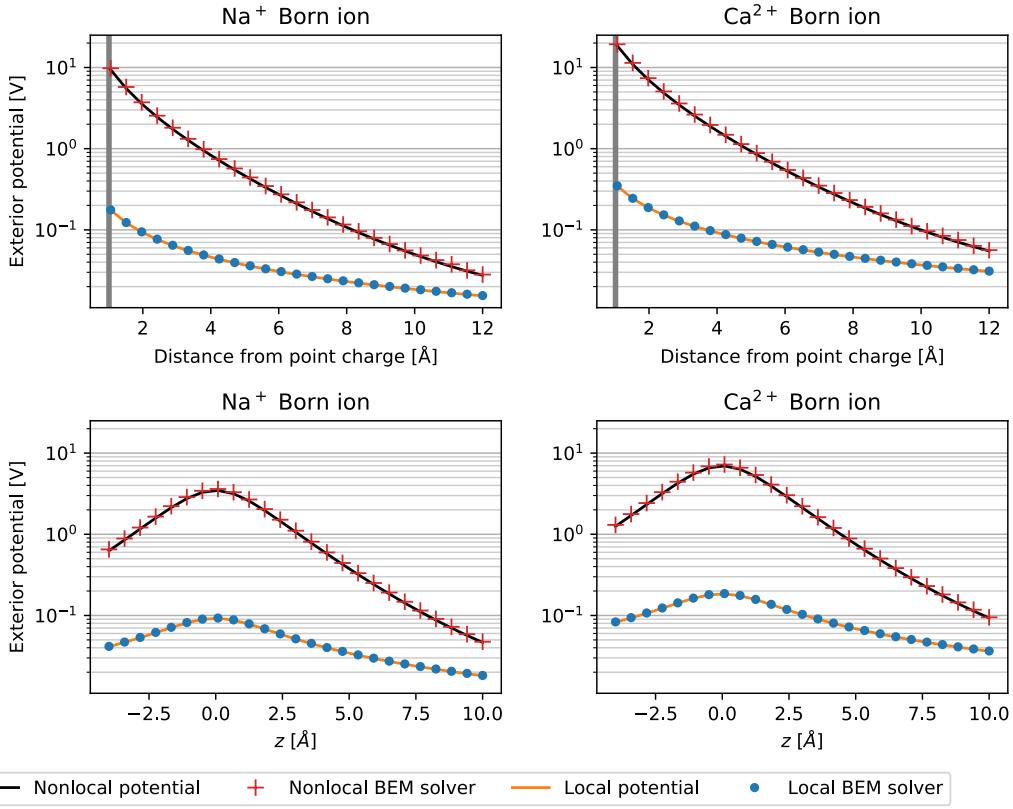


Figure 9.4. – Exterior electrostatic potentials of different monovalent and divalent Born ions in water, analytically and numerically solved for $\lambda = 23$, $\varepsilon_\Omega = 1$, and $\varepsilon_\Sigma = 78$. The point charges are located at the origin $P_0 = (0, 0, 0)$. Top: The potentials were computed along a straight line through the origin, with the solid vertical bar representing the boundary of the respective Born sphere. Bottom: The potentials were computed for observation points $P_z = (0, 2, z)$ on a straight line at 2 Å distance from the origin.

charges while retaining the spherically-symmetric cavity in the solvent space. As described in Section 6.4.2, we implemented the first variant of the Poisson test models in the `NESSie.TestModel` module. However, for an extensive evaluation of our solvers, we also compare their results to the local and the second nonlocal test models. Thus, we utilize the free Fortran package briefly introduced in the same section as well as the test model implementation in NESSie for the evaluation.

All results shown in this section are based on the 2LZX structure (cf. Section 9.2.3), translated and rescaled to fit into a sphere of 1Å radius centered at the origin. More precisely, the point charge model is translated in such a way that its centroid coincides with the origin. The new position vectors of the point charges are then uniformly scaled by a factor that positions the outermost charge at a distance of at most 0.8Å distance from the origin, leaving a margin of about 20% of the sphere radius where no charge is allowed to be located. The scaling factor is computed

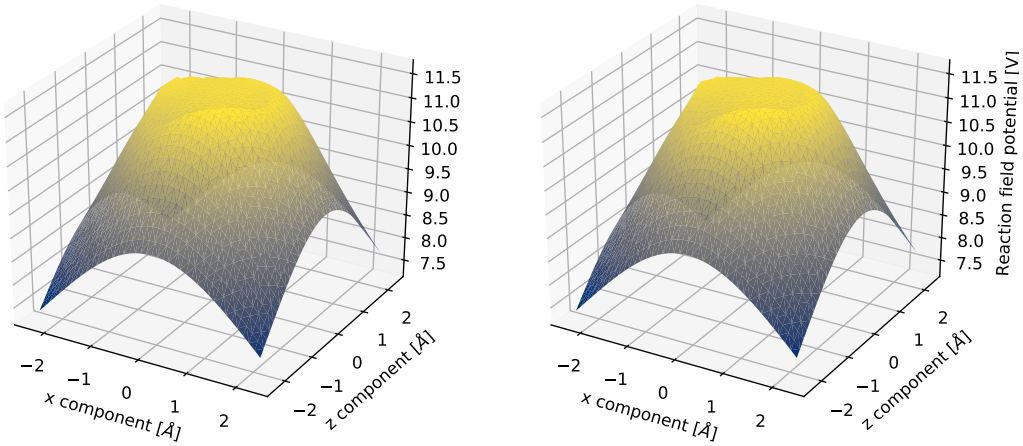
as 0.8 times the sphere radius divided by the maximum vector norm of all position vectors. In the Fortran package, the latter norm is further rounded up to the next integer. We added a `compat` option to the test model implementation in NESSie to mimic this exact same behavior.

The Fortran package creates a test model from a given PQR file, performing the described embedding process automatically after specifying a sphere radius. For all results shown here, we set the customizable parameters of the package to radius = 1, $L = 2.3$, $m = 20$, and $N = 41$, truncating the infinite series of the potential equations (cf. Appendix A, Section A.2) after m terms and computing the potentials for a uniform $N \times N \times N$ mesh of $[-L, L]^3 \subset \mathbb{R}^3$. We chose an odd value for N to make sure the xz-plane was represented in the mesh, facilitating the visualization of the potentials. The Fortran package further assumes $\varepsilon_\Omega = 2$, $\varepsilon_\Sigma = 80$, $\varepsilon_\infty = 2$, and $\lambda = 10$. Since these values cannot be modified without rebuilding the whole package, we kept them unchanged unless explicitly stated otherwise.

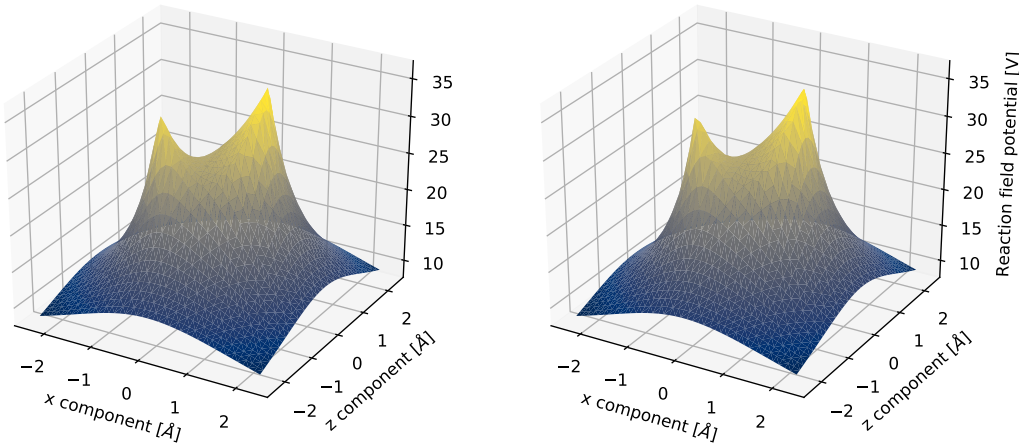
The NESSie system model was created for the same PQR file, using the `XieModel` type of the `NESSie.TestModel` module with enabled `compat` option to recreate the same representation as the Fortran package. Further, the system constants as well as the parameters for the sphere radius and the number of series terms were chosen to match the values above. The observation points for the potential computations were then created through NESSie's `obspoints_plane` function, creating a uniform 40×40 mesh of $[-2.3, 2.3]^2 \subset \mathbb{R}^2$ in the xz-plane.

All plots presented here show triangulations of the point clouds generated by computing the potentials for the respective mesh nodes. We mostly restrict the presentations to reaction field potentials rather than full electrostatic potentials. This is owing to the infinite self-energies of the point charges contributing to the molecular potentials, which make the full electrostatic potentials more challenging to inspect visually. The reaction field potentials do not suffer from this issue.

The tests performed with the Poisson test models can be roughly divided into three categories: First, we want to make sure that our implementation of the first nonlocal model variant in NESSie returns correct results. For this, we directly compare the potentials of the Fortran package to our implementation. As shown in Figure 9.5a, both tools return the same results for the given dataset. In the second test, we compare the potentials computed with our local BEM solvers (implemented in both NESSie and CuNESSie) to the local model of the Fortran package. Again, we find the same results across all implementations (Figure 9.5b).



(a) Nonlocal Poisson test model: Fortran package vs. `NESSie.TestModel`



(b) Local Poisson test model: Fortran package vs. BEM solvers

Figure 9.5. – Reaction field potentials for Poisson test models based on the `2lzx` dataset embedded into an origin-centered unit sphere, with $\varepsilon_\Omega = 2$, $\varepsilon_\Sigma = 80$, $\varepsilon_\infty = 2$, and $\lambda = 10$. The potentials were computed with the Fortran package reported in [151] (left) and NESSie/CuNESSie (right) for a section of the xz-plane.

For the third and final category, we compare the potentials computed with our nonlocal BEM solvers (in NESSie and CuNESSie) to the analytic test models. As discussed in Section 6.4.2, we cannot expect the numerical solver results to match the test model implementations – neither in NESSie nor the Fortran package – as they are built on the basis of different *nonlocal* solvent response models. So far, we have not seen any effects of these differences, since the comparisons have only been performed either between two test model implementations or for the *local* Poisson test model. What is more, the two variants of the nonlocal model themselves already show slight differences in the computed reaction field potentials. A one-on-one

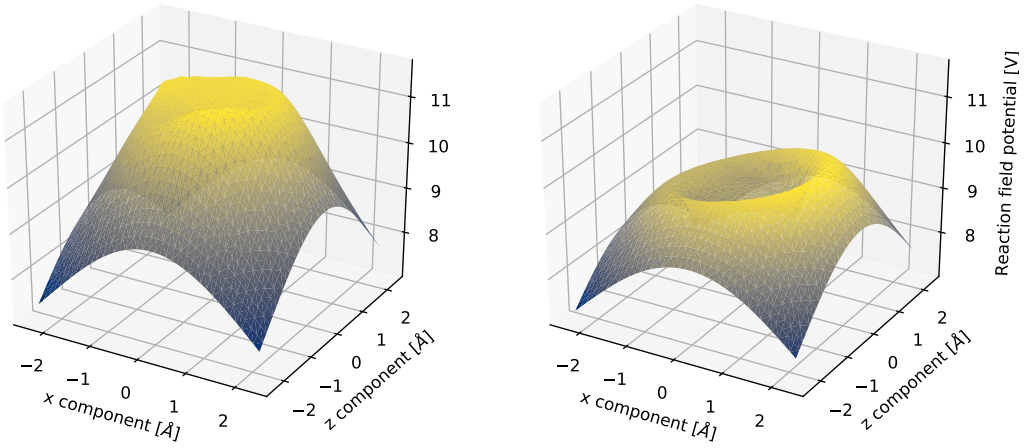
comparison of the two variants by means of the Fortran package can be reviewed in Appendix B (Figure B.1).

Figure 9.6 presents the reaction field potentials for the (first) nonlocal Poisson test model, computed through `NESSie.TestModel` and the NESSie/CuNESSie BEM solvers. As expected, the potentials for both approaches differ considerably. However, we can make a few important observations: Firstly, the transition between the interior and exterior potentials (i.e, in close proximity of the unit sphere) is completely smooth, indicating that the coupling of the potential terms in the BEM formulations works correctly. Secondly, the potentials of both approaches undergo a similar transformation when viewed as a function of the correlation length λ. The potential plots for the BEM solvers always seem to be more "dented" on the interior part than the `NESSie.TestModel` plots for the same λ. At the same time, the former plots seem to be a bit "further ahead" of their competitors when considering λ variable, with a common trend being visible in both approaches.

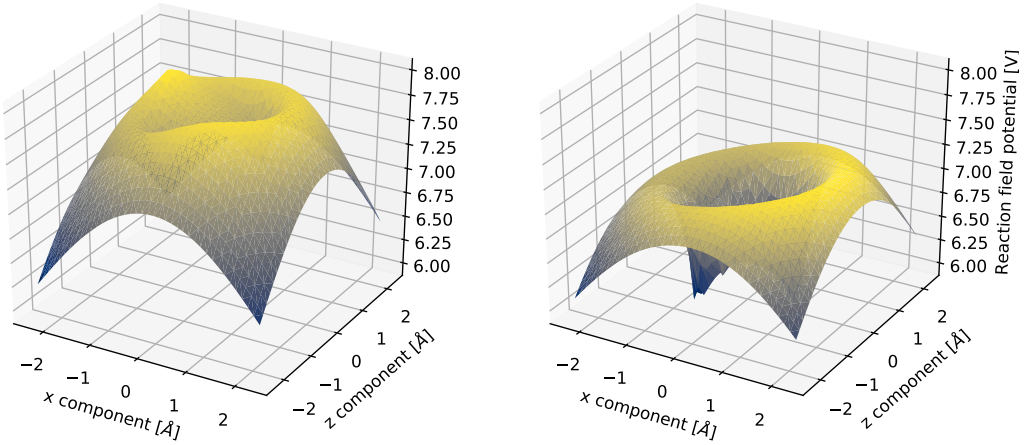
A similar effect is visible in comparison to the second nonlocal Poisson test model, albeit less pronounced. Corresponding plots can be found in Appendix B (Figure B.2). Apart from that, the differences are most prominently visible inside of or close to the sphere. In the full electrostatic potentials, they are dominated by the molecular potentials, which are unaffected by the differing solvent response models, and become negligible (Figure 9.7). Overall, we still consider the results computed with our BEM solvers to be in very good agreement with the test models but would generally rather recommend this comparison as a means to investigate different solvent response models.

9.4 Matrix-vector products

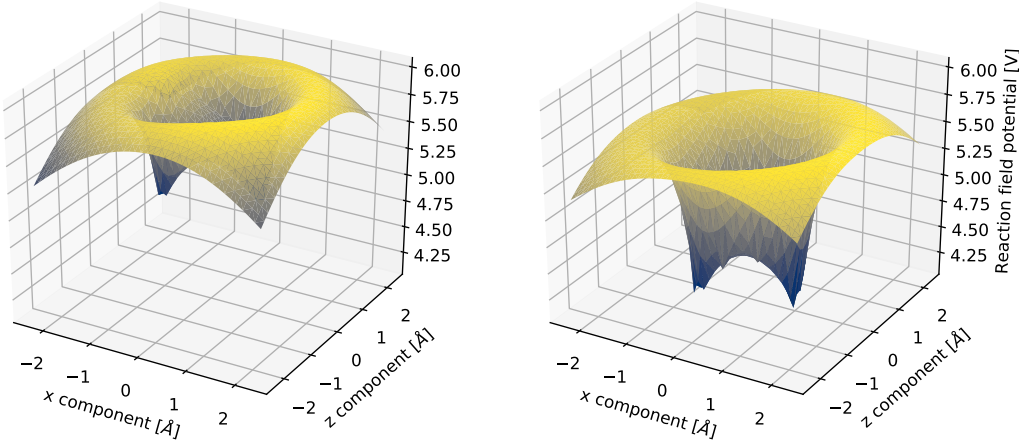
In this section, we evaluate the performance of the matrix-vector product implementations for the iterative linear system solvers. Although the individual products are not invoked separately in most use cases, they are used to assemble the local and nonlocal system matrices and thus represent the core operation of the solving process. In the context of the implicit system representations, they additionally present themselves as the dominating contributors to the overall runtime by a substantial margin. In the following, we first discuss the implications of explicit and implicit system representations and subsequently evaluate the different CPU- and GPU-based product implementations in terms of their average time consumption.



(a) $\lambda = 10$



(b) $\lambda = 15$



(c) $\lambda = 20$

Figure 9.6. – Reaction field potentials for the nonlocal Poisson test model based on the `2lzx` dataset embedded into an origin-centered unit sphere, with $\varepsilon_\Omega = 2$, $\varepsilon_\Sigma = 80$, $\varepsilon_\infty = 2$, and varying λ. The potentials were computed with `NESSie.TestModel` (left) and our BEM solvers (right) for a section of the xz-plane.

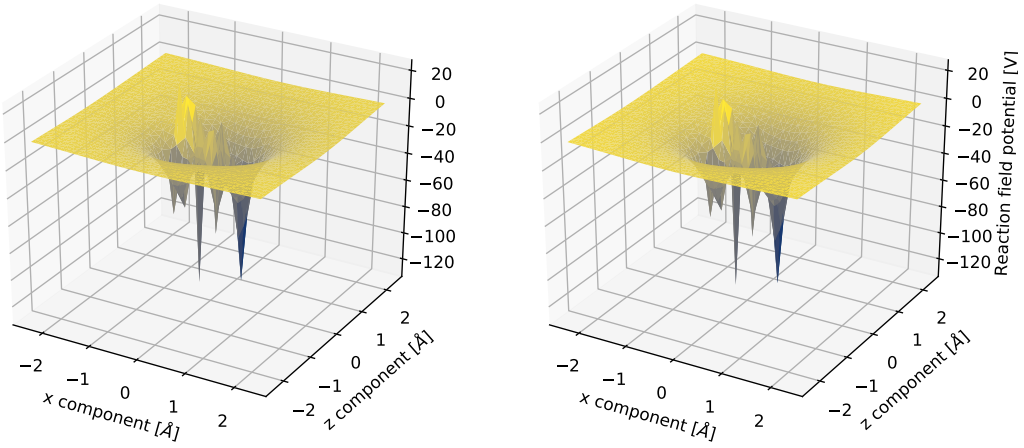


Figure 9.7. – Electrostatic potentials for the nonlocal Poisson test model based on the `2lzx` dataset embedded into an origin-centered unit sphere, with $\varepsilon_\Omega = 2$, $\varepsilon_\Sigma = 80$, $\varepsilon_\infty = 2$, and $\lambda = 10$. The potentials were computed with `NESSie.TestModel` (left) and our BEM solvers (right) for a section of the xz-plane.

9.4.1 Explicit vs. implicit representations

As discussed in Section 5.2, we chose interaction matrices as implicit representations for the potential matrices, as they elegantly capture all defining properties of these matrices at a small cost in terms of memory. Furthermore, the local and nonlocal system matrices can be assembled from the same matrices with a negligible memory overhead (i.e., for the four system constants ε_Ω, ε_Σ, ε_∞, and λ). Generally, for a single matrix-vector product, both implicit and explicit matrix representations behave very similarly, since each matrix element is only required once during the computation. In the explicit case, there is an inherent runtime overhead due to the memory allocation, which might be substantial for large matrices. On the other hand, this is a one-time cost that allows the matrix elements to be reused multiple times, e.g., for several matrix-vector products. In fact, this is a huge advantage of the explicit representation, because the actual matrix-vector product for the pre-computed potential matrices has only an insignificant impact on the overall runtime, in contrast to the computation of the matrix elements. For example, in the case of the `2lzx-5k` dataset, the matrix-vector product of the pre-computed potential matrix $\boldsymbol{V}$ accounts for roughly 0.1 percent of the runtime for computing $\boldsymbol{V}$. This imbalance intensifies with larger matrices.

Unfortunately, for a given surface approximation of n triangles, the explicit representations of the potential and system matrices require memory on the order of $\Theta(n^2)$, which might easily exceed available resources. More specifically, all potential matrices and the local system matrices contain exactly n^2 elements each, while

	2lzx-5k	2h11-200k
Potential matrices†		
○ Number of elements	24,780,484	40,232,336,400
○ Theoretical size (FP32/FP64)	94.5 / 189.1 MiB	149.9 / 299.8 GiB
○ Actual size (FP32/FP64)	291.7 / 583.4 KiB	11.5 / 23.0 MiB
Nonlocal system matrix		
○ Number of elements	223,024,356	362,091,027,600
○ Theoretical size (FP32/FP64)	0.83 / 1.7 GiB	1.3 / 2.6 TiB
○ Actual size (FP32/FP64)	291.7 / 583.4 KiB	11.5 / 23.0 MiB

Table 9.6. – Memory requirements of the interaction matrices for the smallest and largest test datasets. The theoretical sizes correspond to the explicit representations and all sizes are given for both single (FP32) and double (FP64) floating-point precision. † The values for the potential matrices also apply to the first and second local system matrices.

the nonlocal system matrices contain exactly $9n^2$ elements. When using single- or double-precision floating-point values to represent the matrices, each element consumes four or eight bytes of memory, respectively. Table 9.6 presents the resulting memory requirements of the smallest (`2lzx-5k`) and the largest (`2h11-200k`) protein-based datasets utilized in this chapter. For the implicit representations, we assume a total of 15 floating-point values per triangle, covering 3 position vectors for the triangle nodes and 2 additional position vectors for the normal vector and triangle centroid[2].

As is evident from the table, the matrices of the smallest mesh easily fit into the memory of both workstations in their explicit representation. However, for the largest mesh only a single instance of either single-precision potential matrix (or the local system matrix) fits into the memory of the more generously equipped (second) workstation. Also, it should be noted that most of the structures used as a basis for the meshes are comparatively small. At the time of writing, the Protein Data Bank contains several thousand structures with atom counts above 19,000, including large macromolecular complexes of dozens of proteins [154].

An example for such large complexes is the human 80S ribosome (PDB ID 4UG0) [155], which is listed as a composition of 76 proteins, totaling over 200,000 atoms in the first biological assembly. We expect any sufficiently detailed surface triangulation of such large structures to exceed realistic memory capacities, which is why we consider the explicit system representations an infeasible option for such use cases.

[2]This actually overestimates the memory requirements for the implementations used in AnyBEM by one position vector per triangle, since the normal vectors are computed on demand from the triangle nodes.

