Institut für Parallele und Verteilte Systeme
Abteilung Simulation großer Systeme

Diplomarbeit Nr. 3650

Design moderner Numerikbibliotheken am Beispiel von SG++
Design of Modern Numerical Libraries on the Examle of SG++

Özcan Kara

Studiengang:	Softwaretechnik
Prüfer:	Jun.-Prof. Dr. rer. nat. Dirk Pflüger
Betreuer:	Dipl.-Inf. David Pfander
Beginn am:	14. April 2014
Beendet am:	14. Oktober 2014
CR-Nummer:	D.2.11

Kurzfassung

SG++ ist eine in C++ geschriebene Numerikbibliothek, die umfangreiche Anwendungsmöglichkeiten bietet. Basierend auf dünnen Gittern können mit *SG++* Probleme in der Finanzmathematik und im Data-Mining, als auch partielle Differenzialgleichungen gelöst werden. Dabei steht die Effizienz der Berechnungen im Vordergrund. Diese Arbeit befasst sich jedoch mit der effizienten Verwendung und der Softwarequalität der Bibliothek. Es wird definiert, was eine gute Programmierschnittstelle ausmacht und welche Techniken und Konzepte zur Umsetzung verwendet werden können. Anschließend wird ein Refactoring der Bibliothek *SG++* durchgeführt, damit es den aufgestellten Richtlinien entspricht und es wird erklärt, welche Vorteile die umgestaltete Bibliothek von Benutzersicht bietet. Laufzeittests verdeutlichen, welche Auswirkungen die Änderungen auf die Performance haben. Zusätzlich wird die Unterstützung weiterer Schnittstellen für andere Sprachen untersucht. Hier wird speziell auf Python und Java eingegangen und der zusätzliche Rechenaufwand gemessen, den die beiden Sprachen beim Aufrufen von C++-Funktionen benötigen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Unter Softwarequalität werden oft unterschiedliche Aspekte zusammengefasst. Eine Definition von Softwarequalität bieten Ludewig und Lichter [LL07] an. Sie unterscheiden zwischen der Qualität des Entwicklungsprozesses und der Qualität des Softwareprodukts. In dieser Diplomarbeit wird die Produktqualität einer Softwarebibliothek untersucht. Bei Bibliotheken ist eine gute Softwarequalität besonders wichtig, weil die Benutzer der Bibliothek selbst Programmierer sind. Eine Bibliothek mit schlechter Qualität kostet Benutzern unnötig viel Zeit beim Einlernen. Sie ist oft schwer verständlich, frustriert Programmierer und ist schlecht oder gar nicht erweiterbar. Um eine große Benutzerbasis zu gewinnen, sollte demnach auf die Produktqualität geachtet werden.

In dieser Diplomarbeit soll ein Refactoring einer Bibliothek durchgeführt werden. Das bedeutet, dass die Software umstrukturiert wird, um die Softwarequalität zu steigern, ohne das Programmverhalten zu ändern. Das Hauptaugenmerk liegt dabei in der Gestaltung der Programmierschnittstelle, wobei aber auch bibliotheksinterne Aspekte besprochen werden. Es werden Richtlinien definiert und beispielhaft deren Umsetzung gezeigt. Die besondere Herausforderung dabei ist, dass die zu untersuchende Bibliothek aus dem Bereich der Numerik stammt. In der Numerik gilt die Effizienz des Programms oft als der wichtigste Aspekt. Das ist auch leicht nachzuvollziehen, wenn beispielsweise große Probleme gelöst werden müssen und die Rechenzeit mehrere Stunden oder Tage beträgt. Wenn nun anstelle der Effizienz andere Softwarequalitätsmerkmale im Vordergrund stehen würden und somit das Programm zusätzliche Stunden und Tage benötigen würde, dann kostet es Benutzern nicht nur Zeit, sondern auch Geld. Aber die Effizienz mit weiteren Softwarequalitäten zu vereinen ist von großer Wichtigkeit, weil nicht nur die Rechenzeit Kosten verursacht, sondern auch die Entwicklungszeit der Benutzer.

Als Beispielbibliothek wird *SG++* untersucht. *SG++* ist eine umfangreiche Bibliothek mit vielen Anwendungsmöglichkeiten. Basierend auf dem Verfahren der dünnen Gitter, ermöglicht es unter anderem Data-Mining zu betreiben und partielle Differenzialgleichungen zu lösen. Diese Fülle von Möglichkeiten und unterschiedlichen Szenarien erschwert den Designprozess der Bibliothek zusätzlich, damit nicht nur effizientere Berechnungen, sondern auch effizientere Verwendungen der Programmierschnittstelle ermöglicht werden.

Für Numeriksoftware kommt häufig Fortran oder C zum Einsatz, wie es beispielsweise bei *LAPACK* oder *ATLAS* der Fall ist. *SG++* ist jedoch in C++ implementiert und verwendet objektorientierte Konzepte. Dieser Ansatz verlangt beim Design der Bibliothek eine andere Vorgehensweise als Bibliotheken, die das imperative Programmierparadigma nutzen. In dieser Diplomarbeit werden einige Konzepte der objektorientierten Programmierung beschrieben und es wird gezeigt, wie sie in *SG++* angewendet werden.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Das Verfahren der dünnen Gitter: Zuerst soll das grundlegende mathematische Verfahren der dünnen Gitter eingeführt werden, auf dem die Bibliothek aufbaut. Am Beispiel der Interpolation wird zunächst das Verfahren auf vollen Gittern erklärt. Danach werden die dünnen Gitter eingeführt und die Vorteile dieses Ansatzes aufgezeigt.

Kapitel 3 – Die Software SG++: Dieses Kapitel gibt einen kurzen Überblick über die Software SG++. Hier wird auf den Aufbau der Bibliothek eingegangen, als auch auf die verwendeten Werkzeuge.

Kapitel 4 – Bibliotheksdesign: In diesem Kapitel werden allgemeine Designrichtlinien für die Gestaltung einer Programmierschnittstelle aufgestellt. Diese Richtlinien sind speziell angepasst auf Bibliotheken im Bereich der Numerik. Weiterhin wird erläutert, welche Techniken in *SG++* eingesetzt wurden, um die Bibliothek richtlinienkonform zu gestalten. Zusätzlich wird analysiert, welche Vorteile die Unterstützung mehrerer Programmiersprachen hat, wobei vor allem Java und Python näher betrachtet werden. Weil in der Numerik die Effizienz eine große Rolle spielt, werden auch Laufzeittests durchgeführt. Mit den Tests soll gezeigt werden, ob es sich für Benutzer leistungstechnisch lohnt die Java- und Python-Schnittstellen zu verwenden.

Kapitel 5 – Refactoring der Bibliothek: Die in Kapitel 4 eingeführten Richtlinien werden hier am Beispiel von *SG++* umgesetzt. Es wird gezeigt, wie die vorgestellten Techniken verwendet wurden, damit die Programmierschnittstelle vereinfacht wird. Auch einige bibliotheksinterne Änderungen werden angesprochen, um eine verständlichere Softwarearchitektur zu erhalten und die Wartbarkeit der Bibliothek zu verbessern.

Kapitel 6 – Vergleich der alten mit der neuen SG++-Version: Die Ergebnisse der Bibliotheksumgestaltungen werden in diesem Kapitel erläutert. Als Beispiel dient ein einfaches Interpolationsbeispiel. Das Beispiel wurde einmal mit der unveränderten Schnittstelle von *SG++* implementiert und noch einmal mit der geänderten. In dieser Arbeit werden die beiden Versionen als alte und neue *SG++*-Version bezeichnet. Im Anschluss folgen dann weitere Leistungsvergleiche. Diesmal werden die neuen mit den alten Schnittstellen verglichen, damit verdeutlicht wird, welche Auswirkungen die Änderungen auf die Effizienz der Bibliothek haben.

Kapitel 7 – Zusammenfassung, Fazit und Ausblick Zum Schluss werden die Ergebnisse der Arbeit zusammengefasst und einige Anknüpfungspunkte vorgestellt.

2. Das Verfahren der dünnen Gitter

Ob beim Data-Mining oder beim Lösen von partiellen Differenzialgleichungen, in der Numerik spielen Gitter zur Diskretisierung eines Gebiets eine wichtige Rolle. Volle Gitter sind oft aber nur bei niedrigen Dimensionen praktisch einsetzbar. Bei hohen Dimensionen schlägt der Fluch der Dimensionalität zu, was bedeutet, dass die Anzahl der Gitterpunkte exponentiell mit der Dimensionalität wächst und damit auch die Anzahl der benötigten Funktionsauswertungen.

Mit dünnen Gittern wird versucht, diesen Fluch der Dimensionalität zu bewältigen oder zumindest abzuschwächen. Dünne Gitter verwenden dafür weit weniger Gitterpunkte während die asymptotische Fehlerentwicklung eines vollen Gitters bis auf einen logarithmischen Faktor erhalten bleibt.

Diese Arbeit soll nur eine grobe Übersicht über das Verfahren der dünnen Gitter geben, weil hier der Fokus auf Softwareengineering in der Numerik liegt. Detailliertere Ausführungen sind unter anderem bei Garcke [Gar07] zu finden, wobei die Beschreibung in dieser Diplomarbeit größtenteils der Struktur und Notation von Pflüger [Pfl10] folgt.

2.1. Interpolation mit vollen Gittern

Zuerst werden volle Gitter vorgestellt, um im darauf folgenden Kapitel die dünnen Gitter besser nachvollziehen zu können. Als Anwendungsbeispiel wird die lineare Interpolation einer Funktion $f : \Omega \to \mathbb{R}$ auf dem Gebiet Ω betrachtet. Die Funktionswerte auf dem Rand $\partial\Omega$ sollen dabei auf null beschränkt sein.

2.1.1. Eindimensionaler Fall

Im eindimensionalen Fall ist das Gebiet, auf dem die zu interpolierende Funktion f definiert ist, beschränkt auf das Intervall $\Omega = [0, 1]$. Dieses Intervall wird diskretisiert mit einem vollen regulären Gitter mit der Maschenweite $h_n = 2^{-n}$ auf der Verfeinerungsstufe n. Die dadurch entstandenen äquidistanten Gitterpunkte x_i werden mit der Funktion f ausgewertet und interpoliert.

Dazu sei V_n der Raum der stückweise linearen Funktionen auf Verfeinerungsstufe n. Durch die geeignete Wahl von linearen Basisfunktionen $\varphi(x)$ aus V_n erhält man die Interpolationsfunktion $u(x)$ durch die gewichtete Summe der Basisfunktionen, also durch

$$u(x) \coloneqq \sum_{i}^{N} \alpha_i \varphi_i(x), \tag{2.1}$$

mit den Koeffizienten $\alpha_i \in \mathbb{R}$ und N Gitterpunkten. Es gilt $f(x) \approx u(x)$, womit dann das Interpolationsproblem gelöst werden kann. Als Basisfunktionen kommen üblicherweise Hutfunktionen

$$\varphi(x) = \max(0, 1 - |x|) \tag{2.2}$$

zum Einsatz, die auf den Gitterpunkten zentriert und entsprechend skaliert sind. Die Koeffizienten erhält man dann durch die Auswertung $\alpha_i = f(x_i)$ der zu interpolierenden Funktion, an der die Hutfunktionen eben eins sind.

2.1.2. Multidimensionaler Fall

Die Verallgemeinerung in mehrere Dimensionen ist bei vollen Gittern relativ einfach. Das zu betrachtende Gebiet sei nun der d-dimensionale Einheitswürfel $\Omega = [0, 1]^d$. Als Maschenweite sei wieder $h_n = 2^{-n}$ in jede Dimension. Die Basisfunktionen werden durch die Tensorproduktkonstruktion in d Dimensionen erweitert durch

$$\varphi_{\vec{l},\vec{i}}(\vec{x}) \coloneqq \prod_{j=1}^{d} \varphi_{l_j,i_j}(x_j) \tag{2.3}$$

mit d-dimensionalen Multiindizes $\vec{l} \in \mathbb{N}^d$ für ein vorgegebenes Level und $\vec{i} \in \mathbb{N}^d$ für den Index an einem Punkt in jede Dimension. Zum Indizieren der Gitterpunkte ist die Menge

$$I_{\vec{l}} \coloneqq \{\vec{i} \in \mathbb{N}^d : 1 \leq i_j \leq 2^{l_j} - 1, i_j \text{ ist ungerade und } 1 \leq j \leq d\} \tag{2.4}$$

definiert. Mit dem Unterraum der Basisfunktionen

$$W_{\vec{l}} \coloneqq \operatorname{span}\{\varphi_{\vec{l},\vec{i}}(\vec{x}) : \vec{i} \in I_{\vec{l}}\} \tag{2.5}$$

für die jeweiligen Level $\vec{l}$, lässt sich der Raum der d-linearen Funktionen V_n mit der Definition der Maximumsnorm für Multiindizes

$$|\vec{l}|_\infty \coloneqq \max_{1 \leq j \leq d} |l_j| \tag{2.6}$$

und der direkten Summe konstruieren

$$V_n \coloneqq \bigoplus_{|\vec{l}|_\infty \leq n} W_{\vec{l}}. \tag{2.7}$$

Ganz analog zum eindimensionalen Fall kann die Interpolationsfunktion $u(\vec{x}) \in V_n$ aufgestellt werden durch

$$f(\vec{x}) \approx u(\vec{x}) \coloneqq \sum_{|\vec{l}|_\infty \leq n, \vec{i} \in I_{\vec{l}}} \alpha_{\vec{l},\vec{i}} \varphi_{\vec{l},\vec{i}}(\vec{x}), \tag{2.8}$$

mit den Koeffizienten $\alpha_i \in \mathbb{R}$ aus den Funktionsauswertungen.

Mit $N = 2^n - 1$ Gitterpunkten aus dem eindimensionalen Fall, erhält man durch diese Konstruktion im d-dimensinalen Fall $N^d = (2^n - 1)^d$ viele Knotenpunkte. Das führt zu $\mathcal{O}(2^{nd})$ Funktionsauswertungen, was bei hohen Dimensionen rechenenintensiv ist. Der Fluch der Dimensionen schlägt bei diesem Ansatz also voll zu.

2.2. Interpolation mit dünnen Gittern

Um Probleme in höheren Dimensionen lösen zu können, verwenden die dünnen Gitter einen anderen Funktionsraum als Basis. Gesucht wird eine Basis, welche die zu interpolierende Funktion mit so wenig Basisfunktionen wie möglich darstellt und somit die Anzahl der Funktionsauswertungen gering hält. Dünne Gitter basieren auf einer hierarchischen Basis und einer geeigneten hierarchischen Unterraumzerlegung.

2.2.1. Hierarchische Basis in einer Dimension

Wieder sei das Gebiet Ω beschränkt auf das Intervall $\Omega = [0, 1]$. Aus der Hutfunktion 2.2 lässt sich durch Verschiebung und Streckung die Basisfunktionen definieren

$$\varphi_{l,i}(x) \coloneqq \phi(2^l x - i), \tag{2.9}$$

welche vom Level l und dem Index $0 < i < 2^l$ abhängig sind. Die lokalen Träger der Basisfunktionen sollen somit auf den jeweiligen Gitterpunkten $x_{l,i}$, auf denen Interpoliert werden soll, zentriert werden. Die zu den Gitterpunkten zugehörige Indexmenge ist

$$I_l \coloneqq \{i \in \mathbb{N} : 1 \leq i \leq 2^l - 1, i \text{ ist ungerade}\}. \tag{2.10}$$

Mit Hilfe dieser Indexmenge lassen sich die hierarchischen Unterräume

$$W_l \coloneqq \operatorname{span}\{\varphi_{l,i}(x) : i \in I_l\} \tag{2.11}$$

durch das direkte Produkt definieren sowie der Raum der stückweise linearen Funktionen

$$V_n \coloneqq \bigoplus_{l \leq n} W_l. \tag{2.12}$$

Anders als bei der nodalen Basis überlappen die Träger der Funktionen in der hierarchischen Basis nicht auf dem gleichen Level. Trotzdem überdecken alle Basisfunktionen zusammen das gesamte Intervall Ω.

Wie üblich kann die Interpolationsfunktion $u(x) \in V_n$ als lineare Kombination geschrieben werden

$$u(x) \coloneqq \sum_{l \leq n, i \in I_l} \alpha_{l,i} \varphi_{l,i}(x), \tag{2.13}$$

dabei heißen die Koeffizienten $\alpha_{l,i}$ hierarchische Überschüsse.

Im eindimensionalen Fall ist der Dünngitterraum noch identisch mit dem Vollgitterraum. In höheren Dimensionen wird der Unterschied der beiden Ansätze jedoch deutlich.

2.2.2. Dünne Gitter in mehreren Dimensionen

Für den Schritt auf ein mehrdimensionales Gebiet $\Omega = [0,1]^d$ werden die Basisfunktionen $\varphi_{l,i}(x)$ durch ein Tensorprodukt verallgemeinert

$$\varphi_{\vec{l},\vec{i}}(\vec{x}) \coloneqq \prod_{j=1}^{d} \varphi_{l,i}(x_j), \tag{2.14}$$

wobei die Träger dieser Funktionen wieder zentriert sind auf die entsprechenden Gitterpunkte. Zusammen mit der Indexmenge

$$I_{\vec{l}} \coloneqq \{\vec{i} \in \mathbb{N}^d : 1 \leq i_j \leq 2^{l_j} - 1, i_j \text{ ist ungerade und } 1 \leq j \leq d\} \tag{2.15}$$

lässt sich der hierarchische Unterraum

$$W_{\vec{l}} \coloneqq \operatorname{span}\{\varphi_{\vec{l},\vec{i}} : \vec{i} \in I_{\vec{l}}\} \tag{2.16}$$

definieren. Dünne Gitter erhält man nun, wenn Basisfunktionen weggelassen werden, die nur wenig zur Lösung beitragen. Das kann a priori geschehen, wenn die gemischten zweiten Ableitungen der zu interpolierenden Funktion beschränkt sind. Der Raum der dünnen Gitter erhält man mit

$$V_n \coloneqq \bigoplus_{|\vec{l}|_1 \leq n+d-1} W_{\vec{l}}, \tag{2.17}$$

wobei die Summennorm für Multiindizes definiert ist als

$$|\vec{l}|_1 \coloneqq \sum_{j=1}^{d} l_j. \tag{2.18}$$

Die Interpolationsfunktion $u(\vec{x}) \in V_n$ ergibt sich dann aus

$$u(\vec{x}) = \sum_{|\vec{l}|_1 \leq n+d-1, \vec{i} \in I_{\vec{l}}} \alpha_{\vec{l},\vec{i}} \varphi_{\vec{l},\vec{i}}(\vec{x}). \tag{2.19}$$

In der L^2- und der Maximumsnorm reduziert sich die Anzahl der Gitterpunkte von $\mathcal{O}(h_n^{-d})$ bei einem vollen Gitter, zu $\mathcal{O}(h_n^{-1}(\log h_n^{-1})^{d-1})$. Die Genauigkeit wird aber nur um einen Faktor $(\log h_n^{-1})^{d-1}$ schlechter [BG04].

3. Die Software SG++

3.1. Beschreibung und Aufbau

Das im vorherigen Kapitel beschriebene Verfahren bietet viele Anwendungsmöglichkeiten. *SG++*, was für dünne Gitter im Englischen (Sparse Grids) steht, ist eine in C++ geschriebene Softwarebibliothek. Sie bietet verschiedene Verfahren auf dünnen Gittern an und implementiert viele Anwendungsmöglichkeiten. *SG++* ist modular aufgebaut und trennt die Anwendungsmöglichkeiten auf mehrere Module auf. Das Diagramm 3.1 zeigt eine Übersicht über die Module und deren Abhängigkeiten.

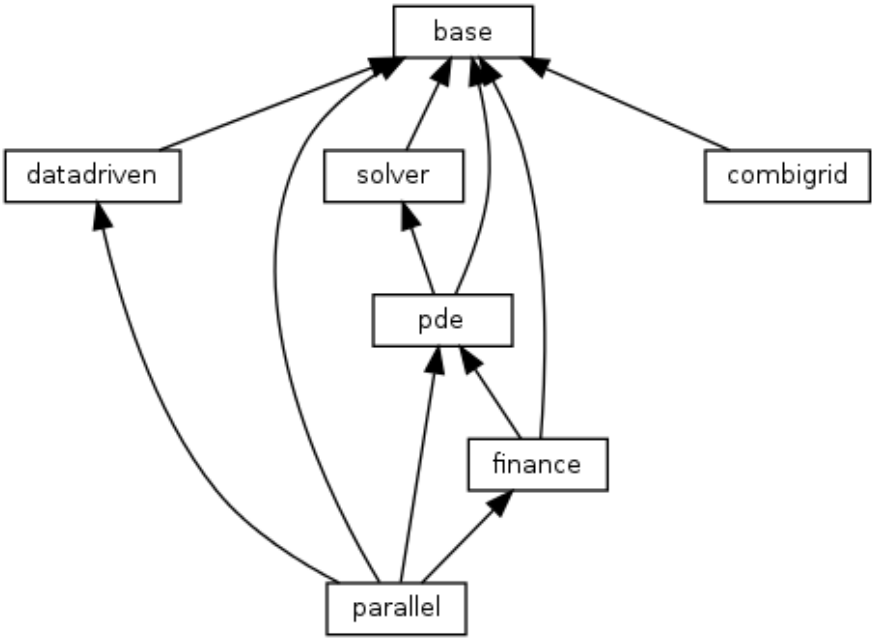

Abbildung 3.1.: Module und Abhängigkeiten - Graph aus der *SG++*-Doxygen-Dokumentation

3.1.1. Beschreibung der Module

Base ist das wichtigste Modul und bildet die Basis für alle weiteren Module. In diesem Modul sind die Implementierungen von dünnen Gittern mit verschiedenen Basisfunktionen zu finden. Ebenso enthält es grundlegende Funktionalitäten zum Hierarchisieren und Dehierarchisieren, also zum Umrechnen der nodalen in die hierarchische Basis und umgekehrt, sowie zum adaptiven Verfeinern eines Gitters und auch Operationen zum Auswerten einer Funktion auf einem Gitter. Mit diesem Modul ist es ebenso möglich einfache Interpolations- und Quadraturaufgaben zu lösen. Auch Monte-Carlo-Integrationen sind mit diesem Modul durchführbar.

Solver ist ein weiteres grundlegendes Modul. Es beherbergt zwei verschiedene Arten von Lösern. Da gibt es zum einen die Löser für lineare Gleichungssysteme. *SG++* bietet die Auswahl zwischen dem Verfahren der konjugierten Gradienten und dem BiCGStab-Verfahren. Dann gibt es noch die Löser für Differenzialgleichungen. Implementierungen für das implizite und explizite Euler-Verfahren, das Crank-Nicolson-Verfahren und die Adams-Bashforth-Methode sind in diesem Modul zu finden.

Datadriven stellt Funktionalitäten fürs Data-Mining bereit. Mithilfe dieses Moduls können Regressionsanalysen durchgeführt und auch Klassifikationsprobleme gelöst werden.

PDE wird zum Lösen von parabolischen und elliptischen partiellen Differenzialgleichungen mit Neumann- und Dirichlet-Randbedingungen verwendet. Um die Benutzung zu vereinfachen, stellt es auch einfache Anwendungen bereit, wie zum Beispiel die Wärmeleitungsgleichung.

Finance befasst sich mit den Anwendungen der dünnen Gitter auf die Finanzmathematik. Damit lassen sich die Gleichungen des Black-Scholes-Modells, des Hull-White-Modells und auch des Heston-Modells lösen.

3.2. Eingesetzte Werkzeuge

In der Softwareentwicklung wird es wichtig erachtet, geeignete Werkzeuge zu wählen. Mit guten Werkzeugen kann die Qualität der zu schreibenden Software verbessert und die Produktivität der Entwickler gesteigert werden. *SG++* verwendet dazu einige freie Werkzeuge, die hier kurz vorgestellt werden.

3.2.1. SWIG

SWIG steht für Simplified Wrapper and Interface Generator. In *SG++* wird es verwendet um Schnittstellen zu generieren, sodass Benutzer aus Java und Python die *SG++*-Bibliothek benutzen können.

Die Verwendung von *SWIG* ist relativ simpel. Jede Klasse und Funktion, für die eine Schnittstelle zu anderen Sprachen generiert werden soll, muss in einer Interfacedatei spezifiziert werden. *SWIG* erstellt aus diesen Interfacedateien dann entsprechenden Wrappercode, welcher dann noch kompiliert und mit der Bibliothek gelinkt wird.

SWIG bietet somit eine komfortable Möglichkeit, Schnittstellen für viele verschiedene Sprachen automatisch zu generieren. Welche Sprachen unterstützt werden und welche weiteren Features *SWIG* noch bietet, kann der Dokumentation entnommen werden [SWI].

3.2.2. SCons

Zum Bauen von *SG++*-Bibliothek wird *SCons* (Software Construction Tool) eingesetzt. Es ist ein in Python geschriebenes Werkzeug, welches gegenüber *Make* und anderen Buildsystemen einige Vorteile bietet [SCob]. Der größte Vorteil für *SG++* ist jedoch, dass es standardmäßig C++ und *SWIG* unterstützt. Mehr Informationen über *SCons* gibt es auf der Webseite [SCoa].

3.2.3. Doxygen

Doxygen ist ein Softwaredokumentationswerkzeug mit dem es möglich ist automatisch aus dem Quellcode eine Dokumentation erstellen. Es können damit *HTML*, *PDF* und weitere Formate generiert werden.

Mit *Doxygen* lässt sich ohne großen Aufwand der Quellcode dokumentieren. Implementierer müssen nur die Kommentare *Doxygen*-gerecht halten. Mit *Doxygen* kann die Dokumentation immer auf dem aktuellsten Stand gehalten werden, ohne zusätzliche Zeit in das Schreiben der Dokumentation investieren zu müssen.

Welche weiteren Features *Doxygen* bietet, kann auf der Website [Dox] nachgelesen werden.

4. Bibliotheksdesign

4.1. Vorteile objektorientierten Designs

Durch objektorientierte Programmierung (OOP) erhoffen sich Softwarearchitekten und Programmierer eine übersichtlichere Programmstruktur, flexiblere Architektur, erhöhte Produktivität bei der Entwicklung und bessere Wartbarkeit der Software vor allem bei größerer Software. Mit Hilfe von Konzepten wie Klassen, Datenkapselung, Polymorphie und Vererbung soll so die Software modular aufgebaut werden und somit die Erweiterbarkeit und Wiederverwendbarkeit von Code gefördert werden.

Ein Kritikpunkt, welcher gegen OOP gebracht wird, ist die geringere Effizienz gegenüber prozedurale Programmierung. Chatzigeorgiou und Stephanides [CS02] zeigten, dass OOP zu einer geringeren Effizienz führen kann. Um den Performanceverlust niedrig zu halten, entstand das Konzept der objektorientierten Numerik (OON) [ABL97]. OON sind OOP-Konzepte angewandt auf Anwendungen in der Numerik. OON will die Vorteile von OOP mit möglichst effizienten Berechnungen verbinden, indem beispielsweise rechenintensiver Code im imperativen Programmierparadigma geschrieben ist und dieser dann durch eine API im OOP-Stil umhüllt wird.

4.2. Allgemeine Designrichtlinien

Im Folgenden werden allgemeine Designrichtlinien zur Gestaltung von APIs für Numerikbibliotheken beschrieben. Die folgenden Richtlinien wurden aus verschiedenen Quellen für allgemeines API-Design ([Bla08], [Blo06]), objektorientierter Programmierung ([Mey97]) und Programmdesign für spezifische Programmiersprachen ([Lan09], [Str13], [Red11], [Tul08]) zusammengefasst und speziell für Numerikbibliotheken angepasst.

4.2.1. Benutzerfreundlichkeit

Benutzerfreundlichkeit bedeutet, dass sich das Werkzeug den Benutzern anpasst, um so die Erlernbarkeit zu steigern und die Arbeit mit der Software effizienter zu gestalten. Im Folgenden soll näher erläutert werden, welche Kriterien eine benutzerfreundliche API ausmachen.

Einfache Erlernbarkeit

Einfache Erlernbarkeit bedeutet, dass Benutzer wenig Aufwand betreiben müssen, um sich mit der API vertraut zu machen. Das heißt, dass die API intuitiv ist, sodass Benutzer nicht erst viele Seiten im Handbuch nachschlagen müssen, um die API zu benutzen. Dazu gehört auch eine gute Namensgebung und der logische Aufbau der Softwarearchitektur. Einfachere Designs sind dabei vorzuziehen, weil eine höhere Komplexität bedeutet, dass Benutzer mehr Zeit aufbringen müssen, um die API zu verstehen. Außerdem gilt, je weniger Methoden und Klassen es gibt, mit denen sich Benutzer vertraut machen müssen, desto früher können sie anfangen mit der API arbeiten.

