Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Fachstudie Nr. 249

Vergleich von Bibliotheken zur Linearen Algebra

Joscha Götzer, Simon Reiß, Dominik Schreiber

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Miriam Mehl
Betreuer/in:	Dipl.-Inf. Florian Lindner
Beginn am:	12. Juli 2016
Beendet am:	7. November 2016
CR-Nummer:	G.0

Kurzfassung

Bibliotheken zur Linearen Algebra sind für eine reiche Vielfalt verschiedener Projekte, Anwendungen und Vorhaben notwendig. Doch die Wahl einer für die konkreten Bedürfnisse optimalen Bibliothek ist häufig nicht-trivial, da viele verschiedene Projekte bestehen, die mit unterschiedlicher Zielsetzung jeweils gute Ergebnisse erzielen.

In der vorliegenden Arbeit werden Kriterien, die gute und nützliche Bibliotheken zur Linearen Algebra auszeichnen, erarbeitet. Dazu zählen funktionale Anforderungen wie die implementierten Features, jedoch ebenso nicht-funktionale Anforderungen wie die Bedienbarkeit, Lizenz und Dokumentation.

Anschließend werden fünf beliebte Bibliotheken zur Linearen Algebra – Armadillo, Eigen, PETSc, Scipy und ViennaCL – zunächst isoliert und daraufhin im direkten Vergleich intensiv auf Vorteile und Schwächen untersucht. Recherechen über die Bibliotheken werden im Rahmen der Untersuchungen durch Codebeispiele, praktische Erfahrungen und einen Performancevergleich ergänzt.

Das Ergebnis der Arbeit stellt eine Reihe von Empfehlungen dar, welche Bibliothek zu welchem speziellen Anwendungsfall eine gute Wahl darstellt. Insgesamt kann PETSc insbesondere im Bereich aufwendiger verteilter Berechnungen überzeugen, während Eigen und Scipy hervorragend für Endanwendungen geeignet sind. Für wissenschaftliche Anwendungen kann situationsbedingt und je nach verfügbarer Hardware unter anderem eine Kombination der gut zueinander kompatiblen Bibliotheken Eigen, ViennaCL und Armadillo eingesetzt werden, um optimale Resultate zu erzielen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1 Einleitung

1.1 Motivation

Für verschiedenste Arten von Computeranwendungen ist die Berechnung von Problemen aus der Linearen Algebra zentraler Bestandteil. Ob es sich um die Abbildung einer räumlichen Verzerrung auf dem zweidimensionalen Bildschirm, um die Lösung eines Optimierungsproblems oder um intelligente Entscheidungsfindung autonomer Systeme handelt; die Entwickler sind auf die Berechnung von Problemen aus der Linearen Algebra angewiesen.

Grundsätzlich können mithilfe einer beliebigen gängigen Programmiersprache[1] ohne weitere Hilfsmittel die Routinen der Linearen Algebra implementiert werden, die für den konkreten Anwendungsfall benötigt werden. Aus Sicht des Software Engineerings bietet es sich jedoch aus nahezu allen Gesichtspunkten an, eine bestehende Bibliothek zu verwenden. Die Entwicklungskosten für die benötigten Routinen entfallen und häufig enthalten derartige Bibliotheken bereits aufwendige hardwarenahe Optimierungen und Schnittstellen zu Technologien wie MPI oder CUDA, um hocheffizient und parallel berechnen zu können. Die Entscheidung „Make or Buy“ fällt dabei normalerweise trivial aus, da eine große Zahl von LA-Bibliotheken Freie Software ist, wie im Verlauf der vorliegenden Arbeit ersichtlich wird. Zusammenfassend stellt es im Regelfall die beste Entscheidung dar, eine „gute“ Bibliothek zur Linearen Algebra zu verwenden, um ein entsprechendes Softwareprojekt zu realisieren.

Doch selbstverständlich muss das Attribut „gut“ hier weiter ausgeführt werden; welche Faktoren sind relevant bei der Entscheidung für oder gegen eine bestimmte Bibliothek? Und, aufbauend auf diese Faktoren, wie verhalten sich die heutigen bekanntesten Bibliotheken zur Linearen Algebra demgegenüber konkret? Diese Fragestellungen sollen in der vorliegenden Fachstudie erörtert werden.

1.2 Herangehensweise

Zunächst werden sinnvolle Anforderungen an eine Bibliothek zur Linearen Algebra gesammelt, die eine Entscheidung für oder gegen eine Bibliothek potentiell beeinflussen. Daraufhin werden einige bekannte Bibliotheken zur Linearen Algebra vorgestellt und analysiert. Diese

[1] Dabei wird Turing-Mächtigkeit der Sprache vorausgesetzt.

Bibliotheken werden schließlich auf die gesammelten Aspekte hin bewertet und miteinander verglichen – sowohl in Bezug auf ihre generellen Eigenschaften, als auch in einem konkreten Performancetest. Die zentralen Ergebnisse der Arbeit sind zum einen eine übersichtliche Gegenüberstellung der Bibliotheken in Hinsicht auf ihre jeweiligen Stärken und Schwächen sowie eine Reihe von Empfehlungen in Bezug auf verschiedene praktische Anwendungsfälle.

1.3 Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen: Hier werden grundsätzliche Anforderungen und Eigenschaften von Bibliotheken zur Linearen Algebra beschrieben.

Kapitel 3 – Planung enthält die Methodik für den Vergleich und die Vergleichskriterien der zu untersuchenden Bibliotheken.

Kapitel 4 – Behandelte Bibliotheken: Dieses Kapitel stellt jede Bibliothek im Einzelnen vor und analysiert ihre jeweiligen individuellen Eigenschaften.

Kapitel 5 – Implementierung: Es werden technische Details zur Programmierung der Test-Programme mit den einzelnen Bibliotheken sowie zur Durchführung des Performance-Benchmarks beschrieben.

Kapitel 6 – Ergebnisse enthält den Vergleich der Bibliotheken miteinander, sowie eine direkte Gegenüberstellung der jeweiligen Eigenschaften.

Kapitel 7 – Zusammenfassung fasst die Ergebnisse der Arbeit zusammen.

2 Grundlagen

2.1 Eigenschaften von LA-Bibliotheken

Die Lineare Algebra stellt ein großes Gebiet der Mathematik dar, sowohl in der Theorie als auch in ihrer Anwendung. Da die vielfältigsten Anwendungen auf Lineare Algebra angewiesen sind, dabei jedoch auch ganz verschiedene individuellen Anforderungen mit sich bringen, lohnt es sich an dieser Stelle darauf einzugehen welche verschiedenen Eigenschaften eine LA-Bibliothek (und damit deren Eignung) definieren.

Die Eigenschaften werden in funktionale und nicht-funktionale Eigenschaften getrennt, gemäß der üblichen Taxonomie der Anforderungen im Bereich der Softwaretechnik. Dabei sind funktionale Eigenschaften jene, die sich gemäß ihrer Natur eindeutig mit geeigneten Metriken messen und vergleichen lassen. Nicht-funktional sind „weiche“ Eigenschaften wie Bedienbarkeit, Lizenz und die Community, für welche keine solche eindeutige Messung existiert. [LL13, Kapitel 16.4.2]

2.1.1 Funktional

Zunächst werden die wichtigsten funktionalen Eigenschaften von Bibliotheken zur Linearen Algebra beschrieben.

Generelle Herangehensweise

Eine wichtige Eigenschaft von LA-Bibliotheken stellt dar, welches *Backend* verwendet wird beziehungsweise welche *Standards* aufgenommen und implementiert sind. Dabei geht es um niedrig abstrahierte Eigenschaften wie die Ausführung der grundlegenden Vektor- und Matrixoperationen, die Ausnutzung von Parallelität sowie die optimale Verwendung vorhandener Hardware.

BLAS

Die *Basic Linear Algebra Subprograms* stellen eine Schnittstelle zwischen der Hardware und der LA-Bibliothek dar; sie enthalten grundlegende Vektor- und Matrix-Rechenoperationen

(Addition, Multiplikation sowie Skalarprodukte), die von Bibliotheken genutzt werden können.[1] Verschiedene Bibliotheken machen von unterschiedlichen BLAS-Implementierungen Gebrauch, doch die eigentlichen BLAS-Schnittstellen sind ein *de facto*-Standard.

LAPACK

Die *Linear Algebra Package* ist eine Freie Softwarebibliothek, die auf BLAS aufbaut und weitere Funktionalität aus dem Bereich der Linearen Algebra bereitstellt, darunter die Lösung Linearer Gleichungssysteme und Eigenwertprobleme.[2] Viele Bibliotheken nutzen LAPACK, um wiederum darauf aufbauen zu können.

Parallelität

Probleme, die für parallele Berechnungen gut geeignet sind (beispielsweise die Matrizenmultiplikation) sollten auf Multicore-Architekturen parallel gelöst werden, da der Zeitaufwand somit um ein Vielfaches reduziert werden kann[3]. Parallele Berechnungen sind ein anspruchsvolles und komplexes Thema, weshalb Standards wie das MPI (*Message Passing Interface*) eingeführt wurden. Die durchgängige Nutzung eines solchen Standards bei einer Bibliothek ist als positiv zu bewerten, solange dabei eine gute Benutzbarkeit gewahrt wird.

Vektorisierung

Das Prinzip *Single Instruction, Multiple Data* (SIMD) ermöglicht es, pro Berechnungsschritt der CPU (oder GPU– siehe unten) mehrere gleichartige Rechenoperationen, etwa die Multiplikation von Fließkommazahlen, zugleich zu berechnen. Dies nennt man auch *Vektorisierung*. Die Technologie wird von verschiedenen CPUs unterstützt, beispielsweise durch *Neon* auf ARM-Architekturen[4].

GPU-Nutzung

Technologien wie *Compute Unified Device Architecture* (CUDA) oder *Open Computing Language* (OpenCL) nutzen das große Potential, Berechnungen auf der Grafikkarte auszubringen (sofern vorhanden). Die hohe Spezialisierung der Grafikprozessoren auf schnelle parallele Berechnungen kann einen enormen Zuwachs von Performance bedeuten[5], insbesondere bei hoch skalierten wissenschaftlichen Berechnungen.

[1] http://www.netlib.org/blas/

[2] http://www.netlib.org/lapack/index.html

[3] Diese Forderung nach der grundsätzlichen parallelen Berechnung geeigneter Probleme ist sinnvoll auf Hochleistungs- und stationären Desktoprechnern, kann aber bei der Ausführung von Berechnungen auf Mobilgeräten hinterfragt werden, da hier der Energieverbrauch möglicherweise sogar ansteigt und damit die Nutzbarkeit des Geräts einschränkt.

[4] http://www.arm.com/products/processors/technologies/neon.php

[5] http://gpgpu.org/about

Features

Als zentrale Anforderung an eine LA-Bibliothek ist eine komfortable Auswahl an verschiedenen Verfahren und Algorithmen zu nennen. Grundlegende Verfahren wie Matrizenmultiplikation oder die Lösung von Linearen Gleichungssystemen werden vorausgesetzt, doch gute Bibliotheken unterscheiden neben den einzelnen Kategorien der Lösungsverfahren auch verschiedenartige Konditionierungen der Problemstellung. So kann eine LA-Bibliothek für ein einzelnes Verfahren verschiedene Implementierungen aufweisen, die jeweils besser für dünn respektive dicht besetzte Matrizen geeignet sind, und damit bei korrekter Nutzung einen erheblichen, unter Umständen sogar dringend benötigten Vorteil in Laufzeit– und Speichereffizienz erbringen.

Als Beispiel seien hier die Bandmatrizen angegeben. Die herkömmliche Lösung eines LGS

$$Ax = b$$

mit Matrix A und Vektoren x und b besitzt eine Komplexität von $O(n^3)$. Falls es sich jedoch bei A um eine k-Bandmatrix handelt (für ein festes $k \in N$), so verringert sich mit dem geeigneten Verfahren die Komplexität der Berechnung auf $O(kn^2)$, was für konstante k der Komplexität $O(n^2)$ entspricht. [REU, S. 89f.] Eine herkömmliche, für dichte Matrizen vorgesehene Implementierung wird damit verglichen unnötig viel Speicherplatz allokieren und im Allgemeinen eine deutlich längere Laufzeit aufweisen.

LA-Bibliotheken sollten die Entwickler dazu ermutigen, die verschiedenen Verfahren optimal auszunutzen, indem sie für die verschiedenen Konditionierungen des Problems jeweils eine gut zugängliche Implementierung anbieten.

Performance

Die schiere Größe bestimmter Eingabedaten setzt voraus, dass eine LA-Bibliothek höchst effizient arbeitet. Besondere Kriterien sind je nach Anwendung eine hervorragende Nutzung der vorhandenen Hardware durch effizientes Caching, Vektorisierungs-Technologie, Nutzung der GPU oder durch Parallelisierung der angebotenen Berechnungen. Diese einzelnen Technologien werden im vorhergehenden Abschnitt zur Generellen Herangehensweise beleuchtet.

2.1.2 Nicht-funktional

Anschließend werden einige wichtige nicht-funktionale Eigenschaften der Bibliotheken im Allgemeinen beschrieben.

Programmiersprache

Die Wahl der Programmiersprache für ein Projekt hat weitreichende Konsequenzen, da verschiedene Sprachen unterschiedliche Eigenschaften, Anforderungen, Stärken und Risiken mit sich bringen. Im Folgenden werden einige der gängigsten Programmiersprachen im Bezug auf Bibliotheken zur Linearen Algebra betrachtet. Dabei wird nur von der Programmiersprache der Nutzerschnittstelle ausgegangen; Programmiersprachen wie Fortran, die hauptsächlich in der internen Implementierung der Bibliotheken auftreten, werden nicht berücksichtigt.

C [16a]

C ist eine imperative Programmiersprache, die aus heutiger Sicht eine niedrige Abstraktionsebene und eher geringen Funktionsumfang mit sich bringt. Programme in reinem C geschrieben – abgegrenzt zu dessen Erweiterung C++ – zeichnen sich dadurch aus, dass sie streng imperativ aufgebaut sind, also insbesondere keine Objektorientierung oder Funktionale Programmierung nutzen. Zudem fehlen viele heute weit verbreitete Konzepte wie Ausnahmebehandlungen, Information Hiding oder hierarchische Modularisierung.

Ein (bedingter) Vorteil der Sprache, auch noch in der heutigen Programmiersprachen-Landschaft, ist die Möglichkeit der sehr hardwarenahen Programmierung. Durch unmittelbare Speicherallokation und -manipulation sowie Zeigerarithmetik lassen sich Berechnungen auf einem niedrigen Abstraktionslevel mit hoher Effizienz implementieren, wie sie mit Sprachen wie Python oder Java aufgrund der inhärenten Laufzeitprüfungen nicht möglich ist. Im Gegenzug bringt dies auch große Probleme mit sich; nicht umsonst haben sich die meisten der heutigen Hochsprachen gegen eine direkte Speichermanipulation entschieden, denn es können nicht nur Speicherfehler mit schwerwiegenden Konsequenzen, sondern auch gefährliche Sicherheitslücken (wie im Falle des `Buffer Overflows`) entstehen.

C genießt eine weite Verbreitung vor allem im Bereich der Systemprogrammierung, während in der Anwendungsprogrammierung häufig auf die Erweiterung C++ oder aber gänzlich andere Hochsprachen gesetzt wird. High-Performance-Anwendungen werden hingegen noch oft in reinem C geschrieben.

Eine in C geschriebene Bibliothek zur Linearen Algebra hat im Speziellen das Potential zu besonders hoher Performanz. Ein Risiko sind dagegen die Unsicherheiten, die C-Code potentiell mit sich bringt, sowie eine aus heutiger Sicht schlechtere Benutzerfreundlichkeit gegenüber den Anwendungsentwicklern (geschuldet der älteren Syntax und fehlenden Sprachfeatures).

C++ [16b]

Die Erweiterung von C namens C++ erhielt im Verlauf ihrer Entwicklung weitgehend alle Features, die von einer modernen Hochsprache erwartet werden, darunter Objektorientierung, Template-Programmierung und Funktionale Ausdrücke. Gleichzeitig behielt C++ alle aus C bekannten Sprachkonstrukte, was C++ heute zu einer Sprache macht, die wohlwollend gesehen als „vielfältig“ oder kritisch gesehen als „unübersichtlich und überladen“ bezeichnet werden kann. Daher ist in C++ der Programmierstil außerordentlich ausschlagkräftig; bei konsequenter

Benutzung abstrahierender Bibliotheken und Frameworks wie etwa `Qt` kann moderner und gut wartbarer Code geschrieben werden, der beispielsweise Java-Code ähnelt; gleichzeitig kann auch hardwarenaher Code in C-Manier geschrieben werden, um möglichst hohe Effizienz zu ermöglichen. Eine Vermischung der vielen verschiedenen Sprachkonzepte von C++ kann jedoch auch zu beliebig komplexem Code führen, beispielsweise wenn großzügig eingesetzte Preprocessor-Ausdrücke und Zeigerarithmetik auf generische Objektorientierung treffen.

C++ ist heute eine der zentralen Programmiersprachen für System- sowie Anwendungsentwicklung. Seine modernen Features gekoppelt mit der Möglichkeit zu hocheffizienter Programmierung machen C++ auch im High-Performance-Bereich sehr beliebt, und (gemeinsam mit seiner weiten Verbreitung) zu einer vielversprechenden Programmiersprache für Bibliotheken zur Linearen Algebra. Besondere Vorsicht muss bei der Art seiner Benutzung gewaltet werden, um eine zuverlässige, sichere und wartbare Software zu gewährleisten.

Python [16e]

Bei Python handelt es sich um eine höhere Programmiersprache, die seit 1991 existiert. In der Praxis wird noch sehr häufig Python 2.7 verwendet, die aktuellste Version ist jedoch Python 3.5.2[6].

Code in Python wird durch Einrückungen formatiert, welche geschweifte Klammern aus anderen Programmiersprachen ersetzten. Dieser Umstand kann durchaus als Vorteil gesehen werden, wenn es darum geht, Programmieranfänger aus Fachbereichen, welche die Grenzen der informatiknahen Disziplinen überschreiten, in das wissenschaftliche Programm einzuweisen, ebenso wie möglichst nahe an Pseudocode entlang implementieren zu können. Hinzuzufügen ist, dass eine große Vielfalt an Python Tutorials in verschiedenen Medienformaten durch das Internet zugänglich sind, sowohl für Programmieranfänger als auch für erfahrene Programmierer.