The implicit system representations, on the other hand, scale excellently with the number of surface triangles: Owing to the linear memory constraints of the row and column elements underlying the interaction matrices, all matrices presented in our analyses only require up to a few megabytes of memory. What is more, all matrices of the same dataset share a common memory footprint, most notably including the nonlocal system matrices, which contain nine times more elements than the other matrices. To be even more precise, all matrices of the same dataset *combined* require no more than the listed size of any individual matrix (plus system constants), as they all share the exact same row and column elements. As a result, surface approximations of as many as 100 million triangles result in implicit matrices that occupy less than 12 GiB of memory.

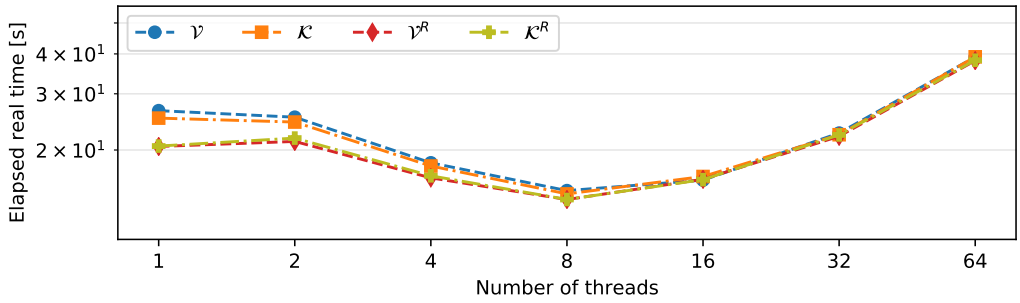
These outstanding properties render interaction matrices viable candidate representations for the nonlocal protein electrostatics problem of almost arbitrarily-sized systems. Undeniably, there is a computational overhead involved for the repeated evaluation of the expensive interaction functions. This is why we direct our focus to the actual implications of this overhead for the CPU- and GPU-based matrix-vector product implementations in the following sections.

9.4.2 CPU-based implementations

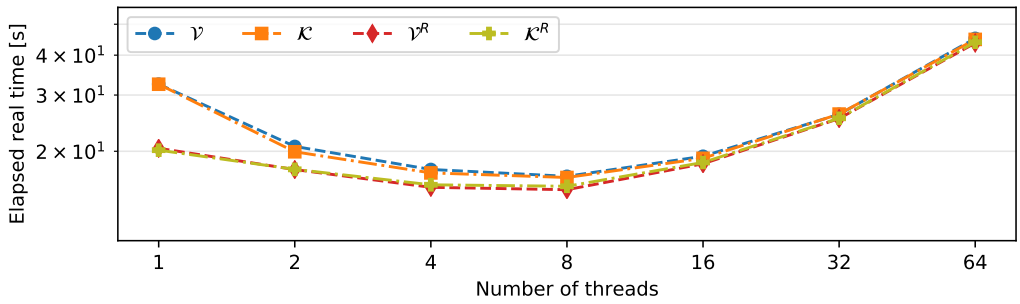
In the category of CPU-based implementations, we evaluate the runtimes of the implicit matrix representations in AnyBEM (compiled with the `cpu` back end) and NESSie. For each protein-based dataset, we compute the matrix-vector product for the interaction matrices $\mathcal{V}$, $\mathcal{K}$, $\mathcal{V}^R$, and $\mathcal{K}^R$ as well as the local and nonlocal system matrices. In the following, *local* refers to the first local system matrix, while the second local system matrix is identical to $\mathcal{V}$ and not listed separately (cf. Section 5.2.1). All time measurements comprise the creation of the matrix object (with pre-allocated row and column elements) and the product computation using a pre-allocated and accordingly-sized vector of ones as a second operand.

Each product computation is repeated ten times and timed separately[3]. The runtimes presented here refer to the average elapsed real times of the individual products on the second workstation, with more detailed result tables (including the corresponding standard deviations) being available in Appendix B (Tables B.4 and B.5). Among the interaction matrices, $\mathcal{V}$ and $\mathcal{K}$ use similar interaction functions and

[3] In Julia-based implementations, the timed computations are preceded by an additional computation to compensate for differing runtimes due to Julia's just-in-time (JIT) compilation feature during the first computation.



(a) Single precision



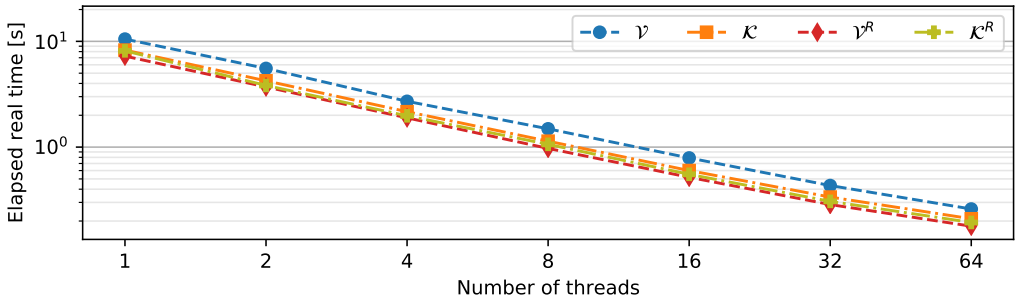
(b) Double precision

Figure 9.8. – Average runtimes for a single matrix-vector product of the implicit potential matrices associated with the `2lzx-5k` dataset in NESSie.

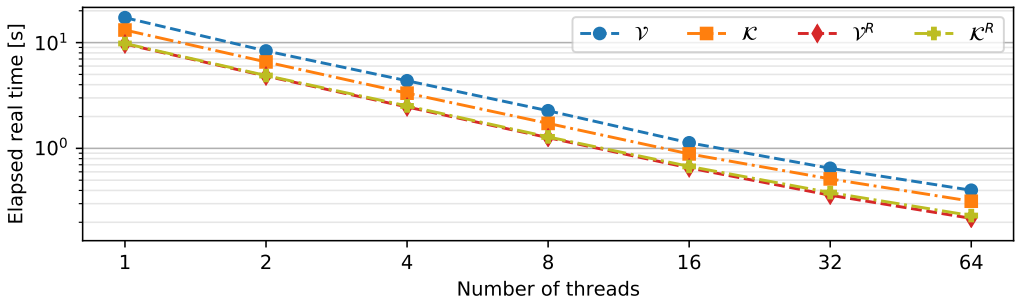
solving techniques. The same it true for $\mathcal{V}^R$ and $\mathcal{K}^R$. As as result, we will usually see similar performance results for these matrices. Furthermore, the quadrature-based interaction functions of $\mathcal{V}^R$ and $\mathcal{K}^R$ can often be computed more efficiently than the ones of $\mathcal{V}$ and $\mathcal{K}$, which heavily rely on trigonometric and logarithmic functions.

In both NESSie and AnyBEM, the matrix-vector products are computed row-wise, where the rows are equally distributed over all available threads of execution (cf. Sections 6.3.2 and 8.4). Owing to the large number of rows (usually exceeding the number of CPU threads by far) and the embarrassingly parallel nature of the matrix element computations, we expect the parallelized product implementations to scale (roughly) linearly with the number of threads. However, the Julia-based implementation does not even remotely meet this expectation, as can be reviewed in Figure 9.8. Multithreading support in NESSie is implemented through Julia's `@threads` macro, which is the only difference to the sequential implementation.

Interestingly, the simple multithreading approach causes a substantial computational overhead that only yields slight performance improvements for a small number of threads and results in a severely degraded performance for large numbers. Even in the best case (8 threads), NESSie only achieves an effective speedup of 1.5 to 1.8 for the single-precision matrices and a speedup of about 1.3 to 2.0 for the



(a) Single precision

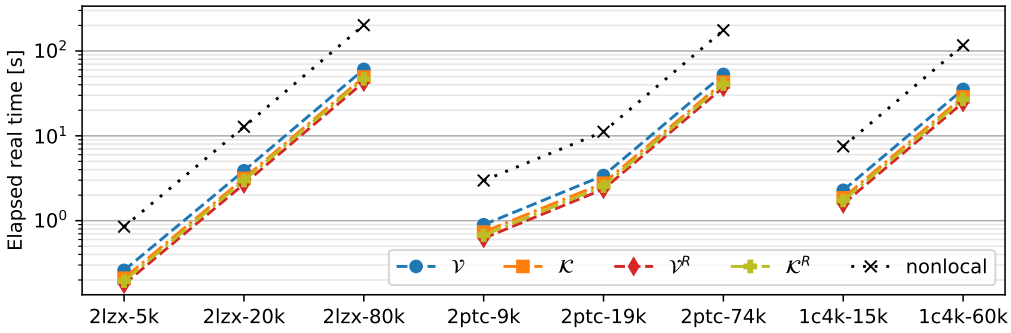


(b) Double precision

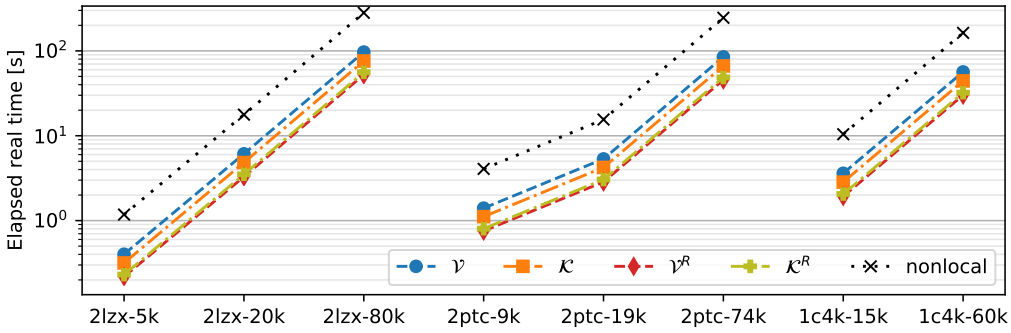
Figure 9.9. – Average runtimes for a single matrix-vector product of the implicit potential matrices associated with the `2lzx-5k` dataset in AnyBEM (CPU back end).

double-precision matrices. Apart from this, the performance results are generally not very promising. Albeit being on par with the explicit system representations in the sequential case, all matrix-vector products for the smallest dataset take more than ten seconds. For the iterative solving process, we can expect several dozens or hundreds of iterations to be required before convergence, rendering the whole process highly time-consuming. After all, the multithreading facilities in Julia are marked an experimental feature at the time of writing and we expect an improved scaling behavior in the future. Hence, we cannot recommend NESSie for the protein electrostatics problem with implicit representations for the time being.

Our second competitor, on the other hand, yields more promising results, as presented in Figure 9.9. While utilizing the same row-based parallelization scheme as NESSie, AnyBEM provides matrix-vector products that scale roughly linearly with the number of threads. With 64 threads of execution, the AnyBEM matrices reach a maximum speedup of 38.9 to 42.0. On top of that, any choice of threads result in an overall runtime below any of the results generated with NESSie. This is particularly true for the sequential implementations in AnyBEM, which show only a slight advantage over NESSie in the double-precision setting and a substantial lead for the single-precision matrices. For the given dataset, AnyBEM provides matrix-vector



(a) Single precision



(b) Double precision

Figure 9.10. – Average runtimes for a single matrix-vector product of the implicit potential matrices in AnyBEM (CPU back end with 64 threads of execution).

products that appear highly feasible for the iterative solving process. However, said dataset still is the smallest representative so far. Also, there are multiple potential matrices involved in the solving process. More precisely, the two local systems each require the comparatively expensive matrices $\mathcal{V}$ and $\mathcal{K}$, while the nonlocal systems are composed of all four interaction matrices at once.

Figure 9.10 shows the average runtimes for a single matrix-vector product using 64 threads of execution and additional protein-based datasets. Generally, the single-precision matrices perform slightly better than their double-precision counterparts. Also, the quadrature-based matrices $\mathcal{V}^R$ and $\mathcal{K}^R$ can be computed fastest among the interaction matrices. The performance of the first local system matrix (Appendix B, Table B.5) is nearly the same as for potential matrix $\mathcal{K}$, which is not surprising as its matrix-vector product represents the operation $\alpha\boldsymbol{x} + \beta(\mathcal{K}\boldsymbol{x})$ for some $\alpha, \beta \in \mathbb{R}$ and a vector $\boldsymbol{x} \in \mathbb{R}^n$: As expected, the uniform scaling and summation of vectors do not contribute significantly to the runtime of the operation. In the nonlocal case, each potential matrix is computed once. Thus, the average elapsed real time for a single matrix-vector product of the nonlocal system matrix corresponds to the sum of the

runtimes of the individual interaction matrices. Again, the vector operations further involved in the assembly do not contribute significantly to the overall runtimes.

In total, the performance of the CPU-based AnyBEM matrix operations can be considered viable for the iterative solving process of the low-resolution protein meshes. The largest meshes, on the other hand, still show a problematic tendency towards infeasible runtimes. Especially among the nonlocal system matrices of the largest meshes `6ds5-170k` and `2h1l-200k`, only the single-precision version of the former managed to finish in under a thousand seconds (for a single product). Assuming several hundreds of solver iterations, we do not currently consider the CPU-based AnyBEM matrix approach a viable option for larger protein meshes.

9.4.3 GPU-based implementations

For the GPU-based matrix-vector products, we assess the performance of AnyBEM once again, this time compiled with the `cuda` back end. AnyBEM is here accompanied by CuNESSie instead of NESSie. The basic setup remains the same: We measure the elapsed real time of each individual product computation for all protein-based datasets, implicit potential matrices, and system matrices. In contrast to the last section, we utilize both workstations (i.e., two different hardware accelerators) for the evaluations. As before, each computation is repeated ten times and the results are summarized in this section, with standard deviations and more detailed timings being available in Appendix B (Tables B.6, B.7, B.8, and B.9).

While the core of the parallelization scheme is similar to the CPU-based implementations, that is, the row-wise computation of the products, there is an additional layer of complexity regarding the distribution of the workload. More specifically, in a CUDA context, the individual tasks (i.e., matrix rows) need to be grouped into equally-sized blocks. Several device limits need to be taken into account for maximum efficiency, such as the maximum number of warps, blocks, and threads per streaming multiprocessor (SM).

One common way to express this efficiency is by means of the so-called *occupancy*, namely, the rate of the concurrently active warps versus the theoretical limit of the same. The RTX 2070 SUPER has a limit of 1,024 active threads per SM, resulting in a theoretical limit of 32 warps[4]. Furthermore, the number of active blocks per SM is limited to 16. The TITAN V doubles these limitations and thus allows up to 2,048 threads, 64 warps, and 32 blocks per SM. The theoretical occupancy is usually

[4]Each warp comprises a fixed number of 32 threads.

	Single precision		Double precision	
Matrix	CuNESSie	AnyBEM	CuNESSie	AnyBEM
$\mathcal{V}$	70	58	96	109
$\mathcal{K}$	67	58	94	99
$\mathcal{V}^R$	69	41	98	69
$\mathcal{K}^R$	72	47	103	75

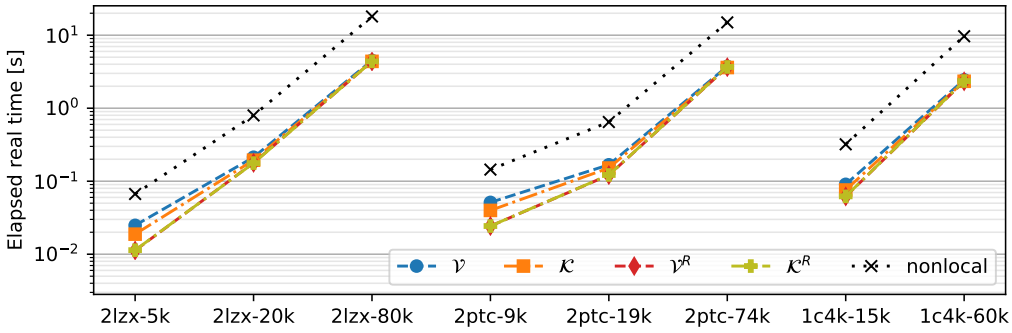
Table 9.7. – Number of per-thread registers required for the CUDA-accelerated matrix-vector products, depending on the tool and floating-point precision used

further restricted by the shared memory used per block. Since our implementations do not utilize shared memory at all, this factor does not apply here.

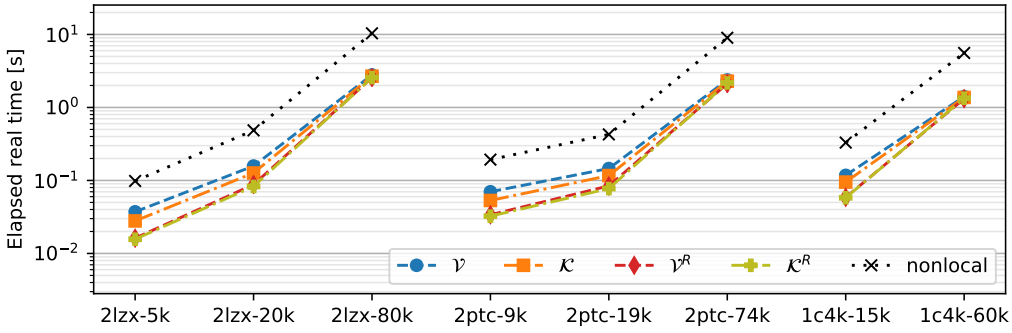
In contrast, another limiting factor that does apply is the number of available 32-bit registers per block and SM. When dividing the maximum number of registers per SM (65,536) by the maximum number of active warps and the warp size, we obtain the register-imposed limits of active warps. Thus, to allow for the maximum occupancy on both machines, the number of registers is limited to 64 per thread on the RTX 2070 SUPER and to 32 per thread on the TITAN V. However, the interaction functions involved in the matrix-vector products are comparatively complex and demand a fair number of registers. Table 9.7 shows the actual numbers of registers required for each interaction matrix and thread, depending on the implementation and floating-point precision used. Only the single-precision AnyBEM option satisfies these limits with the RTX 2070 SUPER. More specifically, the cheapest option in terms of registers (41 per thread) allows up to 49 active warps. At the other end of the spectrum, we find a requirement of 109 registers per thread, resulting in a maximum of 18 active warps for this option.

For our performance assessments, we tested different partitioning schemes with regard to different numbers of warps per block. We found a block size of 128 threads, i.e., 4 warps per block, to be the best-performing option across all potential matrices, devices, and implementations (although only by a small margin for some combinations). This partitioning scheme limits the number of active blocks to 4 in the worst case (i.e., 16 active warps) and to 12 (i.e., 48 active warps) in the best case, resulting in a theoretical occupancy of 50% to 100% for the RTX 2070 SUPER and 25% to 76.6% for the TITAN V. While we strictly refrain from claiming to have exhausted all possible optimization options here, we consider 128 threads per block a suitable default partitioning scheme for the feasibility assessment of the implicit system representations.

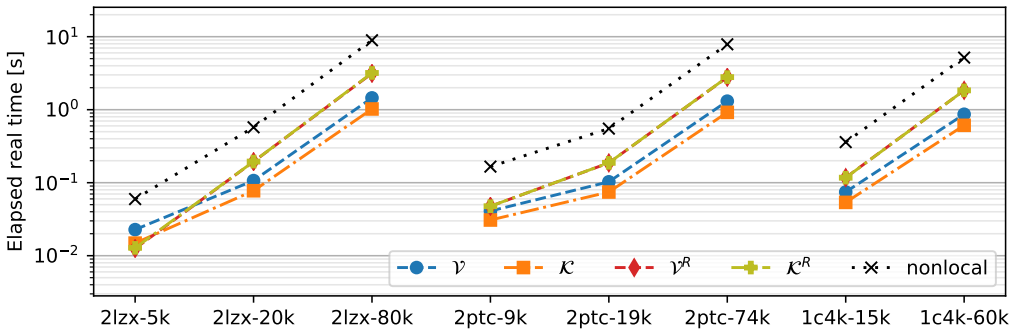
Figure 9.11 summarizes the average runtimes for the single-precision matrix-vector products. As a first observation, the CUDA-enabled implementations outperform



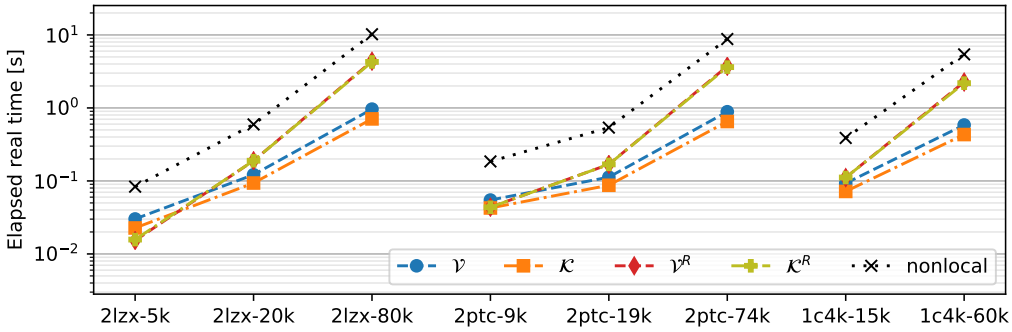
(a) CuNESSie: RTX 2070 SUPER



(b) CuNESSie: TITAN V



(c) AnyBEM: RTX 2070 SUPER



(d) AnyBEM: TITAN V

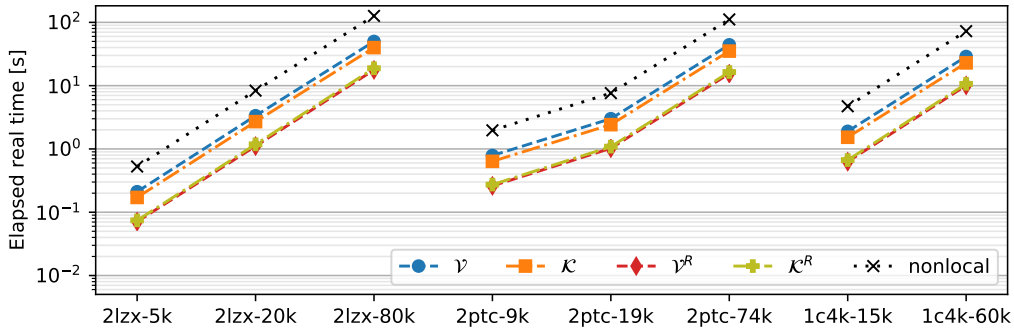
Figure 9.11. – Average runtimes for a single CUDA-accelerated matrix-vector product of the implicit potential matrices (single-precision matrices).

all CPU-based implementations by at least one order of magnitude. Interestingly, AnyBEM copes very well with the potential matrices $\mathcal{V}$ and $\mathcal{K}$, which are usually less efficient to compute than their quadrature-based counterparts $\mathcal{V}^R$ and $\mathcal{K}^R$ (cf. Section 9.4.2). In direct comparison to the sequential CPU-based AnyBEM implementation using the `2lzx-5k` dataset, the GPU-based AnyBEM version reaches a speedup of 270 to 353 on the first workstation (RTX 2070 SUPER), depending on the matrix, and a speedup of 348 to 517 on the second workstation (TITAN V). Similarly, CuNESSie reaches a speedup of 247 to 359 on the first and 282 to 519 on the second workstation (with respect to the sequential AnyBEM version).