Namensgebung

Es ist wichtig gute Namen für Klassen, Methoden, Parameter und Variablen zu wählen. Das schließt mit ein, dass Benutzer am Namen die Bedeutung der Klasse, Methode, Parameter oder Variable erkennen können, ohne weitere Dokumentation lesen zu müssen. Ein deskriptiver Name für Klassen und Methoden hilft Benutzern, eine gesuchte Funktionalität schneller zu finden, was die API besser erschließbar macht. Ein guter Name lässt sich auch einfacher merken, wodurch die Benutzbarkeit verbessert wird. Zu einem guten Methodennamen gehört auch, dass es sich ähnlich wie Prosa ließt. Methodennamen wie z.B. *DSYEV* oder *DGETRF* in LAPACK sind zwar deskriptiv, jedoch schwer zu verstehen, wenn man das Namensschema nicht kennt.

Damit Benutzer sich die Methoden, Klassen und Parameter besser merken oder besser wiederfinden können, sollten diese konsistent gehalten werden. Das bedeutet, dass alle Methoden, Klassen und Parameter, die ähnliche Funktionalitäten und Bedeutung haben, auch ähnlich heißen sollten. Außerdem sollte ein Styleguide festlegen, welche Richtlinien für die Benennung gelten. Namen sollten eindeutig vergeben werden. Es sollte keine falsche Konsistenz benutzt werden. Verschiedene Konzepte müssen verschiedene Namen haben. Zum Beispiel sollte der *set*-Präfix nur für Setter-Methoden benutzt werden.

Namen sollten zielgruppenorientiert gewählt werden. Das bedeutet, dass bei der Namensgebung darauf geachtet werden sollte, wer die Benutzer der API sind und welches Wissen sie mitbringen. Es sollten außerdem Fachbegriffe benutzt werden, die dem Kontext entsprechen.

Abkürzungen sollten vermieden werden, weil diese nicht jedem Benutzer bekannt sein könnten. Ausgenommen davon sind bekannte Akronyme (wie z.B. *XML*) oder Abkürzungen, die aus dem Kontext verständlich werden (wie z.B. *dim* für Dimension). Auch Abkürzungen sollten konsistent verwendet werden.

Namen sollten so kurz wie möglich gehalten werden, ohne gegen die vorhin genannten Richtlinien zur Namensgebung zu verstoßen. Zu lange Namen sind schwerer zu merken und die Lesbarkeit des Codes des Benutzers könnte darunter leiden, wenn z.B. Funktionsaufrufe sich deswegen um mehrere Zeilen erstrecken.

Spezielle Namen sollten generellen vorgezogen werden. In einer Bibliothek mit vielen Klassen, die ähnliche Funktionalitäten bieten, ist es einfacher für den Benutzer die gewünschte Klasse wiederzufinden. Außerdem leidet die Erweiterbarkeit darunter, wenn zu generelle Namen gewählt wurden, weil dann die Namensfindung für neue Klassen und Module zu sehr eingeschränkt ist.

Deskriptive Methodensignaturen

Eine Methodensignatur definiert typischerweise die Schnittstelle, die Programmierer verwenden zum Aufrufen der Methode. Diese umschießt den Methodennamen, die Anzahl der Parameter, Parametertypen und die Reihenfolge der Parameter.

Eine deskriptive Methodensignatur beinhaltet eine gute Namensgebung der Methode (s.h Abschnitt *Namensgebung* 4.2.1.1).

Die Parameter sollten ebenso deskriptive Namen haben, damit dem Benutzer klar wird, was welcher Parameter bezweckt. Dazu gehört auch, dass die geeigneten Parametermodifizierer gewählt werden (z.B. *const* in C++), damit das Benutzen der Methode erleichtert wird, weil dann die Bedeutung des Parameters klarer wird. Um eine gute Erlernbarkeit zu gewährleisten, sollten auch die Parameternamen konsistent gehalten werden sowie deren Reihenfolge. Eine zu lange Parameterliste ist schwerer zu merken als eine kurze. Außerdem kann eine lange Parameterliste den Code des Benutzers unleserlich machen, weswegen eine kürzere vorzuziehen ist. Aber auch hier gilt es, keine falsche Konsistenzen zu erzeugen.

Trennung von Interface und Implementierung

Es ist wichtig, das Interface von der Implementierung zu trennen. Der Benutzer sollte sich nicht mit den Implementierungsdetails beschäftigen müssen, um die API benutzen zu können. Je weniger der Benutzer wissen muss über die Bibliothek, desto einfacher wird das Erlernen der Programmierschnittstelle.

Ein weiterer Vorteil der Trennung ist, dass die Implementierung geändert werden kann, ohne dass das Interface geändert werden muss. So müssen Benutzer ihre Programme seltener anpassen, wenn eine neue Version der Bibliothek vorhanden ist.

Ein damit verwandtes Konzept ist die Datenkapselung. Es dient dazu, interne Daten vor direkten Zugriffen von außen zu schützen und damit Fehler zu vermeiden. Eine mögliche Fehlerquelle wäre beispielsweise, wenn Werte von zusammengehörigen Daten, wie die eines Objekts, nicht zueinanderpassen. Der Zugriff erfolgt stattdessen über wohldefinierte Schnittstellen, wodurch die API übersichtlicher wird und die Daten konsistent gehalten werden können. Zusätzlich wird dadurch die Anzahl der zu schreibenden Testfälle (z.B. in Modultests) reduziert, weil solche Inkonsistenzen nicht mehr getestet werden müssen.

Prinzip der geringsten Überraschung

Das Prinzip der geringsten Überraschung besagt, dass das Verhalten einer API den Erwartungen eines Benutzers entsprechen sollte. Beim Entwurf sollte also die Perspektive des Benutzers eingenommen werden, der nicht weiß, wie die Software intern arbeitet.

Dazu sollten unerwartete Seiteneffekte minimiert werden, passende Namen für Methoden und Variablen gegeben werden, wie beschrieben in Kapitel 4.2.1, Standards und Richtlinien eingehalten, sinnvolle Standardwerte vergeben und die API konsistent gehalten werden. Falls Seiteneffekte benötigt werden, sollten sie Benutzern nicht verschwiegen werden. Zum Beispiel können durch geeignete Kommentare aufgezeigt werden, welche Methoden Seiteneffekte auslösen.

Modularisierung

Modularisierung ist ein Konzept, um ein komplexes Gesamtsystem aus kleineren, wiederverwendbaren Modulen zusammenzusetzen. Module geben der Software eine klare Struktur. Somit wird es API-Benutzern einfacher gemacht, gewünschte Funktionalitäten einfacher zu finden.

Module sollten klein gehalten werden. Die Funktionalität eines Moduls sollte auf das Nötigste beschränkt sein und weitere Funktionalitäten sollten in andere Module ausgelagert werden, damit jedes Modul nur eine Sache macht. Das macht das Lernen einzelner Module einfacher und Benutzer können dann nur die Module einbinden, die auch gebraucht werden.

Es sollte jedoch auf eine lose Kopplung der Module geachtet werden. Das bedeutet, dass die Abhängigkeiten zwischen den Modulen minimiert werden sollten. Wenn Module zu stark voneinander abhängen, kann das die Vorteile der Modularisierung wieder aufheben.

Eine gute Modularisierung erlaubt es auch die Funktionalitäten zu verstecken, die nicht vom Benutzer verwendet werden sollen. Die API kann so einfach gehalten werden. Das steigert die Erlernbarkeit, weil die Anzahl an Modulen, die Benutzer kennenlernen müssen, beschränkt wird.

Prägnante API

Eine API sollte zu möglichst leserlichem Code führen. Das bedeutet, dass die API Benutzer nicht dazu zwingt, Codefragmente an mehreren Stellen in nahezu unveränderter Form einzufügen, also nicht zu Boilerplatecode führt. Boilerplatecode führt oft zu Copy & Paste, wodurch sich schnell Fehler einschleichen können.

Eine gute API nimmt Benutzern Arbeit ab. Das bedeutet, dass die API ein genügend hohes Abstraktionslevel verfügen sollte, damit Benutzer keinen unnötigen Code schreiben müssen, wenn die API das auch automatisch machen könnte.

Weiterhin sollten einfache Aufgaben mithilfe der Programmierschnittstelle auch einfach zu lösen sein. Das heißt, dass man für einfachere Aufgaben nicht unnötig viel Code schreiben muss. Auch die Verwendung der Klassen sollte nicht zu komplex sein. Benutzer sollten nicht unnötig viele Konfigurationen durchführen müssen, die mit dem trivialen Problem, welches sie lösen möchten, nicht zusammen hängt.

Konkret bedeutet das, dass beispielsweise für Parameter einer Methode, die nicht immer benötigt werden, ein Standardwert gesetzt ist.

Eine prägnante API ist logisch aufgebaut, sodass Benutzer intuitiv verstehen können welche Module, Klassen und Methoden sie verwenden müssen, welche Parameter die Methoden erwarten und wie mit Rückgabewerten umgegangen werden soll.

Erschwert Fehlgebrauch

Die API sollte so gestaltet sein, dass Benutzern wenige Möglichkeiten gegeben werden, Fehler zu machen. Dazu gehört neben einer guten Namensgebung (4.2.1.1) auch eine überraschungsarme API (4.2.1) ohne unerwartete Seiteneffekte.

Die API sollte Benutzer nicht dazu zwingen, eine Abfolge von Methoden in einer bestimmten Reihenfolge aufzurufen. Wenn beispielsweise bei der Initialisierung bestimmter Klassen mehrere Schritte durchgeführt werden müssen und diese in einer fest definierten Reihenfolge vorkommen müssen, dann erzeugt dies unnötigen Boilerplatecode. Benutzer tendieren dann meist dazu, solchen Codeschnipsel mit Copy&Paste in den eigenen Code einzufügen. Durch falsches Kopieren oder einfach nur durchs Vergessen einiger Schritte können dann Fehler passieren.

Um die Verwendung von uninitialisierten Membervariablen zu verhindern, sollten für diese Standardwerte gesetzt werden. Uninitialisierte Variablen können leicht vergessen werden, vor allem wenn die Programmierschnittstelle Benutzer nicht dazu zwingt, diese zu setzen. Berechnungen mit fehlerhaften Werten oder auch Abstürze können die Folge davon sein.

Gute Ausnahmebehandlung

Eine gute API beinhaltet auch eine gute Ausnahmebehandlung. Zuerst sollte sichergestellt werden, dass Ausnahmen überhaupt behandelt werden. Eine Bibliothek, die einfach abstürzt ist für Benutzer schwer zu debuggen. Ausnahmebehandlungen sollten ebenso auch vollständig sein. Für eine multilinguale Bibliothek ist es besonders wichtig auch Fehlerquellen zu berücksichtigen, welche erst durch das Interfacing von mehreren Sprachen entsteht. Ein Beispiel wurde im Kapitel 4.4.4 gezeigt.

Weiterhin ist zu beachten, dass Fehlermeldungen deskriptiv und konsistent sein sollten. Die Meldungen sollen den Benutzern mitteilen, welche Ausnahme warum ausgelöst wurde und von wo sie geworfen wurde. Außerdem sollten die Meldungen so gestaltet sein, sodass ein Benutzer versteht, was der Fehler ist und wie ein Benutzer den Fehler verbessern könnte, wenn es möglich ist. Wenn ein Fehler dann noch so früh wie möglich gemeldet wird, dann haben es Benutzer einfacher die Fehlerstelle wiederfinden zu können.

Ausnahmen sollten nicht für den Kontrollfluss verwendet werden. Das Werfen von Ausnahmen ist weniger performant und sollte nur für tatsächliche Fehler verwendet werden. Es sollte sich daher gut überlegt werden, was ein tatsächlicher Ausnahmefall ist, bei dem ein Programm nicht weiter rechnen kann, und was nicht.

Wenn echte Ausnahmen auftreten, sollten diese nicht still von der Bibliothek behandelt werden, wie z.B. automatisch auf Standardwerte zurück zu fallen und einfach weiter zu rechnen. Alle Fehler und Ausnahmen müssen immer gemeldet werden, damit diese behoben werden können und Benutzer kein unvorhergesehenes Verhalten debuggen müssen. Das verstößt gegen das Prinzip der geringsten Überraschung (s.h. 4.2.1).

Ausnahmen sollten sich auch als solche bemerkbar machen. Zum Beispiel sollte in C++ kein Nullzeiger zurückgegeben werden von einer Methode, wenn ein Fehler aufgetreten ist. Benutzer müssten somit selbst Nullzeigerabfragen in ihren Code einbauen, was den Benutzercode unleserlich macht. Eine bessere Variante wäre es hier stattdessen eine Exception zu werfen.

Generell sollten Rückgabewerte von Methoden und Funktionen nicht für die Ausnahmebehandlungen benutzt werden. Benutzer könnten leicht vergessen, die Rückgabewerte zu prüfen und die Ausnahmen zu behandeln.

Gute Dokumentation

Als Teil der Softwaredokumentation dient das Benutzerhandbuch zur Orientierung. Es sollte beschreiben, wie die Software zu installieren ist, welche Bibliotheken benötigt werden, wie die Softwareumgebung konfiguriert werden muss und auch die Hardwareanforderungen klären. Es sollte Benutzern helfen, einen Überblick über die Software zu erhalten und erklären, was sie kann und was nicht. Der gesamte Funktionsumfang sollte gut beschrieben sein und am besten mit Beispielen unterlegt werden. Zusätzlich sollten die verwendeten Algorithmen transparent gemacht werden und alle relevanten technischen Berichte referenziert werden.

Auch Codekommentare gehören zu einer guten Dokumentation. Klassen und Methoden der Schnittstelle sollten vollständig und korrekt kommentiert werden, aber nicht zu ausschweifend werden. Allzu häufig wird bei Änderungen an Methoden vergessen, die entsprechenden Kommentare zu aktualisieren. Gute Codekommentare beschreiben den Sinn und Zweck der jeweiligen Klasse oder Methode sowie die zugehörigen Parameter und Rückgabewerte.

4.2.2. Minimalität und Vollständigkeit

Vollständig

Eine vollständige API bietet Benutzern genau die Funktionalitäten an, die sie brauchen. Welche Funktionalitäten die Benutzer brauchen, kann durch geeignetes Anforderungsmanagement herausgefunden werden. Weil es trotzdem nahezu unmöglich ist, eine 100% vollständige API zu gestalten, sollte die API erweiterbar sein, damit Benutzer fehlende Funktionalitäten selbst implementieren können.

Minimal

Die API sollte so klein wie möglich gehalten werden, ohne gegen das Prinzip der Vollständigkeit zu verstoßen. Das bedeutet, dass nur die Module nach außen geöffnet werden, welche auch von API-Benutzern verwendet werden sollen. Alle anderen Module sollten vor Benutzern versteckt werden. Um dies zu realisieren, sollte es eine klare Trennung zwischen internen Komponenten und der API geben.

Redundanz vermeiden

Die API sollte wenig bis keine Redundanzen vorweisen. Das bedeutet, dass Codeduplikationen vermieden werden sollten. Codeduplikationen machen den Code schlecht wartbar, weil eine Änderung in mehreren Stellen nachgezogen werden muss. Es sollte so viel Code wie möglich wiederverwendet werden. Das hilft auch Benutzern zur Orientierung. Um eine gewünschte Funktionalität zu finden, müssen Benutzer nicht herausfinden, ob zwei ähnlich aussehende Methoden tatsächlich gleich sind oder es kleine Unterschiede gibt. Zum Beispiel sollte doppelter Code in eigene Funktionen oder Methoden ausgelagert werden und abstrakte Basisklassen verwendet werden.

Zusätzlich sollte eine API Benutzern dabei behilflich sein, benötigte Funktionalitäten schnell zu finden. Wenn nun aber die API mehrere Möglichkeiten bietet, eine Funktionalität zu benutzen, dann müssen Benutzer herausfinden, ob und welche Unterschiede die verschiedenen Ansätze sich unterscheiden. Um Redundanzen zu vermeiden, sollte die Programmierschnittstelle in dieser Hinsicht eingeschränkt werden.

4.2.3. Effizienz

Für Numerikbibliotheken wie *SG++* ist es wichtig, dass sie effizient arbeiten. Daher sollten die Verbesserungen der Schnittstelle die Performance möglichst wenig beeinträchtigen. Trotzdem sollte die API nicht zu sehr umgebogen werden, um kleine Leistungsgewinne zu erhalten.

4.2.4. Erweiterbarkeit und Portierbarkeit

Plattformunabhängigkeit

Die Bibliothek sollte unabhängig vom Betriebssystem sein. Das heißt, dass Systemaufrufe vermieden werden sollten, zum Beispiel indem eine geeignete Bibliothek benutzt wird. Falls systemabhängige Funktionen gebraucht werden, sollten die entsprechenden Module als solches gekennzeichnet werden, damit die Bibliothek einfacher portiert werden kann. Am besten trennt man die systemabhängigen Module von den systemunabhängigen, um die Wartbarkeit zu verbessern.

Ähnliches gilt auch für Hardware. Die Bibliothek sollte nicht von einer bestimmten Hardware abhängen. Häufig werden aber Algorithmen in der Numerik an spezielle Hardware angepasst, wie zum Beispiel an spezielle GPUs, um die Berechnungen schneller zu machen. In solchen Fällen sollte der allgemeine Code, welcher auf beliebiger Hardware läuft, getrennt werden von dem Code, welcher an spezielle

Hardware angepasst ist. Benutzern sollte somit die Wahlmöglichkeit gegeben werden, welchen Code sie nun verwenden wollen.

Open-Closed-Prinzip

Die Definition des Open-Closed-Prinzips nach Bertrand Meyer [Mey97] besagt, dass Module, Klassen, Funktionen usw. für Erweiterungen offen, aber für Modifikationen geschlossen sein.

Module sind offen für Erweiterungen, wenn Benutzer die Funktionalitäten des Moduls selbst erweitern können. Ein Beispiel wäre es Benutzern zu ermöglichen eigene Operatoren zu schreiben. Diese Offenheit ist wichtig, da ein API-Designer nicht vorhersehen kann, welche Funktionalitäten ein Benutzer benötigt. Das erhöht die Flexibilität der Software und macht es einfach die Software zu erweitern.

Gleichzeitig sollten die Module geschlossen für Veränderungen sein. Das bedeutet, dass das Modul ein stabiles Interface hat und die Implementierung nur zum Ausbessern von Fehlern geändert wird. Neue Funktionalitäten müssten also durch weitere Klassen hinzugefügt werden.

Das Open-Closed-Prinzip soll es also Benutzer ermöglichen Module zu erweitern, ohne dass deren Quellcode geändert werden muss.

4.2.5. Fehlerfreiheit

Ein Fehler in der Bibliothek betrifft auch alle Software, welche die Bibliothek benutzt. Deshalb ist es besonders wichtig, dass Bibliotheken möglichst fehlerfrei sind.

Zur Fehlerfreiheit gehört, dass alle Module richtig kompilieren. Dazu sollten alle möglichen Kombinationen getestet werden. Folgende Aspekte sollten dazu berücksichtigt werden:

- Kompiliert die Software als Ganzes? Compilieren einzelne Module?
- Werden die Interfaces korrekt generiert?
- Kompiliert die Bibliothek auf allen vorgesehenen Plattformen? Wird an spezifische Hardware angepasster Code korrekt kompiliert?

Einzelne Module sollten möglichst fehlerfrei funktionieren. Dies kann zum Beispiel durch Unittests sichergestellt werden.

Die Software als Ganzes sollte stabil laufen. Dafür können beispielsweise Integrationstests verwendet werden.

Falls es noch bekannte Fehler in der Software gibt, sollte dies den Benutzern klar dargelegt werden. Es sollte zusätzlich noch eine Möglichkeit geben, neue Fehler zu melden.

4.3. Techniken

Um die im letzten Kapitel definierten Designrichtlinien umsetzen zu können, werden im Folgenden einige Techniken vorgestellt. Es ist eine Sammlung aus Konventionen zum Codestil, Entwurfsmustern und C++-spezifischen Programmierkonzepten.

4.3.1. Allgemeine Techniken

Styleguide

Ein Styleguide ist eine Sammlung von Konventionen, welche beschreibt, nach welchen Regeln und Vorschriften ein Quellcode geschrieben werden sollte. Er beschreibt, wie die Form und Struktur eines Codes gestaltet sein soll. Viele Vorschriften, die in Styleguides beschrieben sind, sind oft Geschmackssache. Sie dienen dazu, den Programmierstil durch das Projekt hinweg konsistent zu halten. Ein guter Styleguide kann aber auch den Quellcode lesbarer und verständlicher machen und kann dabei helfen, mögliche Fehlerquellen zu eliminieren.

Einige Regeln wurden im schon vorhandenen Styleguide für *SG++* definiert. Sinnvolle Regeln, bzw. Regeln, die keine Nachteile bringen, wurden übernommen, um Anpassungen im Code konsistent zu halten mit der restlichen Codebasis.

Namensgebung: Die Styleregeln für die Namensgebung wird direkt aus dem schon vorhandenen Styleguide von *SG++* übernommen, um konsistent zu bleiben.

- Verzeichnisnamen sollen kleingeschrieben werden.
- Namen von Konstanten und Defines sollen durchweg groß geschrieben werden.
- Namen von Funktionen und Variablen sollen in CamelCase geschrieben sein und mit kleinem Buchstaben anfangen.
- Namen von Klassen sollen in CamelCase geschrieben sein und mit großem Buchstaben anfangen.
- Namen sollten nicht mit einem Unterstrich beginnen.

Außerdem kommen noch die folgenden Regeln hinzu, um mit der API-Designrichtlinie für eine gute Namensgebung konform zu sein:

- Abkürzungen sollten vermieden werden.
- Alle Getter-Methoden sollten den Präfix *get* haben und alle Setter-Methoden den Präfix *set*. Andere Methoden dürfen keinen *get*- oder *set*-Präfix besitzen.

  Eine Ausnahme kann für Getter-Methoden gemacht werden, wenn diese einen Booleanwert zurückgeben. Anstelle des *get*-Präfixes kann ein *is*-Präfix oder ein *has*-Präfix benutzt werden.

Um die Lesbarkeit von *if*-Abfragen zu verbessern, sollte anstelle von

```
if(obj.getIsVerbose()) ...
```

die Methode umbenannt werden. Eine bessere Alternative ist

```
if(obj.isVerbose()) ...
```

Klammerung: Einzelne Anweisungen nach *if*-Abfragen sollten entweder einen Block gesetzt werden (also mit geschweifte Klammern umschließen) oder in die selbe Zeile geschrieben werden wie die Abfrage.

```
if(...)
  x = 5;
y = 6;
```

Hier könnte beispielsweise durch fehlerhafte Autoformatierung die Anweisung *y = 6;* eingerückt werden, was dann den Eindruck erweckt, dass sie ebenfalls in die Abfrage gehört.

Klammern zu setzen ist häufig besser, vor allem, wenn bei der Wartung neue Anweisungen in die *if*-Abfragen hinzukommen können:

```
if(...)
{
  x = 5;
}

y = 6;
```

Oder wenn man sicher ist, dass es bei einer einzelnen Anweisung bleibt, einfach in die gleiche Zeile schreiben:

```
if(...) x = 5;

y = 6;
```

Somit wird die Wartung von Software erleichtert, weil verdeutlicht wird, welche Anweisungen nun zur Abfrage gehören.

Kommentare: Ein Code sollte vollständig kommentiert sein. Das ist wichtig, damit Benutzer sich schnell im Code zurechtfinden. Dies beinhaltet, dass jede Klasse und Methode einen Kopfkommentar enthält.

Kommentare sollten idealerweise so verfasst sein, damit Benutzer verstehen, was der Code macht, ohne den Code selbst anschauen zu müssen. In Methodenkommentaren sollte zusätzlich noch der Zweck der Parameter und Rückgabewerte beschrieben werden.

Für *SG++* wird zur Generierung von Dokumentation das Programm *Doxygen* benutzt. Das erfordert, dass die Kommentare *Doxygen*-konform sind.

Klassen und Strukturen: Eine Struktur und eine Klasse verhalten sich in C++ ähnlich. Der einzige Unterschied ist, dass der Zugriff auf Variablen und Methoden für Strukturen standardmäßig öffentlich ist, während bei Klassen der Zugriff privat ist.

Typischerweise wird zwischen Strukturen und Klassen ein semantischer Unterschied gemacht, wie z.B. beim Google-Styleguide [WSE+]. So werden Strukturen nur zur reinen Datenhaltung benutzt, wohingegen Klassen weitere Funktionalitäten besitzen. Strukturen sollten also keine Methoden besitzen. Eine Ausnahme bilden Konstruktoren, weil es in C++ sonst nicht möglich ist, Standardwerte zu definieren.

Standardwerte: Um das Instanziieren von Objekten zu erleichtern, können in C++ im Konstruktor für Strukturen und Klassen definiert werden.

Mit guten Standardwerten kann die Menge an Code, die ein Benutzer schreiben muss, reduziert werden, welches wiederum die Lesbarkeit des Benutzercodes verbessert. Außerdem verhindern Standardwerte, dass Variablen ausversehen uninitialisiert verwendet werden, was den Fehlgebrauch mindert. Weiterhin geben gute Standardwerte den Benutzern eine Orientierungshilfe. Gute Standardwerte sind solche Werte, die häufig benutzt werden und selten abgeändert werden müssen vom Benutzer.

Strings vermeiden: Es ist von Vorteil Strings zu vermeiden, wenn es bessere Alternativen gibt. Beispielsweise können Strings dazu benutzt werden Typinformationen zu speichern. Das ist aber sehr fehleranfällig. Benutzer könnten sich verstippen und der Kompiler hätte keine Möglichkeit den Fehler zu bemerken.

Wenn anstelle von Strings Aufzählungstypen benutzt werden, kann zur Kompilierzeit geprüft werden, ob die richtigen Typnamen benutzt wurden. In Kombination mit einer *switch*-Anweisung kann der Kompiler zusätzlich noch prüfen, ob auch alle Elemente des Aufzählungstyps berücksichtigt wurden. Ein weiterer Vorteil wäre, dass optimierende Kompiler *switch*-Anweisungen in Sprungtabellen umwandeln können, anstelle mehrere *if*-Abfragen durchzulaufen, um die Effizienz des Programms zu steigern.

Auch die Erschließbarkeit wird verbessert. Das heißt, dass Benutzer am Aufzählungstypen sehen können, welche Werte denn benutzt werden können. Bei Strings hingegen gibt es keine zentrale Stelle an der sich Benutzer orientieren könnten.

Include-Direktiven: Wenn es möglich ist, dann sollten *#include*-Direktiven alphabetisch sortiert und nach Modulen gruppiert werden. Dadurch wird das Wiederfinden und Hinzufügen von Headern erleichtert.

Entwurfsrichtlinien und -muster

In der Softwaretechnik haben sich bestimmte Designs für Softwarearchitekturen bewährt. Solche Designs, die allgemein genug sind, um sie auch in anderen Softwareprojekten einsetzen zu können, heißen Entwurfsmuster. Im Folgenden werden einpaar solcher Entwurfsmuster vorgestellt, welche beim Refactoring von *SG++* zum Einsatz kamen.

Als Entwurfsrichtlinien ist hier gemeint, wo und wie typische Programmierkonzepte, vor allem aus der objektorientierten Programmierung, verwendet werden sollten. Es wird aufgezeigt, welche Vorteile sie bringen und wann sie vermieden werden sollten.

Vererbung: Die objektorientierte Programmierung bietet die Möglichkeit Code wiederzuverwenden durch die Vererbung von Attributen und Methoden der Basisklasse. Zusätzlich können durch die Überschreibung von virtuellen Methoden und durch das Hinzufügen eigener Attribute und Methoden das Verhalten von abgeleiteten Klassen verändert werden.