Python ermöglicht ein schnelles Verfassen von prozeduralen Skripting-Programmen, unterstützt gleichzeitig aber auch Programmierparadigmen wie funktionales und objektorientiertes Programmieren für umfangreichere Projekte und Programmierung im Sinne einer sorgfältigen Softwareentwicklung. Eine weitere Eigenschaft von Python ist die dynamische Typisierung, welche wiederum disziplinierten Umgang mit Variablen und Rückgabetypen erfordert, da sonst zur Laufzeit Typ-bezogene Fehler auftreten können.

Es gibt verschiedene Implementierungen von Python (CPython, Jython, IronPython, PyPy), die je nach Projekt unterschiedliche Vorteile beinhalten könnten; diese werden hier aber nicht weiter vorgestellt.

Da Python eine große, engagierte Community besitzt (87174 Pakete im PyPI - the Python Package Index[7]), ist es in vielen Fällen nicht nötig, häufig auftretende Probleme von Grund auf neu zu programmieren. Einige Pakete, welche sich für die wissenschaftliche Arbeit mit

[6] Stand: 24.08.2016
[7] Stand: 24.08.2016

Python für hilfreich erweisen können, sind `Matplotlib` (Plotten von Graphen), NumPy/SciPy (Wissenschaftliche Berechnungen), `scikit-learn` (Machine Learning) und viele weitere. Außerdem enthält die Python Standardbibliothek bereits von Haus aus hilfreiche Pakete für Reguläre Ausdrücke, Datum und Zeit, Funktionen für URLs und vieles mehr. Ein weiteres Werkzeug mit positivem Effekt auf wissenschaftliche Arbeiten im Zusammenhang mit Python bieten Projekte wie Jupyter mit den IPython Notebooks, die das Teilen und das interaktive Verwenden von Python Code fördern.

Bei wissenschaftlichen Berechnungen ist Performance durchaus von Bedeutung, was die Einbindung von C oder C++ wünschenswert macht. In Python existieren automatisierte Werkzeuge, die genau diese Integration ermöglichen. Das Modul `weave.scipy` beispielsweise schreibt von 1,5-facher bis 30-facher Verbesserung von Algorithmen in C/C++ gegenüber selbigen in purem Python. Das ermöglicht die Vorteile von Python zu nutzen, aber gleichzeitig bei zeitkritischen Berechnungen nicht auf die Schnelligkeit anderer Programmiersprachen verzichten zu müssen.

Es ist allerdings auch zu bemerken, dass in Python teilweise Methoden undurchsichtig hinsichtlich ihres Ressourcen-Managements sind. So gibt es beispielsweise in Python 2.7 zwei Möglichkeiten eine for-Schleife aufzubauen. Dabei wird bei der ersten Variante über eine Liste iteriert, während die ressourcensparende zweite Variante mit „lazy-evaluation" funktioniert. Diese Konzepte und Eigenheiten müssen dem Entwickler beim Implementieren bekannt sein, um hochperformanten Code zu schreiben.

Wenn der Python eigene Dialekt, der sogenannte Pythonic Code Style berücksichtigt wird, resultiert das in einer guten Lesbarkeit und teilweise auch in der Vermeidung oben genannter negativer Eigenheiten, erfordert aber eine Zeitinvestition in den Code Style.

Weitere Sprachen

Es existieren viele weitere Sprachen, die relevant für Bibliotheken zur Linearen Algebra sind. Die noch junge Sprache Rust etwa vereint hohe Effizienz mit modernen Programmierkonzepten, und verhindert dabei durch eine spezielle, restriktive Semantik viele Arten von Fehlern. Zudem ist Rust auf einfache, sichere Parallelisierung ausgelegt. All diese Eigenschaften machen Rust zu einer vielversprechenden Sprache für Bibliotheken zu High-Performance-Berechnungen oder allgemein zu effizienter Linearer Algebra. Jedoch ist Rust momentan noch in einem frühen Stadium, weshalb auch die LA-Bibliotheken noch nicht allzu weit fortgeschritten sind. Dennoch befinden sich bereits verschiedene Projekte in aktiver Entwicklung, wie etwa `rusty-machine` (eine Bibliothek für Maschinelles Lernen, deren Modul `linalg` für Lineare Algebra zeitnah ausgelagert werden soll)[8] oder `ndarray` für grundlegende Matrizen-Operationen[9].

[8] https://www.reddit.com/r/rust/comments/4qvb2v/linear_algebra/d4wanpr
[9] http://bluss.github.io/rust-ndarray/master/ndarray/index.html

Portabilität und Bindings

Für viele wissenschaftliche Anwendungen für aufwendige Berechnungen, die auf hohe Performanz angewiesen sind, sind zunächst effiziente Systemsprachen wie C bzw. C++ naheliegend, doch falls beispielsweise verteilte Berechnungen mit Mobilgeräten eine Rolle spielen, müssen gegebenenfalls andere Programmiersprachen verwendet werden. In einem anderen Fall können große wissenschaftliche Workflows von der Datenbeschaffung bis zur Visualisierung und Auswertung auf einer einzigen Sprache basieren (z.B. Python), wobei jeder Bruch der Sprache die Komplexität der Methodik erhöht und die Wartbarkeit verringert.

Vorteilhaft ist in diesen Fällen der Einsatz solcher LA-Bibliotheken, die auf verschiedenen Systemen eingesetzt werden können und Bindings für verschiedene Sprachen besitzen. Somit kann das Potenzial des Frameworks voll ausgeschöpft werden.

Bei Bindings sind weitere Faktoren zu überprüfen:

- *Sind die Bindings vollständig?* Falls nur einige grundsätzliche Verfahren von sekundären Programmiersprachen angesprochen werden können, andere Verfahren hierbei jedoch fehlen, geht ein Großteil des Nutzens dieser Bindings verloren.
- *Welcher Performance-Overhead muss in Kauf genommen werden?* Falls die Schnittstellen hoch abstrahiert sind und viele „Konversionsschritte“aufweisen, kann das die Performanz der Berechnungen negativ beeinflussen.

Lizenz

Für die Nutzung einer Bibliothek gilt es grundsätzlich, die unterliegende Lizenz zu berücksichtigen. Bibliotheken mit proprietären Lizenzen dürfen gegebenenfalls nicht kostenfrei in öffentlichen Projekten eingesetzt werden; auf der anderen Seite verbieten Copyleft-Lizenzen die Software in proprietäre Anwendungen einzubinden, ohne den Quellcode offenzulegen. Je nach den individuellen Bedürfnissen des jeweiligen Projekts sollten unter Umständen bestimmte Bibliotheken im Vornherein ausgeschlossen werden, sofern deren Lizenz inkompatibel mit den Intentionen ist.[10]

GPL, LGPL und MPL

Die General Public License (GPL) gewährt die volle kommerzielle Nutzung und Weiterverbreitung der Software, setzt dabei aber voraus, dass jegliche Änderungen an der Software unter dieselbe Lizenz gestellt und veröffentlicht werden (*Copyleft*). Eine Einbindung GPL-lizensierten Codes in eigene Software ist nur zulässig, sofern diese ebenfalls GPL-lizensiert wird. Die GPL zählt daher zu den freigiebigsten Lizenzen aus Sicht der Softwarenutzer, bringt jedoch Einschränkungen aus Sicht der Entwickler mit sich. Um das starke Copyleft der GPL

[10]Die hier sowie in späteren Kapiteln aufbereiteten Informationen zu einzelnen Lizenzen sind [16f] entnommen.

im Hinblick auf Softwarebibliotheken zu relativieren, existiert die Lesser General Public License (LGPL), welche die Einbindung der (unveränderten) Software in proprietäre Software erlaubt. Somit können LGPL-lizensierte Bibliotheken und Frameworks bedenkenlos in eigener Software verwendet werden, ungeachtet deren Lizensierung. Bei der MPL, der *Mozilla Public License*, handelt es sich wie bei der LGPL ebenfalls um eine Lizenz mit schwachem Copyleft und vergleichbaren Bedingungen.

MIT-, Apache- und BSD-Lizenzen

Zu den freizügigsten Lizenzen zählen unter anderem die BSD-, Apache- und MIT-Lizenzen, denn diese enthalten kein Copyleft. Sie erlauben ebenso wie die GPL die umfassende Nutzung und Weiterverbreitung der Software, stellen es aber darüber hinaus frei, die Software zu verändern und proprietär zu verbreiten ohne den Code veröffentlichen zu müssen. Damit sind diese Lizenzen als besonders entwicklerfreundlich anzusehen. Von den BSD-Lizenzen existieren verschiedene Formen; die heute gängigste ist die 2-clause BSD, welche gegenüber den Vorgängern (3-clause und 4-clause) bestimmte Einschränkungen in Bezug auf das Marketing der Drittsoftware nicht mehr enthält.

Gemeinsam mit der GPL und der LGPL stellen diese Lizenzen einige der wichtigsten Lizenzen für Freie Software dar.

Proprietäre Lizenzen

Kommerzielle Bibliotheken können unter Umständen deutlich strengere Auflagen an die Nutzung und Einbindung setzen als die Lizenzen zu Freier Software. Im Falle einer proprietären Lizenz sollten die entsprechenden Bedingungen eingehend untersucht werden.

Quellen-Verfügbarkeit

Mit der Lizensierung verbunden, aber durchaus auch ein eigenes Kriterium ist die Verfügbarkeit des Quellcodes. Quelloffene Projekte sind häufig mit einer sehr lebendigen Community verbunden, wie die Beliebtheit großer Online-Repositories ähnlich zu Sozialen Netzwerken (insbesondere Github[11]) zeigt. Zudem kann bei Bedarf die Implementierung eines Verfahrens beliebig genau nachvollzogen werden, ohne auf eine erschöpfende Dokumentation angewiesen zu sein.

Community und Verbreitung

Da eine aktive und hilfreiche Nutzerbasis eines Softwareprojekts im Regelfall die verfügbare Online-Dokumentation (in Form von Mailinglisten, Foreneinträgen oder Antworten in entsprechenden Hilfe-Websites wie StackOverflow[12]) auf natürliche Weise erhöht, ist bei ansonsten

[11] https://www.github.com
[12] http://www.stackoverflow.com

gleicher Eignung tendenziell die Bibliothek mit größerer Verbreitung bzw. Community zu wählen.

Dokumentation

Die besten Features sind nur von geringem Wert, wenn deren Nutzung nicht hinreichend dokumentiert ist. Speziell im Bereich der untersuchten LA-Bibliotheken sollten die folgenden Kriterien für die Dokumentation zutreffen:

- *vollständig* – Jede nutzbare Methode sollte in der Dokumentation erwähnt, sowie deren korrekte Nutzung erklärt werden.
- *korrekt* – Die Dokumentation muss aktuell sein und den tatsächlichen Ist-Stand des Projekts abbilden.
- *einsteigerfreundlich* – Ein reines Glossar der verfügbaren Methoden ist geeignet für Kenner der grundsätzlichen Funktionsweise, doch Entwickler, die mit den Grundfunktionen der Bibliothek noch nicht vertraut sind, benötigen stattdessen ein Tutorial, oder aussagekräftige Minimal-Beispiele. Auch diese sollten im Rahmen der Dokumentation zur Verfügung gestellt werden.

2.2 Anwendungsgebiete

Die Anzahl und Vielfalt der Anwendungen für Lineare Algebra ist beachtlich. Im Folgenden werden nur einige der Gebiete erwähnt, welche auf Berechnungen der Linearen Algebra aufbauen.

- Computergrafik: Eines der Kernelemente der Linearen Algebra stellen die Vektoren dar, welche als räumliche, richtungsweisende Linien interpretiert werden können. Sie sind das elementare Mittel zur Beschreibung dreidimensionaler Umgebungen und daher – gemeinsam mit entsprechenden Transformationen – unerlässlich für die Berechnung räumlicher Vorgänge sowie für grafische Darstellung (beispielsweise eines dreidimensionalen Raums auf einem zweidimensionalen Bildschirm). [Mut06]
- Optimierung: Zur Lösung von mehrdimensionalen Optimierungsproblemen, wie sie an zahlreichen Stellen in der Praxis auftreten, müssen ebenfalls Methoden der Linearen Algebra herangezogen werden. [Mut06]
- Simulationen: Für den Erkenntnisgewinn in der Wissenschaft bieten Computersimulationen neben Theorien und Experimenten eine weitere wichtige Methode, durch welche viele komplexe Sachverhalte analysiert und erklärt werden können. Für viele Simulationen müssen dabei große Lineare Systeme periodisch gelöst werden sowie auf Basis Linearer Algebra mehrdimensionale Zeitschrittverfahren berechnet werden. [Buc09]

- Statistik und Data Science: Für die Verarbeitung, Einordnung und Auswertung großer Datenmengen sind Methoden wie Lineare Regression und andere Lineare Ausgleichsrechnungen unumgänglich.
- Machinelles Lernen und Künstliche Intelligenz: Das autonome, „intelligente“ Handeln von Agenten erfordert einen umfassenden mathematischen Unterbau. Ob Graphentheorie, Markov-Ketten oder Spieltheorie, die Daten werden meist in Matrixform gehalten und ebenso manipuliert. [Buc09]
- Kryptographie
- Bildverarbeitung und -kompression
- …

3 Planung

Im Folgenden werden die vorbereitenden Schritte zu einem Vergleich der LA-Bibliotheken durchgeführt, darunter die Festsetzung einer bestimmten Methodik, die Definition der Vergleichskriterien sowie die Konzeption eines Performancevergleichs.

3.1 Methodik

Im Folgenden wird die zugrundeliegende Methodik und Vorgehensweise für den Vergleich der Bibliotheken beschrieben.

In einem ersten Schritt werden die einzelnen Bibliotheken isoliert betrachtet, Informationen darüber zusammengetragen und durch praktische Anwendung Eindrücke gesammelt. So ergibt sich ein klares, abstrahiertes Bild der Bibliothek mit ihren Eigenschaften, ihren Vorzügen und Nachteilen.

Erst in einem zweiten Schritt werden die Bibliotheken aufgrund der in Erfahrung gebrachten Eigenschaften miteinander auf Grundlage der später definierten Vergleichskriterien kritisch miteinander verglichen. Ziel der Arbeit soll dabei nicht sein, ein eindimensionales „Ranking“ der Bibliotheken zu bestimmen – handelt es sich doch bei der Bewertung einer Bibliothek um ein vieldimensionales Problem, das nicht ohne starken Verlust der Aussagekraft derart reduziert werden kann. Stattdessen sollen im Kontext verschiedener Anwendungsfälle jeweils bestimmte Bibliotheken empfohlen werden, die sich besonders gut für diese Anwendung eignen und sich hier gegenüber den anderen Bibliotheken durchsetzen können. Auch die Empfehlung der Vermeidung einer Bibliothek für einen bestimmten Anwendungsfall kann sich anbieten.

Der ebenfalls durchzuführende Performancevergleich ist hier dem zweiten Schritt zuzuordnen, da Performancezahlen an sich erst dann mit einer Bewertung versehen werden können, wenn ein Vergleich herangezogen werden kann. Ein zentrales Ziel des Performancevergleichs ist dabei neben der Erhebung von Performancewerten auch die praktische Arbeit und Implementierung mit jeder der untersuchten Bibliotheken, da somit die Bedienbarkeit derselben effektiv überprüft werden kann.

3.2 Vergleichskriterien

Um einen objektiven Vergleich der Bibliotheken zu ermöglichen, werden nun die zentralen Vergleichskriterien und, sofern praktikabel, dazugehörige Metriken aufgestellt. Nun werden diese Eigenschaften als entsprechende Vergleichskriterien formuliert.

3.2.1 Funktional

Die wichtigsten funktionalen Eigenschaften von LA-Bibliotheken wurden in Kapitel 2.1.1 vorgestellt und erläutert.

Generelle Herangehensweise

Die Herangehensweise der Bibliothek bezüglich des Backends wird zur Kenntnis genommen, aber nicht inhärent als ein Vergleichskriterium herangezogen, es sei denn, durch die Wahl des Backends ergeben sich klare Vorteile in der Nutzung (etwa durch erhöhte Performance, vereinfachte Installation oder leichtere Bedienung).

Die Möglichkeiten der Parallelisierung von Operationen wird bewertet anhand des Anteils der tatsächlich parallelisierbaren Berechnungen sowie anhand des daraus hervorgehenden Performancegewinns (siehe Performancevergleich).

Die Möglichkeit der Vektorisierung und/oder der Verwendung eines Grafikprozessors für schnelle parallele Berechnungen wird positiv zur Kenntnis genommen. Ein Praxistest solcher Berechnungen konnte aus praktischen Gründen im Rahmen der Arbeit nicht vorgenommen werden.

Features

Die Vielfalt der vorhandenen Verfahren trägt unmittelbar zur Bewertung bei. Betrachtet werden die folgenden Operationen und Strukturen:

1. Grundlegende Rechenoperationen mit Matrizen und Vektoren (Addition, Multiplikation, Invertierung)
2. Speziell für dünn besetzte Matrizen konzipierte Datenstrukturen
3. Berechnung mit einfacher oder doppelter Genauigkeit, sowie mit komplexen Zahlen
4. Direkte Löser für Lineare Gleichungssysteme der Form $Ax = b$
5. Zerlegungen für Matrizen (LU, ILU, Cholesky, QR)

6. Iterative Näherungsverfahren für Lineare Gleichungssysteme (vor allem das GMRES-Verfahren)

Performance

Die Performance der Bibliotheken wird durch einen eigens durchgeführten Performancevergleich verglichen. Dieser wird in Kapitel 3.3 näher beleuchtet.

3.2.2 Nicht-funktional

Auch nicht-funktionale Aspekte der Bibliotheken sollen in die Bewertung einfließen.

Portabilität und Bindings

Die Portabilität wird anhand der unterstützten Architekturen und/oder Betriebssysteme bewertet. Bindings in andere Sprachen werden eingestuft anhand ihrer Vollständigkeit und deren aktiver Weiterentwicklung.