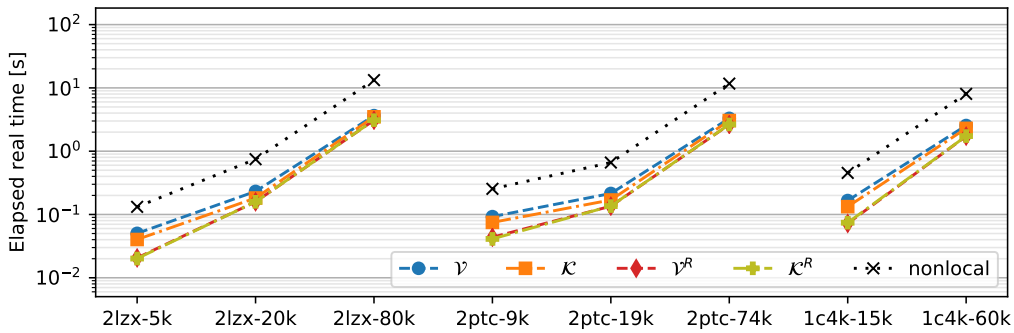
Overall, AnyBEM shows similar runtimes on both devices. While both are on par for the small protein meshes, the RTX 2070 SUPER is only about 20% faster on the largest meshes than its competitor. CuNESSie, on the other hand, proves more susceptible to the choice of device. Here, the RTX 2070 SUPER is slightly faster on the small meshes but substantially slower on larger meshes (almost by a factor of two in the worst case). When comparing the different implementations on the same device, AnyBEM is consistently faster than CuNESSie with the RTX 2070 SUPER and scales better with the mesh sizes. On the TITAN V, both implementations are roughly on par across all mesh sizes. With the exception of CuNESSie on the RTX 2070 SUPER, all options allow to compute a single matrix-vector product of the largest dataset's nonlocal system matrix in roughly one minute. While this still renders the iterative solving process of such single-precision systems highly time-consuming, we finally reach runtimes that are manageable in real application scenarios.

The results corresponding to the double-precision matrices are shown in Figure 9.12. Here, we do not expect the RTX 2070 SUPER to perform particularly well due to the comparatively small number of double-precision hardware units (cf. Section 9.1). Indeed, the figure clearly shows the device's impaired performance across all matrices and implementations, with maximum speedups of 45 to 66 as compared to the sequential AnyBEM implementation for the `2lzx-5k` dataset.

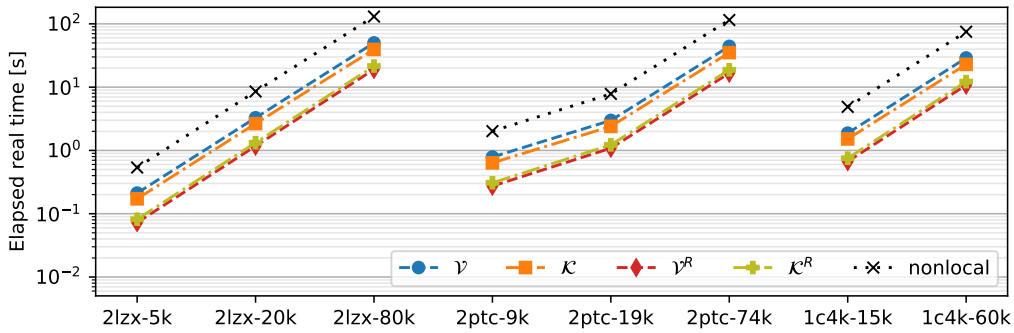
That being said, the RTX 2070 SUPER continues to yield faster results for all matrices than the CPU-based approach and is able to compute the matrix-vector products for the largest datasets within a few hundred seconds. On the other hand, the TITAN V copes well with the double-precision matrices, with maximum speedups of 326 to 487 for `2lzx-5k`. Generally, the individual products are more time-consuming for double-precision values than for single precision. However, aside from the point that the resulting average runtimes remain manageable, they also do not directly translate into the overall runtime of the iterative solving process. In fact, the different



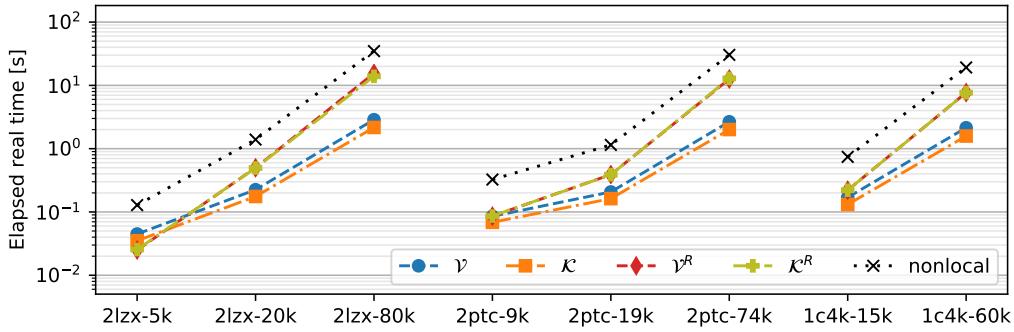
(a) CuNESSie: RTX 2070 SUPER



(b) CuNESSie: TITAN V



(c) AnyBEM: RTX 2070 SUPER



(d) AnyBEM: TITAN V

Figure 9.12. – Average runtimes for a single CUDA-accelerated matrix-vector product of the implicit potential matrices (double-precision matrices).

precision levels show a significant influence on the convergence rate for the systems, resulting in vastly diverging solving times, as we will see in the next section.

In conclusion, CuNESSie presents itself as a reliable choice for the matrix-vector products computed on the TITAN V, so that we can finally try to solve the local and nonlocal BEM systems using their implicit representations and iterative solvers in the next step. On the RTX 2070 SUPER, the situation is less clear. While CuNESSie still yields good runtimes for the single-precision matrices, the device is less suited for double precision, hampering a direct comparison of the BEM solvers on the first workstation. The same argument applies to a possible solver implementation in AnyBEM. Nevertheless, the package's performance in terms of matrix-vector products on the TITAN V justifies further investigation in this regard.

9.5 BEM solvers

In the last section of this chapter, we evaluate the runtime performance of our BEM solver implementations. In the first step, we compare the nonlocal BEM solvers for the explicit system representations in NESSie to the prototypical reference program published as [61] (cf. Chapter 6). Although we primarily focus on implicit system representations in this manuscript, this comparison serves as a general justification of the Julia-based approach and establishes a performance baseline. In the second and final step, we put the CUDA-accelerated matrix-vector product implementations to the test and evaluate the performance of our BEM solvers for different iterative linear system solvers.

9.5.1 Explicit system representations

The performance evaluation for the explicit system representations is performed for 60 datasets in total, that is, the 57 meshes presented in Section 9.2.2 and the three meshes `2lzx-5k`, `2ptc-9k`, and `1c4k-15k` from Section 9.2.3. Further, the time measurements cover the assembly of the linear systems in double precision – including the computation of the potential matrices $\boldsymbol{V}$, $\boldsymbol{K}$, $\boldsymbol{V}^R$, and $\boldsymbol{K}^R$ – as well as the solving process. In the reference program, the latter step is delegated to the ATLAS library [136] (here used in version 3.10.2), while Julia uses LAPACK [131] directly for the same purposes. Since the reference program does not support any kind of multithreading, we prohibit the same in NESSie accordingly by

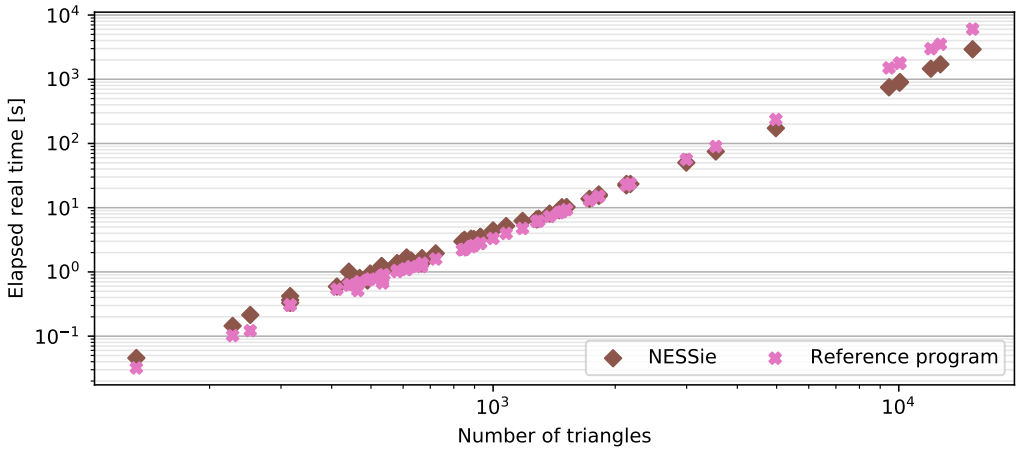


Figure 9.13. – Single-thread performance of the nonlocal BEM solvers provided by NESSie and the reference program presented in [61] for the explicit system representations of differently sized surface meshes.

setting the `JULIA_NUM_THREADS` environment variable to one and explicitly call `LinearAlgebra.BLAS.set_num_threads(1)`.

Figure 9.13 shows the elapsed runtimes for the linear system assembly and solving process of all 60 datasets on the second workstation. Generally, both implementations yield comparable runtimes for the same meshes. While the reference program consistently finishes execution faster than NESSie for meshes up to a few thousand triangles, the Julia-based solver performs better with larger meshes. In particular, NESSie requires 173 seconds for `2lzx-5k`, 748 seconds for `2ptc-9k`, and 2,929 seconds for `1c4k-15k`, while the reference program requires 236, 1,501, and 6,041 seconds for the same datasets.

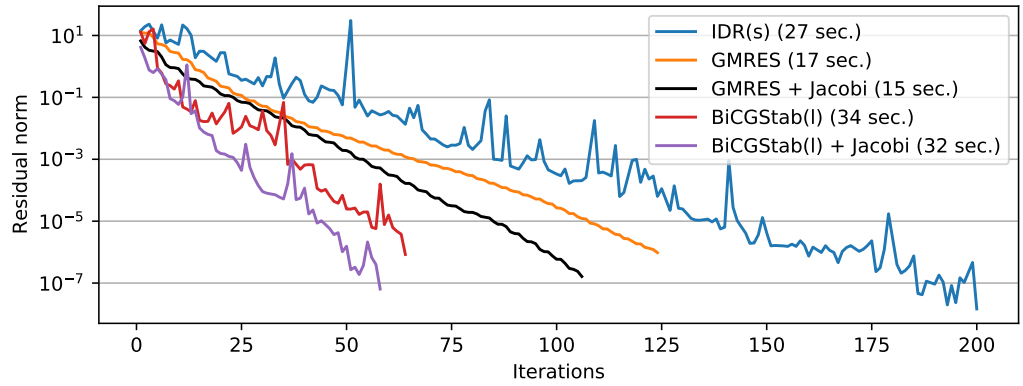
It should be noted here that ATLAS was installed as a pre-built library through the operating system's official package repository. As a result, the library was built without performance optimizations specific to the workstation. Enabling the performance optimizations is a disruptive process and requires elevated user privileges on most machines with active CPU throttling, which is why we chose a less disruptive alternative for the second workstation. ATLAS is known to be particularly susceptible to these build-time optimizations [156], such that the performance results shown here do not reflect the best results possible. However, we have shown previously [75] for the same 57 meshes from Section 9.2.2 and a different machine that the reference program is able to reduce NESSie's lead with the largest meshes to just a few seconds when performing the build-time optimizations as suggested.

9.5.2 Implicit system representations

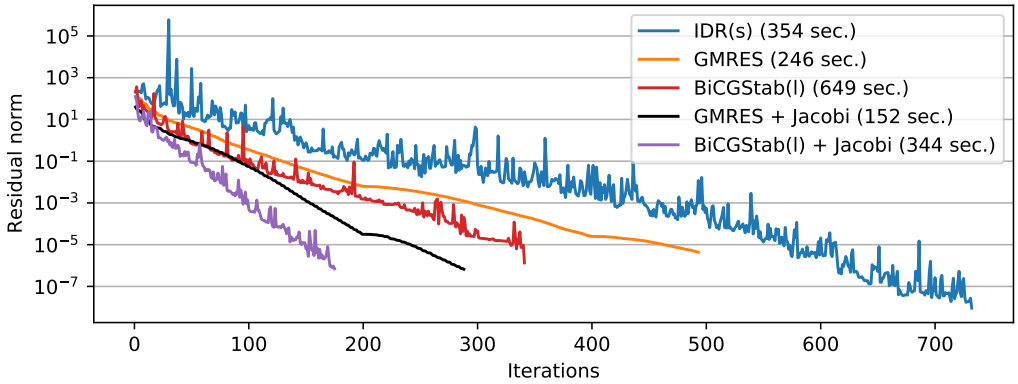
The performance evaluation of the sequential BEM solvers for the explicit system representations with double-precision values resulted in solving times of about 3 minutes for `2lzx-5k`, 13 minutes for `2ptc-9k`, and 49 minutes for `1c4k-15k`. These values include the one-time computation of all potential matrices $\boldsymbol{V}$, $\boldsymbol{K}$, $\boldsymbol{V}^{\boldsymbol{R}}$, and $\boldsymbol{K}^{\boldsymbol{R}}$ (each of size $n \times n$ for n surface triangles) as well as the actual solving of the nonlocal system by means of LU factorization (with a system matrix size of $3n \times 3n$). Owing to the different dimensions of the involved matrices, the latter step quickly becomes the dominant contributor to the overall runtime. For instance, the LU factorization alone accounts for roughly half of the total time spent on `2lzx-5k`, while this contribution increases to almost 80% for `1c4k-15k`. In the following, we finally evaluate whether an iterative solving approach with accelerated matrix-vector products, which does not require a LU factorization, can balance out the costs of repeatedly recomputing all potential matrices and serve as a viable alternative to the classic approach.

As a consequence of the performance results presented in Section 9.4, we do not consider CPU-based implementations of the matrix-vector products for the iterative solving process. Among the GPU-based implementations, both CuNESSie and AnyBEM showed reasonable runtimes for an attempt at solving the systems on both workstations (with the exception of double-precision matrices on the RTX 2070 SUPER). However, AnyBEM does not currently provide a linear system solver suitable for this purpose, which is why we henceforward exclusively utilize CuNESSie for the performance evaluation.

Since CuNESSie is Julia-based software, it benefits from a range of options regarding compatible iterative solvers. Here, we consider the implementations provided by the IterativeSolvers.jl package (cf. Section 5.4.1), namely, restarted GMRES, BiCGStab(l) and IDR(s). Where possible, we further compare the implementations with and without a simple Jacobi preconditioner as left preconditioner (cf. Section 5.4.2). Figure 9.14 shows the convergence behavior of the aforementioned solver options for the nonlocal BEM systems in double precision. The stated runtimes correspond to the second workstation, while the remaining values are independent of the device. The stopping criterion for each solver is specific to the individual implementation and floating-point precision. For double precision, IDR(s) assumes convergence if the residual norm for the solution vector falls below $1.49 \cdot 10^{-8}$. GMRES and BiCGStab(l) additionally scale this value by the residual norm of the initial guess for the solution vector (here, a vector of zeros) and assume convergence once the current residual norm falls below this scaled value.



(a) `2lzx-5k`



(b) `1c4k-15k`

Figure 9.14. – Convergence behavior of different iterative solvers for the nonlocal BEM systems using CuNESSie and double floating-point precision. All runtimes correspond to the computation on the second workstation (TITAN V). For better readability, the legend entries are sorted by the total number of iterations (in descending order).

Notwithstanding these different criteria, the figure shows clear distinctions between the solvers across most residual norms. As a first observation, the systems of both datasets can be solved significantly faster than before, no matter the solver used. Secondly, the preconditioned solver variants converge faster than their counterparts. Apart from these general observations, the IDR(s) solver requires a substantial amount of iterations more than any other solver, even when considering a less strict stopping criterion. Also, the solver experiences large fluctuations in terms of the residual norm – sometimes spanning multiple orders of magnitude between iterations. At the other end of the spectrum, the BiCGStab(l) solvers are among the fastest ones in terms of the number of iterations. At the same time, they are also among the slowest solvers in terms of runtime. This is due to each BiCGStab(l) iteration requiring $2l$ matrix-vector products (with l being the number of internal GMRES steps per iteration), while IDR(s) and GMRES only compute one product

	Single precision		Double precision	
Dataset	Iterations	Residual norm	Iterations	Residual norm
`2lzx-5k`	54	$3.64 \cdot 10^{-3}$	106 (47)	$1.63 \cdot 10^{-7}$
`2lzx-20k`	89	$7.88 \cdot 10^{-3}$	131 (58)	$2.78 \cdot 10^{-7}$
`2lzx-80k`	833	$1.64 \cdot 10^{-2}$	171 (76)	$6.64 \cdot 10^{-7}$
`2ptc-9k`	122	$6.63 \cdot 10^{-3}$	193 (100)	$2.68 \cdot 10^{-7}$
`2ptc-19k`	165	$1.05 \cdot 10^{-2}$	289 (122)	$4.50 \cdot 10^{-7}$
`2ptc-74k`	–	(DNF)	554	$6.87 \cdot 10^{-7}$
`1c4k-15k`	156	$1.60 \cdot 10^{-2}$	288 (117)	$6.66 \cdot 10^{-7}$
`1c4k-60k`	1397	$3.27 \cdot 10^{-2}$	380 (147)	$1.36 \cdot 10^{-6}$

Table 9.8. – Nonlocal BEM solver performance using CuNESSie with restarted GMRES, Jacobi preconditioning, and default convergence criteria. Iteration numbers in parentheses correspond to the single-precision convergence criterion. Solving attempts exceeding 3,000 iterations are marked as *did-not-finish* (DNF).

per iteration. By default, BiCGStab(l) uses $l = 2$, rendering each iteration about four times as expensive as compared to the other solvers.

The clear winner in terms of runtime is the preconditioned GMRES solver. Here, the residual norm is minimized monotonously, with possible plateau phases due to the GMRES process being restarted after a fixed number m of iterations. A small value for m, such as the default setting $m = 20$, results in a severely impeded convergence behavior with our systems, as convergence tends to start slowly at the beginning of each GMRES process. This effect can be inspected in Figure 9.14b (with $m = 200$). Conversely, a high value for m increases the memory footprint of the solver up to a point equivalent to the explicit system representations[5]. We chose $m = 200$ as a compromise to maintain a small total memory footprint and as a reasonable default for all datasets. In a real application scenario, the parameter should then be adapted to the particular systems at hand.

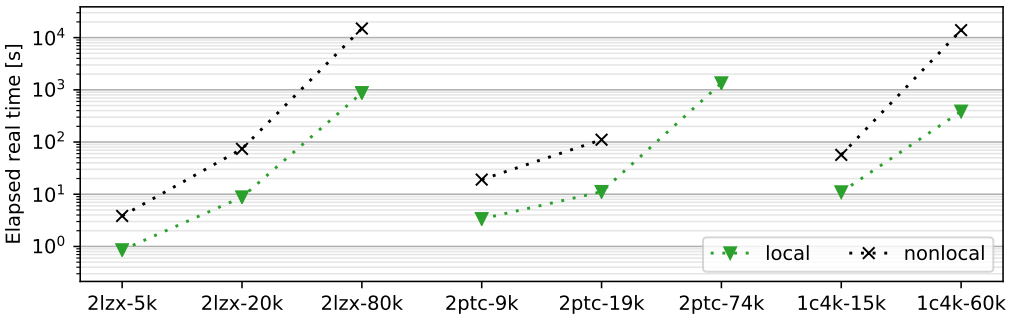
Table 9.8 lists the number of iterations required by the restarted and preconditioned GMRES solving attempts as well as the resulting residual norms for the nonlocal BEM systems of the protein-based datasets. As is evident from the table, all systems of low-resolution meshes (`2lzx-5k`, `2ptc-9k`, and `1c4k-15k`) can be solved within a few dozens or a few hundreds of iterations, both with respect to single and double precision. Furthermore, the single-precision variants usually require fewer iterations to reach convergence. However, this is mainly owing to the convergence criteria for different precision levels and that the double-precision solvers meet the single-precision convergence criteria consistently in fewer iterations.

[5]The solver internally allocates a matrix of size $3n \times m$ for the nonlocal systems corresponding to n surface triangles.

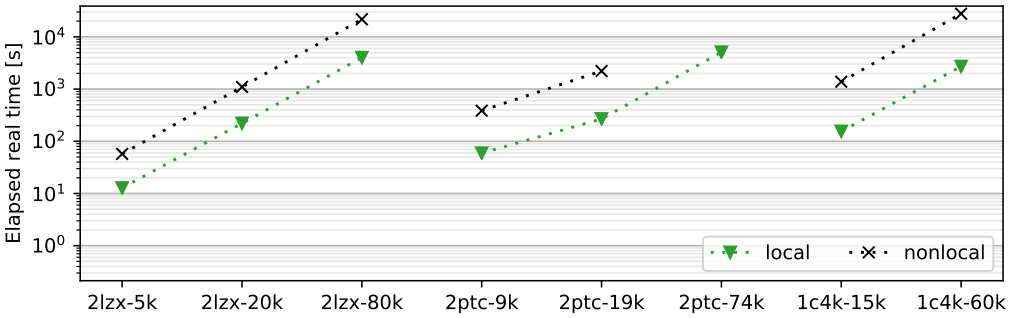
Before we inspect how these observations translate into runtimes, we first need to address the proverbial elephant in the room: Obviously, the combination of high-resolution meshes and single-precision values severely impacts the solving process. We have briefly discussed this issue in Section 9.2.3. To further reconstruct the source of this issue, we need to remember that the system matrix originates from a point-collocation method (cf. Sections 3.2.2 and 3.3.2), such that each element effectively represents a pair of surface triangles. In a sense, the information represented by the matrix is distributed over all triangles, where the individual matrix elements are weighted by the area of the corresponding triangles. Thus, increasing the mesh resolution, that is, the number of triangles, has two direct implications on the matrix-vector products: Firstly, the per-row dot products consist of more components. Secondly, the matrix elements become smaller. In other words, with increasing mesh resolution, the dot products involved in the matrix-vector products need to be computed for more and smaller operands, increasing the chance of floating-point cancellations and other numerical instabilities. Such effects are visible for all high-resolution meshes `2lzx-80k`, `2ptc-74k`, and `1c4k-60k`. To make matters worse, the `2ptc-74k` dataset even contains three triangles with an area below $3.45 \cdot 10^{-4}$, which are treated as being zero in the single-precision solvers[6]. While these issues primarily occur in high-resolution meshes, low-resolution meshes should also be inspected carefully before usage. In particular, large variations in the triangle sizes might similarly result in cancellations. Ideally, meshes should contain roughly equally-sized triangles to avoid such problems. Overall, a suitable mesh resolution needs to be determined for each biomolecular system to find a compromise between the structure and triangle sizes.