Für die Verwendung von Vererbung sollten einige Richtlinien beachtet werden [Red11]:

- Klassen, von denen abgeleitete Klassen gebildet werden können, sollten explizit dafür entworfen werden. Das heißt, dass klar sein sollte, welche Methoden virtuell sein sollen und somit in den abgeleiteten Klassen neu implementiert werden können. Falls eine Klasse nicht abgeleitet werden soll, sollte dies explizit als solches gekennzeichnet werden.
- Eine Klasse sollte nur dann von einer anderen Klasse abgeleitet werden, wenn das *Ist-Ein*-Kriterium erfüllt wird. Das bedeutet, dass in einem Programm die Basisklassen ersetzt werden können durch deren abgeleitete Klassen, ohne dabei das Verhalten des Programms zu verändern. Das nennt man das liskovsche Substitutionsprinzip [LW94]. Falls dieses Kriterium nicht erfüllt ist, sollte anstelle von Vererbung besser die Komposition verwendet werden.
- Zu tiefe Vererbungshierarchien sollten vermieden werden. Die erhöhte Komplexität erschwert das Erlernen der API und ist fehleranfällig. Benutzer müssen nachvollziehen können, welche Methoden wo neu definiert werden und wo Membervariablen gesetzt werden.
- Anstelle von Mehrfachvererbung und tiefen Vererbungshierarchien sollten, wenn möglich, Interfaces verwendet werden.

  Durch rein virtuelle Methoden können abstrakte Klassen definiert werden, die dann als Interfaces dienen. Solche rein virtuellen Methoden setzen fest, welche Methoden die abgeleiteten Klassen anbieten und implementieren müssen. Somit kann die Schnittstelle von der Implementierung getrennt werden, was die Implementierungen austauschbar macht und die Bibliothek flexibler.

  Ein weiterer Vorteil von Interfaces ist es, dass Benutzer unterschiedliche Klassen mit gleichem Verhalten auch gleichbehandeln können, als ob es die gleichen Klassen wären. Das erleichtert die Verwendung der Bibliothek.

  Hier sollte jedoch angemerkt werden, dass zu virtuelle Methoden die Performance der Bibliothek beeinträchtigen können [Bla08] und Benutzern einen falschen Eindruck geben, welche Methoden reimplementiert werden können und welche nicht reimplementiert werden sollen. Es sollte gut abgewogen werden, welche Methoden virtuell deklariert sein sollen.
- Um eine API minimal zu halten, sollten keine extra Abstraktionsebenen hinzugefügt werden, die unnötig sind. Beispielsweise sind Basisklassen, von denen nur eine Klasse ableitet, unnötige Abstraktionen.

Konstante Klassen, Methoden, Rückgabewerte und Referenzparemeter: Klassen werden dann als konstant bezeichnet, wenn diese selbst keinen inneren veränderbaren Zustand besitzen.

Auch wenn eine Klasse einen veränderbaren Zustand braucht, sollten alle Methoden, die den Zustand der Klasse nicht ändern, als konstant deklariert werden. Methoden, die den Zustand ändern, sollten durch eine gute Dokumentation den Nutzer darauf hinweisen, wie der Zustand geändert wird.

Wenige bzw. keine Zustände schränken die Möglichkeiten ein, wo Fehler auftauchen können. Mit zustandslosen Klassen wird somit das Debuggen einfacher und es müssen weniger Tests geschrieben werden.

Bei der Benutzung von konstanten Methoden kann der Benutzer sicher sein, dass keine unerwünschten Seiteneffekte auftreten, was die Fehleranfälligkeit reduziert und Benutzern Überraschungen erspart.

Dasselbe Argument gilt auch für Parameter, die als Referenz übergeben werden. Deswegen sollten Referenzparameter immer als konstant übergeben werden, wenn sie nicht verändert werden in der Methode.

Konstante Zeiger als Rückgabewerte teilen Benutzern mit, wer Besitzer des Objekts ist und wer es somit es auch wieder löschen sollte. Wenn Methoden also Zeiger auf interne Daten zurückgeben, dann sollte dies nur über konstante Zeiger geschehen. Ansonsten könnten Benutzer auf die Idee kommen, den Rückgabewert selbst zu löschen, was dann zu doppelten Speicherfreigaben und damit zum Programmabsturz führen könnte. Konstante Zeiger signalisieren Benutzern also, wie sie mit dem Rückgabewert umgehen sollen, und erfüllen damit die Richtlinie der prägnanten API (4.2.1).

Fabrik: In objektorientierter Programmierung ist eine Fabrik ein Objekt, welches hilft, viele verschiedene Objekte einer gemeinsamen Basisklasse zu erstellen. Ein Fabrik-Objekt erlaubt es also, die Polymorphie von Klassen für die Objekterstellung auszunutzen. Konkrete, abgeleitete Klassen können somit abgekapselt werden. Benutzer müssen dann nur noch die Basisklasse und die Fabrik kennen, was die Erlernbarkeit der API steigert.

Weiterhin kann eine Fabrik komplexere Objekterzeugungen übernehmen, um Benutzern die Objektgenerierung zu vereinfachen. Dies kann auch die Fehleranfälligkeit verringern, wenn für die Erzeugung von Objekten mehrere Schritte in einer bestimmten Reihenfolge benötigt werden. Damit kann eine Fabrik Boilerplatecode im Code des Benutzers verhindern und den ihn lesbarer machen.

Zugriffsmodifizierer: In der objektorientierten Programmierung werden Zugriffsmodifizierer für Membervariablen und Methoden benutzt, um deren Sichtbarkeit für andere Klassen zu regeln.

Der Zugriffsmodifizierer *public* ermöglich es zum Beispiel, dass Methoden und Variablen von außerhalb der zugehörigen Klasse benutzt werden können. Der Zugriffsmodifizierer *private* hingegen verbietet einen Zugriff von außen. Dann gibt es noch *protected*, die es nur abgeleiteten Klassen erlaubt, auf die Elemente der Basisklasse zuzugreifen.

Den Zugriff auf Methoden und Membervariablen in der API zu regeln ist wichtig, um Benutzern mitzuteilen, welche Methoden benutzt werden können und welche nicht. Somit können Schnittstellen von Implementierungen getrennt werden und Daten besser gekapselt werden.

Als *public* sollten demnach nur die Methoden und Variablen deklariert sein, die Benutzer auch verwenden sollen. Alles andere sollte als *private* oder *protected* deklariert sein.

C++ spezifische Techniken

Zum Schluss werden noch einige Techniken vorgestellt, die es in C++ gibt, aber nicht in allen objektorientierten Programmiersprachen zu finden sind. Hier wird speziell auf C++ eingegangen, weil *SG++* in dieser Sprache implementiert ist.

Const-Correctness: Konstante Klassen, Methoden, Rückgabewerte und Referenzparameter bringen viele Vorteile mit sich, wie in Kapitel 4.3.1 erläutert wurde. Für die Realisierung gibt es in C++ das Schlüsselwort *const*. Die konsistente Nutzung des Schlüsselworts, überall da wo Variablen, Rückgabewerte und Parameter nicht verändert werden sollen, wird als Const-Correctnes bezeichnet. Die Benutzung von *const* bringt viele Vorteile, die sich vor allem bei größeren Programmen bemerkbar machen [Str13]. Es dient dazu konstante Klassen und Methoden zu deklarieren, um damit eine gewisse Typsicherheit zu gewährleisten.

Es kann auch als eine Art Dokumentation des Programmcodes gesehen werden, wobei die Beschränkungen durch das Benutzen von *const* durch den Kompiler zusätzlich geprüft werden.

Zum Beispiel werden große Objekte oft als Referenzen in Funktionen und Methoden übergeben. Wenn die entsprechenden Parameter als *const* deklariert sind, dann wird so Benutzern mitgeteilt, dass dies nur aus Effizienzgründen geschieht, nicht weil das Objekt selbst verändert wird. Das Weglassen von *const* in der Parameterdeklaration würde für Referenzparameter darauf hindeuten, dass die Funktion oder Methode die Absicht hat, das entsprechende Objekt zu ändern.

Konstante Rückgabewerte können ebenso durch die Verwendung von *const* verwirklicht werden.

```
class MyClass {
  int* doSomething();
};
```

Im obigen Beispiel können Benutzer nicht wissen, ob sie den Rückgabewert selbst löschen sollen oder ob der Wert automatisch durch den Desktruktor von *MyClass* gelöscht wird.

```
class MyClass {
  int* const doSomething();
};
```

Durch das Deklarieren des Rückgabewerts als konstant können Benutzer sichergehen, dass der Wert von *MyClass* gelöscht wird.

Const-Correctness erspart Benutzern unvorhergesehene Seiteneffekte zu vermeiden und hilft das Programm korrekt zu halten.

Namespaces: In C++ können Namensräume verwendet werden, um den Code zu strukturieren und um zusammengehörige Funktionalitäten zusammenzufassen. Das gibt dem Code eine modulare Struktur und kann Namenskonflikte verhindern. Jedoch sollte die *using*-Direktive vermieden werden, da es zu falschen Namensauflösungen führen kann und Namenskonflikte wieder einführen könnte.

Template-Metaprogrammierung: Templates sind ein mächtiges Konzept, um in C++ generisch programmieren zu können. Damit können Klassen- und Funktionstemplates definiert werden, um Datentypen zu parametrisieren. So wird nur eine Implementierung für verschiedene Typen gebraucht. Da die Instanziierung der Template-Klassen und -Funktionen zur Kompilierzeit geschieht, ist der Code zur Laufzeit auch effizient. Das erspart Entwicklern, den Code zu duplizieren und Benutzern den redundanten Code bzw. die redundante Schnittstelle zu lesen.

Ein Nachteil der Template-Metaprogrammierung in C++ ist, dass alle Methodendefinitionen sich in der Headerdatei befinden müssen. Für Benutzer, die diese Dateien ansehen, kann es schnell unübersichtlich werden. Eine Abhilfe hierfür wäre, die Implementierungen in eine separate Datei zu stecken und diese dann durch eine *#include*-Direktive in der Headerdatei einzufügen. Solange es gut kommentiert wurde, ist diese ungewöhnliche Art die beste Möglichkeit die Schnittstelle von der Implementierung für Templateklassen zu trennen.

Bei umfangreicher Nutzung von Templates kann die Kompilierzeit darunter leiden. Wenn Templates auf die gesamte Bibliothek ausgeweitet werden, erhält man sogar eine Header-Only-Bibliothek. Für Benutzer bedeutet das, dass sie Templateklassen jedes Mal selbst kompilieren müssen, wenn sie ihren eigenen Code bauen und somit mehr Zeit zum Kompilieren benötigt wird. Daher sollte man sich gut überlegen, welche Klassen tatsächlich Templateklassen sein sollten.

4.4. Unterstützung mehrerer Sprachen

Die *SG++*-Bibliothek soll in der Lage sein, Schnittstellen für weitere Programmier- und Skriptsprachen anzubieten. Mit der Unterstützung von mehreren Programmiersprachen kann der Kreis der Benutzer vergrößert werden. Benutzer müssen somit nicht unbedingt C++ können, um die Bibliothek zu verwenden. Sie können die Programmiersprache wählen, mit der sie sich am besser auskennen.

Die Vorteile, mehrere Sprachen zu unterstützen, sind also offensichtlich. Jedoch ist damit ein erhöhter Implementierungs- und Wartungsaufwand verbunden. Bei Numerikbibliotheken stellt sich außerdem die Frage, welche Leistungseinbußen die Verwendung anderer Programmiersprachen mit sich bringen. Um diese Frage zu klären, werden Laufzeiten der verschiedenen Sprachen gemessen und miteinander verglichen.

In dieser Arbeit wird sich vor allem auf Python und Java konzentriert. Jedoch wird die Erweiterbarkeit auf weitere Sprachen nicht außer Acht gelassen.

4.4.1. Vorteile von Python und Java gegenüber von C++

Java und Python haben gegenüber C++ viele Vorteile. Da wären zum Beispiel die simple Syntax, die automatische Speicherverwaltung, welches Speicherlecks verhindert, Arrays die es nicht erlauben über den gültigen Arraygrenzen zu hinaus gehen und ein Modulsystem, welche den Buildprozess vereinfachen.

Vor allem Python bietet die Möglichkeit, verschiedene Softwarepakete miteinander zu verbinden um so eine für die Bedürfnisse von Wissenschaftlern und Ingenieuren angepasste Softwareumgebung zu erstellen. Die Ergebnisse von Berechnungen können einfach visualisiert und analysiert werden [Lan09]. Das Erstellen von GUIs und das Parsen von verschiedenen Formaten wird durch viele verfügbare Softwaremodule vereinfacht. Durch die Flexibilität und die einfache Benutzbarkeit von Python kann so die Produktivität der Benutzer gesteigert werden.

4.4.2. Java und Python Schnittstellen

SWIG ist eine viel benutzte Bibliothek zum Generieren von Schnittstellen für Skriptsprachen. Auf der Webseite gibt es eine Auflistung einiger Projekte, die *SWIG* verwenden (`http://www.swig.org/projects.html`). Unter den Projekten sind auch einige Anwendungen in der Numerik zu finden.

Ein anderer Ansatz C-Erweiterungen für Python zu schreiben ist *Cython* zu verwenden. *Cython* ist eine Erweiterung der Skriptsprache Python, die es zusätzlich erlaubt C- und C++-Funktionen aufzurufen. *Cython* wird beispielsweise in *SciPy* viel benutzt.

Weitere Alternativen wären hier für Python manuell Wrapper in C++ zu schreiben. Python bietet hierfür geeignete C++-Schnittstellen an. Für Java gibt das Java Native Interface (JNI) und das Java Native Access (JNA). Jede Sprache besitzt eigene Schnittstellen, um C- oder C++-Code aufzurufen. Das erfordert von Entwicklern der Bibliothek, dass sie sich in die verschiedenen Schnittstellen einarbeiten. Für jede weitere Sprache, die unterstütz werden soll, müssten eigene Wrapper geschrieben werden. Das ist mühsam und erfordert viel Zeit, vor allem, wenn die C++Schnittstellen sich ändern und die Wrapper umgeschrieben werden müssen.

Weil für *SG++* geplant ist Python und Java (und evtl. in Zukunft noch weitere Sprachen) zu unterstützen und weil das manuelle Schreiben von Wrappern dafür zu aufwendig ist, scheint *SWIG* das beste Werkzeug für das Erstellen von Schnittstellen zu sein.

4.4.3. Leistungsunterschiede

Wie im Abschnitt 4.4.1 beschrieben wurde, bringen Java und Python im Gegensatz zu C++ einige Vorteile mit sich. Trotzdem werden Numerikbibliotheken üblicherweise nicht in diesen Sprachen geschrieben, weil diese als langsam gelten. Aber in der Numerik ist eine hohe Performance eben wichtig, vor allem wenn große Probleme gerechnet werden sollen.

Als Kompromiss zwischen effizientem Code und erhöhter Benutzerproduktivität ist *SG++* in C++ implementiert, bietet aber Schnittstellen für Java und Python an. Im folgenden Unterkapitel wird der von Java und Python produzierte Overhead gegenüber C++ analysiert. Durch diesen Leistungsvergleich

soll verdeutlicht werden, ob es sich lohnt die Java- und Python-Schnittstellen zu nutzen oder ob die Leistungseinbußen doch zu hoch sind.

Mit Overhead ist hier der zusätzliche Aufwand und die damit verbundenen Laufzeiteinbußen gemeint, den Java- und Python im Gegensatz zu C++ benötigen, um die gleichen Berechnungen durchzuführen. Das bedeutet auch, dass beispielsweise die Zeit, welche die Java VM zum Starten braucht, nicht mitgemessen wird. Die Messungen müssen deshalb innerhalb der Testprogramme selbst durchgeführt werden, nicht durch externe Programme. Alle Ausgaben der Testprogramme werden ausgeschlossen von der Messung. Das Einlesen von Datensätzen wird mit gemessen, weil dafür die C++-API benutzt wird. Alle weiteren Eingaben werden aber ebenfalls ausgeschlossen.

Es sollen die Laufzeiten einer Programmausführung in Sekunden gemessen werden. Dafür werden mehrere Programmdurchläufe ausgeführt und deren Laufzeit gemessen, um dann den Durchschnittswert zu berechnen. Die mehreren Programmdurchläufe müssen jedoch durch ein Skript realisiert werden, welches das jeweilige Testprogramm mehrmals aufruft. Eine Schleife im Testprogramm selbst kann in Java beispielsweise den Effekt haben, dass die späteren Durchläufe schneller sind als die früheren, weil der Java JIT-Kompiler Optimierungen durchführen könnte. In Java würde man somit nicht den Durchschnitt mehrerer Einzelausführungen messen.

Für die Leistungstests werden zwei verschiedene Szenarien betrachtet. Für beide Szenarien wurde die neue *SG++*-Version verwendet. Als C++-Kompiler für die Testprogramme wird der *GCC* verwendet mit dem C++11-Standard und dem Kompilerflag *-O3*. Java wird in der Version 6 verwendet und Python in der Version 2.7. Die verwendete Hardware hat acht *Intel i7-2600*-Prozessoren mit 15 GB Arbeitsspeicher. Auf dem Testrechner läuft das Betriebssystem *Kubuntu 12.04*.

Szenario: Quadratur mit Verfeinerung

In diesem Szenario wird ein reguläres Gitter auf Level 3 mit linearen Basisfunktionen erstellt und 50-mal adaptiv verfeinert. Vor jedem Verfeinerungsschritt wird noch zusätzlich eine Quadratur der n-dimensionalen Funktion $f(x) = 4x \cdot (1 - x)$ durchgeführt. Die Messungen werden auf Gittern mit verschiedenen Dimensionalitäten durchgeführt, nämlich von 1 bis 10. Das heißt, dass pro Skript- und Programmiersprache 10 verschiedene Durchschnittswerte gemessen werden.

Die dafür verwendeten *SG++*-Funktionalitäten sind wenig rechenintensiv, benötigen aber viele API-Aufrufe. Dieses Szenario verdeutlicht, welche Auswirkungen das Verhältnis der API-Aufrufe zum Rechenaufwand auf die Gesamtlaufzeiten der verschiedenen Versionen haben.

Um die Durchschnittswerte zu berechnen, werden pro Dimensionalität des Gitters 1000 Programmdurchläufe ausgeführt und gemessen. Die Testprogramme nutzen nur einen CPU-Kern, weswegen zum Messen die CPU-Clock verwendet wird, um genauere Messergebnisse zu erhalten.

Python vs. C++ Das Diagramm 4.1 zeigt für die Dimensionen 1 bis 10 die Durchschnittswerte der Laufzeiten für 1000 Durchläufe.

Am Diagramm kann man erkennen, dass Python einen messbaren Overhead produziert. Die Tabelle 4.1 zeigt die zum Diagramm zugehörigen Durchschnittwerte der Laufzeiten.

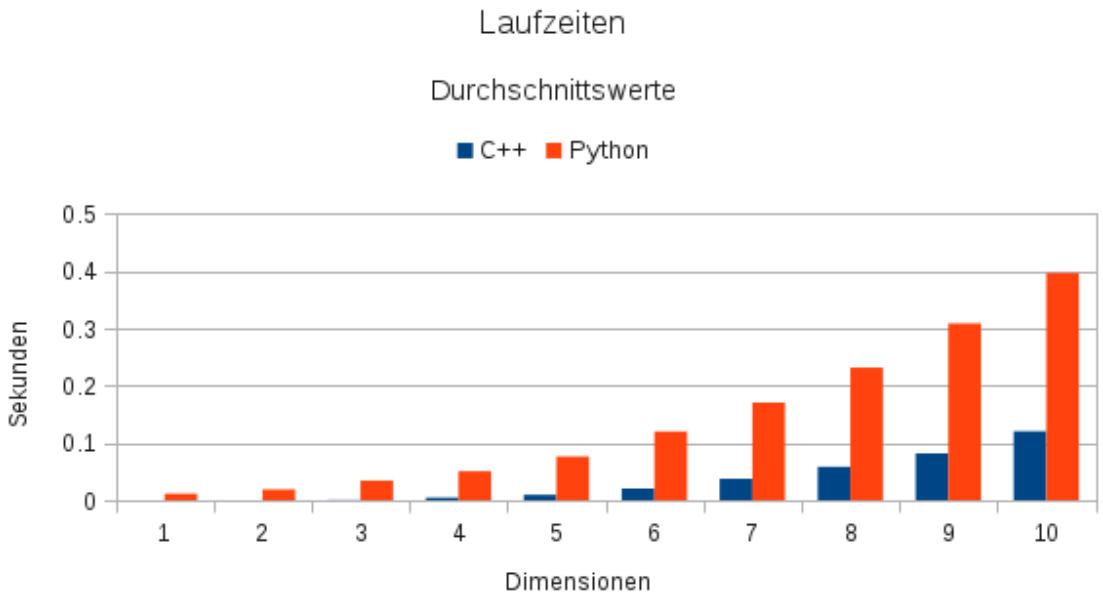

Abbildung 4.1.: Vergleich der Durchschnittslaufzeiten von C++ mit Python

Dimension	C++	Python	Differenz (absolut)	Differenz (relativ)
1	0.00000	0.01119	0.01119	-
2	0.00000	0.01875	0.01875	-
3	0.00041	0.03405	0.03364	8204.878049%
4	0.00439	0.05049	0.04610	1050.113895%
5	0.00948	0.07622	0.06674	704.008439%
6	0.02033	0.11976	0.09943	489.080177%
7	0.03741	0.17031	0.13290	355.252606%
8	0.05841	0.23138	0.17297	296.130800%
9	0.08161	0.30847	0.22686	277.980640%
10	0.12036	0.39676	0.27640	229.644400%

Tabelle 4.1.: Durchschnittslaufzeiten in Sekunden über 1000 Durchläufe: C++ gegen Python

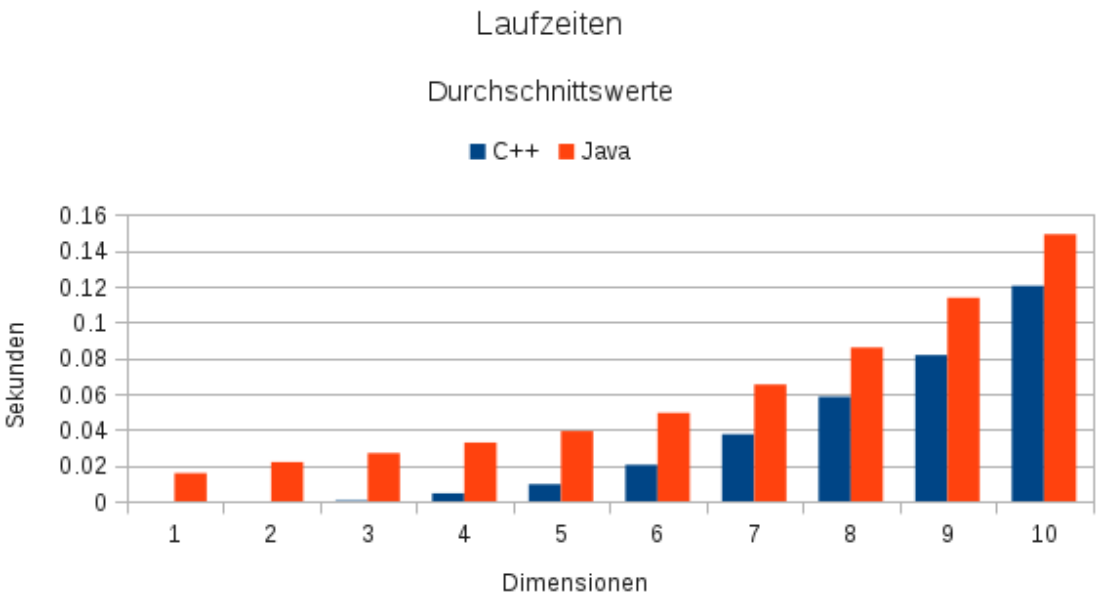

Abbildung 4.2.: Vergleich der Durchschnittslaufzeiten von C++ mit Java

Für die Dimensionen 1 und 2 ist die Laufzeit der C++ nicht messbar, weswegen es mit einer 0.0 gekennzeichnet ist. Für niedrige Dimensionen, bei denen der Rechenaufwand gering und die Anzahl der API-Aufrufe relativ hoch ist, ist der Unterschied extrem. Für Dimension 3 ist die Python-Version um ca. 8000% langsamer als die C++-Version. Aber je höher die Dimension wird und je mehr Rechenaufwand die C++-Bibliothek hat, desto geringer wird der Unterschied zwischen den Versionen. So ist für Dimension 10 die Python-Version nur ca. 230% langsamer als die C++-Version.

Java vs. C++ Auch für Java ist ein messbarer Overhead zu erkennen, was am Diagramm 4.2 zu erkennen ist. Es zeigt für Dimensionen 1 bis 10 die Durchschnittswerte der Laufzeiten für 1000 Durchläufe.

Der Overhead ist, wie zu erwarten war, in Java geringer als in Python. Trotzdem ist der Unterschied von Java zu C++ deutlich zu erkennen. Die Tabelle 4.2 zeigt die zum Diagramm zugehörigen Durchschnittwerte der Laufzeiten.

Hier ist zu sehen, dass bei niedrigen Dimensionen der Overhead ähnlich extrem ist wie bei Python. Java ist auf Dimension 3 um ca. 6500% langsamer als C++. Für Dimension 10 jedoch ist Java nur noch um ca. 24% langsamer und somit schneller als Python. Die rapide Abnahme des Overheads bei höheren Dimensionen liegt vermutlich an den Optimierungen, die der JIT-Kompiler vornimmt.

Szenario: Data-Mining

In diesem Szenario geht es um Data-Mining. Es ist deutlich rechenintensiver als das letzte Szenario und der Rechenaufwand ist verhältnismäßig höher als die Anzahl der API-Aufrufe.

Dimension	C++	Java	Differenz (absolut)	Differenz (relativ)
1	0.00000	0.01564	0.01564	-
2	0.00000	0.02186	0.02186	-
3	0.00041	0.02691	0.02650	6463.41463%
4	0.00439	0.03277	0.02838	646.46925%
5	0.00948	0.03920	0.02972	313.50211%
6	0.02033	0.04934	0.02901	142.69552%
7	0.03741	0.06519	0.02778	74.25822%
8	0.05841	0.08585	0.02744	46.97826%
9	0.08161	0.11364	0.03203	39.24764%
10	0.12036	0.14912	0.02876	23.89498%

Tabelle 4.2.: Durchschnittslaufzeiten in Sekunden über 1000 Durchläufe: C++ gegen Java

Als Datensatz wird ein generischer, sieben-dimensionaler Datensatz, welcher zum Anlernen genutzt wird, verwendet. Dafür wurde ein zwei-dimensionales Gitter verwendet, welches noch zweimal verfeinert wird.

Da die Data-Mining-Algorithmen auf mehreren Prozessoren laufen, wurde hierfür die Wall-Clock verwendet. Die Anzahl der Durchläufe wurde auf 10 beschränkt, weil die Laufzeit eines einzelnen Durchlaufs höher ist.

Python vs. C++ Die Tabelle 4.3 zeigt die Laufzeiten pro Durchlauf für C++ und Python.

Im Durchschnitt ist die Python-Version nur ca. 1.24% langsamer als die C++-Version. Der Overhead ist also sehr gering. Das ist ein Indiz dafür, dass der Overhead mit steigender Problemgröße und größerem Rechenaufwand zu vernachlässigen ist.

Java vs. C++ Für Java sind die Ergebnisse der Laufzeittests unerwartet, wie der Tabelle 4.4 zu entnehmen ist.

In der Tabelle sind keine Zahlenverdreher! Nach den Laufzeittests zu urteilen ist Java tatsächlich um ca. 6.36% schneller als C++ und das konsistent. Es wurden weitere Laufzeittests durchgeführt mit verschiedenen Datensätzen, mit verschiedenen Einstellungen, auf unterschiedlichen Computern. Immer war die Java-Version schneller als die C++-Version und das obwohl beide Versionen die gleiche C++-Bibliothek für die Berechnungen verwenden. Dieses Phänomen entstand nicht durch das verwendete Messverfahren, weil *SG++* während der Trainingsphase interne Zeitmessungen durchführt und auch schon die internen Zeitmessungen den gleichen Effekt vorzeigen. Beide Versionen berechnen tatsächlich das Gleiche. Die resultierenden Gitter wurden nach dem Training verglichen aber es konnte kein Unterschied festgestellt werden. Speicherzugriffsfehler

Durchlauf	C++	Python
1	130.49874100	133.38559699
2	133.54225700	131.66770387
3	133.78209000	132.13581896
4	133.94574900	136.64736295
5	134.12513400	136.10028791
6	133.53400500	136.36841488
7	134.38875200	137.13743401
8	133.47476900	135.67673802
9	133.65263200	136.23624587
10	133.86155800	136.02787995
Durchschnitt	133.48056870	135.13834834

Tabelle 4.3.: Laufzeiten in Sekunden: C++ gegen Python

Durchlauf	C++	Java
1	130.49874100	121.650045404
2	133.54225700	125.124958401
3	133.78209000	125.828040391
4	133.94574900	125.139306179
5	134.12513400	125.388435349
6	133.53400500	125.239889909
7	134.38875200	125.286968906
8	133.47476900	125.006250818
9	133.65263200	125.041518352
10	133.86155800	126.140401897
Durchschnitt	133.48056870	124.9845815606

Tabelle 4.4.: Laufzeiten in Sekunden: C++ gegen Java

können ebenso ausgeschlossen werden und das Verwenden eines Performanceprofilers brachte auch keine Erklärung.