Lizenz und Quellen-Verfügbarkeit

Freizügige, Freie Softwarelizenzen (ohne oder mit schwachem Copyleft) werden positiv zur Kenntnis genommen. Auf Freie Softwarelizenzen mit starkem Copyleft wird explizit hingewiesen, da hier bei Entwicklung proprietärer Software unter Umständen rechtliche Probleme entstehen können. Ebenso wird im Fall proprietärer Lizenzen darauf hingewiesen und die Lizenz je nach den Nutzungsbedingungen bewertet.

Die Verfügbarkeit des Quellcodes wird positiv bewertet.

Community und Verbreitung

Für die Verbreitung und die Community der Bibliothek wird unter anderem eine Metrik angesetzt, welche der Anzahl der Fragestellungen zu der Bibliothek auf der Hilfewebsite *stackoverflow.com* entspricht. Diese Metrik gibt nicht zwangsläufig ein repräsentatives Abbild der Community wieder, da auch Aspekte wie die Qualität der vorhandenen Dokumentation und die allgemeinde Bedienbarkeit Einfluss auf die Anzahl der Fragen nehmen; dennoch eignet sie sich gut, um eine grobe Richtlinie für die Verbreitung der Bibliothek zu erhalten.

Sofern Dienste wie *github.com* genutzt werden, stellen die Anzahl der Beitragenden, der geöffneten (und wieder geschlossenen) Themen und Bugs und die Anzahl der Forks weitere Ansatzpunkte für die Community dar.

Dokumentation

Das primäre Kriterium bezüglich der Dokumentation stellt ein Dokument (als Webseite oder PDF) mit einer Referenz aller ansprechbaren Schnittstellen dar. Hierbei muss die Funktionsweise, die Art der Eingabeparameter und die Ausgabe erklärt werden. Zudem wird die Existenz zusätzlicher Dokumentation bewertet, die etwa Einsteigern die Entwicklung mit der Bibliothek erleichtert oder „Schablonen“ für häufig auftretende Problemstellungen anbietet und erklärt.

3.3 Performancevergleich

Einen zentralen Teil des Vergleichs stellt ein Benchmark dar, bei welchem die Bibliotheken in Gesichtpunkten der Performance direkt miteinander verglichen werden.

3.3.1 Anforderungen

Die Anforderungen an den Performancevergleich lauten wie folgt:

- Es soll die Performance gängiger, für die Praxis relevanter Operationen ermittelt werden.
- Es soll *nicht* die Performance aller Features der Bibliotheken untersucht werden; vielmehr soll es sich um geeignete „Stichproben“ handeln.
- Für den Benchmark sollen eine ansteigende Dimension der Eingabe sowie die parallele Ausführung (sofern möglich) untersucht werden.
- Für speziell konditionierte Eingaben soll jeweils die (offiziell vorhandene) Implementierung verwendet werden, welche am Speziellsten auf die Eingabe angepasst ist.

3.3.2 Auswahl der Berechnungen

Gemäß den aufgestellten Anforderungen wurden die folgenden Berechnungen ausgewählt:

- Die Lösung eines Gleichungssystems $Ax = b$ mithilfe einer LU-Zerlegung. Diese Zerlegung im Voraus verringert die für die Berechnung benötigten Schritte.
  Die Berechnung soll mit verschiedenartig konditionierten Eingaben erfolgen:
  - Dichte Matrizen (alle Einträge der Matrix werden zufällig bestimmt und sind im Allgemeinen ungleich null): Dabei handelt es sich um die allgemeinste Problemstellung, welche die höchste Komplexität besitzt.

  - Dünne Matrizen (jeder Eintrag der Matrix ist zu einem (geringen) Prozentsatz ungleich null): Diese werden in vielen realen Problemstellungen aufgestellt, da komplexe lineare Systeme oft aus vielen linearen Beziehungen zwischen einigen wenigen Parametern bestehen. Bei dieser Art von Matrizen kann eine hohe Speichereffizienz erreicht werden und damit auch die Komplexität der Berechnungen erheblich reduziert werden. [GJ+10, Dokumentation: *Sparse Matrix Format*]
  - Tridiagonalmatrizen (bis auf die Hauptdiagonale und die beiden anliegenden Diagonalen sind alle Einträge null): Diese Matrizen entstehen aus Problemstellungen wie der Interpolation von Funktionswerten durch kubische Splines und können bei korrekter Implementierung mit asymptotischer Komplexität $O(n)$ gelöst werden [REU, S. 89f.].
- Die iterative Lösung eines Gleichungssystems $Ax = b$ mithilfe des GMRES-Verfahrens. Iterative Lösungsverfahren erarbeiten eine zunehmend bessere Approximation des Ergebnisses, bis ein bestimmtes Residuum gegenüber der exakten Lösung erreicht wird. Sie können mit geringer Komplexität eine Näherungslösung berechnen, die unter Umständen ebenso akzeptabel ist wie die exakte Lösung. Das GMRES-Verfahren stellt unter diesen Verfahren den Quasi-Standard dar. Es wird ausschließlich auf dünn besetzten Matrizen ausgeführt. [16c]

Jede der genannten Berechnungen wird sequentiell sowie mit mehreren Threads parallel ausgeführt, sofern die Bibliothek dies unterstützt.

Nicht alle der untersuchten Bibliotheken weisen eine Implementierung des GMRES-Verfahrens auf; diese werden somit für diesen Teil des Performance-Vergleichs nicht berücksichtigt.

Nähere Details zur Implementierung und Durchführung des Performancevergleichs können Kapitel 5 entnommen werden.

4 Behandelte Bibliotheken

4.1 Eigen

Eigen ist eine C++-Bibliothek für Lineare Algebra. Sie entstand aus dem KDE-Projekt für eine Linux-Desktopumgebung und dem daraus hervorgegangenen Bedürfnis nach einer „einzelnen, vereinheitlichten Matrix-Bibliothek“für die praktische Nutzung durch das KDE-Framework [GJ+10].

4.1.1 Generelle Herangehensweise

Hervorzuheben ist an Eigen, dass es sich um eine reine *Header*-Bibliothek handelt; es sind keine vorkompilierten Binärobjekte vorhanden, stattdessen besteht die nutzbare Software allein aus einer übersichtlichen Menge von `.h`-Dateien. Die Einbindung der Bibliothek in Anwendungscode geschieht somit mit minimalem Aufwand; die Header werden in ein beliebiges Verzeichnis abgelegt, und können dann in eigenem C++-Quellcode durch einfache `#include`-Anweisungen eingebunden werden. Eine Installation entfällt damit ebenfalls (abgesehen von der Verschiebung des Eigen-Verzeichnisses in ein globales Verzeichnis, um, falls gewünscht, die Bibliothek systemweit verwenden zu können).

Backend

Interessanterweise enthält Eigen keine Abhängigkeiten zu BLAS-Implementierungen; damit entfallen auch native Möglichkeiten zur Parallelisierung sowie das Potential zur Nutzung der GPU[1].

[1] http://eigen.tuxfamily.org/index.php?title=Todo#BLAS_and_LAPACK_backends

Template-Programmierung

Die Bibliothek ist auf Grundlage von *Templates* programmiert; einem Feature von C++, das generische Implementierungen voraussetzt und diese dann ohne zusätzlichen Aufwand mit realen Objekten realisieren lässt. Bereits zur Kompilierzeit sind die einzusetzenden Klassen statisch bekannt, und die generischen Platzhalter werden durch diese ersetzt. So werden alle Datenstrukturen und Berechnungen in Eigen grundsätzlich mit generischen Skalaren ausgedrückt; bei der Initialisierung derselben lässt sich dann ein konkreter Datentyp wie `float` oder `double` einsetzen. Auch komplexe Datentypen sowie Ganzzahlen werden voll unterstützt[2].

Templates bringen jedoch auch Komplikationen mit sich; versucht man etwa, eine Matrix des Typen `double` mit einem Vektor des Typen `float` zu multiplizieren, so erhält man vom Compiler die folgende, sperrige Fehlermeldung (gefolgt von vielen weiteren, ähnlichen Meldungen):

```
./Eigen/src/Core/Product.h: In instantiation of
'struct Eigen::internal::traits<Eigen::Product<Eigen::Matrix<double, -1, -1>,
Eigen::Matrix<float, -1, 1>, 0> >':
./Eigen/src/Core/Product.h:115:7: required from 'class
Eigen::internal::dense_product_base<Eigen::Matrix<double, -1, -1>,
Eigen::Matrix<float, -1, 1>, 0, 7>'
./Eigen/src/Core/Product.h:147:7: required from 'class
Eigen::ProductImpl<Eigen::Matrix<double, -1, -1>,
Eigen::Matrix<float, -1, 1>, 0, Eigen::Dense>'
./Eigen/src/Core/Product.h:71:7: required from 'class
Eigen::Product<Eigen::Matrix<double, -1, -1>,
Eigen::Matrix<float, -1, 1>, 0>'
test.cpp:13:18: required from here
./Eigen/src/Core/Product.h:29:127: error: no type named 'ReturnType' in 'struct
Eigen::ScalarBinaryOpTraits<double, float,
Eigen::internal::scalar_product_op<double, float> >'
typedef typename ScalarBinaryOpTraits<typename traits<LhsCleaned>::Scalar,
typename traits<RhsCleaned>::Scalar>::ReturnType Scalar;
```

Listing 4.1: Fehlermeldung bei Vermischung von `float`- und `double`-Datenstrukturen

Deutlich benutzerfreundlicher wäre hier eine kurze Fehlermeldung mit dem Hinweis, dass eine Multiplikation von Matrizen verschiedener Skalartypen nicht zulässig ist.

Parallelisierung

In Eigen lassen sich einige Operationen, darunter Matrixmanipulationen, mithilfe von `OpenMP` parallelisieren. Aus Nutzersicht ist dies sehr einfach zu bewerkstelligen; es muss lediglich dem Compiler die Nutzung von OpenMP mitgeteilt werden (für GCC etwa der Schalter `-fopenmp`)

[2] http://eigen.tuxfamily.org/dox/TopicCustomizingEigen.html#CustomScalarType

und für die Ausführung selbst entweder durch eine Zuweisung im Code oder durch die Umgebungsvariable `OMP_NUM_THREADS` die Zahl der zu startenden Threads festgelegt werden. Ein kleines Beispiel mit einer Multiplikation zweier 700×700 Matrizen ergab auf einer Dual-Core-CPU die folgenden Ergebnisse:

```
Genutzte Threads: 1  Genutzte Threads: 3
real  0m3.973s       real  0m2.284s
user  0m3.968s       user  0m6.052s
sys   0m0.004s       sys   0m0.000s

Genutzte Threads: 2  Genutzte Threads: 4
real  0m2.186s       real  0m2.193s
user  0m4.008s       user  0m7.292s
sys   0m0.004s       sys   0m0.008s
```

Listing 4.2: Zeitmessungen einer Matrizenmultiplikation unter variabler Anzahl zu nutzender Threads

Die Gesamtzeit der Berechnung wird bei der Nutzung zweier Threads fast halbiert, was auf einen guten Lastenausgleich und einen geringen Overhead bei der Parallelisierung spricht. Für eine noch höhere Anzahl Threads ergeben sich aufgrund der berechnenden Architektur keine weiteren Verbesserungen mehr, doch die Berechnung wird auch nicht erheblich verlangsamt.

Laut der Dokumentation ist bisher jedoch nur die Matrixmultiplikation sowie das `PartialPivLU`-Verfahren derart parallelisierbar. Damit sind insbesondere vorimplementierte Lösungsverfahren nur sequentiell ausführbar, was für aufwendige Berechnungen einen deutlichen Nachteil darstellt.

4.1.2 Funktionsumfang

Gegliedert in *Module* (thematischen Zusammenfassungen bestimmter Strukturen und Algorithmen) unterstützt Eigen eine Vielfalt von Operationen, die jeweils für spezielle Szenarien optimiert wurden. Enthalten sind unter anderem Datenstrukturen für spezielle Matrizen wie Bandmatrizen, symmetrische und Hermitsche Matrizen. Neben elementaren Dreiecks-Lösungsverfahren für LGS wie der LU– oder Cholesky-Zerlegung existieren auch Implementierungen für QR–, QL– und generalisierte Faktorisierungen. Zudem werden in einem dedizierten Modul auch viele Verfahren in spezieller Implementierung für dünn besetzte Matrizen angeboten, etwa `SparseLU` oder `SparseQR`.

Zusatzmodule

Neben den offiziellen Implementierungen bietet Eigen auch eine Menge von Modulen mit dem Status *unsupported* (nicht unterstützt) an, die von der Community geschrieben wurden

und zur freien Nutzung verfügbar sind. Darunter befinden sich unter anderem Fast Fourier Transformationen, nichtlineare Optimierungen, und Löser für Polynome.

4.1.3 Sprache und Bindings

Die Wahl der Sprache C++ hat zum einen die Konsequenz, dass Algorithmen ohne weitere Schnittstellen in hochperformantem Code implementiert werden können. Zum anderen ist C++ eine Hochsprache, die Modularisierung und Abstraktion ermöglicht, wovon Eigen konsequent Gebrauch macht; es wird auf eine abstrakte Schnittstelle aufgesetzt, sodass der Anwendungsentwickler durchgängig auf Ebene von Klassen und Objekten agiert.

Eigen besitzt durch das Projekt `RcppEigen` vollständige Bindings für die Sprache R. Bindings für Java bietet das Projekt `jeigen` an, die jedoch nicht vollständig sind, sondern nur die grundlegenden Operationen unterstützen. Für Python existieren verschiedene Projekte, die jeweils bestimmte Bindings umsetzen; `SpPy` für einige `Sparse`-Module (Alpha-Phase), `pyeigen` (Pre-Alpha-Phase, wird momentan offenkundig nicht weiterentwickelt) und `minieigen` für einige grundlegende Module.

4.1.4 Bedienbarkeit

> Implementing an algorithm on top of Eigen feels like just copying pseudocode.
>
> *(Eigen Homepage [GJ+10])*

In der Tat wirkt der Code für konzeptionell einfache Berechnungen unter Benutzung von Eigen übersichtlich und gut lesbar. Durch die Objektorientierung werden Vektoren und Matrizen sehr einfach initialisiert, und einfache imperative Anweisungen wie `solver.solve(...)` machen verständlich, was durch deren Ausführung geschieht.

Beispielhaft ist in Listing 4.3 die Anwendung des in Eigen implementierten GMRES-Verfahrens zu sehen; zwar mithilfe eines *unsupported* Moduls, das aber dennoch regulär in der Standardinstallation enthalten ist.

```
int n = 10000;
VectorXd x(n), b(n);
SparseMatrix<double> A(n,n);
// fill A and b

GMRES<SparseMatrix<double> > solver(A);
x = solver.solve(b);

std::cout << "#iterations: " << solver.iterations() << std::endl;
std::cout << "estimated error: " << solver.error() << std::endl;
```

```
// update b, and solve again
x = solver.solve(b);
```

Listing 4.3: Beispielcode für eine GMRES-Berechnung mit Eigen

Die Anwendung des aufwendigen GMRES-Verfahrens verläuft hier aus Sicht des Anwendungscodes als Blackbox-Aufruf; genauere Kenntnisse über die interne Herangehensweise des Algorithmus sind nicht notwendig.

4.1.5 Dokumentation

Bei der vollständigen Onlinedokumentation von Eigen[3] handelt es sich nicht nur um eine reine API-Referenz, sondern ein umfangreiches Nachschlagewerk. Die Dokumentation enthält an prominenter Stelle eine bemerkenswert kurze *Getting Started*-Anleitung mit „Installations"-Anweisung, minimalem Codebeispiel zur Erzeugung und Ausgabe einer Matrix, und einer ausführlichen Erklärung dazu. Auch Entwickler mit geringer Praxiserfahrung in Richtung Linearer Algebra und/oder Programmierung mit C++ können sich hier sofort zurechtfinden.

Für jedes der Kapitel – *Manipulation dichter Matrizen und Arrays*, *Dichte lineare Probleme und Zerlegungen*, sowie *Dünnbesetzte Lineare Algebra* – wird eine Schnellreferenz angeboten, in welchen kompakt die benötigten Codezeilen für verschiedene Aufgaben nachzulesen sind. Die Dokumentation geht auch in die fachliche Tiefe und beschreibt Vorgehensweisen, um je nach Problem die bestmögliche Performanz zu erzielen. Dabei werden zum ersten Mal erwähnte Konstrukte und Begriffe aus der Linearen Algebra meist noch einmal kurz beschrieben, womit auch mit nur grundlegenden Kenntnissen der Linearen Algebra wesentliche Teile der Bibliothek genutzt werden können.

4.1.6 Lizenz und Quellen-Verfügbarkeit

Da Eigen aus dem KDE-Projekt hervorgegangen ist, das sich seinerseits intensiv für Freie Software engagiert, ist auch Eigen Freie Software, also insbesondere quelloffen und privat sowie kommerziell nutzbar. Während frühere Versionen von Eigen unter der *Lesser General Public License* lizensiert sind, ist die aktuelle Version lizensiert unter der *Mozilla Public License 2.0*. Dabei handelt es sich um eine schwache Copyleft-Lizenz; das bedeutet, dass alle Änderungen am Code öffentlich zugänglich und ebenfalls gemäß der MPL lizensiert werden müssen, doch der Code darf auch mit proprietärer Software kombiniert werden, so lange der MPL-lizensierte Code in separaten Dateien vorliegt. Binärpakete, die den MPL-Code enthalten, dürfen proprietär lizensiert werden, solange die Quelldateien unter der MPL-Lizenz verfügbar sind.

[3] http://eigen.tuxfamily.org/dox/

Unter den heutigen Releases von Eigen ist noch LGPL-lizensierter Code vorhanden; dieser lässt sich, falls gewünscht, durch ein Präprozessorsymbol explizit unnutzbar machen.[4] Zudem existieren verschiedene Schnittstellen hin zu Bibliotheken von Dritten; deren Lizensierung ist teilweise proprietär, worauf unter anderem in den entsprechenden Modulbeschreibungen hingewiesen wird.

4.1.7 Bewertung

Bei Eigen handelt es sich um eine in erster Linie benutzerfreundliche und angenehm verwendbare Bibliothek, die durch eine Vielfalt von Features und gute Dokumentation überzeugt. Sie eignet sich daher bestens für Projekte verschiedenster Art und ermöglicht ein vergleichsweise schnelles Anfertigen von zuverlässigem Produktivcode.