Figure 9.15 and Table 9.9 finally present the runtimes corresponding to the local and nonlocal BEM solvers on both workstations. The results shown for the local case represent the combined solving times for both local systems (cf. Section 5.2.1). Nonetheless, the local solvers usually finish execution much faster than their nonlocal variants. For low- and mid-resolution meshes, the single-precision systems can be solved significantly faster than their double-precision counterparts. The latter observation was to be expected for two reasons: Firstly, both devices are equipped with an unproportionally large number of single-precision hardware units (cf. Section 9.1). This is a particular challenge for the RTX 2070 SUPER, which is no match for the TITAN V in the double-precision category, while dominating in the single-precision category for low-resolution meshes. Secondly, as discussed above, the permissive convergence criteria of the single-precision solvers allow for fewer iterations in general. However, for the high-resolution meshes, the faster

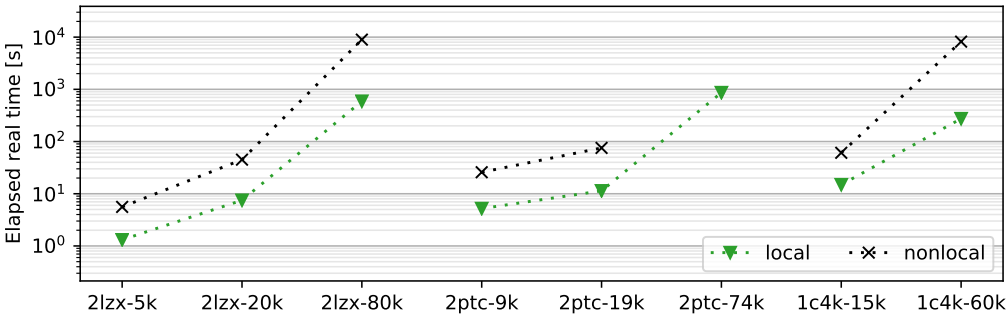
[6] Length values are given in angstroms. `2ptc-74k` is the only dataset with this particular issue.



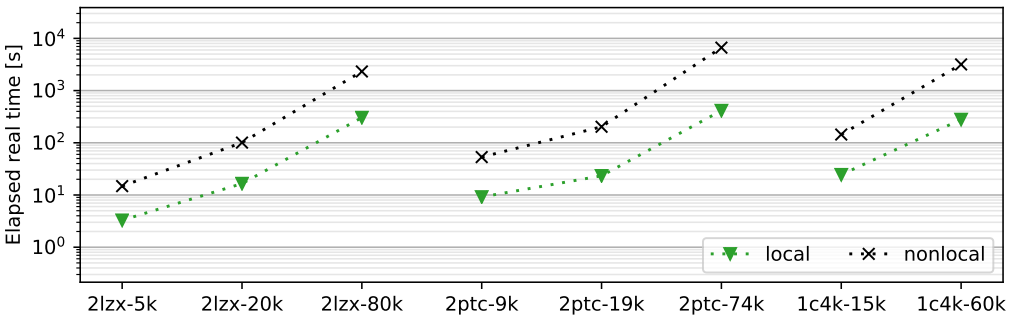
(a) RTX 2070 SUPER: single precision



(b) RTX 2070 SUPER: double precision



(c) TITAN V: single precision



(d) TITAN V: double precision

Figure 9.15. – Elapsed solving time for the local and nonlocal BEM systems using CuNESSie with a restarted and preconditioned GMRES solver. Solving attempts exceeding 3,000 iterations or 8 hours are not shown.

		RTX 2070 SUPER runtime [s]		TITAN V runtime [s]	
System	Dataset	Single prec.	Double prec.	Single prec.	Double prec.
local	2lzx-5k	0.85	12.78	1.31	3.28
	2lzx-20k	8.79	220.72	7.50	16.54
	2lzx-80k	875.49	4010.75	591.08	303.89
	2ptc-9k	3.37	59.14	5.20	9.17
	2ptc-19k	11.16	268.34	11.36	23.30
	2ptc-74k	1346.79	5097.05	864.60	413.42
	1c4k-15k	11.01	154.71	14.89	24.46
	1c4k-60k	386.69	2710.17	274.14	279.15
nonlocal	2lzx-5k	3.84	57.31	5.59	14.79
	2lzx-20k	73.94	1100.30	44.97	100.90
	2lzx-80k	14934.93	21736.05	8957.64	2322.68
	2ptc-9k	19.08	385.67	25.84	53.26
	2ptc-19k	110.84	2214.14	75.48	202.80
	2ptc-74k	(DNF)	(DNF)	(DNF)	6648.82
	1c4k-15k	56.75	1383.62	60.84	143.96
	1c4k-60k	13916.15	27770.35	8196.53	3169.99

Table 9.9. – Elapsed solving time for the local and nonlocal BEM systems using CuNESSie with a restarted and preconditioned GMRES solver. Solving attempts exceeding 3,000 iterations or 8 hours are marked as *did-not-finish* (DNF).

convergence behavior of the double-precision solvers similarly results in faster solving times as compared to the single-precision solvers. Even though we only optimized the matrix-vector products rather than the iterative solvers themselves, the BEM solvers in CuNESSie achieve a nearly uninterrupted 100% GPU utilization. This is in spite of multiple CUDA kernel invocations, host-side code execution, and memory transfers per matrix-vector product. The latter transfers only involve a few megabytes of data (e.g., input and result vectors) at once, rendering the CUDA part of our implementation fully compute-bound.

In conclusion, the TITAN V presents itself as a clear favorite device for the local and nonlocal BEM solvers in CuNESSie. While it is technically outperformed by the RTX 2070 SUPER on the low-resolution meshes, it is able to solve the protein electrostatics problem for both single- and double-precision values in a short amount of time, which is unrivaled by any of the presented CPU-based implementations. Although the solving process for meshes of several hundred thousands of triangles still remains a matter of hours and days, CuNESSie leaves many optimization options that will be targeted in future projects, such as more dataset-specific parallelization and partitioning schemes as well as preconditioners.

10 Conclusion

Accurate models for the electrostatics of biomolecular systems are essential for the study and prediction of long-range protein interactions. Among other things, such interactions build the foundation of the mechanisms behind many diseases and enable the development of counteracting drugs and therapies. One important yet commonly underrepresented factor influencing the long-range visibility of proteins in the cell are polarization effects of the water-based environment. In this manuscript, we have thus focused on electrostatics models for biomolecular system based on the framework of nonlocal electrostatics, which account for solvent molecule interdependencies and were repeatedly shown to provide more realistic energy estimates than commonly used alternatives. In particular, we have developed space-efficient and exact implicit representations for an existing implementation of nonlocal electrostatics, namely, a boundary element method (BEM) for the local and nonlocal protein electrostatics problem in water. The dense and asymmetric nature of the associated matrices previously rendered applications of the BEM approach infeasible for realistically-sized biomolecular systems when no suitable compression schemes for the matrices were present.

Based on our analyses of the BEM systems in Chapter 3, we have developed a number of different implicit matrix and vector types for general-purpose use, complemented by reference implementations for the Julia and Impala languages. We have further shown how to implement specialized operations for these types that can, for example, be used in conjunction with standard iterative solvers for linear systems (cf. Chapter 5). In the case of Julia, we have demonstrated how to inject specialized operations into certain functions by means of function overloads for custom data types. For instance, by providing a CUDA-enabled matrix-vector product implementation for our implicit system matrices, all utilized iterative solvers – originally serial and purely CPU-based – effectively became CUDA-accelerated versions of themselves, without any alteration of their code. Since the presented injection techniques are neither specific to the protein electrostatics problem nor to our implicit types but rather implications of Julia's multiple dispatch mechanism and dynamic type system, they can be transferred to similar application scenarios.

In Chapters 6, 7, and 8, we have presented BEM solvers for the local and nonlocal protein electrostatics problem in Julia and Impala, based on our implicit system representations and specialized matrix operations. All software accompanying this manuscript is open source and freely available online[1].

The NESSie package provides a full-fledged and highly accessible software suite for CPU-based local and nonlocal BEM solvers, with package features including a generic system model with read and write support for several different data formats as well as solvers and post-processors for explicit and implicit BEM systems. NESSie is complemented by the CuNESSie package, which offers CUDA-accelerated variants of NESSie's solvers and post-processors for the implicit BEM systems. Owing to Julia's scripting language character, NESSie and CuNESSie can be used to their full extent by the scientific community (without requiring extensive programming skills) to perform exploratory analyses in the field of nonlocal protein electrostatics.

The same implicit representations, alongside dedicated CPU- and GPU-based implementations for generic matrix-vector products, are alternatively provided in the form of the AnyBEM package for Impala. AnyBEM mostly serves as a showcase example for the AnyDSL framework's partial evaluation (PE) mechanism and as a contrast to Julia's just-in-time (JIT) compilation approach. The domain-specific implementation of our BEM solvers in the package contains most of the program logic in a generic code base that is common to all supported platforms and allows for an easy switching between them. AnyBEM additionally ships with a C++ core containing a high-level system model description, routines for input data preparation, and serves as C/C++ interface to the Impala-based solvers.

In Chapter 9, we have shown the high-performance capabilities of all three BEM solver packages with real-world protein structures. In particular, we have demonstrated that our Julia-based solvers can compete with an optimized reference implementation in C. Apart from that, we have found that both Julia's JIT-based and Impala's PE-based approaches generally yield similar runtimes for the CUDA-accelerated matrix-vector products of the local and nonlocal system matrices, although details vary between implementations, devices, and problem sizes. Overall, all GPU-based matrix-vector products and solvers outperformed their CPU-based counterparts by at least one order of magnitude in terms of runtime, effectively rendering only the GPU-based implementations suitable for larger biomolecular systems, for the time being. Our implicit representations reduce the effective memory footprint of the BEM matrices from terabytes to a few megabytes for most biomolecular systems without any loss of information. Hence, the CUDA-based implementations are not

[1] `https://github.com/tkemmer`

only compute-bound but also, to the best of our knowledge, allow for the first time to solve the original BEM systems in a reasonable time frame, e.g., within a few seconds or minutes for the analyzed systems.

At this point, we see a number of different options for further improvements of our solvers. So far, we have utilized pre-existing iterative solvers where matrix-vector products were the exclusive target of optimization. While we still achieve a GPU utilization of almost 100%, this approach is not necessarily comparable to a CUDA-enabled version of the *whole* iterative solver, as a large number of additional memory transfers and kernel invocations are required. Custom iterative solvers could enable previously inaccessible optimizations that are tailored specifically to the implicit system representations, e.g., by more aggressively reusing matrix elements to avoid expensive recomputations. Since the effect of such optimizations highly depends on the concrete solver, this should trigger a full reevaluation of suitable solvers (and preconditioners). The same considerations naturally apply to the CPU-based parallel matrix-vector products. Especially in the case of Julia, where multithreading support is still experimental as of the time of writing, we are eager to reevaluate the high-performance capabilities of CPU-based solvers in the future.

For our current CUDA-based matrix-vector products, we expect an improved runtime performance when introducing device-specific partitioning schemes and more sophisticated parallelization schemes, eventually also reducing the runtime of the overall solving process. The current implementations always use a fixed block size for partitioning and assign full matrix rows to individual threads, which might limit efficiency for certain combinations of devices and problem sizes. Moreover, additional code inspections will be performed to identify and eliminate avoidable bottlenecks in the kernel functions, e.g., to mitigate diverging branches of execution in the computation of the matrix elements.

We have advertised NESSie as a feature-rich and intuitive package for the nonlocal protein electrostatics problem in Julia. In this spirit, we aim at adding alternative solvers in the future to facilitate the study of different nonlocal models, e.g., those based on different assumptions regarding mobile ion terms or dielectric solvent responses. For the BEM solvers themselves, we will investigate the applicability of compression schemes for the system matrices and compare them to the exact implicit representations. We have already shown in Chapter 6 how our BEM solvers use dedicated result types and multiple dispatch to comply with a uniform solver and post-processor interface. The same techniques can also be used to neatly integrate new functionality into NESSie, without impairing user experience. Due to the inherent similarities of boundary and finite element methods (FEM) as well as the

availability of FEM formulations for nonlocal protein electrostatics, we consider the FEM a suitable candidate for the addition of alternative solvers to NESSie. As a result of our previous work [75], NESSie already supports the required volume mesh-based system models and offers tools to generate such meshes for given protein structures. However, further assessment of existing FEM-related Julia packages, including those of the JuliaFEM project [137], is required to identify beneficial synergetic effects.

Lastly, the current status of the AnyBEM project is best described as a construction site on hold. While the implicit system representations and accelerated matrix-vector products for the local and nonlocal settings are already fully functional, we do not provide any BEM solvers right now. This is mostly owing to the absence of usable third-party libraries, which slows down development cycles substantially in contrast to Julia. Apart from that, the limited accessibility of Impala-based software in direct comparison to Julia further lowered the priority of a focus on AnyBEM in the past. However, our performance analyses with real protein structures have shown that the partial evaluation approach harbors a lot of unused potential. More specifically, the CUDA-accelerated matrix-vector products in Impala slightly but regularly outperformed their Julia counterparts. What is more, adding support for additional platforms (e.g., for GPUs from other vendors) boils down to implementing only a few platform-specific functions, as long as suitable LLVM back ends for Impala exist. The C++ core of AnyBEM can alternatively be extended to include fully C++-based solver variants based on the Impala counterparts, if need be. With the additional expectation that more and more general-purpose libraries for the AnyDSL framework will become available as time progresses, we will be happy to revisit and finalize the AnyBEM project in the future.

Test model functions

A

In this chapter, we summarize the equations for the local or nonlocal electrostatic potential functions as implemented in the `TestModel` module of our NESSie.jl package for the Julia programming language. Short descriptions of the models can be found in Section 6.4 earlier in this manuscript. Each of the following sections includes the original sources for the potential functions.

A.1 Electrostatic potentials for the Born test model

The interior and exterior potentials for the Born test model in the `NESSie.TestModel` module, based on the derivations presented in [49, Section 3.5.1], can be computed for a point charge ζ, located at the origin and surrounded by a vacuum-filled sphere ($\varepsilon_\Omega = 1$) with radius a, can be computed through

$$\begin{aligned}
\varphi_\Omega^{\text{local}}(\boldsymbol{\xi}) &= \frac{\zeta}{4\pi\varepsilon_0}\left\{\frac{1}{|\boldsymbol{\xi}|} + \frac{1}{a}\left(\frac{1}{\varepsilon_\Sigma} - 1\right)\right\}, && \boldsymbol{\xi} \in \Omega \\
\varphi_\Sigma^{\text{local}}(\boldsymbol{\xi}) &= \frac{\zeta}{4\pi\varepsilon_0\varepsilon_\Sigma}\frac{1}{|\boldsymbol{\xi}|}. && \boldsymbol{\xi} \in \Sigma
\end{aligned}$$

The nonlocal versions of the potentials can be computed as

$$\begin{aligned}
\varphi_\Omega^{\text{nonlocal}}(\boldsymbol{\xi}) &= \frac{\zeta}{4\pi\varepsilon_0}\left\{\frac{1}{|\boldsymbol{\xi}|} + \frac{1}{a\varepsilon_\Sigma}\left(1 - \varepsilon_\Sigma + \frac{\varepsilon_\Sigma - \varepsilon_\infty}{\varepsilon_\infty}\frac{\sinh\nu}{\nu}\mathrm{e}^{-\nu}\right)\right\}, && \boldsymbol{\xi} \in \Omega \\
\varphi_\Sigma^{\text{nonlocal}}(\boldsymbol{\xi}) &= \frac{\zeta}{4\pi\varepsilon_0\varepsilon_\Sigma}\frac{1}{|\boldsymbol{\xi}|}\left\{1 + \frac{\varepsilon_\Sigma - \varepsilon_\infty}{\varepsilon_\infty}\frac{\sinh\nu}{\nu}\mathrm{e}^{-\nu\frac{|\boldsymbol{\xi}|}{a}}\right\}, && \boldsymbol{\xi} \in \Sigma
\end{aligned}$$

where the quantity ν is defined as

$$\nu := \sqrt{\frac{\varepsilon_\Sigma}{\varepsilon_\infty}}\frac{a}{\lambda}.$$

A.2 Electrostatic potentials for the Poisson test model

The interior and exterior potentials for the nonlocal Poisson test model in the `NESSie.TestModel` module, which is based on the *first nonlocal Poisson test model* from [151], can be computed for a sphere with radius a and N point charges modeled according to Eq. (2.1) through the following equations:

$$\varphi_\Omega(\boldsymbol{\xi}) = \frac{e_c}{\varepsilon_0}\sum_{j=1}^{N}\zeta_j\left\{\sum_{n=0}^{\infty}A_{3n}|\boldsymbol{\xi}|P_n\left(\frac{\boldsymbol{r}_j\cdot\boldsymbol{\xi}}{|\boldsymbol{r}_j||\boldsymbol{\xi}|}\right)+\frac{1}{4\pi\varepsilon_\Omega|\boldsymbol{\xi}-\boldsymbol{r}_j|}\right\} \qquad \boldsymbol{\xi}\in\Omega$$

and

$$\varphi_\Sigma(\boldsymbol{\xi}) = \frac{e_c}{\varepsilon_0}\sum_{j=1}^{N}\zeta_j\left\{\sum_{n=0}^{\infty}\left[\frac{\varepsilon_\infty-\varepsilon_\Sigma}{\varepsilon_\infty}A_{2n}k_n(\kappa|\boldsymbol{\xi}|)+\frac{A_{1n}}{|\boldsymbol{\xi}|^{n+1}}\right]P_n\left(\frac{\boldsymbol{r}_j\cdot\boldsymbol{\xi}}{|\boldsymbol{r}_j||\boldsymbol{\xi}|}\right)\right\}, \quad \boldsymbol{\xi}\in\Sigma$$

where P_n is the Legendre polynomial of degree n [157, Chapter 18] and the quantities A_{in} for $i=1,2,3$ are defined as

$$\begin{aligned}
A_{1n} &= \frac{2n+1}{4\pi d_n a}\left[\left(\frac{|\boldsymbol{r}_j|}{a}\right)^n w_n + \frac{n\lambda(\varepsilon_\infty-\varepsilon_\Sigma)}{a\varepsilon_\infty}i_n\left(\frac{|\boldsymbol{r}_j|}{\lambda}\right)k_n(\kappa a)\right],\\
A_{2n} &= \frac{(2n+1)\lambda}{4\pi\varepsilon_\Sigma d_n a^{n+3}}\left[(\varepsilon_\Sigma-\varepsilon_\Omega)(n+1)\frac{|\boldsymbol{r}_j|^n}{a^n}i_n\left(\frac{a}{\lambda}\right)-[n(\varepsilon_\Omega-\varepsilon_\Sigma)+\varepsilon_\Sigma]i_n\left(\frac{|\boldsymbol{r}_j|}{\lambda}\right)\right],\\
A_{3n} &= \frac{A_{1n}}{a^{2n+1}}+A_{2n}\frac{\varepsilon_\infty-\varepsilon_\Sigma}{\varepsilon_\infty a^n}k_n(\kappa a)-\frac{|\boldsymbol{r}_j|^n}{4\pi\varepsilon_\Omega a^{2n+1}},
\end{aligned}$$

with $i_n(z)$ [157, Eq. 10.47.7] and $k_n(z)$ [157, Eq. 10.47.9] being the modified spherical Bessel functions and

$$\begin{aligned}
\kappa &= \frac{1}{\lambda}\sqrt{\frac{\varepsilon_\Sigma}{\varepsilon_\Omega}},\\
d_n &= \frac{n(2n+1)\lambda\varepsilon_\Omega(\varepsilon_\Sigma-\varepsilon_\infty)}{a^{n+2}\varepsilon_\infty}i_n\left(\frac{a}{\lambda}\right)k_n(\kappa a)\\
&\quad+\frac{n(\varepsilon_\Omega+(n+1)\varepsilon_\Sigma)}{a^{n+1}}\left[\frac{\varepsilon_\Sigma}{\varepsilon_\infty}i_{n+1}\left(\frac{a}{\lambda}\right)k_n(\kappa a)+\kappa\lambda i_n\left(\frac{a}{\lambda}\right)k_{n+1}(\kappa a)\right],\\
w_n &= \frac{n\lambda}{a}\frac{\varepsilon_\Sigma-\varepsilon_\infty}{\varepsilon_\infty}i_n\left(\frac{a}{\lambda}\right)k_n(\kappa a)+\frac{\varepsilon_\Sigma}{\varepsilon_\infty}i_{n+1}\left(\frac{a}{\lambda}\right)k_n(\kappa a)+\kappa\lambda i_n\left(\frac{a}{\lambda}\right)k_{n+1}(\kappa a).
\end{aligned}$$

Additional tables and figures

B

Name	Surface model	Volume model	Charge model	Generator
HMO+	✓	–	✓	NESSie.jl
Mcsf	–	✓	–	GAMer [139]
MSMS	✓	–	–	MSMS [158]
OFF	✓	–	–	GAMer
PQR	–	–	✓	PDB2PQR [138]
STL	–	✓	–	NESSie.jl

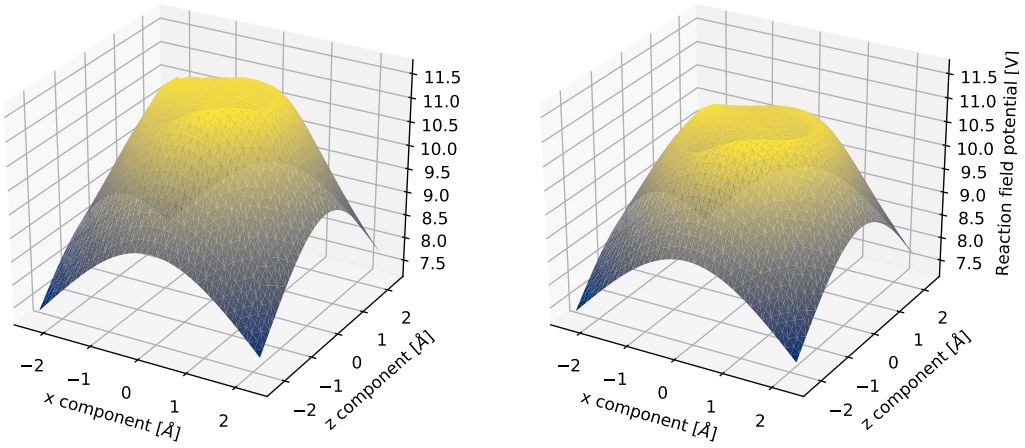
Table B.1. – Input formats supported by NESSie.jl. Mcsf support was reverse-engineered from the mesh files. The HMO+ format specification can be found in Section 6.2.3, all others in Table B.3.