Dieses Ergebnis widerspricht allen Erwartungen und deutet auf irgendeinen Fehler in *SG++* hin, weswegen hier nicht gesagt wird, dass die Java-Version schneller ist als die C++-Version. Eventuell ist es ein Programmierfehler in *SG++*, vielleicht hat es mit Threading zu tun. Es könnte aber auch ein Fehler bei der Generierung des Java-Interfaces sein oder etwas völlig anderes. Was nun die Ursache ist, konnte noch nicht geklärt werden. Dieses Phänomen tritt bei der neuen als auch der alten Version von *SG++* auf.

Dieses Ergebnis wirft aber auch die Frage auf, ob die Ergebnisse der Laufzeittests mit Python korrekt sind.

Fazit

Nach den Laufzeittests zu urteilen, kann es sich für Benutzer von *SG++* lohnen die Python- und Java-Schnittsetllen zu verwenden, vor allem bei großen Problemen die einen hohen Rechenaufwand haben. Aber auch bei kleinen Problem kann Java oder Python anstelle von C++ benutzt werden, obwohl diese verhältnismäßig sehr viel langsamer sind, sind die Laufzeiten immer noch im hundertstel Sekundenbereich.

SG++ scheint also einen guten Kompromiss zwischen effizienten Code und erhöhter Benutzerproduktivität gefunden zu haben.

4.4.4. Auswirkungen auf die C++-Schnittstelle

Programmiersprachen können sich wesentlich voneinander unterscheiden. Manche sind dynamisch typisiert, andere wiederum statisch. Manche erlauben die manuelle Speicherverwaltung und andere haben einen Garbage-Collector. Das Bereitstellen von Programmierschnittstellen zu anderen Sprachen ist daher kein einfaches Unterfangen.

Beim Entwurf muss also große Acht gegeben werden, wie andere Sprachen mit den Typen und Daten umgehen. In diesem Kapitel werden einige Beschränkungen aufgezeigt, die für das Design der neuen *SG++*-Schnittstelle hingenommen werden mussten, damit das Interfacing zwischen den Sprachen auch funktioniert.

Geschachtelte Strukturen

In der für *SG++* verwendeten *SWIG*-Version 2.0 gibt es für C++ keine Unterstützung für geschachtelte Strukturen und Klassen. Konstrukte wie

```
struct MyStruct {
  int a;

  struct {
    int b;
    int c;
  } others;
};
```

werden von *SWIG* ignoriert. *SWIG* bietet dafür Workarounds an, jedoch werden diese in Version 3.0 nicht mehr unterstützt (s.h. *SWIG*-Dokumentation für Version 3.0, Abschnitt *Nested Classes*). Die Workarounds sollten vermieden werden, um die Kompatibilität mit neueren *SWIG*-Versionen zu gewährleisten.

Eine bessere Lösung ist es, die geschachtelte Struktur aus der umschließenden Struktur herauszuziehen, wie etwa

```
struct MyOtherStruct {
  int b;
  int c;
};

struct MyStruct {
  int a;
  MyOtherStruct others;
};
```

Const-Correctness

Wie beschrieben in Kapitel 4.3.1, ist Const-Correctness ein wichtiges Mittel um korrekte Programme zu schreiben. Const-Correctness ist aber ein Konzept der Programmiersprache C++, was Java und Python eben fehlt. *SWIG* garantiert nicht die Const-Correctness für die Zielsprache.

Ein Beispiel aus der Dokumentation [SWI]:

```
const Object * foo();
void bar(Object *);

...

// C++ code
void blah() {
    bar(foo());    // Error: bar discards const
};
```

Hier wird ein konstantes Objekt in eine Funktion übergeben, die ein nicht-konstantes Objekt erwartet. In C++ würde dieser Code nicht kompilieren.

In Python hingegen stellt derselbe Funktionsaufruf kein Problem dar:

```
>>> bar(foo())   # Okay
```

Aufzählungstypen

Bei der Verwendung von Aufzählungstypen in C++ muss beim Design und bei der Implementierung besonders aufgepasst werden. Zum Beispiel können C++-Kompiler zur Kompilierzeit prüfen, ob in einer *switch*-Anweisung alle Elemente eines Aufzählungstyps behandelt wurden.

```
enum Numbers = {ONE, TWO, THREE};

void doSomething(Numbers n)
{
  switch(n) {
    case ONE:
      /* do something */
    case TWO:
      /* do something */
  }
}
```

Im obigen Beispiel könnte ein Kompiler mit den entsprechend gesetzten Flags einen Fehler bzw. eine Warnung ausgeben, weil das Element *THREE* des Aufzählungstyps in der *switch*-Anweisung ausgelassen wurde. Es ist aber nicht genug den dritten Fall hinzuzufügen. Es muss auch ein Standardfall implementiert werden:

```
enum Numbers = {ONE, TWO, THREE};

void doSomething(Numbers n)
{
  switch(n) {
    case ONE:
      /* do something */
    case TWO:
      /* do something */
    case THREE:
      /* do something */
    default:
      /* do something */
  }
}
```

Auch wenn ein C++-Kompiler sicherstellt, dass der Parameter *n* niemals etwas anderes sein kann als was in der Aufzählung *Numbers* spezifiziert wurde, könnte es dennoch Probleme geben, wenn man von einer anderen Programmiersprache diese C++-Methode *doSomething(Numers n)* aufgerufen wird.

Python 2.7 zum Beispiel hat keine nativen Aufzählungstypen. Die obige Methode könnte man aus dem Python-Code mit einem beliebigen Integerwert aufrufen. Der Standardfall muss dann eben die Fälle abfangen, in der *n* keinem der Elemente im Aufzählungstyp *Numbers* entspricht.

Globale Funktionen

In Java gibt es keine globalen Funktionen. Um eine globale C++-Funktion in Java verfügbar zu machen, erstellt SWIG eine neue Java-Klasse, welche denselben Namen hat wie das *SWIG*-Modul in der die Funktion spezifiziert wurde und macht aus dieser Funktion eine Methode der neuen Java-Klasse.

Dadurch können aber Inkonsistenzen zwischen den Sprachen entstehen. Ein Java-Benutzer könnte dadurch Schwierigkeiten bekommen bestimmte Funktionen wiederzufinden, da die neu erzeugte Java-Klasse einen anderen Namen haben könnte als der Namensraum, indem sich die globalen C++-Funktionen befinden. Dieses Problem kann vermieden werden, wenn keine globalen Funktionen in der API benutzt werden. Anstelle dessen können die Funktionen als statische Methoden einer Klasse implementiert werden.

Mehrfachvererbung

Mehrfachvererbung bringt im Gegensatz zur Einfachvererbung einige Probleme mit sich, wie beispielsweise das Diamantproblem. In C++ können die Probleme umgangen werden, Java jedoch verzichtet völlig auf Mehrfachvererbung. Da laut der Dokumentation [SWI] auch *SWIG* keine Mehrfachvererbung für Java nachbilden kann, sollte es gänzlich vermieden werden.

Template-Metaprogrammierung

In C++ bieten Templates ein Mittel zur generischen Programmierung. Jedoch können mit *SWIG* generierte Python- und Java-Schnittstellen das Konzept der generischen Programmierung nicht genutzt werden, da diese Sprachen keine äquivalenten Sprachkonstrukte haben. *SWIG* kann zwar aus Template-Funktionen und -Klassen Schnittstellen generieren, aber nur, wenn in der SWIG-Interfacedatei angegeben wird, mit welchen Templateparametern die Funktion oder Klasse generiert werden soll. In Java und Python stehen dann eben nur diese vorgenerierten Klassen und Funktionen bereit, nicht aber die generischen Versionen.

Zum Beispiel kann folgende Klasse in C++ vom Benutzer mit beliebigen Templateparametern instanziiert werden:

```
template <class T>
class MyClass {
  ...
};
```

Für Java und Python hingegen muss in der *SWIG*-Interfacedatei angegeben werden, mit welchen Parametern die Klasse instanziiert werden soll:

```
%template(MyClass_d) MyClass<double>;
%template(MyClass_i) MyClass<int>;
```

Somit wären nur die beiden Klassen *MyClass_d* und *MyClass_i* für Java und Python verfügbar, nicht jedoch die generische Klasse *MyClass*. Für die C++-Schnittstelle bedeutet es also, dass schon feststehen

sollte, welche Templateparameter verwendet werden sollen, weil generische Klassen und Funktionen mit *SWIG* nicht auf andere Sprachen abgebildet werden können.

Funktionszeiger als Parameter

In C++ können Funktionszeiger als Parameter in Funktionen und Methoden übergeben werden, wie im Beispiel aus der *SWIG*-Dokumentation [SWI]:

```
int binary_op(int a, int b, int (*op)(int,int));
```

Funktionen, die hier als Parameter übergeben werden können, müssen in der *SWIG*-Interfacedatei als Konstanten oder als Callback-Funktionen deklariert werden. So ist es möglich eine vordefinierte Menge an Funktionen aus Python und Java heraus als Parameter in die Methode *binary_op* zu übergeben.

Im Beispiel könnten dann folgende Funktionen benutzt werden:

```
%callback("%s_cb");
int add(int,int);
int sub(int,int);
int mul(int,int);
```

Wenn Benutzer aber Funktionen, die sie selbst definiert haben, übergeben sollen, wird es etwas schwieriger. Für jede unterstützte Sprache müssen eigene Wrapper geschrieben werden um die Funktionen der Quellsprache auf die C++-Funktionen abzubilden.

Um die Wartung der Schnittstelle zu vereinfachen und die Interfacedateien zu vereinheitlichen können anstelle von Funktionszeigern virtuelle Methoden verwendet werden.

Das obige Beispiel könnte dann wie folgt aussehen:

```
class Operation {
  public:
  virtual eval(int a, int b) = 0;
};

int binary_op(int a, int b, Operation* op);
```

Statt dem Funktionszeiger wurde hier also eine Klasse *Operation* eingeführt, welche eine abstrakte *eval*-Methode besitzt. Diese Methode kann dann von Benutzern überschrieben werden. Es ist also möglich, eine Basisklasse in C++ zu definieren und in einer anderen Sprache zu überschreiben. *SWIG* erlaubt es also, das Konzept der Polymorphie über mehrere Sprachen hinweg zu verwenden. Welche Klassen polymorph sein sollen, kann in der Interfacedatei von *SWIG* definiert werden.

Durch die Verwendung des *director*-Konzepts in der Interfacedatei kann *SWIG* die Klasse *Operation* über mehrere Sprachen hinweg als polymorphe Klasse behandeln. Dazu muss das Konzept benutzen zu können, muss es am Anfang der Interfacedatei aktiviert werden wie im folgenden Beispiel:

```
%module(directors="1") module_name
```

Dann kann spezifiziert werden, welche Klassen polymorph sein sollen.

```
%feature("director") Operation;
```

Es ist ebenso möglich alle Klassen in einer Interfacedatei auf einmal als polymorph zu deklarieren oder aber auch nur einzelne Methoden. Somit könnte man in Java oder Python von der Klasse *Operation* ableiten, die *eval*-Methode redefinieren und als Argument in die Funktion *binary_op* übergeben. Für Java und Python funktioniert diese Strategie recht zuverlässig, ohne zusätzlichen Wrapper schreiben zu müssen. Die Wartung und das Hinzufügen von neuen Sprachen würde somit vereinfacht werden.

5. Refactoring der Bibliothek

In diesem Kapitel wird gezeigt, wie die *SG++* umgestaltet wurde und welche Verbesserungen die Änderungen mit sich bringen. Anhand von ausgewählten Beispielen wird gezeigt, welche Techniken wie eingesetzt werden und wie die Architektur der Software geändert wurde. Das Hauptaugenmerk wurde dabei auf die Schnittstelle, welche Benutzer der Bibliothek verwenden sollen, gelegt. Aber es wird auch auf einige interne Implementierungsdetails eingegangen.

Dieses Kapitel konzentriert sich hauptsächlich auf die Module *Base*, *Solver* und *Datadriven*. Die restlichen Module wurden dahin gehend angepasst, sodass sie kompilieren, weil sich die Änderungen in den drei erwähnten Modulen durch die restlichen durchziehen. Viele Änderungen, die in den Modulen *Base*, *Solver* und *Datadriven* durchgeführt wurden, könnten auch auf die restlichen Module angewendet werden. Das hätte jedoch den Rahmen dieser Arbeit gesprengt und wurden deswegen nicht genauer betrachtet.

5.1. Allgemein

Bevor auf ein spezielles Modul eingegangen wird, werden einige allgemeine Probleme in *SG++* besprochen, die in mehreren Modulen auftreten. Beispielhaft wurden einige Codestellen ausgesucht, um die Probleme zu erläutern und zu zeigen, welche Änderungen durchgeführt wurden um die Probleme anzugehen.

5.1.1. Const-Correctness

Problembeschreibung

Methoden und Parameter, welche eigentlich konstant sein sollten, wurden nicht als *const* deklariert.

- Im folgenden Beispiel wurde der Parameter im Kopierkonstruktor der Klasse *BoundingBox* nicht als konstant deklariert:

```
BoundingBox(BoundingBox& copyBoundingBox);
```

- In der gleichen Klasse findet man Getter-Methoden, die nicht als konstant deklariert wurden, wie man es ansonsten erwarten würde:

```
size_t getDimensions();
```

- Gleiche Probleme findet man ebenso in der Klasse *Stretching*. Der folgenden Methode ist nicht anzusehen, dass sie konstant sein könnte:

```
void logXform(Stretching1D& str1D, size_t dimension);
```

  Auf den ersten Blick ist eben nicht offensichtlich, dass diese Methode keine Seiteneffekte auslöst.

- Es fehlen einige konstante Methodenüberladungen. Es ist es oft hilfreich einige Methoden zu überladen und diese als konstant zu deklarieren, damit sie von konstanten Objekten benutzt werden können.

  Zum Beispiel hat die Klasse *DataMatrix* folgende Operatorüberladung:

```
inline double& operator[](size_t i);
```

  Jedoch kann dieser Operator nicht für konstante Instanzen der Klasse *DataMatrix* benutzt werden. Somit könnte verhindert werden, dass im Benutzercode die *Const-Correctness* nicht eingehalten werden kann.

Alle Beispiele verstoßen gegen die Richtlinie deskriptive Methodensignaturen zu verwenden. Die letzten Beispiele verstoßen zusätzlich gegen die Richtlinie konstante Methoden zu verwenden.

Die Verstöße gegen die Richtlinien können dazu führen, dass Benutzer einen Blick in die Implementierung werfen, um sicherzugehen, dass keine Seiteneffekte auftreten. Außerdem leidet auch die Erlernbarkeit, da ein Benutzer sich nun mit mehr Code vertraut machen muss, als es nötig wäre.

Verbesserungen

Alle Methoden und Parameter, die unveränderbar sein sollen, wurden als *const* deklariert.

- Im Kopierkonstruktor ist nun klar erkennbar, dass der Parameter *copyBondingBox* nicht geändert wird:

```
BoundingBox(const BoundingBox& copyBoundingBox);
```

- Der Signatur der Getter-Methode ist nun klar anzusehen, dass diese keine Seiteneffekte auslöst:

```
size_t getDimensions() const;
```

- Dasselbe gilt für die Methode *logXform*:

```
void logXform(Stretching1D& str1D, size_t dimension) const;
```

- Es wurde folgende Methode zur Klasse *DataMatrix* hinzugefügt:

```
inline const double& operator[](size_t i) const;
```

  Jetzt kann dieser Operator auch für konstante Objekte genutzt werden. Das erleichtert Benutzern, in ihrem eigenen Code die *Const-Correctness* einzuhalten.

5.1.2. Fehlermeldungen

Problembeschreibung

Im Projekt wird der Richtlinie der guten Ausnahmebehandlungen nicht gefolgt. In der Klasse *AlgorithmAdaBoostBase* gibt es zum Beispiel die folgenden zwei Meldungen:

```
throw new sg::base::operation_exception("AlgorithmAdaBoostBase::doAdaBoostR2 : An unknown loss
    function type was specified!");
```

```
throw new sg::base::operation_exception("AlgorithmAdaboost : Only 1 or 2 or 3 are supported
    gridType(1 = Linear Grid, 2 = LinearBoundary Grid, 3 = ModLinear Grid)!");
```

- Die beiden Meldungen sind inkonsistent. Während in der ersten Meldung die Klasse und die Methode angezeigt wird, in der der Fehler erscheint, ist in der zweiten Meldung nur der inkorrekte Name der Klasse zu sehen (*AlgorithmAdaboost* anstelle von *AlgorithmAdaboostBase*).
- Die erste Meldung ist auch unvollständig. Es wird zwar beschrieben, welche Ausnahme geworfen wurde, jedoch fehlt der Meldung, wie ein Benutzer die Ausnahme vermeiden könnte.
- Die zweite Meldung ist fehlerhaft. Anstelle des Typs *LinearBoundary* sollte dort *LinearTrapezoidBoundary* stehen.
- Die Ausnahmen werden dynamisch allokiert und geworfen. Wenn die Ausnahmen gefangen werden, dann müssen diese Objekte auch wieder gelöscht werden. Das bedeutet zusätzliche Schreibarbeit für Benutzer und verursacht Boilerplatecode.

Verbesserungen

- Die erste Meldung wurde ergänzt und zeigt nun an, welche Typen erlaubt sind.
- Die zweite Meldung wurde der ersten angepasst, indem der Methodenname hinzugefügt wurde. Der Klassenname und der Gittertyp wurde korrigiert.
- Die Ausnahmen werden nicht mehr dynamisch allokiert.

Die erste Meldung beschreibt nun, welche Typen gewählt werden können:

```
throw sg::base::operation_exception("AlgorithmAdaBoostBase::doAdaBoostR2 : An unknown loss
    function type was specified! The only supported loss functions types are linear, square and
    exponential.");
```

Die Meldungen sind jetzt auch konsistent und Benutzer müssen sich nicht an verschiedene Formate gewöhnen:

```
throw sg::base::operation_exception("AlgorithmAdaboostBase::doRealAdaBoost : An unknown grid type
    was specified. The only supported grid types are Linear, LinearTrapezoidBoundary and
    ModLinear.");
```

Alle Fehlermeldungen zeigen nun an, wo der Fehler ist und wie ein Benutzer ihn vermeiden könnte. Ein weiterer Vorteil ist, dass die Ausnahmen nun per Referenz gefangen werden können. Benutzer müssen sich nicht um das Löschen der Ausnahmeobjekte kümmern.

5.1.3. Erzeugung von Operatoren

Problembeschreibung

- Die Erzeugung von Operatoren wie zum Beispiel *OperationHierarchisation* wird durch Fabrikmethoden realisiert. Das Problem dabei ist jedoch, dass globale Funktionen benutzt wurden und dadurch die Java-Schnittstelle inkonsistent zur C++-Schnittstelle wird, wie es in Kapitel 4.4.4 erklärt wurde. So sind die Funktionen in C++ und in Python unter dem Namensraum *op_factory* zu finden, während in Java diese in *jsgpp* sind.

  In C++ wird die Methode folgendermaßen aufgerufen:

```
op_factory::createOperationHierarchisation(*grid);
```

  Aber in Java hingegen ist die Methode in der Klasse *jsgpp* zu finden:

```
jsgpp.createOperationHierarchisation(grid);
```

- Der Namensraum *op_factory* verstößt gegen die Richtlinie der guten Namensgebung, da es eine Abkürzung benutzt. Dasselbe gilt für Sourcefiles wie *BaseOpFactory* und andere Dateien nach dem gleichen Namensschema. Diese Dateinamen passen nicht zu den restlichen Quelldateien, welche die Abkürzung *Op* ausschreiben, wie zum Beispiel *OperationHierarchisation*.

Verbesserungen

- Anstelle des Namensraums *op_factory* wurde eine entsprechende Klasse eingeführt und die globalen Funktionen als statische Methoden dieser Klasse deklariert.
- Die Quelldateien mit der Abkürzung *Op* wurden umbenannt, indem die Abkürzung einfach ausgeschrieben wurde.

Die Benutzung der Fabrikmethoden ist nun über die Sprachen hinweg konsistent. Um zum Beispiel grundlegende Operationen auf dem Gitter zu erzeugen, kann jetzt in C++ folgender Code verwendet werden:

```
BaseOperationsFactory::createOperationHierarchisation(*grid);
```

In Java sieht der entsprechende Code dann wie folgt aus:

```
BaseOperationsFactory.createOperationHierarchisation(grid);
```

Die Namensgebung der Quelldateien ist nun ebenso konsistent und entsprechen der Richtlinie.

5.2. Modul: Base

5.2.1. Ausnahmebehandlung in der Klasse Stretching

Problembeschreibung

Die Klasse *Stretching* leitet von *BoundingBox* ab implementiert die Methode *getDiscreteVector*, welche folgendermaßen definiert ist:

```
std::vector<double>* Stretching::getDiscreteVector(bool bSort) {
  std::vector<double>* vec;
  int elemsToRead = 0;

  if (*stretchingMode == "discrete") {
    vec = new std::vector<double>[nDim];

    ...

    return vec;
  } else {
    std::cout << "Unsupported stretchingMode\n";
    return 0;
  }
}
```

Hier wird zuerst abgefragt, ob für das Objekt ein bestimmter Modus gewählt wurde. Falls nicht, wird eine Fehlermeldung ausgegeben und ein Nullzeiger zurückgegeben. Der Benutzer muss also im eigenen Code abfragen, ob ein Nullzeiger zurückgegeben wurde und den Fall entsprechend behandeln.

Verbesserungen

Es wurde die Klasse *configuration_exception* für Ausnahmen neu angelegt. Diese Klasse soll geworfen werden, wenn falsche Konfigurationen und Einstellungen verwendet wurden. Die Meldung und die Rückgabe des Nullzeigers wurde durch das Auslösen einer Ausnahme ersetzt:

```
std::vector<double>* Stretching::getDiscreteVector(bool bSort) const {
  std::vector<double>* vec;
  int elemsToRead = 0;

  if (*stretchingMode == "discrete") {
    vec = new std::vector<double>[nDim];

    ...

    return vec;
  } else {
    throw configuration_exception("Stretching::getDiscreteVector : Unsupported stretchingMode.
        Only supported mode is discrete.");
  }
}
```

Benutzer können nun sichergehen, dass kein Nullzeiger zurückgegeben wird und müssen dafür keinen zusätzlichen Code schreiben. Die angegebene Fehlermeldung ist deskriptiv und gibt an, wie der Fehler zu beheben ist.

5.2.2. Konfigurationen

Problembeschreibung

- Die Strukturen *AdaptivityConfiguration* und *RegularGridConfiguration* sind zusammen mit der Klasse *Grid* in der Datei *Grid.hpp* definiert, obwohl die *Grid*-Klasse die beiden Strukturen nicht benutzt. Das ist ein Verstoß gegen die Modularisierungsrichtlinie.
- Weiterhin verstoßen Membervariablen der Strukturen gegen die Richtlinie der guten Namensgebung. Das folgende Codefragment illustriert das Problem:

```
struct AdpativityConfiguration {
  /// number of refinements
  size_t numRefinements_;
  /// refinement threshold for surpluses
  double threshold_;
  /// refinement type: false: classic, true: maxLevel
  bool maxLevelType_;
  /// max. number of points to be refined
  size_t noPoints_;
  /// max. percent of points to be refined
  double percent_;
};
```

  Die Variablen haben alle einen Unterstrich als Suffix, was unnötig ist und leicht vom Benutzer vergessen werden könnte. Außerdem ist die Verwendung der Abkürzung für *number* in *numRefinements_* und *noPoints_* inkonsistent.
- Die Initialisierung der Strukturen könnte erleichtert werden, um somit den Code der API-Benutzer klein zu halten. Zum Vermeiden von Fehlern durch die Benutzer, sollte die API sicherstellen, dass kein Feld uninitialisiert verwendet wird.
- Name der Struktur einen Schreibfehler, was Benutzer verwirren könnte. Es wurden die Buchstaben ‚d' und ‚p' vertauscht.

Verbesserungen

- Beide Strukturen wurden in die neu angelegte Headerdatei *GridConfigurations.hpp* verschoben.
- Der Unterstrich für die Variablen wurde einfach entfernt. Auch die inkonsistenten Abkürzungen wurden ersetzt.
- Der Schreibfehler im Namen der Struktur wurde ausgebessert.

- Konstruktoren wurden eingeführt, damit man eine weitere Möglichkeit hat, die Strukturen zu initialisieren. Zusätzlich wurde zu jedem Parameter ein Standardwert festgelegt. Für Konstruktorparameter und Membervariablen wurden dieselben Namen gewählt. Für die Initialisierung stellt das kein Problem dar, da C++-Compiler den Unterschied automatisch erkennen.

Die resultierende Struktur sieht wie folgt aus:

```
struct AdaptivityConfiguration
{
 AdaptivityConfiguration(
    size_t numberOfRefinements = 15,
    double threshold    = -1.0,
    bool maxLevelType   = false,
    size_t maxPointCount = 100,
    double percent      = 100.0) :
  numberOfRefinements(numberOfRefinements),
  threshold(threshold),
  maxLevelType(maxLevelType),
  maxPointCount(maxPointCount),
  percent(percent)
 {}

 /// number of refinements
 size_t numberOfRefinements;
 /// refinement threshold for surpluses
 double threshold;
 /// refinement type: false: classic, true: maxLevel
 bool maxLevelType;
 /// max. number of points to be refined
 size_t maxPointCount;
 /// max. percent of points to be refined
 double percent;
};
```

Bei jeder Instanziierung dieser Struktur sind nun immer alle Variablen initialisiert. Somit ist es nicht mehr möglich für den Benutzer zu vergessen Variablen zu setzen, was zu seltsamen Ergebnissen in der Berechnung führen kann. Ein weiterer Vorteil ist, dass der Benutzer nun nur die Werte ändern muss, die vom Standard abweichen sollen, was zu weniger Code führt.

5.2.3. Gittertypen

SG++ enthält einen Aufzählungstyp mit dem Namen *GridType*, welcher alle erhältlichen Gittertypen auflistet. Gitter werden hierbei nach Basis und nach dem Rand des Gebiets unterschieden.

```
enum GridType {
 Linear,
 LinearStretched,
 ... // usw. // ...
 SquareRoot,
 TruncatedTrapezoid
};
```

Problembeschreibung

- Das erste Problem ist, dass die Liste nicht vollständig ist. Es fehlen die beiden Elemente *LinearStencil* und *ModLinearStencil*.
- Die Benutzung des Aufzählungstyps ist inkonsequent. Zum Beispiel hat die Klasse *Grid* folgende Methode:

```
virtual const char* getType() = 0;
```

  Die Methode gibt den Typ eines Gitterobjekts zurück. Anstelle von einem *GridType* gibt es jedoch einen C-String zurück. Das macht die Benutzung der Methode fehleranfällig.

Verbesserungen

- Es wurde ein neuer Aufzählungstyp eingeführt. Der Typ *Grids::Type* ist eine Aufzählung, die alle möglichen Gittertypen beinhaltet. Sie ersetzt die Aufzählung *GridType*, welche in *Grid.hpp* zu finden war:

```
struct Grids {
  enum Type {
    Linear,
    LinearStretched,
    ... // usw. // ...
    SquareRoot,
    TruncatedTrapezoid,
    LinearStencil,
    ModLinearStencil
  };
};
```

  Außerdem wurden die zwei fehlenden Elemente *LinearStencil* und *ModLinearStencil* ergänzt. Die Reihenfolge der Elemente wurde beibehalten.

- Alle Methoden, die einen String benutzen, um Gittertypen zu unterscheiden, wurden umgestellt. Anstelle der Strings wird nun der Typ *Grids::Type* verwendet.