Um wissenschaftliche Berechnungen in großem Stil durchzuführen, ist zu einem gewissen Teil die mangelhaft vorhandene Parallelisierbarkeit der Berechnungen (sowie das Fehlen etwaiger Möglichkeiten zur Berechnung auf GPUs) als Nachteil zu bewerten, da somit das Potential der Hardware meist nicht voll ausgeschöpft werden kann.

[4] http://eigen.tuxfamily.org/index.php?title=Main_Page#License

4.2 PETSc

Das **Portable, Extensible Toolkit for Scientific Computation** (Portabler, Erweiterbarer Werkzeugkoffer für Wissenschaftliches Rechnen), kurz **PETSc** (sprich: Petsie) ist eine Suite verschiedener Datenstrukturen und Verfahren speziell für wissenschaftliche Berechnungen, die als partielle Differentialgleichungen modelliert vorliegen. Die Besonderheit des Frameworks liegt in seinem hohen Fokus auf inhärente Parallelität der Berechnungen, woraus eine hohe Skalierbarkeit derselben folgt. [BAA+16b]

4.2.1 Generelle Herangehensweise

PETSc ist darauf ausgerichtet, anspruchsvolle numerische Probleme durch massiv paralleles Rechnen zu lösen. [BAA+16b] Das Framework baut auf das *Message Passing Interface* (MPI) auf, das im Folgenden kurz erklärt wird.

MPI

Das *Message Passing Interface* (Nachrichtenaustausch-Schnittstelle) ist ein Kommunikationsprotokoll für die parallele Ausführung von Berechnungen. Das Prinzip basiert auf einer standardisierten Interprozesskommunikation durch Nachrichten (*Messages*) über eine wohldefinierte Schnittstelle. Durch diese Abstraktion lassen sich Programme mittels MPI sowohl auf parallelen als auch auf verteilten Systemen (ohne gemeinsamem Speicher) ausführen.

Das MPI-Modell basiert auf die Gruppierung einzelner Prozesse durch sogenannte *Communicators* („Kommunikatoren“). Ein Kommunikator fasst eine Menge von Prozessen zusammen, unter welchen dorthin versendete Nachrichten sichtbar sind. Beispiele für Kommunikatoren sind `MPI_COMM_WORLD`, was für alle Prozesse des Programms gilt, oder `MPI_COMM_SELF`, was nur den aktuellen Prozess einschließt.

Backend

PETSc baut auf eine BLAS-Implementierung und auf die *Linear Algebra Package* LAPACK auf. Diese Module werden für die Nutzung benötigt, können aber im Rahmen der Installation ausgewählt und automatisch mitinstalliert werden.

Steuerung über Kommandozeilenoptionen

PETSc-Routinen lassen sich grundsätzlich mittels der dem Programm übergebenen Argumente parametrisieren. Alle wichtigen Aspekte wie der Typ von bzw. der Umgang mit Vektoren und Matrizen, die Wahl eines Preconditioners, einer Krylow-Methode und vieles mehr, können über die Optionen eingerichtet werden, solange im Anwendungscode entsprechend `PetscInitialize(&argc,&args,file,help);` aufgerufen wird (wobei optional `help` ein darzustellender Hilfetext für die Kommandozeile und `file` ein optionale, zusätzliche Options-Konfigurationsdatei ist). Die globale PETSc-Hilfeausgabe bietet über 170 Kommandozeilenoptionen an. Für die parameterlose Ausführung des später vorgestellten Beispielcodes (Listing 4.4) wird etwa als Krylow-Methode standardmäßig GMRES und als Preconditioner eine Jacobi-Umformung ausgewählt. Es kann aber auch eine Vielzahl alternativer Verfahren verwendet werden.

4.2.2 Funktionsumfang

PETSc ist äußerst reich an Features; da es sich nicht nur um eine Bibliothek zur Linearen Algebra, sondern um ein vollständiges Framework für die Lösung derer Anwendungen handelt, weist PETSc eine hohe Menge implementierter Verfahren auf.

PETSc ist modular aufgebaut. Auf die grundlegenden Implementierungen zu *Vektoren* und *Matrizen* bauen die *Vorkonditionierer* auf, zu denen unter anderem das Jacobi–, LU– oder ILU-Verfahren zählen. Die *Krylow-Methoden* wenden Krylow-Unterraum-Verfahren zur Lösung dünnbesetzter linearer Gleichungssysteme an, und enthalten beispielsweise das GMRES-Verfahren. Eine weitere Ebene darüber schließlich finden sich *Zeitschrittverfahren* zur numerischen Lösung Gewöhnlicher Differentialgleichungen (z.B. Runge-Kutta-Verfahren) sowie *Nichtlineare Löser* (z.B. Newton-Methode). Jede der angesprochenen Ebenen kann für sich genutzt werden, wobei die Verfahren höherer Ebenen Gebrauch der darunterliegenden Implementierungen machen. So kann PETSc für grundlegende Aufgaben der Linearen Algebra ebenso genutzt werden wie für Problemlösungen auf höheren Abstraktionsebenen, etwa einer Zeitschrittberechnung für anspruchsvolle numerische Simulationen.

Zusatzmodule

PETSc trägt die Erweiterbarkeit durch ein „E“ wie *Extensible* bereits im Namen; für jedes der Module von PETSc wird eine frei zugängliche Plugin-Schnittstelle angeboten. Diese Schnittstellen werden auch lebhaft genutzt, und es bestehen viele Erweiterungen, wobei es sich insbesondere um Faktorisierungen und Lösungsverfahren handelt[5].

[5] Vorhandene Verfahren, darunter viele Zusatzmodule:
http://www.mcs.anl.gov/petsc/documentation/linearsolvertable.html

4.2.3 Sprache und Bindings

PETSc ist in der Sprache C verfasst. Durch das Projekt `petsc4py` existieren vollständige Bindings für Python; das Projekt `jPETScTao` gewährt Bindings für Java, doch diese sind bisher unvollständig, und teilweise problematisch, da die Parallelität von PETSc auf die Virtuellen Maschinen von Java übertragen werden müssen.

4.2.4 Bedienbarkeit

PETSc ist ein Framework um die Programmiersprache C herum, und ist spezialisiert auf die effiziente Ausführung großer Berechnungen durch Nutzung massiver Parallelität. Diese ambitionierte Zielsetzung schlägt sich auch in der Gestaltung des Anwendungscodes nieder; verglichen mit der hoch-abstrahierten Bibliothek Eigen oder numerischen Berechnungen in Python ist für ein funktionierendes PETSc-Programm merklich mehr Code notwendig. Der Vorteil ist dabei im Gegenzug, dass hocheffizient und parallel berechnet wird.

Geschuldet der Programmiersprache existieren für mathematische Konstrukte wie Skalare, Vektoren und Matrizen zwar Objekte (`structs`), jedoch sind auf diesen selbst keine Operationen definierbar. Infolge dessen sind alle Operationen in einer Präfix-Notation implementiert. Zudem existieren nicht die von modernen Sprachen bekannten Ausnahmebehandlungen, weshalb jede PETSc-eigene Methode einen Rückgabewert des Typen `PetscErrorCode` hat, welcher über den Erfolg der Operation Auskunft gibt und im direkten Anschluss daran überprüft werden sollte. Somit entstehen Anweisungen wie `ierr = MatSetSizes(A,PETSC_DECIDE, PETSC_DECIDE, n, n); CHKERRQ(ierr);` anstelle von `A.setSize(n,n);`, was die Klarheit und Lesbarkeit des Codes verringert. Der natürliche Rückgabewert der Methoden ist bereits durch den Fehlercode belegt, weshalb der Speicherort des eigentlichen Ergebnisses als Argument übergeben werden muss. Für Entwickler, die moderne und objektorientierte Sprachen gewohnt sind, ist entsprechend eine längere Einarbeitungsphase sowie ein höherer Aufwand in der Entwicklung und Wartung zu erwarten.

PETSc bietet eine sehr hohe Transparenz bei der Ausführung von Berechnungen; es lassen sich hoch differenzierte Beschreibungen über die gewählten Algorithmen, Eigenschaften der mathematischen Objekte, und genutzten Algorithmen ausgeben (vgl. Listing 4.5). Dies ist im Speziellen von hohem Nutzen, wenn

- mit verschiedenen Verfahren experimentiert wird;
- Performance-Benchmarking durchgeführt wird; oder
- ein Vergleich zwischen verschiedenen Vorgehensweisen durchgeführt wird.

Eine hohe Nutzerfreundlichkeit von PETSc ist damit nicht im Bereich des tatsächlichen Quellcodes, jedoch durchaus im Aufruf der entstehenden Programme zu verzeichnen. Die grundsätzliche Möglichkeit der Parametrisierung durch Kommandozeilen-Argumente, jeweils mit passender Hilfestellung, ermöglichen die Implementierung sehr generischer Programme,

welche dann über die Kommandozeile (oder umschließende Skripte wie `bash`) je nach Situation ansprechbar sind. Zusammen mit der beliebigen Auswahl der Ein- und Ausgabedaten und vielen Optionen zum Logging der Berechnung eignen sich PETSc-Programme bestens zur Nutzung durch Skripte und automatisierte Berechnungen.

```
Vec        x, b, u;  /* approx. Ergebnis, rechte Seite, exaktes Ergebnis */
Mat        A;        /* Matrix des LGS */
KSP        ksp;      /* Kontext des Linearen Solvers */
PC         pc;       /* Kontext des Preconditioners */
PetscReal  norm;     /* Norm des Fehlers der Loesung */
PetscErrorCode ierr;
PetscInt   i,n = 10,col[3],its;
PetscMPIInt size;
PetscScalar neg_one  = -1.0,one = 1.0,value[3];
PetscBool  nonzeroguess = PETSC_FALSE;

// Initialisiere die MPI-Umgebung
PetscInitialize(&argc,&args,(char*)0,help);
ierr = MPI_Comm_size(PETSC_COMM_WORLD,&size);CHKERRQ(ierr);
if (size != 1) SETERRQ(PETSC_COMM_WORLD,1,"This is a uniprocessor example only!");
ierr = PetscOptionsGetInt(NULL,NULL,"-n",&n,NULL);CHKERRQ(ierr);
ierr = PetscOptionsGetBool(NULL,NULL,"-nonzero_guess",&nonzeroguess,NULL);CHKERRQ(ierr);

// Erstelle Vektoren x,b,u
ierr = VecCreate(PETSC_COMM_WORLD,&x);CHKERRQ(ierr);
ierr = PetscObjectSetName((PetscObject) x, "Solution");CHKERRQ(ierr);
ierr = VecSetSizes(x,PETSC_DECIDE,n);CHKERRQ(ierr);
ierr = VecSetFromOptions(x);CHKERRQ(ierr);
ierr = VecDuplicate(x,&b);CHKERRQ(ierr);
ierr = VecDuplicate(x,&u);CHKERRQ(ierr);

// Erstelle Matrix
ierr = MatCreate(PETSC_COMM_WORLD,&A);CHKERRQ(ierr);
ierr = MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,n,n);CHKERRQ(ierr);
ierr = MatSetFromOptions(A);CHKERRQ(ierr);
ierr = MatSetUp(A);CHKERRQ(ierr);

// Fuelle Matrix (tridiagonal) ...

// Initialisiere exaktes Ergebnis; berechne rechte Seite (b)
ierr = VecSet(u,one);CHKERRQ(ierr);
ierr = MatMult(A,u,b);CHKERRQ(ierr);

// Erstelle Linearen Solver, setze Optionen
ierr = KSPCreate(PETSC_COMM_WORLD,&ksp);CHKERRQ(ierr);
// Die das LGS definierende Matrix dient hier auch als Preconditioning Matrix
ierr = KSPSetOperators(ksp,A,A);CHKERRQ(ierr);
ierr = KSPSetFromOptions();CHKERRQ(ierr);

// Loese das LGS, stelle Loesung dar
```

```
ierr = KSPSolve(ksp,b,x);CHKERRQ(ierr);
ierr = KSPView(ksp,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);

// Fehler pruefen
ierr = VecAXPY(x,neg_one,u);CHKERRQ(ierr);
ierr = VecNorm(x,NORM_2,&norm);CHKERRQ(ierr);
ierr = KSPGetIterationNumber(ksp,&its);CHKERRQ(ierr);
ierr = PetscPrintf(PETSC_COMM_WORLD,"Norm of error %g, Iterations
    %D\n",(double)norm,its);CHKERRQ(ierr);

// Aufraeumen
ierr = VecDestroy(&x);CHKERRQ(ierr); ierr = VecDestroy(&u);CHKERRQ(ierr);
ierr = VecDestroy(&b);CHKERRQ(ierr); ierr = MatDestroy(&A);CHKERRQ(ierr);
ierr = KSPDestroy(&ksp);CHKERRQ(ierr);
ierr = PetscFinalize();
```

Listing 4.4: Beispielcode für das (sequentielle) Lösen eines Tridiagonal-LGS mit PETSc

```
KSP Object: 1 MPI processes
  type: gmres
    GMRES: restart=30, using Classical (unmodified) Gram-Schmidt Orthogonalization with no
        iterative refinement
    GMRES: happy breakdown tolerance 1e-30
  maximum iterations=10000, initial guess is zero
  tolerances: relative=1e-05, absolute=1e-50, divergence=10000.
  left preconditioning
  using PRECONDITIONED norm type for convergence test
PC Object: 1 MPI processes
  type: lu
    LU: out-of-place factorization
    tolerance for zero pivot 2.22045e-14
    matrix ordering: nd
    factor fill ratio given 5., needed 1.35714
      Factored matrix follows:
        Mat Object:    1 MPI processes
          type: seqaij
          rows=10, cols=10
          package used to perform factorization: petsc
          total: nonzeros=38, allocated nonzeros=38
          total number of mallocs used during MatSetValues calls =0
            not using I-node routines
  linear system matrix = precond matrix:
  Mat Object: 1 MPI processes
    type: seqaij
    rows=10, cols=10
    total: nonzeros=28, allocated nonzeros=50
    total number of mallocs used during MatSetValues calls =0
      not using I-node routines
Norm of error 8.88178e-16, Iterations 1
```

Listing 4.5: Ausgabe von Listing 4.4 mit übergebener Option `-pc_type lu`

4.2.5 Dokumentation und Community

PETSc besitzt ein umfassendes Nutzerhandbuch, das als verlinktes und durchsuchbares PDF vorliegt. Darin werden alle Operationen und Funktionsweisen der Bibliothek erklärt. [BAA+16a]

Vermissen lässt sowohl das Handbuch als auch die Website ein zentrales Starttutorial, das durch ein Minimalbeispiel leicht verständlich in die Nutzung der Bibliothek einführt. Auf der Homepage wird auf zahlreiche Tutorial-Folien und -Papiere verwiesen, was jedoch beim Einstieg in die Bibliothek eher überwältigt.

Die Community der aktiven PETSc-Nutzer ist auf verschiedenen Portalen vertreten. Hilfe bei Problemen mit der Bibliothek findet sich teilweise auf bekannten Websites wie `stackoverflow.com` sowie auf der (auch von gängigen Suchmaschinen erfassten) Mailingliste für PETSc-Nutzer. Anfragen finden hier meist eine Antwort, die das vorhandene Problem löst und/oder erklärt, was späteren Entwicklern wiederum weiterhilft, wenn sie auf ähnliche Probleme stoßen.

4.2.6 Lizenz und Quellen-Verfügbarkeit

Die Bibliothek ist quelloffen; der Quellcode ist auf Bitbucket gehostet[6]. Auch der Entwicklungsprozess ist vollständig transparent; so lassen sich zu jedem Zeitpunkt Entwicklungsversionen, Nightly Builds und Testlauf-Ergebnisse einsehen.

PETSc unterliegt einer *2-clause BSD Lizenz*; diese gewährt Entwicklern alle relevanten Freiheiten, insbesondere die kommerzielle Nutzung und Weiterverbreitung. Es handelt sich um keine Copyleft-Lizenz; somit können auch proprietäre Anwendungen ohne weiteres Teile von PETSc verwenden und einbinden.

4.2.7 Bewertung

PETSc ist ein sehr mächtiges Framework, das für seine Nutzer durch die hohen Konfigurationsmöglichkeiten und sperrige Syntax ein hohes Hintergrundwissen voraussetzt, diese aber nach fertiger Implementierung weitreichende Möglichkeiten nicht nur zur Berechnung selbst, sondern auch zur Evaluation derselben gewährt. Da es sich nicht um eine reine Bibliothek zur Linearen Algebra handelt, sondern auch darauf aufsetzt und eine große Menge von Lösern für lineare und nichtlineare Systeme sowie einer Reihe von Zeitschrittverfahren bietet, erfüllt PETSc voll und ganz seinen Anspruch, ein umfassendes Framework für wissenschaftliche Berechnungen zu sein.

[6] https://bitbucket.org/petsc/petsc

Für die Ausführung umfangreicher Berechnungen aus Forschungszwecken etwa ist PETSc daher hervorragend geeignet; für Berechnungen in (konzeptionell) kleinem Maße hingegen – ohne Ausnutzung massiver Parallelität, und nur um des Ergebnisses willen – ist der Einarbeitungs– und Implementierungsaufwand im Allgemeinen als vergleichsweise groß zu bewerten. Die umfassenden Kommandozeilen-Optionen bieten dabei zwar eine gute alternative Form einer weithin „abstrahierten“ Programmierung, was die Implementierung des initialen Anwendungscodes jedoch nicht einfacher gestaltet. Wie auch anhand der trockenen Gestaltung der Homepage zu sehen, ist es nicht das (Haupt-)Ziel der Entwickler, gegenüber Endanwendungsentwicklern aktiv für PETSc zu werben.