Name	Point cloud	Surface model	Volume model	Charge model
HMO+	–	✓	–	✓
SKEL	–	✓	✓	–
STL	–	✓	–	–
VTK	–	✓	✓	–
XML3D	✓	✓	–	–

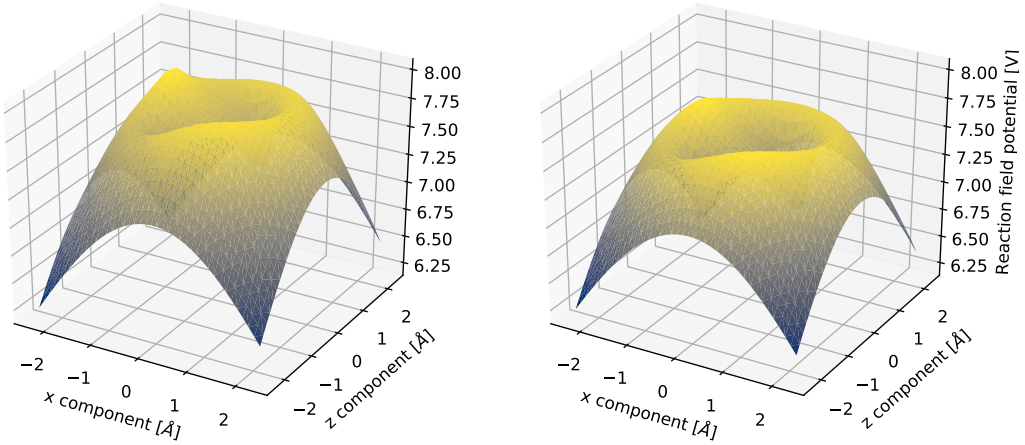
Table B.2. – Output formats supported by NESSie.jl. Check marks refer to the information included in the resulting output file. The HMO+ format specification can be found in Section 6.2.3, all others in Table B.3.

Format	Specification
HMO	`https://espace.cern.ch/roxie/Documentation/ironFileDoc.pdf`
OFF	`http://www.geomview.org/docs/html/OFF.html`
PQR	`https://apbs-pdb2pqr.readthedocs.io/en/latest/formats/pqr.html`
SKEL	`http://www.geomview.org/docs/html/SKEL.html`
STL	`http://www.fabbers.com/tech/STL_Format`
VTK	`http://www.vtk.org/VTK/img/file-formats.pdf`
XML3D	`https://github.com/xml3d/xml3d.js/wiki/External-resources`

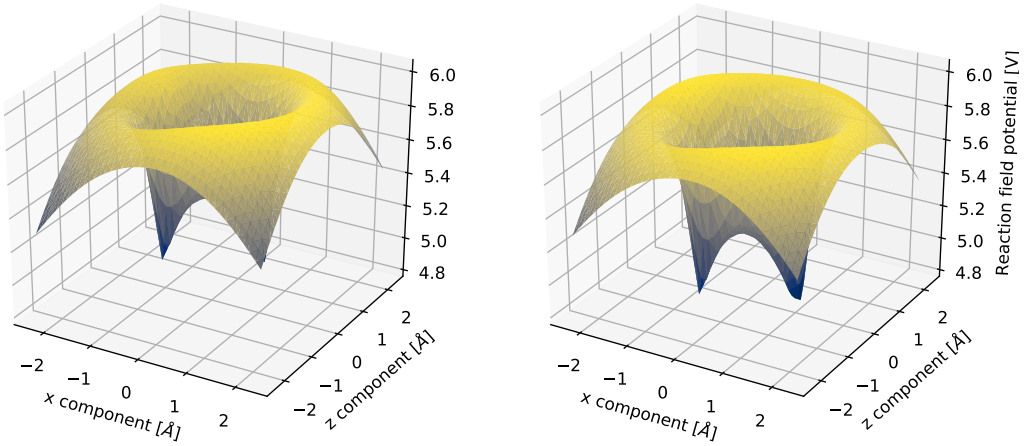
Table B.3. – Specifications used for the data format implementations in NESSie.jl. HMO refers to the original, unmodified format; for the specification of the HMO+ format, see Section 6.2.3. Web resources are shown as of the time of writing (June 2020), with an updated version available in the NESSie documentation.



(a) $\lambda = 10$

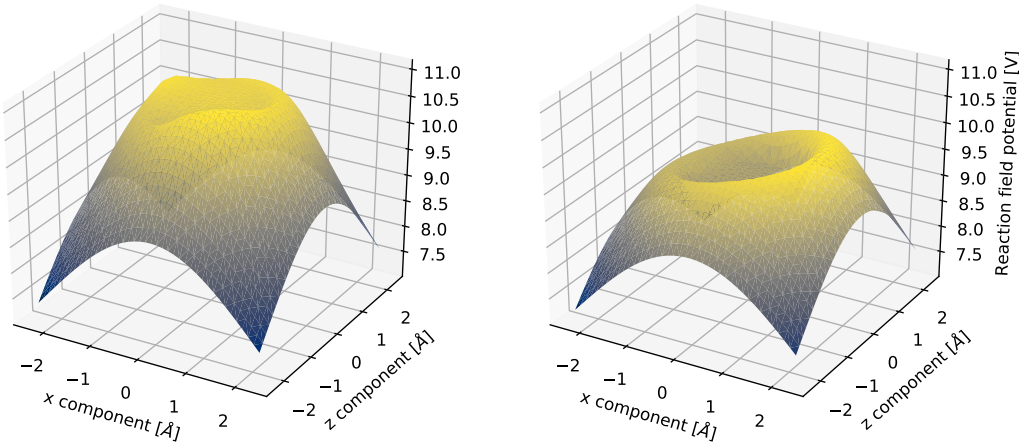


(b) $\lambda = 15$

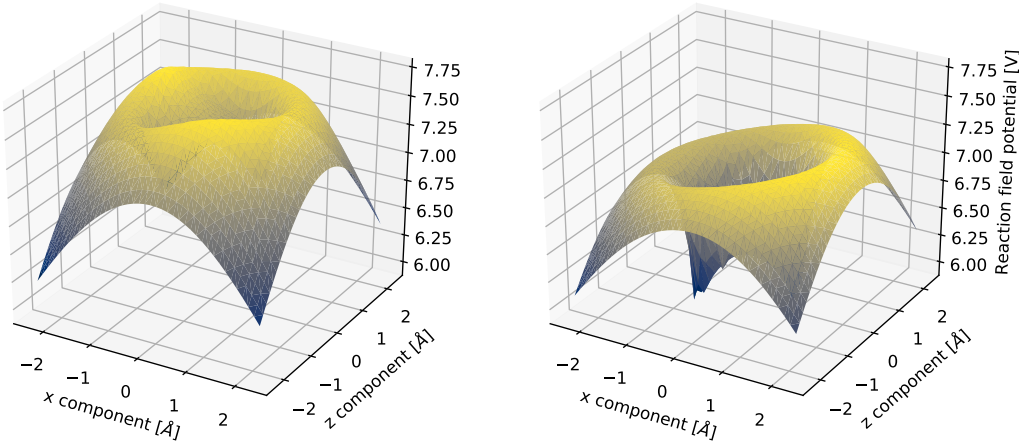


(c) $\lambda = 20$

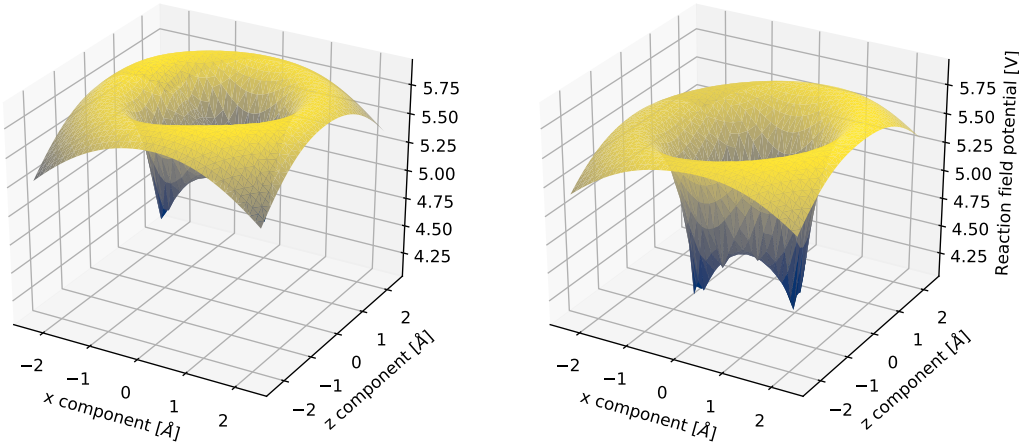
Figure B.1. – Reaction field potentials for the nonlocal Poisson test model based on the `21zx` dataset embedded into an origin-centered unit sphere, with $\varepsilon_\Omega = 2$, $\varepsilon_\Sigma = 80$, $\varepsilon_\infty = 2$, and varying λ. The potentials correspond to the first (left) and second (right) test model variants, computed with the Fortran package reported in [151] for a section of the xz-plane.



(a) $\lambda = 10$



(b) $\lambda = 15$



(c) $\lambda = 20$

Figure B.2. – Reaction field potentials for the nonlocal Poisson test model based on the 2lzx dataset embedded into an origin-centered unit sphere, with $\varepsilon_\Omega = 2$, $\varepsilon_\Sigma = 80$, $\varepsilon_\infty = 2$, and varying λ. The potentials on the left correspond to the second test model variant computed with the Fortran package reported in [151]. The right-hand plots show the results computed with our BEM solvers.

		Single-precision runtime [s]				Double-precision runtime [s]			
Dataset	Matrix	min	max	std	avg	min	max	std	avg
2lzx-5k	$\mathcal{V}$	0.35	0.48	0.044	**0.43**	1.70	1.73	0.011	**1.71**
	$\mathcal{K}$	0.34	0.43	0.042	**0.37**	1.34	1.57	0.076	**1.41**
	$\mathcal{V}^R$	0.32	0.33	0.003	**0.32**	0.59	0.78	0.069	**0.67**
	$\mathcal{K}^R$	0.41	0.43	0.006	**0.42**	0.64	0.69	0.013	**0.66**
	local	0.33	0.36	0.010	**0.35**	1.40	1.58	0.053	**1.44**
	nonlocal	1.44	1.88	0.147	**1.60**	4.23	4.46	0.097	**4.32**
2lzx-20k	$\mathcal{V}$	5.86	6.49	0.227	**6.07**	26.57	27.51	0.359	**26.91**
	$\mathcal{K}$	5.45	5.97	0.198	**5.63**	21.13	21.57	0.131	**21.28**
	$\mathcal{V}^R$	4.99	5.05	0.019	**5.02**	9.23	9.52	0.086	**9.32**
	$\mathcal{K}^R$	6.49	6.54	0.015	**6.50**	10.03	10.27	0.069	**10.08**
	local	5.46	5.57	0.032	**5.49**	21.10	21.29	0.057	**21.17**
	nonlocal	22.87	23.05	0.059	**22.92**	67.13	68.17	0.314	**67.39**
2lzx-80k	$\mathcal{V}$	91.97	92.51	0.179	**92.17**	413.16	414.46	0.425	**413.76**
	$\mathcal{K}$	85.81	86.78	0.283	**86.00**	332.42	335.64	0.924	**333.99**
	$\mathcal{V}^R$	79.91	80.49	0.164	**80.17**	147.32	147.76	0.174	**147.48**
	$\mathcal{K}^R$	103.83	104.14	0.114	**103.96**	160.25	166.92	2.425	**163.20**
	local	86.41	86.73	0.107	**86.54**	333.22	358.59	8.007	**342.47**
	nonlocal	363.56	365.80	0.750	**364.08**	–	–	–	**(DNF)**
2ptc-9k	$\mathcal{V}$	1.27	1.64	0.135	**1.35**	6.04	6.35	0.102	**6.11**
	$\mathcal{K}$	1.19	1.24	0.015	**1.21**	4.85	5.65	0.227	**5.05**
	$\mathcal{V}^R$	1.13	1.35	0.067	**1.18**	2.09	2.10	0.005	**2.09**
	$\mathcal{K}^R$	1.46	1.48	0.007	**1.47**	2.27	2.30	0.011	**2.28**
	local	1.18	1.23	0.013	**1.20**	4.84	4.91	0.022	**4.86**
	nonlocal	5.06	5.81	0.228	**5.27**	15.27	15.32	0.018	**15.29**
2ptc-19k	$\mathcal{V}$	4.81	5.02	0.078	**4.90**	23.23	24.52	0.398	**23.46**
	$\mathcal{K}$	4.52	5.39	0.246	**4.75**	18.52	18.61	0.033	**18.55**
	$\mathcal{V}^R$	4.33	4.37	0.012	**4.35**	8.02	8.20	0.058	**8.07**
	$\mathcal{K}^R$	5.62	5.69	0.020	**5.65**	8.73	8.78	0.018	**8.75**
	local	4.54	4.59	0.017	**4.56**	18.50	18.59	0.038	**18.54**
	nonlocal	19.38	19.86	0.165	**19.51**	58.78	60.74	0.679	**59.53**
2ptc-74k	$\mathcal{V}$	80.13	80.89	0.216	**80.36**	361.73	365.48	1.401	**363.16**
	$\mathcal{K}$	74.64	74.99	0.121	**74.81**	291.23	292.82	0.498	**291.72**
	$\mathcal{V}^R$	69.46	69.81	0.112	**69.62**	127.90	129.98	0.602	**128.35**
	$\mathcal{K}^R$	90.23	91.26	0.361	**90.59**	139.68	140.27	0.169	**139.98**
	local	74.84	75.20	0.118	**75.03**	291.19	298.63	3.025	**294.32**
	nonlocal	314.65	315.30	0.175	**314.88**	924.06	949.48	9.540	**930.26**
1c4k-15k	$\mathcal{V}$	3.21	3.45	0.086	**3.29**	15.67	16.52	0.268	**15.99**
	$\mathcal{K}$	2.99	3.02	0.008	**3.01**	12.42	12.62	0.076	**12.49**
	$\mathcal{V}^R$	2.90	3.42	0.165	**2.99**	5.38	5.51	0.045	**5.43**
	$\mathcal{K}^R$	3.77	4.05	0.083	**3.85**	5.86	6.07	0.066	**5.91**
	local	3.00	3.04	0.011	**3.02**	12.51	12.59	0.026	**12.54**
	nonlocal	12.88	12.97	0.027	**12.93**	39.63	39.82	0.060	**39.75**
1c4k-60k	$\mathcal{V}$	49.97	51.12	0.345	**50.18**	241.25	242.55	0.370	**241.66**
	$\mathcal{K}$	47.10	47.31	0.078	**47.20**	194.32	197.91	1.094	**195.17**
	$\mathcal{V}^R$	45.79	45.87	0.029	**45.82**	84.50	84.79	0.092	**84.64**
	$\mathcal{K}^R$	59.42	60.19	0.224	**59.58**	92.02	92.95	0.264	**92.27**
	local	47.36	47.59	0.073	**47.44**	194.16	194.77	0.178	**194.31**
	nonlocal	203.14	203.82	0.188	**203.43**	614.67	617.77	0.889	**615.65**

Table B.4. – Runtime evaluation for the CPU-based matrix-vector product implementations in AnyBEM on the first workstation (8 threads). Computations exceeding 1,000 seconds are marked as *did-not-finish* (DNF).

		Single-precision runtime [s]				Double-precision runtime [s]			
Dataset	Matrix	min	max	std	avg	min	max	std	avg
2lzx-5k	$\mathcal{V}$	0.25	0.28	0.008	**0.26**	0.40	0.41	0.005	**0.40**
	$\mathcal{K}$	0.21	0.22	0.003	**0.21**	0.31	0.32	0.003	**0.32**
	$\mathcal{V}^R$	0.18	0.18	0.001	**0.18**	0.21	0.23	0.004	**0.22**
	$\mathcal{K}^R$	0.19	0.20	0.002	**0.19**	0.23	0.24	0.003	**0.23**
	local	0.21	0.22	0.004	**0.21**	0.31	0.32	0.003	**0.32**
	nonlocal	0.84	0.86	0.007	**0.85**	1.15	1.31	0.047	**1.18**
2lzx-20k	$\mathcal{V}$	3.86	3.90	0.011	**3.88**	6.10	6.17	0.025	**6.12**
	$\mathcal{K}$	3.15	3.16	0.003	**3.15**	4.80	4.92	0.040	**4.83**
	$\mathcal{V}^R$	2.69	2.74	0.014	**2.70**	3.28	3.30	0.007	**3.29**
	$\mathcal{K}^R$	2.93	3.26	0.101	**2.97**	3.50	3.54	0.009	**3.51**
	local	3.14	3.24	0.030	**3.16**	4.80	4.87	0.022	**4.82**
	nonlocal	12.76	12.80	0.012	**12.78**	17.70	17.75	0.014	**17.73**
2lzx-80k	$\mathcal{V}$	60.66	60.72	0.018	**60.69**	96.72	97.03	0.096	**96.88**
	$\mathcal{K}$	49.22	49.36	0.049	**49.28**	75.80	76.41	0.161	**76.00**
	$\mathcal{V}^R$	42.97	43.31	0.108	**43.05**	52.40	52.52	0.041	**52.47**
	$\mathcal{K}^R$	46.82	46.94	0.042	**46.89**	55.92	56.15	0.066	**55.97**
	local	49.19	49.41	0.072	**49.25**	75.77	76.02	0.093	**75.87**
	nonlocal	201.48	201.92	0.133	**201.67**	280.87	281.97	0.328	**281.22**
2ptc-9k	$\mathcal{V}$	0.89	0.91	0.005	**0.90**	1.40	1.41	0.004	**1.40**
	$\mathcal{K}$	0.73	0.77	0.012	**0.74**	1.10	1.12	0.005	**1.11**
	$\mathcal{V}^R$	0.61	0.63	0.005	**0.62**	0.75	0.76	0.004	**0.76**
	$\mathcal{K}^R$	0.67	0.68	0.003	**0.67**	0.80	0.81	0.003	**0.80**
	local	0.73	0.73	0.001	**0.73**	1.10	1.12	0.004	**1.11**
	nonlocal	2.94	3.21	0.085	**2.97**	4.06	4.07	0.005	**4.07**
2ptc-19k	$\mathcal{V}$	3.38	3.51	0.038	**3.40**	5.32	5.36	0.011	**5.34**
	$\mathcal{K}$	2.76	2.77	0.005	**2.76**	4.21	4.30	0.031	**4.23**
	$\mathcal{V}^R$	2.34	2.44	0.030	**2.36**	2.85	2.89	0.011	**2.86**
	$\mathcal{K}^R$	2.55	2.56	0.003	**2.56**	3.05	3.07	0.007	**3.05**
	local	2.76	2.76	0.003	**2.76**	4.20	4.27	0.019	**4.22**
	nonlocal	11.14	11.32	0.053	**11.17**	15.49	15.57	0.025	**15.54**
2ptc-74k	$\mathcal{V}$	52.79	52.99	0.057	**52.88**	84.90	85.21	0.092	**85.04**
	$\mathcal{K}$	42.93	43.01	0.021	**42.96**	66.29	66.63	0.106	**66.42**
	$\mathcal{V}^R$	37.31	37.48	0.055	**37.36**	45.51	45.80	0.093	**45.61**
	$\mathcal{K}^R$	40.66	40.75	0.029	**40.70**	48.61	48.84	0.079	**48.68**
	local	42.90	43.08	0.063	**42.98**	66.15	66.30	0.050	**66.24**
	nonlocal	175.25	175.91	0.203	**175.49**	245.39	246.31	0.269	**245.81**
1c4k-15k	$\mathcal{V}$	2.27	2.29	0.006	**2.28**	3.58	3.72	0.042	**3.61**
	$\mathcal{K}$	1.85	1.86	0.004	**1.86**	2.83	2.86	0.007	**2.84**
	$\mathcal{V}^R$	1.57	1.58	0.005	**1.58**	1.92	1.96	0.013	**1.93**
	$\mathcal{K}^R$	1.71	1.81	0.030	**1.73**	2.05	2.07	0.006	**2.06**
	local	1.85	1.86	0.002	**1.86**	2.82	2.84	0.004	**2.83**
	nonlocal	7.50	7.52	0.008	**7.51**	10.40	10.46	0.025	**10.42**
1c4k-60k	$\mathcal{V}$	35.19	35.43	0.072	**35.26**	56.38	56.74	0.122	**56.55**
	$\mathcal{K}$	28.75	28.85	0.029	**28.79**	44.21	44.43	0.068	**44.27**
	$\mathcal{V}^R$	24.67	24.79	0.041	**24.73**	30.09	30.26	0.048	**30.15**
	$\mathcal{K}^R$	26.89	26.96	0.020	**26.92**	32.14	32.21	0.019	**32.16**
	local	28.72	28.84	0.031	**28.78**	44.17	44.38	0.069	**44.25**
	nonlocal	116.53	116.73	0.068	**116.64**	163.00	163.33	0.099	**163.14**
6ds5-170k	$\mathcal{V}$	276.32	276.88	0.178	**276.49**	446.32	447.22	0.281	**446.59**
	$\mathcal{K}$	225.38	225.85	0.152	**225.68**	349.62	350.11	0.152	**349.81**
	$\mathcal{V}^R$	196.03	196.43	0.116	**196.13**	239.15	239.65	0.169	**239.37**
	$\mathcal{K}^R$	213.68	214.20	0.157	**213.82**	255.47	256.00	0.181	**255.72**
	local	225.12	225.50	0.144	**225.27**	349.64	350.66	0.346	**350.06**
	nonlocal	919.11	927.78	2.622	**920.37**	–	–	–	**(DNF)**
2h1l-200k	$\mathcal{V}$	382.68	383.32	0.174	**382.91**	617.29	618.48	0.418	**617.79**
	$\mathcal{K}$	312.70	312.94	0.090	**312.78**	484.95	488.13	0.972	**485.48**
	$\mathcal{V}^R$	272.34	272.69	0.123	**272.48**	333.11	334.48	0.389	**333.53**
	$\mathcal{K}^R$	296.81	297.10	0.084	**296.93**	352.49	353.04	0.156	**352.75**
	local	312.41	312.80	0.117	**312.65**	485.58	486.67	0.304	**485.90**
	nonlocal	–	–	–	**(DNF)**	–	–	–	**(DNF)**

Table B.5. – Runtime evaluation for the CPU-based matrix-vector product implementations in AnyBEM on the second workstation (64 threads). Computations exceeding 1,000 seconds are marked as *did-not-finish* (DNF).