  Zum Beispiel sieht die Methode *getType()* in der Klasse *Grid* nun folgendermaßen aus:

```
virtual Grids::Type getType() const = 0;
```

- Zusätzlich wurden zwei weitere Typen eingeführt. Die Aufzählungen *StretchedGrids::Type* und *BasicGrids::Type* sollen Subtypen von *Grids::Type* darstellen.

  Da C++ keine Subtypen enthält, wurde ein Trick angewendet. Hier zum Beispiel der Typ *StretchedGrids*:

```
struct StretchedGrids{
  enum Type {
    LinearStretched          = Grids::LinearStretched,
    LinearStretchedTrapezoidBoundary = Grids::LinearStretchedTrapezoidBoundary
  };
};
```

Diese beinhaltet einen Teil der Elemente, die auch in *Grids::Type* zu finden ist, nämlich alle Gittertypen, die auf einem gestreckten Gebiet arbeiten. Mithilfe dieses Tricks können in C++ Subtypen für Aufzählungen definiert werden. Um Namenskonflikte zwischen den Aufzählungen zu vermeiden, wurden alle in eigene Strukturen gepackt. Den Elementen dieser Subtypen wurden Nummern zugewiesen, welche den Nummern in *Grids::Type* entsprechen. Somit können Elemente der Subtypen leicht in Elemente vom *Grids::Type* umgewandelt werden.

In Java können diese Aufzählungen durch *SWIG* in native Java-Aufzählungstypen umgewandelt werden. Diese Typen bieten dann die Methode *ordinal()* an, welche dann die Ordnungszahl eines Elements der Aufzählung zurückgibt. Diese Nummerierung muss aber nicht mit der ursprünglichen Nummerierung in C++ übereinstimmen! Die Umwandlung von einem Subtyp in einen Basistyp wäre auf der Java-Ebene also nicht möglich. Um die gleiche Nummerierung wie in C++ zu erhalten, sollte die Methode *swigValue()* benutzt werden. Die Generierung von nativen Java-Aufzählungstypen kann durch *SWIG* auch unterbunden werden, sodass die Methode *ordinal()* nicht benutzt werden kann, sondern nur *swigValue()*. So würde eine potenzielle Fehlerquelle beseitigt werden.

Durch diese Änderungen wurde die Abfrage, welchen Typ ein Gitter besitzt, vereinfacht. Folgendes Codefragment in der Datei *BaseOpFactory.cpp*

```
if (strcmp(grid.getType(), "linear") == 0)
```

konnte umgeschrieben werden zu

```
if (grid.getType() == sg::base::Grids::Linear)
```

Schreibfehler werden nun durch den Compiler abgefangen. Benutzer müssen sich auch nicht mehr merken, wie genau der Gittertyp heißt, also ob es nun *linear* oder *Linear* ist. Außerdem sind Aufzählungen platzsparender als Strings, weil diese intern als Integerwert dargestellt werden.

Dieses Design hat auch den Vorteil, dass leicht neue Subtypen hinzugefügt werden können, was den Code erweiterbar macht. Ein weiterer positiver Nebeneffekt ist, dass die Funktion *strcmp* aus *<cstring>* nicht mehr gebraucht wird und somit Abhängigkeiten reduziert wurden.

Das Design bring aber auch einen Nachteil mit sich. Es wurde Codeduplikation verwendet, um eine höhere Flexibilität und Sicherheit zu erreichen. Der doppelte Code ist jedoch beschränkt auf nur eine Headerdatei, weswegen es übersichtlich bleibt. Außerdem wurden Kommentare hinzugefügt, um Benutzern darauf aufmerksam zu machen.

Diese Unterscheidung von Typen dient zum Gewährleisten einer gewissen Typsicherheit. Wenn man z.B. eine Struktur vom Typ *GridStretchingConfiguration* instanziieren will, sollen nur die entsprechenden Gittertypen gewählt werden können. Durch die Aufteilung in Subtypen kann der Compiler prüfen, ob der passende Gittertyp gewählt wurde.

Den genauen Zweck dieser Unterscheidung wird im nächsten Punkt *Gittergenerierung* erläutert.

5.2.4. Gittergenerierung

Die Gittergenerierung ist der zentrale Bestandteil der Bibliothek. Es wird von allen weiteren Modulen verwendet, kann aber auch als selbstständiges Modul verwendet werden. Daher ist es wichtig, dass Benutzern dessen Verwendung erleichtert wird.

Problembeschreibung

Der folgende Code zeigt die Generierung eines regulären, stückweise bi-linearen Gitters der Dimension zwei.

```
// create a two-dimensional piecewise bi-linear grid
int dim = 2;
Grid* grid = Grid::createLinearGrid(dim);

// create regular grid, level 3
int level = 3;
GridGenerator* gridGen = grid->createGridGenerator();
gridGen->regular(level);
```

- Um ein Gitter zu erzeugen, muss ein zuerst ein *Grid*-Objekt erzeugt werden, dann daraus ein *GridGenerator*-Objekt erstellt werden, um daraus wiederum das Gitter zu initialisieren.

  Dieser Ablauf macht das Benutzen fehleranfällig. Wenn vergessen wird ein *GridGenerator*-Objekt zu erzeugen, um dann das Gitter zu initialisieren, stürzt das Programm ab, genauso wie das doppelte Aufrufen der Methode *regular(level)*. Das verstößt gegen die Richtlinie der guten Ausnahmebehandlung.

  Weiterhin ist anzumerken, dass ein *GridGenerator*-Objekt die Daten des *Grid*-Objekts, aus dem es erzeugt wurde, ändert. Diese Seiteneffekte verstoßen gegen das Prinzip der geringsten Überraschung.

- Die Benutzung ist nicht intuitiv. Benutzern könnte nicht klar sein, welche Rolle die Klasse *GridGenerator* spielt und warum diese aus der Klasse *Grid* erzeugt werden muss, um ein reguläres Gitter erstellen.

  Die API scheint hier verkehrt herum zu funktionieren. Normalerweise würde man erwarten, dass ein *Grid*-Objekt aus einem *GridGenerator*-Objekt erzeugt wird, wie es die Namen suggerieren. Das verstößt klar gegen die Richtlinie der guten Namensgebung und ist wenig prägnant.

- Die API lässt Benutzer unnötigen Boilerplatecode schreiben und führt zu unleserlichem Benutzercode, welches gegen die Richtlinie der prägnanten API verstößt. Typischerweise kopieren Benutzer solchen Code in ihre eigenen Projekte anstelle es selbst auszuschreiben. Jedoch könnten sich durch Copy&Paste Fehler in den Benutzercode einschleichen.

Verbesserungen

Die statischen Methoden der Klasse *Grid* zum Erzeugen eines neuen *Grid*-Objekts, wie zum Beispiel

```
static Grid* createLinearGridStencil(size_t dim);
```

wurden entfernt.

Es wurde eine neue Klasse *GridFactory* eingeführt, welche die Erzeugung der Gitter übernimmt. Dafür wurden Strukturen in *GridConfigurations.hpp* angelegt. Diese Strukturen speichern Konfigurationen. Mit diesen Konfigurationen kann der Benutzer steuern, welche Art von Gitter erzeugt werden soll.

Die Struktur *RegularGridConfiguration* existierte in der alten *SG++*-Version bereits in der Datei *Grid.hpp*. Sie wurde nur im *Datadriven*-Modul als Parameter in einigen Methoden gebraucht. Die Struktur wurde in *GridConfigurations.hpp* verschoben, weil sie nie in *Grid.hpp* benutzt wurde. Außerdem wurde ein Konstruktor hinzugefügt, um Standardwerte setzen zu können. In der neuen Version wird diese Struktur noch zusätzlich dafür verwendet, um reguläre Gitter zu erzeugen.

```
struct RegularGridConfiguration
{
 RegularGridConfiguration(
    Grids::Type type  = Grids::Linear,
    size_t dim        = 2,
    int level         = 3) :
  type(type),
  dim(dim),
  level(level)
 {}

 /// Grid Type, see enum
 Grids::Type type;

 /// number of dimensions
 size_t dim;

 /// number of levels
 int level;
};
```

RegularGridConfiguration stellt die Basis für die Struktur *GridConfiguration*. Sie wird dazu benutzt, um beliebige Gitter erzeugen zu können.

```
struct GridConfiguration : public RegularGridConfiguration
{
 GridConfiguration(
    Grids::Types type       = Grids::Linear,
    GridSparsity gridSparsity = Regular,
    size_t dim              = 2,
    int level               = 3,
    size_t degree           = 3,
    size_t levelUser        = 0) :
  RegularGridConfiguration(type, dim, level),
  gridSparsity(gridSparsity)
 {
```

```
    polynomial.degree = degree;
    truncatedGrid.levelUser = levelUser;
  }

  // See enum
  GridSparsity gridSparsity;

  /// Only needed for grids with polynomial basis
  PolynomialBasisConfiguration polynomial;

  /// Only needed on truncated grids
  TruncatedGridConfiguration truncatedGrid;
};
```

Die Variablen *polynomial* und *truncatedGrid* sind weitere Strukturen, welche Variablen speichern, die nur für die Generierung von Gittern mit polynomialen Basisfunktionen, beziehungsweise nur für Gitter mit beschränkten Leveln benutzt werden. Da dies eben nicht für alle Gittertypen zutrifft, wurden sie in eigene Strukturen gekapselt. Das Kapseln dieser Variablen soll Benutzern klar machen, in welchen Fällen diese gesetzt werden sollen. Anstelle eigene Strukturen zu definieren, hätten hier auch geschachtelte Strukturen benutzt werden können. Da *SWIG* das aber nicht erlaubt, wie in Kapitel 4.4.4 beschrieben wurde, wurde deswegen dieser Ansatz gewählt.

Es wurde eine weitere Struktur mit dem Namen *DomainGridConfiguration* erstellt. Sie ähnelt der Struktur *GridConfiguration* bis auf die Templateparametrisierung.

```
template <class GridType, class Domain>
struct DomainGridConfiguration
{
  DomainGridConfiguration (
     const Domain& domain,
     GridType type,
     GridSparsity gridSparsity = Regular,
     int level               = 3,
     size_t degree           = 3,
     size_t levelUser        = 0) :
    domain(domain),
    type(type),
    level(level),
    gridSparsity(gridSparsity)
  {
    polynomial.degree = degree;
    truncatedGrid.levelUser = levelUser;
  }

  const Domain& domain;

  /// Grid Type, see enum
  GridType type;

  /// number of levels
  int level;

  /// See enum
  GridSparsityConfig gridSparsity;
```

```
  /// Only needed for grids with polynomial basis
  PolynomialBasisConfiguration polynomial;

  /// Only needed on truncated grids
  TruncatedGridConfiguration truncatedGrid;
};
```

Die Membervariable *dim* der Struktur *GridConfiguration* wurde ersetzt durch den parametrisierten Typ *Domain* und der Typ der Variable *type* wurde ebenso parametrisiert.

Diese Struktur soll dazu dienen, Gitterkonfigurationen erstellen zu können, indem man anstelle der Dimension ein Gebiet bestimmt, z.B. durch eine Boundingbox. Deswegen wurden auch zwei Typdefinitionen eingeführt.

Zum Erstellen eines Gitters mit einer einfachen Boundingbox:

```
typedef DomainGridConfiguration<BasicGrids::Type, BoundingBox> BoundingBoxGridConfiguration;
```

Und zum Erstellen eines Gitters mit einer gestreckten Boundingbox:

```
typedef DomainGridConfiguration<StretchedGrids::Types, Stretching> StretchedGridConfiguration;
```

BoundingBoxGridConfiguration und *StretchedGridConfiguration* nehmen verschiedene Gittertypen an. Während *StretchedGridConfiguration* den Gittertyp *StretchedGrids::Type* annimmt, speichert *BoundingBoxGridConfiguration* den Typ *BasicGrids::Type* und *GridConfiguration* zusammen mit *RegularGridConfiguration* den Gittertyp *Grids::Type*. Somit wird es Benutzern erleichtert Gitter zu generieren, da man gleich sieht, welche Konfigurationsoptionen zueinander kompatibel sind, was auch zu weniger Fehlern auf der Benutzerseite führt.

Im gesamten Projekt werden dann auch nur die Typdefinitionen *BoundingBoxGridConfiguration* und *StretchedGridConfiguration* anstelle des Basistyps *DomainGridConfiguration* benutzt.

Dieser Ansatz hat den Nachteil, dass keine Standardparameter für *domain* und *type* vergeben werden können. Eine Alternative wäre gewesen ein Flag zu benutzen, anstelle den Aufzählungstyp für Gittertypen aufzutrennen, um zu unterscheiden ob die Boundingbox gestreckt ist oder nicht. Das hätte aber den Nachteil, dass Benutzer Konfigurationen erstellen könnten, für die die Typen der Variablen *domain* und *type* nicht zusammenpassen würden.

Zum Beispiel hat das Gitter *ModPolyGrid* keinen Konstruktor, welchen als Parameter den Typ *Stretching* annimmt. Somit ist es nicht möglich, so ein Gitter mit einem gestreckten Gebiet zu instanziieren. Ein Fehler würde erst bei dem Versuch der Instanziierung gemeldet werden, was gegen die Richtlinie der guten Ausnahmebehandlung verstößt, da der Fehler nicht früh genug gemeldet wurde. Außerdem würde die Benutzung erschwert werden. Wenn ein Benutzer beispielsweise die folgende Methode der Klasse *Grid* aufrufen will, um den Typ des Gitters abzufragen:

```
virtual Grids::Type getType() const = 0;
```

Dann müsste der Benutzer noch zusätzlich eine weitere Methode der Klasse aufrufen, um zu prüfen, ob das Gebiet gestreckt wurde oder nicht. Das lässt Benutzer mehr Code schreiben und kann zu Fehlern führen.

Der gewählte Ansatz hat auch Vorteile für die Implementierung der Fabrikmethoden. Anstelle von *if*-Anweisungen wurden *switch*-Anweisungen zum Abarbeiten aller möglichen Gittertypen benutzt. Damit kann der Compiler zur Kompilierzeit prüfen, ob auch alle Elemente benutzt werden. So können keine Gittertypen vergessen werden, was es auch einfacher macht neue Gittertypen hinzuzufügen.

Die Deklaration der Klasse *GridFactory* sieht nach den Änderungen wie folgt aus:

```
/**
 * Grid factory for generating grids.
 */
class GridFactory {
  public:
    /**
     * Creates a grid from the given configuration.
     *
     * @param config A configuration for regular grids.
     *
     * @return The generated grid, according to the given configuration.
     */
    static Grid* createGrid(const GridConfiguration& config);

    /**
     * Creates only a regular grid from the given configuration.
     *
     * @param config A configuration for all grids.
     *
     * @return The generated grid, according to the given configuration.
     */
    static Grid* createGrid(const RegularGridConfiguration& config);

    /**
     * Creates a grid from the given configuration using a predefined
     * bounding box.
     *
     * @param config A configuration with a bounding box.
     *
     * @return The generated grid, according to the given configuration.
     */
    static Grid* createGrid(const BoundingBoxGridConfiguration& config);

    /**
     * Creates a grid from the given configuration using a predefined
     * stretched bounding box.
     *
     * @param config A configuration with a stretched bounding box.
     *
     * @return The generated grid, according to the given configuration.
     */
    static Grid* createGrid(const StretchedGridConfiguration& config);

  private:
    GridFactory();
};
```

Die Klasse besitzt nun nur diese eine überladene Methode *createGrid*, welche die verschiedenen Konfigurationen als Parameter annimmt und ein Objekt der abstrakten Klasse *Grid* zurückgibt.

Die Methoden sind alle statisch und der Standardkonstruktor ist privat, um Benutzer daran zu hindern ein Objekt dieser Klasse zu instanziieren. Somit wird klar vorgegeben, wie die Klasse zu benutzen ist, was Benutzern Orientierung gibt.

Anstelle von Konfigurationen hätte man den *createGrid*-Methoden auch alle benötigten Parameter direkt übergeben können. Es müssten jedoch bis zu sechs Parameter übergeben werden, was die Parameterliste zu lange machen würde.

Ein weiterer Vorteil ist, dass bei der Erstellung von Konfigurationen Standardwerte vergeben werden und ein Benutzer nur die Werte abändern muss, welche vom Standard abweichen. Standardparameter für Methoden in C++ hingegen können nicht einzeln abgeändert werden. Wenn man einen Parameter setzen will, müssen alle vorherigen Parameter in der Parameterliste ebenso gesetzt werden, auch wenn diese nicht vom Standard abweichen sollen. Das erzeugt unnötigen Boilerplatecode.

Weiterhin ist anzumerken, dass die Implementierung dieser Klasse Hilfsmethoden benötigt. Die Hilfsmethoden wurden jedoch in eine eigene *GridFactoryImplementation*-Klasse ausgelagert. Diese Trennung von internen Hilfsmethoden und dem Interface von *GridFactory* hilft, die Schnittstelle übersichtlich zu halten.Somit sieht ein Benutzer beim Blick in die Headerdatei nur die Methoden, die relevant sind.

Die Gittererzeugung wurde vereinfacht:

```
// create a two-dimensional piecewise bi-linear grid, level 3
GridConfiguration gridConfig(Grids::Linear);
gridConfig.dim = 2;
gridConfig.level = 3;

Grid* grid = GridFactory::createGrid(gridConfig);
```

Es muss nur noch die richtige Konfiguration gewählt werden und eine der Methoden der Fabrik aufgerufen werden. Dieses Design verhindert auch, dass ein Benutzer die API falsch benutzt und zum Abstürzen bringt.

5.2.5. Gittergenerator

Problembeschreibung

- Die Klassen *Grid* und *GridGenerator* hängen zu eng zusammen. Sie greifen auf die gleichen Daten zu und können zu unvorhergesehenen Seiteneffekten und Fehlern führen.

  Zum Beispiel hat die Klasse *GridGenerator* die Methode *regular* um ein reguläres Gitter zu erzeugen. Wenn diese doppelt aufgerufen wird, wird eine Ausnahme geworfen. Eine gute API sollte verhindern, dass Benutzer solche Methoden falsch benutzen könnten.

- Die Klasse *GridGenerator* ist nicht intuitiv zu verwenden. Ihr Nutzen könnte Benutzern nicht klar werden, wie Beschrieben im Kapitel 5.2.4 über Gittergenerierung. Sie enthält Methoden, die man in der Klasse *Grid* erwarten würde, wie zum Beispiel das Verfeinern und Vergröbern des Gitters.

- Benutzer müssen um Objekte zweier unterschiedlicher Klassen kümmern. Wenn ein Benutzer zum Beispiel ein Gitter verfeinern möchte, muss entweder ein *GridGenerator*-Objekt zu dem zugehörigen *Grid*-Objekt mitgezogen werden oder es wird immer wieder ein neues *GridGenerator*-Objekt aus dem *Grid*-Objekt erzeugt, wie z.B. in der Klasse *PDESolver* zu sehen ist:

  ```
  myGrid->createGridGenerator()->refine(myRefineFunc);
  ```

  Hier wird extra ein Objekt der Klasse *GridGenerator* erzeugt, um das Gitter zu verfeinern. Das erzeugt Overhead und ist nicht intuitiv.

- Objekte von Typ *GridGenerator* müssen von Benutzern wieder gelöscht werden, was unnötigen Boilerplatecode erzeugt. Außerdem ist die Schnittstelle der alten *SG++*-Version anfällig für Speicherlecks, weil das Löschen durch Benutzer leicht vergessen werden kann.

Verbesserungen

Der Klasse *Grid* wurde ein weiteres Feld vom Typ *GridGenerator* hinzugefügt. Das ist sinnvoll, da die Klasse *GridGenerator* keine eigenen Daten speichert und somit nicht viel Speicherplatz beansprucht, sondern nur Methoden anbietet, die auf dem entsprechenden Gitter arbeiten. Das hat den Vorteil, dass Benutzer nun keine zwei Objekte unterschiedlicher Klassen mehr handhaben müssen und die Benutzung dadurch vereinfacht wird.

Beim Erzeugen eines Gitters in der Fabrik *GridFactory* wird nun der entsprechende Gittergenerator mitgespeichert im Gitterobjekt. Gittergenerator-Objekte müssen somit nicht immer neu erzeugt werden, wenn das Gitter verfeinert oder vergröbert werden soll, was den Overhead verringert. Da die Klasse *GridGenerator* nur Methoden anbietet und selbst keine eigenen Membervariablen speichert (außer einem Pointer zu den Gitterpunkten in der zugehörigen *Grid*-Klasse), ist das Speichern eines Gittergenerators im Gitterobjekt mit nur wenig Overhead verbunden.

Das Feld für den Gittergenerator wurde als privat deklariert, damit Benutzer damit nicht mehr in Kontakt kommen müssen. Das Verwirrungspotenzial für Benutzer wird dadurch verringert.

Die Klasse *Grid* erhielt weitere Methoden, welche den sicheren Zugriff auf den internen Gittergenerator ermöglichen. Dabei wurden nur die Methoden nach außen geöffnet, die Benutzer auch brauchen können, wie beispielsweise die Methoden zum Verfeinern oder Vergröbern des Gitters. Methoden die leicht falsch zu benutzen sind wie *regular* sind für API-Benutzer nicht mehr zugreifbar.

Nun kann ein Gitter verfeinert oder vergröbert werden, ohne einen neuen Gittergenerator erstellen zu müssen:

```
myGrid->refine(myRefineFunc);
```

Benutzer müssen sich jetzt nur noch um das Gitterobjekt kümmern. Der Benutzercode ist etwas übersichtlicher und logischer. Fehler und unerwünschte Seiteneffekte, welche durch die Kopplung von *Grid* und *GridGenertor* entstanden, werden verringert.

Ein weiterer Vorteil ist, dass Benutzer nun nicht mehr vergessen können, Gittergeneratoren zu löschen, welche nur kurz zum Initialisieren, Verfeinern oder Vergröbern eines Gitters erzeugt wurden. Das geschieht nun automatisch, wenn das zugehörige Gitterobjekt gelöscht wird.

5.2.6. OperationQuadratureMC

Um eine Monte-Carlo-Integration durchzuführen, gibt es die Klasse *OperationQuadratureMC*. Beispielsweise enthält die Klasse eine Methode zur Integration einer beliebigen Funktion. In der alten *SG++* sieht die Methodensignatur wie folgt aus:

```
double doQuadratureFunc(FUNC func, void* clientdata);
```

Hierbei ist der Typ *FUNC* eine Typdefinition auf einen Funktionszeiger der Form:

```
typedef double (*FUNC)(int, double*, void*);
```

Problembeschreibung

Da in Java und Python keine Funktionszeiger existieren, ist es nicht möglich ohne Weiteres für die Methode *doQuadratureFunc* entsprechende Schnittstellen mit *SWIG* zu generieren.

In *SG++* wurde für Python daher ein Workaround implementiert. Es wurde manuell ein Wrapper geschrieben, welche gewöhnliche Python-Funktionen in den C++-Typ *FUNC* übersetzt. Daher verlangt die Methode *doQuadrature* und der Funktionszeiger *FUNC* jeweils einen weiteren Parameter (*clientdata*), welche zum Übersetzen benötigte Daten speichert. C++-Funktionen müssen nicht übersetzt werden, weshalb dieser Parameter aber nicht benötigt wird. Deshalb wird bei der Verwendung der C++-Schnittstelle dafür typischerweise *NULL* als Argument übergeben.

```
opMC.doQuadratureFunc(f, NULL);
```

Für Java fehlt so ein Wrapper, was der Richtlinie der Vollständigkeit in Kapitel 4.2.2 widerspricht. Sie könnte durch die JNI-Schnittstelle ähnlich realisiert werden. Das Problem hierbei ist, dass dieser Ansatz zusätzlichen Implementierungsaufwand mit sich bringt, vor allem wenn Schnittstellen für weitere Sprachen unterstützt werden sollen (siehe dazu Kapitel 4.4.2).

Verbesserungen

Anstelle von Funktionszeigern werden nun virtuelle Methoden benutzt, wie in Kapitel 4.4.4 beschrieben ist. Dafür wurde zuerst eine neue Klasse eingeführt, die als Interface dienen soll:

```
class FunctionQuadratureMC {
 public:
  /**
   * Function to evaluate.
   *
   * @param x Coordinates.
   */
  virtual double eval(std::vector<double>& x) const = 0;

  virtual ~FunctionQuadratureMC();
};
```

Die Funktion *eval* kann dann überschrieben werden, um beliebige mathematische Funktionen zu implementieren. Als Parameter wird ein Vektortyp aus der Standardbibliothek übergeben. Weitere Parameter wie die Dimension können so weggelassen werden. Trotzdem können Funktionen beliebiger Dimensionen realisiert werden.

Parameter, die für den Python-Wrapper gebraucht wurden, werden nun auch nicht mehr benötigt, was die Verwendung der API erleichtert.

Zum Schluss wurden die Methoden angepasst, welche Funktionszeiger als Parameter übergeben bekommen haben:

```
double doQuadratureFunc(const FunctionQuadratureMC& function);
```

Anstelle einer Funktionsdefinition muss nun eine von *FunctionQuadratureMC* abgeleitete Klasse definiert werden. Ein Objekt dieser abgeleiteten Klasse kann dann ganz einfach verwendet werden:

```
opMC.doQuadratureFunc(*f);
```

Hier müssen keine unnötigen *NULL*-Werte mehr als Parameter übergeben werden.

Die Klasse *FunctionQuadratureMC* kann eben auch in Python und Java abgeleitet und als Parameter in der Methode *doQuadrature* benutzt werden. Somit entfällt die zusätzliche Arbeit für jede Sprache eigene Wrapper zu schreiben, weil *SWIG* mit Polymorphismus über mehrere Sprachen hinweg umgehen kann.

Hier ein Beispiel einer von *FunctionQuadratureMC* abgeleiteten Klasse in Java:

```
public class F extends FunctionQuadratureMC {
 @Override
  public double eval(sgpp.DoubleVector x) {
    double res = 1.0;
    for (int i=0; i < x.size(); ++i) {
      res *= 4*x.get(i)*(1-x.get(i));
    }

    return res;
  }
}
```

Der Typ *DoubeVector* ersetzt hier den Vektor aus der C++-Standardbibliothek. Es kommt aus der Templateinstanziierung in der *SWIG*-Interfacedatei.

Die Methoden, welche Funktionszeiger als Parameter annehmen, wurden aber nicht ersetzt, sondern ergänzt. Es ist also immer noch möglich in C++ und Python Funktionszeiger in die Methode *doQuadrature* zu übergeben. Diese wurden beibehalten, weil das Definieren und Instanziieren von abgeleiteten Klassen umständlicher ist als das Definieren von Funktionen und Funktionszeigern.

5.3. Modul: Solver

5.3.1. Eulerverfahren

Problembeschreibung

In der Klasse *Euler* gibt es folgende Membervariable, die bestimmen soll, ob das implizite oder explizite Eulerverfahren angewendet werden soll:

```
/// specifies the type of euler that should be executed
std::string ExMode;
```

Jedoch wird die Variable weder intern benutzt, noch kann ein Benutzer auf sie zugreifen. Sie ist also gänzlich nutzlos. Noch hinzu kommt, dass ein String als Typ verwendet wurde. Das ist nicht benutzerfreundlich und kann frustrierend sein bei Schreibfehlern, weil im Konstruktor noch abgefragt wird, ob der String den richtigen Inhalt hat (also entweder *ImEul* oder *ExEul*).

Welches der beiden Verfahren angewendet werden soll, wird durch den Parameter *System* der folgenden Methode in der Klasse *Euler* bestimmt:

```
virtual void solve(SLESolver& LinearSystemSolver, sg::pde::OperationParabolicPDESolverSystem&
    System, bool bIdentifyLastStep = false, bool verbose = false);
```

Der Typ des Parameters *System* ist eine abgeleitete Klasse von *OperationParabolicPDESolverSystem* und beschreibt ein Gleichungssystem zum Lösen parabolischer partieller Differentialgleichungen. Diese Klasse speichert auch einen Operationsmodus, der angibt, welcher Löser für das Gleichungssystem verwendet werden soll. Je nachdem, welcher Gleichungslöser benutzt wird, muss also auch der entsprechende Operationsmodus gewählt werden.

Das Problem dabei ist, dass man in *System* beliebige partielle Differentialgleichungslöser spezifizieren kann, was zu Inkonsistenzen führt. Zum Beispiel ist es möglich, einen Löser wie das Crank-Nicolson-Verfahren in einem *OperationParabolicPDESolverSystem*-Objekt zu bestimmen und diesen dann mit dem Eulerverfahren zu lösen. Benutzer müssen also selbst darauf achten, dass der Modus des Gleichungssystemobjekts übereinstimmt mit dem verwendeten Löser. Das ist für Benutzer unerwartet und verstößt damit gegen die Richtline der geringsten Überraschung als auch gegen die Richtlinie des erschwerten Fehlgebrauchs.