4.3 Armadillo

Armadillo ist eine Softwarebibliothek für die Programmiersprache C++, welche auf eine gute Balance zwischen Benutzerfreundlichkeit und Performanz abzielt, wodurch ein geradliniger Übergang vom wissenschaftlichen Prototyp zum produktionsreifen Algorithmus realisiert werden soll. Die Anwendungsgebiete von Armadillo liegen nach eigener Aussage unter anderen in den Bereichen maschinelles Lernen, Mustererkennung, Computer Vision, Signalverarbeitung, Bioinformatik, Statistik und Ökonometrie. [SC]

4.3.1 Generelle Herangehensweise

Die Codierung erfolgt wie bei Eigen mittels C++-Template-Programmierung. Alle gängigen Standardklassen wie (dünnbesetzte) Matrizen, Skalare und Würfel sind enthalten, wobei ein `Field` verschiedene dieser Klassen unterschiedlicher Größe enthalten darf. Dazu kommen über 200 zugehörige Funktionen für die Manipulation der Daten in den hieraus erzeugten Objekten. [San10]

Weiterhin kann der Nutzer aus den folgenden Datentypen für die Initialisierung von Matrizen auswählen: `float, double, std::complex<float>, std::complex<double>, short, int, long` und vorzeichenlose Versionen von `short, int und long`. [SC16]

Template-Programmierung

Durch Template-Programmierung können in Armadillo die folgenden primitiven Operatoren auf alle korrekt gekennzeichneten Objekte angewendet werden, was direkte und intuitive Anweisungen im Code ermöglicht:

```
+ - * / % == = <= >= < >
```

Nutzer können Armadillo zusätzliche Informationen über Objekte geben, sodass Operationen effizienter durchgeführt werden können. Als Beispiele seien für eine Matrix `A` die Klassifikatoren `diagmat(A)` und `trimatu(A)` genannt, welche die Matrix als Diagonal- bzw. Dreiecksmatrix kennzeichnen, sodass die Algorithmen zum Lösen der LGS effizienter ablaufen können. Im gleichen Zuge wird auch `inv(inv(A))` mathematisch korrekt nichts berechnen und lediglich `A` zurückgeben, denn die Auswertung von Anweisungen wird durch Pipelining verzögert, bis der eigentliche Zugriff auf das Ergebnis zur Laufzeit erfolgt.

Backends

Die *Armadillo run-time library* wird bei der Installation durch CMake generiert und ist ein Wrapper um alle der Bibliotheken *OpenBLAS, LAPACK, BLAS, Intel MKL, ARPACK, SuperLU und ATLAS*, die auf dem System vorhanden sind. Danach werden sie bei Verwendung des Compilerschalters `-larmadillo` mit dem entstehenden Programm verknüpft ("linking") und verwendet. Auch GPU-Beschleunigung von Matrixmultiplikationen kann durch Verwendung von *NVBLAS* oder *AMD ACML* gewährleistet werden. [SC16]

Parallelisierung

Für die Parallelisierung in Armadillo sind allein die verfügbaren Backends ausschlaggebend. So können mit *LAPACK* die Berechnungen von Eigenwerten, Invertierungen usw. asynchron ablaufen, doch bei der Nutzung von *BLAS* für Matrixmultiplikationen sollte beachtet werden, dass die Geschwindigkeit der Berechnungen in höchstem Maße von der jeweiligen BLAS-Implementierung abhängig ist. Es kann daher von Vorteil sein, eine der o.g. hochperformanten Varianten zu wählen, beispielsweise OpenBLAS, Intel MKL oder AMD ACML, um deren Parallelisierung bestmöglich auszunutzen. [SC16]

Metaprogrammierung

Durch C++ Metaprogrammierung wird eine verzögerte Berechnung zur Kompilierzeit umgesetzt, sodass mehrere Operationen zu einer verschmolzen und temporäre Berechnungsergebnisse vermieden werden, was anderweitig auch als „Pipelining“ bezeichnet wird.

4.3.2 Installation

Als erster Schritt sollten die gewünschten Backends und die zugehörigen `development`-Pakete installiert werden, welche im Normalfall im Paketmanager der Distribution bereitstehen sollten.

Danach kann das Armadillo-Paket von der Sourceforge Webseite heruntergeladen und die C++ Header-Dateien im Systempfad verankert werden.[7]

[7] http://arma.sourceforge.net/download.html

4.3.3 Funktionsumfang

Das Fehlen einer geeigneten Implementierung für das GMRES-Lösungsverfahren überrascht, da Armadillo ansonsten eine reichhaltige Menge an Lösungsverfahren bietet. Die Bibliothek überzeugt auch mit verschiedenen Methoden zur Zufallsgenerierung von Objekten, von `linspace` und `logspace` zur Erstellung von Vektoren mit linear bzw. logarithmisch verteilten Elementen bis zu `toeplitz` für den Aufbau einer Toeplitz-Matrix.

Eine Schnittstelle zu anderen Bibliotheken erhält Armadillo sowohl durch eine mögliche Nutzung von Sekundärspeicher ("auxiliary memory") für Matrizen/Würfel als auch durch den Zugriff auf Elemente über Iteratoren im Sinne der STL, oder aber über die `.memptr()`-Funktion, welche direkt einen Zeiger auf den Speicherbereich von Matrizen zurückgibt. [San10]

Die Unterstützung von C++11 ist ein weiterer positiver Aspekt von Armadillo, da dies zurzeit von vielen Entwicklern gewünscht wird.

Erweiterbarkeit / Zusatzmodule

Überraschenderweise finden sich trotz der hohen Resonanz im Internet nur sehr wenige angepasste Versionen oder Zusatzmodule für Armadillo. Es ist schwer, hierfür einen klaren Grund auszumachen; möglicherweise ist schlicht ein Großteil aller Anwendungsfälle durch die vielfältigen Konfigurationsmöglichkeiten der zahlreichen verfügbaren Backends abgedeckt.

Im-/Export

Armadillo kann Dateien in den Formaten CSV, HDF5, PGM, PPM, reinem ASCII, sowie einem eigenen `arma_binary`-Format speichern und laden. Man beachte hierbei jedoch, dass dünnbesetzte Matrizen in allen Formaten außer `arma_binary` nicht direkt als solche geladen werden können, sondern als Standard-Matrizen eingelesen und danach konvertiert werden müssen.

Sprache und Bindings

Es stehen Schnittstellen bzw. Bindings für die folgenden Sprachen bereit:

- **Java**: *ArmadilloJava*
- **R**: durch die *RcppArmadillo* Erweiterung
- **Python**: *cyarma*, welches sich zum aktuellen Zeitpunkt (Oktober 2016) noch in einer frühen Entwicklungsphase befindet

4.3.4 Bedienbarkeit

Die Bibliothek wartet mit allerhand nützlichem Syntaktischen Zucker auf und erlaubt das Lösen von Problemen mit straffem und prägnantem Quellcode, und auch die Ausnahmebehandlung ist sauber und liefert recht präzise Anhaltspunkte für eine Fehlersuche.

Ein Beispiel ist die komfortable Konvertierung von Matrizentypen, hier die Erstellung eines "Spaltenvektors" `colvec` auf Grundlage einer beliebigen Matrix `m`:

```
colvec vec = conv_to<colvec>::from(m);
```

Armadillos Syntax hat eine starke Ähnlichkeit mit Matlab- bzw. Octave-Syntax, was aus der Entwicklungsgeschichte der Bibliothek als Ersatz für diese Mathematikumgebungen hervorgeht und gewährleistet, dass mathematische Operationen in gewohnter und natürlicher Weise ausgedrückt werden können. Es ist deshalb auch nicht verwunderlich, dass Armadillo mit einer Tabelle zur Übersetzung von in diesen Sprachen verfassten Programmen aufwartet.

Die Konfiguration von Armadillo erfolgt über eine inkludierte Datei, welche alle verfügbaren Optionen (in auskommentierter Form) enthält. [SC16]

Im folgenden Beispiel wird ein Gleichungssystem aus einer zufällig generierten, dünnbesetzten Matrix und einem ebenfalls zufälligen Vektor mit normalverteilten Elementen gelöst.

```
#include "armadillo"

using namespace std;
using namespace arma;

// Compile with g++ example.cpp -o example -O2 -larmadillo
int main(int argc, char* argv[])
{
  // sprandu() uses a uniform distribution in the [0,1] interval
  sp_mat A = sprandu<sp_mat>(100, 100, 0.1); // rows, cols, density

  // vector with regularly spaced elements
  vec v = linspace< vec>(0, 99);

  mat X;
  X = spsolve(A, v);

  std::cout << X << std::endl;
}
```

Listing 4.6: Beispiel für das Lösen eines Gleichungssystems

4.3.5 Dokumentation und Community

Die gesamte Dokumentation befindet sich auf einer einzelnen HTML-Seite und bietet ausreichende, aber teilweise knappe Erklärungen zu allen angebotenen Funktionen. Dort findet sich auch die Versionshistorie wieder, sodass mittels der Suchfunktion des Browsers sofort *alle* Informationen über Armadillo abrufbereit sind.

Es existiert für Armadillo keine dedizierte Community, jedoch findet man auf Foren Dritter zu praktisch jedem Problem mit einer schnellen Suche Anhaltspunkte für die Lösung.

4.3.6 Lizenz und Quelloffenheit

Armadillo wird unter der *Mozilla Public License* (MPL) vertrieben, wodurch der originale Quellcode in jeder proprietären Distribution stets mitgeführt werden und für proprietäre ausführbare Dateien immter der unter MPL lizensierte Quellcode veröffentlicht werden muss.

4.3.7 Bewertung

Eine angenehme Syntax und eingebaute Optimierungen tragen dazu bei, dass das selbsterklärte Ziel, eine leicht zu benutzende Bibliothek zu sein, bei Armadillo erreicht wurde und sich auch Laien auf dem Gebiet der linearen Algebra hier schnell zurechtfinden sollten.

Dennoch können nicht alle Anwendungsfälle von vornherein optimiert werden und die Fehlersuche bei hochkomplexem Code kann aufgrund der Template-Metaprogrammierung zum Problem werden.

Eine klare Empfehlung für Armadillo kann deshalb und aufgrund der leicht zu erlernden Syntax bei Einsteigern in die LA, Nutzern von Mathematikprogrammen wie Matlab und Octave, sowie auch für Prototyping [San10] in wissenschaftlichem und industriellem Kontext ausgesprochen werden.

> Our recommendation: Use it!
> Don't try to understand it.
>
> *(http://hpac.rwth-aachen.de/teaching/sem-accg-14/Armadillo.pdf)*

4.4 ViennaCL

ViennaCL ist eine quelloffene, wissenschaftliche Softwarebibliothek für die Programmiersprache C++. Sie ist ausgelegt für abstrakten Zugriff auf Berechnungen in der linearen Algebra auf parallelisierten Systemen wie Grafikkarten und Mehrkernprozessoren (MIC) und bietet iterative Gleichungslöser mit optionalen Vorkonditionierern für große Gleichungssysteme.

4.4.1 Generelle Herangehensweise

ViennaCL ist wie Eigen eine "header-only"-Bibliothek, was bedeutet, dass das Softwarepaket lediglich aus C++ Header-Dateien besteht.

Die CUDA und OpenCL Backends für Grafikprozessoren bieten an sich eine hohe Beschleunigung, für eine effiziente Programmierung auf CPUs müssen aber händische Optimierungen des OpenMP Backends in Kauf genommen werden. Hierzu schreiben die Entwickler, dass die ursprünglich nur als Ausweichoption für reine CPU-Systeme gedachte Implementierung von OpenMP mittlerweile ausgereift sei. Dennoch sollte man sich in seinem Anwendungskontext nur dann auf ViennaCL verlassen, wenn die Zielplattform bekannt ist und somit die entsprechenden Optimierungen vorgenommen werden können, oder wenn vor der Entwicklung feststeht, dass hohe Rechenleistung auf den Grafikprozessoren vorhanden sein wird.

Als Basis für die Ausführung auf verschiedenen Geräten wird eine vereinheitlichte Schicht für den Zugriff auf CUDA, OpenCL und/oder OpenMP genutzt. Daher ist ViennaCL nicht auf Produkte eines bestimmten Herstellers beschränkt, sondern kann auf verschiedenen Plattformen (alle aktuellen CPU- und GPUs) betrieben werden.

Backend(s)

ViennaCL unterstützt CUDA, OpenCL und OpenMP Backends, zudem ist für uBLAS geschriebener Quellcode durch einfache Veränderung der Imports sofort auch für ViennaCL gültig. Die iterativen Gleichungslöser können außerdem direkt mit Eigen und *MTL 4* genutzt werden.

Parallelisierung

Performanz-Charakteristika können sich auf den verschiedenen Backends stark unterscheiden. Vor allem wird sich die Nutzung von GPUs nicht auszahlen, falls die Eingabegröße zu klein sein sollte, wodurch die Datenübertragung über PCI-Express der ausschlaggebende Flaschenhals sein wird. [SC16] Dafür können über *BLAS* der Stufe 3 auf mittel- bis hochklassigen Grafikprozessoren im Vergleich zu den Einkern-CPU-Implementierungen gute Ergebnisse erzielt werden. Allgemein kann die höchste Performance aber nur durch vorsichtiges Tuning für das spezifische Endsystem erreicht werden und da der Fokus bei ViennaCL stark auf der

Berechnung mittels Grafikprozessoren liegt, kommen Mehrkern-CPU-Architekturen ohne GPU tendenziell zu kurz.

4.4.2 Installation

Da ViennaCL eine reine header-Bibliothek ist, reicht es aus, den über die Versionsverwaltung oder die Webseite bezogenen Wurzelordner in das Projektverzeichnis oder den globalen Systempfad zu kopieren (auf Unix-Systemen normalerweise `/usr/include/` oder `/usr/local/include/`).

Für die Nutzung der verfügbaren Backends können die folgenden Compilerschalter, oder alternativ ein Flag zu Beginn des Quellcodes gesetzt werden, welche für den Fall einer simultanen Nutzung eine hierarchische Struktur aufbauen und hierdurch die Prioritäten (in absteigender Reihenfolge) regeln:

`VIENNACL_WITH_CUDA`: CUDA
`VIENNACL_WITH_OPENCL`: OpenCL
`VIENNACL_WITH_OPENMP`: CPU mit OpenMP

4.4.3 Funktionsumfang

Als Typen dünnbesetzter Matrizen kann zwischen CSR, COO, ELL, HYB und Sliced-ELL gewählt werden, wobei Objekte komfortable Schnittstellen von und zu `STL`, `uBlas`, Armadillo, Eigen und `MTL4` haben. Der Vorteil hierbei ist, dass Algorithmen wie beispielsweise die iterativen Gleichungslöser (Mixed-Precision CG, BiCGStab, GMRES) direkt mit Objekten aus den genannten Bibliotheken arbeiten können und so eine Parallelisierung auf Mehrkernprozessoren oder Grafikkarten für Module stattfinden kann, die das nativ nicht anbieten. Es ist außerdem eine Reihe von Vorkonditionierern wie hochauflösende ICHOL- und ILU0-Faktorisierung, LU Faktorisierung (mit Schwellenwert), Block-ILU Konditionierer (mit ILU0 oder ILUT), Mehrgitterverfahren, sowie Jacobi und Zeilennormalisierung vorhanden.

Im-/Export

Vektoren und Matrizen können im standardisierten `matrix_market`-Format gespeichert und geladen werden.

4.4.4 Sprache und Bindings

Mit *PyViennacl* existiert ein vollständiger Python-Wrapper, der die erforderlichen Bindings in idiomatischem Python Code-Style bereitstellt. Dabei wurde besonderer Wert auf eine zu den weit verbreiteten Bibliotheken NumPy bzw. SciPy ähnliche Syntax gelegt, sodass sich erfahrene Python-Programmierer sofort zurechtfinden und auf einfache Weise die massiven Stärken in der Parallelisierung von ViennaCL ausnutzen können.

Außerdem gibt es durch RViennaCL Bindings für die Programmiersprache R.

4.4.5 Bedienbarkeit

Da es sich bei ViennaCL hauptsänlich um eine wissenschaftliche Biliothek handelt, wird ein hoher Wert auf Standards gelegt, was vor allem eine Portierung bestehender Programme erleichtert. ViennaCL implementiert alle Operationen des BLAS-Standards, wodurch wie Eingangs beschrieben Wiederverwendbarkeit und eine hohe Effizienz der Programmierungen ermöglicht wird.

Codebeispiel

Dem Folgenden Code können Dateinamen für eine Matrix und einen Vektor als Kommandozeilenparameter übergeben werden, woraufhin dann das entsprechende Gleichungssystem mithilfe des GMRES-Verfahrens gelöst wird.

```
#include <iostream> #include "viennacl/io/matrix_market.hpp" #include
"viennacl/linalg/gmres.hpp" #include "viennacl/linalg/detail/ilu/ilut.hpp"

using namespace viennacl::linalg; // to keep solver calls short

typedef double ScalarType;

void solveGmresSparse(std::string filename_m, std::string filename_v) {

 // Read Matrix and Vector from file viennacl::compressed_matrix<double>
 vcl_matrix; viennacl::compressed_matrix<double> vcl_rhs_2d;
 viennacl::io::read_matrix_market_file(vcl_matrix, filename_m, 1);
 viennacl::io::read_matrix_market_file(vcl_rhs_2d, filename_v, 1);
 viennacl::vector<double> vcl_rhs(vcl_rhs_2d.size1());
 viennacl::vector<double> vcl_result; for (int i = 0; i < vcl_rhs_2d.size1();
 i++) { vcl_rhs[i] = vcl_rhs_2d(i, 0); }

 ilut_tag ilut_conf(10, 1e-5);
 ilut_precond<viennacl::compressed_matrix<ScalarType> > vcl_ilut(vcl_matrix,
 ilut_conf);
```

```
  // solution with gmres vcl_result = solve(vcl_matrix, vcl_rhs, gmres_tag(),
  vcl_ilut); }

int main(int argc, char **argv) { solveGmresSparse(std::string(argv[1]),
  std::string(argv[2])); }
```

Listing 4.7: Beispiel für das Lösen eines Gleichungssystems mittels GMRES

4.4.6 Dokumentation und Community

Für die Dokumentation kommt der Generator *Doxygen* zum Einsatz, wodurch Standards wie gute Durchsuchbarkeit und Versionshistorie gewährleistet werden. Sie ist äußerst umfassend und es werden zahlreiche Codebeispiele für die einzelnen Kapitel gelistet. Auch die mathematischen Hintergründe werden durch Einbindung eines Formelparsers anschaulich dargestellt.

Für die Recherche bei Problemen sollte die Suchmaschine Google im Gegensatz zu beispielsweise DuckDuckGo verwendet werden, denn dort ist die spezifische Mailingliste erfasst, welche als Hauptmedium für Konversationen im Zusammenhang mit der Nutzung von ViennaCL dient.