		Single-precision runtime [s]				Double-precision runtime [s]			
Dataset	Matrix	min	max	std	avg	min	max	std	avg
2lzx-5k	$\mathcal{V}$	0.02	0.03	0.003	**0.02**	0.21	0.21	0.000	**0.21**
	$\mathcal{K}$	0.02	0.02	0.000	**0.02**	0.17	0.17	0.000	**0.17**
	$\mathcal{V}^R$	0.01	0.01	0.000	**0.01**	0.07	0.07	0.000	**0.07**
	$\mathcal{K}^R$	0.01	0.01	0.000	**0.01**	0.07	0.07	0.000	**0.07**
	local	0.02	0.02	0.000	**0.02**	0.17	0.17	0.000	**0.17**
	nonlocal	0.06	0.08	0.005	**0.07**	0.53	0.53	0.000	**0.53**
2lzx-20k	$\mathcal{V}$	0.21	0.21	0.001	**0.21**	3.35	3.35	0.000	**3.35**
	$\mathcal{K}$	0.19	0.19	0.001	**0.19**	2.70	2.70	0.000	**2.70**
	$\mathcal{V}^R$	0.18	0.18	0.001	**0.18**	1.11	1.11	0.000	**1.11**
	$\mathcal{K}^R$	0.18	0.18	0.001	**0.18**	1.18	1.18	0.000	**1.18**
	local	0.19	0.20	0.001	**0.19**	2.70	2.70	0.000	**2.70**
	nonlocal	0.79	0.80	0.002	**0.80**	8.34	8.34	0.000	**8.34**
2lzx-80k	$\mathcal{V}$	4.50	4.53	0.011	**4.51**	49.73	50.09	0.121	**49.92**
	$\mathcal{K}$	4.37	4.40	0.010	**4.38**	39.45	41.26	0.613	**39.91**
	$\mathcal{V}^R$	4.34	4.38	0.011	**4.36**	17.71	17.72	0.003	**17.72**
	$\mathcal{K}^R$	4.37	4.39	0.009	**4.38**	18.83	18.83	0.002	**18.83**
	local	4.37	4.40	0.008	**4.39**	39.48	40.75	0.353	**39.81**
	nonlocal	18.02	18.09	0.020	**18.06**	126.03	126.54	0.145	**126.28**
2ptc-9k	$\mathcal{V}$	0.05	0.06	0.005	**0.05**	0.79	0.79	0.000	**0.79**
	$\mathcal{K}$	0.04	0.04	0.000	**0.04**	0.64	0.64	0.000	**0.64**
	$\mathcal{V}^R$	0.02	0.02	0.000	**0.02**	0.26	0.26	0.000	**0.26**
	$\mathcal{K}^R$	0.02	0.02	0.000	**0.02**	0.27	0.27	0.000	**0.27**
	local	0.04	0.04	0.000	**0.04**	0.64	0.64	0.000	**0.64**
	nonlocal	0.14	0.17	0.011	**0.14**	1.96	1.98	0.009	**1.97**
2ptc-19k	$\mathcal{V}$	0.17	0.17	0.001	**0.17**	3.00	3.00	0.000	**3.00**
	$\mathcal{K}$	0.15	0.15	0.001	**0.15**	2.41	2.43	0.009	**2.42**
	$\mathcal{V}^R$	0.12	0.12	0.000	**0.12**	1.02	1.02	0.000	**1.02**
	$\mathcal{K}^R$	0.12	0.12	0.001	**0.12**	1.09	1.09	0.000	**1.09**
	local	0.15	0.15	0.001	**0.15**	2.43	2.43	0.000	**2.43**
	nonlocal	0.64	0.65	0.001	**0.65**	7.57	7.63	0.020	**7.62**
2ptc-74k	$\mathcal{V}$	3.73	3.75	0.005	**3.74**	44.01	44.29	0.080	**44.18**
	$\mathcal{K}$	3.60	3.62	0.005	**3.61**	35.05	35.26	0.080	**35.16**
	$\mathcal{V}^R$	3.63	3.67	0.011	**3.65**	15.46	15.48	0.007	**15.47**
	$\mathcal{K}^R$	3.66	3.69	0.009	**3.67**	16.40	16.42	0.005	**16.41**
	local	3.60	3.63	0.009	**3.61**	35.10	35.38	0.084	**35.22**
	nonlocal	14.94	15.01	0.024	**14.98**	110.98	111.61	0.227	**111.27**
1c4k-15k	$\mathcal{V}$	0.09	0.10	0.004	**0.09**	1.90	1.90	0.000	**1.90**
	$\mathcal{K}$	0.08	0.08	0.000	**0.08**	1.53	1.53	0.000	**1.53**
	$\mathcal{V}^R$	0.06	0.06	0.000	**0.06**	0.63	0.63	0.000	**0.63**
	$\mathcal{K}^R$	0.06	0.06	0.000	**0.06**	0.67	0.67	0.000	**0.67**
	local	0.08	0.08	0.000	**0.08**	1.53	1.53	0.000	**1.53**
	nonlocal	0.32	0.32	0.001	**0.32**	4.73	4.73	0.000	**4.73**
1c4k-60k	$\mathcal{V}$	2.42	2.48	0.026	**2.45**	28.80	29.01	0.082	**28.86**
	$\mathcal{K}$	2.33	2.37	0.013	**2.34**	22.84	22.97	0.033	**22.91**
	$\mathcal{V}^R$	2.31	2.37	0.016	**2.33**	10.07	10.07	0.001	**10.07**
	$\mathcal{K}^R$	2.34	2.37	0.010	**2.35**	10.67	10.67	0.001	**10.67**
	local	2.33	2.37	0.013	**2.34**	22.87	23.03	0.065	**22.95**
	nonlocal	9.65	9.75	0.031	**9.69**	72.52	72.67	0.048	**72.61**
6ds5-170k	$\mathcal{V}$	20.75	20.89	0.044	**20.80**	231.60	232.37	0.267	**231.99**
	$\mathcal{K}$	20.14	20.25	0.045	**20.20**	184.85	185.68	0.258	**185.32**
	$\mathcal{V}^R$	21.26	21.32	0.023	**21.29**	80.35	80.37	0.007	**80.36**
	$\mathcal{K}^R$	21.42	21.49	0.026	**21.46**	85.15	85.15	0.001	**85.15**
	local	20.15	20.30	0.044	**20.24**	184.79	185.74	0.307	**185.16**
	nonlocal	88.94	89.21	0.110	**89.04**	581.91	584.16	0.645	**583.20**
2h1l-200k	$\mathcal{V}$	28.22	28.41	0.063	**28.33**	314.78	315.74	0.361	**315.30**
	$\mathcal{K}$	27.56	27.68	0.040	**27.62**	249.79	250.82	0.292	**250.23**
	$\mathcal{V}^R$	28.18	28.23	0.017	**28.20**	111.74	111.80	0.026	**111.77**
	$\mathcal{K}^R$	28.31	28.41	0.028	**28.37**	118.02	118.07	0.020	**118.03**
	local	27.57	27.72	0.047	**27.61**	249.71	250.84	0.356	**250.17**
	nonlocal	113.11	113.34	0.077	**113.25**	795.23	796.76	0.422	**795.94**

Table B.6. – Runtime evaluation for the GPU-based matrix-vector product implementations in CuNESSie on the first workstation (RTX 2070 SUPER)

		Single-precision runtime [s]				Double-precision runtime [s]			
Dataset	Matrix	min	max	std	avg	min	max	std	avg
2lzx-5k	$\mathcal{V}$	0.04	0.04	0.002	**0.04**	0.05	0.05	0.000	**0.05**
	$\mathcal{K}$	0.03	0.03	0.000	**0.03**	0.04	0.04	0.000	**0.04**
	$\mathcal{V}^R$	0.02	0.02	0.000	**0.02**	0.02	0.02	0.000	**0.02**
	$\mathcal{K}^R$	0.02	0.02	0.000	**0.02**	0.02	0.02	0.000	**0.02**
	local	0.03	0.03	0.000	**0.03**	0.04	0.04	0.000	**0.04**
	nonlocal	0.10	0.11	0.004	**0.10**	0.13	0.13	0.000	**0.13**
2lzx-20k	$\mathcal{V}$	0.16	0.16	0.001	**0.16**	0.23	0.23	0.000	**0.23**
	$\mathcal{K}$	0.13	0.13	0.000	**0.13**	0.18	0.18	0.001	**0.18**
	$\mathcal{V}^R$	0.09	0.09	0.001	**0.09**	0.16	0.16	0.001	**0.16**
	$\mathcal{K}^R$	0.08	0.08	0.001	**0.08**	0.16	0.16	0.000	**0.16**
	local	0.13	0.13	0.000	**0.13**	0.18	0.18	0.001	**0.18**
	nonlocal	0.48	0.49	0.001	**0.49**	0.74	0.75	0.002	**0.75**
2lzx-80k	$\mathcal{V}$	2.73	2.86	0.042	**2.79**	3.61	3.70	0.024	**3.65**
	$\mathcal{K}$	2.65	2.73	0.030	**2.68**	3.46	3.51	0.017	**3.49**
	$\mathcal{V}^R$	2.51	2.61	0.025	**2.56**	3.06	3.10	0.014	**3.08**
	$\mathcal{K}^R$	2.54	2.62	0.023	**2.57**	3.07	3.11	0.013	**3.09**
	local	2.63	2.74	0.037	**2.67**	3.44	3.51	0.024	**3.48**
	nonlocal	10.30	10.45	0.040	**10.36**	13.24	13.31	0.023	**13.28**
2ptc-9k	$\mathcal{V}$	0.07	0.08	0.002	**0.07**	0.09	0.09	0.000	**0.09**
	$\mathcal{K}$	0.05	0.05	0.000	**0.05**	0.07	0.08	0.000	**0.07**
	$\mathcal{V}^R$	0.03	0.03	0.000	**0.03**	0.04	0.04	0.000	**0.04**
	$\mathcal{K}^R$	0.03	0.03	0.000	**0.03**	0.04	0.04	0.000	**0.04**
	local	0.05	0.05	0.000	**0.05**	0.07	0.08	0.000	**0.08**
	nonlocal	0.19	0.21	0.006	**0.19**	0.25	0.26	0.002	**0.25**
2ptc-19k	$\mathcal{V}$	0.15	0.15	0.001	**0.15**	0.21	0.22	0.001	**0.21**
	$\mathcal{K}$	0.12	0.12	0.000	**0.12**	0.17	0.17	0.001	**0.17**
	$\mathcal{V}^R$	0.08	0.08	0.000	**0.08**	0.13	0.14	0.001	**0.14**
	$\mathcal{K}^R$	0.08	0.08	0.000	**0.08**	0.13	0.14	0.001	**0.14**
	local	0.12	0.15	0.011	**0.12**	0.17	0.17	0.001	**0.17**
	nonlocal	0.43	0.43	0.001	**0.43**	0.66	0.66	0.001	**0.66**
2ptc-74k	$\mathcal{V}$	2.35	2.42	0.024	**2.38**	3.29	3.32	0.010	**3.30**
	$\mathcal{K}$	2.26	2.33	0.021	**2.29**	3.00	3.08	0.021	**3.05**
	$\mathcal{V}^R$	2.10	2.14	0.012	**2.11**	2.62	2.65	0.009	**2.63**
	$\mathcal{K}^R$	2.14	2.21	0.028	**2.16**	2.63	2.66	0.008	**2.65**
	local	2.27	2.33	0.018	**2.29**	3.01	3.14	0.034	**3.06**
	nonlocal	8.96	9.14	0.061	**9.01**	11.62	11.74	0.042	**11.69**
1c4k-15k	$\mathcal{V}$	0.12	0.12	0.002	**0.12**	0.17	0.17	0.001	**0.17**
	$\mathcal{K}$	0.09	0.10	0.000	**0.09**	0.13	0.13	0.001	**0.13**
	$\mathcal{V}^R$	0.06	0.06	0.000	**0.06**	0.07	0.07	0.000	**0.07**
	$\mathcal{K}^R$	0.06	0.06	0.000	**0.06**	0.07	0.07	0.000	**0.07**
	local	0.09	0.10	0.000	**0.10**	0.13	0.13	0.001	**0.13**
	nonlocal	0.33	0.33	0.000	**0.33**	0.45	0.45	0.001	**0.45**
1c4k-60k	$\mathcal{V}$	1.41	1.41	0.002	**1.41**	2.53	2.55	0.006	**2.54**
	$\mathcal{K}$	1.36	1.38	0.004	**1.37**	2.27	2.31	0.013	**2.29**
	$\mathcal{V}^R$	1.32	1.32	0.002	**1.32**	1.71	1.75	0.013	**1.73**
	$\mathcal{K}^R$	1.33	1.34	0.003	**1.34**	1.71	1.75	0.011	**1.73**
	local	1.37	1.38	0.004	**1.37**	2.18	2.32	0.041	**2.28**
	nonlocal	5.57	5.59	0.007	**5.58**	7.99	8.13	0.047	**8.04**
6ds5-170k	$\mathcal{V}$	11.38	11.46	0.029	**11.43**	15.90	16.05	0.047	**16.00**
	$\mathcal{K}$	11.24	11.46	0.066	**11.31**	15.55	15.95	0.102	**15.76**
	$\mathcal{V}^R$	10.69	10.77	0.024	**10.73**	14.53	14.65	0.042	**14.59**
	$\mathcal{K}^R$	10.96	11.02	0.020	**10.99**	14.58	14.79	0.087	**14.67**
	local	11.29	11.38	0.028	**11.33**	15.59	15.81	0.069	**15.71**
	nonlocal	45.21	45.45	0.069	**45.31**	60.29	60.89	0.180	**60.53**
2h1l-200k	$\mathcal{V}$	15.75	15.88	0.040	**15.80**	21.85	21.98	0.044	**21.91**
	$\mathcal{K}$	15.63	15.75	0.035	**15.68**	21.42	21.66	0.078	**21.54**
	$\mathcal{V}^R$	15.26	15.37	0.037	**15.31**	19.89	19.99	0.029	**19.91**
	$\mathcal{K}^R$	15.52	15.59	0.023	**15.55**	19.97	20.06	0.030	**20.02**
	local	15.67	15.75	0.025	**15.69**	21.38	21.60	0.080	**21.49**
	nonlocal	63.72	63.95	0.069	**63.84**	83.34	83.85	0.162	**83.62**

Table B.7. – Runtime evaluation for the GPU-based matrix-vector product implementations in CuNESSie on the second workstation (TITAN V)

		Single-precision runtime [s]				Double-precision runtime [s]			
Dataset	Matrix	min	max	std	avg	min	max	std	avg
2lzx-5k	$\mathcal{V}$	0.02	0.02	0.002	**0.02**	0.21	0.21	0.000	**0.21**
	$\mathcal{K}$	0.01	0.01	0.000	**0.01**	0.17	0.17	0.000	**0.17**
	$\mathcal{V}^R$	0.01	0.01	0.000	**0.01**	0.07	0.07	0.000	**0.07**
	$\mathcal{K}^R$	0.01	0.01	0.000	**0.01**	0.08	0.08	0.000	**0.08**
	local	0.01	0.01	0.000	**0.01**	0.17	0.17	0.000	**0.17**
	nonlocal	0.06	0.06	0.000	**0.06**	0.54	0.54	0.000	**0.54**
2lzx-20k	$\mathcal{V}$	0.10	0.12	0.004	**0.11**	3.27	3.29	0.012	**3.28**
	$\mathcal{K}$	0.08	0.08	0.000	**0.08**	2.65	2.65	0.000	**2.65**
	$\mathcal{V}^R$	0.19	0.20	0.000	**0.19**	1.17	1.17	0.000	**1.17**
	$\mathcal{K}^R$	0.19	0.19	0.000	**0.19**	1.32	1.33	0.005	**1.32**
	local	0.08	0.08	0.000	**0.08**	2.69	2.69	0.001	**2.69**
	nonlocal	0.57	0.58	0.002	**0.57**	8.54	8.55	0.001	**8.54**
2lzx-80k	$\mathcal{V}$	1.45	1.46	0.003	**1.45**	49.95	50.14	0.070	**50.03**
	$\mathcal{K}$	1.02	1.03	0.004	**1.02**	39.43	39.47	0.015	**39.44**
	$\mathcal{V}^R$	3.13	3.16	0.012	**3.15**	18.99	19.14	0.074	**19.08**
	$\mathcal{K}^R$	3.16	3.19	0.008	**3.18**	21.53	21.54	0.001	**21.54**
	local	1.06	1.07	0.003	**1.06**	39.48	39.85	0.159	**39.62**
	nonlocal	8.93	8.94	0.004	**8.93**	130.37	130.65	0.092	**130.48**
2ptc-9k	$\mathcal{V}$	0.04	0.05	0.002	**0.04**	0.79	0.79	0.000	**0.79**
	$\mathcal{K}$	0.03	0.03	0.000	**0.03**	0.64	0.64	0.000	**0.64**
	$\mathcal{V}^R$	0.05	0.05	0.000	**0.05**	0.27	0.27	0.000	**0.27**
	$\mathcal{K}^R$	0.05	0.05	0.000	**0.05**	0.31	0.31	0.000	**0.31**
	local	0.03	0.03	0.000	**0.03**	0.64	0.64	0.000	**0.64**
	nonlocal	0.17	0.17	0.000	**0.17**	2.00	2.02	0.004	**2.02**
2ptc-19k	$\mathcal{V}$	0.10	0.11	0.004	**0.10**	2.98	2.98	0.000	**2.98**
	$\mathcal{K}$	0.07	0.07	0.000	**0.07**	2.39	2.41	0.009	**2.40**
	$\mathcal{V}^R$	0.19	0.19	0.000	**0.19**	1.08	1.09	0.004	**1.09**
	$\mathcal{K}^R$	0.19	0.19	0.000	**0.19**	1.22	1.22	0.000	**1.22**
	local	0.08	0.08	0.000	**0.08**	2.42	2.42	0.000	**2.42**
	nonlocal	0.55	0.55	0.000	**0.55**	7.82	7.82	0.001	**7.82**
2ptc-74k	$\mathcal{V}$	1.31	1.31	0.001	**1.31**	44.01	44.26	0.088	**44.06**
	$\mathcal{K}$	0.92	0.92	0.001	**0.92**	35.08	35.17	0.031	**35.10**
	$\mathcal{V}^R$	2.78	2.78	0.001	**2.78**	16.56	16.70	0.065	**16.62**
	$\mathcal{K}^R$	2.78	2.81	0.011	**2.79**	18.78	18.79	0.001	**18.79**
	local	0.94	0.94	0.001	**0.94**	35.16	35.55	0.156	**35.29**
	nonlocal	7.86	7.87	0.002	**7.86**	114.97	115.08	0.035	**115.03**
1c4k-15k	$\mathcal{V}$	0.07	0.08	0.006	**0.07**	1.87	1.88	0.007	**1.88**
	$\mathcal{K}$	0.05	0.05	0.000	**0.05**	1.52	1.52	0.000	**1.52**
	$\mathcal{V}^R$	0.12	0.12	0.000	**0.12**	0.67	0.67	0.000	**0.67**
	$\mathcal{K}^R$	0.12	0.12	0.000	**0.12**	0.75	0.75	0.000	**0.75**
	local	0.05	0.05	0.000	**0.05**	1.53	1.53	0.000	**1.53**
	nonlocal	0.36	0.36	0.000	**0.36**	4.88	4.88	0.000	**4.88**
1c4k-60k	$\mathcal{V}$	0.87	0.87	0.000	**0.87**	28.90	28.90	0.000	**28.90**
	$\mathcal{K}$	0.61	0.61	0.000	**0.61**	22.87	22.87	0.000	**22.87**
	$\mathcal{V}^R$	1.83	1.84	0.002	**1.84**	10.77	10.83	0.017	**10.78**
	$\mathcal{K}^R$	1.84	1.84	0.003	**1.84**	12.21	12.21	0.001	**12.21**
	local	0.62	0.62	0.001	**0.62**	22.87	23.14	0.087	**23.02**
	nonlocal	5.17	5.18	0.003	**5.17**	74.99	75.02	0.011	**75.02**
6ds5-170k	$\mathcal{V}$	6.62	6.64	0.006	**6.62**	231.48	232.59	0.345	**232.20**
	$\mathcal{K}$	4.58	4.59	0.003	**4.59**	184.10	184.74	0.198	**184.24**
	$\mathcal{V}^R$	14.54	14.54	0.002	**14.54**	86.63	86.71	0.023	**86.70**
	$\mathcal{K}^R$	14.54	14.54	0.001	**14.54**	97.57	97.57	0.001	**97.57**
	local	4.65	4.66	0.004	**4.66**	184.61	185.36	0.293	**184.90**
	nonlocal	40.31	40.33	0.005	**40.32**	601.23	602.57	0.403	**602.04**
2h1l-200k	$\mathcal{V}$	9.12	9.13	0.004	**9.12**	316.08	316.77	0.236	**316.49**
	$\mathcal{K}$	6.30	6.31	0.002	**6.31**	249.33	250.40	0.312	**249.82**
	$\mathcal{V}^R$	20.00	20.01	0.002	**20.00**	120.17	120.18	0.001	**120.17**
	$\mathcal{K}^R$	20.00	20.02	0.008	**20.01**	135.23	135.23	0.001	**135.23**
	local	6.40	6.41	0.003	**6.41**	251.24	252.08	0.287	**251.69**
	nonlocal	55.45	55.48	0.008	**55.46**	824.95	826.70	0.456	**825.79**