Es kommt noch hinzu, dass der Kommentar der Variablen für den Operatormodus in *OperationParabolicPDESolverSystem* unvollständig ist:

```
/**
 * specifies in which solver this matrix is used, valid values are:
 * ExEul for explicit Euler
 * ImEul for implicit Euler
 * CrNic for Crank Nicolson solver
 */
std::string tOperationMode;
```

Hier werden nicht alle möglichen Löser aufgelistet. Es ist auch zu bezweifeln, dass dieser Kommentar aktuell gehalten wird, wenn ein neuer Löser hinzukommt. Erfahrungsgemäß vergessen es Implementierer oft solche Kommentare aktuell zu halten, was bei Benutzern dann zu Verwirrung führt.

Außerdem zwingt die Verwendung von Abkürzungen die Benutzer dazu, sich diese genau zu merken. Das ist nicht benutzerfreundlich und kann dazu führen, dass Benutzer immer wieder nachschauen müssen, wie die Abkürzungen für die verschiedenen Löser nun heißen.

Verbesserungen

- Es wurde ein neuer Aufzählungstyp *ODESolvers::Type* erstellt, welcher die verfügbaren Arten von gewöhnlichen Differenzialgleichungslösern beinhaltet. Zusätzlich wurde der „Subtyp“ *EulerSolvers::Type* der Aufzählung *ODESolvers::Type* mit dem gleichen Trick wie bei den Gittertypen erstellt.

  Der Typ der Variable *ExMode* wurde auf den neuen Aufzählungstyp *EulerSolvers::Type* umgestellt und in *mode* umbenannt, um mit den Richtlinien zur guten Namensgebung konform zu sein.

  Der Aufzählungstyp erstetzt alle entsprechenden Strings in *SG++*, zum Beispiel die Membervariable von *OperationParabolicPDESolverSystem*:

  ```
  /**
   * Specifies in which solver this matrix is used.
   */
  sg::solver::ODESolvers::Type operationMode;
  ```

  Der Kommentar muss nun nicht mehr aufzählen, welche Strings angegeben werden können und muss daher auch nicht immer aktuell gehalten werden. Benutzer können am Typ sehen, welche Optionen gewählt werden können. Eine gute IDE kann dann auch die Wahlmöglichkeiten dem Benutzer direkt anzeigen.

- Um sicherzustellen, dass der Operationsmodus des Gleichungssystemobjekts mit dem verwendeten Löser übereinstimmt, gibt es mehrere Möglichkeiten. Beispielsweise würde es reichen eine einfache Abfrage in der *solve*-Methode der Löser einzubauen, welche abprüft, ob der richtige Modus im Gleichungssystem spezifiziert wurde. So wäre die inkonsistente Benutzung nicht mehr möglich, jedoch müssten Benutzer immer noch selbst darauf achten, dass der Operationsmodus des Gleichungssystems übereinstimmt mit dem verwendeten Löser.

  Statt eine Abfrage in die Löser einzubauen, setzen die Löser einfach den gewünschten Operationsmodus im Gleichungssystem selbst, wie zum Beispiel ist in der ersten Zeile der *solve*-Methode im Eulerlöser:

  ```
  System.setODESolver(static_cast<ODESolvers::Type>(mode));
  ```

  Die statische Typumwandlung wird hier gebraucht, weil die Variable *mode* vom Typ *EulerSolvers::Type* ist. Weil dieser Typ aber einen „Subtyp“ von *ODESolvers::Type* darstellt, ist die Umwandlung kein Problem.

  Diese Variante wurde gewählt, weil der Adams-Bashforth-Löser diese Methode schon implementiert, da er im ersten Schritt zuerst den Modus für das explizite Eulerverfahren verwendet. Die

restlichen Löser sollen also durch diesen Ansatz konsistent gehalten werden. Zusätzlich wird im Kopfkommentar der *solve*-Methode darauf hingewiesen, dass ein Seiteneffekt auftritt, damit Benutzer keine unerwünschten Überraschungen erleben.

Der Vorteil dieser Variante ist, dass Benutzer den Operationsmodus des Gleichungssystems nicht selbst konsistent halten müssen zum verwendeten Löser, weil das nun automatisch geschieht.

Die inkonsistente Benutzung von Gleichungssystemen und Lösern ist nun nicht mehr möglich. Das Setzen des Operationsmodus durch den Benutzer für Objekte der Klasse *OperationParabolicPDESolverSystem* ist überflüssig geworden.

5.3.2. Vereinheitlichung der Klassen

Problembeschreibung

In *SG++* können die Gleichungssystemlöser mit doppelter oder einfacher Genauigkeit arbeiten. Um dies zu erreichen, wurden jeweils zwei verschiedene Klassen angelegt.

Zum Beispiel gibt es den Gleichungslöser *ConjugateGradients* für doppelte Genauigkeit und noch mal als *ConjugateGradientsSP* für einfache Genauigkeit, was redundant ist.

Änderungen in einer Klasse müssten genauso auch in der anderen nachgezogen werden, um sie konsistent zu halten. Das verdoppelt die Arbeit der Implementierer und macht Anpassungen fehleranfällig.

Benutzer müssen sich hingegen erst Mal klar machen, was der Unterschied der Klassen ist, worunter die Erlernbarkeit leidet.

Verbesserungen

- Zunächst wurden Templates für die Klassen *DataMatrix*, *DataVector* und *OperationMatrix* eingeführt werden, weil die Solver-Klassen von diesen Datenstrukturen abhängen. Danach konnten für die Klassen zum Lösen linearer Gleichungssysteme Templates eingeführt werden.

  Es wurde nur die Genauigkeit, mit der die Klassen arbeiten, parametrisiert, also ob nun *double* oder *float* genutzt werden soll. Die entsprechenden Klassen, welche auf einfacher Genauigkeit gearbeitet hatten, wie *SGSolverSP* und *SLESolverSP*, wurden entfernt.

  Den parametrisierten Klassen wurde der Suffix *XP* hinzugefügt, also statt *SGSolver* heißt die Templateklasse *SGSolverXP*. Zusätzlich wurden Typdefinitionen für alle Templateklasse hinzugefügt. Für die von *SLESolverXP* abgeleitete Klasse *ConjugateGradientsXP* wären diese:

```
typedef ConjugateGradientsXP<double> ConjugateGradients;
typedef ConjugateGradientsXP<float> ConjugateGradientsSP;
```

Die Typdefinitionen der anderen Templateklassen folgen diesem Schema.

Den Suffix *XP* für die Templateklassen anzuhängen ist zugegebenermaßen nicht ganz intuitiv für den Benutzer. Es leitet sich aus der Konvention in *SG++* ab, wonach alle Klassen, die mit einfacher Genauigkeit arbeiten, den Suffix *SP* (für Single Precision) besitzen. Der Suffix *XP* soll dann eben verdeutlichen, dass die Templateklassen mit verschiedenen Genauigkeiten arbeiten können. Warum der Suffix nicht einfach weggelassen wurde liegt daran, dass die Klassen, die mit doppelter Genauigkeit arbeiten, in der alten *SG++*-Version keinen eigenen Suffix hatten. Um diese Konvention nicht zu brechen, wurden die Templateklassen durch eben diesen Suffix ergänzt. Klassen, die mit doppelter Präzision rechnen haben weiterhin keinen Suffix, während die Klassen, die mit einfacher Genauigkeit rechnen ihren Suffix beibehalten. Hier sollte nochmal verdeutlicht werden, dass die Templateklasse *ConjugateGradientsXP* nur für interne Zwecke gedacht ist. Benutzer verwenden weiterhin die Klassen *ConjugateGradients* und *ConjugateGradientsSP* wie sie es gewohnt sind. Auch andere Klassen und Module, die nur eine der beiden Versionen brauchen, verwenden weiterhin die Typdefinition. Die Parametrisierung der Klassen wurde nicht durch alle weiteren Klassen, welche die Klassen aus dem *Solver*-Modul benutzen, durchgezogen. Das verringert die Kompilierzeit (siehe dazu Kapitel 4.3.1).

In Java und Python stehen dann auch nur zwei Versionen (doppelte und einfache Genauigkeit) zur Verfügung, weil die Parametrisierung der Klassen sowieso nicht möglich ist für die beiden Sprachen. Das Kapitel 4.4.4 erläutert das Problem genauer.

- Für nicht abstrakte Templateklassen wurden die Schnittstellen von den Implementierungen getrennt, wie beschrieben in dem Kapitel über Template-Metaprogrammierung, indem einfach die Implementierungsdatei in der Headerdatei durch eine *#include*-Direktive eingefügt wird.

Durch die Templates kann aber nun sichergestellt werden, dass alle Versionen einer Klasse auch konsistent sind. Als Beispiel sei hier die Methode *addReduce* der Klasse *DataMatrix* genannt, die eine Überladung hatte, welche in der alternativen Version *DataMatrixSP* gefehlt hatte. Das war somit ein Verstoß gegen die Richtlinie der geringsten Überraschung.

Da nun alle Versionen einer Klasse in einer Datei zusammengefasst wurden, wird es einfacher für Benutzer sich zurechtzufinden, ohne mehrere Headerdateien verschiedener Alternative einer Klasse ansehen zu müssen.

5.3.3. Basisklasse SGSolver

Problembeschreibung

Die Klasse *SGSolver* dient als Basisklasse für alle weiteren Löserklassen. Das Problem hierbei ist jedoch, dass das *Ist-ein*-Kriterium hier nicht erfüllt wird. Die Basisklasse soll einen abstrakten Löser darstellen, während die abgeleiteten Klassen entweder Löser für lineare Gleichungssysteme sind oder Löser für gewöhnliche Differenzialgleichungen. Die Klassen sind also nicht ohne Weiteres austauschbar.

Die beiden Löserarten sind zu unterschiedlich, um sie unter ein bestimmtes Interface zusammenzufassen. Der Unterschied wird besonders an der Methode *solve* deutlich, welche nicht in der Klasse *SGSolver* selbst nicht deklariert ist, sondern nur in den abgeleiteten Klassen.

Zum Beispiel sieht in der Klasse *ODESolver* die Methode wie folgt aus:

```
virtual void solve(SLESolver& LinearSystemSolver, sg::pde::OperationParabolicPDESolverSystem&
    System, bool bIdentifyLastStep = false, bool verbose = false) = 0;
```

In der Klasse *SLESolver* sieht sie jedoch anders aus:

```
virtual void solve(sg::base::OperationMatrix& SystemMatrix, sg::base::DataVector& alpha,
    sg::base::DataVector& b, bool reuse = false, bool verbose = false, double max_threshold =
    DEFAULT_RES_THRESHOLD) = 0;
```

Das Einzige, was die beiden Löserarten gemeinsam haben, sind drei Membervariablen mit zugehörigen Getter- und Setter-Methoden. Diese Funktionalität ist auch das einzige, was die Basisklasse *SGSolver* anbietet. Der Nutzen der Klasse ist also begrenzt. Benutzer müssen sowieso Objekte eines abgeleiteten Typs (*SLESolver* oder *ODESolver*) deklarieren, damit eine *solve*-Methode zur Verfügung steht.

Hinzu kommt, dass eine weitere Variable in *SGSolver* eine doppelte Bedeutung hat, je nachdem welche Löserart benutzt wird:

```
/// epsilon needed in the, e.g. final error in the iterative solver, or a timestep
double myEpsilon;
```

Hier werden zwei verschiedene Konzepte mit dem gleichen Namen verwendet und somit eine falsche Konsistenz erzeugt.

Verbesserungen

Die Basisklasse *SGSolver* wurde entfernt. Die Variablen und die dazugehörigen Getter- und Setter-Methoden von *SGSolver* wurden jeweils in die abgeleiteten Klassen *ODESolver* und *SLESolver* verschoben. Die beiden Klassen sind nun voneinander getrennt.

Die Variable *myEpsilon* wurde zusammen mit den zugehörigen Methoden für die Klasse *ODESolver* umbenannt in *timestepSize* und die doppelte Bedeutung somit aufgelöst.

Zusätzlich wurde auch die Vererbungshierarchie auch verflacht und somit die Softwarearchitektur vereinfacht. Für Benutzer ändert sich an der Verwendung der Klassen *ODESolver* und *SLESolver* fast nichts. Nur die Benennung der Variablen ist nun eindeutig und die Anzahl der Module, mit denen sich Benutzer beschäftigen muss, wurde verringert.

5.4. Modul: Datadriven

5.4.1. Anwendungen

Im Modul *Datadriven* sind Klassen zu finden, welche Anwendungen der dünnen Gittern fürs Data-Mining implementieren. Die Klassen *Learner*, *LearnerDensityBased*, *LearnerDensityBasedReg* und *LearnerVectorizedIdentity* sind alles konkrete Implementierungen der abstrakten Basisklasse *LearnerBase*.

Diese Klassen fassen Funktionalitäten mehrerer Module und Klassen zusammen und bieten eine vereinfachte Schnittstelle für Benutzer an. Benutzer, die Data-Mining betreiben wollen, müssen somit nicht selbst die gewünschten Funktionalitäten aus den grundlegenden Klassen zusammenstellen, sondern können stattdessen die vereinfachte Schnittstelle nutzen. Dieses Entwurfsmuster wird Fassade genannt. Es verringert Abhängigkeiten im Benutzercode, fördert die lose Kopplung der Module und vereinfacht die Verwendung der API.

Problembeschreibung

- Die konkrete Klasse *Learner* hat einen zu generischen Namen. Es passt nicht zu den anderen konkreten Klassen *LearnerDensityBased*, *LearnerDensityBasedReg* und *LearnerVectorizedIdentity* und verstößt damit gegen die Richtlinie der guten Namensgebung.
- Als Basisklasse *LearnerBase* sollte eine einheitliche Basis für die konkreten abgeleiteten Klassen bieten. Vererbung sollte dazu dienen Klassen zu erweitern oder deren Verhalten zu ändern (4.3.1), aber die Basisklasse enthält Methoden und Variablen, die nicht in allen abgeleiteten Klassen benötigt werden.

  In der Basisklasse *LearnerBase* werden die folgenden Membervariablen deklariert, welche in der Klasse *LearnerDensityBased* aber nicht verwendet werden:

```
/// the grid's coefficients
sg::base::DataVector* alpha_;
/// sparse grid object
sg::base::Grid* grid_;
```

  Die Basisklasse *LearnerBase* enthält die rein virtuelle Methode *createDMSSystem*. In der abgeleiteten Klasse *LearnerDensityBased* wird sie wie folgt implementiert:

```
sg::datadriven::DMSystemMatrixBase* LearnerDensityBased::createDMSystem(
   sg::base::DataMatrix& trainDataset, double lambda) {
  // Is not used
  return NULL;
}
```

  Diese Definition ist nutzlos und verstößt gegen die Richtlinie der guten Ausnahmebehandlung, weil sie nur einen Nullzeiger zurückgibt.

- Die eben erwähnten unnötigen Variablen in *LearnerDensityBased* verdecken Implementierungsfehler. In der Basisklasse *LearnerBase* gibt es die nicht-virtuellen Methoden *dumpGrid* und *dumpFunction*, welche die Variablen *alpha_* und *grid_* verwenden.

  Die Klasse *LearnerDensityBased* verwendet die beiden Variablen gar nicht und überschreibt auch die beiden Methoden nicht. Das bedeutet, dass die für die Klasse *LearnerBasedDensity* die beiden Methoden nicht die gewünschte Funktionalität bieten und damit gegen die Richtlinie der geringsten Überraschung verstößt.
- Es ist nicht einfach nachzuvollziehen, welche Implementierungen die Klassen benutzen. Zum Beispiel greifen die Klassen *Learner* und *LearnerVectorizedIdentity* für die Methode *train* auf

die Implementierung der Basisklasse zurück. Diese ruft aber die Methode *createDMSystem* auf, welche verschiedene Implementierungen in den beiden Klassen haben.

Die anderen beiden Klassen *LearnerDensityBased* und *LearnerDensityBasedReg* hingegen benutzen die Methode *createDMSystem* nicht und implementieren dafür eigene *train*-Methoden.

Das ist inkonsistent und eine unzureichende Trennung von Interface und Implementierung. Die Softwarearchitektur wird dadurch unübersichtlich, wodurch die Wartbarkeit leidet. Außerdem wird es Benutzern erschwert, den Code der Bibliothek zu verstehen.

- Die abstrakte Klasse *LearnerBase* ist nicht ganz vollständig. Methoden wie *getExecTime* und *getNrGridPoints*, welche in der Klasse *LearnerDensityBased* definiert sind, fehlen in der Basisklasse.
- Die Basisklasse *LearnerBase* gibt es in zwei verschiedenen Versionen. Eine die mit doppelter Genauigkeit rechnet und eine andere, die mit einfacher Genauigkeit rechnet. Zwei verschiedene Klassen (*LearnerBase* und *LearnerBaseSP*) dafür zu haben ist redundant.
- In der Klasse *Learner* wurde der Aufzählungstyp *LearnerRegularizationType* definiert. Dieser Typ wird jedoch auch von den anderen konkreten Klassen gebraucht. Diese müssen deswegen unnötigerweise die Klasse *Learner* einbinden.

Verbesserungen

- Die Klasse *Learner* wurde in *LearnerBasic* umbenannt und die Basisklasse *LearnerBase* zu *Learner* damit Benutzer diese nicht verwechseln. Das Interface hat somit einen generischen Namen und *LearnerBasic* passt namentlich nun auch zu den anderen konkreten Klassen.
- Die Basisklasse, welches nun *Learner* heißt, wurde zu einem Interface umgewandelt und wurde aufgeräumt.

  Die Membervariablen *grid_* und *alpha_* wurden entfernt. Es wurde eine neue Klasse *LearnerSingleGrid* angelegt, die von *LearnerBase* ableitet, in welcher die beiden Variablen nun deklariert sind. Diese ebenso abstrakte Klasse vereinheitlicht die Funktionalitäten, für die eben nur ein Gitter gebraucht wird. Auf eine weitere Klasse *LearnerMultipleGrid* wurde aber verzichtet, weil nur die Klasse *LearnerDensityBased* auf mehreren Gittern arbeitet. Bei Bedarf kann diese aber nachgefügt werden.

  Das Hinzufügen einer zusätzlichen Klasse in die Hierarchie ist damit zu rechtfertigen, um Code wiederzuverwenden und damit Redundanzen zu vermeiden.

  Öffentliche Methoden wie *train* und *predict* wurden zu rein virtuellen Methoden umgewandelt. Durch die Umstellung zu rein virtuellen Methoden müssen nun alle konkreten Klassen diese Methoden selbst implementieren. Das bedeutet, dass die Klassen *LearnerBasic* und *LearnerVectorizedIdentity* nun sehr ähnliche Implementierungen der *train*-Methode haben. Das ist zwar redundant, aber dafür erhält man eine übersichtlichere Architektur und es ist einfacher nachzuvollziehen, welche Klasse welche Implementierung benutzt.

Dadurch konnte auch die Methode *createDMSystem* aus der Basisklasse entfernt werden, weil sie nur von der *train*-Methode aufgerufen wurde. Die Klassen, welche diese Methode brauchen, besitzen nun ihre eigene Definition. Alle anderen Klassen können auf diese Methode verzichten.

- Die Methoden *dumpGrid* und *dumpFunction* wurden zu rein virtuellen Methoden umgewandelt, weil Klassen mit nur einem Gitter eine andere Implementierung brauchen als Klassen die mehrere Gitter verwenden. Die Methoden wurden in den jeweiligen Klassen *LearnerSingleGrid* und *LearnerDensityBased* implementiert.
- In der Basisklasse wurden die Methoden *getExecutionTime* und *getGridPointCount* definiert. Sie ersetzen die Methoden *getExecTime* und *getNrGridPoints* in der abgeleiteten Klasse *LearnerDensityBased*.
- Für die Basisklasse *Learner* wurden, wie für die Gleichungslöser in Kapitel 5.3.2, Templates eingeführt.

  Ebenso wurden für die Klasse *LearnerSingleGrid* Templates eingeführt. Hier musste jedoch beachtet werden, dass einige Methoden unterschiedliche Implementierungen besitzen können, je nachdem, ob sie mit einfacher oder mit doppelter Genauigkeit rechnen. Um dies realisieren zu können wurden Templatespezialisierungen benutzt. Für bestimmte Templateparameter lassen sich Spezialisierungen für Methoden definieren.

  Zum Beispiel hat die Methode *dumpGrid* für beliebige Templateparameter *P* folgende Implementierung:

```
template <class P>
void LearnerSingleGridXP<P>::dumpGrid(std::string tFilename) {
  if (this->isTrained_) {
    sg::base::GridPrinter myPlotter(*grid_);
    myPlotter.printSparseGrid(*alpha_, tFilename, false);
  }
}
```

  Wenn nun aber der Templateparameter *P* vom Typ *float* ist, wird folgende Implementierung verwendet:

```
template <>
inline void LearnerSingleGridXP<float>::dumpGrid(std::string tFilename) {
  if (this->isTrained_) {
    sg::base::GridPrinter myPlotter(*grid_);
    sg::base::DataVector tmp_alpha(alpha_->getSize());
    sg::base::PrecisionConverter::convertDataVectorSPToDataVector(*alpha_, tmp_alpha);
    myPlotter.printSparseGrid(tmp_alpha, tFilename, false);
  }
}
```

- Der Aufzählungstyp *LearnerRegularizationType* wurde in das Interface *Learner* verschoben, weil diese sowieso von allen konkreten Klasse referenziert wird.

Die Basisklasse verhält sich nun wie ein Interface. Unbenutzte Variablen und Methoden wurden entfernt und die Architektur wurde verständlicher. Ein Diagramm der Vererbungshierarchie ist im Anhang unter A.2 zu finden. Konkrete Klassen haben nun konkrete Namen und es wurde eine Unterscheidung

zwischen Klassen, die nur ein Gitter benutzen, und Klassen, welche mehrere Gitter benutzen, gemacht. Dafür musste eine weitere Abstraktionsebene hinzugefügt werden.

Die alte Architektur scheint auf den ersten Blick einfacher zu sein, wie den Diagrammen im Anhang in A.1 zu entnehmen ist. Jedoch sind die Abhängigkeiten zwischen den Vererbungsebenen komplizierter und die Architektur inkonsistent. Außerdem fehlen im Diagramm A.2 die Klassen, die mit einfacher Genauigkeit rechnen, weil in einer eigenen, vollständig separaten Klassenhierarchie zu finden sind, zu sehen unter A.1. In der neuen Version wurden die Klassen in einer Hierarchie vereinigt.

5.4.2. Einlesen von ARFF-Dateien

Problembeschreibung

ARFF ist ein spezielles Format für Datensätze fürs Data-Mining. *SG++* bietet hierfür eine Klasse an, um solche Dateien zu lesen und die benötigten Daten daraus zu extrahieren.

Jedoch ist die Verwendung der Klasse recht mühsam und führt zu Boilerplatecode, da für jedes Datum eine eigene Methode aufgerufen werden muss, wie im folgenden Beispielcode zu sehen ist:

```
const std::string data_path = "path/to/data.arff";
sg::datadriven::ARFFTools data;

size_t dim = data.getDimension(data_path);
size_t instanceCount = data.getNumberInstances(data_path);

sg::base::DataVector* classes = new sg::base::DataVector(instanceCount);
data.readClasses(data_path, *classes);

sg::base::DataMatrix* trainingData = new sg::base::DataMatrix(instanceCount, dim);
data.readTrainingData(data_path, *trainingData);
...
delete trainingData;
delete classes;
```

Außerdem zwingt diese API Benutzer dazu, die Methoden in der richtigen Reihenfolge aufzurufen, da Benutzer den Vektor für die Klassen und die Matrix für die Trainingsdaten selbst mit den richtigen Dimensionen instanziieren müssen. Das heißt, dass zuerst die richtigen Dimensionen gelesen werden müssen, bevor die anderen Daten initialisiert werden müssen.

Bei der Instanziierung des Vektors und der Matrix können Benutzern leicht Fehler unterlaufen, wenn zum Beispiel Spalten- und Zeilenanzahl vertauscht werden.

Verbesserungen

- Zuerst wurde die neue Klasse *Dataset* eingeführt. Sie speichert die Dimension, die Anzahl der Instanzen, den Vektor für die Klassen und die Matrix für die Trainingsdaten.

Um auf die Daten zugreifen zu können, wurden zugehörige Getter-Methoden hinzugefügt, wobei die Getter-Methoden für die Klassen und Trainingsdaten jeweils konstante Zeiger auf die entsprechenden Daten zurückgeben, wie im folgenden Codebeispiel:

```
sg::base::DataVectorXP<P>* const getClasses() const;
```

Das hat den Vorteil, dass Benutzer die Daten nicht einzeln löschen müssen, weil die Klasse *Dataset* dies übernimmt.

- Der Klasse *ARFFTools* wurde um eine neue Methode erweitert:

```
static Dataset* getDataset(const std::string& filename);
```

Diese Methode führt die im Beispiel angeführten Schritte zum Einlesen von *ARFF*-Dateien durch und speichert das Ergebnis in einem *Dataset*-Objekt. Solche Methoden werden Convenience-Methode genannt. Sie helfen dabei Boilerplatecode zu vermeiden, indem mehrere Methoden, welche häufig zusammen aufgerufen werden, unter einer Methode zusammengefasst werden. Diese Technik ist verwandt mit dem Fassade-Entwurfsmuster, jedoch erstreckt es sich üblicherweise nicht über mehrere Module.

- Alle Methoden der Klasse *ARFFTools* wurden alle als statisch deklariert, weil die Klasse selbst keine Variablen besitzt, sondern nur Methoden anbietet. Somit muss kein Objekt dieser Klasse mehr instanziiert werden.

Den Code, welchen Benutzer nun schreiben müssen um eine ARFF-Datei einzulesen und die nötigen Datenstrukturen zu initialisieren, ist nun deutlich kürzer und einfacher zu verwenden:

```
const std::string data_path = "path/to/data.arff";
sg::datadriven::Dataset* dataset = sg::datadriven::ARFFTools::getDataset(data_path)
...
delete dataset;
```

Auf die benötigten Daten kann nun über das Objekt der Klasse *Dataset* zugegriffen werden und Benutzer müssen keine eigenen Objekte mehr instanziieren, was den Aufwand zum Schreiben des Codes reduziert.

Benutzer haben nun auch weniger Möglichkeiten die API falsch zu benutzen, da es eben nur auf einen Methodenaufruf reduziert ist.

5.4.3. Kleinere Änderungen

In diesem Kapitel werden weitere kleinere Änderungen im Modul *Datadriven* zusammengefasst. Es sind Änderungen, die nicht in anderen Kapiteln behandelt wurden, jedoch nicht unerwähnt bleiben sollten.

Dynamische Speicherallokation

Im Code wurden einige Objekte unnötigerweise dynamisch angelegt. Zum Beispiel in der Methode *train* in der Klasse *LearnerDensityBased* wurde folgende Variable angelegt:

```
sg::base::SGppStopwatch* myStopwatch = new sg::base::SGppStopwatch();
```

Das sollte aber nur dann verwendet werden, wenn erst zur Laufzeit bekannt ist wie viel Speicher benötigt wird, wenn die Lebensdauer von Objekten über den Gültigkeitsbereich der Methode hinaus erweitert werden soll, wenn Polymorphie benötigt wird oder wenn sehr große Objekte angelegt werden müssen, welche nicht auf den Stack passen.

Keiner dieser Kriterien trifft hier zu. Deswegen wurde es ersetzt durch statische Stackallokation:

```
sg::base::SGppStopwatch myStopwatch;
```

Das hat den Vorteil, dass Implementierer nicht mehr selbst den Speicher freigeben müssen, was Speicherlecks vorbeugt. Außerdem führen zu viele dynamische Allokationen zu einer Fragmentierung des Speichers.