4.4.7 Lizenz und Quelloffenheit

Auch ViennaCL wird ohne Einschränkungen unter der *Mozilla Public License* (MPL) vertrieben, wodurch die gleichen Anforderungen wie bei Armadillo gelten.

4.4.8 Bewertung

Die vielen Schnittstellen sowohl zu anderen Bibliotheken als auch Backends machen ViennaCL zu einer ausgezeichneten Wahl, wenn bereits eine Codebasis für eine dieser Plattformen besteht. Außerdem können bei Verfügbarkeit hoch parallelisierbarer Ressourcen vor allem über die Nutzung der Grafikprozessoren sehr gute Ergebnisse erzielt werden. Es sollte daher aber auch von ViennaCL abgesehen werden, wenn nur kleine Datenmengen verarbeitet werden oder nur eine geringe Anzahl von Prozessoren bereitsteht.

> ViennaCL aims at compatibility with as many other libraries as possible.
>
> *([Rup14])*

4.5 NumPy und SciPy

NumPy ist ein Erweiterungsmodul für die Programmiersprache Python, ist quelloffen und wird seit Dezember 2001 entwickelt. NumPy zeichnet sich mit mehr als 500 Beitragenden und einer seit 2001 ansteigenden Anzahl an monatlichen Entwicklern[8] aus. Das Paket ist zu großen Teilen in C und Python geschrieben.

Das Framework **SciPy** nutzt NumPy, um häufige wissenschaftliche Probleme zu lösen. Dabei sind die zeitkritischen Schleifen entweder in C oder Fortran geschrieben, der größte Teil ist dennoch in Python verfasst.

4.5.1 Generelle Herangehensweise

NumPy und SciPy sind Bibliotheken, die eine Fülle von wissenschaftlichen Werkzeugen mit sich bringen und für häufig auftretende Probleme Lösungen bieten. Einige der vorhandenen Funktionen dienen zu Verfahren, wie der Interpolation, Verfahren der Statistik oder die Verarbeitung von dünnbesetzten Matrizen. Die angebotenen Lösungen für Probleme der linearen Algebra werden im folgenden genauer betrachtet.

Backend

Die Implementierung bei NumPy beruht auf den Funktionen der sogenannten „Basic Linear Algebra Subprograms", zusätzlich verwendet NumPy auch LAPACK. Diese Bibliotheken dienen der Effizienz bei den Berechnungen.

4.5.2 Funktionsumfang

Arrays in NumPy

Für Matrizen werden NumPy-Arrays (*numpy.ndarray*) verwendet. Diese können über ein Vielfalt von Methoden und einfacher Syntax neu angeordnet, gefüllt, kopiert und manipuliert werden. Zusätzlich kann bei Matrizen einfach angegeben werden mit welcher Genauigkeit Float-Zahlen dargestellt werden sollen.

Da in der Standardeinstellung die Genauigkeit der installierten Python-Umgebung verwendet wird (zum Beispiel 64-Bit), kann über eine niedrigere Genauigkeit Speicher entlastet werden.

[8] https://www.openhub.net/p/numpy/contributors/summary

Einlesen und Speichern

Der Anwendungsfall Matrizen einzulesen oder in eine Datei zu speichern ist mit Numpy sehr einfach möglich. Dazu gibt es mehrere Möglichkeiten, die unterschiedliche Vorteile haben. So ist das Einlesen und Speichern von Matrizen in Textdateien möglich. Die durch das Speichern entstandene Textdatei ist lesbar und kann modifiziert werden.

```
import numpy as np

# Erstellung eines Numpy Arrays
m = np.array([ [1, 2, 3],
[4, 5, 6],
[7, 8, 9]])

# Speichern in eine Textdatei
np.savetxt('somefile.txt', m)

# Form der Textdatei:
# 1.00000000000000000e+00 2.00000000000000000e+00 3.00000000000000000e+00
# 4.00000000000000000e+00 5.00000000000000000e+00 6.00000000000000000e+00
# 7.00000000000000000e+00 8.00000000000000000e+00 9.00000000000000000e+00

# Einlesen aus der Datei
q = np.loadtxt('somefile.txt')

# Speichern in eine Binaerdatei
np.save('somefile', m)

# Einlesen aus der Binaerdatei
p = np.load('somefile.npy')
```

Listing 4.8: Einlesen und Speichern mit Numpy

Eine weitere Möglichkeit Matrizen in eine Datei zu speichern sowie aus einer Datei zu laden bieten die Befehle *save()* und *load()*. Dabei werden Binärdateien erzeugt und die Daten dort gespeichert. Diese Art, Daten aus einer Datei zu laden, ist vor allem bei großen Daten schneller.

In Listing 4.8 sind beide Arten einzulesen in einem Codebeispiel dargestellt. Eine weitere Option, wenn Speicherplatz eine Rolle spielt ist der Befehl *numpy.savez()*. Dabei werden die gespeicherten Daten noch komprimiert, was den Befehl jedoch langsamer als *numpy.save()* macht. [Bre12, Seite 12 - 14]

Funktionen zur linearen Algebra

Funktionen zur linearen Algebra sind über *numpy.linalg*, sowie *scipy.linalg* zugänglich. Dabei enthält *scipy.linalg* alle Funktionen von *numpy.linalg* und fügt einige weitere hinzu. [16d]

Einige in *scipy.linalg* enthaltenen Funktionen sind Invertierung, Determinanten finden, Normen berechnen, Zerlegungen errechnen, Eigenwerte und Eigenvektoren errechnen.

Zerlegungen sind für das Lösen von linearen Gleichungssystemen von Bedeutung. Die bei *scipy.linalg* enthaltenen Zerlegungen sind folgende:

- LU-Zerlegung
- Cholesky-Zerlegung
- QR-Zerlegung
- Schur-Zerlegung
- Interpolative Zerlegung

Zu den Zerlegungen sind ebenfalls Funktionen zur Lösung von linearen Gleichungssystemen vorhanden. So berechnet die Funktion *scipy.linalg.lu_solve(a, b)*, unter Eingabe der LU-Zerlegung *a* einer Matrix und einem Vektor *b*, den Lösungsvektor. Weitere Verfahren in *scipy.linalg* sind das Lösen auf Grundlage der Cholesky-Zerlegung und der QR-Zerlegung.

Das Paket *scipy.sparse.linalg* enthält ebenfalls Funktionen zur Berechnung von Lösungen von linearen Gleichungssystemen, allerdings für dünnbesetzte Matrizen. Eine dieser Funktionen ist *scipy.sparse.linalg.gmres*, welche mit dem GMRES-Verfahren eine Lösung einer dünnbesetzen Matrix und einem Vektor errechnet.

In Listing 4.9 ist beispielhaft angegeben, wie ein lineares Gleichungssystem mithilfe des GMRES-Verfahrens gelöst wird. Dabei muss die Matrix *m* und der Vektor *v* der Methode *scipy.sparse.linalg.gmres(m, v)* lediglich als Parameter übergeben werden. Das Ergebnis ist ein Tupel aus dem resultierenden Vektor und der Information über den Ausgang der Berechnung.

```
import numpy as np
import scipy.sparse.linalg as lin

# Erstellung einer Matrix m
m = np.array([[1, 2, 3],
              [1, 1, 1],
              [3, 3, 1]])

# Erstellung eines Vektors v
v = np.array([[10], [11], [12]])

# Loese LGS mit gmres-Verfahren
lsg = lin.gmres(m,v)

# Ausgabe: (array([ 22.5, -22. , 10.5]), 0)
# Tupel aus Ergebnisvektor und
# Info (0: successful exit)
```

Listing 4.9: Lösung eines linearen Gleichungssystems mit GMRES.

4.5.3 Bedienbarkeit

Die Bibliotheken ermöglichen, wie in Listing 4.8 und Listing 4.9 zu sehen ist, mit wenigen Codezeilen bereits das Einlesen, Speichern und Lösen von Matrizen.

Die Syntax der Methoden in den Bibliotheken ist einfach gestaltet und ermöglicht es schnell Quelltext zu verfassen um das betreffende Problem zu lösen. Allerdings ist zu beachten, dass beispielsweise die Matrixmultiplikation nicht, wie zu vermuten wäre, mit dem Befehl *A * b* funktioniert, sondern mit *A.dot(b).* Auf Eigenheiten dieser Art muss anfangs geachtet werden.

Da SciPy ein Zusatz zu NumPy ist, kommt es vor, dass Pakete beider Bibliotheken gleiche Methoden enthalten, was zu Verwirrung führen kann. So enthalten die Pakete *numpy.linalg*, *scipy.linalg* und *scipy.sparse.linalg* alle Funktionen der linearen Algebra. Eine kurze Recherche bringt dennoch schnell Auskunft über die Unterschiede der Pakete und hilft bei der Entscheidung, welches Paket verwendet werden sollte. Im Fall von *numpy.linalg* und *scipy.linalg* bietet *scipy.linalg* eine komplettere Lösung, da dort ein Fortran Wrapping für LAPACK verwendet wird.

Python ist eine Programmiersprache, die mit aber auch ohne vorherigen Programmierkenntnissen sehr schnell erlernt werden kann. Dadurch wird auch die Benutzung von Numpy und SciPy, die wie beschrieben eine sehr kurze und einfache Syntax haben, begünstigt. Da die Bibliotheken nicht nur auf die lineare Algebra spezialisiert sind, kann eine Vielfalt von für wissenschaftliche Berechnungen relevante Methoden direkt aus diesen Bibliotheken verwendet werden.

4.5.4 Dokumentation und Community

Die Dokumentation beider Bibliotheken ist unter *www.scipy.org* zugänglich. Dort sind Verlinkungen zu den Bibliotheken und deren Inhalt zu finden.

Zur Einführung in die Bibliotheken sind dort direkt Tutorials vorhanden, welche die grundlegenden Funktionen und Befehle erklären. Dort sind auch bereits Hinweise auf mögliche Stolpersteine bei den ersten Schritten mit den Bibliotheken angegeben.

Die Dokumentation ist in einer API Referenz, welche die *docstrings* enthält, zusammengefasst. Die Beschreibungen der Funktionen sowie der Parameter und Rückgabewerte sind kurz und einfach verständlich formuliert. Dabei sind direkt Verlinkungen zu der Stelle im Quelltext vorhanden, so ist das genauere Betrachten der Implementierung ebenfalls möglich.

SciPy und Numpy gehören zu einer Menge von wissenschaftlichen Berechnungswerkzeugen. Darin befinden sich auch Bibliotheken wie zum Beispiel Matplotlib, das eine einfache Visualisierung ermöglicht. Der Bezug zu solchen Bibliotheken legt eine Verwendung über die lineare Algebra hinaus, nahe.

Eine Mailingliste dient zu Ankündigungen und zur Organisation des Projekts. So gibt es Mailinglisten für Diskussionen über Numpy oder für Fragen von SciPy Benutzern. Auch die Archive der Mailinglisten sind zugänglich und können durchsucht werden. Zudem ist Scipy äußerst stark auf stackoverflow.com und ähnlichen Webseiten vertreten, was Fehlerbehebungen meist einfach gestaltet.

Es finden Konferenzen, welche Bezug zu SciPy und wissenschaftlichen Python Projekten haben statt. Als Beispiel ist EuroSciPy zu nennen, welche 2016 in Deutschland stattfand.

NumPy und SciPy haben beide über 400 Entwickler, die zu den Projekten beigetragen haben.

4.5.5 Lizenz und Quellen-Verfügbarkeit

NumPy ist lizenziert unter der *BSD-3-Clause*, welche die kommerzielle Benutzung mit oder ohne Anpassungen gestattet. Die Lizenz, der SciPy unterliegt, ist die sehr ähnliche *BSD license*.

4.5.6 Bewertung

Die Verwendung von Numpy und SciPy beitet sich an, wenn auf vielfältige Features Wert gelegt wird und auch auf Methoden, die die lineare Algebra übersteigen. Zusätzlich können diese Bibliotheken mit ihrer Nähe zu anderen wissenschaftlichen Bibliotheken des *SciPy-Stacks* aufwarten. Die Benutzerfreundlichkeit ermöglicht ein schnelles Einarbeiten sowie einen angenehmen Entwicklungsprozess. Durch eine hohe Anzahl an Nutzern sind viele Probleme, die beim Entwickeln auftreten bereits beantwortet im Internet zu finden.

Wegen der Skriptsprachen-Natur der Bibliothek kann die Performance für Berechnungen mit großer Eingabe im Allgemeinen jedoch kaum an vergleichbare kompilierte Bibliotheken herankommen und ist für High-Performance-Berechnungen großer Skalierung daher nur sehr eingeschränkt zu empfehlen.

5 Implementierung

Im Folgenden werden technische Details zur Implementierung der betrachteten Anwendungen, insbesondere zum Zweck des Performancevergleichs, beschrieben.

5.1 Programmierung mit den Bibliotheken

Obwohl gewisse Grundaufgaben wie die Matrizenmultiplikation von allen betrachteten Bibliotheken gleichermaßen unterstützt werden, weist in den meisten anderen Bereichen jede der Bibliotheken andere Features, verschiedene Schwerpunkte sowie andere Vorgehensweisen auf. Daher ist es nicht möglich, die Bibliotheken im Bereich der Löser von Linearen Systemen unter exakt identischen Voraussetzungen zu prüfen. Stattdessen wurde Wert darauf gelegt, auf Grundlage der zur Verfügung stehenden Hard- und Software die Bibliothek möglichst so zu verwenden, wie sie sich optimal verhalten sollte.

Alle Bibliotheken wurden mit `double`-Genauigkeit ausgeführt. Die C- und C++-Bibliotheken wurden mit höchster Optimierung (`-O3`) kompiliert. Zudem wurde, soweit vorhanden und dokumentiert, bei der Installation beziehungsweise Konfiguration der Bibliotheken entsprechende Parameter übergeben, die einen möglichst performanten Livebetrieb (mit eingeschränkten Debugging-Möglichkeiten) bewirken.

5.1.1 Eigen

Für die Bibliothek Eigen wurde ein direkter Linearer Löser mit LU-Faktorisierung und vollständiger Pivotisierung für sequentielle Berechnungen sowie mit der hauseigenen SparseLU-Zerlegung für parallele Berechnungen verwendet. Das GMRES-Verfahren wurde anhand eines *unsupported*, aber in der Standardinstallation enthaltenen Zusatzmoduls realisiert. Eine explizite Faktorisierung vor dem GMRES-Verfahren sieht diese Implementierung nicht vor.

Der Code wurde derart kompiliert, dass eine Parallelisierung durch Setzen der entsprechenden Umgebungsvariable grundsätzlich möglich ist, auch wenn die meisten Operationen von Eigen keine parallele Ausführung unterstützen.

Eigen ist von keinen Backends (außer ggf. OpenMP) abhängig.

5.1.2 PETSc

Der Benchmark für PETSc wurde auf Grundlage bestehender offizieller Beispiele zur Lösung von allgemeinen sowie tridiagonalen Linearen Systemen implementiert. Dabei kann durch Kommandozeilenparameter zur Laufzeit die Art der Faktorisierung und des KSP-Verfahrens gewählt werden. Für das sequentielle direkte Lösen wurde die enthaltene LU-Zerlegung verwendet und auf ein KSP-Verfahren entsprechend verzichtet. Die parallele Ausführung dieser Art benötigte einen Vorkonditionierer, welcher auf parallele Ausführung ausgelegt ist und zugleich für ein direktes Lösen ohne KSP-Verfahren gedacht ist. Da derartige Faktorisierungen nicht in der Standardinstallation aufrufbar sind, wurde das externe, aber offiziell unterstützte Modul `SuperLU` hinzugefügt, das eine parallelisierbare LU-Zerlegung zur Verfügung stellt. Für die Ausführung von GMRES wurde im Sequentiellen eine LU-Zerlegung und im Parallelen eine Jacobi-Zerlegung verwendet.

PETSc wurde mit den gängigen Backends `BLAS`, `LAPACK` sowie `OpenMP` ausgestattet.

5.1.3 Armadillo

Für Armadillo wurde für dichte Matrizen ein direkter Löser mit vorangegangener LU-Zerlegung angesprochen. Für dünne Matrizen wird mithilfe des dazuinstallierten Backends `SuperLU` eine parallelisierbare LU-Zerlegung ausgeführt und damit das dünne Lineare System gelöst. Parallelisierung wurde mithilfe einer BLAS-Umgebungsvariable ermöglicht. Eine GMRES-Implementierung liegt bei Armadillo nicht vor.

Es wurden die Backends BLAS und SuperLU verwendet.

5.1.4 ViennaCL

ViennaCL bietet eine LU-Faktorisierung für dichte Matrizen an, welche entsprechend verwendet wurde. Für dünn besetzte Matrizen existiert ein solches Verfahren jedoch nicht, weshalb hier ebenfalls die Variante für dichte Matrizen verwendet wurde. Dieser Nachteil ist in den Ergebnissen des Performancevergleichs dringend zu berücksichtigen. Das GMRES-Verfahren folgt auf eine Unvollständige LU-Zerlegung (ILU).

Der Benchmark von ViennaCL gestaltete sich als etwas problematisch. Die Bibliothek ist durch Nutzung von OpenCL und/oder CUDA ausgelegt auf schnelle parallele Berechnungen mithilfe der GPU, welche für den Performancevergleich jedoch nicht zur Verfügung stand. Stattdessen wurde der Bibliothek das – laut Dokumentation ebenfalls unterstützte – OpenMP-Backend für parallele Ausführung zur Verfügung gestellt.

5.1.5 Numpy/Scipy

Für dichte Matrizen wurde die Numpy-Datenstruktur `ndarray` verwendet, während dünne Arrays durch Scipys `csr_matrix`-Strukturen repräsentiert wurden. Für dünne, dichte und Bandmatrizen wurden jeweils die dafür vorgesehenen Solver aus den Packages `scipy.linalg` und `scipy.sparse.linalg` verwendet; für dichte und dünne Matrizen jeweils mit einer passenden LU-Zerlegung. Das GMRES-Verfahren wurde ebenfalls mithilfe von `scipy.sparse.linalg`, ohne explizite Vorkonditionierung im Anwendungscode, durchgeführt.