Table B.8. – Runtime evaluation for the GPU-based matrix-vector product implementations in AnyBEM on the first workstation (RTX 2070 SUPER)

		Single-precision runtime [s]				Double-precision runtime [s]			
Dataset	Matrix	min	max	std	avg	min	max	std	avg
2lzx-5k	$\mathcal{V}$	0.03	0.03	0.002	**0.03**	0.04	0.05	0.002	**0.04**
	$\mathcal{K}$	0.02	0.02	0.000	**0.02**	0.03	0.04	0.000	**0.04**
	$\mathcal{V}^R$	0.02	0.02	0.000	**0.02**	0.02	0.03	0.000	**0.03**
	$\mathcal{K}^R$	0.02	0.02	0.000	**0.02**	0.03	0.03	0.000	**0.03**
	local	0.02	0.02	0.000	**0.02**	0.04	0.04	0.000	**0.04**
	nonlocal	0.08	0.08	0.000	**0.08**	0.13	0.13	0.000	**0.13**
2lzx-20k	$\mathcal{V}$	0.12	0.12	0.000	**0.12**	0.22	0.23	0.000	**0.22**
	$\mathcal{K}$	0.09	0.09	0.000	**0.09**	0.18	0.18	0.000	**0.18**
	$\mathcal{V}^R$	0.19	0.19	0.000	**0.19**	0.50	0.50	0.000	**0.50**
	$\mathcal{K}^R$	0.19	0.19	0.000	**0.19**	0.49	0.50	0.000	**0.50**
	local	0.09	0.09	0.000	**0.09**	0.18	0.18	0.000	**0.18**
	nonlocal	0.59	0.59	0.000	**0.59**	1.39	1.39	0.001	**1.39**
2lzx-80k	$\mathcal{V}$	0.96	0.96	0.000	**0.96**	2.86	2.88	0.008	**2.87**
	$\mathcal{K}$	0.71	0.71	0.000	**0.71**	2.16	2.18	0.006	**2.16**
	$\mathcal{V}^R$	4.33	4.41	0.023	**4.37**	15.07	15.89	0.303	**15.51**
	$\mathcal{K}^R$	4.24	4.30	0.020	**4.27**	13.89	13.94	0.017	**13.92**
	local	0.71	0.71	0.000	**0.71**	2.15	2.16	0.003	**2.16**
	nonlocal	10.19	10.26	0.020	**10.22**	34.41	35.02	0.237	**34.84**
2ptc-9k	$\mathcal{V}$	0.05	0.05	0.000	**0.05**	0.09	0.09	0.000	**0.09**
	$\mathcal{K}$	0.04	0.04	0.000	**0.04**	0.07	0.07	0.000	**0.07**
	$\mathcal{V}^R$	0.04	0.04	0.000	**0.04**	0.09	0.09	0.000	**0.09**
	$\mathcal{K}^R$	0.04	0.04	0.000	**0.04**	0.09	0.09	0.000	**0.09**
	local	0.04	0.04	0.000	**0.04**	0.07	0.07	0.000	**0.07**
	nonlocal	0.18	0.19	0.000	**0.19**	0.33	0.33	0.000	**0.33**
2ptc-19k	$\mathcal{V}$	0.11	0.11	0.000	**0.11**	0.21	0.21	0.000	**0.21**
	$\mathcal{K}$	0.09	0.09	0.000	**0.09**	0.16	0.16	0.000	**0.16**
	$\mathcal{V}^R$	0.17	0.17	0.000	**0.17**	0.39	0.39	0.000	**0.39**
	$\mathcal{K}^R$	0.17	0.17	0.000	**0.17**	0.39	0.39	0.000	**0.39**
	local	0.09	0.09	0.000	**0.09**	0.16	0.16	0.000	**0.16**
	nonlocal	0.54	0.54	0.000	**0.54**	1.15	1.15	0.000	**1.15**
2ptc-74k	$\mathcal{V}$	0.89	0.89	0.000	**0.89**	2.65	2.67	0.005	**2.66**
	$\mathcal{K}$	0.65	0.65	0.000	**0.65**	1.99	2.01	0.006	**2.01**
	$\mathcal{V}^R$	3.67	3.78	0.036	**3.73**	12.14	12.93	0.219	**12.61**
	$\mathcal{K}^R$	3.57	3.67	0.034	**3.62**	12.87	12.93	0.018	**12.91**
	local	0.66	0.66	0.000	**0.66**	1.99	2.01	0.006	**2.00**
	nonlocal	8.70	8.84	0.048	**8.79**	30.25	30.57	0.118	**30.37**
1c4k-15k	$\mathcal{V}$	0.09	0.09	0.000	**0.09**	0.17	0.17	0.000	**0.17**
	$\mathcal{K}$	0.07	0.07	0.000	**0.07**	0.13	0.13	0.000	**0.13**
	$\mathcal{V}^R$	0.11	0.11	0.001	**0.11**	0.22	0.22	0.000	**0.22**
	$\mathcal{K}^R$	0.11	0.11	0.001	**0.11**	0.22	0.22	0.000	**0.22**
	local	0.07	0.07	0.000	**0.07**	0.13	0.13	0.000	**0.13**
	nonlocal	0.39	0.39	0.000	**0.39**	0.74	0.74	0.000	**0.74**
1c4k-60k	$\mathcal{V}$	0.58	0.58	0.000	**0.58**	2.14	2.15	0.003	**2.14**
	$\mathcal{K}$	0.43	0.43	0.000	**0.43**	1.57	1.59	0.007	**1.58**
	$\mathcal{V}^R$	2.22	2.30	0.029	**2.26**	7.68	7.72	0.015	**7.69**
	$\mathcal{K}^R$	2.14	2.23	0.027	**2.18**	7.64	7.67	0.009	**7.66**
	local	0.44	0.44	0.000	**0.44**	1.56	1.59	0.007	**1.57**
	nonlocal	5.33	5.49	0.050	**5.42**	19.17	19.24	0.022	**19.21**
6ds5-170k	$\mathcal{V}$	5.34	5.68	0.108	**5.45**	14.82	15.05	0.069	**14.90**
	$\mathcal{K}$	3.94	4.06	0.050	**3.99**	11.20	11.39	0.059	**11.27**
	$\mathcal{V}^R$	20.10	20.73	0.185	**20.37**	62.12	62.81	0.254	**62.50**
	$\mathcal{K}^R$	19.61	20.02	0.132	**19.77**	59.79	60.33	0.185	**60.07**
	local	3.96	4.08	0.053	**4.02**	11.15	11.25	0.033	**11.22**
	nonlocal	47.60	48.34	0.230	**47.96**	149.05	149.86	0.337	**149.53**
2h1l-200k	$\mathcal{V}$	7.21	7.22	0.002	**7.22**	18.02	18.05	0.011	**18.03**
	$\mathcal{K}$	5.30	5.30	0.002	**5.30**	13.51	13.54	0.009	**13.52**
	$\mathcal{V}^R$	28.28	28.77	0.150	**28.46**	85.70	86.91	0.358	**86.46**
	$\mathcal{K}^R$	27.61	28.13	0.176	**27.83**	85.86	86.53	0.207	**86.22**
	local	5.33	5.35	0.006	**5.34**	13.47	13.52	0.017	**13.50**
	nonlocal	67.07	67.98	0.278	**67.62**	205.25	206.76	0.491	**205.75**

Table B.9. – Runtime evaluation for the GPU-based matrix-vector product implementations in AnyBEM on the second workstation (TITAN V)

List of Figures

List of Tables

Listings

Bibliography

[1] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. "Julia: A fresh approach to numerical computing". In: *SIAM Review* 59.1 (2017), pp. 65–98. DOI: 10.1137/141000671.

[2] Roland Leißa, Klaas Boesche, Sebastian Hack, et al. "AnyDSL: A Partial Evaluation Framework for Programming High-Performance Libraries". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), pp. 1–30. DOI: 10.1145/3276489.

[3] NVIDIA Corporation. *CUDA Zone*. URL: https://developer.nvidia.com/cuda-zone (visited on Sept. 23, 2020).

[4] Anna Katharina Hildebrandt, Thomas Kemmer, and Andreas Hildebrandt. "The electromagnetic nature of protein-protein interactions". In: *Conductive Polymers – Electrical Interactions in Cell Biology and Medicine*. Ed. by Ze Zhang, Mahmoud Rouabhia, and Simon E Moulton. Boca Raton, FL: CRC Press, 2017, pp. 153–174.

[5] Michael S Waterman. *Introduction to computational biology: Maps, sequences and genomes*. Boca Raton, FL: CRC Press, 1995.

[6] Catherine Sanchez, Corinne Lachaize, Florence Janody, et al. "Grasping at molecular interactions and genetic networks in Drosophila melanogaster using FlyNets, an Internet database". In: *Nucleic Acids Research* 27.1 (1999), pp. 89–94. DOI: 10.1093/nar/27.1.89.

[7] Jean-François Rual, Kavitha Venkatesan, Tong Hao, et al. "Towards a proteome-scale map of the human protein–protein interaction network". In: *Nature* 437.7062 (2005), pp. 1173–1178. DOI: 10.1038/nature04209.

[8] Emmanuel Barillot, Laurence Calzone, Philippe Hupé, Jean-Philippe Vert, and Andrei Zinovyev. *Computational Systems Biology of Cancer*. Boca Raton, FL: CRC Press, 2013.

[9] Thomas Lengauer. *Bioinformatics – From Genomes to Therapies*. Weinheim, Germany: WILEY-VCH, 2007.

[10] The UniProt Consortium. "UniProt: a worldwide hub of protein knowledge". In: *Nucleic Acids Research* 47.D1 (2018), pp. D506–D515. DOI: 10.1093/nar/gky1049.

[11] Christian B Anfinsen, Edgar Haber, Michael Sela, and FH White Jr. “The kinetics of formation of native ribonuclease during oxidation of the reduced polypeptide chain”. In: *Proceedings of the National Academy of Sciences of the United States of America* 47.9 (1961), pp. 1309–1314. DOI: 10.1073/pnas.47.9.1309.

[12] Christian B Anfinsen. “Principles that govern the folding of protein chains”. In: *Science* 181.4096 (1973), pp. 223–230. DOI: 10.1126/science.181.4096.223.

[13] Douglas Lee Brutlag. “Inferring Protein Function from Sequence”. In: *Bioinformatics – From Genomes to Therapies*. Ed. by Thomas Lengauer. Weinheim, Germany: WILEY-VCH, 2007, pp. 1087–1119.

[14] Vladimir N. Uversky. “Intrinsically disordered proteins from A to Z”. In: *The International Journal of Biochemistry & Cell Biology* 43.8 (2011), pp. 1090–1103. DOI: 10.1016/j.biocel.2011.04.001.

[15] Vladimir N. Uversky. “Intrinsically Disordered Proteins and Their “Mysterious” (Meta)Physics”. In: *Frontiers in Physics* 7 (2019), p. 10. DOI: 10.3389/fphy.2019.00010.

[16] Antonina Andreeva, Dave Howorth, Cyrus Chothia, Eugene Kulesha, and Alexey G. Murzin. “SCOP2 prototype: a new approach to protein structure mining”. In: *Nucleic Acids Research* 42.D1 (2013), pp. D310–D314. DOI: 10.1093/nar/gkt1242.

[17] Antonina Andreeva, Eugene Kulesha, Julian Gough, and Alexey G Murzin. “The SCOP database in 2020: expanded classification of representative family and superfamily domains of known protein structures”. In: *Nucleic Acids Research* 48.D1 (2019), pp. D376–D382. DOI: 10.1093/nar/gkz1064.

[18] Sara El-Gebali, Jaina Mistry, Alex Bateman, et al. “The Pfam protein families database in 2019”. In: *Nucleic Acids Research* 47.D1 (2019), pp. D427–D432. DOI: 10.1093/nar/gky995.

[19] Helen M. Berman, John Westbrook, Zukang Feng, et al. “The Protein Data Bank”. In: *Nucleic Acids Research* 28.1 (2000), pp. 235–242. DOI: 10.1093/nar/28.1.235.

[20] Francis Crick. “Central dogma of molecular biology”. In: *Nature* 227.5258 (1970), pp. 561–563. DOI: 10.1038/227561a0.

[21] Marshall Nirenberg. “The genetic code”. In: *Nobel Lectures, Physiology or Medicine 1963-1970*. Amsterdam: Elsevier, 1972.

[22] Steven L. Salzberg. “Open questions: How many genes do we have?” In: *BMC Biology* 16.94 (2018). DOI: 10.1186/s12915-018-0564-x.

[23] Mihaela Pertea, Alaina Shumate, Geo Pertea, et al. “CHESS: a new human gene catalog curated from thousands of large-scale RNA sequencing experiments reveals extensive transcriptional noise”. In: *Genome Biology* 19.208 (2018). DOI: 10.1186/s13059-018-1590-2.

[24] J. Craig Venter, Mark D. Adams, Eugene W. Myers, et al. “The Sequence of the Human Genome”. In: *Science* 291.5507 (2001), pp. 1304–1351. DOI: 10.1126/science.1058040.

[25] International Human Genome Sequencing Consortium. “Finishing the euchromatic sequence of the human genome”. In: *Nature* 431.7011 (2004), pp. 931–945. DOI: 10.1038/nature03001.

[26] Michele Clamp, Ben Fry, Mike Kamal, et al. “Distinguishing protein-coding and noncoding genes in the human genome”. In: *Proceedings of the National Academy of Sciences* 104.49 (2007), pp. 19428–19433. DOI: 10.1073/pnas.0709013104.

[27] Brent Ewing and Phil Green. “Analysis of expressed sequence tags indicates 35,000 human genes”. In: *Nature Genetics* 25.2 (2000), pp. 232–234. DOI: 10.1038/76115.

[28] Iakes Ezkurdia, David Juan, Jose Manuel Rodriguez, et al. “Multiple evidence strands suggest that there may be as few as 19 000 human protein-coding genes”. In: *Human Molecular Genetics* 23.22 (2014), pp. 5866–5878. DOI: 10.1093/hmg/ddu309.

[29] The UniProt Consortium. *What is UniProt’s human proteome?* URL: https://www.uniprot.org/help/human_proteome (visited on Aug. 17, 2020).

[30] Andreas Hildebrandt, Anna Katharina Dehof, Alexander Rurainski, et al. “BALL - Biochemical ALgorithms Library 1.3”. In: *BMC Bioinformatics* 11.531 (2010). DOI: 10.1186/1471-2105-11-531.

[31] Shoshana J. Wodak and Joël Janin. “Computer analysis of protein-protein interaction”. In: *Journal of molecular biology* 124.2 (1978), pp. 323–342. DOI: 10.1016/0022-2836(78)90302-9.

[32] Andreas Hildebrandt, Oliver Kohlbacher, and Hans-Peter Lenhof. “Modelling Protein-Protein and Protein-DNA docking”. In: *Bioinformatics – From Genomes to Therapies*. Ed. by Thomas Lengauer. Weinheim, Germany: WILEY-VCH, 2007, pp. 601–650.

[33] Emil Fischer. “Einfluss der Configuration auf die Wirkung der Enzyme (in German)”. In: *Berichte der deutschen chemischen Gesellschaft* 27.3 (1894), pp. 2985–2993. DOI: 10.1002/cber.18940270364.

[34] Daniel E. Koshland. “Application of a Theory of Enzyme Specificity to Protein Synthesis”. In: *Proceedings of the National Academy of Sciences of the United States of America* 44.2 (1958), pp. 98–104. DOI: 10.1073/pnas.44.2.98.

[35] E Katchalski-Katzir, I Shariv, M Eisenstein, et al. “Molecular surface recognition: determination of geometric fit between proteins and their ligands by correlation techniques”. In: *Proceedings of the National Academy of Sciences* 89.6 (1992), pp. 2195–2199. DOI: 10.1073/pnas.89.6.2195.

[36] William J. Allen, Trent E. Balius, Sudipto Mukherjee, et al. “DOCK 6: Impact of new features and current docking performance”. In: *Journal of Computational Chemistry* 36.15 (2015), pp. 1132–1156. DOI: 10.1002/jcc.23905.

[37] Oleg Trott and Arthur J. Olson. “AutoDock Vina: Improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading”. In: *Journal of Computational Chemistry* 31.2 (2010), pp. 455–461. DOI: 10.1002/jcc.21334.

[38] Richard A. Friesner, Jay L. Banks, Robert B. Murphy, et al. “Glide: A New Approach for Rapid, Accurate Docking and Scoring. 1. Method and Assessment of Docking Accuracy”. In: *Journal of Medicinal Chemistry* 47.7 (2004), pp. 1739–1749. DOI: 10.1021/jm0306430.

[39] Thomas A. Halgren, Robert B. Murphy, Richard A. Friesner, et al. “Glide: A New Approach for Rapid, Accurate Docking and Scoring. 2. Enrichment Factors in Database Screening”. In: *Journal of Medicinal Chemistry* 47.7 (2004), pp. 1750–1759. DOI: 10.1021/jm030644s.

[40] Marcel L. Verdonk, Jason C. Cole, Michael J. Hartshorn, Christopher W. Murray, and Richard D. Taylor. “Improved protein-ligand docking using GOLD”. In: *Proteins: Structure, Function, and Bioinformatics* 52.4 (2003), pp. 609–623. DOI: 10.1002/prot.10465.

[41] Kenneth M. Merz Jr. “The role of quantum mechanics in structure-based drug design”. In: *Drug design: Structure- and ligand-based approaches*. Ed. by Kenneth M. Merz Jr., Dagmar Ringe, and Charles H. Reynolds. New York, NY: Cambridge University Press, 2010.

[42] D. A. Case, D. S. Cerutti, III T. E. Cheatham, et al. *AMBER 2017*. Tech. rep. University of California, San Francisco, 2017.

[43] Nathan Schmid, Andreas P. Eichenberger, Alexandra Choutko, et al. “Definition and testing of the GROMOS force-field versions 54A7 and 54B7”. In: *European Biophysics Journal* 40.7 (2011), p. 843. DOI: 10.1007/s00249-011-0700-9.

[44] B. R. Brooks, C. L. Brooks, A. D. Mackerell, et al. “CHARMM: The biomolecular simulation program”. In: *Journal of Computational Chemistry* 30.10 (2009), pp. 1545–1614. DOI: 10.1002/jcc.21287.

[45] Raymond C. Tan, Thanh N. Truong, J. Andrew McCammon, and Joel L. Sussman. “Acetylcholinesterase: electrostatic steering increases the rate of ligand binding”. In: *Biochemistry* 32.2 (1993), pp. 401–403. DOI: 10.1021/bi00053a003.

[46] Daniel Ripoll, Carlos Faerman, P Axelsen, I Silman, and Joel Sussman. “An electrostatic mechanism for substrate guidance down the aromatic gorge of acetycholinesterase”. In: *Proceedings of the National Academy of Sciences of the United States of America* 90.11 (1993), pp. 5128–5132. DOI: 10.1073/pnas.90.11.5128.

[47] James Clerk Maxwell. “A dynamical theory of the electromagnetic field”. In: *Philosophical Transactions of the Royal Society of London* 155 (1865), pp. 459–512. DOI: 10.1098/rstl.1865.0008.

[48] J. D. Jackson. *Classical Electrodynamics*. New York: John Wiley, 1962.

[49] Andreas Hildebrandt. “Biomolecules in a structured solvent – A novel formulation of nonlocal electrostatics and its numerical solution”. PhD thesis. Saarbrücken, Germany: Saarland University, 2005.

[50] Revaz R. Dogonadze. *The chemical physics of solvation, Part A*. Elsevier Science Ltd., 1985.

[51] Aleksei A. Kornyshev. “On the non-local electrostatic theory of hydration force”. In: *Journal of Electroanalytical Chemistry and Interfacial Electrochemistry* 204.1 (1986), pp. 79–84. DOI: 10.1016/0022-0728(86)80509-5.

[52] Jun Wang, Qin Cai, Zhi-Lin Li, Hong-Kai Zhao, and Ray Luo. “Achieving energy conservation in Poisson–Boltzmann molecular dynamics: Accuracy and precision with finite-difference algorithms”. In: *Chemical Physics Letters* 468.4 (2009), pp. 112–118. DOI: 10.1016/j.cplett.2008.12.049.

[53] M. Holst, N. Baker, and F. Wang. “Adaptive multilevel finite element solution of the Poisson–Boltzmann equation I. Algorithms and examples”. In: *Journal of Computational Chemistry* 21.15 (2000), pp. 1319–1342. DOI: 10.1002/1096-987X(20001130)21:15<1319::AID-JCC1>3.0.CO;2-8.

[54] Barry Honig and Anthony Nicholls. “Classical electrostatics in biology and chemistry”. In: *Science* 268.5214 (1995), pp. 1144–1149. DOI: 10.1126/science.7761829.

[55] A. Hildebrandt, R. Blossey, S. Rjasanow, O. Kohlbacher, and H.-P. Lenhof. “Novel formulation of nonlocal electrostatics”. In: *Physical Review Letters* 93.10 (2004), p. 108104. DOI: 10.1103/PhysRevLett.93.108104.

[56] Dexuan Xie and Yi Jang. “A nonlocal modified Poisson-Boltzmann equation and finite element solver for computing electrostatics of biomolecules”. In: *Journal of Computational Physics* 322 (2016), pp. 1–20. DOI: 10.1016/j.jcp.2016.06.028.

[57] Dexuan Xie, Yi Jiang, and L. Ridgway Scott. “Efficient algorithms for a nonlocal dielectric model for protein in ionic solvent”. In: *SIAM Journal of Scientific Computing* 35.6 (2013), B1267–B1284. DOI: 10.1137/120899078.

[58] Jaydeep P. Bardhan. “Nonlocal continuum electrostatic theory predicts surprisingly small energetic penalties for charge burial in proteins”. In: *Journal of Chemical Physics* 135.104113 (2011). DOI: 10.1063/1.3632995.

[59] Jaydeep P. Bardhan and Andreas Hildebrandt. “A Fast Solver for Nonlocal Electrostatic Theory in Biomolecular Science and Engineering”. In: *Proceedings of the 48th Design Automation Conference*. 2011, pp. 801–805. DOI: 10.1145/2024724.2024904.

[60] S. Weggler, V. Rutka, and A. Hildebrandt. “A new numerical method for nonlocal electrostatics in biomolecular simulations”. In: *Journal of Computational Physics* 229.11 (2010), pp. 4059–4074. DOI: 10.1016/j.jcp.2010.01.040.

[61] Andreas Hildebrandt, Ralf Blossey, Sergej Rjasanow, Oliver Kohlbacher, and Hans-Peter Lenhof. “Electrostatic potentials of proteins in water: a structured continuum approach”. In: *Bioinformatics* 23.2 (2007), e99–e103. DOI: 10.1093/bioinformatics/btl312.

[62] Lothar Gaul, Martin Kögl, and Marcus Wagner. *Boundary element methods for engineers and scientists: an introductory course with advanced topics*. Springer Science & Business Media, 2013.

[63] Olaf Steinbach. *Numerical approximation methods for elliptic boundary value problems: finite and boundary elements*. New York, NY: Springer, 2008.

[64] Sergej Rjasanow. *Vorkonditionierte iterative Auflösung von Randelementgleichungen für die Dirichlet-Aufgabe (in German)*. Tech. rep. Wissenschaftliche Schriftreihe der Technischen Universität Karl-Marx-Stadt, 1990.

[65] Johann Radon. “Zur mechanischen Kubatur (in German)”. In: *Monatshefte für Mathematik* 52.4 (1948), pp. 286–300. DOI: 10.1007/BF01525334.

[66] JuliaLang.org contributors. *Julia Micro-Benchmarks*. URL: https://julialang.org/benchmarks/ (visited on Sept. 8, 2020).

[67] Kenta Sato, Koki Tsuyuzaki, Kentaro Shimizu, and Itoshi Nikaido. “Cell-Fishing.jl: an ultrafast and scalable cell search method for single-cell RNA sequencing”. In: *Genome Biology* 20.31 (2019). DOI: 10.1186/s13059-019-1639-x.

[68] Dewi Harjanto, Theodore Papamarkou, Chris J. Oates, et al. “RNA editing generates cellular subsets with diverse sequence within populations”. In: *Nature Communications* 7.12145 (2016). DOI: 10.1038/ncomms12145.

[69] Joe G. Greener, Ioannis Filippis, and Michael J.E. Sternberg. “Predicting Protein Dynamics and Allostery Using Multi-Protein Atomic Distance Constraints”. In: *Structure* 25.3 (2017), pp. 546–558. DOI: 10.1016/j.str.2017.01.008.

[70] Carlo Baldassi, Marco Zamparo, Christoph Feinauer, et al. “Fast and Accurate Multivariate Gaussian Modeling of Protein Families: Predicting Residue Contacts and Protein-Interaction Partners”. In: *PLOS ONE* 9 (2014), e92721. DOI: 10.1371/journal.pone.0092721.

[71] Alfredo Braunstein, Anna Paola Muntoni, and Andrea Pagnani. “An analytic approximation of the feasible space of metabolic networks”. In: *Nature Communications* 8.14915 (2017). DOI: 10.1038/ncomms14915.

[72] Jorge Fernandez-de-Cossio-Diaz, Kalet Leon, and Roberto Mulet. “Characterizing steady states of genome-scale metabolic networks in continuous cell cultures”. In: *PLOS Computational Biology* 13.11 (2017), pp. 1–25. DOI: 10.1371/journal.pcbi.1005835.

[73] Charles Le Losq and Daniel R. Neuville. “Molecular structure, configurational entropy and viscosity of silicate melts: Link through the Adam and Gibbs theory of viscous flow”. In: *Journal of Non-Crystalline Solids* 463 (2017), pp. 175–188. DOI: 10.1016/j.jnoncrysol.2017.02.010.

[74] Magnus Röding, Erich Schuster, Katarina Logg, et al. “Computational high-throughput screening of fluid permeability in heterogeneous fiber materials”. In: *Soft Matter* 12 (2016), pp. 6293–6299. DOI: 10.1039/C6SM01213B.

[75] Thomas Kemmer, Sergej Rjasanow, and Andreas Hildebrandt. “NESSie.jl – Efficient and Intuitive Finite Element and Boundary Element Methods for Nonlocal Protein Electrostatics in the Julia Language”. In: *Journal of Computational Science* 28 (2018), pp. 193–203. DOI: 10.1016/j.jocs.2018.08.008.

[76] Jeffrey W. Bezanson. “Abstraction in Technical Computing”. PhD thesis. Cambridge, MA: Massachusetts Institute of Technology, 2015.

[77] Jameson Nash. *Inference Convergence Algorithm in Julia*. 2016. URL: https://juliacomputing.com/blog/2016/04/04/inference-convergence.html (visited on Sept. 9, 2020).

[78] Jameson Nash. *Inference Convergence Algorithm in Julia – Revisited*. 2017. URL: https://juliacomputing.com/blog/2017/05/15/inference-converage2.html (visited on Sept. 9, 2020).

[79] Bezanson, Jeff and Chen, Jiahao and Chung, Benjamin and Karpinski, Stefan and Shah, Viral B and Vitek, Jan and Zoubritzky, Lionel. “Julia: Dynamism and performance reconciled by design”. In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), pp. 1–23. DOI: 10.1145/3276490.

[80] Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, et al. “Julia subtyping: a rational reconstruction”. In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), pp. 1–27. DOI: 10.1145/3276483.

[81] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, et al. “CommonLoops: Merging Lisp and Object-Oriented Programming”. In: *OOPSLA ’86: Conference proceedings on Object-oriented programming systems, languages and applications*. New York, NY, USA: Association for Computing Machinery, 1986, pp. 17–29. DOI: 10.1145/28697.28700.

[82] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04)*. Palo Alto, California, 2004. DOI: 10.1109/CGO.2004.1281665.

[83] Leonardo Dagum and Ramesh Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *Computational Science & Engineering, IEEE* 5.1 (1998), pp. 46–55. DOI: 10.1109/99.660313.

[84] Cray, CAPS, Nvidia, and PGI. *The OpenACC Application Programming Interface Version 2.5*. Oct. 2015. URL: https://www.openacc.org/specification.

[85] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.1*. 2015. URL: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf (visited on Sept. 18, 2020).

[86] Roland Leißa, Marcel Köster, and Sebastian Hack. “A graph-based higher-order intermediate representation”. In: *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2015, pp. 202–212. DOI: 10.1109/CGO.2015.7054200.

[87] Roland Leißa, Klaas Boesche, Sebastian Hack, Richard Membarth, and Philipp Slusallek. “Shallow Embedding of DSLs via Online Partial Evaluation”. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. 2015, pp. 11–20. DOI: 10.1145/2814204.2814208.

[88] Marcel Köster, Roland Leißa, Sebastian Hack, Richard Membarth, and Philipp Slusallek. “Code Refinement of Stencil Codes”. In: *Parallel Processing Letters* 24.03 (2014), p. 1441003. DOI: 10.1142/S0129626414410035.

[89] Arsène Pérard-Gayot, Martin Weier, Richard Membarth, et al. “RaTrace: Simple and Efficient Abstractions for BVH Ray Traversal Algorithms”. In: *GPCE 2017: Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. 2017, pp. 157–168. DOI: 10.1145/3170492.3136044.

[90] Arsène Pérard-Gayot, Richard Membarth, Roland Leißa, Sebastian Hack, and Philipp Slusallek. “Rodent: Generating Renderers without Writing a Generator”. In: *ACM Transactions on Graphics* 38.4 (2019). DOI: 10.1145/3306346.3322955.

[91] André Müller, Bertil Schmidt, Andreas Hildebrandt, et al. “AnySeq: A High Performance Sequence Alignment Library based on Partial Evaluation”. In: *Proceedings of the 34th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. New Orleans, LA, USA, 2020, pp. 1030–1040.

[92] Yoshihiko Futamura. “Partial computation of programs”. In: *RIMS Symposia on Software Science and Engineering*. 1983, pp. 1–35. DOI: 10.1007/3-540-11980-9_13.

[93] Yoshihiko Futamura. “Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler”. In: *Higher-Order and Symbolic Computation* 4.12 (1999), pp. 381–391. DOI: 10.1023/A:1010095604496.

[94] John E. Stone, David Gohara, and Guochun Shi. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. In: *Computing in Science Engineering* 12.3 (2010), pp. 66–73. DOI: 10.1109/MCSE.2010.69.

[95] Simon Moll and Sebastian Hack. “Partial Control-Flow Linearization”. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018*. 2018, pp. 543–556. DOI: 10.1145/3192366.3192413.

[96] Murray Irwin Cole. “Algorithmic skeletons : a structured approach to the management of parallel computation”. PhD thesis. Edinburgh, UK: University of Edinburgh, 1988.

[97] James Reinders. *Intel Threading Building Blocks*. O’Reilly, 2007.

[98] NVIDIA Corporation. *CUDA C++ Programming Guide, version 11.1.0*. 2020. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide (visited on Sept. 23, 2020).

[99] Bertil Schmidt, Jorge González-Domínguez, Christian Hundt, and Moritz Schlarb. *Parallel Programming – Concepts and Practics*. Cambridge, MA, USA: Morgan Kaufmann, 2018. DOI: 10.1016/C2015-0-02113-X.

[100] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors*. 3rd Ed. Cambridge, MA, USA: Morgan Kaufmann, 2017. DOI: 10.1016/C2015-0-02431-5.

[101] Michael J. Flynn. “Some Computer Organizations and Their Effectiveness”. In: *IEEE Transactions on Computers* C-21.9 (1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071.

[102] Julia developers. *JuliaGPU: High-performance GPU programming in a high-level language*. URL: https://juliagpu.org (visited on Sept. 27, 2020).

[103] Tim Besard. “Abstractions for Programming Graphics Processors in High-Level Programming Languages”. PhD thesis. Ghent, Belgium: Ghent University, 2019.

[104] T. Besard, C. Foket, and B. De Sutter. “Effective Extensible Programming: Unleashing Julia on GPUs”. In: *IEEE Transactions on Parallel and Distributed Systems* 30.4 (2019), pp. 827–841. DOI: 10.1109/TPDS.2018.2872064.

[105] Shuai Che, Michael Boyer, Jiayuan Meng, et al. “Rodinia: A benchmark suite for heterogeneous computing”. In: *2009 IEEE international symposium on workload characterization (IISWC)*. 2009, pp. 44–54. DOI: 10.1109/IISWC.2009.5306797.

[106] Yousef Saad. *Iterative methods for sparse linear systems*. 2nd. Vol. 82. Society for Industrial and Applied Mathematics, 2003.

[107] Valeria Simoncini and Daniel B Szyld. “Recent computational developments in Krylov subspace methods for linear systems”. In: *Numerical Linear Algebra with Applications* 14.1 (2007), pp. 1–59. DOI: 10.1002/nla.499.

[108] Jörg Liesen and Zdenek Strakos. *Krylov subspace methods: principles and analysis*. Oxford University Press, 2013.

[109] Magnus R Hestenes and Eduard Stiefel. “Methods of conjugate gradients for solving linear systems”. In: *Journal of research of the National Bureau of Standards* 49.6 (1952), pp. 409–436. DOI: 10.6028/jres.049.044.

[110] Cornelius Lanczos. “Solution of systems of linear equations by minimized iterations”. In: *Journal of Research of the National Bureau of Standards* 49.1 (1952), pp. 33–53. DOI: 10.6028/jres.049.006.

[111] Roger Fletcher. “Conjugate gradient methods for indefinite systems”. In: *Numerical analysis*. Springer, 1976, pp. 73–89. DOI: 10.1007/BFb0080116.

[112] Henk A Van der Vorst. “Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems”. In: *SIAM Journal on scientific and Statistical Computing* 13.2 (1992), pp. 631–644. DOI: 10.1137/0913035.

[113] Youcef Saad and Martin H Schultz. “GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems”. In: *SIAM Journal on scientific and statistical computing* 7.3 (1986), pp. 856–869. DOI: 10.1137/0907058.

[114] Stefan Kaczmarz. “Angenäherte Auflösung von Systemen linearer Gleichungen (in German)”. In: *Bull. Int. Acad. Pol. Sci. Lett. Class. Sci. Math. Nat* 35 (1937), pp. 355–357.

[115] Gianfranco Cimmino. “Cacolo approssimato per le soluzioni dei systemi di equazioni lineari (in Italian)”. In: *La Ricerca Scientifica (Roma)* 9 (1938), pp. 326–333.

[116] Gerard LG Sleijpen and Diederik R Fokkema. “BiCGstab (l) for linear equations involving unsymmetric matrices with complex spectrum”. In: *Electronic Transactions on Numerical Analysis* 1.11 (1993), p. 2000.

[117] Åke Björck and Tommy Elfving. “Accelerated projection methods for computing pseudoinverse solutions of systems of linear equations”. In: *BIT Numerical Mathematics* 19.2 (1979), pp. 145–163. DOI: 10.1007/BF01930845.

[118] Dan Gordon and Rachel Gordon. "CGMN revisited: robust and efficient solution of stiff linear systems derived from elliptic partial differential equations". In: *ACM Transactions on Mathematical Software (TOMS)* 35.3 (2008), pp. 1–27. DOI: 10.1145/1391989.1391991.

[119] Simon Bartels, Jon Cockayne, Ilse CF Ipsen, and Philipp Hennig. "Probabilistic linear solvers: a unifying view". In: *Statistics and Computing* 29.6 (2019), pp. 1249–1263. DOI: 10.1007/s11222-019-09897-7.

[120] Robert M Gower. "Sketch and Project: Randomized Iterative Methods for Linear Systems and Inverting Matrices". PhD thesis. Edinburgh, UK: The University of Edinburgh, 2016.

[121] Thomas Strohmer and Roman Vershynin. "A randomized Kaczmarz algorithm with exponential convergence". In: *Journal of Fourier Analysis and Applications* 15.2 (2009), p. 262. DOI: 10.1007/s00041-008-9030-4.

[122] Anastasios Zouzias and Nikolaos M Freris. "Randomized extended Kaczmarz for solving least squares". In: *SIAM Journal on Matrix Analysis and Applications* 34.2 (2013), pp. 773–793. DOI: 10.1137/120889897.

[123] Deanna Needell, Ran Zhao, and Anastasios Zouzias. "Randomized block Kaczmarz method with projection for solving least squares". In: *Linear Algebra and its Applications* 484 (2015), pp. 322–343. DOI: 10.1016/j.laa.2015.06.027.

[124] Dennis Leventhal and Adrian S Lewis. "Randomized methods for linear constraints: convergence rates and conditioning". In: *Mathematics of Operations Research* 35.3 (2010), pp. 641–654. DOI: 10.1287/moor.1100.0456.

[125] Yin Tat Lee and Aaron Sidford. "Efficient accelerated coordinate descent methods and faster algorithms for solving linear systems". In: *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*. 2013, pp. 147–156. DOI: 10.1109/FOCS.2013.24.

[126] Zheng Qu, Peter Richtárik, Martin Takác, and Olivier Fercoq. "SDNA: Stochastic dual Newton ascent for empirical risk minimization". In: *Proceedings of the 33rd International Conference on Machine Learning*. 2016, pp. 1823–1832.

[127] Michele Benzi. "Preconditioning techniques for large linear systems: a survey". In: *Journal of computational Physics* 182.2 (2002), pp. 418–477. DOI: 10.1006/jcph.2002.7176.

[128] Olaf Steinbach and Wolfgang L Wendland. “The construction of some efficient preconditioners in the boundary element method”. In: *Advances in Computational Mathematics* 9.1-2 (1998), pp. 191–216. DOI: 10.1023/A:1018937506719.

[129] Joseph M Elble, Nikolaos V Sahinidis, and Panagiotis Vouzis. “GPU computing with Kaczmarz’s and other iterative algorithms for linear systems”. In: *Parallel computing* 36.5-6 (2010), pp. 215–231. DOI: 10.1016/j.parco.2009.12.003.

[130] L. S. Blackford, J. Demmel, J. Dongarra, et al. “An Updated Set of Basic Linear Algebra Subprograms (BLAS)”. In: *ACM Transactions on Mathematical Software* 28.2 (2002), pp. 135–151. DOI: 10.1145/567806.567807.

[131] E. Anderson, Z. Bai, C. Bischof, et al. *LAPACK Users’ Guide*. Third. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.

[132] Timothy A. Davis. *Direct Methods for Sparse Linear Systems*. Philadelphia, PA: SIAM, 2006. DOI: 10.1137/1.9780898718881.

[133] Pieter Wesseling and Peter Sonneveld. “Numerical experiments with a multiple grid and a preconditioned Lanczos type method”. In: *Approximation methods for Navier-Stokes problems*. Springer, 1980, pp. 543–562.

[134] Peter Sonneveld and Martin B Van Gijzen. “IDR (s): A family of simple and fast algorithms for solving large nonsymmetric systems of linear equations”. In: *SIAM Journal on Scientific Computing* 31.2 (2009), pp. 1035–1062. DOI: 10.1137/070685804.

[135] Martin B Van Gijzen and Peter Sonneveld. “Algorithm 913: An elegant IDR (s) variant that efficiently exploits biorthogonality properties”. In: *ACM Transactions on Mathematical Software (TOMS)* 38.1 (2011), pp. 1–19. DOI: 0.1145/2049662.2049667.

[136] R Clint Whaley, Antoine Petitet, and Jack J Dongarra. “Automated Empirical Optimization of Software and the ATLAS Project”. In: *Parallel Computing* 27.1–2 (2001), pp. 3–35. DOI: 10.1016/S0167-8191(00)00087-9.

[137] Tero Frondelius and Jukka Aho. “JuliaFEM - open source solver for both industrial and academia usage”. In: *Rakenteiden Mekaniikka* 50.3 (2017), pp. 229–233. DOI: 10.23998/rm.64224.

[138] Todd J. Dolinsky, Paul Czodrowski, Hui Li, et al. “PDB2PQR: expanding and upgrading automated preparation of biomolecular structures for molecular simulations”. In: *Nucleic Acids Research* 35.suppl2 (2007), W522. DOI: 10.1093/nar/gkm276.

[139] Zeyun Yu, Michael J Holst, Yuhui Cheng, McCammon, and J. Andrew. "Feature-preserving adaptive mesh generation for molecular shape modeling and simulation". In: *Journal of Molecular Graphics and Modelling* 26.8 (2008), pp. 1370–1380. DOI: 10.1016/j.jmgm.2008.01.007.

[140] Stephan Russenschuck. "ROXIE: the Routine for the Optimization of Magnet X-sections, Inverse Field Computation and Coil End Design". In: CERN-AT-93-27-MA. LHC-NOTE-238. CERN-LHC-Note-238 (1993). DOI: 10.5170/CERN-1999-001.1.

[141] Stephan Russenschuck. "ROXIE: A Computer Code for the Integrated Design of Accelerator Magnets". In: LHC-Project-Report-276. CERN-LHC-Project-Report-276 (1999).

[142] Altair Engineering. *Altair HyperWorks 7.0 - The Engineering Framework for Product Design*. 2004.

[143] Pavel Solin, Lukas Korous, and Pavel Kus. "Hermes2D, a C++ library for rapid development of adaptive hp-FEM and hp-DG solvers". In: *Journal of Computational and Applied Mathematics* 270 (2014), pp. 152–165. DOI: 10.1016/j.cam.2014.02.007.

[144] Blender Online Community. *Blender – a 3D modelling and rendering package*. Blender Foundation. Stichting Blender Foundation, Amsterdam, 2018. URL: http://www.blender.org.

[145] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit*. Fourth. Kitware, 2006.

[146] James Ahrens, Berk Geveci, and Charles Law. *ParaView: An End-User Tool for Large Data Visualization, Visualization Handbook*. Elsevier, 2005.

[147] Paolo Cignoni, Marco Callieri, Massimiliano Corsini, et al. "MeshLab: an Open-Source Mesh Processing Tool". In: *Sixth Eurographics Italian Chapter Conference*. 2008, pp. 129–136. DOI: 10.2312/LocalChapterEvents/ItalChap/ItalianChapConf2008/129-136.

[148] Nina Amenta, Stuart Levy, Tamara Muzner, and Mark Phillips. "Geomview: a system for geometric visualization". In: *SCG '95 Proceedings of the eleventh annual symposium on Computational geometry*. 1995, pp. 412–413. DOI: 10.1145/220279.220327.

[149] Kristian Sons, Felix Klein, Dmitri Rubinstein, Sergiy Byelozyorov, and Philipp Slusallek. "XML3D – Interactive 3D Graphics for the Web". In: *Proceedings of the 15th International Conference on Web 3D Technology (Web3D)*. 2010, pp. 175–184. DOI: 10.1145/1836049.1836076.

[150] Johan Åqvist. “Ion-water interaction potentials derived from free energy perturbation simulations”. In: *The Journal of Physical Chemistry* 94.21 (1990), pp. 8021–8024. DOI: 10.1021/j100384a009.

[151] Dexuan Xie, Hans W. Volkmer, and Jinyong Ying. “Analytical solutions of nonlocal Poisson dielectric models with multiple point charges inside a dielectric sphere”. In: *Physical Review E* 93 (2016), p. 043304. DOI: 10.1103/PhysRevE.93.043304.

[152] Tim Besard, Valentin Churavy, Alan Edelman, and Bjorn De Sutter. “Rapid software prototyping for heterogeneous and distributed platforms”. In: *Advances in Engineering Software* 132 (2019), pp. 29–46. DOI: 10.1016/j.advengsoft.2019.02.002.

[153] Sophie Weggler. “Correlation induced electrostatic effects in biomolecular systems”. PhD thesis. Saarbrücken, Germany: Saarland University, 2010.

[154] RCSB PDB. *PDB Statistics: PDB Data Distribution by Atom Count*. URL: https://www.rcsb.org/stats/distribution-atom-count (visited on July 14, 2020).

[155] Heena Khatter, Alexander G. Myasnikov, S. Kundhavai Natchiar, and Bruno P. Klaholz. “Structure of the human 80S ribosome”. In: *Nature* 520 (2015), pp. 640–645. DOI: 10.1038/nature14427.

[156] R. Clint Whaley. *ATLAS Installation Guide (ATLAS 3.10.2)*. 2014. URL: http://math-atlas.sourceforge.net/.

[157] Frank WJ Olver, Daniel W Lozier, Ronald F Boisvert, and Charles W Clark. *NIST handbook of mathematical functions*. Cambridge, UK: Cambridge university press, 2010.

[158] Michael F. Sanner, Arthur J. Olson, and Jean-Claude Spehner. “Fast and robust computation of molecular surfaces”. In: *SCG ’95 Proceedings of the eleventh annual symposium on Computational geometry*. 1995, pp. 406–407. DOI: 10.1145/220279.220324.