Fehlende Ausnahmebehandlung

Im folgenden Code der Klasse *LearnerBasic* fehlt eine Fehlermeldung:

```
if (this->CMode_ == Laplace) {
  C_ = sg::PdeOperationsFactory::createOperationLaplace(*this->grid_);
} else if (this->CMode_ == Identity) {
  C_ = sg::BaseOperationsFactory::createOperationIdentity(*this->grid_);
} else {
  // should not happen
}
```

Hier wird nun eine Ausnahme mit entsprechender Meldung geworfen, auch wenn der Else-Zweig normalerweise gar nicht auftreten kann. Wie in Kapitel 4.4.4 gezeigt wurde, können Methodenaufrufe aus anderen Sprachen unerwartete Effekte hervorrufen. Durch die zusätzliche Ausnahmebehandlung sollen Fehlbenutzungen aus anderen Sprachen ausgeschlossen werden.

Einbinden von Headern

Die *#include*-Direktiven wurden in neu angeordnet, um dem Styleguide gerecht zu werden. Dadurch wurde deutlich, dass einige Headerdateien mehrfach eingebunden wurden.

Beispielsweise in der CPP-Datei der Klasse *LearnerDensityBasedReg* wurde die Headerdatei *BaseOpFactory* doppelt eingebunden.

5.5. Weitere Verbesserungsvorschläge

In diesem Unterkapitel sind weitere, noch nicht realisierte Vorschläge zur Umgestaltung der API zu finden.

5.5.1. Operatoren

Um auf Gittern Operationen wie Hierarchisierung oder Evaluation ausführen zu können, gibt es in *SG++* die Operator-Klassen.

Hier ein Beispiel zum Hierarchisieren eines Gitters:

```
sg::base::OperationHierarchisation* opHier =
    sg::op_factory::createOperationHierarchisation(*grid);
opHier->doHierarchisation(alpha);
```

Hier wird durch eine Fabrikmethode zuerst ein zum Gittertyp passender Hierarchisierungsoperator erstellt und dann angewendet. Das Problem hierbei ist die starke Kopplung zwischen den Operatoren und Gittern, weil die Operatoren Zeiger zu den Daten des Gitters besitzen und somit Seiteneffekte auf das Gitter ausüben.

Ein anderer Ansatz wäre hier gewesen, dass die Gitterklasse selbst die passenden Operationen anbietet, dann sähe der Code wie folgt aus:

```
grid->doHierarchisation(alpha);
```

Diese Version ist kürzer, einfacher zu verwenden, einfacher zu erlernen, weil Benutzer sich mit weniger Klassen beschäftigen müssen und hat dadurch auch eine verbesserte Erschließbarkeit. Jedoch hat diese Variante auch Nachteile; sie ist nicht erweiterbar für Benutzer.

Die Variante mit den Operatoren kann aber verbessert werden, um einige Nachteile auszugleichen. Anstelle von Operatoren könnten *Non-Member-Funktionen* benutzt werden. Das sind Funktionen, welche keine Klassenmethoden sind, aber trotzdem auf den Daten von Klassen arbeiten. Durch diese Technik wird die Kopplung der Klassen reduziert und die Kapselung gefördert [Mey].

Der Code würde dann wie folgt aussehen:

```
doHierarchisation(alpha, grid);
```

Hier wäre *doHierarchisation* eine freie Funktion, welche auf den Daten des Gitters arbeitet. Weil aber, wie in Kapitel 4.4.4 beschrieben, keine globalen Funktionen benutzt werden sollten, kann die Funktion als statische Methode einer Klasse *OperationHierarchisation* implementiert werden:

```
OperationHierarchisation::doHierarchisation(alpha, grid);
```

Hier sollte beachtet werden, dass die Klasse *OperationHierarchisation* keine eigenen Variablen speichert, sondern nur Methoden anbietet.

5.5.2. Smartpointer

Zeiger in C++ sind problematisch und können zu einigen Fehlern führen [Red11], wie zum Beispiel Nullzeigerdereferenzierung, doppelte Freigabe eines Zeigers, Speicherzugriffsfehler und Speicherlecks. Benutzer nämlich selbst darauf aufpassen, ob ein Zeiger auf eine valide Speicheradresse zeigt und oder nicht.

Um diese Probleme zu vermeiden und Benutzern Arbeit abzunehmen gibt es die sogenannten *Smart-Pointer*.

Es gibt mehrere Varianten von Smart-Pointern. Eine Variante wären die *Scoped-Pointer*. Das sind Zeiger, welche ein Objekt automatisch löschen, wenn der Zeiger den Gültigkeitsbereich verlässt. Solche Zeiger sind einzigartig, was bedeutet, das keine zwei Zeiger auf ein Objekt zeigen können.

Als Anwendung in *SG++* könnten dann alle Fabrikmethoden solch einen *Scoped-Pointer* zurückgeben. Also anstelle von

```
static Grid* createGrid(const GridConfiguration& config);
```

wäre der Rückgabewert

```
static std::unique_ptr<Grid> createGrid(const GridConfiguration& config);
```

wobei hier *std::unique_ptr<Grid>* die C++11 Variante eines *Scoped-Pointers* ist.

5.5.3. Aktuellerer C++-Standard

Der aktuellste C++-Standard ist der C++14 vom August. Dieser ist aber noch sehr neu und aktuelle Compiler unterstützen den Standard noch nicht vollständing.

Trotzdem kann auf einen aktuelleren Standard umgestellt werden, nämlich den C++11-Standard. Dieser Standard bringt viele neue Features mit sich. Es folgt eine Auswahl der Features, welche für die Gestaltung einer Softwarebibliothek von Nutzen sein könnten:

Smart pointer

Wie im vorherigen Abschnitt beschrieben, helfen *Smart-Pointer* Benutzern dabei ein fehlerfreies Programm zu schreiben. Im C++03-Standard gibt es *auto_ptr*. Dieser wird aber im neueren C++11-Standard durch den *unique_ptr* ersetzt, weil *auto_ptr* einpaar Probleme mit sich bringt. Beispielsweise ist die Kopiersemantik nicht kompatibel mit *STL-Containern*.

Zusätzlich bringt der neuere C++-Standard noch weitere intelligente Zeiger mit sich, wie *weak_ptr* und *shared_ptr*.

final

Das *final*-Schlüsselwort kann für Klassen- und Methodendeklarationen verwendet werden. Als *final* deklarierte Klassen können nicht abgeleitet werden. Virtuelle Methoden, die als *final* deklariert wurden, können nicht überschrieben werden.

Durch eine konsistente Verwendung des Schlüsselworts *final* wird gekennzeichnet, welche Methoden zum Überschreiben vorgesehen sind und welche Klassen abgeleitet werden können. *Final* hilft dabei, die Richtlinien zur Verwendung von Vererbung im Kapitel 4.3.1 einzuhalten.

override

Mit *override* soll sichergestellt werden, dass eine Methode virtuell ist und eine Methode aus der Basisklasse überschreibt. Mit *override* können Fehler wie im folgenden Beispiel vermieden werden:

```
class BaseClass
{
    virtual void doSomething() const;
};

class DerivedClass : BaseClass
{
    virtual void doSomething() // <- const-Deklaration vergessen
    {
      ...
    }
};
```

Im Beispiel leitet die Klasse *DerivedClass* von der Klasse *BaseClass* ab und die Methode *doSomething* soll eigentlich die Methode in der Basisklasse überschreiben. Weil aber in der abgeleiteten Klasse die *const*-Deklaration fehlt, wird die Methode nicht überschrieben. Solche Fehler werden nicht vom Compiler abgefangen und sind schwer zu finden.

Durch die Verwendung von *override* kann der Compiler jedoch diesen Fehler abfangen:

```
class BaseClass
{
    virtual void doSomething() const;
};

class DerivedClass : BaseClass
{
    virtual void doSomething() override // Compiler wirft hier einen Fehler
    {
      ...
    }
};
```

Wenn zusätzlich noch eine aktuellere *SWIG*-Version (mind. Version 3), dann können die *final*- und *override*-Deklarationen auch für Java-Klassen und -Methoden durch *SWIG* generiert werden.

6. Vergleich der alten mit der neuen SG++-Version

In diesem Kapitel soll aufgezeigt werden, wie sich die Änderungen an *SG++* auf den Benutzercode auswirken. Anhand eines einfachen Beispiels soll illustriert werden, was die neue Schnittstelle benutzerfreundlicher macht als die alte. Natürlich können mit dem simplen Beispiel nicht alle Verbesserungen der neuen Schnittstelle aufgezeigt werden. Es bietet jedoch einen guten Eindruck einiger Schwachstellen der alten Schnittstelle.

Das Beispiel wurde zwei Mal implementiert, eine Version angepasst auf die neue Schnittstelle und die andere angepasst auf die alte. Als Implementierungssprache wurde C++ gewählt, jedoch hätte auch Java oder Python die Unterschiede der beiden *SG++*-Versionen genauso verdeutlichen können.

Wie in Kapitel 4.2.3 erwähnt, spielt nicht nur die Benutzerfreundlichkeit eine Rolle, sondern auch die Effizienz des Codes. Dafür wurden Tests durchgeführt um jeweils die Performance der alten und neuen C++-, Python- und Java-Schnittstellen zu vergleichen.

6.1. Vergleich der alten und neuen API anhand eines Beispiels

Schritt für Schritt wird gezeigt, wie der Code aussehen muss, um eine einfache Interpolation und Auswertung einer Funktion durchzuführen. Dabei sollen die Vorteile der neuen Schnittstelle gegenüber der alten verdeutlicht werden und erklärt werden, was die neue API benutzerfreundlicher macht.

1. Die zu interpolierende Funktion f ist

$$f(x_0, x_1) \quad = \quad 16 \cdot (x_0 - 1) \cdot x_0 \cdot (x_1 - 1) \cdot x_1$$

   und wird an der Stelle $(x_0, x_1) = (0.52, 0.73)$ ausgewertet.

   Im Beispielcode wird die Funktion dann wie folgt implementiert:

```
double f(double x0, double x1) {
   return 16.0 * (x0-1)*x0 * (x1-1)*x1;
}
```

   Diese Implementierung ist in beiden Versionen des Beispielcodes identisch.

2. Die Funktion ist zwei-dimensional. Dafür muss zuerst ein zwei-dimensionales Gitter erstellt werden. Es werden lineare Basisfunktionen gewählt und ein reguläres Gitter auf Level 3 erzeugt.

   In der alten Version werden dafür die folgenden Schritte benötigt:

```
// Create a two-dimensional piecewise bi-linear grid
int dim = 2;
sg::base::Grid* grid = sg::base::Grid::createLinearGrid(dim);

// Create regular grid on level 3
int level = 3;
sg::base::GridGenerator* gridGenerator = grid->createGridGenerator();
gridGenerator->regular(level);
```

   Benutzer könnten hier mehrere Problemen bekommen, so wie es in Kapitel 5.2.4 beschrieben ist.

   Mit der neuen Schnittstelle treten diese Probleme nicht mehr auf.

```
// Set up configuration for a regular two-dimensional piecewise bi-linear grid on level 3
sg::base::RegularGridConfiguration gridConfig;
gridConfig.type = sg::base::Grids::Linear;
gridConfig.dim = 2;
gridConfig.level = 3;

// Create grid
sg::base::Grid* grid = sg::base::GridFactory::createGrid(gridConfig);
```

   Der Benutzercode wirkt dadurch aufgeräumter, weil alle Konfigurationswerte des Gitters zusammen in einer Struktur gespeichert sind und an einer Stelle definiert werden. Und wenn so eine Konfigurationsstruktur einmal erstellt ist, ist das Generieren eines Gitters nur durch einen Fabrikmethodenaufruf erledigt. Das macht den Benutzercode dadurch weniger fehleranfällig und ist benutzerfreundlicher.

3. Danach muss die Funktion ausgewertet und im Koeffizientenvektor *alpha* gespeichert werden.

   In der alten Version würde das wie folgt implementiert werden:

```
// Create coefficient vector
sg::base::DataVector alpha(grid->getSize());
alpha.setAll(0.0);

// Set function values in alpha
sg::base::GridIndex* gp;
sg::base::GridStorage* gridStorage = grid->getStorage();
for (int i=0; i < grid->getSize(); ++i) {
  gp = gridStorage->get(i);
  alpha[i] = f(gp->abs(0), gp->abs(1));
}
```

Die neue Version benötigt die Variable *gridStorage* nicht mehr. Dafür bietet das *grid*-Objekt die benötigten Methoden an.

```
// Create coefficient vector
sg::base::DataVector alpha(grid->getPointCount());
alpha.setAll(0.0);

// Set function values in alpha
sg::base::GridIndex* gp;
for (int i=0; i < grid->getPointCount(); ++i) {
  gp = grid->getIndex(i);
  alpha[i] = f(gp->abs(0), gp->abs(1));
}
```

Benutzer müssen sich mit weniger Klassen auskennen, um die gleiche Funktionalität zu erhalten. Das bedeutet bessere Erlernbarkeit und weniger Schreibarbeit für Benutzer.

4. Das Gitter muss noch hierarchisiert werden und die Interpolationsfunktion evaluiert werden. Der dazugehörige Code ist für beide Versionen aber fast identisch.

```
// Hierarchize
sg::base::OperationHierarchisation* opHier =
    sg::op_factory::createOperationHierarchisation(*grid);
opHier->doHierarchisation(alpha);

// Evaluation operation
sg::base::OperationEval* opEval = sg::op_factory::createOperationEval(*grid);
sg::base::DataVector p(dim);

// Evaluate on (0.52, 0.73)
p[0] = 0.52;
p[1] = 0.73;
std::cout << "u" << p.toString() << " = " << opEval->eval(alpha, p) << std::endl;
```

In der neuen Version wird der Namensraum *op_factory* durch die Klasse *BaseOperationsFactory* ersetzt.

```
// Hierarchize
sg::base::OperationHierarchisation* opHier =
    sg::BaseOperationsFactory::createOperationHierarchisation(*grid);
opHier->doHierarchisation(alpha);

// Evaluation operation
sg::base::OperationEval* opEval = sg::BaseOperationsFactory::createOperationEval(*grid);
sg::base::DataVector p(gridConfig.dim);

// Evaluate on (0.52, 0.73)
p[0] = 0.52;
p[1] = 0.73;
std::cout << "u" << p.toString() << " = " << opEval->eval(alpha, p) << std::endl;
```

Obwohl *BaseOperationsFactory* ein sehr langer Bezeichner ist, die Zeilenbreite dadurch zu weit und weniger leserlich werden kann, ist die Ersetzung dennoch sinnvoll, so wie in Kapitel 4.4.4 beschrieben ist. Es hätten auch kürzere Bezeichner gewählt werden können. Jedoch gibt es in

anderen Modulen auch ähnliche Fabrikklassen. Um einen Namenskonflikt zu vermeiden, wurde eben dieser Bezeichner gewählt.

5. Zum Schluss sollte noch aufgeräumt werden. In der alten Version müssen Benutzer vier Objekte selbst löschen.

```
// Clean up
delete gridGenerator;
delete grid;
delete opEval;
delete opHier;
// Do not delete gridStorage !!
```

Der Kommentar am Schluss soll verdeutlichen, dass Benutzer das Objekt *gridStorage* nicht löschen sollen, weil es zum *grid*-Objekt gehört und von dessen Desktruktor gelöscht wird. Es könnte also eine doppelte Freigabe dadurch entstehen und das Programm zum Absturz bringen.

In der neuen Version wird die Variable *gridStorage* gar nicht benutzt, weswegen so ein Hinweis nicht nötig ist.

```
// Clean up
delete grid;
delete opEval;
delete opHier;
```

In der neuen Version ist es noch immer möglich auf das *gridStorage*-Objekt zuzugreifen, was dann eben zu den gleichen Fehlern führen kann. Aber in der neuen *SG++*-Version ist es für den üblichen Gebrauch nicht mehr nötig, auf das *gridStorage*-Objekt zuzugreifen. Dadurch erschwert die neue Schnittstelle den Fehlgebrauch.

Auf die Variable *gridGenerator* muss hingegen gar nicht mehr zugegriffen werden. Sie wird nur intern im *grid*-Objekt verwendet und wird auch wieder davon gelöscht, sodass Benutzer sich nicht mehr darum kümmern müssen.

Der vollständige Code beider Versionen ist im Anhang B.1 zu finden.

Im Anhang ist auch ein weiteres Beispiel mit Data-Mining zu finden in B.2. Weil aber die alte und neue Schnittstelle fürs Data-Mining sich nicht wesentlich unterscheiden, wird es hier nicht näher betrachtet. Der einzige große Unterschied liegt im Einlesen der *ARFF*-Dateien (5.4.2), welche in der neuen Version den Benutzercode deutlich verkürzt und weniger Boilerplatecode verursacht.

6.2. Leistungsvergleich

Als Kriterium für ein gutes API-Design wurde in Kapitel 4.2.3 festgelegt, dass eine benutzerfreundliche API die Performance der Bibliothek möglichst wenig beeinträchtigen soll. Schon ein Faktor von zwei kann bei der Berechnung großer Probleme viel ausmachen.

Für den Leistungsvergleich der alten mit der neuen *SG++*-Version wurden die Szenarien aus Kapitel 4.4.3 verwendet. Es wurden die gleichen Hardware- und Softwarekonfigurationen benutzt.

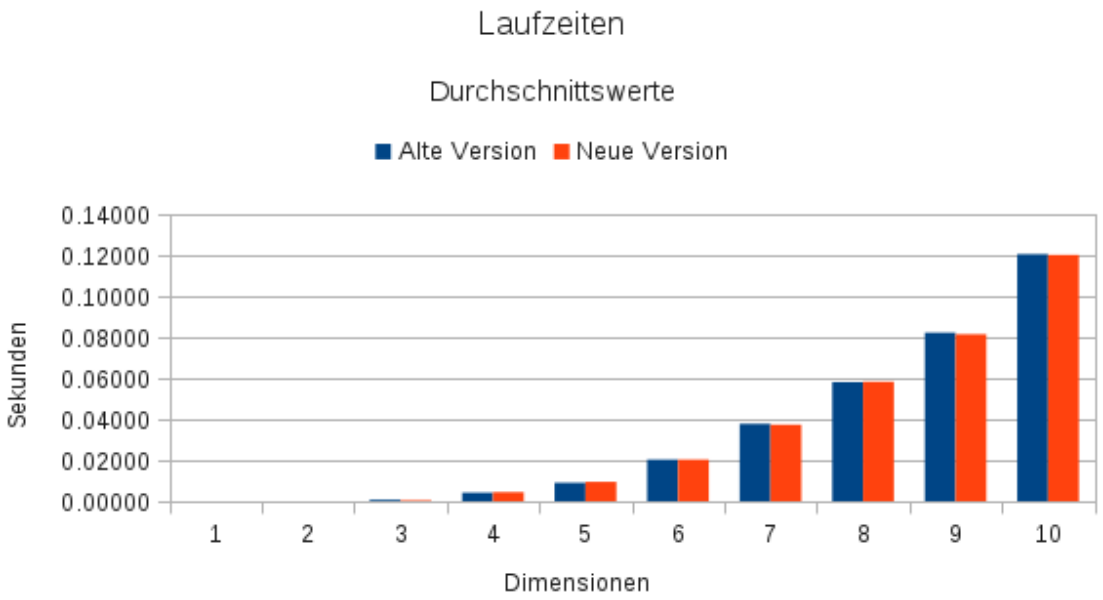

Abbildung 6.1.: Vergleich der Durchschnittslaufzeiten der alten und neuen *SG++*-Version in C++

6.2.1. Szenario: Quadratur mit Verfeinerung

Das erste Szenario ist wieder die Quadratur der Funktion in den Dimensionen 1 bis 10 mit 50 Verfeinerungen. Es wurden wieder 1000 Durchläufe pro Dimension genommen und jeweils der Durchschnitt gebildet.

C++

Wie dem Diagramm 6.1 entnommen werden kann, ist kein Unterschied zwischen den beiden Versionen zu erkennen.

Auch bei genauerer Betrachtung der Werte in der Tabelle 6.1 ist nicht festzustellen, dass eine Version konsistent schneller wäre als die andere.

Daraus kann abgeleitet werden, dass die beiden Versionen sich leistungsmäßig nicht unterscheiden. Das Ziel, das die neue Version nicht viel langsamer sein darf wie die alte, ist also erfüllt.

Python

Für die Leistungstests mit Python ist das Ergebnis ähnlich.

Dimensionen	Alte Version	Neue Version	Differenz (absolut)
1	0.00000	0.00000	0.00000
2	0.00000	0.00000	0.00000
3	0.00066	0.00041	0.00025
4	0.00419	0.00439	-0.00020
5	0.00898	0.00948	-0.00050
6	0.02033	0.02033	0.00000
7	0.03786	0.03741	0.00045
8	0.05820	0.05841	-0.00021
9	0.08239	0.08161	0.00078
10	0.12078	0.12036	0.00042

Tabelle 6.1.: Durchschnittslaufzeiten in Sekunden über 1000 Durchläufe: Alte gegen neue C++-Version

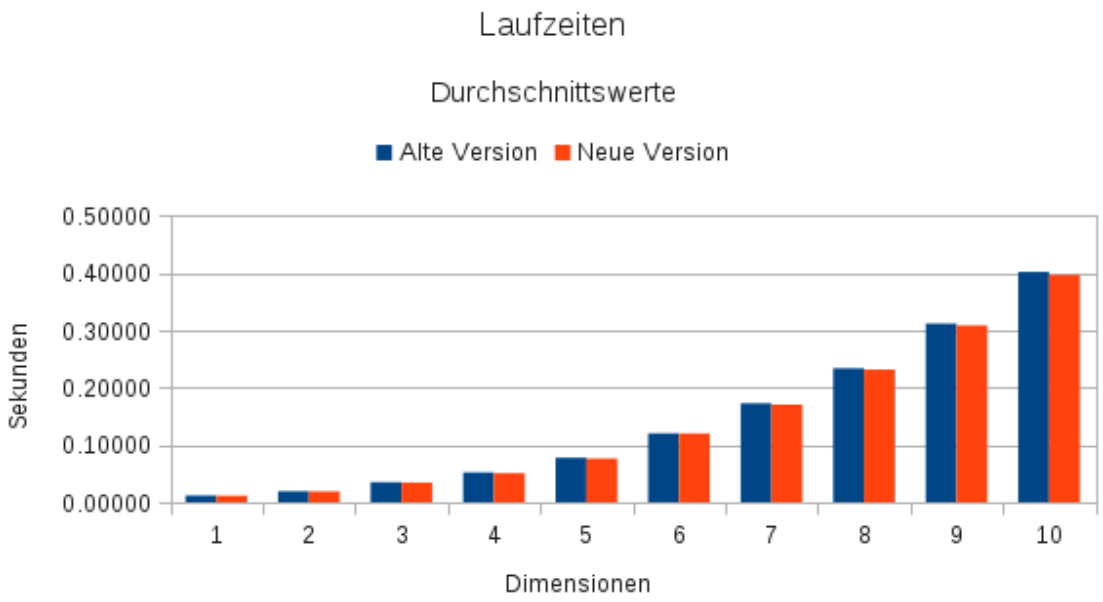

Abbildung 6.2.: Vergleich der Durchschnittslaufzeiten der alten und neuen *SG++*-Version in Python

Dimensionen	Alte Version	Neue Version	Differenz (absolut)	Differenz (relativ)
1	0.01162	0.01119	0.00043	3.70052%
2	0.01907	0.01875	0.00032	1.67803%
3	0.03487	0.03405	0.00082	2.35159%
4	0.05191	0.05049	0.00142	2.73550%
5	0.07771	0.07622	0.00149	1.91739%
6	0.12006	0.11976	0.00030	0.24988%
7	0.17275	0.17031	0.00244	1.41245%
8	0.23376	0.23138	0.00238	1.01814%
9	0.31170	0.30847	0.00323	1.03625%
10	0.40197	0.39676	0.00521	1.29612%

Tabelle 6.2.: Durchschnittslaufzeiten in Sekunden über 1000 Durchläufe: Alte gegen neue Python-Version

Der Unterschied zwischen den beiden Versionen ist minimal. Die Tabelle 6.2 zeigt die zum Diagramm zugehörigen Durchschnittswerte der Laufzeiten.

Die neue Version scheint ungefähr 1% bis 2% schneller zu sein. In Durchlauf 1 liegt der Unterschied sogar bei 3.70052% während in Durchlauf 6 der Unterschied nur ca. 0.24988% beträgt. Durch diese Schwankungen ist es schwer festzustellen, ob und um wie viel schneller die neue Version tatsächlich ist. Es kann aber festgehalten werden, dass die neue Python-Version nicht wesentlich langsamer ist als die alte.

Java

Für die Java-Tests sind die Ergebnisse interessanter.

Die genauen Durchschnittswerte können der Tabelle 6.3 entnommen werden.

Die neue Java-Version scheint, vor allem bei niedrigen Dimensionen, langsamer zu sein als die alte. Bei genauerer Betrachtung fällt jedoch auf, dass die Schwankungen zu groß sind, um definitive Aussagen treffen zu können. In Dimension 5 und 6 schneidet die neue Version sogar besser ab als die alte.

Wegen diesen inkonsistenten Ergebnissen ist es nur schwer möglich zu sagen, ob das Ziel erreicht wurde. Klar ist jedoch, dass die neue Version nicht um ein vielfaches langsamer ist.

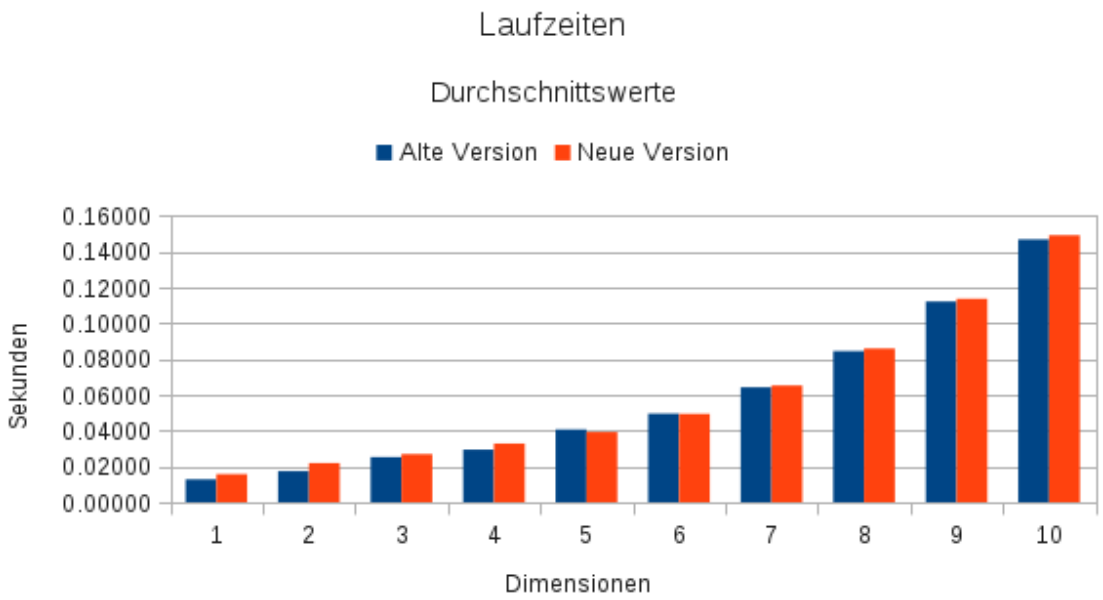

Abbildung 6.3.: Vergleich der Durchschnittslaufzeiten der alten und neuen *SG++*-Version in Java

Dimensionen	Alte Version	Neue Version	Differenz (absolut)	Differenz (relativ)
1	0.01278	0.01564	-0.00286	-22.37872%
2	0.01737	0.02186	-0.00449	-25.84917%
3	0.02521	0.02691	-0.00170	-6.74336%
4	0.02938	0.03277	-0.00339	-11.53846%
5	0.04060	0.03920	0.00140	3.44828%
6	0.04953	0.04934	0.00019	0.38361%
7	0.06416	0.06519	-0.00103	-1.60536%
8	0.08439	0.08585	-0.00146	-1.73006%
9	0.11209	0.11364	-0.00155	-1.38282%
10	0.14677	0.14912	-0.00235	-1.60114%

Tabelle 6.3.: Durchschnittslaufzeiten in Sekunden über 1000 Durchläufe: Alte gegen neue Java-Version

Anmerkung

Während den Leistungstests fiel auf, wie einfach es war in der alten *SG++*-Version ineffizienten Python- und Java-Code zu schreiben.