Ein Aufruf der Numpy- und Scipy-Konfiguration ergab, dass BLAS und LAPACK von den Bibliotheken erkannt wurden und entsprechend genutzt werden. Eine Parallelisierung ist auf expliziter Ebene schwierig durchzuführen, daher wurde dieser Benchmark unter sequentieller und paralleler Berechnung gleichartig ausgeführt.

5.2 Ausführungsumgebung

Die Ausführungsumgebung ist ein Großrechner mit 144 CPU-Kernen (Intel Xeon(R) CPU E7-8880 v3 mit jeweils 2,3GHz) und einem 64-Bit Ubuntu 14.04-Betriebssystem mit Linux-Kernel 3.19. Die soeben beschriebenen Backends waren verfügbar beziehungsweise wurden nachinstalliert.

5.3 Durchführung des Benchmarks

Die Informationen zur Art und Weise der einzelnen Berechnungen werden in Form einer tabellarischen Textdatei übergeben, die die folgenden Informationen enthält:

- ID der Berechnung (aufsteigende Ganzzahl) für spätere Referenzierung
- Anzahl der gleichartigen zu generierenden Matrizen
- Dimension der Matrix
- Typ der Matrix (dünn, dicht oder tridiagonal)
- Berechnungsart (Direktes Lösen oder GMRES-Berechnung)
- Anzahl der zu verwendenden Kerne (1 = sequentiell)

Es wurden für jede Zeile dieser Datei mithilfe eines Python-Skripts zufällige (aber dabei der vorausgesetzten Konditionierung entsprechende) Matrizen erzeugt und in Dateien gespeichert.

Ein zentrales Bash-Skript automatisiert die Ausführung aller Programme unter den vorgesehenen Eingaben. Dazu wird für jede Zeile der Informationsdatei jede der Test-Programme mit

geeigneten Parametern angesprochen, insbesondere mit dem Pfad zu den entsprechenden Matrixdateien, die zuvor erzeugt worden sind. Die Ausgabe der Programme wird in entsprechende Logs umgeleitet.

Jedes Testprogramm ist dafür verantwortlich, die Laufzeit (nur im Intervall der eigentlichen Berechnung) zu messen und in die Standardausgabe zu schreiben. Nach abgeschlossenem Benchmark können diese Laufzeiten aus den einzelnen Logs extrahiert und automatisiert unmittelbar in eine einzelne tabellarische Textdatei geschrieben werden. Von dort stehen sie für die Auswertung zu Verfügung.

6 Ergebnisse

6.1 Praktischer Vergleich

Durch den durchgeführten Performancevergleich konnten Erkenntnisse über die Installation und Nutzung der behandelten Bibliotheken erworben und Kenngrößen für die Performance bestimmter Berechnungen ermittelt werden.

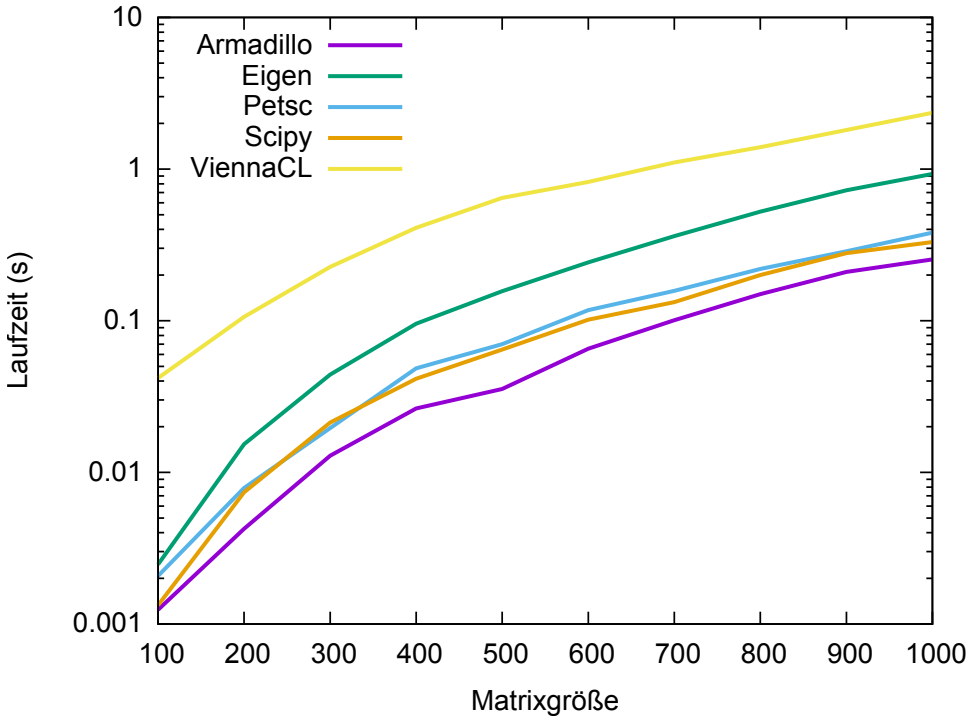

Abbildung 6.1: Direktvergleich der Bibliotheken beim sequentiellen Lösen eines dichten Linearen Systems

Abbildung 6.1 zeigt die Performance der Bibliotheken im Direktvergleich bei der sequentiellen Berechnung eines dicht besetzten Linearen Systems. Armadillo zeigt sich für diese Berechnungen am schnellsten, darauf folgen Scipy und PETSc. Eigen berechnet kleine Problemeingaben praktisch ebenso schnell wie PETSc, skaliert dann jedoch schlechter. ViennaCL ist in diesem Bereich abgeschlagen und benötigt deutlich länger für die Berechnung als die restlichen Bibliotheken.

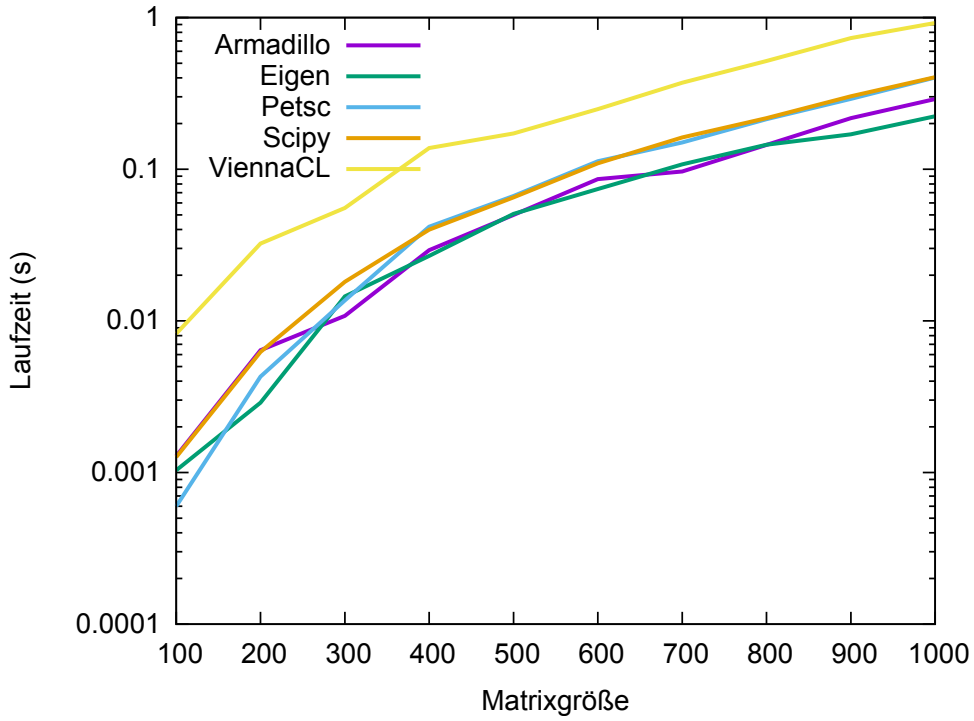

Abbildung 6.2: Direktvergleich der Bibliotheken beim sequentiellen Lösen eines dünnen Linearen Systems

In Abbildung 6.2 sind die Laufzeiten für das sequentielle direkte Lösen eines dünnen Linearen Systems dargestellt. Eigen schneidet hier gegenüber der Berechnung auf dichten Matrizen deutlich besser ab und kann sich für die größten untersuchten Eingaben gegenüber sämtlichen anderen Bibliotheken durchsetzen. Zu ViennaCL ist anzumerken, dass mangels einer LU-Zerlegung für dünn besetzte Matrizen erneut die Zerlegung dichter Matrizen verwendet wurde, was zumindest zu Teilen zur verschlechterten Laufzeit geführt haben wird.

Den Direktvergleich zwischen sequentieller und paralleler Lösung von Gleichungssystemen bietet der Vergleich von Abbildung 6.3 mit Abbildung 6.2. ViennaCL profitiert stark von der Parallelisierung und kann mit seinen Routinen zur Lösung dünn besetzter Systeme sehr geringe Laufzeiten erzielen. Auch die Verbesserung von PETSc ist sichtbar, da sich die Laufzeitkurve von jener von Scipy für große Eingaben deutlich nach unten hin absetzt. Für Scipy hingegen konnte keine eindeutige Möglichkeit zur expliziten Parallelisierung verwendet werden, weshalb die Berechnungen hier schlechter skalieren.

Beim Umgang mit Bandmatrizen (Abb. 6.4) weist Scipy hervorragende Ergebnisse auf. Die Laufzeit ist vergleichsweise sehr gering und steigt mit größerer Eingabe nur sehr langsam an. PETSc, Armadillo und Eigen weisen ähnliche Ergebnisse auf, wobei insbesondere die Berechnung durch Eigen schlechter skaliert. Die Ergebnisse zur parallelen Ausführung derselben Berechnung sind sehr ähnlich, weshalb sie an dieser Stelle nicht abgebildet wurden.

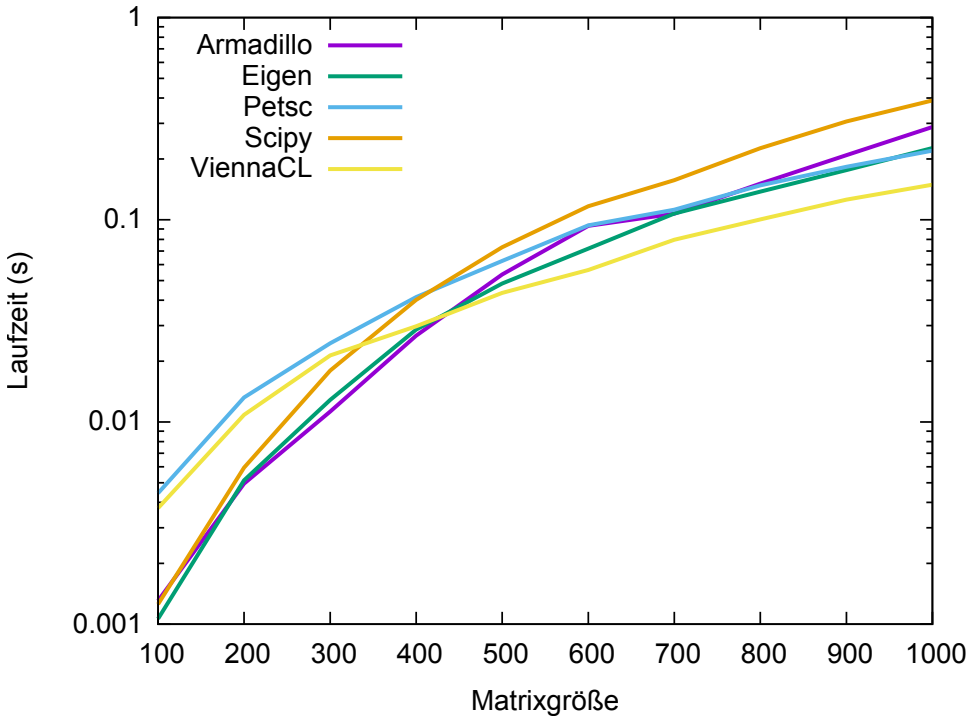

Abbildung 6.3: Direktvergleich der Bibliotheken beim parallelen Lösen eines dünnen Linearen Systems

Für das iterative GMRES-Lösungsverfahren haben sich die Bibliotheken sehr unterschiedlich verhalten. Sowohl im Sequentiellen (Abb. 6.5) als auch im Parallelen (Abb. 6.6) erreichen sowohl ViennaCL als auch Scipy schnell sehr hohe Laufzeiten, während Eigen und PETSc vergleichsweise gut skalieren. Insbesondere PETSc überzeugt hier durch sehr gute Performance und kann zudem durch Parallelisierung einen beträchtlichen zusätzlichen Performancegewinn verzeichnen.

Einzelne, nicht weiter ausgewertete Laufzeittests einzig für PETSc haben ergeben, dass bei Eingabe von Matrizen der Dimension 40 000 x 40 000 das Programm bei etwa 60 Kernen optimale Performance erzielt und das Problem in etwa 3 Sekunden gelöst wird. Eine höhere Anzahl von Kernen führt tendenziell zu schlechteren Ergebnissen, da die einzelnen Kerne dann nicht ausreichend ausgelastet sind. Alle anderen Bibliotheken können bei dieser Größenordnung nicht mehr in zufriedenstellender Zeit zu einem Ergebnis gelangen.

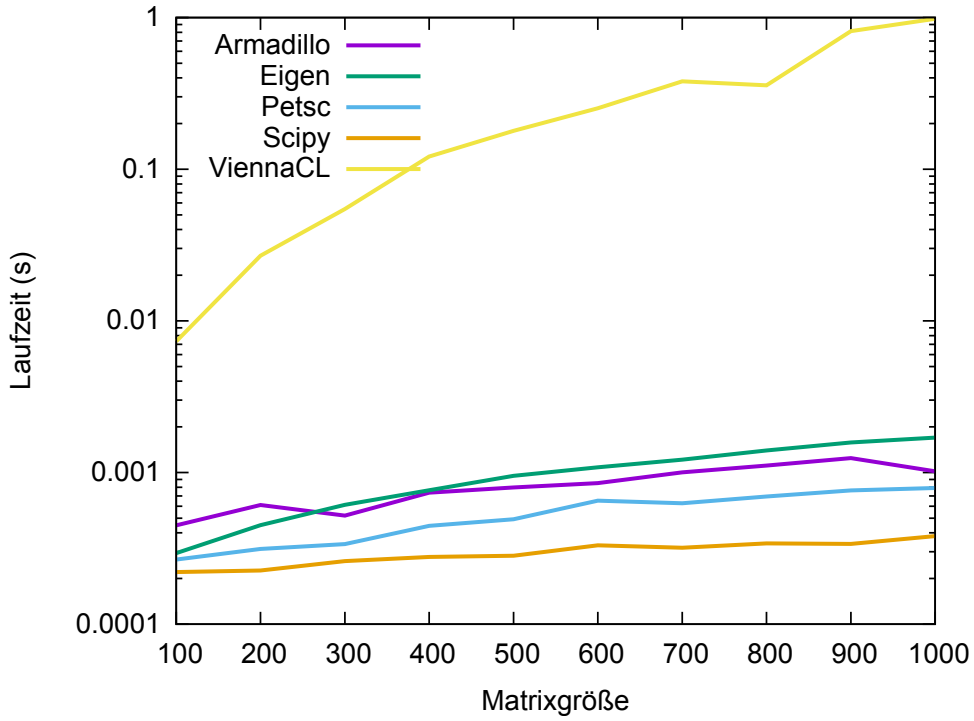

Abbildung 6.4: Direktvergleich der Bibliotheken beim sequentiellen Lösen einer Bandmatrix

Insgesamt zeigt der Performancevergleich ein gemischtes Bild, wodurch keine der Bibliotheken in jedem der untersuchten Anwendungsfälle die höchste Performance erzielt. Zu den wichtigsten Ergebnissen zählt die sehr gute Skalierung von PETSc und Eigen zum GMRES-Verfahren im Parallelen, eine beeindruckende Performance von Scipy im Bereich von Tridiagonalmatrizen sowie die Tatsache, dass sich die Bibliotheken im Bereich der direkten Lösungsverfahren deutlich weniger unterscheiden als bei Iterativer Lösung. ViennaCL ist in diesem Bereich auszuklammern, da aufgrund der suboptimalen Implementierung und der Optimierung für GPUs keine für die Bibliothek repräsentativen Ergebnisse erzielt werden konnten.