Zum Beispiel wurde in den Laufzeittests in Python das folgende Codefragment benutzt, um den Koeffizientenvektor zu initialisieren:

```
gridStorage = grid.getStorage()

# Refine adaptively
 for i in xrange(50):
 # Set function values in alpha
  p = [0] * dim

  for j in xrange(grid.getSize()):
   gp = gridStorage.get(j)

   for k in xrange(dim):
    p[k] = gp.abs(k)

    alpha[j] = f(p)
```

Bei vorherigen Versuchen die Laufzeit zu messen wurde aber folgender Code verwendet:

```
# Refine adaptively
 for i in xrange(50):
 # Set function values in alpha
  p = [0] * dim

  for j in xrange(grid.getSize()):
   gridStorage = grid.getStorage()
   gp = gridStorage.get(j)

   for k in xrange(dim):
    p[k] = gp.abs(k)

    alpha[j] = f(p)
```

Die beiden Fragmente sind fast identisch bis auf die Anweisung *gridStorage = grid.getStorage()*. Das Verschieben dieser Anweisung in die innerste Schleife hat den Effekt, dass die alte Python-Version konsistent langsamer war als die neue Version.

Die Ergebnisse dieses Tests können der Tabelle 6.4 entnommen werden.

Auch in Java ergab sich ein ähnliches Bild, wie in der Tabelle 6.5 zu sehen ist.

Vor allem bei niedrigen Dimensionen, bei denen das Verhältnis der Anzahl der Bibliotheksaufrufe zur Rechenzeit höher ist, als bei höheren Dimensionen, wird deutlich welche Auswirkungen eine falsch gesetzte Anweisung haben kann. Die neue *SG++*-Schnittstelle erschwert es Benutzern solchen ineffizienten Code zu schreiben, weil nicht mehr auf das *GridStorage*-Objekt zugegriffen werden muss. Das kann also als weiteren Vorteil für die neue *SG++*-Version gewertet werden. Vor allem wenn eine

Dimensionen	Alte Version	Neue Version	Differenz (absolut)	Differenz (relativ)
1	0.01187	0.01119	0.00068	5.72873%
2	0.02259	0.01875	0.00384	16.99867%
3	0.03850	0.03405	0.00445	11.55844%
4	0.05795	0.05049	0.00746	12.87317%
5	0.08545	0.07622	0.00923	10.80164%
6	0.13194	0.11976	0.01218	9.23147%
7	0.18684	0.17031	0.01653	8.84714%
8	0.25138	0.23138	0.02000	7.95608%
9	0.33194	0.30847	0.02347	7.07055%
10	0.42582	0.39676	0.02906	6.82448%

Tabelle 6.4.: Durchschnittslaufzeiten in Sekunden über 1000 Durchläufe: Alte gegen neue Python-Version

Dimensionen	Alte Version	Neue Version	Differenz (absolut)	Differenz (relativ)
1	0.01871	0.01564	0.00307	16.40834%
2	0.02521	0.02186	0.00335	13.28838%
3	0.03196	0.02691	0.00505	15.80100%
4	0.03615	0.03277	0.00338	9.34993%
5	0.04208	0.03920	0.00288	6.84411%
6	0.05233	0.04934	0.00299	5.71374%
7	0.06796	0.06519	0.00277	4.07593%
8	0.08945	0.08585	0.00360	4.02459%
9	0.11697	0.11364	0.00333	2.84688%
10	0.15231	0.14912	0.00319	2.09441%

Tabelle 6.5.: Durchschnittslaufzeiten in Sekunden über 1000 Durchläufe: Alte gegen neue Java-Version

Durchlauf	Alte Version	Neue Version
1	129.8479940	130.4987410
2	129.5814580	133.5422570
3	130.5573100	133.7820900
4	133.5512930	133.9457490
5	134.1095310	134.1251340
6	133.6839440	133.5340050
7	135.0782210	134.3887520
8	135.7193620	133.4747690
9	133.4698910	133.6526320
10	133.8273090	133.8615580
Durchschnitt	132.9426313	133.4805687

Tabelle 6.6.: Laufzeiten in Sekunden: Alte gegen neue C++-Version

große Anzahl kleinerer Probleme gelößt werden muss, sollte die Performance nicht zu stark von den Programmierfehlern der Benutzer abhängen.

6.2.2. Szenario: Datamining

Im zweiten Szenario werden wieder die sieben-dimensionalen Testdaten zum Anlernen verwendet, um Leistungsunterschiede zu messen. Es wurden wieder 10 Durchläufe gemessen und deren Durchschnitt gebildet.

C++

Die Ergebnisse der Leistungstests der C++-Versionen können der Tabelle 6.6 entnommen werden.

Dem Ergebnis nach zu urteilen ist die neue Version um ca. 0.5379374 Sekunden (0.40%) langsamer als die alte. Jedoch ist die Abweichung zu gering um eine Aussage treffen zu können, welche Version nun tatsächlich schneller ist. Beide Versionen scheinen in etwa gleich schnell zu sein, womit bestätigt ist, dass die Änderungen an der Bibliothek die Performance nicht zu sehr beeinträchtigt haben.

Durchlauf	Alte Version	Neue Version
1	129.74234319	133.38559699
2	131.88920403	131.66770387
3	134.44257116	132.13581896
4	134.22002912	136.64736295
5	134.60731101	136.10028791
6	134.19060087	136.36841488
7	134.06077004	137.13743401
8	134.23229194	135.67673802
9	134.09691787	136.23624587
10	135.18614197	136.02787995
Durchschnitt	133.66681812	135.13834834

Tabelle 6.7.: Laufzeiten in Sekunden: Alte gegen neue Python-Version

Python

Für Python war es nicht möglich zu vergleichen, welche Auswirkungen die Änderungen an der Bibliothek auf die Performance haben. Es fehlen leider die *SWIG*-Schnittstellen. *SG++* bietet jedoch eine Alternative an. Es wurde Python-Code geschrieben, welche ähnliche Funktionalitäten anbieten und den entsprechenden C++-Code aufrufen. Dieser Ansatz macht die Bibliothek schlechter wartbar und ist aus Benutzersicht inkonsistent mit den restlichen Schnittstellen, weil beispielsweise andere Bibliotheken eingebunden werden müssen als bei der C++ und die Verwendung der Schnittstelle einfach anders ist.

Auch wenn mit diesem Test nich analysiert werden kann, ob die Umstellungen in der Bibliothek die Performance beeinträchtigen, ist dieser Leistungstest dennoch sinnvoll. Hier soll aufgezeigt werden, ob es sich lohnt eigene Python-Klassen zu schreiben oder ob die Generierung von *SWIG*-Schnittstellen nicht nur softwaretechnisch besser, sondern auch schneller ist.

Die Tabelle 6.7 zeigt die Unterschiede in den Zeiten in Sekunden:

Im Durchschnitt ist die neue Version um etwa 1.4715302229 Sekunden, also ungefähr 1% langsamer. Dieser Unterschied ist zwar etwas größer als bei den C++-Tests, liegt aber jedoch noch im Rahmen. Der zusätzliche Aufwand scheint dafür aber nicht gerechtfertigt zu sein.

Das bedeutet aber nicht, dass es generell eine schlechte Idee ist, eigene Wrapper zu schreiben. Dieses Beispiel ist sehr rechenintensiv und die Funktionsaufrufe im Verhältnis dazu gering. Bei einem umgekehrten Verhältnis könnte ein spürbarer Unterschied bemerkt werden. Aber für das Data-Mining in *SG++*, wo diese Annahme üblicherweise nicht gilt, ist es nicht besonders lohnenswert.

Durchlauf	Alte Version	Neue Version
1	121.78876854	121.650045404
2	122.21400997	125.124958401
3	124.40285854	125.828040391
4	125.00301967	125.139306179
5	127.12185997	125.388435349
6	124.75443722	125.239889909
7	125.41619155	125.286968906
8	125.56542892	125.006250818
9	127.11946024	125.041518352
10	125.23826356	126.140401897
Durchschnitt	124.8624298173	124.9845815606

Tabelle 6.8.: Laufzeiten in Sekunden: Alte gegen neue Java-Version

Java

In der alten Version von *SG++* gibt es leider gar keine Schnittstellen für Java um Datamining zu betreiben. Für die alte Version wurde deswegen speziell eine Java-Schnittstelle für das Data-Mining generiert. Die Ergebnisse dieses Tests sind wieder etwas eindeutiger, wie der Tabelle 6.8 zu entnehmen ist.

Den Ergebnissen nach zu urteilen ist die neue Version im Durchschnitt gerade einmal um ca. 0.1% langsamer, was einen nicht signifikanten Unterschied darstellt.

7. Zusammenfassung, Fazit und Ausblick

Das Ziel dieser Arbeit war es der Numerikbibliothek *SG++* einem Refactoring zu unterziehen und die Softwarequalität zu steigern, ohne dabei deutliche Performancenachteile zu erhalten. Dabei wurde vor allem die Programmierschnittstelle genauer untersucht, damit die Verwendung der Bibliothek vereinfacht wird und Benutzer effizienter damit arbeiten können.

Dafür wurden zuerst Richtlinien definiert, die vorgeben, wie eine gute Schnittstelle auszusehen hat. Es wurden Techniken vorgestellt um die Richtlinien umzusetzen und es wurde erklärt, wie objektorientierte Konzepte eingesetzt werden können und welche Vorteile sie bringen. Zusätzlich wurde auch die Unterstützung von anderen Programmiersprachen untersucht. Es wurde beschrieben, welche Vorteile in der Programmierung Java und Python gegenüber C++ haben. Mit Leistungstest wurde gezeigt, dass die Auswirkungen auf die Performance gering sind, wenn aus Java und Python die entsprechenden C++-Funktionen aufgerufen werden. Für die drei Programmiersprachen Java, Python und C++ wurden Messungen mit wenig rechenintensiven Beispielen und mit sehr rechenintensiven Beispielen durchgeführt und verglichen, mit dem Ergebnis, dass eine Unterstützung mehrerer Programmiersprachen lohnenswert sein kann. Weiterhin wurde darauf eingegangen, welche Einschränkungen im Design der C++-Schnittstelle die Unterstützung von Java und Python mit sich bringen.

Mit den vorgestellten Techniken und Konzepten wurde dann *SG++* umstrukturiert. Mit dem Fokus auf die beiden grundlegenden Module *Base* und *Solver*, sowie dem Modul *Datadriven* für Data-Mining-Anwendungen wurde beispielhaft erläutert, wo die Probleme liegen und wie diese gelöst wurden, um mit den aufgestellten Richtlinien konform zu gehen. Dabei wurden kleinere Probleme, wie beispielsweise Ausnahmebehandlungen und Fehlermeldungen, angegangen, als auch größere Probleme, welche zu Fehlern und Abstürzen führen könnten.

Zum Schluss wurde noch die Leistung der *SG++*-Version, welche einem Refactoring unterzogen wurde, verglichen mit der unveränderten *SG++*-Version. Dabei wurde festgestellt, dass die Änderungen an der Bibliothek keine signifikanten Leistungseinbußen mit sich brachten. Bei den Leistungstest konnte nur ein Performanceverlust von maximal 1% eindeutig festgestellt werden.

Fazit

Der Versuch, die Bibliothek *SG++* einem Refactoring zu unterziehen, kann als gelungen bezeichnet werden. Die neue Schnittstelle erlaubt es, kürzeren und verständlicheren Code zu schreiben, ohne einen merklichen Performanceverlust erleiden zu müssen. Ebenso wurden einige Fehlerquellen im Code verbessert, welche erst durch das Refactoring sichtbar wurden.

Das Refactoring einer Software bringt immer einige Schwierigkeiten mit sich. Zum einen kann jede Änderung in einer der grundlegenden Module große Auswirkungen auf die anderen Module haben. Dadurch sollte man sich gut überlegen, ob sich die große Zeitinvestition überhaupt lohnt. Ein weiteres Problem beim Refactoring der Bibliotheken ist, dass sich die Schnittstellen ändern können. Falls die Änderungen zu groß sein sollten, könnte es nicht einfach sein Benutzer der alten Version zum Umsteigen auf die neue zu überzeugen. Eine stabile API ist nun einmal auch ein wichtiges Qualitätsmerkmal, auf welches Benutzer achten.

Mit dieser Diplomarbeit wurde also gezeigt, dass es sich auch in der Numerik durchaus lohnen kann, auf softwaretechnische Qualitäten zu achten. Oft erfordert dies jedoch eine hohe Disziplin der Implementierer, welche aber durch Werkzeuge wie automatische Stylechecker und durch Codereview sichergestellt werden können.

Ausblick

Das Refactoring von *SG++* ist noch nicht abgeschlossen. Die weiteren Verbesserungsvorschläge in Kapitel 5.5 könnten noch umgesetzt werden. Außerdem müssen noch die weiteren Module, welche nicht näher betrachtet wurden, einem Refactoring unterzogen werden.

Das im Unterkapitel 4.4.3 Phänomen, welches Java schneller als C++ darstellt, müsste ebenso noch weiter untersucht werden. Eventuell deckt es noch weitere Fehler in der Bibliothek auf.

Diese Diplomarbeit soll nur einen Anfang darstellen. Die konsequente Umgestaltung und Qualitätsverbesserung wird noch einiges an Zeit kosten. Wenn man aber dazu bereit ist, diese zu investieren, dann kann aus *SG++* nicht nur eine schnelle, sondern auch eine benutzerfreundliche Numerikbibliothek werden.

A. Vererbungshierarchie im Modul Datadriven

A.1. Vererbungshierarchie der alten SG++-Version

A.1.1. Hierarchie für einfache Präzision

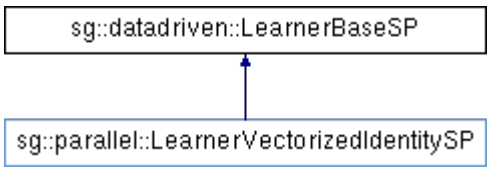

Abbildung A.1.: Alte Klassenhierarchie für einfache Präzision. Entnommen aus der Doxygen-Dokumentation.

A.1.2. Hierarchie für doppelte Präzision

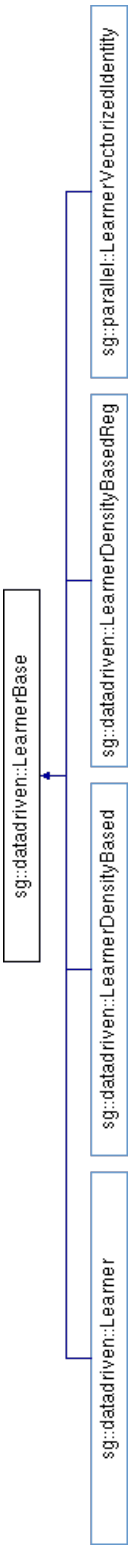

Abbildung A.2.: Alte Klassenhierarchie für doppelte Präzision. Entnommen aus der Doxygen-Dokumentation.

A.2. Vererbungshierarchie der neuen SG++-Version

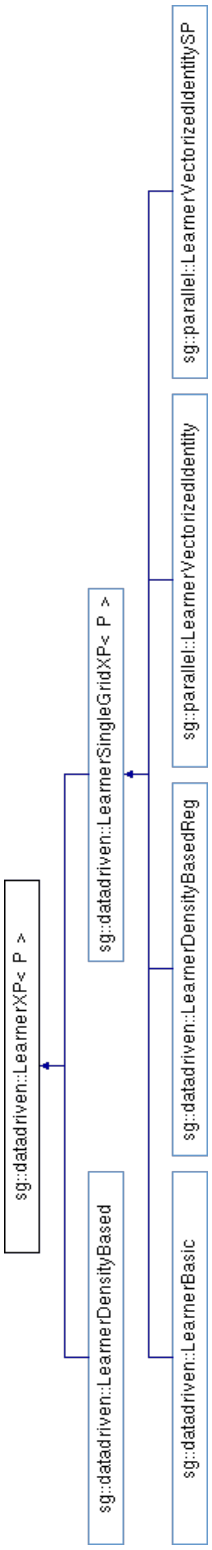

Abbildung A.3.: Neue Klassenhierarchie. Entnommen aus der Doxygen-Dokumentation.

B. Beispielcode zum Vergleich der alten und neuen SG++-Version

B.1. Interpolation und Evaluation

Im Folgenden ist der vollständige Code aus dem Beispiel in Kapitel 6.1 in beiden Versionen zu sehen.

B.1.1. Beispielcode mit der alten Schnittstelle

```
#include <iostream>

#include "base/datatypes/DataVector.hpp"
#include "base/grid/Grid.hpp"
#include "base/grid/GridStorage.hpp"
#include "base/grid/generation/GridGenerator.hpp"
#include "base/operation/OperationHierarchisation.hpp"
#include "base/operation/OperationEval.hpp"
#include "base/operation/BaseOpFactory.hpp"

// Function to interpolate
double f(double x0, double x1) {
  return 16.0 * (x0-1)*x0 * (x1-1)*x1;
}

int main() {
  // Create a two-dimensional piecewise bi-linear grid
  int dim = 2;
  sg::base::Grid* grid = sg::base::Grid::createLinearGrid(dim);

  // Create regular grid on level 3
  int level = 3;
  sg::base::GridGenerator* gridGenerator = grid->createGridGenerator();
  gridGenerator->regular(level);

  // Create coefficient vector
  sg::base::DataVector alpha(grid->getSize());
  alpha.setAll(0.0);

  // Set function values in alpha
  sg::base::GridIndex* gp;
  sg::base::GridStorage* gridStorage = grid->getStorage();
  for (int i=0; i < grid->getSize(); ++i) {
    gp = gridStorage->get(i);
    alpha[i] = f(gp->abs(0), gp->abs(1));
```

```
  }

  // Hierarchize
  sg::base::OperationHierarchisation* opHier =
      sg::op_factory::createOperationHierarchisation(*grid);
  opHier->doHierarchisation(alpha);

  // Evaluation operation
  sg::base::OperationEval* opEval = sg::op_factory::createOperationEval(*grid);
  sg::base::DataVector p(dim);

  // Evaluate on (0.52, 0.73)
  p[0] = 0.52;
  p[1] = 0.73;
  std::cout << "u" << p.toString() << " = " << opEval->eval(alpha, p) << std::endl;

  // Clean up
  delete gridGenerator;
  delete grid;
  delete opEval;
  delete opHier;
  // Do not delete gridStorage !!

  return 0;
}
```

B.1.2. Beispielcode mit der neuen Schnittstelle

```
#include <iostream>

#include "base/datatypes/DataVector.hpp"
#include "base/grid/Grid.hpp"
#include "base/grid/GridFactory.hpp"
#include "base/operation/OperationHierarchisation.hpp"
#include "base/operation/OperationEval.hpp"
#include "base/operation/BaseOperationsFactory.hpp"

// Function to interpolate
double f(double x0, double x1) {
  return 16.0 * (x0-1)*x0 * (x1-1)*x1;
}

int main() {
  // Set up configuration for a regular two-dimensional piecewise bi-linear grid on level 3
  sg::base::RegularGridConfiguration gridConfig;
  gridConfig.type = sg::base::Grids::Linear;
  gridConfig.dim = 2;
  gridConfig.level = 3;

  // Create grid
  sg::base::Grid* grid = sg::base::GridFactory::createGrid(gridConfig);

  // Create coefficient vector
  sg::base::DataVector alpha(grid->getPointCount());
  alpha.setAll(0.0);
```

```
  // Set function values in alpha
  sg::base::GridIndex* gp;
  for (int i=0; i < grid->getPointCount(); ++i) {
    gp = grid->getIndex(i);
    alpha[i] = f(gp->abs(0), gp->abs(1));
  }

  // Hierarchize
  sg::base::OperationHierarchisation* opHier =
      sg::BaseOperationsFactory::createOperationHierarchisation(*grid);
  opHier->doHierarchisation(alpha);

  // Evaluation operation
  sg::base::OperationEval* opEval = sg::BaseOperationsFactory::createOperationEval(*grid);
  sg::base::DataVector p(gridConfig.dim);

  // Evaluate on (0.52, 0.73)
  p[0] = 0.52;
  p[1] = 0.73;
  std::cout << "u" << p.toString() << " = " << opEval->eval(alpha, p) << std::endl;

  // Clean up
  delete grid;
  delete opEval;
  delete opHier;

  return 0;
}
```

B.2. Data-Mining

Das zweite Szenario ist ein generisches Data-Miningbeispiel. Es müssten nur noch die Pfade für die Ein- und Ausgabedateien angepasst werden um ein vollständiges Beispiel zu erhalten.

B.2.1. Beispielcode mit der alten Schnittstelle

```
#include <iostream>
#include <string>

#include "base/datatypes/DataVector.hpp"
#include "base/datatypes/DataMatrix.hpp"
#include "base/grid/Grid.hpp"
#include "solver/TypesSolver.hpp"
#include "datadriven/application/Learner.hpp"
#include "datadriven/tools/TypesDatadriven.hpp"
#include "datadriven/tools/ARFFTools.hpp"

int main() {
  // Read dataset from ARFF
  std::string data_path = "./path/to/input.arff";
  sg::datadriven::ARFFTools data;
```

```
  size_t dim = data.getDimension(data_path);
  size_t instanceCount = data.getNumberInstances(data_path);

  sg::base::DataVector* classes = new sg::base::DataVector(instanceCount);
  data.readClasses(data_path, *classes);

  sg::base::DataMatrix* trainingData = new sg::base::DataMatrix(instanceCount, dim);
  data.readTrainingData(data_path, *trainingData);

  // Define learner
  sg::datadriven::LearnerRegularizationType regType = sg::datadriven::Identity;
  sg::datadriven::LearnerBase* learner = new sg::datadriven::Learner(regType, true);

  // Set up grid configurations
  sg::base::RegularGridConfiguration gridConfig;
  gridConfig.dim_ = dim;
  gridConfig.level_ = 5;
  gridConfig.type_ = sg::base::Linear;

  // Set up solver configurations
  sg::solver::SLESolverConfiguration solverConfig;
  solverConfig.eps_    = 0.0001;
  solverConfig.maxIterations_ = 100;
  solverConfig.threshold_ = 0.001;
  solverConfig.type_   = sg::solver::CG;

  // Set up adaptivity configurations
  sg::base::AdpativityConfiguration adaptivityConfig;
  adaptivityConfig.maxLevelType_ = false;
  adaptivityConfig.noPoints_ = 100;
  adaptivityConfig.numRefinements_ = 15;
  adaptivityConfig.percent_ = 100.0;
  adaptivityConfig.threshold_ = -1.0;

  // Train
  double regularization = 1.0;
  sg::datadriven::LearnerTiming learnerTiming;
  learnerTiming = learner->train(*trainingData, *classes, gridConfig, solverConfig, solverConfig,
      adaptivityConfig, true, regularization);

  // Write grid
  learner->dumpGrid("./path/to/output.dat");

  // Clean up
  delete classes;
  delete trainingData;
  delete learner;

  return 0;
}
```

B.2.2. Beispielcode mit der neuen Schnittstelle

```
#include <iostream>
#include <string>
```

```
#include "base/grid/GridTypes.hpp"
#include "base/grid/GridConfigurations.hpp"
#include "solver/SolverTypes.hpp"
#include "datadriven/application/LearnerBasic.hpp"
#include "datadriven/tools/DatadrivenTypes.hpp"
#include "datadriven/tools/ARFFTools.hpp"
#include "datadriven/tools/Dataset.hpp"

int main() {
  // Read dataset from ARFF
  const std::string data_path = "./path/to/input.arff";
  sg::datadriven::Dataset* dataset = sg::datadriven::ARFFTools::getDataset(data_path);

  // Define learner
  sg::datadriven::LearnerRegularizationType regType = sg::datadriven::Identity;
  sg::datadriven::Learner* learner = new sg::datadriven::LearnerBasic(regType, true);

  // Set up grid configuration
  sg::base::RegularGridConfiguration gridConfig;
  gridConfig.dim = dataset->getDimension();
  gridConfig.level = 5;
  gridConfig.type = sg::base::Grids::Linear;

  // Set up solver configuration
  sg::solver::SLESolverConfiguration solverConfig;
  solverConfig.eps    = 0.0001;
  solverConfig.maxIterations = 100;
  solverConfig.threshold = 0.001;
  solverConfig.type   = sg::solver::SLESolvers::CG;

  // Set up adaptivity configuration
  sg::base::AdaptivityConfiguration adaptivityConfig;
  adaptivityConfig.maxLevelType = false;
  adaptivityConfig.maxPointCount = 100;
  adaptivityConfig.numberOfRefinements = 15;
  adaptivityConfig.percent    = 100.0;
  adaptivityConfig.threshold  = -1.0;

  // Train
  double regularization = 1.0;
  sg::datadriven::LearnerTiming learnerTiming;
  learnerTiming = learner->train(dataset, gridConfig, solverConfig, solverConfig,
      adaptivityConfig, true, regularization);

  // Write grid
  learner->dumpGrid("./path/to/output.dat");

  // Clean up
  delete dataset;
  delete learner;

  return 0;
}
```

Literaturverzeichnis

[ABL97] E. Arge, A. M. Bruaset, H. P. Langtangen. Object-oriented numerics. In *Numerical Methods and Software Tools in Industrial Mathematics*, S. 7–26. Birkhäuser Boston, 1997. (Zitiert auf Seite 17)

[BG04] H.-J. Bungartz, M. Griebel. Sparse Grids. *Acta Numerica*, 13:147–269, 2004. URL http://www5.in.tum.de/pub/bungartz04sparse.pdf. (Zitiert auf Seite 12)

[Bla08] J. Blanchette. *The Little Manual of API Design.* Trolltech, 2008. (Zitiert auf den Seiten 17 und 28)

[Blo06] J. Bloch. How to Design a Good API and Why It Matters. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, S. 506–507. ACM, New York, NY, USA, 2006. doi:10.1145/1176617.1176622. (Zitiert auf Seite 17)

[CS02] A. Chatzigeorgiou, G. Stephanides. Evaluating Performance and Power of Object-Oriented Vs. Procedural Programming in Embedded Processors. In J. Blieberger, A. Strohmeier, Herausgeber, *Ada-Europe*, Band 2361 von *Lecture Notes in Computer Science*, S. 65–75. Springer, 2002. (Zitiert auf Seite 17)

[Dox] Doxygen. Doxygen. http://www.stack.nl/~dimitri/doxygen/. (Zitiert auf Seite 15)

[Gar07] J. Garcke. Sparse Grids Tutorial. Technischer Bericht, Technische Universität Berlin, 2007. (Zitiert auf Seite 9)

[Lan09] H. P. Langtangen. *Python Scripting for Computational Science.* Springer Publishing Company, Incorporated, 3. Auflage, 2009. (Zitiert auf den Seiten 17 und 32)

[LL07] J. Ludewig, H. Lichter. *Software Engineering - Grundlagen, Menschen, Prozesse, Techniken.* dpunkt.verlag, 2007. (Zitiert auf Seite 7)

[LW94] B. H. Liskov, J. M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16, 1994. (Zitiert auf Seite 28)

[Mey] S. Meyers. How Non-Member Functions Improve Encapsulation. http://www.drdobbs.com/cpp/how-non-member-functions-improve-encapsu/184401197. (Zitiert auf Seite 74)

[Mey97] B. Meyer. *Object-oriented Software Construction.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2. Auflage, 1997. (Zitiert auf den Seiten 17 und 24)

[Pfl10] D. Pflüger. *Spatially Adaptive Sparse Grids for High-Dimensional Problems.* Verlag Dr. Hut, München, 2010. URL http://www5.in.tum.de/pub/pflueger10spatially.pdf. (Zitiert auf Seite 9)

[Red11] M. Reddy. *API Design for C++*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1. Auflage, 2011. (Zitiert auf den Seiten 17, 28 und 75)

[SCoa] SCons. SCons. http://www.scons.org/. (Zitiert auf Seite 15)

[SCob] SCons. SCons vs other build tools. http://www.scons.org/wiki/SconsVsOtherBuildTools. (Zitiert auf Seite 15)

[Str13] B. Stroustrup. *The C++ Programming Language*. Addison–Wesley, 4. Auflage, 2013. (Zitiert auf den Seiten 17 und 30)

[SWI] SWIG. SWIG Doukmentation. http://www.swig.org/Doc2.0/index.html. (Zitiert auf den Seiten 15, 39, 41 und 42)

[Tul08] J. Tulach. *Practical API Design: Confessions of a Java Framework Architect*. Definitive Guide Series. Apress, 2008. (Zitiert auf Seite 17)

[WSE+] B. Weinberger, C. Silverstein, G. Eitzmann, M. Mentovai, T. Landray. Google C++ Style Guide. http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Structs_vs._Classes. (Zitiert auf Seite 27)

Alle URLs wurden zuletzt am 07. 10. 2014 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