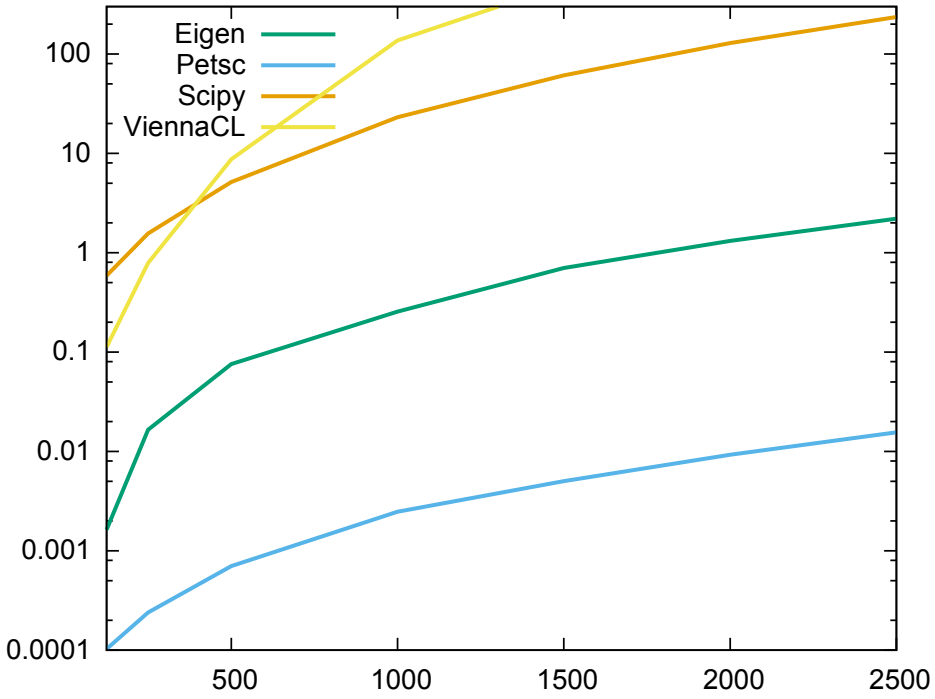

Abbildung 6.5: Direktvergleich der Bibliotheken beim sequentiellen GMRES-Verfahren

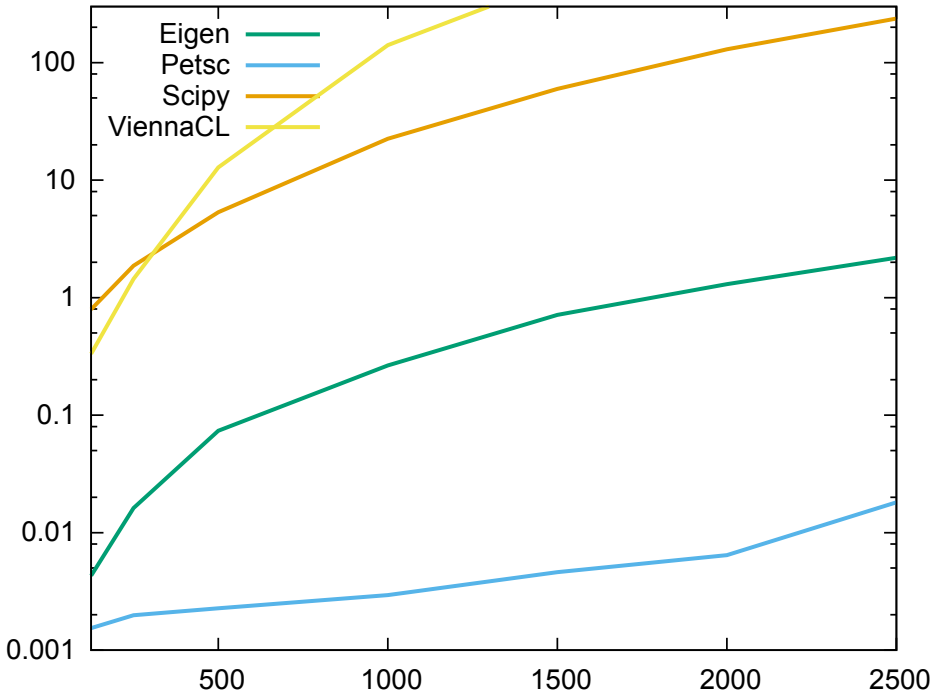

Abbildung 6.6: Direktvergleich der Bibliotheken beim parallelen GMRES-Verfahren

6.2 Erhebung von Community Metriken

Es wurde jeweils die Anzahl der explizit mit dem Tag der Bibliothek versehenen Fragen auf der Webseite `stackoverflow.com` ermittelt. Zusätzlich bietet die Anzahl der offenen sowie der gesamt vorhandenen Issues und Bugs in den entsprechenden Trackern Indikatoren für die Aktivität der Entwicklung und der Community. Da PETSc und Armadillo keine Online Bug Tracker verwenden, sondern stattdessen auf eMails direkt an die Entwickler setzen, sind hier keine Daten vorhanden.

Bibliothek	Stackoverflow-Fragen	Aktive Issues	Issues gesamt
Armadillo	555	N/A	N/A
Eigen	1225	330	1334
PETSc	63	N/A	N/A
Scipy	8055	790	3799
ViennaCL	12	40	161

Tabelle 6.1: Einige Metriken zur Community der Bibliotheken

Es zeigt sich, dass in der Tendenz besonders Scipy und Eigen in der breiten Masse beliebt sind und häufig genutzt werden.

6.3 Empfehlung nach Anwendungsfall

Für die vorliegenden Bibliotheken wurden durch inhaltliche Recherche, praktische Verwendung und Performancemessungen Vor- und Nachteile festgestellt. Im Folgenden werden aus diesen Erkenntnissen abschließende Empfehlungen für Bibliotheken im Hinblick auf verschiedene Anwendungsfälle ausgesprochen.

6.3.1 Anwendungsprogramme

Für die Entwicklung von nicht-wissenschaftlichen Endbenutzer-Anwendungen für Desktoprechner oder Mobilgeräte ist eine Bibliothek wünschenswert, die mit geringem Entwicklungsaufwand akzeptable Performance erzielt; parallele Berechnungen und eine gute Skalierbarkeit für sehr große Eingaben sind tendenziell nur sekundäre Anforderungen. Hingegen sind hohe Portabilität und eine gute Les- und Wartbarkeit des Codes zu den wichtigeren Kriterien zu zählen.

Die Bibliotheken Scipy sowie Eigen bieten für derartige Anwendungsfälle gut dokumentierte und intuitive Schnittstellen an und können viele verschiedene Problemstellungen einfach und effizient lösen. Durch ihre große Community ist die Entwicklung mit den Bibliotheken als

vergleichsweise einfach einzustufen, da online eine Vielzahl beantworteter Fragestellungen dazu einsehbar sind. Beide Bibliotheken können – auf verschiedene Arten – als sehr leichtgewichtig eingestuft werden; Eigen, da es sich um reine Header handelt und kein Backend benötigt wird, und Scipy, da sehr schnell kleinere Pythonskripte entwickelt werden können, die ohne zusätzlichen Kompilierschritt ausgeführt werden können. Abstriche sind bei diesen Bibliotheken hauptsächlich in der Parallelisierbarkeit der Berechnungen zu machen, denn diese konnte im Rahmen der Untersuchungen nicht hinreichend überzeugen.

6.3.2 High-Performance-Berechnungen

Groß angelegte Datenanalysen, komplexe Simulationsberechnungen und aufwendige Bildsynthesen benötigen exzellente Performance und die optimale Nutzung vorhandener Hardware, darunter nicht nur mehrere CPU-Kerne, sondern auch GPUs. Die Programmierschnittstelle darf hier im Allgemeinen von technischer und niedrig abstrahierter Natur sein, sofern dies zu besserer Performance führt.

Für diesen Anwendungsfall eignen sich mehrere Bibliotheken gut; während Eigen aufgrund geringer Parallelisierbarkeit hier nur bedingt empfehlenswert ist, können die Bibliotheken ViennaCL und Armadillo den Anforderungen vollständig entsprechen und, falls gewünscht, auch explizit miteinander synergieren. ViennaCL kann dabei durch die vorhandenen CUDA- und OpenCL-Schnittstellen optimalen Gebrauch von Grafikprozessoren machen. Armadillo bietet unter anderem die Einbindung der Boost-Bibliothek uBLAS an und enthält durch umsichtige Namensgebung von Routinen sowie durch viele explizite Schnittstellen eine hervorragende Kompatibilität hin zu anderen Bibliotheken wie Eigen oder PETSc, die möglicherweise an anderen Stellen des wissenschaftlichen Workflows verwendet werden.

Sofern ein entsprechend hoher Aufwand vertretbar ist, können für die konkrete Nutzung zur Lösung eines gegebenen Problems auf gegebener Hardware die genannten Bibliotheken unter verschiedenen Backend-Technologien erprobt und miteinander verglichen werden. Somit kann für aufwendige Berechnungen ein optimales Ergebnis erzielt werden, ob mit einer einzigen Bibliothek oder einer Kombination davon. Die vielen Konfigurations- und Optimierungsmöglichkeiten von ViennaCL und Armadillo ergeben damit für erfahrene und versierte Nutzer ein besonders hohes Potential für effziente Berechnungen.

Auch PETSc stellt eine gute Wahl für derartige Berechnungen dar. Dieses ist tendenziell im Hinblick auf Abhängigkeiten und Entwicklungsaufwand zunächst als schwergewichtiger einzustufen, besitzt jedoch für sich allein genomman die weitreichendsten Konfigurationsmöglichkeiten im Anwendungsbereich und eine exzellente Transparenz in der Ausführung.

6.3.3 Verteilte Berechnungen

Für ähnlich anspruchsvolle Berechnungen, die zudem auf mehreren, verteilten Rechenknoten ausgeführt werden sollen, eignet sich die vergleichsweise sehr schwergewichtige Bibliothek PETSc, die durch umfangreiche Unterstützung diverser Backends und eine hardwarenahe Programmiersprache besonders für große Eingaben eine unvergleichliche Performance erzielt. Im Bereich wissenschaftlicher Berechnungen, für welche nicht nur das Endergebnis, sondern auch Metadaten von Interesse sind, kann zudem die Unterstützung vieler verschiedener Kommandozeilenoptionen von großer Hilfe sein. Jedoch ist der Einarbeitungsaufwand größer als bei den anderen Bibliotheken und der Code ist weniger gut lesbar, insbesondere wenn der Leser Hochsprachen gewohnt ist.

6.4 Direkte Gegenüberstellung (Tabelle)

Die folgende Tabelle enthält stark verkürzt die Bewertung jeder untersuchten Bibliothek in Bezug auf die verschiedenen Kriterien. Eigenschaften, die nur als „vorhanden“ oder „nicht vorhanden“ eingestuft werden können, sind durch einen Haken (✓) oder ein Minus (-) markiert, während Kriterien, die in variablem Umfang erfüllt sein können, auf einer Skala {- -, -, 0, +, ++} verortet wurden.

Name	Eigen	PETSc	Armadillo	ViennaCL	SciPy / NumPy
Schlagwörter	Templates	MPI	Templates	GPU	scientific computing
Sprache	C++	C	C++	C++	Python
Backends					
AMD ACML			✓		
ARPACK			✓		
ATLAS			✓		✓
BLAS		✓	✓		✓
CUDA		✓	✓	✓	
Intel MKL			✓		✓
LAPACK		✓	✓		✓
NVBLAS			✓		
OpenBLAS			✓		✓
OpenCL				✓	✓
OpenMP		✓		✓	
SuperLU			✓		
Features					
Komplexe Zahlen	✓	✓	✓	✓	✓
Dünnbesetzte LA	✓	✓	✓	✓	✓
Faktorisierungen	++	++	+	++	++
Iterative Löser	✓[1]	✓	–	✓	✓
Im- und Export	-	+	+	+	+
Bedienbarkeit	+	-	++	+	++
Dokumentation	++	+	+	++	++
Parallelisierbarkeit	-	++	0	++	-
Erweiterbarkeit	++	++	-	-	++
Ausführungstransparenz	0	++	-	0	0
Bindings	R, Java	R, Java, Python	Python	R, Python	C, Fortran
Entwicklung					
Online-Repository	Bitbucket	Bitbucket	Sourceforge	Github	Github
Support	IRC, Mailingliste	Mailingliste	Mail an Entwickler	Mailingliste	Mailingliste
Bugtracking	Bugzilla	eMail	eMail	Github	Github

Tabelle 6.2: Direkte Gegenüberstellung der zentralen Eigenschaften der Bibliotheken

[1] Durch mitgeliefertes, aber *unsupported* Zusatzmodul

6.5 Fazit

Die Analyse der verschiedenen Bibliotheken hat eine Vielzahl von Erkenntnissen ergeben.

Zunächst ist festzustellen, dass das Problem der Wahl einer Bibliothek zur Linearen Algebra ein äußerst vieldimensionales ist, weshalb jede der untersuchten Bibliotheken zumindest in einigen Anwendungsfällen durch ihre jeweiligen Eigenheiten überzeugen kann.
Insbesondere im Bereich der Parallelisierbarkeit der Berechnungen ist dabei zwischen der Performanz und der Einfachheit der Entwicklung abzuwägen. Bei paralleler Ausführung Linearer Algebra handelt es sich um eine äußerst anspruchsvolle Aufgabe, die an verschiedenen Stellen scheitern kann, so etwa bei Restriktionen der Sprache oder fehlenden Schnittstellen zu hardwarenahen Algebrasystemen (wie beobachtet bei Eigen). Bibliotheken wie PETSc, die umfassende Parallelisierung aller Berechnungen anbieten, sind wiederum tendenziell schwieriger in der Anwendungsimplementierung.

Ein weiterer Aspekt der untersuchten Bibliotheken ist, dass es sich ausnahmslos um Freie Software handelt (ohne bei der Auswahl darauf Wert gelegt zu haben). Die Methodik der Open-Source-Programmierung verbunden mit der Gewähr der kommerziellen Nutzung und der freien Änderbarkeit haben sich in diesem Softwarebereich zu großen Teilen durchgesetzt. Anwender der Software können in Folge dessen von kostenfreier Nutzung und vollständiger Transparenz profitieren.

Letztendlich bleibt es als erfreulich zu verzeichnen, dass im Bereich der Linearen Algebra viele verschiedene Projekte existieren, welche mit jeweils individueller Fokussierung auf bestimmte Gebiete und Anwendungsbereiche optimale Lösungen für die entsprechenden Problemstellungen anbieten können. Dabei bleibt es dem Anwender überlassen, sich für eine der vielen in Frage kommenden Bibliotheken zu entscheiden, indem er die harten sowie die weichen Eigenschaften der Softwareprojekte nach eigenem Ermessen gewichtet und dementsprechend eine für ihn optimale Entscheidung trifft.

7 Zusammenfassung

In der vorliegenden Arbeit wurden anhand zuvor festgelegter Kriterien verschiedene bekannte Bibliotheken aus dem Bereich der Linearen Algebra untersucht und miteinander verglichen. Dazu wurden zunächst wichtige Eigenschaften derartiger Bibliotheken gesammelt, eine Vergleichsmethodik aufgestellt und die relevanten Vergleichskriterien genannt. Daraufhin wurden die Bibliotheken im Einzelnen analysiert sowie anhand praktischer Erfahrungen und einem Performancevergleich miteinander verglichen.

Die Bibliothek Eigen stellte sich als einsteigerfreundliche und dennoch sehr effektive Bibliothek für verschiedenste Zwecke heraus, die einzig im Bereich der Parallelisierung zu wünschen übrig lässt.

Das Framework PETSc wurde als schwergewichtige, aber äußerst mächtige Alternative insbesondere für hoch skalierte, wissenschaftliche (und potentiell verteilte) Anwendungen bewertet, bei welchen auch der Weg hin zur Lösung ein wichtiges Ergebnis darstellen kann.

Armadillo konnte als Bibliothek mit vielen nutzbaren Backends und sehr prägnanter Syntax überzeugen, mit der durch manuelle Optimierung sehr viel erreicht werden kann, jedoch das Debugging problematisch sein kann.

Die Bibliothek ViennaCL präsentierte sich als hochoptimiert auf GPU-Hardware, mithilfe derer hochperformante und parallele Berechnungen ermöglicht werden. Bemerkenswert sind die vielen Schnittstellen hin zu anderen Bibliotheken und eine sehr defensive Wahl der Syntax, die daher vielen bekannten Frameworks ähnelt und eine hohe Austauschbarkeit derer ermöglicht.

Die Python-Frameworks SciPy und Numpy konnten insbesondere durch ein hervorragendes Ökosystem wissenschaftlicher Frameworks und sehr einfache Bedienung überzeugen, während die Natur von Python als Skriptsprache nur bedingt eine Verwendung dieser Bibliotheken für High-Performance-Anwendungen ermöglicht.

Abschließend wurden für bestimmte Anwendungsfälle Empfehlungen für besonders gut geeignete Bibliotheken ausgesprochen, welche gemeinsam mit einem tabellarischen Direktvergleich das finale Ergebnis der Untersuchungen darstellen.

Literaturverzeichnis

[16a] *C Tutorial*. 2016. URL: https://www.tutorialspoint.com/cprogramming/index.htm (zitiert auf S. 20).

[16b] *C++ Tutorial*. 2016. URL: https://www.tutorialspoint.com/cplusplus/ (zitiert auf S. 20).

[16c] *Generalized Minimal Residual Method – from Wolfram MathWorld*. 2016. URL: http://mathworld.wolfram.com/GeneralizedMinimalResidualMethod.html (zitiert auf S. 31).

[16d] *Linear Algebra (scipy.linalg)*. 2016. URL: https://docs.scipy.org/doc/scipy-0.18.1/reference/tutorial/linalg.html (zitiert auf S. 56).

[16e] *The Python Tutorial*. 2016. URL: https://docs.python.org/3/tutorial/index.html (zitiert auf S. 21).

[16f] *tl;drLegal – Software Licences explained in Plain English*. 2016. URL: https://tldrlegal.com/ (zitiert auf S. 23).

[BAA+16a] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, L. C. McInnes, K. Rupp, B. Smith, S. Zampini, H. Zhang, H. Zhang. *PETSc Users Manual*. Techn. Ber. ANL-95/11 - Revision 3.7. Argonne National Laboratory, 2016. URL: http://www.mcs.anl.gov/petsc (zitiert auf S. 44).

[BAA+16b] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, L. C. McInnes, K. Rupp, B. Smith, S. Zampini, H. Zhang, H. Zhang. *PETSc Web page*. http://www.mcs.anl.gov/petsc. 2016. URL: http://www.mcs.anl.gov/petsc (zitiert auf S. 39).

[Bre12] E. Bressert. *SciPy and NumPy*. O'Reilly Media, 2012 (zitiert auf S. 56).

[Buc09] D. Buchholz Martin und Pflüger. *Modellbildung und Simulation*. Springer, 2009 (zitiert auf S. 25, 26).

[GJ+10] G. Guennebaud, B. Jacob et al. *Eigen v3*. http://eigen.tuxfamily.org. 2010 (zitiert auf S. 31, 33, 36).

[LL13] J. Ludewig, H. Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. dpunkt. verlag, 2013 (zitiert auf S. 17).

[Mut06] H. J. Muthsam. *Lineare Algebra und ihre Anwendungen*. Elsevier, Spektrum Akad. Verlag, 2006 (zitiert auf S. 25).

[REU] A. REUSKEN. *Numerik für Ingenieure und Naturwissenschaftler* (zitiert auf S. 19, 31).

[Rup14] K. Rupp. „ViennaCL 1.5.2 User Manual“. In: (2014) (zitiert auf S. 54).

[San10] C. Sanderson. „Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments“. In: (2010) (zitiert auf S. 46, 48, 50).

[SC] C. Sanderson, R. Curtin. *Armadillo: C++ linear algebra library*. URL: http://arma.sourceforge.net/ (zitiert auf S. 46).

[SC16] C. Sanderson, R. Curtin. „Armadillo: a template-based C++ library for linear algebra“. In: *Journal of Open Source Software* (2016) (zitiert auf S. 46, 47, 49, 51).

Alle URLs wurden zuletzt am 20. 10. 2016 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift