Deutsches Zentrum
für Luft- und Raumfahrt e.V.
Institut für Antriebstechnik

Fernuniversität Hagen
Fakultät für Mathematik und Informatik
Lehrgebiet für Kooperative Systeme
Betreuer: Prof. Dr. Christian Icking

Masterarbeit

Entwicklung eines objektorientierten und parallelisierten Gradient Enhanced Kriging Ersatzmodells

Andreas Schmitz
Studiengang Master in Praktischer Informatik

Köln, September 2013

FernUniversität in Hagen

DLR **Deutsches Zentrum für Luft- und Raumfahrt** e.V.
in der Helmholtz-Gemeinschaft

Hiermit versichere ich an Eides statt und durch meine Unterschrift, dass die vorliegende Arbeit von mir selbstständig, ohne fremde Hilfe angefertigt worden ist. Inhalte und Passagen, die aus fremden Quellen stammen und direkt oder indirekt übernommen worden sind, wurden als solche kenntlich gemacht. Ferner versichere ich, dass ich keine andere, außer der im Literaturverzeichnis angegebenen Literatur verwendet habe. Diese Versicherung bezieht sich sowohl auf Textinhalte sowie alle enthaltenden Abbildungen, Skizzen und Tabellen. Die Arbeit wurde bisher keiner Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Köln den 2.10.2013

Andreas Schmitz

Inhaltsverzeichnis

Nomenklatur

β	Unbekannter Erwartungswert beim Ordinary Kriging Verfahren
δ	Kronecker Delta
κ	Konditionszahl
λ	Diagonalaufschlag
$\mathbf{Cov}$	Kovarianzmatrix
$\mathbf{R}$	Korrelationsmatrix
σ^2	Varianz
$\mathrm{cov}(X)$	Kovarianz der Zufallsvariable X
$\mathrm{var}(X)$	Varianz der Zufallsvariable X
$\vec{\theta}$	Hyperparameter einer Korrelationsfunktion
$\vec{F}$	Der Vektor entspricht beim Ordinary-Kriging $\vec{1}$ und beim Gradient-Enhanced-Kriging sind die ersten $n-m$ Einträge Eins und für die restlichen m Einträge Null
$\vec{h}$	Euklidischer Abstandsvektor
$\vec{r}$	Korrelationsvektor
$\vec{x}$	Ortsvektor
$\vec{y}_s$	Vektor, welcher alle bekannten Stützstellen enthält
Ξ	Eigenwert einer Matrix
$c(X,Y)$	Korrelationsfunktion zwischen den Zufallsvariablen X und Y
$E(X)$	Erwartungswert der Zufallsvariable X
$F\left(\vec{x}\right)$	Fehlerfunktion an der Stelle $\vec{x}$
L	Likelihood Funktion

N	Multivariate Normalverteilung
w_i	Gewichte eines Kriging Modells
$y\left(\vec{x}\right)$	Bekannter Funktionswert oder Stützstelle an der Stelle $\vec{x}$
$y^*\left(\vec{x}\right)$	Geschätzter Funktionswert an der Stelle $\vec{x}$
k	Anzahl der freien Variablen eines Members
l	Index, welcher eine freie Variable bezeichnet
m	Anzahl der gegebenen partiellen Ableitungen
Member	Ein Satz freier Variablen mit dazugehöriger Zielfunktion, bspw. ein Satz geometrischer Parameter mit berechnetem Wirkungsgrad als Zielfunktion
n	Anzahl der beprobten Stützstellen, auch Member oder Samples genannt
p	Index, welcher eine freie Variable bezeichnet

Kurzfassung

Zur Erfüllung der im europäischen Strategiedokument „Flightpath 2050“ [N.A11] definierten Ziele in Hinblick auf die Reduzierung des Treibstoffverbrauchs und der CO_2-Emission ist eine weitere Steigerung des Wirkungsgrades von Verdichtern als zentrale Triebwerkskomponente erforderlich. Das Ziel ist die Auslegung und das Design von effizienten Verdichtern mit verbesserten und neuen Auslegungstechniken sowie Simulationsverfahren. Diese sind elementar mit der Qualität der Auslegung und der Zertifizierung verknüpft. Gerade durch das immer stärkere Ausloten und Ausnutzen der technischen Grenzen und Detailgestaltungen steigen die Anforderungen an die Auslegungswerkzeuge und die rechnerische Simulation erheblich an. Insbesondere die automatisierte Mehrzieloptimierung der Verdichterkomponenten ermöglicht dabei das Auffinden der optimalen Geometrien unter Berücksichtigung aller wesentlichen Auslegungskriterien.

Am Institut für Antriebstechnik wird daher ein solcher Optimierer namens AutoOpti entwickelt. Um eine Optimierung möglichst effizient zu gestalten, werden Ersatzmodelle verwendet, welche die Zielfunktion modellieren und damit den Optimierungsprozess beschleunigen. Eines der am häufigsten verwendeten Ersatzmodelle ist das Kriging Verfahren, welches beim Institut für Antriebstechnik bereits zum Einsatz kommt. Da dieses Verfahren immer mehr an Bedeutung gewonnen hat, sollte es innerhalb des Rahmens dieser Arbeit komplett neu entwickelt werden. Ziel dieser Neuentwicklung war es, den Code zu modularisieren, um das Verfahren schnell erweitern und verändern zu können. Zudem war es ein wichtiges Ziel, das Verfahren zu beschleunigen.

Um den Code für zukünftige Anwendungen modularer zu gestalten, wurde das gesamte Programm objektorientiert in C++ geschrieben. Außerdem wurde die Boost Bibliothek verwendet um mehr Möglichkeiten bei der Programmierung zu haben, beispielsweise bietet diese Bibliothek sehr flexible Funktionsobjekte. Durch Einsatz dieser Mittel und moderner Software Engineering Methoden war es möglich, den neuen Code so weit zu modularisieren, dass nahezu alle Teile des Programms austauschbar und einfach erweiterbar sind.

Der Code konnte um den Faktor 2 beschleunigt werden, dies wurde durch Minimierung der Speicherzugriffe und durch Nutzung effizienter CPU-Befehle erreicht. Zudem war es möglich, ein neues Trainingsverfahren höherer Ordnung zu implementieren. Die Verwendung dieses Trainingsverfahrens beschleunigte den Code nochmals um den Faktor 3.

Um die Ergebnisse zu validieren, wurde ein Test und Benchmark durchgeführt, in dem die Ergebnisse und auch der Geschwindigkeitszuwachs bestätigt werden konnten.

1 Einleitung

Diese Arbeit entstand am Deutschen Zentrum für Luft- und Raumfahrt im Institut für Antriebstechnik - Abteilung Fan und Verdichter in Köln. In diesem Kapitel werden die Hintergründe und die Motivation der Masterarbeit erklärt. Im ersten Teil werden die Grundlagen von Flugzeugtriebwerken erläutert, da sich das Institut für Antriebstechnik hauptsächlich mit der Optimierung bestehender Triebwerke und der Entwicklung neuer Konzepte beschäftigt. Im zweiten Teil geht es dann um den Optimierungsprozess, wie er im Institut bisher durchgeführt wird und wo bei diesem Prozess Ersatzmodelle eine Rolle spielen. Im dritten Teil wird die Aufgabenstellung und Entwicklungsumgebung spezifiziert.

1.1 Grundlagen Gasturbinen und Flugzeugtriebwerke

In diesem Abschnitt soll anhand von Abbildung 1.1 der grundlegende Aufbau eines einfachen Flugzeugtriebwerks erklärt werden. Die entsprechenden Baugruppen werden der Reihe nach beschrieben.

Verdichter

Durch den Verdichter erhält die strömende Luftmasse Energie in Form von Druck. Dies geschieht in den diffusorförmigen Zwischenräumen der Kompressorschaufeln,

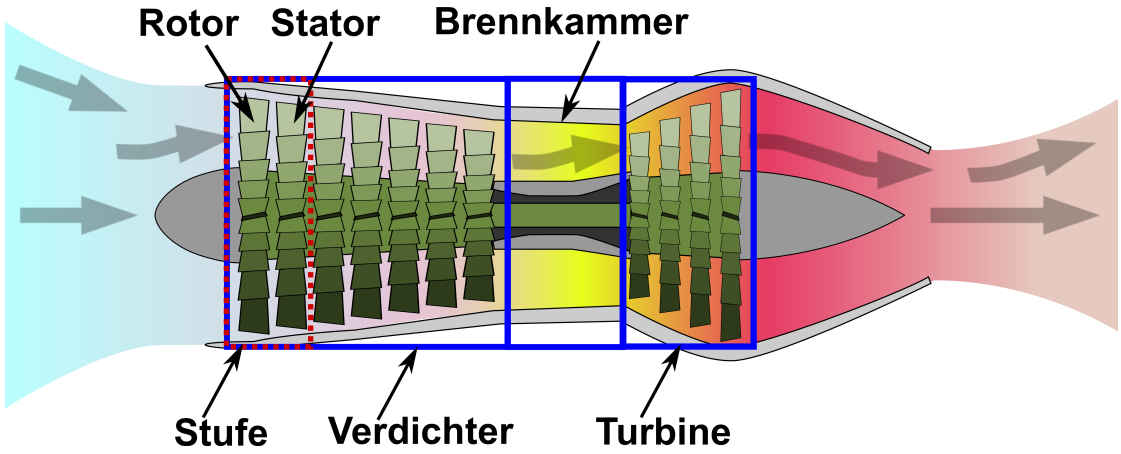


Abbildung 1.1: Schematischer Querschnitt eines Turbojet Triebwerks, http://commons.wikimedia.org/wiki/File:Turbojet_operation-axial_flow-es.svg

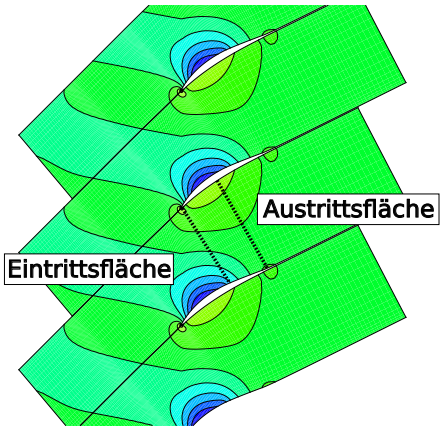


Abbildung 1.2: Beispielhaftes Verdichterschaufelgitter mit entsprechender qualitativer Druckverteilung

siehe 1.2. Nach dem Bernoullischen Gesetz erhöht sich in einem Kanal, welcher an Querschnittsfläche zunimmt, der statische Druck, bei sinkender Strömungsgeschwindigkeit. In 1.2 strömt die Luft von links nach rechts und die Eintrittfläche zwischen 2 Schaufelprofilen ist geringer als die Austrittsfläche, daher die statische Druckerhöhung.

Die nun verlorene kinetische Energie wird in einer Rotorstufe durch Rotation der Schaufeln wieder zugeführt. Eine komplette Verdichterstufe eines Axialverdichters besteht also aus einer Rotorstufe, in der sowohl Druck und Temperatur als auch die Geschwindigkeit steigen und einer Statorstufe, in der der Druck zu Ungunsten der Geschwindigkeit steigt.

Quellen: [Brä04, Lon08, Cum04]

Brennkammer

Durch die starke Verdichtung, steigt auch die Temperatur der Luft an und strömt anschließend in die Brennkammer, wo der Luft Kraftstoff zugemischt wird. Das Gemisch wird beim Triebwerksstart durch eine Zündung entflammt, danach ist die Verbrennung kontinuierlich. Durch die Verbrennung steigen die Temperatur und Druck stark an und Temperaturen von bis zu 1800 Kelvin belasten das Material erheblich. Um diesen Belastungen stand zu halten, müssen hochbelastbare Materialien verwendet und die Brennkammerwände gekühlt werden. Dies erfolgt durch Einleitung von Verdichterluft in die Brennkammer, die sogenannte Sekundärluft. Durch diese entsteht ein dünner Luftfilm, welcher die Brennkammerwände zusätzlich kühlt.

Quellen: [Brä04, Lon08]

Turbine

Die heißen Gase aus der Brennkammer treffen schließlich auf die Turbine. Die Energie im Gas wird in mechanische Energie umgewandelt und über eine Welle wiederum an

den Kompressor übertragen und treibt diesen an. Insbesondere die ersten Turbinenstufen müssen hohen thermischen Belastungen standhalten. Aus diesem Grund werden die Schaufeln mit Sekundärluft gekühlt und hoch wärmebeständige Materialien verwendet.

Quellen: [Brä04, Lon08]

1.2 Automatisierte Optimierung im DLR

Da der weltweite Flugverkehr sich seit dem Jahr 2000 fast verdreifacht hat, ergeben sich besondere Problemstellungen z.B. die Brennstoffknappheit, die Kapazitäten der Flughäfen und -routen sowie die steigende Umweltbelastung durch Lärm- und Schadstoffemissionen. Diese stehen immer mehr im Interesse der Öffentlichkeit. Um den genannten Problem gewachsen zu sein, sind weitreichende Forschungen in den verschiedensten Gebieten notwendig. Immer häufiger werden automatisierte Optimierer eingesetzt, um Triebswerkskomponenten hinsichtlich der genannten Ziele zu optimieren. Im Institut für Antriebstechnik des DLR wurde ein solcher automatisierter Optimierer namens AutoOpti [Voß08, Voß10] entwickelt, welcher mittlerweile auch für Optimierungen außerhalb des Verdichterdesigns eingesetzt wird. Da die aerodynamische Auslegung von Triebswerkskomponenten immer einen Kompromiss zwischen den unterschiedlichsten Anforderungen darstellt, wurde bei der Entwicklung von AutoOpti besonderer Wert auf die Möglichkeit zur simultanen Optimierung mehrerer Zielfunktionen gelegt. Ein typisches Optimierungsziel / Zielfunktion für einen Verdichter wäre z.B. die simultane Maximierung des Wirkungsgrads (Verhältnis von nutzbarer Energie zu genutzter Energie) bei gleichzeitiger Maximierung des Druckverhältnisses (Verhältnis vom Druck am Eintritt des Verdichters zum Druck am Austritt des Verdichters). Um die Zielfunktion(en) zu optimieren, kann der Optimierungsalgorithmus eine bestimmte Anzahl von freien Variablen verändern. Bei einem Verdichter wären das typischerweise Parameter, welche die Schaufelgeometrien variieren, beispielsweise die Länge der Schaufel oder die Dicke. In einer realen Optimierung werden oftmals hunderte von freien Variablen verwendet. Ein Satz von freien Parametern mit der entsprechenden Zielfunktion bezeichnet man als Member. Eine automatisierte Optimierung, wie sie im Institut für Antriebstechnik in Köln ausgeführt wird, basiert grundsätzlich auf einer Evolutionsstrategie [Rec94]. Da die Optimierung meist auf einem Rechencluster ausgeführt wird, ist es sinnvoll, die Arbeit in verschiedene Prozesse einzuteilen, damit diese auf unterschiedlichen Rechnern verteilt werden können. In diesem Fall gibt es zwei verschiedene Arten von Prozessen, einen Root Prozess und mehrere Slave Prozesse.

- Der Root Prozess ist für die Verwaltung und Steuerung der Optimierung zuständig und läuft auf nur einem Rechner.
- Die Slave Prozesse, werden am Anfang der Optimierung durch den Root Prozess gestartet. In der Regel läuft jeder Slave Prozess auf einem anderen Rechner, da diese Prozesse den größten Teil der Arbeit übernehmen. Die Slave Prozesse bekommen vom Root Prozess freie Variablen gesendet und berechnen die dazugehörige Zielfunktion. Wie diese Berechnung abläuft, ist von Optimierung zu

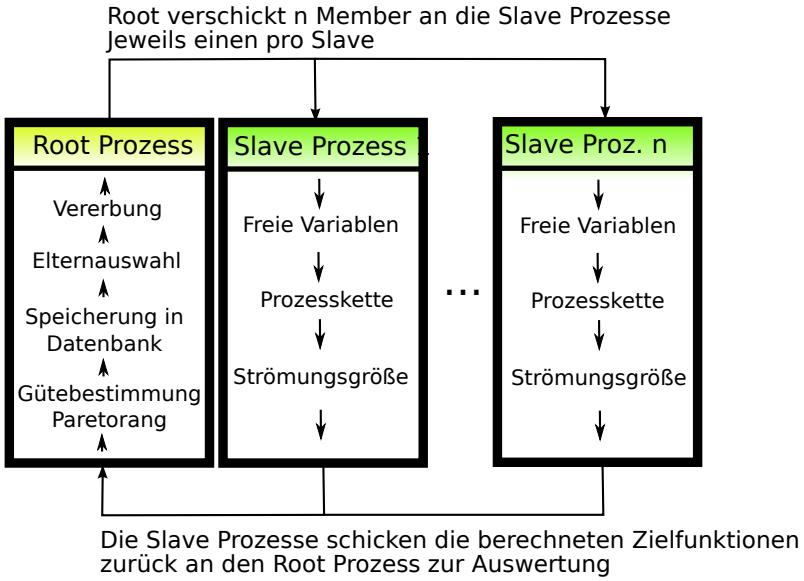


Abbildung 1.3: Prozesskette des genetischen Algorithmus in AutoOpti

Optimierung unterschiedlich und kann vom Benutzer vorher in einer Prozesskette festgelegt werden. Meist werden Strömungslöser oder Programme zur mechanischen Berechnung gestartet.

Die Optimierungen laufen i.d.R. auf einem Cluster, wobei der Root Prozess auf einem Knoten ausgeführt wird und die Slaves jeweils einen eigenen Knoten zur Berechnung der Prozesskette bekommen.

Der grundsätzliche Ablauf einer Optimierung mit AutoOpti läuft folgendermaßen ab (siehe Abbildung 1.3):

1. Vor der eigentlichen Optimierung müssen initial einige Member erzeugt werden. Hierfür werden oftmals zufällig die freien Variablen variiert und die Zielfunktionen berechnet und dann in die Datenbank eingetragen. Dies wird solange durchgeführt, bis eine gewisse Anzahl von Membern in der Datenbank steht.

2. Der Root Prozess wählt aus der vorhandenen Datenbank eine gewisse Anzahl von Membern als Eltern aus.

3. Der Root Prozess wendet Vererbungsoperatoren auf die Eltern an (Mutation, Kreuzungen usw.) und erzeugt so einen neuen Satz freier Variablen also einen neuen Member.

4. Der Root Prozess verschickt den neuen Member zur Berechnung an einen freien Slave X. Falls kein Slave frei ist, wartet der Root Prozess bis dies der Fall ist.

5. Der Slave Prozess X startet mit den empfangenen freien Variablen die Prozesskette und wartet bis diese beendet ist.

6. Ist die Prozesskette durchlaufen, wertet der Slave Prozess X die Ergebnisse der Prozesskette aus und berechnet daraus die Zielfunktion. Meist werden die Ergebnisse der Berechnung einfach direkt als Zielfunktion übernommen. Manchmal sind aber Normierungen oder andere Transformationen notwendig, daher der Zwischenschritt der Zielfunktionsberechnung.

7. Der Slave Prozess schickt die berechneten Zielfunktionen und weitere physikalische Parameter an den Root Prozess zurück und ist im Status bereit. Die weiteren physikalischen Parameter können beispielsweise für eine Nebenbedingung oder aber für den Benutzer zur Überwachung der Optimierung verwendet werden.

8. Die berechneten Zielfunktionen gehen zurück an den Root Prozess. Dieser bestimmt anhand der bestehenden Datenbasis und der Zielfunktion einen Gütewert. Danach wird der Member, gemäß seiner Güte, in die Datenbank einsortiert.

Vorteile dieser Strategie:

- Die Suche nach dem Optimum wird global durchgeführt.
- Die Zielfunktion muss nicht differenzierbar sein, was die Umsetzung dieser Strategie stark vereinfacht.
- Die Zielfunktion muss nicht immer bewertbar sein, beispielsweise bei nicht Konvergenz.
- Die Strategie ist sehr gut parallelisierbar, z.B. die oben genannte Einteilung in Root und Slave Prozesse.

Nachteile:

- Der größte Nachteil ist die Geschwindigkeit. Da die Prozesskette in der Regel sehr aufwendig zu berechnen ist, möchte man mit möglichst wenig erzeugten Membern zum Optimum der Zielfunktion gelangen. Allerdings müssen bei der Erzeugung über Vererbungsoperatoren sehr viele Member generiert werden, bis man zu einem zufriedenstellenden Ergebnis gelangt. Insbesondere aerodynamische Berechnungen sind extrem rechenintensiv. Diese nehmen trotz hoher Parallelisierung mehrere Stunden bis Tage auf mehreren Knoten in Anspruch.

Um den Prozess zu beschleunigen, versucht man die Membererzeugung des Root Prozesses (Punkt 2 und 3) zu verbessern. Insbesondere versucht man Member zu erzeugen, die eine möglichst große Verbesserung der Zielfunktion aufweisen. Da die Erzeugung in der Evolutionsstrategie durch z.B. Mutation oder Kreuzung von Membern zufallsbasiert ist, sind die einzelnen Verbesserungen der Member eher klein und viele Iterationen und damit auch reale Funktionsauswertungen (z.B. Strömungslösungen) notwendig, um das gewünschte Ziel zu erreichen. Ersatzmodelle wie z.B. das Kriging Modell approximieren oder interpolieren anhand der vorhandenen Daten die Zielfunktion. Bei jeder Iteration muss ein solches Modell trainiert werden. Das Training ist in der Regel der zeitaufwendigste Teil der Verwendung eines Ersatzmodells, im Vergleich zu einer Strömungslösung ist der Aufwand allerdings zu vernachlässigen.

Nach dem Training können Funktionswerte an beliebigen Stellen vorhergesagt werden. Die Vorhersage von Funktionswerten durch ein Ersatzmodell ist meist auch um einige Größenordnungen schneller als die echte Funktion, bspw. ein Strömungslöser. Die Ersatzmodelle werden dann herangezogen, um mit den Vorhersagen neue Member zu erzeugen. Die Erzeugung kann dann z.B. durch eine kleine Optimierung auf dem Ersatzmodell erfolgen. Der von dem Ersatzmodell beste vorhergesagte Member wird als neuer Member verwendet und anschließend mit der echten Funktion (bspw. ein Strömungslöser) nachgerechnet. Durch dieses Verfahren lassen sich die Anzahl der echten Funktionsauswertungen stark reduzieren, wie stark hängt natürlich von der Güte des verwendeten Modells ab.

Abbildung 1.4 zeigt den veränderten Root Prozess (die Slave Prozesse wurden aus Platzgründen ausgeblendet). Die Punkte 2 und 3 werden dabei durch folgenden Ablauf ersetzt:

1. Das Ersatzmodell wird mit der bereits vorhandenen Datenbank trainiert. Das Training stellt auch den aufwendigsten Teil der Nutzung des Modells dar.

2. Danach wird eine eigene Optimierung auf dem Ersatzmodell gestartet, wobei das Ersatzmodell hier den Teil der Prozesskette ersetzt und eine Approximation dieser darstellt. Da die Funktionsauswertungen auf dem Ersatzmodell erheblich schneller sind als die Prozesskette selbst, geht dieser Schritt relativ schnell.

3. Nach der Optimierung muss eine Auswahl der vom Ersatzmodell vorhergesagten Member vorgenommen werden. Hierfür gibt es diverse Ansätze, im einfachsten Fall würde man den Member mit der niedrigsten vorhergesagten Zielfunktion nehmen.

4. Der Root Prozess schickt die ausgewählten Member nun an einen Slave Prozess. Der Slave Prozess berechnet dann die echte Zielfunktion, sodass diese dann auch mit der Vorhersage des Ersatzmodells verglichen werden kann.

In AutoOpti verwendete Ersatzmodelle sind bayesisch trainierte Neuronale Netzwerke [Mac91] und Kriging.

1.3 Aufgabenstellung und Entwicklungsumgebung

Das bisher implementierte Kriging Modell soll in Zukunft stark erweitert werden. Insbesondere ist ein Co-Kriging geplant, welches den Daten verschiedene Vertrauensstufen zuordnet und so Daten verschiedener Güte verarbeiten kann. Das bisherige Kriging ist in C geschrieben. Ein objektorientierter Ansatz eignet sich allerdings für die wachsenden Anforderungen deutlich besser. Beispielsweise soll im Co-Kriging die Möglichkeit bestehen, für verschiedene Vertrauensstufen verschiedene Korrelationsfunktionen zu verwenden. Zudem wären grundsätzlich verschiedene Minimierungsverfahren denkbar, die austauschbar sein sollten. Aufgrund dieser und anderer Anforderungen macht es zukünftig Sinn, auf ein objektorientiertes Modell umzusteigen,

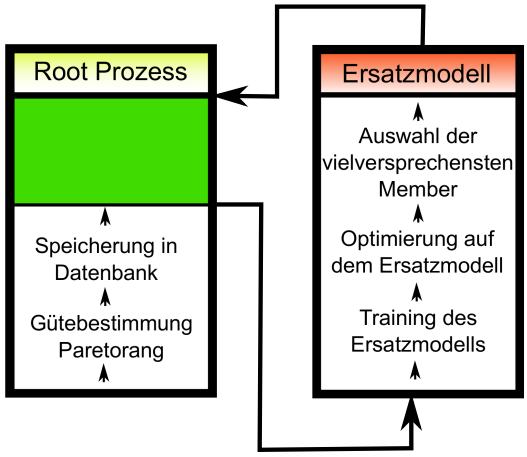


Abbildung 1.4: Nutzung eines Ersatzmodells im Optimierungsprozess

da viele der Problemstellungen sehr gut durch Generalisierungen und gemeinsame Interfaces lösbar sind. Als neue Programmiersprache wurde C++ gewählt, da die Programmiersprache C den anderen Entwicklern geläufig ist und hauptsächlich genutzt wird, sollte der Umstieg so einfacher ausfallen. Ein Refactoring kommt bei der Anzahl an notwendigen Änderungen nicht mehr in Frage, daher wird der Code im Rahmen dieser Arbeit komplett neu entwickelt.

Zusätzlich soll das Gradient Enhanced Kriging Verfahren implementiert werden. Dadurch ist es möglich, neben den Datenpunkten und Zielfunktionswerten auch Gradienten an den Datenpunkten vorzugeben. In Abbildung 1.5 ist das Prinzip qualitativ erläutert, auf der horizontalen Achse ist eine freie Variable aufgetragen und auf der vertikalen Achse die dazugehörige Zielfunktion. Die reale Funktion, welche approximiert werden soll, ist in diesem Fall eine einfache Parabel.

Als Trainingsmuster für das Kriging Modell sind zwei Punkte gewählt worden. Das Ordinary Kriging Verfahren legt in diesem Fall eine Ausgleichsgerade durch die zwei Punkte, da sonst keine Information über den weiteren Verlauf der Funktion bekannt ist.

Beim Gradient Enhanced Kriging werden zusätzlich zu den Datenpunkten noch die entsprechenden Ableitungen nach der freien Variable an den zwei Punkten vorgegeben. Dadurch kann das Verfahren die Parabel deutlich besser approximieren. Für die Optimierung ist dies deshalb interessant, da moderne aerodynamische Strömungslöser mit relativ wenig Aufwand die Gradienten approximieren können und die Vorhersagen der Ersatzmodelle erheblich besser werden.

Das Ersatzmodell soll in AutoOpti eingebunden werden und eine Testoptimierung zur Validierung durchgeführt werden.

Entwicklungsumgebung

Die verwendete Sprache war C++ und der verwendete Compiler war Gnu C++ Compiler (gcc) Version 4.5.3. Zusätzlich wurde das Programm auch mit der Version 4.3.4

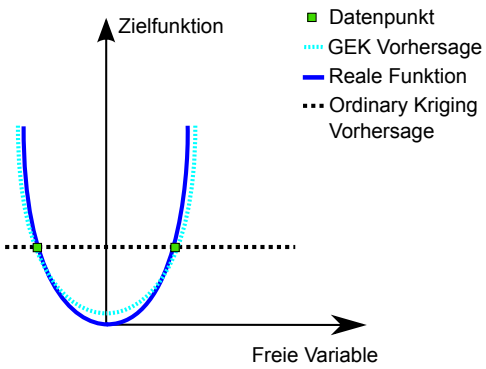


Abbildung 1.5: Qualitative Darstellung der verschiedenen Kriging Verfahren

getestet, da diese von einigen Industriepartnern verwendet wird. Der Gnu Compiler ab Version 4.2 enthält standardmäßig OpenMP, diese Bibliothek wurde zur Thread Parallelisierung verwendet. Außerdem wurden aus der Boost Bibliothek Version 1.49 einige String Funktionen und Funktionsobjekte verwendet. Die Funktionsobjekte wurden hier als Schnittstelle zwischen einer C Library mit Minimierungsalgorithmen und dem C++ Code verwendet. Dies wurde gemacht, da verschiedene Minimierungsalgortihmen aus der C Bibliothek verwendet werden, diese allerdings keine einheitliche Schnittstelle aufweisen. Mit den Boost Funktionsobjekten ließen sich die alten C Funktionen quasi "umbauen". In Kapitel 5 wird dies im Detail erklärt. Zusätzlich wurde eine Matrix Klasse verwendet, welche die hier benötigten Matrix Operationen wie z.B. Multiplikation, Addition, Invertierung von quadratisch symmetrischen Matrizen bereitstellt. Diese Klasse ist ebenfalls über OpenMP parallelisiert und verwendet den Streaming SIMD Extensions 2 Befehlssatz aktueller Prozessoren. Die Klasse wurde im Rahmen dieser Arbeit entwickelt und wird in Kapitel 3 genauer dargestellt. Als Syntax Editor wurde "Eclipse (Kepler) for Parallel Application Developers" verwendet, dieser unterstützt C/C++ und bietet einige zusätzliche Funktionen für OpenMP Programme an. Um die entsprechenden Versionen zu verwalten, wurde der institutseigene SVN Server verwendet.

2 Grundlagen Kriging

Unter Kriging versteht man statistische Verfahren zur Interpolation von Werten an unbeprobten Orten. Der Name des Verfahrens stammt von dem südafrikanischen Bergbauingenieur Daniel Krige (1951), dieser versuchte eine optimale Interpolationsmethode für den Bergbau zu entwickeln, die auf der räumlichen Abhängigkeit von Messpunkten [Kri53] basiert. Das Verfahren wurde später nach ihm benannt. Der französische Mathematiker Georges Matheron (1963) entwickelte aus Kriges Arbeit, schließlich die Kriging Theorie[Mat63]. Das Kriging Verfahren hat heute in den Geowissenschaften sowie vielen anderen Forschungsbereichen Verwendung gefunden.

Die Vorteile von Kriging gegenüber anderer Methoden, wie z.B. Inverser Distanzwichtung [She68], insbesondere bei der Nutzung innerhalb eines Optimierungsverfahrens, wie es im Institut für Antriebstechnik existiert, sind:

- Die Initialisierung ist sehr einfach, d.h. das Verfahren liefert so gut wie immer gute Ergebnisse unabhängig von irgendwelchen Startparametern. Bei Neuronalen Netzwerken z.B. muss vor dem Training die Netztopologie, also die Anzahl der Gewichte und die Struktur des Netzes angegeben werden. Werden diese Parameter ungünstig gewählt, liefert das Neuronale Netz schlechte oder gar keine Ergebnisse.

- Das Extrapolationsverhalten ist bei einer Optimierung sehr vorteilhaft. Da man bei einer Optimierung sehr schnell in Bereiche kommt, wo das Ersatzmodell extrapolieren muss, sollten die vorhergesagten Werte möglichst sinnvoll sein. Beim Ordinary Kriging liefert das Modell bei einer Extrapolation den Erwartungswert. Ein Neuronales Netzwerk bspw. würde hier zufällige Werte vorhersagen, dies könnte bei einer Optimierung zu Schwierigkeiten führen.

- Kriging ist ein BLUE Schätzer [Pla50], Best Linear Unbiased Estimator. Also ein linearer Erwartungstreuer Schätzer minimaler Varianz. Im nächsten Abschnitt wird dies genauer erläutert.

- Viele nichtstatistische Verfahren, wie z.B. die Inverse Distanzwichtung [She68], beachten eine lokale Häufung von Stützstellen nicht. Bei einer lokalen Häufung von Stützstellen sollten diese weniger stark gewichtet werden. Ein statistisches Verfahren wie Kriging beinhaltet die gesamte räumliche Verteilung der Stützstellen [Krü12].

2.1 Grundlegender Ansatz bei Kriging Verfahren

Im Folgenden sei $D \subseteq \mathbb{R}^k$ das Untersuchungsgebiet. Der zu schätzende Wert an der Stelle $\vec{x}_0 \in R^k$ sei $y^*(\vec{x}_0)$. Zudem sind $y(\vec{x}_1)..y(\vec{x}_n)$ bereits bekannte Werte oder auch Stützstellen. Da Kriging ein statistisches Interpolationsverfahren ist, welches für den räumlichen Zusammenhang von beprobten Stützstellen statistische Begriffe wie Erwartungswert und Varianz verwendet, werden die raumbezogenen Daten als Realisation von Zufallsvariablen modelliert [Krü12]. Die Stützstellen werden also als Realisierung von Zufallsvariablen $Z(\vec{x}_1)..Z(\vec{x}_n)$ betrachtet. Der zu schätzende unbeprobte Wert $y^*(\vec{x}_0)$ wird ebenfalls als Realisierung der Zufallsvariable $Z(\vec{x}_0)$ betrachtet, besitzt also einen Erwartungswert und eine Varianz.
Eine grundlegende Annahme bei Kriging ist, dass der zu schätzende Wert $y^*(x_0)$ durch eine gewichtete Summe berechnet werden kann. Diese Summe wird gebildet aus den Stützstellen $y(\vec{x}_1)..y(\vec{x}_n)$ und den zu bestimmenden Gewichten $w_1...w_n$.

$$y^*(\vec{x}_0) = \sum_{i=1}^{n} w_i y(\vec{x}_i) \tag{2.1}$$

Durch die Annahme, dass die beprobten und unbeprobten Werte als Realisierungen von Zufallsvariablen dargestellt werden, kommt man auf folgende Formulierung:

$$Z^*(\vec{x}_0) = \sum_{i=1}^{n} w_i Z(\vec{x}_i) \tag{2.2}$$

Best Linear Unbiased Estimator

In diesem Abschnitt sollen einige grundlegende Annahmen getroffen werden, über diese und weitere Annahmen können dann die eigentlichen Gewichte bestimmt werden. F sei die Fehlerfunktion

$$F(\vec{x}_0) = Z(\vec{x}_0) - Z^*(\vec{x}_0) = Z(\vec{x}_0) - \sum_{i=1}^{n} w_i Z(\vec{x}_i)$$

Kriging ist ein erwartungstreuer Schätzer, was bedeutet, dass der Schätzfehler im Mittel 0 ist. $E[X]$ stellt hier den Erwartungswert der Variable X dar.

$$E[F(\vec{x}_0)] = 0 \tag{2.3}$$

$$\Leftrightarrow E[Z(\vec{x}_0)] - E[Z^*(\vec{x}_0)] = 0$$

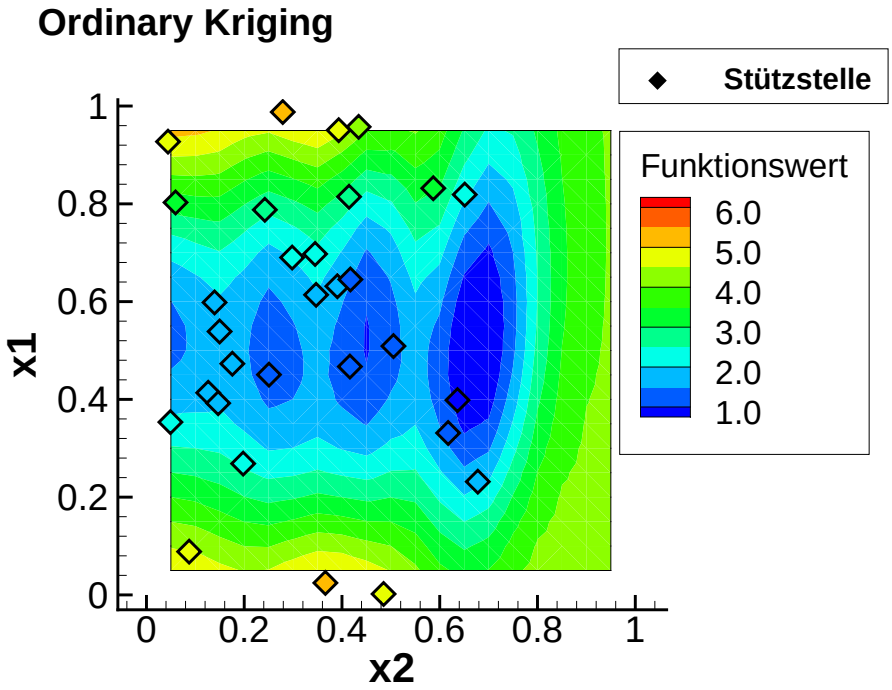


Abbildung 2.1: Beispiel für eine Kriging Interpolation, die Rauten stellen Stützstellen $y(x_i), i \in \{1...n\}$ am Ort $x_i, i \in \{1...n\}$ im Untersuchungsgebiet $D \subseteq \mathbb{R}^k$ und die Interpolation durch das Ordinary Kriging Verfahren

$$\Leftrightarrow E\left[Z^*(\vec{x}_0)\right] = E\left[Z(\vec{x}_0)\right]$$

Der Krige Schätzer soll ein bestmöglicher Schätzer sein. Daraus folgt, dass die Varianz des Schätzfehlers minimal ist.

$$\text{var}\left[F(\vec{x}_0)\right] = \min_{w_1,\ldots,w_n} \text{var}\left[Z(\vec{x}_0) - \sum_{i=1}^{n} w_i Z(\vec{x}_i)\right] \tag{2.4}$$

Aus diesen Bedingungen ergibt sich, dass der Krige Schätzer ein sogenannter BLUE (Best Linear Unbiased Estimator) Schätzer ist. Das Best ergibt sich aus 2.4, Linear aus der gewichteten Summe 2.1 und unbiased aus der Erwartungstreue 2.3.

2.2 Ordinary Kriging

Die Annahme beim Ordinary Kriging ist, dass der statistische Prozess $Z(\vec{x})$ einen konstanten aber unbekannten Erwartungswert hat, hier $\beta \in \mathbb{R}$.

$$E\left[Z(\vec{x})\right] = \beta$$

Durch diese Annahme kann die Erwartungstreue aus Gleichung 2.3 umformuliert werden zu:

$$E\left[F\left(x_0\right)\right] = 0$$

$$E\left[Z\left(\vec{x}_0\right) - \sum_{i=1}^{n} w_i Z\left(\vec{x}_i\right)\right] = 0$$

$$E\left[Z\left(\vec{x}_0\right)\right] - \sum_{i=1}^{n} w_i E\left[Z\left(\vec{x}_i\right)\right] = 0$$

$$\beta - \sum_{i=1}^{n} w_i \beta = 0$$

$$\beta\left(1 - \sum_{i=1}^{n} w_i\right) = 0$$

$$1 = \sum_{i=1}^{n} w_i$$

Bestimmung der Gewichte

Die Varianz des Schätzfehlers kann wie folgt umformuliert werden:

$$\begin{aligned}
\operatorname{var}\left[F\left(\vec{x}_0\right)\right] = \operatorname{var}\left[Z\left(\vec{x}_0\right) - Z^*\left(\vec{x}_0\right)\right] &= E\left[\left(\left(Z\left(\vec{x}_0\right) - Z^*\left(\vec{x}_0\right)\right) - E\left[Z\left(\vec{x}_0\right) - Z^*\left(\vec{x}_0\right)\right]\right)^2\right] \\
&= E\left[\left(\left(Z\left(\vec{x}_0\right) - Z^*\left(\vec{x}_0\right)\right) - \left(E\left[Z\left(\vec{x}_0\right)\right] - E\left[Z^*\left(\vec{x}_0\right)\right]\right)\right)^2\right] \\
&= E\left[\left(\left(Z\left(\vec{x}_0\right) - E\left[Z\left(\vec{x}_0\right)\right]\right) - \left(Z^*\left(\vec{x}_0\right) - E\left[Z^*\left(\vec{x}_0\right)\right]\right)\right)^2\right] \\
&= E\left[\left(Z\left(\vec{x}_0\right) - E\left[Z\left(\vec{x}_0\right)\right]\right)^2\right] \\
&\quad -E\left[2 * \left(Z\left(\vec{x}_0\right) - E\left[Z\left(\vec{x}_0\right)\right]\right)\left(Z^*\left(\vec{x}_0\right) - E\left[Z^*\left(\vec{x}_0\right)\right]\right)\right] \\
&\quad +E\left[\left(Z^*\left(\vec{x}_0\right) - E\left[Z^*\left(\vec{x}_0\right)\right]\right)^2\right] \\
&= E\left[\left(Z\left(\vec{x}_0\right) - E\left[Z\left(\vec{x}_0\right)\right]\right)^2\right] \\
&\quad -2 * E\left[\left(Z\left(\vec{x}_0\right) - E\left[Z\left(\vec{x}_0\right)\right]\right)\left(Z^*\left(\vec{x}_0\right) - E\left[Z^*\left(\vec{x}_0\right)\right]\right)\right] \\
&\quad +E\left[\left(Z^*\left(\vec{x}_0\right) - E\left[Z^*\left(\vec{x}_0\right)\right]\right)^2\right]
\end{aligned}$$

$$= \text{var}\left[Z\left(\vec{x}_0\right)\right] - 2 * \text{cov}\left(Z\left(\vec{x}_0\right), Z^*\left(\vec{x}_0\right)\right) + \text{var}\left[Z^*\left(\vec{x}_0\right)\right]$$

Setzt man nun die Formulierung 2.1 für $Z^*\left(\vec{x}_0\right)$ und $var\left[Z^*\right] = E\left[\left(Z^* - E\left[Z^*\right]\right)^2\right]$ kommt man nach einiger Rechnung zu folgender Gleichung. Die Kovarianzfunktion wird im Folgenden als gegeben und differenzierbar angesehen.

$$\text{var}\left[F\left(\vec{x}_0\right)\right] = \text{var}\left[Z\left(\vec{x}_0\right)\right] - 2\sum_{i=1}^{n}\text{cov}\left(Z\left(\vec{x}_0\right), w_i Z\left(\vec{x}_i\right)\right) + \sum_{i=1}^{n}\sum_{j=1}^{n} w_i w_j \text{cov}\left(Z\left(\vec{x}_i\right), Z\left(\vec{x}_j\right)\right)$$

In Matrix Schreibweise ergibt sich die folgende Formulierung, wobei $\vec{w} \in \mathbb{R}^{\text{n}}$ den Gewichtsvektor, $\mathbf{Cov} \in \mathbb{R}^{\text{n}\times\text{n}}$ die Kovarianzmatrix und $\overrightarrow{\text{cov}} \in \mathbb{R}^{\text{n}}$ darstellt:

$$\text{var}\left[F\left(\vec{x}_0\right)\right] = \text{var}\left[Z\left(\vec{x}_0\right)\right] - 2\overrightarrow{\text{cov}}\left(Z\left(\vec{x}_0\right), Z\left(\vec{x}\right)\right)^T * \vec{w} + \vec{w}^T * \mathbf{Cov} * \vec{w} \tag{2.5}$$

Die Minimierungsaufgabe stellt sich nun also wie folgt:

$$\begin{cases} \min_{w_1,\ldots,w_n} \text{var}\left[Z\left(\vec{x}_0\right) - \sum_{i=1}^{n} w_i Z\left(\vec{x}_i\right)\right] \\ \sum_{i=1}^{n} w_i = 1 \end{cases} \tag{2.6}$$

Mit dieser Formulierung kann nun die Minimierungsaufgabe aus Gleichung 2.6 gemäß der Multiplikatorenmethode von Lagrange gelöst werden [BS08]. Als Lagrange Funktion $\Lambda\left(w, \lambda\right)$ mit dem Lagrange Multiplikator λ ergibt sich:

$$\Lambda\left(w, \lambda\right) := \text{var}\left[F\left(\vec{x}_0\right)\right] + \lambda\left(\sum_{i=1}^{n} w_i - 1\right)$$

Die notwendige Bedingung für das Minimum ist:

$$\nabla_{\lambda,w}\left(\text{var}\left[F\left(\vec{x}_0\right)\right] + \lambda\left(\sum_{i=1}^{n} w_i - 1\right)\right) = \vec{0}$$

Setzt man Gleichung 2.5 ein, erhält man das folgende Gleichungssystem:

$$\nabla_w\left(\text{var}\left[Z\left(\vec{x}_0\right)\right] - 2\overrightarrow{\text{cov}}\left(Z\left(\vec{x}_0\right), Z\left(\vec{x}_i\right)\right)^T * \vec{w} + \vec{w}^T * \mathbf{Cov} * \vec{w} + \lambda\left(\sum_{i=1}^{n} w_i - 1\right)\right) = \vec{0}$$

$$\wedge\ \nabla_\lambda\left(\text{var}\left[Z\left(\vec{x}_0\right)\right] - 2\overrightarrow{\text{cov}}\left(Z\left(\vec{x}_0\right), Z\left(\vec{x}_i\right)\right)^T * \vec{w} + \vec{w}^T * \mathbf{Cov} * \vec{w} + \lambda\left(\sum_{i=1}^{n} w_i - 1\right)\right) = \vec{0} \tag{2.7}$$

Einschub Korrelation

Da die Kovarianzfunktion beliebig große Werte annehmen kann, ist es für die spätere Modellierung günstiger diese zu normieren. Die normierte Kovarianz ist die Korrelation und der Zusammenhang zwischen Kovarianz und Korrelation sieht wie folgt aus:

$$c\left(X,Y\right)=\frac{\text{cov}\left(X,Y\right)}{\sqrt{\text{var}\left(X\right)}\sqrt{\text{var}\left(Y\right)}} \tag{2.8}$$

Zudem soll der räumliche Zusammenhang zwischen zwei Zufallsvariablen nicht von seiner absoluten Lage im Raum abhängig sein, sondern nur von deren Abstandsvektor $h\in\mathbb{R}^{\text{n}}=\{x_1-y_1,...,x_n-y_n\}$
Die Kovarianz zwischen zwei gleichen Zufallsvariablen ist per Definition die Varianz dieser Zufallsvariable:

$$\text{cov}\left(Z,Z\right)=\text{var}\left(Z\right)$$

Da die Kovarianz allerdings nur vom Abstandsvektor abhängig ist und der Abstandsvektor bei zwei gleichen Zufallsvariablen immer $\vec{0}$ ist, kann man daraus folgern, dass die Kovarianz zwischen zwei gleichen Zufallsvariablen immer konstant ist.

$$\text{cov}\left(Z,Z\right)=\text{cov}\left(h=\vec{0}\right)=const.$$

Und somit ist auch die Varianz konstant:

$$\Longrightarrow\text{var}\left[Z\right]=\sigma^2=const. \tag{2.9}$$

Durch die Gleichungen 2.8 und 2.9 kann man die Kovarianzfunktion durch die Korrelationsfunktion ersetzen

$$c\left(X,Y\right)\sigma^2=\text{cov}\left(X,Y\right)$$

Und somit auch die Kovarianzmatrix durch die Korrelationsmatrix $\mathbf{R}\in\mathbb{R}^{\text{n}\times\text{n}}$

$$\mathbf{Cov}=\sigma^2\mathbf{R}$$

Damit lässt sich Gleichung 2.7 umformulieren, wobei der Korrelationsvektor mit $\vec{r}\in\mathbb{R}^{\text{n}}=\left(c\left[Z\left(\vec{x}_0\right),Z\left(\vec{x}_1\right)\right],...,c\left[Z\left(\vec{x}_0\right),Z\left(\vec{x}_n\right)\right]\right)^T,\vec{x}\in\mathbb{R}^{\text{k}}$ bezeichnet wird. Dieser beschreibt die Korrelationen zwischen dem Funktionswert, der vorhersagt werden soll und dem Stützstellenvektor $\vec{y}_s$. Da der Funktionswert $y\left(x_0\right)$ nicht bekannt ist, hängt die Korrelationsfunktion nur von dem Abstandsvektor $\vec{h}\in\mathbb{R}^{\text{k}}=\{x_0-x_1,...,x_k-x_k\}$ ab. Der Korrelationsvektor ergibt sich damit zu $\vec{r}\in\mathbb{R}^{\text{n}}=\left(c\left[\vec{x}_0,\vec{x}_1\right],...,c\left[\vec{x}_0,\vec{x}_n\right]\right)^T,\vec{x}\in\mathbb{R}^{\text{k}}$. Wie eine solche Korrelationsfunktion konkret umgesetzt wird, folgt in Kaptitel 3.3.

Fortsetzung vom Gleichung 2.7:
Es ergibt sich folgendes Gleichungssystem:

$$\sigma^2 \nabla_w \left(1 - 2\vec{r}^T * \vec{w} + \vec{w}^T * \mathbf{R} * \vec{w}\right) + \nabla_w \left(\lambda \left(\sum_{i=1}^{n} w_i - 1\right)\right) = \vec{0}$$

$$\wedge\ \sigma^2 \nabla_\lambda \left(1 - 2\vec{r}^T * \vec{w} + \vec{w}^T * \mathbf{R} * \vec{w}\right) + \nabla_\lambda \left(\lambda \left(\sum_{i=1}^{n} w_i - 1\right)\right) = \vec{0}$$

Aufgelöst:

$$\sigma^2 \left(\nabla_w \left(-2\vec{r}^T * \vec{w}\right) + \nabla_w \left(\vec{w}^T * \mathbf{R} * \vec{w}\right)\right) + \nabla_w \left(\lambda \left(\sum_{i=1}^{n} w_i - 1\right)\right) = \vec{0}$$

$$\wedge\ \sigma^2 \nabla_\lambda \left(\left(-2\vec{r}^T * \vec{w}\right) + \nabla_\lambda \left(\vec{w}^T * \mathbf{R} * \vec{w}\right)\right) + \nabla_\lambda \left(\lambda \left(\sum_{i=1}^{n} w_i - 1\right)\right) = \vec{0}$$

Da cov $(X, Y) =$ cov (Y, X) muss die Korrelationsmatrix $\mathbf{R}$ symmetrisch sein. Die Ableitung der quadratischen Form $\nabla_w \left(\vec{w}^T * \mathbf{R} * \vec{w}\right)$ ergibt sich somit zu $2\mathbf{R} * \vec{w}$

$$\sigma^2 \left((-2\vec{r}) + 2\mathbf{R}\vec{w}\right) + \lambda \vec{F} = \vec{0}$$

$$\wedge\ \sum_{i=1}^{n} w_i - 1 = \vec{0}$$

wobei $F_i = 1$ für $i = 1...n$ und $F_i = 0$ für $i > n$

$$\mathbf{R}\vec{w} + \frac{\lambda \vec{F}}{2\sigma^2} = \vec{r}$$

$$\wedge\ \vec{F}^T \vec{w} = 1$$

In Matrix Schreibweise:

$$\begin{pmatrix} \mathbf{R} & \vec{F} \\ \vec{F}^T & 0 \end{pmatrix} \begin{pmatrix} \vec{w} \\ \frac{\lambda}{2\sigma^2} \end{pmatrix} = \begin{pmatrix} \vec{r} \\ 1 \end{pmatrix}$$

$$\Leftrightarrow \begin{pmatrix} \vec{w} \\ \frac{\lambda}{2\sigma^2} \end{pmatrix} = \begin{pmatrix} \mathbf{R} & \vec{F} \\ \vec{F}^T & 0 \end{pmatrix}^{-1} \begin{pmatrix} \vec{r} \\ 1 \end{pmatrix} \tag{2.10}$$

Die Inverse der Blockmatrix ergibt sich nach [Tho06] zu:

$$\begin{pmatrix} \mathbf{R} & \vec{F} \\ \vec{F}^T & 0 \end{pmatrix}^{-1} = \begin{pmatrix} \mathbf{R}^{-1} - \frac{\mathbf{R}^{-1}\vec{F}\vec{F}^T\mathbf{R}^{-1}}{\vec{F}^T\mathbf{R}^{-1}\vec{F}} & \frac{\mathbf{R}^{-1}\vec{F}}{\vec{F}^T\mathbf{R}^{-1}\vec{F}} \\ \frac{\vec{F}^T\mathbf{R}^{-1}}{\vec{F}^T\mathbf{R}^{-1}F} & -\frac{1}{\vec{F}^T\mathbf{R}^{-1}\vec{F}} \end{pmatrix}$$

Eingesetzt in 2.10:

$$\begin{pmatrix} \vec{w} \\ \frac{\lambda}{2\sigma^2} \end{pmatrix} = \begin{pmatrix} \mathbf{R}^{-1} - \frac{\mathbf{R}^{-1}\vec{F}\vec{F}^T\mathbf{R}^{-1}}{\vec{F}^T\mathbf{R}^{-1}\vec{F}} & \frac{\mathbf{R}^{-1}\vec{F}}{\vec{F}^T\mathbf{R}^{-1}\vec{F}} \\ \frac{\vec{F}^T\mathbf{R}^{-1}}{\vec{F}^T\mathbf{R}^{-1}F} & -\frac{1}{\vec{F}^T\mathbf{R}^{-1}\vec{F}} \end{pmatrix} \begin{pmatrix} \vec{r} \\ 1 \end{pmatrix}$$

Daraus ergibt sich die folgende Formulierung der gesuchten Gewichte:

$$\vec{w} = \mathbf{R}^{-1}\vec{r} - \frac{\mathbf{R}^{-1}\vec{F}\vec{F}^T\mathbf{R}^{-1}}{\vec{F}^T\mathbf{R}^{-1}\vec{F}}\vec{r} + \frac{\mathbf{R}^{-1}\vec{F}}{\vec{F}^T\mathbf{R}^{-1}\vec{F}} \tag{2.11}$$

Der zweite Term für $\frac{\lambda}{2\sigma^2}$ wird hierbei ignoriert, da nur die Formulierung der Gewichte gesucht war. Wird der Gewichtsvektor auf diese Weise berechnet, dann ist die Varianz var $[F(x_0)]$ unter der geforderten Nebenbedingung minimal.

Krige Schätzer

Setzt man diese Formulierung in Gleichung 2.2 ein, erhält man die Formel des Krige Schätzers, wobei der Vektor $\vec{y}_s \in \mathbb{R}^n$ die bekannten Funktionswerte enthält.

$$Z^*(\vec{x}_0) = \sum_{i=1}^{n} w_i Z(\vec{x}_i)$$

$$E[Z^*(\vec{x}_0)] = \sum_{i=1}^{n} w_i E[Z(\vec{x}_i)]$$

$$E\left[Z^*\left(\vec{x}_0\right)\right] = \vec{w}^T \vec{y}_s$$

$$E\left[Z^*\left(\vec{x}_0\right)\right] = \left(\mathbf{R}^{-1}\vec{r} - \frac{\mathbf{R}^{-1}\vec{F}\vec{F}^T\mathbf{R}^{-1}}{\vec{F}^T\mathbf{R}^{-1}\vec{F}}\vec{r} + \frac{\mathbf{R}^{-1}\vec{F}}{\vec{F}^T\mathbf{R}^{-1}\vec{F}}\right)^T \vec{y}_s$$

$$E\left[Z^*\left(\vec{x}_0\right)\right] = \left(\mathbf{R}^{-1}\vec{r}\right)^T \vec{y}_s - \left(\frac{\mathbf{R}^{-1}\vec{F}\vec{F}^T\mathbf{R}^{-1}}{\vec{F}^T\mathbf{R}^{-1}\vec{F}}\vec{r}\right)^T \vec{y}_s + \left(\frac{\mathbf{R}^{-1}\vec{F}}{\vec{F}^T\mathbf{R}^{-1}\vec{F}}\right)^T \vec{y}_s$$

$$E\left[Z^*\left(\vec{x}_0\right)\right] = \vec{r}^T\mathbf{R}^{-1}\vec{y}_s - \vec{r}^T\frac{\mathbf{R}^{-1}\vec{F}\vec{F}^T\mathbf{R}^{-1}}{\vec{F}^T\mathbf{R}^{-1}\vec{F}}\vec{y}_s + \frac{\vec{F}^T\mathbf{R}^{-1}\vec{y}_s}{\vec{F}^T\mathbf{R}^{-1}\vec{F}}$$

$$\beta = \frac{\vec{F}^T\mathbf{R}^{-1}\vec{y}_s}{\vec{F}^T\mathbf{R}^{-1}\vec{F}}$$

$$E\left[Z^*\left(\vec{x}_0\right)\right] = \vec{r}^T\mathbf{R}^{-1}\vec{y}_s - \vec{r}^T\mathbf{R}^{-1}\beta\vec{F} + \beta$$

$$E\left[Z^*\left(\vec{x}_0\right)\right] = \vec{r}^T\mathbf{R}^{-1}\left(\vec{y}_s - \beta\vec{F}\right) + \beta$$

$\vec{F}$ entspricht hier dem Einsvektor.

Vorhersage der Varianz

Setzt man die Formulierung der Gewichte 2.11 in Gleichung 2.5 ein, dann erhält man eine Gleichung für die Vorhersage für die Varianz des Kriging Modells. Die Zwischenschritte wurden hier aus Platzgründen weggelassen.

$$\text{var}\left[F\left(\vec{x}_0\right)\right] = \sigma^2\left(1 - r^T * \mathbf{R}^{-1} * r + \frac{1}{F^T\mathbf{R}^{-1}F}\left[r^T\mathbf{R}^{-1}F - 1\right]^2\right) \tag{2.12}$$

Quellen: [Kri53, Aul12, Mat63]

2.3 Gradient Enhanced Kriging

Das Gradient Enhanced Kriging ist eine Erweiterung des Ordinary Kriging. Beim GEK gehen partielle Ableitungen in der Form $\frac{\partial y(x)}{\partial x}$ an einigen beprobten Orten mit in die Bildung des Modells ein. Der Trainingsvektor $\vec{y}_s \in \mathbb{R}^{\mathrm{n+m}}$ kann in diesem Fall also die folgende Form annehmen:

$\vec{y}_s = \left(y\left(\vec{x}_1\right), ..., y\left(\vec{x}_n\right), \frac{\partial y(\vec{x}_{n+1})}{\partial x^j}, ..., \frac{\partial y(\vec{x}_{n+m})}{\partial x^j}\right), j \in \{1, ..., k\}$, wobei der obere Index von x die Variable darstellt, nach der abgeleitet wird und m ist die Anzahl der gegebenen partiellen Ableitungen. Beim Aufstellen der Korrelationsmatrix $\mathbf{R}$ müssen also auch Kovarianzen/Korrelationen zwischen den Gradienten und den Funktionswerten gebildet werden können. Die Kovarianzfunktion zwischen zwei Zufallsvariablen ist wie folgt definiert:

$$\mathrm{cov}(Z\left(\vec{x}_1\right), Z\left(\vec{x}_2\right)) = E\left[\left(Z\left(\vec{x}_1\right) - E\left[Z\left(\vec{x}_1\right)\right]\right)\left(Z\left(\vec{x}_2\right) - E\left[Z\left(\vec{x}_2\right)\right]\right)\right] \tag{2.13}$$

Um im Kriging Modell partielle Ableitungen verarbeiten zu können ist es notwendig, dass die Kovarianzfunktionen in der folgenden Form gebildet werden:

$$\mathrm{cov}(\frac{\partial Z\left(\vec{x}_1\right)}{\partial x_1^p}, \frac{\partial Z\left(\vec{x}_2\right)}{\partial x_2^l}), p, l \in \{1, ..., k\} \tag{2.14}$$

$$\mathrm{cov}(\frac{\partial Z\left(\vec{x}_1\right)}{\partial x_1^p}, Z\left(\vec{x}_2\right)), p \in \{1, ..., k\} \tag{2.15}$$

$$\mathrm{cov}(Z\left(\vec{x}_1\right), \frac{\partial Z\left(\vec{x}_2\right)}{\partial x_2^l}), l \in \{1, ..., k\} \tag{2.16}$$

Per Definition aus Gleichung 2.13 ergibt sich für Gleichung 2.14:

$$\mathrm{cov}(\frac{\partial Z\left(\vec{x}_1\right)}{\partial x_1^p}, Z\left(\vec{x}_2\right)) = E\left[\left(\frac{\partial Z\left(\vec{x}_1\right)}{\partial x_1^p} - E\left[\frac{\partial Z\left(\vec{x}_1\right)}{\partial x_1^p}\right]\right)\left(Z\left(\vec{x}_2\right) - E\left[Z\left(\vec{x}_2\right)\right]\right)\right] \tag{2.17}$$

Die Korrelationsfunktion ist im Kriging vorgegeben (Kapitel3.3). Im Folgenden soll gezeigt werden, dass die Kovarianzfunktion $\mathrm{cov}(\frac{\partial Z(\vec{x}_1)}{\partial x_1^p}, Z\left(\vec{x}_2\right))$ auch durch einfaches Differenzieren von 2.13 bilden lässt.

$$\frac{\partial}{\partial x_1^p}\mathrm{cov}(Z\left(\vec{x}_1\right), Z\left(\vec{x}_2\right)) = \frac{\partial}{\partial x_1^p}\left(E\left[\left(Z\left(\vec{x}_1\right) - E\left[Z\left(\vec{x}_1\right)\right]\right)\left(Z\left(\vec{x}_2\right) - E\left[Z\left(\vec{x}_2\right)\right]\right)\right]\right)$$

$$\begin{aligned} = & E\left[\left(Z\left(\vec{x}_2\right) - E\left[Z\left(\vec{x}_2\right)\right]\right)\frac{\partial}{\partial x_1^p}\left(Z\left(\vec{x}_1\right) - E\left[Z\left(\vec{x}_1\right)\right]\right)\right] \\ & + E\left[\left(Z\left(\vec{x}_1\right) - E\left[Z\left(\vec{x}_1\right)\right]\right)\frac{\partial}{\partial x_1^p}\left(Z\left(\vec{x}_2\right) - E\left[Z\left(\vec{x}_2\right)\right]\right)\right] \end{aligned}$$

$$
\begin{aligned}
=&E\left[\left(Z\left(\vec{x}_2\right)-E\left[Z\left(\vec{x}_2\right)\right]\right)\frac{\partial}{\partial x_1^p}\left(Z\left(\vec{x}_1\right)-E\left[Z\left(\vec{x}_1\right)\right]\right)\right]\\
&+E\left[\left(Z\left(\vec{x}_1\right)-E\left[Z\left(\vec{x}_1\right)\right]\right)\frac{\partial}{\partial x_1^p}\left(Z\left(\vec{x}_2\right)-E\left[Z\left(\vec{x}_2\right)\right]\right)\right]
\end{aligned}
$$

Es gilt $\frac{\partial}{\partial x_1^p}\left(Z\left(x_2\right)-E\left[Z\left(x_2\right)\right]\right)=0$, da der Ausdruck unabhängig von x_1 ist.

$$
=E\left[\left(Z\left(\vec{x}_2\right)-E\left[Z\left(\vec{x}_2\right)\right]\right)\left(\frac{\partial}{\partial x_1^p}Z\left(\vec{x}_1\right)-E\left[\frac{\partial}{\partial x_1^p}Z\left(\vec{x}_1\right)\right]\right)\right]
$$

Dieser Ausdruck entspricht Gleichung 2.17, also gilt:

$$
\operatorname{cov}(\frac{\partial Z\left(\vec{x}_1\right)}{\partial x_1^p},Z\left(\vec{x}_2\right))=\frac{\partial}{\partial x_1^p}\operatorname{cov}(Z\left(\vec{x}_1\right),Z\left(\vec{x}_2\right)) \tag{2.18}
$$

für die Gleichungen 2.14 und 2.16 gilt entsprechendes:

$$
\operatorname{cov}(\frac{\partial Z\left(\vec{x}_1\right)}{\partial x_1^p},\frac{\partial Z\left(\vec{x}_2\right)}{\partial x_2^l})=\frac{\partial}{\partial x_1^p\partial x_2^l}\operatorname{cov}(Z\left(\vec{x}_1\right),Z\left(\vec{x}_2\right))
$$

$$
\operatorname{cov}(Z\left(\vec{x}_1\right),\frac{\partial Z\left(\vec{x}_2\right)}{\partial x_2^l})=\frac{\partial}{\partial x_2^l}\operatorname{cov}(Z\left(\vec{x}_1\right),Z\left(\vec{x}_2\right))
$$

Da die Korrelationsfunktion vorgegeben ist, kann diese entsprechend abgeleitet werden (sofern diese differenzierbar ist) und man erhält den nötigen Zusammenhang zwischen den Funktionswerten und Gradienten. Beispiele für Korrelationsfunktionen und deren Ableitungen werden in Kapitel 3.3 behandelt.

Auf die Herleitung des Krige Schätzers und der Vorhersage der Varianz soll hier im Weiteren nicht eingegangen werden, da diese im Wesentlichen der Herleitung des Ordinary Kriging entsprechen. Der Leser sei aber auf [Krü12, HGZ09] verwiesen. Die Bildung des Korrelationsvektors $\vec{r}$ und des Erwartungswertvektors $\beta\vec{F}$ sieht wie folgt aus:

$$
\vec{F}\in\mathbb{R}^{\mathrm{n}}=\left(\underbrace{1,...,1}_{n},\underbrace{0,...,0}_{m}\right)^T
$$

$$
\begin{aligned}
\vec{r}\in\mathbb{R}^{\mathrm{n+m}}=&\left(c\left[\vec{x}_0,\vec{x}_1\right],...,c\left[\vec{x}_0,\vec{x}_n\right],\frac{\partial c\left[\vec{x}_0,\vec{x}_{n+1}\right]}{\partial x^j},...,\frac{\partial c\left[\vec{x}_0,\vec{x}_{n+m}\right]}{\partial x^j}\right)^T\\
&,\vec{x}\in\mathbb{R}^{\mathrm{k}},j\in\{1,...,k\}
\end{aligned}
$$

Wobei erwähnt sei, dass die Ableitungen nach x auch nur teilweise vorhanden sein können. Es muss also nicht für jeden Member jede Ableitung nach jeder freien Variable gegeben sein.

Die Bildung der Matrix $\mathbf{R}$ wird in Kapitel 3 erläutert, da die softwaretechnische Umsetzung stark von der Bildung der Matrix abhängt und dies im Zusammenhang erläutert werden soll.

2.4 Andere Kriging Verfahren

Neben dem bereits erwähnten Ordinary Kriging gibt es eine Reihe anderer Kriging Verfahren. Diese unterscheiden sich meist durch die verschiedenen Annahmen, die im Modell getroffen werden und durch die Art der Daten, welche für das Training verwendet werden können. So war z.B. eine Annahme beim Ordinary Kriging, dass der Erwartungswert für alle Zufallsvariablen konstant aber unbekannt ist.

$$E\left[Z\left(\vec{x}\right)\right] = \beta$$

Ein weiteres Verfahren ist das Simple Kriging. Bei diesem ist die Bedingung das der Erwartungswert für alle Zufallsvariablen 0 ist.

$$E\left[Z\left(\vec{x}\right)\right] = 0$$

Die Daten werden beim Simple Kriging meist auf einen Mittelwert von 0 skaliert:

$$\frac{1}{n}\sum_{i=1}^{n} y_i = 0$$

Der reale Erwartungswert ist jedoch meist unbekannt und muss deswegen geschätzt werden.

Das Simple Kriging ist daher auch die einfachste Form der Kriging Annahmen, aber auch die am wenigsten allgemeine.

Universal Kriging ist eine Erweiterung des Ordinary Kriging. Der Erwartungswert $\beta\left(\vec{x}\right)$ ist nun nicht mehr konstant, sondern kann durch eine beliebige Funktion beschrieben werden.

$$E\left[Z\left(\vec{x}\right)\right] = \beta\left(\vec{x}\right)$$

Einen anderen Ansatz verfolgen Variable Fidelity Methods (VFM). Diese Verfahren sollen Daten verschiedener Güte nutzen können. Das kann beispielsweise in der Strömungssimulation durch unterschiedlich feine Rechennetze von Vorteil sein. Sehr grobe Rechennetze liefern ungenauere Ergebnisse, dafür in deutlich kürzerer Zeit. Die Funktionen von groben und feinen Netzen sind sich vom Verlauf her sehr ähnlich, aber nicht identisch. Zudem vertraut man dem höherwertigen Modell mehr. Um dies zu erreichen, gibt es verschiedene Ansätze. Beispielsweise über eine additive Brückenfunktion. Hierfür würde ein Krigingmodell für die Daten höherer Güte trainiert $y_{hf}\left(\vec{x}\right)$ und

ein Modell auf die Differenz zwischen den Daten niedrigerer Güte $y_{lf}(\vec{x})$ und denen höherer Güte $y_{diff}(\vec{x}) = y_{lf}(\vec{x}) - y_{hf}(\vec{x})$. Die sich daraus ergebende Brückenfunktion $y_b(\vec{x})$ würde dann eine Addition aus dem Modell höherer Güte und dem Differenzmodell sein $y_b(\vec{x}) = y_{lf}(\vec{x}) + y_{diff}(\vec{x})$. Neben dieser Brückenfunktion existiert auch noch ein Co-Kriging Verfahren, welches diesen Ansatz in einem Modell vereint [FSK07].

Zusätzlich zu den bereits erwähnten Kriging Verfahren, gibt es noch diverse andere Verfahren, die in dieser Arbeit aber nicht erwähnt werden. Der Leser sei auf [D+09] verwiesen.

3 Bildung der Korrelationsmatrix

In diesem Kapitel soll die Softwaretechnische Umsetzung des Algorithmus zur Bildung der Korrelationsmatrix dargestellt werden. Da in den Algorithmen sehr viele Matrix Operationen verwendet werden, wurde eine Matrix Klasse eingeführt. Diese wird am Anfang des Kapitels erläutert.

Im nächsten Abschnitt wird dann die Bildungsvorschrift der Matrix gezeigt und darauf folgend eine häufig genutzte Korrelationsfunktion und deren softwaretechnische Umsetzung erläutert.

Der letzter Abschnitt zeigt dann den eigentlichen Algorithmus zur Bildung der Korrelationsmatrix.

3.1 Klasse zur Berechnung von Matrix Operationen

Da für das Training und Vorhersagen des Kriging Ersatzmodells hauptsächlich Matrix Operationen verwendet werden, ist es sinnvoll diese in einer Klasse zusammenzufassen. Außerdem sind für die Matrix Operationen verschiedene Implementierungen möglich, z.B. könnte es eine Klasse für OpenMP parallelisierte Operationen geben und eine andere Klasse wo dieselben Operationen über eine GPU (Graphics Processing Unit) berechnet werden. Daher werden die gemeinsamen Elemente in einer Superklasse Matrix zusammengefasst, siehe Abbildung 3.1. In dem UML Diagramm sind noch zwei andere Klassen erkennbar, eine Superklasse SaveableOnServer und eine Spezialisierung namens OpenMPMatrix.

SaveableOnServer sollte in einer modernen objektorientierten Programmiersprache wie z.B. Java ein kontextspezifisches anbietendes Client/Server Interface [SFK12] sein. Dies ist zur Verdeutlichung nochmals als UML Diagramm in Abbildung 3.2 dargestellt. In C++ muss ersatzweise eine abstrakte Klasse verwendet werden. Das Interface SaveableOnServer muss implementiert werden, um in späteren Anwendungen eine prozessweite Parallelisierung über eine Parallelisierungsbibliothek vornehmen zu können. Diese Parallelisierungsbibliothek wurde ebenfalls im Institut für Antriebstechnik entwickelt, allerdings außerhalb des Rahmens dieser Masterarbeit und soll daher nicht näher erläutert werden.

Die Matrix Klasse besitzt genau drei Attribute, welche als protected deklariert sind. Das erste Attribut “elements” stellt ein eindimensionales Array dar und beinhaltet die Matrixelemente. Da eine Matrix einem zweidimensionalen Array entspricht, wird das eindimensionale Array auf ein zweidimensionales Array übersetzt. Dies wird gemacht, da so ein linearer Aufbau des Arrays garantiert gewährleistet wird und dies

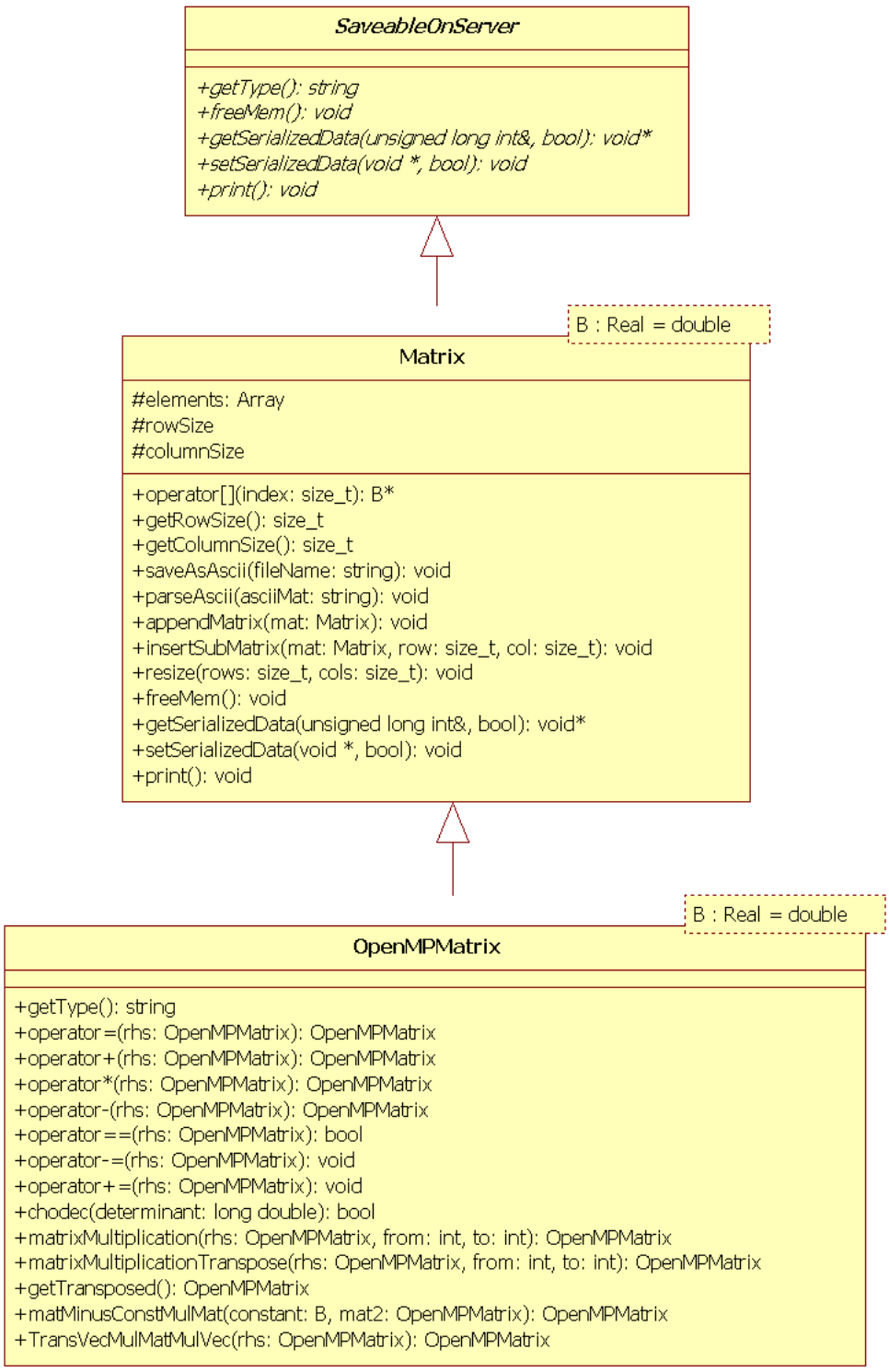


Abbildung 3.1: UML Diagramm der Matrixklasse mit einer Spezialisierung (OpenMP-Matrix) und der abstrakten Klasse SaveableOnServer, welche prinzipiell ein Interface darstellt. Interfaces weden in C++ jedoch nicht unterstützt, daher wird eine abstrakte Klasse verwendet.

kann in einigen Systemen Geschwindigkeitsvorteile bringen. Der Aufbau des Array "elements" sieht wie folgt aus:

Elementnr.	Funktion
1	Zeilenanzahl
2	Spaltenanzahl
3	Dummy
4	Dummy
5 bis Spaltenanzahl	Erste Zeile der Matrix
Spaltenanzahl bis x*Spaltenanzahl	x'te Zeile der Matrix

Die Anzahl der Zeilen und Spalten sind also ebenfalls im Array gespeichert. Dies wird gemacht, um die Daten als einen Block zu serialisieren und damit als einen Datenstrom über das Netzwerk verschicken zu können.

Zusätzlich sind die Anzahl der Zeilen und Spalten als Attribut in der Klasse Matrix gespeichert, da auf diese Attribute sehr oft lesend zugegriffen wird und der Zugriff auf eine einzelne Variable schneller ist, als der Zugriff auf die beiden ersten Elemente des Arrays. Da die Attribute nur über Getter und Setter Methoden zugreifbar sind, kann immer gewährleistet werden, dass diese identisch sind. Die Übersetzung auf ein zweidimensionales Array wird dadurch erreicht, dass man den "[]" Operator der Matrix Klasse in folgender Weise überlädt:

```
T* operator[] (int row) {
    return this->elements + 4 + (row * this->getColumnSize());
}
```

Bei einem Zugriff auf den "[]" Operator wird also ein Zeiger auf das erste Element der entsprechenden Zeile zurückgegeben. Dies wird in C++ als ein Array interpretiert und das Array kann über den Operator "[]" die entsprechende Spalte liefern. Die Verwendung ist also dieselbe wie bei einem "normalen" zweidimensionalen C Array oder C++ Vektor.

Zusätzlich gibt es noch die Methoden saveAsAscii und parseAscii, welche es ermöglichen die Daten in eine Textdatei zu speichern und auszulesen. Zur Änderung der Größe der Matrix gibt es die Methode resize, welche die Matrix entsprechend vergrößert oder verkleinert. Die Methode appendMatrix hängt eine Matrix an die bestehende an und zwar an die letzte Zeile. Dies geht also nur, wenn die Spaltenanzahl identisch ist. Um eine Submatrix als Block in eine bestehende Matrix zu integrieren, gibt es die Funktion insertSubMatrix. Diese fügt die als Parameter angegebene Matrix in die Bestehende ein.

In Abbildung 3.1 ist zusätzlich noch eine Spezialisierung namens OpenMPMatrix eingezeichnet, diese nutzt OpenMP zur Thread Parallelisierung und wird auch für das Kriging Modell verwendet. Im Wesentlichen wird in der Klasse OpenMPMatrix die Superklasse Matrix durch verschiedene Operatoren zur Addition, Multiplikation usw. erweitert und stellt so alle nötigen Operationen für das Kriging Modell bereit.

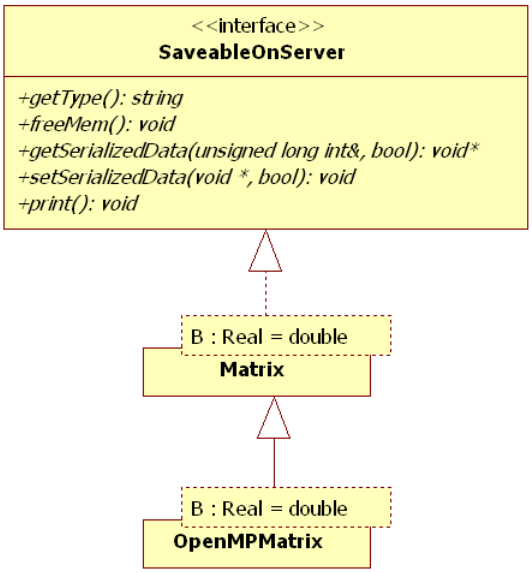


Abbildung 3.2: UML Diagramm der Matrix Klasse, wobei SaveableOnServer in diesem Fall durch ein kontextspezifisches Client Server Interface dargestellt wird.

Cholesky Zerlegung

Die Methode chodec() stellt eine Cholesky Zerlegung für positiv definite und symmetrische Matrizen bereit, für die Matrix $\mathbf{R}$ muss also gelten:

$$\mathbf{R} = \mathbf{R}^T$$

und

$$\overrightarrow{v} * \mathbf{R} * \overrightarrow{v} > 0 \ für alle Vektoren \ \overrightarrow{v}$$

Bei der Cholesky Zerlegung wird die Matrix $\mathbf{R}$ in ein Produkt aus einer unteren Dreicksmatrix und deren Transponierten zerlegt:

$$\mathbf{L}\mathbf{L}^T = \mathbf{R}$$

Die Zerlegung kann nun verwendet werden, um durch Vor- und Rückwärtseinsetzen lineare Gleichungssysteme in der Form $\mathbf{R}\overrightarrow{x} = b$ zu lösen. Hierfür wird zuerst vorwärts eingesetzt:

$$\mathbf{L}\overrightarrow{y} = \overrightarrow{b}$$

und dann durch Rückwärtseinsetzen kann der gesuchte Vektor $\overrightarrow{x}$ erhalten werden:

$$\mathbf{L}^T\overrightarrow{x} = \overrightarrow{y}$$

$$\Longrightarrow \mathbf{R}\overrightarrow{x} = \mathbf{L}\mathbf{L}^T\overrightarrow{x} = \mathbf{L}\overrightarrow{y} = \overrightarrow{b}$$

ersetzt man nun $\overrightarrow{x}$ durch $\mathbf{R}^{-1}$ und $\overrightarrow{b}$ durch die Einheitsmatrix $\mathbf{E}$, kann man mit der Zerlegung die Inverse der Matrix $\mathbf{R}$ berechnen.

Nach diesem Schritt können die Vor- und Rückwärtssubstitutionen spaltenweise durchgeführt werden. Diese lassen sich sehr gut parallelisieren, da jede CPU einfach einen Vektor der Inversen berechnet.

Ein weiterer Vorteil der Zerlegung ist, dass die Determinante der Matrix $\mathbf{R}$ mit der Zerlegung einfach durch Multiplikation der Diagonalelemente der zerlegten Matrix gewonnen werden kann:

$$det\left(\mathbf{R}\right) = \prod_{i=1}^{n} L_{i,i}^{2}$$

Der verwendete Algorithmus ist in [Pre07, Gil07] nochmals detaillierter beschrieben.

Quadratische Form

Da bei dem Training eines Kriging Modells häufig quadratische Formen $\vec{v^T} R \vec{v}$ berechnet werden müssen, lohnt es sich diese zu beschleunigen. Dies wird dadurch gemacht, dass die Multiplikationen nicht nacheinander durchgeführt werden, sondern beide Multiplikationen in einer doppel-Schleife behandelt werden.

$$\sum_{i=1}^{n} \left(v_i \sum_{j=1}^{n} R_{i,j} * v_j \right) = \vec{v^T} R \vec{v}$$

Der Algorithmus ist im folgenden Listing gezeigt:

```
#pragma omp parallel for reduction(+:ret)
for(size_t row=0; row < matrix->getRowSize(); row++){
        double sum=0.;
        for(size_t col=0; col < matrix->getColumnSize(); col++){
                sum += matrix[row][col] * vec[col];
        }
        sum *= vec[row];
        result +=sum;
}
```

In der ersten Zeile wird eine Schleifenparallelisierung über OpenMP initialisiert, wobei das Aufsummieren von result nicht parallelisiert werden darf. Dies würde sonst zu unvorhersehbaren Fehlern führen, da mehrere Threads gleichzeitig in die Variable result schreiben möchten.

3.2 Bildungsvorschrift der Korrelationsmatrix

Die Korrelationsmatrix $\mathbf{R}$ beschreibt den paarweisen linearen Zusammenhang aller Stützstellen untereinander. Im Folgenden ist der allgemeine Aufbau einer Korrelationsmatrix zwischen zwei Membern mit den entsprechenden Korrelationen zwischen den Zufallsvariablen $Z\left(\overrightarrow{x_1}\right)$ und $Z\left(\overrightarrow{x_2}\right)$ an den Orten $\overrightarrow{x_1}, \overrightarrow{x_2} \in \mathbb{R}^{\mathrm{k}}$ dargestellt. Zusätzlich zu den Funktionswerten sind hier noch zwei partielle Ableitungen nach der ersten freien Variablen z.B. $\frac{\partial Z(\vec{x}_2)}{\partial x_2^1}$ gegeben, wobei der obere Index von x die Variable nach der abgeleitet wird angibt und der untere Index die Membernummer. Die Korrelationsmatrix wurde hier aus Platzgründen nur mit zwei Membern gebildet und kann natürlich Korrelationen zwischen beliebig vielen Membern und partiellen Ableitungen enthalten. Die erste Spalte und erste Zeile geben jeweils an, zwischen welchen Variablen die Korrelation gebildet wird, z.B. Spalte zwei $Z\left(\overrightarrow{x_2}\right)$ und Zeile eins $Z\left(\overrightarrow{x_1}\right)$ würden eine Korrelation $c\left(Z\left(\overrightarrow{x_1}\right), Z\left(\vec{x}_2\right)\right)$ ergeben.

$$
R = \left[\begin{array}{c|cccc}
 & Z\left(\vec{x}_1\right) & Z\left(\vec{x}_2\right) & \frac{\partial Z(\vec{x}_1)}{\partial x_1^1} & \frac{\partial Z(\vec{x}_2)}{\partial x_1^1} \\
\hline
Z\left(\vec{x}_1\right) & c\left(Z\left(\vec{x}_1\right), Z\left(\vec{x}_1\right)\right) & c\left(Z\left(\vec{x}_1\right), Z\left(\vec{x}_2\right)\right) & c\left(x_1, \frac{\partial Z(\vec{x}_1)}{\partial x_1^1}\right) & c\left(x_1, \frac{\partial Z(\vec{x}_2)}{\partial x_2^1}\right) \\
Z\left(\vec{x}_2\right) & c\left(Z\left(\vec{x}_2\right), Z\left(\vec{x}_1\right)\right) & c\left(Z\left(\vec{x}_2\right), Z\left(\vec{x}_2\right)\right) & c\left(x_2, \frac{\partial Z(\vec{x}_1)}{\partial x_1^1}\right) & c\left(x_2, \frac{\partial Z(\vec{x}_2)}{\partial x_2^1}\right) \\
\frac{\partial Z(\vec{x}_1)}{\partial x_1^1} & c\left(\frac{\partial Z(\vec{x}_1)}{\partial x_1^1}, Z\left(\vec{x}_1\right)\right) & c\left(\frac{\partial Z(\vec{x}_1)}{\partial x_1^1}, Z\left(\vec{x}_2\right)\right) & c\left(\frac{\partial Z(\vec{x}_1)}{\partial x_1^1}, \frac{\partial Z(\vec{x}_1)}{\partial x_1^1}\right) & c\left(\frac{\partial Z(\vec{x}_1)}{\partial x_1^1}, \frac{\partial Z(\vec{x}_2)}{\partial x_2^1}\right) \\
\frac{\partial Z(\vec{x}_2)}{\partial x_1^1} & c\left(\frac{\partial Z(\vec{x}_2)}{\partial x_2^1}, Z\left(\vec{x}_1\right)\right) & c\left(\frac{\partial Z(\vec{x}_2)}{\partial x_2^1}, Z\left(\vec{x}_2\right)\right) & c\left(\frac{\partial Z(\vec{x}_2)}{\partial x_2^1}, \frac{\partial Z(\vec{x}_1)}{\partial x_1^1}\right) & c\left(\frac{\partial Z(\vec{x}_2)}{\partial x_2^1}, \frac{\partial Z(\vec{x}_2)}{\partial x_2^1}\right)
\end{array}\right]
$$

Da für einen voherzusagenden Wert der Funktionswert unbekannt ist, die Korrelation aber zwischen diesem und den bekannten Membern berechnet werden muss, beschreibt man die Korrelation nur noch in Abhängigkeit vom Abstand der Stützstellen untereinander. Der Korrelationswert ist also unabhängig von den Funktionswerten. Dies stellt allerdings nur eine Schätzung der Korrelation dar, für die verschiedenste Ansätze existieren, einige davon werden in Kapitel 3.3 beschrieben.

$$
c\left(Z\left(\vec{x}_1\right), Z\left(\vec{x}_2\right)\right) = c\left(\vec{x}_1, \vec{x}_2\right)
$$

Die Korrelation zwischen Funktionswerten und partiellen Ableitungen, bzw. zwischen partiellen Ableitungen und partiellen Ableitungen können wie in Gleichung 2.18 umgeformt werden zu

$$
c(\frac{\partial Z\left(\vec{x}_1\right)}{\partial x_1^p}, Z\left(\vec{x}_2\right)) = \frac{\partial}{\partial x_1^p} c(Z\left(\vec{x}_1\right), Z\left(\vec{x}_2\right))
$$

die Korrelation soll ebenfalls nur abhängig vom Abstand im Parameterraum sein

$$
= \frac{\partial}{\partial x_1^p} c(\vec{x}_1, \vec{x}_2)
$$

Die anderen Korrelationen für die partiellen Ableitungen sind in Kapitel 2.3 zu finden.

Mit diesen Annahmen ergibt sich die allgemeine Form der Korrelationsmatrix zwischen zwei Membern zu

$$R = \left[\begin{array}{c|cc|cc} & Z(\vec{x}_1) & Z(\vec{x}_2) & \frac{\partial Z(\vec{x}_1)}{\partial x_1^1} & \frac{\partial Z(\vec{x}_2)}{\partial x_1^1} \\ \hline Z(\vec{x}_1) & c(\vec{x}_1,\vec{x}_1) & c(\vec{x}_1,\vec{x}_2) & \frac{\partial c(\vec{x}_1,\vec{x}_1)}{\partial x_1^1} & \frac{\partial c(\vec{x}_1,\vec{x}_2)}{\partial x_2^1} \\ Z(\vec{x}_2) & c(\vec{x}_2,\vec{x}_1) & c(\vec{x}_2,\vec{x}_2) & \frac{\partial c(\vec{x}_2,\vec{x}_1)}{\partial x_1^1} & \frac{\partial c(\vec{x}_2,\vec{x}_2)}{\partial x_2^1} \\ \frac{\partial Z(\vec{x}_1)}{\partial x_1^1} & \frac{\partial c(\vec{x}_1,\vec{x}_1)}{\partial x_1^1} & \frac{\partial c(\vec{x}_1,\vec{x}_2)}{\partial x_1^1} & \frac{\partial c(\vec{x}_1,\vec{x}_1)}{\partial x_1^1 \partial x_1^1} & \frac{\partial c(\vec{x}_1,\vec{x}_2)}{\partial x_1^1 \partial x_2^1} \\ \frac{\partial Z(\vec{x}_2)}{\partial x_1^1} & \frac{\partial c(\vec{x}_2,\vec{x}_1)}{\partial x_2^1} & \frac{\partial c(\vec{x}_2,\vec{x}_2)}{\partial x_2^1} & \frac{\partial c(\vec{x}_2,\vec{x}_1)}{\partial x_2^1 \partial x_1^1} & \frac{\partial c(\vec{x}_2,\vec{x}_2)}{\partial x_2^1 \partial x_2^1} \end{array}\right]$$

Konkrete Korrelationsfunktionen, welche in dieser Software umgesetzt wurden, werden in Kapitel 3.3 beschrieben. Die Korrelationsfunktionen sollen innerhalb der Matrix austauschbar sein, was im entsprechenden Algorithmus zur Aufstellung der Matrix beachtet werden muss.

3.3 Korrelationsfunktionen

Um die Korrelationsmatrix aufzustellen, ist es nötig, einen Korrelationswert zwischen allen bekannten Stützstellen zu berechnen und auch zwischen den Stützstellen und deren partiellen Ableitungen. Um diesen Wert zu berechnen, wird eine Korrelationsfunktion verwendet. Wie bereits in Kapitel 3.2 beschrieben, kann die Korrelationsfunktion nur angenähert werden und ist abhängig vom Abstand zweier Stützstellen zueinander.

In diesem Abschnitt soll eine sehr häufig verwendete Korrelationsfunktion beschrieben und deren softwaretechnische Umsetzung gezeigt werden. Da diese Korrelationsfunktion einer gaußschen Normalverteilung sehr ähnlich ist, wird diese im Folgenden als gaußsche Korrelationsfunktion bezeichnet. Die Formel sieht wie folgt aus:

$$c(\vec{x}_1,\vec{x}_2) = e^{-\frac{1}{2}\sum_{l=1}^{l<k}\left(e^{\theta_l}|x_{1l}-x_{2l}|^2\right)} \tag{3.1}$$

Die Vektoren $\vec{x}_1, \vec{x}_2 \in \mathbb{R}^k$ stellen hier die Ortsvektoren der Stützstellen dar. Die Funktion liefert immer Werte zwischen Null und Eins, wobei eine Eins bei zwei identischen Stützpunken und eine Null bei unendlich weit entfernten Stützstellen herauskommen würde. Der Vektor $\vec{\theta} \in \mathbb{R}^k$ (auch Hyperparameter genannt) legt fest, wie stark die Korrelation mit steigendem Abstand der Stützstellen zueinander abfällt, wobei ein hoher Wert einen stärkeren Abfall bewirkt. In Abbildung 3.3 ist eine beispielhafte Korrelationsfunktion mit zwei verschiedenen θ Werten zu sehen, wobei die Stützstellen in der Abbildung jeweils nur eine freie Variable haben. Auf der X-Achse des Diagramms ist die Differenz der beiden Stützstellen zueinander aufgetragen und auf der Y-Achse der entsprechende Korrelationswert. Die rote durchgezogene Kurve zeigt die Korrelationsfunktion mit einem $\theta = 0.01$ und die grüne gestrichelte Kurve hat ein $\theta = 0.1$. Nimmt man eine Differenz der beiden Stützstellen von z.B. $\Delta x = -5$ an, dann ergeben sich für einen Hyperparameter von $\theta = 0.01$ ein Korrelationswert von 0.53 und für $\theta = 0.1$ ein Korrelationswert von 0.08. In dem einen Fall wird also eine stärkere Abhängigkeit

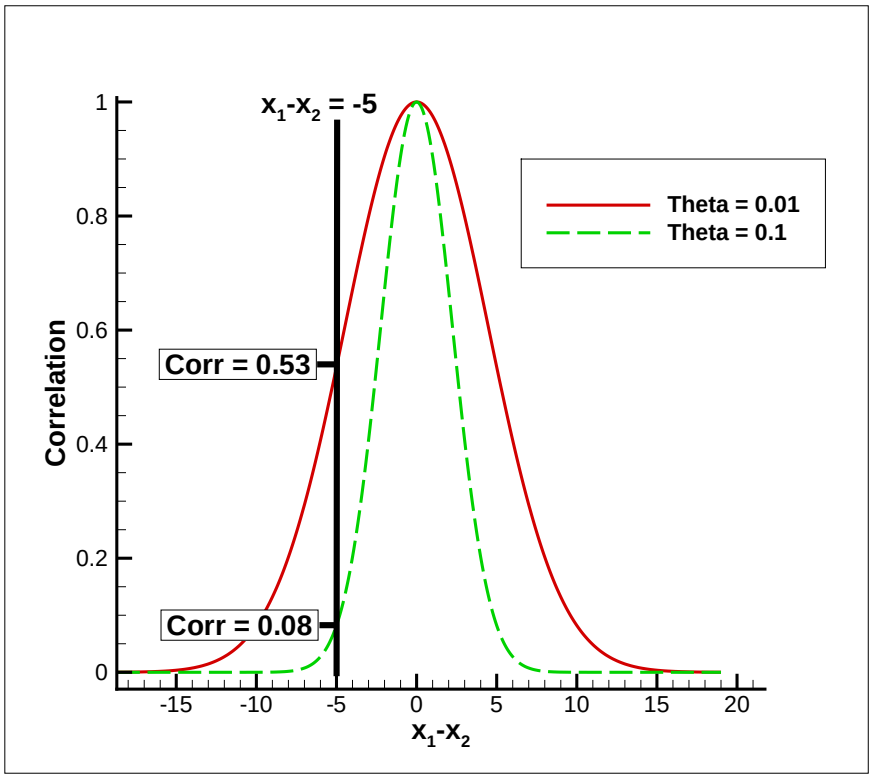


Abbildung 3.3: Beispiel einer Gauss Korrelationsfunktion mit einer freien Variable und zwei unterschiedlichen θ Einstellungen und deren Einfluss auf den Korrelationswert

der beiden Stützstellen angenommen und im anderen Fall eine sehr schwache Abhängigkeit. Für das gesamte Kriging Modell gibt es genau einen $\vec{\theta}$ Vektor, die Bestimmung dieser Werte ist Aufgabe des Trainings.

In Abbildung 3.4 wird eine Korrelationsfunktion zwischen zwei Stützstellen $\vec{x}_1, \vec{x}_2 \in \mathbb{R}^k$ mit zwei freien Variablen $k = 2$ gezeigt. Für diesen Fall hat der Vektor $\vec{\theta} \in \mathbb{R}^k$ ebenfalls zwei Komponenten. Auf der X-Achse ist die Differenz zwischen den beiden ersten Komponenten der beiden Stützstellen aufgetragen, auf der Y-Achse die Differenz zwischen den zweiten Komponenten der Stützstellenvektoren und die Z-Achse zeigt den entsprechenden Korrelationswert.

Die programmiertechnische Umsetzung ist sehr simpel und sieht wie folgt aus:

```
for(size_t i=0; i<point1.getNumVars() ; i++){
    correl += fmath::exp(thetas[i]) * sqr(point1.getVarsRef(i) - point2.getVarsRef(i));
}
correl=fmath::exp(-0.5*correl);
```

Da diese Funktion während eines Trainings sehr häufig aufgerufen wird, macht es Sinn, diese zu beschleunigen. Um dies zu erreichen, wurden SSE (Streaming SIMD Extensions) CPU Befehle verwendet. Die Lesbarkeit des Codes leidet zwar recht stark darunter, da sich diese Methode zukünftig aber kaum noch ändern wird, ist dies vertretbar.

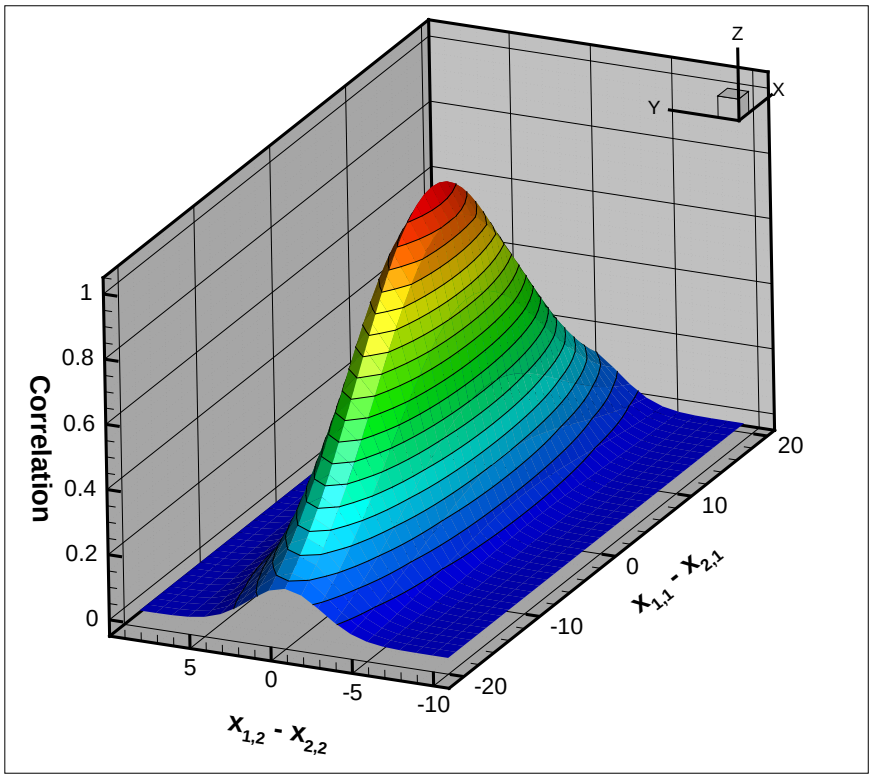


Abbildung 3.4: Beispiel einer Gauss Korrelationsfunktion mit zwei freien Variablen

Streaming SIMD Extensions (SSE)

Die Streaming SIMD Extensions (SSE) sind eine von Intel entwickelte Befehlssatzerweiterung der x86-Architektur. Mit Einführung des Pentium-III-(Katmai)-Prozessors wurde diese 1999 vorgestellt. Aufgabe der SSE Befehle ist es Programme durch Parallelisierung auf Instruktionslevel zu beschleunigen, auch SIMD (Single Instruction Multiple Data) genannt. Die SSE-Befehlssatzerweiterung umfasst ursprünglich 70 Instruktionen und 8 neue Register, genannt XMM0 bis XMM7. Ursprünglich wurden die 128 Bit breiten Register allerdings nicht in einem Schritt verarbeitet. Bei heutigen CPUs (z.B. Intel Core CPUs) werden die Register in einem Schritt verarbeitet, zudem wurde die Anzahl der Register von 8 auf 16 erhöht.

Es gibt zahlreiche Umsetzungen der SSE Befehle. Diese reichen von SSE bis SSE5, wobei ab SSE3 AMD und Intel jeweils eigene Implementationen der SSE Architektur vornahmen. Der Nachfolger von SSE heißt AVX (Advanced Vector Extensions) und verbreitert die Register auf 16x 256 Bit.

Innerhalb dieser Arbeit wurden nur SSE 1 Befehle verwendet, da diese praktisch von allen aktuellen CPUs und auch Compilern unterstützt werden. Durch die neuen 128 Bit Register können nun in einem Rechenschritt vier float (32 Bit) oder zwei double (64 Bit) Werte gleichzeitig verarbeitet werden. Um diese Funktionen zu nutzen, müssen im C++-Code spezielle SSE Befehle verwendet werden [Fog13, K.A11]. Das folgende Listing zeigt die Umsetzung der Gauss Korrelationsfunktion mit SSE Befehlen:

```
1 __m128d varDiff;
2 __m128d var1;
3 __m128d var2;
4 __m128d thetas;
```

```
5  __m128d correlSSE = _mm_setzero_pd();
6  double t[2], tmp[2];
7  size_t size = (point1.getNumVars()%2==0) ? point1.getNumVars()/2 : (point1.getNumVars()-1)/2;
8  for(size_t i=0; i< size; i++){
9    var1 = _mm_loadu_pd(&(point1.getVarsRef(i*2)));
10   var2 = _mm_loadu_pd(&(point2.getVarsRef(i*2)));
11   tmp[0] = fmath::expd(this->thetas[0][i*2]);
12   tmp[1] = fmath::expd(this->thetas[0][i*2+1]);
13   thetas = _mm_loadu_pd(&(tmp[0]));
14   varDiff = _mm_sub_pd(var1,var2);
15   correlSSE = _mm_add_pd(correlSSE, _mm_mul_pd( thetas , _mm_mul_pd(varDiff,varDiff) ) );
16 }
17 _mm_stream_pd( t, correlSSE);
18 _mm_empty();
19 correl = (t[0]+t[1]);
20 if(point1.getNumVars()%2!=0){
21  correl+=fmath::expd(this->thetas[0][point1.getNumVars()-1])*
22          sqr(point1.getVarsRef(point1.getNumVars()-1)-point2.getVarsRef(point1.getNumVars()-1));
23 }
24 correl=fmath::exp(-0.5*correl);
```

In den Zeilen 1-5 werden verschiedene Variablen definiert vom Typ "__m128d", dieser Typ stellt zwei 64 Bit double Werte dar. In Zeile 5 wird die Variable correlSSE mit der _mm_setzero_pd() Funktion auf Null gesetzt. Da in einem Schritt immer zwei Befehle gleichzeitig ausgeführt werden, muss die Anzahl durch zwei teilbar sein. Ist dies nicht der Fall, muss der Rest mit normalen Befehlen durchgeführt werden. Zeile 7 setzt die Anzahl der Schleifendurchläufe auf die Anzahl der freien Variablen geteilt durch zwei. Ist die Anzahl der freien Variablen ungerade, wird eins subtrahiert und dann durch zwei geteilt. Die Zeilen 8-16 stellen die Summation aus der Gauss Formel dar. Zuerst wird in den Zeilen 9-10 zwei Werte der Stützstellen in die SSE Variablen geladen. Dies geschieht mit der Funktion _mm_loadu_pd(). Als Parameter erwartet die Funktion eine Speicheradresse und transferiert dann 128 Bit ab dieser Adresse in die entsprechende SSE Variable. In den Zeilen 11-12 wird die Exponentialfunktion der Hyperparameter berechnet. Dies geschieht auf konventionelle Weise, da es keinen SSE Befehl für die Exponentialfunktion gibt. Die beiden berechneten Werte werden dann in Zeile 13 ebenfalls in eine SSE Variable geladen. Die Differenz der Stützstellenkomponenten wird anschließend in Zeile 14 vorgenommen. Hier werden wie bereits erwähnt, direkt zwei Differenzen gleichzeitig berechnet. In Zeile 15 wird dann die Differenz quadriert, mit den Hyperparametern multipliziert und aufsummiert, alles mit SSE Befehlen.

Ist der Schleifendurchlauf beendet, wird in Zeile 17 das Ergebnis in eine normale Variable zurück transferiert und in den restlichen Zeilen wird (falls die Anzahl der freien Variablen nicht durch zwei teilbar war) das letzte Element mit konventionellen Methoden berechnet. Die gemessenen Geschwindigkeitsvorteile lagen bei ca. 20 % - 30 %.

3.4 Algorithmus zur Bildung der Korrelationsmatrix

Beim Aufstellen der Korrelationsmatrix müssen alle Korrelationen zwischen Membern berechnet werden. Ein Member wird durch die Klasse Point beschrieben, in Abbildung 3.5 ist das UML Diagramm der entsprechenden Klasse dargestellt.

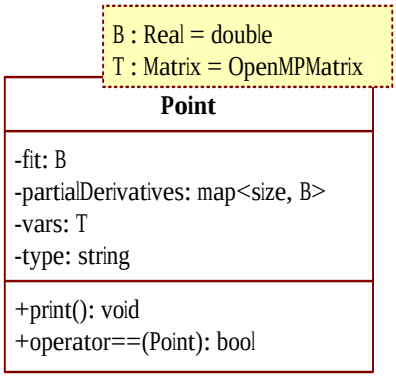


Abbildung 3.5: UML Diagramm der Klasse Point, welche einen Member/Stützstelle beschreibt

Die Klasse verwendet zwei Template Parameter. Zum einen den Parameter B, dieser gibt den verwendeten Fließkommazahlentyp an und zum anderen gibt es den Parameter T, welcher den verwendeten Matrix Typ darstellt. Der Matrix Typ muss ein Subtyp der Matrix Klasse aus Kapitel 3 sein. Die Point Klasse besteht aus vier Attributen und zwei öffentlichen Methoden. Das Attribut fit beschreibt den Funktionswert des Members. PartialDerivatives beinhaltet die vorgegebenen partiellen Ableitungen eines Members. Der Typ des Attributs ist eine map (Hashtabelle), wobei der Schlüssel die Nummer der Variable darstellt, nach der abgeleitet wurde. In vars werden die Variablenwerte des Members gespeichert, der Typ des Attributs ist eine Matrix. Eigentlich wird nur ein Vektor als Typ benötigt, da man aber gerne die Matrix Operationen nutzen möchte, wird vars als einspaltige Matrix verwendet, was letztlich wieder einem Vektor entspricht. Das Attribut type beinhaltet den Typ des Members, dies ist für Variable Fidelity Methods (siehe Kapitel 2.4) wichtig. In diesem Attribut wird dann über einen Identifier angegeben, ob es sich z.B. um einen Member handelt, welcher mit hoher Güte berechnet wurde oder niedriger.

Eine Korrelationsfunktion soll zwischen zwei solcher Point Objekte einen Korrelationswert bestimmen. Daher macht es Sinn, eine gemeinsame abstrakte Superklasse einzuführen, der Aufbau ist in Abbildung 3.6 dargestellt. Da jedes Point Objekt auch die partiellen Ableitungen enthält ist es sinnvoll, dass die Korrelationsfunktion alle Korrelationen zwischen zwei Point Objekten zurück gibt, also auch die Korrelationen zwischen partiellen Ableitungen und Funktionswerten. Zu diesem Zweck wird die Korrelationsmatrix durch Zeilen- und Spaltentausch umsortiert, um die Korrelationen zwischen zwei Point Objekten direkt als Submatrix in die Gesamtmatrix einzufügen:

$$\mathbf{R} = \left[\begin{array}{c|cccc} & Z(\vec{x}_1) & \frac{\partial Z(\vec{x}_1)}{\partial x_1^1} & Z(\vec{x}_2) & \frac{\partial Z(\vec{x}_2)}{\partial x_1^1} \\ \hline Z(\vec{x}_1) & c(\vec{x}_1,\vec{x}_1) & \frac{\partial c(\vec{x}_1,\vec{x}_1)}{\partial x_1^1} & c(\vec{x}_1,\vec{x}_2) & \frac{\partial c(\vec{x}_1,\vec{x}_2)}{\partial x_2^1} \\ \frac{\partial Z(\vec{x}_1)}{\partial x_1^1} & \frac{\partial c(\vec{x}_1,\vec{x}_1)}{\partial x_1^1} & \frac{\partial c(\vec{x}_1,\vec{x}_1)}{\partial x_1^1 \partial x_1^1} & \frac{\partial c(\vec{x}_1,\vec{x}_2)}{\partial x_1^1} & \frac{\partial c(\vec{x}_1,\vec{x}_2)}{\partial x_1^1 \partial x_2^1} \\ Z(\vec{x}_2) & c(\vec{x}_2,\vec{x}_1) & \frac{\partial c(\vec{x}_2,\vec{x}_1)}{\partial x_1^1} & c(\vec{x}_2,\vec{x}_2) & \frac{\partial c(\vec{x}_2,\vec{x}_2)}{\partial x_2^1} \\ \frac{\partial Z(\vec{x}_2)}{\partial x_1^1} & \frac{\partial c(\vec{x}_2,\vec{x}_1)}{\partial x_2^1} & \frac{\partial c(\vec{x}_2,\vec{x}_1)}{\partial x_2^1 \partial x_1^1} & \frac{\partial c(\vec{x}_2,\vec{x}_2)}{\partial x_2^1} & \frac{\partial c(\vec{x}_2,\vec{x}_2)}{\partial x_2^1 \partial x_2^1} \end{array}\right] \tag{3.2}$$

Durch diese Anordnung stehen alle Korrelationen zwischen zwei bestimmten Punkten immer zusammen, dies vereinfacht den Algorithmus zum Aufstellen der Matrix erheblich.

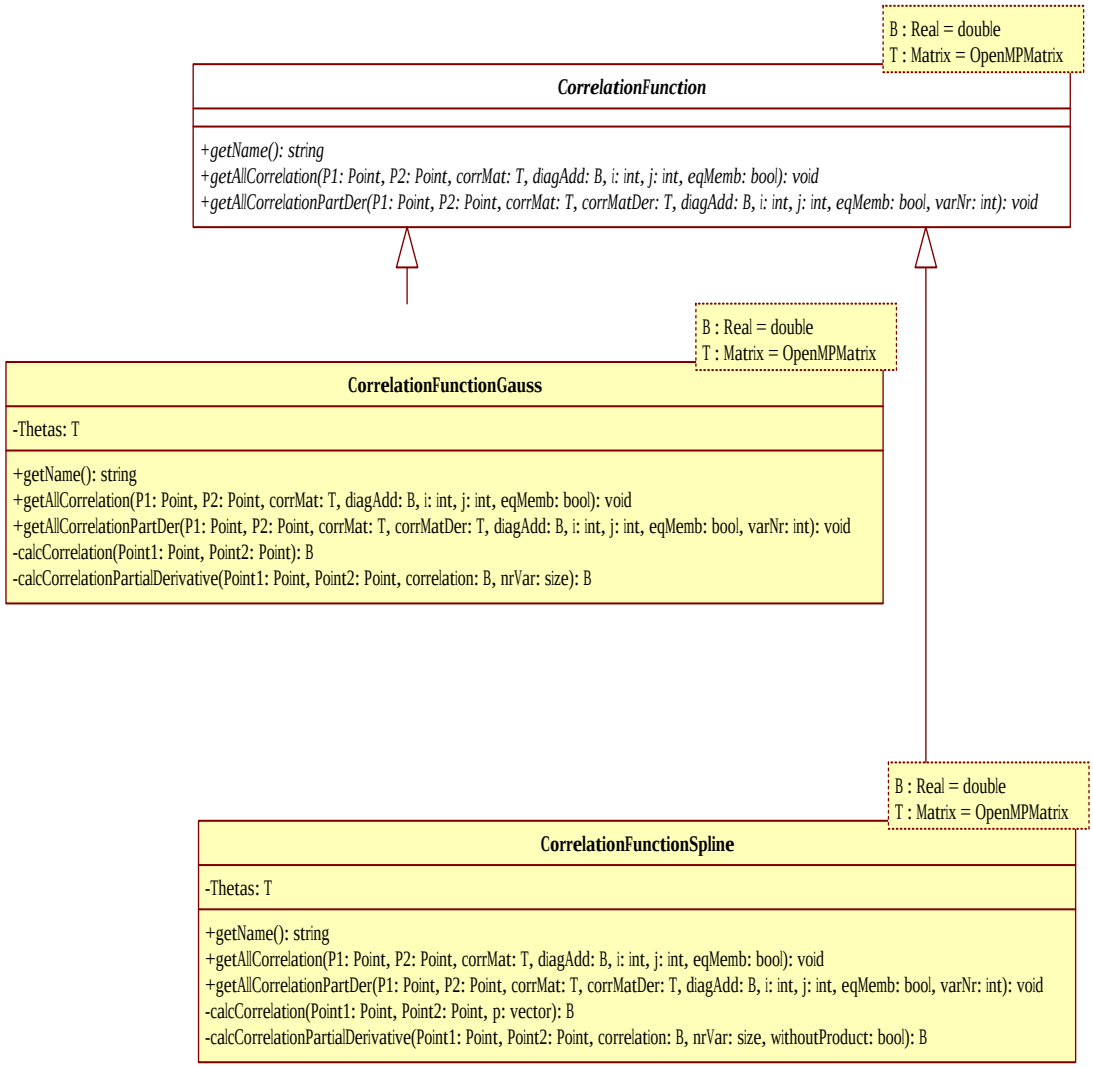


Abbildung 3.6: UML Diagramm der abstrakten Klasse CorrelationFunction mit zwei Subklassen, diese stellen spezifische Korrelationsfunktionen dar

Die Methode CreateCorrelationMatrix

Das UML Diagramm in Abbildung 3.6 zeigt die abstrakte Superklasse CorrelationFunction mit zwei Spezialisierungen namens CorrelationFunctionGauss und Correla-

tionsFunctionSpline. Die Spezialisierungen stellen jeweils verschiedene Korrelationsfunktionen dar. Die Umsetzung der Gauss Korrelationsfunktion wurde im Abschnitt 3.3 dargestellt, auf die Beschreibung der Spline Funktion wird hier verzichtet. Die Methode getAllCorrelation schreibt alle Korrelationswerte zwischen den Stützpunkten P1 und P2 in die ebenfalls übergebene Korrelationsmatrix corrMat. Da die Korrelationswerte zwischen zwei Stützstellen eine Submatrix der Korrelationsmatrix darstellen (siehe Formel 3.2), wird der Methode getAllCorrelation die entsprechende Stelle der Submatrix in der Korrelationsmatrix über die Indizes i und j mit angegebenen. Die beiden Parameter stellen hier die Position der ersten Zeile und Spalte der Submatrix in der Korrelationsmatrix dar. Es wird also im ersten Schritt die Korrelation zwischen den beiden Punkten bestimmt, dann die Korrelationen der partiellen Ableitungen, daraus wird dann die entsprechende Submatrix gebildet und diese in die Korrelationsmatrix eingefügt.

Das folgende Programmlisting zeigt die Funktion, welche unter Benutzung der abstrakten Klasse CorrelationFunktion, eine Korrelationsmatrix mit Werten befüllt. Der gezeigte Code ist der Originalcode in C++, auf Pseudocode wird hier verzichtet da einige Besonderheiten von C++ eine recht große Rolle in der Programmierung spielen. Einige Zeilenumbrüche wären in C++ in der Form nicht zulässig, aus Platzgründen ließen sich diese allerdings nicht vermeiden.

```
1  template <class T, class B>
2  void createCorrelationMatrix(
3                      T &correlationMatrix,
4                      vector<Point<T,B> > &points,
5                      map<string,map<string,CorrelationFunction<T,B>*>> correlationMap){
6
7      T tmp(config::numSamplesDerivatives, config::numSamplesDerivatives);
8      correlationMatrix = tmp;
9      #pragma omp parallel for schedule(dynamic)
10     for(size_t i=0; i<points.size(); i++){
11         for(size_t j=i; j<points.size(); j++){
12             if(i==j){
13                 correlationMap[points[i].getType()][points[j].getType()]->getAllCorrelation(
14                         points[i],
15                         points[j],
16                         correlationMatrix,
17                         0.0,
18                         config::matrixPositions[i],
19                         config::matrixPositions[j],
20                         true );
21                 correlationMatrix[config::matrixPositions[i]][config::matrixPositions[j]]
22                                 += fmath::expd(config::diagonalAddition);
23             }
24             else{
25                 correlationMap[points[i].getType()][points[j].getType()]->getAllCorrelation(
26                     points[i],
27                     points[j],
28                     correlationMatrix,
29                     0.0,
30                     config::matrixPositions[i],
31                     config::matrixPositions[j],
32                     false);
33             }
34         }
35     }
36
37     #pragma omp parallel for
38     for(int i=correlationMatrix.getColumnSize()-1; i>0; i--){
39         for(int j=i-1; j>=0; j--){
40             correlationMatrix[i][j] = correlationMatrix[j][i];
41         }
42     }
```

```
43 }
```

Die Funktion createCorrelationMatrix hat genau drei Parameter. Der erste Parameter ist eine Referenz auf die eigentliche Korrelationsmatrix. Um unnötiges Kopieren zu vermeiden, wird eine Referenz verwendet, weil die Daten direkt in die Matrix geschrieben werden sollen. Da die Matrizen sehr groß werden können (20000x20000 sind keine Seltenheit) und während des Trainings sehr häufig gebildet werden müssen, können solche Überlegungen erhebliche Unterschiede in der Geschwindigkeit ausmachen.

Der Parameter Points vom Typ Point (Abbildung 3.5) ist ein eindimensionaler Vektor, welcher alle Member/Stützstellen beinhaltet.

CorrelationMap ist eine zweidimensionale Hashtabelle, die beiden Indizes sind strings und beschreiben die Typen (vgl. Attribut type aus Abbildung 3.5) der entsprechenden Stützstellen. Durch diese Hashtabelle ist es möglich, jedem Paar von Stützstellen verschiedene Korrelationsfunktionen zuzuordnen. Dies ist bei der weiteren Entwicklung des Verfahrens von großer Bedeutung, insbesondere für Variable Fidelity Models (siehe Kapitel 2.4). Der Wert der Hashtabelle ist vom Typ CorrelationFunction (siehe 3.6), dieser kann also durch jeden Subtyp der abstrakten Klasse CorrelationFunction überladen werden. Eine solche correlationMap könnte z.B. folgendermaßen aufgebaut sein:

Typ1	Typ2	CorrelationFunction
“low”	“low”	correlationFunctionGauss
“low”	“high”	correlationFunctionSpline
“high”	“high”	correlationFunctionGauss

Würde man mit dieser correlationMap z.B. folgenden Aufruf machen, so würde man die Methode getAllCorrelation des SubTyps CorrelationFunctionGauss (siehe 3.6) aufrufen.

```
correlationMap["low"]["low"]->getAllCorrelation(...);
```

Genau diesen Aufruf findet man in den Zeilen 13 und 25.

In Zeile 7 und 8 wird eine neue Matrix vom Template Typ T allokiert und der Parameter correlationMatrix damit überschrieben. Der Wert config::numSamplesDerivatives beschreibt hier die Anzahl der Stützstellen plus die Anzahl der gegebenen partiellen Ableitungen.

In den Zeilen 9 bis 11 wird eine doppelte for Schleife über alle Stützstellen gestartet, um die Korrelationen von allen Stützstellen zu allen Stützstellen zu berechnen. Das “#pragma omp parallel for” ist eine einfache Schleifenparallelisierung von OpenMP. Die Indizes werden automatisch in Bereiche eingeteilt und dann in einzelnen Threads abgearbeitet. Die Anzahl der maximalen Threads wird direkt zu Anfang über ein Parameterfile festgelegt. Der zusätzliche Befehl “schedule(dynamic)” gibt an, wie der OpenMP Scheduler die Arbeit auf die Threads verteilt, es gibt prinzipiell drei Varianten:

- *static*: Jede Teilschleife besitzt eine feste Anzahl von Durchläufen, diese Durchläufe werden dann reihum an die Threads verteilt. Dieses Vorgehen ist, bei gleicher Lastverteilung der Teilschleifen optimal. Im Normalfall gibt es soviele Teilschleifen wie Anzahl Threads.
- *dynamic*: Hier werden die Teilschleifen dynamisch an die Threads verteilt, um

das zu erreichen, werden die Teilschleifen kleiner gewählt als z.B. bei static. Dies ist sinnvoll, wenn die Last stark variiert, allerdings ist der Verwaltungsaufwand für die OpenMP Laufzeitumgebung höher.

- *guided*: Bei diesem Fall werden die Teilschleifen während der Laufzeit exponentiell von groß zu klein verändert. Dies ist ein Spezialfall von *dynamic* und reduziert den Verwaltungsaufwand.

Welche der drei Varianten für einen Fall geeignet ist, lässt sich oftmals nur durch Testen gut bestimmen. Bei der Belegung der Korrelationsmatrix scheint zuerst die Option *static* sinnvoller zu sein, da der Aufwand für einen Schleifendurchlauf in etwa gleich bleibt. Allerdings war die Option *dynamic* in Tests ca. 10 % schneller. Das könnte durch interne Compiler Optimierungen und die Verwendung des If- Else Blocks innerhalb der Schleife zu erklären sein.

Der If- Else Block ist dazu da, um zwischen der Berechnung der Korrelation zwischen zwei gleichen Stützpunkten und der Berechnung der Korrelation zwischen zwei verschiedenen Punkten zu unterscheiden. Dies ist sinnvoll, da sich die Korrelation zwischen zwei gleichen Stützpunkten durch Vereinfachungen der mathematischen Formulierung deutlich schneller berechnen lässt. Eine solche Vereinfachung wird im nächsten Abschnitt genauer beschrieben. Zusätzlich kann auf die Diagonalelemente der Matrix, ein Diagonalaufschlag addiert werden (Zeile 21-22). Dies wird verwendet um die Matrix für die Invertierung numerisch stabiler zu machen, in Kapitel 5.2 wird der Diagonalaufschlag genauer erläutert.

Innerhalb des If- Else Blocks wird die Methode getAllCorrelation der entsprechenden Subklasse von CorrelationFunction (siehe Abbildung 3.6) aufgerufen. Zwei wichtige Parameter dieses Methodenaufrufs sind "*config::matrixPositions[i]*" und "*config::matrixPositions[j]*", diese Arrays enthalten eine Tabelle, welche zu einer gegebenen Membernummer die entsprechende Position der Member in der Korrelationsmatrix zurückgeben soll. Da ein Member in der Korrelationsmatrix durch eine kleinere Submatrix beschrieben wird, stellt die Position immer die erste Zeile bzw. Spalte der Submatrix in der Korrelationsmatrix dar. Die Methode getAllCorrelation wird im folgenden Abschnitt genauer erläutert.

Da die Korrelationsmatrix symmetrisch ist, wird nur die rechte obere Dreiecksmatrix belegt und in den Zeilen 37-43 wird diese dann auf die linke untere kopiert. Dadurch wird die Geschwindigkeit des Algorithmus nochmals erhöht.

Die Methode getAllCorrelation

Die Methode getAllCorrelation, welche in Subklassen des Typs CorrelationFunction (siehe Abbildung 3.6) definiert ist, soll alle Korrelationswerte zwischen zwei Stützstellen berechnen. Diese werden in die Korrelationsmatrix eingetragen und von der Funktion createCorrelationMatrix (siehe Kapitel 3.4) aufgerufen. Das folgende Listing zeigt den Originalcode in C++.

```
template <class T,class B>
inline void CorrelationFunctionGauss<T,B>::getAllCorrelation(
        Point<T,B> &point1,
        Point<T,B> &point2,
        T &corrMatrix,
        B diag,
        size_t iMatrix,
        size_t jMatrix,
        bool equalMember){

    typename map<size_t, B>::iterator derIt;
    typename map<size_t, B>::iterator derItCol;
    size_t freevarNr=0, freevarNr2=0;
    if(equalMember){
         corrMatrix[iMatrix][jMatrix]= 1.0;
         for(size_t i=0; i<point2.getNumPartDerivatives();i++){
            freevarNr=i;
            corrMatrix[iMatrix+i+1][jMatrix+i+1] = fmath::expd(thetas[0][freevarNr]);
         }
    }else{
        corrMatrix[iMatrix][jMatrix] = calcSimpleGauss(point1, point2);
        size_t i=0;
        size_t j=0;
        for(derItCol=point2.getAllPartDervsRef().begin();
                    derItCol!=point2.getAllPartDervsRef().end();derItCol++){

            freevarNr=derItCol->first;
            corrMatrix[iMatrix][jMatrix+i+1] = calcGEKPartialDerivative(point1, point2,
                                                corrMatrix[iMatrix][jMatrix], freevarNr);

            i++;
        }
        if(corrMatrix.getRowSize()>1){
            i=0;
            for(derItCol=point1.getAllPartDervsRef().begin();
                        derItCol!=point1.getAllPartDervsRef().end();derItCol++){

                freevarNr=derItCol->first;
                corrMatrix[iMatrix+i+1][jMatrix] = -calcGEKPartialDerivative(point1, point2,
                                                    corrMatrix[iMatrix][jMatrix], freevarNr);
                i++;
            }
            i=0;
            for(derItCol=point1.getAllPartDervsRef().begin();
                        derItCol!=point1.getAllPartDervsRef().end(); derItCol++){
                freevarNr=derItCol->first;
                j=0;
                for(derIt=point2.getAllPartDervsRef().begin();
                    derIt!=point2.getAllPartDervsRef().end();derIt++){

                    freevarNr2 = derIt->first;
                    if(i==j)
                        corrMatrix[iMatrix+i+1][jMatrix+j+1]=calcGEKPartialDerivative2(point1,
                                                            point2,
                                                            corrMatrix[iMatrix][jMatrix],
                                                            freevarNr);
                    else
                        corrMatrix[iMatrix+i+1][jMatrix+j+1] = calcGEKPartialDerivative2(point1,
                                                                point2,
                                                                corrMatrix[iMatrix][jMatrix],
                                                                freevarNr,
                                                                freevarNr2);
                    j++;
                }
                i++;
            }
        }
    }
}
```

Die Methode wird mit sieben Parametern aufgerufen. Die ersten beiden Parameter sind Referenzen auf zwei Objekte vom Typ Point. Zwischen diesen beiden Stützstellen sollen alle entsprechenden Korrelationswerte berechnet werden. Der nächste Parameter corrMatrix ist eine Referenz auf die Korrelationsmatrix, in welche die entsprechenden Korrelationswerte geschrieben werden sollen. Wie bereits im vorherigen Abschnitt ist der Parameter diag, ein Diagonalaufschlag für die Korrelationsmatrix. Dieser wird auf die Hauptdiagonale der Matrix addiert und kann für die numerische Stabilität der Invertierung von Bedeutung sein, in Kapitel 5.2 wird der Diagonalaufschlag genauer erläutert. Die nächsten beiden Parameter iMatrix und jMatrix geben an, bei welchen Indizes in der Korrelationsmatrix die neuen Korrelationen eingefügt werden sollen.

Der letzte Parameter equalMember ist ein bool'scher Wert und gibt an, ob es sich bei den beiden Stützstellen um dieselben Punkte handelt. Ist dies der Fall, kann die Berechnung der Korrelationswerte stark vereinfacht werden.

In den Zeilen 11-13 werden einige Variablen deklariert. Insbesondere zwei Iteratoren, welche zum iterieren über die partiellen Ableitungen der beiden Stützstellen dienen. Dies ist nötig, da die partiellen Ableitungen als map gespeichert sind und eine map in C++ nur über Iteratoren durchlaufen werden kann.

Die Zeilen 14-19 werden ausgeführt, wenn die beiden Stützstellen identisch sind. Dann vereinfachen sich die Korrelationswerte für eine Gauss Korrelation wie sie in Kapitel 3.3 beschrieben wurde zu:

$$c\left(\vec{x}_1, \vec{x}_1\right) = 1$$

$$\frac{\partial c\left(\vec{x}_1, \vec{x}_1\right)}{\partial x_1^p} = 0$$

$$\frac{\partial c\left(\vec{x}_1, \vec{x}_1\right)}{\partial x_1^p \partial x_1^j} = \begin{cases} 0 & p \neq j \\ e^{\theta_k} & p = j \end{cases}$$

Übertragen auf das Beispiel aus Gleichung 3.2, vereinfacht sich die entsprechende Korrelationsmatrix dann zu:

$$R = \left[\begin{array}{c|cccc} & Z\left(\vec{x}_1\right) & \frac{\partial Z(\vec{x}_1)}{\partial x_1^1} & Z\left(\vec{x}_2\right) & \frac{\partial Z(\vec{x}_2)}{\partial x_1^1} \\ \hline Z\left(\vec{x}_1\right) & 1 & 0 & c\left(\vec{x}_1, \vec{x}_2\right) & \frac{\partial c(\vec{x}_1, \vec{x}_2)}{\partial x_2^1} \\ \frac{\partial Z(\vec{x}_1)}{\partial x_1^1} & 0 & e^{\theta_k} & \frac{\partial c(\vec{x}_1, \vec{x}_2)}{\partial x_1^1} & \frac{\partial c(\vec{x}_1, \vec{x}_2)}{\partial x_1^1 \partial x_2^1} \\ Z\left(\vec{x}_2\right) & c\left(\vec{x}_2, \vec{x}_1\right) & \frac{\partial c(\vec{x}_2, \vec{x}_1)}{\partial x_1^1} & 1 & 0 \\ \frac{\partial Z(\vec{x}_2)}{\partial x_1^1} & \frac{\partial c(\vec{x}_2, \vec{x}_1)}{\partial x_2^1} & \frac{\partial c(\vec{x}_2, \vec{x}_1)}{\partial x_2^1 \partial x_1^1} & 0 & e^{\theta_k} \end{array}\right]$$

Diese Vereinfachung gilt nur für die Gauss Korrelationsfunktion, für andere Korrelationsfunktionen ergeben sich aber ähnliche Vereinfachungen.

Handelt es sich bei den beiden Stützstellen allerdings nicht um dieselben, wird der else-Fall ab Zeile 21 aufgerufen. In dieser Zeile wird dann der einfache Korrelationswert der Matrix gebildet und in die gesamte Korrelationsmatrix eingetragen. Der Code für die private Methode calcSimpleGauss ist identisch mit dem Code für die Gauss Korrelationsfunktion aus Kapitel 3.3.

In den Zeilen 24-32 wird die Ableitung der Korrelationsfunktion zwischen den Stützstellen gebildet, abgeleitet wird nach den Parametern der zweiten Stützstelle. Die Schleife geht alle freien Variablen durch, an denen es eine partielle Ableitung gibt. Die eigentlichen Korrelationswerte werden in der privaten Methode calcGEKPartialDerivative berechnet und danach in die gesamte Korrelationsmatrix geschrieben, in der oberen Beispielmatrix würde das dem Wert $\frac{\partial c(\vec{x}_1,\vec{x}_2)}{\partial x_2^1}$ aus der rechten oberen Ecke entsprechen, da es in dem Beispiel nur eine freie Variable gibt. Der Methode calcGEKPartialDerivative werden anschließend die beide Punkte übergeben. Der genaue Funktionsablauf soll hier aus Platzgründen nicht weiter aufgeführt werden.

Die Abfrage in Zeile 33 prüft, ob es sich bei der Korrelationsmatrix um einen Vektor handelt. Ist dies der Fall, sind nachfolgenden Berechnungen nicht notwendig.

In den Zeilen 35-42 wird wie bereits in den Zeilen 24-32 die Ableitung der Korrelationsfunktion berechnet. In diesem Fall allerdings für die erste Stützstelle, dies würde in der Beispielmatrix dem Wert $\frac{\partial c(\vec{x}_1,\vec{x}_2)}{\partial x_1^1}$ entsprechen.

In den Zeilen 44-66 werden die zweiten Ableitungen der Korrelationsfunktion gebildet. Dafür muss über die freien Variablen von beiden Stützstellen iteriert werden, an denen sich partielle Ableitungen befinden. Die Berechnung der Ableitungen findet in der privaten Methode calcGEKpartialDerivative2 statt. Es gibt zwei verschiedene Implementationen der Methode, einmal eine für den Fall, dass die freien Variablen, nach denen abgeleitet wird für beide Punkte gleich sind und einmal für den Fall, dass sich diese unterscheiden. In der Beispielmatrix würde die Ableitung dem Wert $\frac{\partial c(\vec{x}_1,\vec{x}_2)}{\partial x_1^1 \partial x_2^1}$ entsprechen. Zudem besitzen die beiden Punkte jeweils nur eine freie Variable und auch nur jeweils eine partielle Ableitung. Daher ist die Nummer der freien Variablen nach denen abgeleitet wird, in beiden Fällen eins und damit würde die entsprechende Implementation aus den Zeilen 52-56 aufgerufen werden.

Diese Methode stellt einen recht allgemeingültigen Algorithmus auf, welcher alle notwendigen Korrelationswerte zwischen zwei Stützstellen in die korrekten Positionen einer Korrelationsmatrix einfügt. Im UML Diagramm 3.6 wurde zusätzlich zur Methode getAllCorrelation eine Methode getAllCorrelationPartialDer gelistet. Diese soll die Ableitungen der Korrelationsfunktion nach den Hyperparametern zurückgeben. Dies ist für das Training, welches in Kapitel 5 erklärt wird, wichtig. Die eigentliche Methode ist der Methode getAllCorrelation allerdings relativ ähnlich und soll daher nicht näher dargestellt werden.

4 Bestimmung der Hyperparameter durch die Maximum Likelihood Methode

Wie in den vorherigen Kapiteln gezeigt wurde, hängen die einzelnen Korrelationswerte der Korrelationsmatrix maßgeblich von den verwendeten Hyperparametern ab und damit die Güte des Kriging Modells. Ziel eines Kriging Trainings ist es daher, die optimalen Hyperparameter zu finden. Um dies zu erreichen, wird die Maximum Likelihood Methode verwendet.

Im ersten Teil des Kapitels soll die Maximum Likelihood Methode anhand eines simplen Beispiels erklärt werden. Im Anschluss daran wird die Umsetzung dieser Methode für das hier verwendete Kriging Modell gezeigt.

Der letzte Abschnitt behandelt dann die softwaretechnische Umsetzung dieser Methode.

4.1 Maximum Likelihood Methode

Um die optimalen Hyperparameter für das Kriging Modell zu finden, wird die Maximum Likelihood Methode verwendet. Diese Methode soll in diesem Abschnitt anhand eines einfachen Beispiels erklärt werden und im nächsten Abschnitt die Umsetzung dieser Methode für das Kriging Verfahren. Grundsätzlich ist die Maximum Likelihood Methode ein parametrisches Schätzverfahren in der Statistik. Es werden die Parameter gewählt, die bei einer vorgegebenen Verteilung am plausibelsten erscheinen.

Es wird von einer Zufallsvariablen X ausgegangen, die Dichtefunktion dieser Zufallsvariablen f hängt von einem oder mehreren Parametern ab. Die könnten z.B. bei einer Normalverteilung der Erwartungswert und die Standardabweichung sein. Der Einfachheit halber wird im folgenden ein Parameter p verwendet. Liegt nun eine einfache Stichprobe mit n Proben $x_1..x_n$ vor, lässt sich die Dichtefunktion in folgender Form faktorisieren, die dabei entstehende Funktion L wird auch Likelihood Funktion genannt:

$$L(p) = \prod_{i=1} f(x_i, p) \tag{4.1}$$

Wird diese Funktion nun in Abhängigkeit von q maximiert, erhält man die Maximum-Likelihood-Schätzung für q. Man erhält also die Dichtefunktion bei der die Werte $x_1..x_n$

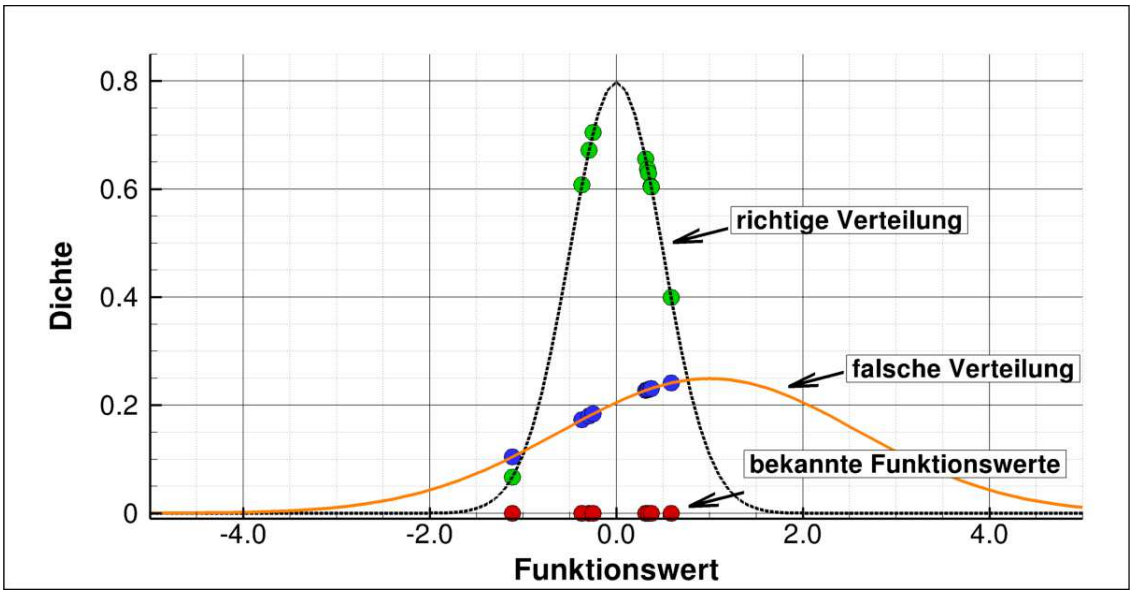


Abbildung 4.1: Zwei Normalverteilungen bei gegebenen Stichproben, die Stichproben wurden gemäß der gestrichelten Normalverteilung erzeugt.

die größte Dichte unter Abhängigkeit von q haben und welche damit auch am plausibelsten erscheint.

In Abbildung 4.1 ist ein einfaches Beispiel dargestellt. Es wurden für das Beispiel zufällig 9 Punkte erzeugt (bekannte Funktionswerte). Hierfür wurde ein Zufallsgenerator verwendet, welcher der schwarzen gestrichelten Verteilung folgt. Dieser stellt also die optimale Verteilungsfunktion für die Stichprobe dar. Aufgabe einer Maximum-Likelihood-Schätzung wäre es nun, die plausibelste Normalverteilung für die gegebene Stichprobe zu finden. Die Parameter für die Schätzung wären hier die Standardabweichung σ und der Erwartungswert μ einer Normalverteilung. In Abbildung 4.1 wurden zwei Normalverteilungen mit zwei verschiedenen Parametersätzen gebildet, wobei die schwarze gestrichelte Linie die richtige und die durchgezogenen orangene die falsche Normalverteilung darstellen. Berechnet man nun die Likelihood Funktion (Gleichung 4.1) für beide Verteilungen, so sieht man bereits anhand der Grafik, dass bei der falschen Verteilung so gut wie alle Funktionswerte kleiner sind als die Funktionswerte bei der richtigen Verteilung. Daher wird auch das Produkt dieser Funktionswerte bei der richtigen Verteilung größer sein und den höheren Likelihood Wert bekommen.

4.2 Maximum Likelihood für Kriging Modelle

Um die Maximum Likelihood Methode für das Kriging Modell zu nutzen, wird angenommen, dass die Funktionswerte normalverteilt sind. Daraus ergibt sich eine multivariate Normalverteilung mit konstantem Erwartungswert:

$$N = \frac{1}{(2\pi)^{\frac{n}{2}} \det(\mathbf{Cov})^{\frac{1}{2}}} e^{-\frac{1}{2}\left(\vec{y}_s - \vec{F}\beta\right)^T \mathbf{Cov}^{-1}\left(\vec{y}_s - \vec{F}\beta\right)}$$

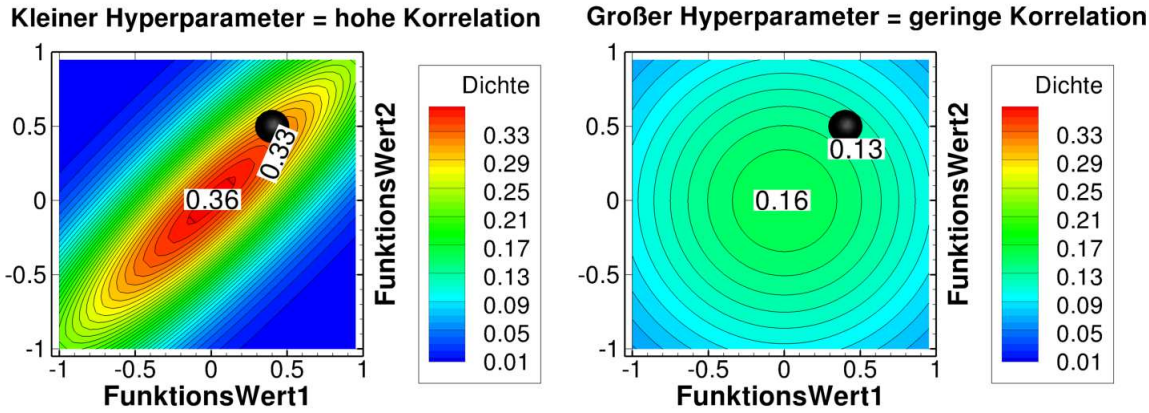


Abbildung 4.2: Multivariate Normalverteilung mit zwei Funktionswerten. Links mit hoher Korrelation zwischen den Funktionswerten und rechts mit niedriger Korrelation.

Wobei der Vektor $\vec{F}$ beim Ordinary-Kriging dem Vektor $\vec{1}$ entspricht und beim Gradient-Enhanced-Kriging sind die ersten $n-m$ Einträge Eins und für die restlichen m Einträge Null. Unter der Annahme einer globalen Varianz σ^2 ergibt sich:

$$N = \frac{1}{(2\pi)^{\frac{n}{2}}\,\sigma^{2n}\,\det\left(\mathbf{R}\left(\overrightarrow{\theta}\right)\right)^{\frac{1}{2}}} e^{-\frac{1}{2}\left(\vec{y}_s-\vec{F}\beta\right)^T \frac{1}{\sigma^2}\mathbf{R}\left(\overrightarrow{\theta}\right)^{-1}\left(\vec{y}_s-\vec{F}\beta\right)}$$

Die Matrix $\mathbf{R}$ wird wie in Kapitel 3 beschrieben durch Korrelationsfunktionen einer echten Korrelationsmatrix angenähert und ist somit von Hyperparametern $\vec{\theta}$ abhängig. Der Maximum Likelihood Schätzer muss die Hyperparameter finden, die den Dichtefunktionswert der Normalverteilung N maximieren. In Abbildung 4.2 sind zwei beispielhafte Normalverteilungen gezeigt. Es gibt in diesen Verteilungen jeweils zwei verschiedene Funktionswerte, der Vektor $\vec{y}_s$ beinhaltet also genau zwei Werte. Die Funktionswerte befinden sich in der Grafik in der rechten oberen Ecke, der Schwarze Punkt markiert die entsprechende Stelle. Ein Maximum Likelihood Schätzer müsste in diesem Fall den höchsten Dichtefunktionswert für den schwarzen Punkt finden. Um dies zu veranschaulichen wurden auf der linken Seite die Hyperparameter so eingestellt, dass eine sehr hohe Korrelation zwischen den Werten angenommen wurde. Der Dichtefunktionswert beträgt hier 0,33. Bei der rechten Grafik wurden die Hyperparameter so eingestellt, dass so gut wie keine Korrelation zwischen den beiden Funktionswerten angenommen wurde. Der Dichtefunktionswert beträgt hier nur 0,13. Der Maximum Likelihood Schätzer würde hier also die Hyperparameter aus der linken Funktion bevorzugen.

Da in der Realität meist hunderte oder tausende Funktionswerte vorhanden sind, wird die Normalverteilung N numerisch maximiert. Um dies so effizient wie möglich zu gestalten, lässt sich die Normalverteilung N noch auf folgende Weise vereinfachen. Der Logarithmus ist eine monotone Funktion. Daher ist das Maximum von N und log(N) an der selben Stelle im Parameterraum.

$$\log(N) = \log(1) - \left(\frac{n}{2}\log(2\pi) + \frac{1}{2}\log\left(\sigma^{2n}\det(\mathbf{R})\right)\right) - \frac{1}{2}\left(\vec{y}_s - \beta\overrightarrow{F}\right)^T \frac{1}{\sigma^2}\mathbf{R}^{-1}\left(\vec{y}_s - \beta\overrightarrow{F}\right)$$

Da das Maximum der Funktion bestimmt werden soll, können die Konstanten ignoriert werden:

$$\log(N) = -\frac{1}{2}\log\left(\sigma^{2n}\det(\mathbf{R})\right) - \frac{1}{2}\left(\vec{y}_s - \beta\overrightarrow{F}\right)^T \frac{1}{\sigma^2}\mathbf{R}^{-1}\left(\vec{y}_s - \beta\overrightarrow{F}\right)$$

$$\log(N) = -\frac{n}{2}\log\left(\sigma^2\right) - \left(\frac{1}{2}\log\left(\det(\mathbf{R})\right)\right) - \frac{1}{2}\left(\vec{y}_s - \beta\overrightarrow{F}\right)^T \frac{1}{\sigma^2}\mathbf{R}^{-1}\left(\vec{y}_s - \beta\overrightarrow{F}\right) \quad (4.2)$$

Der Maximum Likelihood Schätzer der globalen Varianz σ^2 für eine multivariate Normalverteilung ist:

$$\sigma^2 = \frac{1}{n}\left(\vec{y}_s - \beta\overrightarrow{F}\right)^T \mathbf{R}^{-1}\left(\vec{y}_s - \beta\overrightarrow{F}\right)$$

Setzt man dies in den rechten Teil von Gleichung 4.2 ein, erhält man die vereinfachte Form:

$$\log(N) = -\frac{1}{2}\left(\log\left(\sigma^2\right)n + \log\left(\det(\mathbf{R})\right) + n\right) \quad (4.3)$$

Dieser Term ist nur noch abhängig von den Hyperparametern $\vec{\theta}$, welche zur Bildung der Korrelationsmatrix $\mathbf{R}$ und σ^2 benötigt werden:

$$L\left(\vec{\theta}\right) = -\frac{1}{2}\left(\log\left(\sigma^2\left(\overrightarrow{\theta}\right)\right)n + \log\left(\det(\mathbf{R}\left(\overrightarrow{\theta}\right))\right) + n\right)$$

Um diesen Maximum Likelihood Term zu minimieren bzw. zu maximieren, gibt es zahlreiche numerische Verfahren. Innerhalb dieser Arbeit wurden die beiden Verfahren RPROP (Resilient Propagation) und Quasi Newton verwendet. Diese Verfahren werden in Kapitel 5 genauer erläutert. In diesem Zusammenhang ist allerdings wichtig zu wissen, dass für beide Verfahren der Gradient $\nabla L\left(\vec{\theta}\right)$ der Funktion bekannt sein muss. Da die Funktion allerdings die beiden Funktionen σ^2, β enthält und diese ebenfalls von $\vec{\theta}$ abhängig sind, ergibt sich laut Kettenregel:

$$\nabla L\left(\vec{\theta}\right) = \frac{\partial L}{\partial R}\frac{\partial \mathbf{R}}{\partial \vec{\theta}} + \frac{\partial L}{\partial \beta}\frac{\partial \beta}{\partial \vec{\theta}} + \frac{\partial L}{\partial \sigma^2}\frac{\partial \sigma^2}{\partial \vec{\theta}} \quad (4.4)$$

Der globale Erwartungswert β wird ebenfalls über einen Maximum Likelihood Schätzer für eine multivariate Normalverteilung bestimmt:

$$\frac{\partial\left(\log(N)\right)}{\partial\beta} = \frac{\partial}{\partial\beta}\left[\left(\vec{y}_s - \beta\overrightarrow{F}\right)^T \frac{1}{\sigma^2}\mathbf{R}^{-1}\left(\vec{y}_s - \beta\overrightarrow{F}\right)\right] = 0$$

$$\frac{\partial\left(\log(N)\right)}{\partial\beta} = 0 = -\overrightarrow{F}\mathbf{R}^{-1}\left(\vec{y}_s - \beta\overrightarrow{F}\right) - \left(\vec{y}_s - \beta\overrightarrow{F}\right)^T \mathbf{R}^{-1}\overrightarrow{F}$$

$$\Longrightarrow \beta = \frac{\vec{y}_s^T \mathbf{R}^{-1} \vec{F}}{\vec{F}^T \mathbf{R}^{-1} \vec{F}}$$

Da σ^2 und β durch die MLH Schätzung selbst optimale Werte darstellen, werden die entsprechenden partiellen Ableitungen in Gleichung 4.4 Null und es bleibt:

$$\nabla L\left(\vec{\theta}\right) = \frac{\partial L}{\partial R} \frac{\partial \mathbf{R}}{\partial \vec{\theta}}$$

Da die hier verwendeten numerischen Verfahren minimieren und nicht maximieren, wird die Funktion vom Vorzeichen umgedreht:

$$\min\left(L\left(\vec{\theta}\right)\right) = -\max\left(L\left(\vec{\theta}\right)\right)$$

Bildet man nun die partielle Ableitung nach $\partial\theta_l$ der Likelihood Funktion, wobei $\frac{\partial \det(\mathbf{R})}{\partial \theta_l} = \det(\mathbf{R}) \text{Spur}\left(\mathbf{R}^{-1} \frac{\partial \mathbf{R}}{\partial \theta_l}\right)$ und $\frac{\partial \mathbf{R}^{-1}}{\partial \theta_l} = -\mathbf{R}^{-1} \frac{\partial \mathbf{R}}{\partial \theta_l} \mathbf{R}^{-1}$

$$\frac{\partial L\left(\vec{\theta}\right)}{\partial \theta_l} = \frac{1}{2}\left[\text{Spur}\left(\mathbf{R}^{-1}\frac{\partial \mathbf{R}}{\partial \theta_l}\right) - \frac{1}{\sigma^2}\left(\vec{y}_s - \beta\overrightarrow{F}\right)^T \mathbf{R}^{-1}\frac{\partial \mathbf{R}}{\partial \theta_l}\mathbf{R}^{-1}\left(\vec{y}_s - \beta\overrightarrow{F}\right)\right] \tag{4.5}$$

Mit dieser Ableitung ist es nun möglich, die numerischen Minimierer effizient zu nutzen, um die optimalen Hyperparameter zu bestimmen.

4.3 Softwaretechnische Umsetzung

Um den Likelihood Term und seine partiellen Ableitungen zu bilden, wird eine eigene Klasse vorgesehen. Da mehrere Likelihood Funktionen denkbar wären, wird eine abstrakte Klasse namens DensityFunction eingeführt. Abbildung 4.3 zeigt die Umsetzung der Klasse als UML Diagramm, Getter und Setter Methoden wurden hier aus Platzgründen ausgelassen. Bisher ist nur die Likelihood Funktion umgesetzt (siehe Kapitel 4.2). Der Likelihood Term (Gleichung 4.3) wird in der Methode calcDensity() berechnet und die entsprechende Ableitung (Gleichung 4.5) in calcDensityDerivative(). Denkbar wären allerdings auch andere Likelihood Funktionen, welche auf anderen Verteilungen basieren.

Ziel der Klassenstruktur ist es, die beiden Gleichungen 4.3 und 4.5 so effizient wie möglich zu lösen. Die dort verwendeten Matrizen sind in der Regel sehr groß und voll besetzt, was eine effiziente Berechnung sehr wichtig macht. Allerdings sind die Matrizen symmetrisch und positiv definit, was wiederum einige Optimierungen zulässt. Zur Vereinfachung werden einige Terme der Gleichungen zusammengefasst:

$$\vec{R}_f = \mathbf{R}^{-1}\overrightarrow{F} \tag{4.6}$$

$$f_{rf} = \vec{F}^T \vec{R}_f \tag{4.7}$$

$$\overrightarrow{e} = (\vec{y}_s - \beta * \overrightarrow{F}) \tag{4.8}$$

$$\overrightarrow{d} = \mathbf{R}^{-1} \overrightarrow{e} \tag{4.9}$$

Die Terme werden in diese Form auch in der entsprechenden Klasse (ReducedNormalDistribution) einmal berechnet und dann gespeichert. Da diese sehr häufig wiederverwendet werden, muss man diese Termen nur einmal berechnen und eine Änderung ist nur notwendig, wenn die Hyperparameter verändert wurden. Über die Attribute corrMatUpdate und vecUpdate wird festgelegt, ob die Matrizen und Vektoren neu berechnet werden müssen oder nicht. Sie werden nur dann neu berechnet, wenn die Hyperparameter verändert wurden. Da die Hyperparameter nur über Getter und Setter Methoden zugänglich sind, kann man bei jedem Setter Zugriff auf die Hyperparameter die Attribute corrMatUpdate und vecUpdate auf true setzen.

Viele der Attributnamen entsprechen den hier verwendeten Bezeichnungen für die Vektoren/Matrizen, z.B. der Vektor $\vec{R}_f$ entspricht dem Attribut rf. Das Attribut logDeterminantR entspricht dem Logarithmus der Determinante der Korrelationsmatrix $\log(\det(\mathbf{R}))$. Aufgrund dieser Ähnlichkeit werden daher nicht alle einzeln aufgeführt.

Wie bereits in Kapitel 3.1 beschrieben, lässt sich die Determinante durch eine Cholesky Zerlegung sehr effizient berechnen. Insbesondere da die Zerlegung ebenfalls für die Invertierung der Matrix sinnvoll ist. Die Korrelationsmatrix wird in der Methode createAndInvertCorrelmat aufgestellt, zerlegt und dann invertiert. Die zerlegte Matrix wird in decomposedMatrix gespeichert, die invertierte Matrix in inverseCorrMatrix. Die benötigten Vektoren werden in der Methode calcVectors() berechnet und in den Attributen der Klasse gespeichert. Die Methoden predict() und predictVariance() berechnen dann unter Vorgabe eines Ortsvektors eine Schätzung der gesuchten Funktion $y^*(\vec{x}_0)$ (siehe Gleichung 2.1) und der Varianz (Gleichung 2.12).

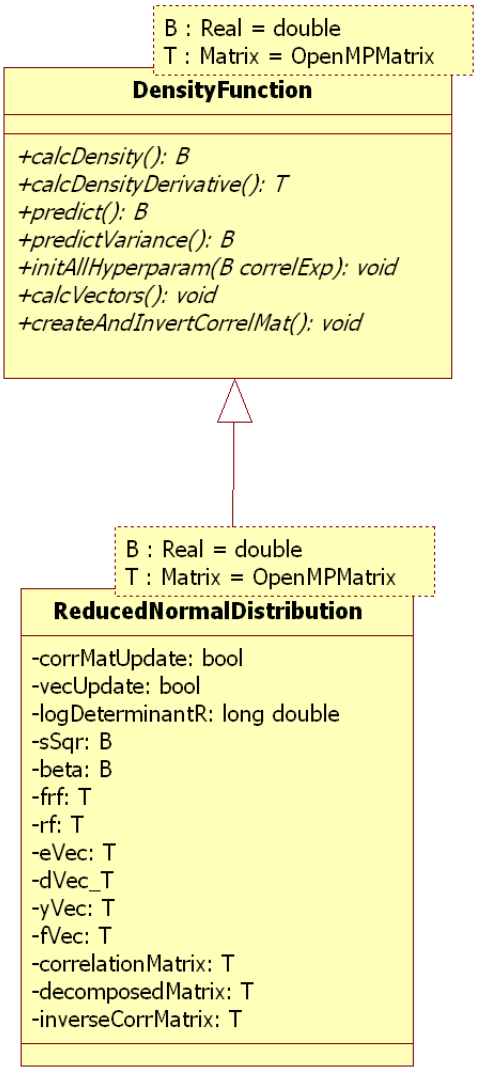


Abbildung 4.3: UML Diagramm der abstrakten Superklasse DensityFunction und der Subklasse ReducedNormalDistribution

Schritte für die Berechnung eines Likelihood Terms:

In diesem Abschnitt soll die Methode zur Berechnung des Likelihood Terms (calcDensity()) nochmals genauer erklärt werden. Das folgende Listing zeigt die Methode im Originalcode:

```
1  template <class T, class B>
2  B ReducedNormalDistribution<T,B>::calcDensity(){
3      try{
4          if(this->corrMatUpdate){
5              this->createAndInvertCorrelMat();
6              this->corrMatUpdate = false;
7          }
8          if(this->vecUpdate){
9              this->calcVectors();
10             this->vecUpdate = false;
11         }
12     }
13     catch(ChodecNotPosDef& e){
14         cout <<"calcDensity() MatrixExceptions:"<<e.what()<<endl;
```

```
15          config::diagonalAddition = log(2.0*exp(config::diagonalAddition));
16
17          this->getCorrelationMatrixRef().saveAsAscii("corrMatFailed");
18          this->getInverseCorrelationMatrixRef().saveAsAscii("corrMatInverseFailed");
19          return  (config::numSamples*1000.0);
20      }
21      catch(...){
22          cout <<"calcDensity() Exception"<<endl;
23          cout <<"-----------------------------------"<<endl;
24          return  (config::numSamples*1000.0);
25      }
26      return 0.5*(log(sSqr)*config::numSamples + logDeterminantR + config::numSamples );
27  }
```

In Zeile 4 wird zuerst überprüft, ob die Korrelationsmatrix neu aufgestellt werden muss oder nicht. Dies geschieht, wie bereits beschrieben, mit dem Attribut corrMatUpdate-Ist dieses true, dann wird die Korrelationsmatrix wie in Zeile 5 über die Methode createAndInvertCorrMat() erzeugt, eine Cholesky Zerlegung durchgeführt und dann invertiert. Nach erfolgreichem Aufruf der Methode wird das Attribut corrMatUpdate wieder auf false gesetzt. Ansonsten werden die Matrizen im Puffer verwendet, also correlationMatrix, decomposedMatrix und inverseCorrMat.

Auf dieselbe Weise wird mit den Vektoren (Gleichungen 4.6-4.9) in den Zeilen 8-10 verfahren. Diese werden durch die Methode calcVectors erzeugt und dann in den Attributen der Klasse gespeichert. Durch die verwendete Matrix Klasse ist die Berechnung der einzelnen Vektoren/Matrizen sehr simpel. Das folgende Listing zeigt dies exemplarisch an der Methode zur Erzeugung von Vektor $\vec{d}$.

```
template <class T, class B>
void ReducedNormalDistribution<T,B>::calcDVec(){
    dVec = inverseCorrelationMatrix.matrixMultiplicationTranspose(eVec);
}
```

Die Methode ist Mitglied der Klasse ReducedNormalDistribution (siehe Abbildung 4.3) und wird durch die Methode calcVectors() aufgerufen. Das Attribut inverseCorrelationMatrix ist vom Typ Matrix und beinhaltet die inverse Korrelationsmatrix. Diese wird mit dem transponierten Vektor $\overrightarrow{e} = (\vec{y}_s - \beta * \overrightarrow{F})$ multipliziert, die Transposition wird innerhalb der Multiplikation vorgenommen. Die anderen Methoden zur Berechnung der Dichtefunktionswerte usw. beinhalten prinzipiell nur andere Matrix Operationen und werden daher nicht alle aufgeführt.

Die Methode InitAllThetas() der Klasse DensityFunction und deren Subklassen soll alle Hyperparameter mit möglichst sinnvollen Werten initialisieren. Die verschiedenen Intialisierungsmöglichkeiten und deren Umsetzung werden in Abschnitt 5.2 noch genauer erläutert.

Da die Cholesky Zerlegung nur für positiv definite symmetrische Matrizen funktioniert, kann es bei der Zerlegung zu einer Exception vom Typ ChodecNotPosDef kommen, dies wird in Zeile 13 abgefangen. Ist diese Exception aufgetreten, wird der Diagonalaufschlag (siehe Kapitel 5.2) erhöht und ein sehr hoher Likelihood Wert zurückgegeben (Zeile 19), damit dieser in der Minimierung nicht mehr berücksichtigt wird.

Zusätzlich wird innerhalb der Matrix Klasse nach erfolgreicher Invertierung eine kurze Überprüfung der invertierten Matrix gemacht. Dies wird durch folgende Gleichung erreicht:

$$\text{Spur}\left(RR^{-1}\right) = n$$

Das Produkt der Inversen und der Korrelationsmatrix, ergibt die Einheitsmatrix. Da die Einheitsmatrix n Diagonalelemente besitzt, welche alle den Wert 1.0 haben, muss die Spur der multiplizierten Matrizen n ergeben. Gibt es numerische Ungenauigkeiten innerhalb der Invertierung, wird dieser Wert wahrscheinlich von n abweichen. Dies wird überprüft und bei Überschreitung eines Grenzwertes wird ebenfalls eine Exception geworfen. Diese wird mit allen anderen unbekannten Exceptions in Zeile 21 gefangen und als Reaktion ein sehr hoher Likelihood Wert zurückgegeben.

Nachdem alle Vektoren und Werte berechnet sind, wird in Zeile 26 die eigentliche Likelihood Funktion berechnet (siehe Gleichung 4.1) und zurückgegeben.

5 Minimierungsverfahren/Training

Im Kapitel 4.2 wurde die Maximum Likelihood Methode vorgestellt. Als Ergebnis dieses Kapitels erhielt man zwei Gleichungen zur Berechnung des Likelihood Terms und der dazugehörigen Ableitung. Ziel ist es, für den Likelihood Term die optimalen Hyperparameter zu finden. Dieser Vorgang ist das eigentliche Training des Modells. Zu diesem Zweck werden zwei numerische Minimierungsverfahren vorgestellt. Beide Minimierungsverfahren waren bereits in einer institutseigenen Bibliothek vorhanden. Die entwickelte Minimierungsklasse sollte beide Verfahren nutzen können. Hierfür wurde ein spezielles Klassenmodell unter Benutzung von Boost Funktionsobjekten entwickelt.

Das Training bringt einige zusätzliche Probleme mit sich, z.B. müssen die Hyperparameter anfangs initialisiert werden. Zudem kann die Korrelationsmatrix schlecht konditioniert sein, für beide Probleme wurden Lösungsansätze entwickelt, welche hier vorgestellt werden.

5.1 Diagonalaufschlag und Vermeidung negativer Hyperparameter

In diesem Abschnitt soll darauf eingegangen werden, welche Probleme negative Hyperparameter hervorrufen und wie man diese vermeiden kann.

Zudem wurde in vorherigen Kapiteln bereits ein Diagonalaufschlag erwähnt, dieser wird auf die Diagonalelemente der Korrelationsmatrix addiert und soll die Kondition der Matrix verbessern. Die genaue Vorgehensweise soll in diesem Abschnitt erläutert werden.

Negative Hyperparameter

Während der Minimierung der Likelihood Funktion kann es je nach gewähltem Minimierungsverfahren oftmals dazu kommen, dass die Hyperparameter negativ werden können. Bei der Gauss Korrelationsfunktion würde dies dazu führen, dass bei großen Abständen zwischen den Membern auch große Korrelationen vorhergesagt werden würden. Die Korrelationen könnten so auch einen Wert über Eins erhalten, was keinen Sinn macht. Daher sind negative Hyperparameter bei der Gauss Funktion zwingend zu vermeiden.

Eine einfache Möglichkeit dies zu tun, wäre während der Minimierung die Hyperpa-

rameter nach unten zu begrenzen. Dies kann bei einigen Minimierungsverfahren allerdings zu schwerwiegenden Problemen führen, sodass diese nicht mehr konvergieren würden.

$$c\left(\vec{x}_1, \vec{x}_2\right) = e^{-\frac{1}{2}\sum_{l=1}^{l<k}\left(\theta_l |x_{1l} - x_{2l}|^2\right)}$$

Wünschenswert wäre es also, dass der gesamte Bereich der reellen Zahlen verwendet werden könnte. Um dies zu erreichen, wurde als erstes versucht, das Quadrat der Hyperparameter zu verwenden:

$$c\left(\vec{x}_1, \vec{x}_2\right) = e^{-\frac{1}{2}\sum_{l=1}^{l<k}\left(\theta_l^2 |x_{1l} - x_{2l}|^2\right)}$$

Diese Formulierung führte allerdings zu einem unerwünschten Verhalten und zwar sieht die partielle Ableitung dieser Funktion wie folgt aus:

$$\frac{\partial c}{\partial \theta_l} = c\left(\vec{x}_1, \vec{x}_2\right)\left(-\theta_l \left|x_{1l} - x_{2l}\right|^2\right)$$

Das Problem hierbei ist, dass wenn der Hyperparameter während der Optimierung Null wird, auch die entsprechende partielle Ableitung Null wird. Während eines Minimierungsverfahrens kann es also passieren, dass die partiellen Ableitungen der Hyperparameter alle zu Null werden und die Minimierung als beendet angesehen wird. Um dieses Problem zu vermeiden, wurde die Exponentialfunktion verwendet:

$$c\left(\vec{x}_1, \vec{x}_2\right) = e^{-\frac{1}{2}\sum_{l=1}^{l<k}\left(e^{\theta_l} |x_{1l} - x_{2l}|^2\right)}$$

$$\frac{\partial c}{\partial \theta_l} = c\left(\vec{x}_1, \vec{x}_2\right)\left(-\frac{1}{2} e^{\theta_l} \left|x_{1l} - x_{2l}\right|^2\right)$$

Diese Formulierung der Gauss Korrelationsfunktion hat sich bisher als vorteilhaft herausgestellt, da der gesamte Raum der reellen Zahlen verwendet werden kann. Zudem ist die Funktion stetig und differenzierbar.

Diagonalaufschlag einstellen über eine maximale Konditionszahl

Während des Trainings kann es passieren, dass Hyperparameter eingestellt werden, die eine schlechte Konditionierung der Korrelationsmatrix hervorrufen. Dies kann bei der Cholesky Zerlegung problematisch werden. Um die Konditionszahl zu verbessern, wird ein Diagonalaufschlag auf die Hauptdiagonale der Matrix addiert. Die Gauss

Korrelationsfunktion würde sich damit wie folgt ändern ($\delta_{i,j}$ beschreibt hier das Kronecker Delta und λ den Diagonalaufschlag):

$$c\left(\vec{x}_i, \vec{x}_j\right) = e^{-\frac{1}{2}\sum_{l=1}^{l<k}\left(e^{\theta_l}\left|x_{i,l}-x_{j,l}\right|^2\right)} + e^{\lambda}\delta_{i,j}$$

Es stellt sich allerdings die Frage, wie groß dieser Diagonalaufschlag im Einzelfall sein muss und inwiefern dieser das Ergebnis beeinflusst bzw. die Kondition verbessert. Grundsätzlich sollte dieser so klein wie möglich gewählt werden, um die Ursprungsmatrix so wenig wie möglich zu verändern. Um einen optimalen Wert zu ermitteln, gibt es zwei verschiedene Möglichkeiten.

Die erste Möglichkeit wäre, die Likelihood Funktion (siehe Gleichung 4.1) nach dem Diagonalaufschlag zu differenzieren. Die Ableitung ist in diesem Fall sehr simpel, da alles außer dem Diagonalaufschlag wegfällt:

$$\frac{\partial c\left(\vec{x}_i, \vec{x}_j\right)}{\partial \lambda} = e^{\lambda}\delta_{i,j}$$

Die Ableitung der Korrelationsmatrix nach dem Diagonalaufschlag ergibt also eine Einheitsmatrix multipliziert mit dem Diagonalaufschlag:

$$\frac{\partial \mathbf{R}}{\partial \lambda} = e^{\lambda}\mathbf{E}$$

Damit ergibt sich die Ableitung der Likelihood Funktion nach dem Diagonalaufschlag zu:

$$\frac{\partial L}{\partial \lambda} = \frac{e^{\lambda}}{2}\left[\text{Spur}\left(\mathbf{R}^{-1}\right) - \frac{1}{\sigma^2}(\vec{y}_s - \beta * \overrightarrow{F})^T\mathbf{R}^{-1}\mathbf{R}^{-1}(\vec{y}_s - \beta * \overrightarrow{F})\right]$$

Mit dieser Formel wäre es nun möglich, den Diagonalaufschlag einfach als zusätzlichen Hyperparameter zu minimieren. Allerdings bedeutet dies natürlich einen zusätzlicher Aufwand. Es soll daher nach einer einfacheren Methode gesucht werden.

Ein anderer möglicher Ansatz ist die Konditionszahl der Matrix. Diese ist definiert als Quotient aus maximalem und minimalem Eigenwert der Korrelationsmatrix:

$$\kappa = \left|\frac{\Xi_{max}\left(\mathbf{R}\right)}{\Xi_{min}\left(\mathbf{R}\right)}\right|$$

Überlegt man sich nun, welche Korrelationsmatrix am schlechtesten konditioniert wäre, kommt man auf die Einsmatrix:

$$\mathbf{R} = \begin{bmatrix} 1 & \ldots & 1 \\ \ldots & \ldots & \ldots \\ 1 & \ldots & 1 \end{bmatrix}$$

Die minimalen und maximalen Eigenwerte der Einsmatrix sind bekannt:

$$\Xi_{min}\left(\mathbf{R}\right) = 0$$

$$\Xi_{max}\left(\mathbf{R}\right) = n$$

Dies würde einer unendlichen Konditionszahl entsprechen:

$$\kappa = \infty$$

Addiert man nun den Diagonalaufschlag, bekommt man folgende Korrelationsmatrix:

$$\mathbf{R} = \begin{bmatrix} 1+e^{\lambda} & ... & 1 \\ ... & ... & ... \\ 1 & ... & 1+e^{\lambda} \end{bmatrix}$$

Daraus ergeben sich die folgenden neuen maximalen und minimalen Eigenwerte:

$$\Xi_{min}\left(\mathbf{R}\right) = e^{\lambda}$$

$$\Xi_{max}\left(\mathbf{R}\right) = e^{\lambda} + n$$

Und die entsprechende Konditionszahl verbessert sich zu:

$$\kappa = \left| \frac{e^{\lambda} + n}{e^{\lambda}} \right|$$

Da die Matrix positiv definit sein muss und damit nur positive Eigenwerte hat, kann der Betrag weggelassen werden:

$$\kappa = \frac{e^{\lambda} + n}{e^{\lambda}}$$

Wählt man nun für die Konditionszahl eine obere Grenze, bekommt man eine Untergrenze für den Diagonalaufschlag:

$$\kappa_{max} > \frac{e^{\lambda} + n}{e^{\lambda}}$$

$$e^{\lambda} > \frac{n}{(\kappa_{max} - 1)}$$

Eine geeignete Grenze für die obere Grenze der Konditionszahl ist aus der Erfahrung bekannt und liegt bei ca. 10^9, dieser Wert kann im Einzelfall natürlich angepasst werden. Der daraus resultierende Diagonalaufschlag würde bei einer üblichen Matrixgröße von 5000x5000, $5 * 10^{-6}$ betragen und es ist davon auszugehen, dass dieser so gut wie keinen Einfluss auf das Endergebnis haben sollte.

Diese Art der Berechnung des Diagonalaufschlags wird momentan im Code verwendet. Die entsprechende Ableitung der Likelihood Funktion ist im Code bereits umgesetzt, wird momentan allerdings nicht verwendet, da bisherige Tests dafür noch keine Notwendigkeit zeigten.

5.2 Initialisierung der Hyperparameter

Um einen Minimierungsalgorithmus starten zu können, ist eine geeignete Initialisierung der Hyperparameter von großer Bedeutung. Diese kann die Konvergenz und auch die Stabilität der Minimierung stark beeinflussen. Innerhalb dieser Arbeit wurden mehrere Ansätze entwickelt, um eine geeignete Initialisierung zu finden.

Abschätzung konstanter Hyperparameter

Eine sehr einfache und schnelle Möglichkeit die Hyperparameter für eine Gauss Verteilung zu schätzen, wäre einen Erwartungswert für die Einträge in der Korrelationsmatrix zu wählen. Da die Korrelation grundlegend zwischen Eins und Null liegen sollte, ist dies recht einfach. Angenommen der Mittelwert der Korrelationsfunktion soll bei einem Wert von $c_{erw} = \{c_{erw} \in \mathbb{R} | 0 \leq c_{erw} \leq 1\}$ liegen.

$$c_{erw} = \frac{1}{n^2} \sum_{i=1}^{i<n} \sum_{j=1}^{i<k} e^{-\frac{1}{2} \sum_{l=1}^{l<k} \left(e^{\theta_l} \left| x_{i,l} - x_{j,l} \right|^2 \right)}$$

Nimmt man weiterhin an, dass die einzelnen Korrelationswerte nahezu identisch sind:

$$c_{erw} = e^{-\frac{1}{2} \sum_{l=1}^{l<k} \left(e^{\theta_l} \left| x_{i,l} - x_{j,l} \right|^2 \right)}$$

$$\log \left(c_{erw} \left(\vec{x}_i, \vec{x}_j \right) \right) = -\frac{1}{2} \sum_{l=1}^{l<k} \left(e^{\theta_l} \left| x_{i,l} - x_{j,l} \right|^2 \right)$$

Nimmt man ferner an, dass x eine Realisierung einer Zufallsvariablen ist und bildet den Erwartungswert:

$$\log \left(c_{erw} \left(\vec{x}_i, \vec{x}_j \right) \right) = E \left[-\frac{1}{2} \sum_{l=1}^{l<k} \left(e^{\theta_l} \left| x_{i,l} - x_{j,l} \right|^2 \right) \right]$$

Als weitere Vereinfachung sollen alle Hyperparameter den gleichen Wert haben:

$$\log\left(c_{erw}\left(\vec{x}_i, \vec{x}_j\right)\right) = -\frac{1}{2}e^{\theta}E\left[\sum_{l=1}^{l<k}\left(|x_{i,l} - x_{j,l}|^2\right)\right]$$

Die beiden Variablen werden als Zufallsvariablen angenommen und der Betrag wird aufgrund des Quadrats vernachlässigt:

$$\log\left(c_{erw}\right) = -\frac{1}{2}e^{\theta}E\left[\sum_{l=1}^{l<k}\left(x_{i,l} - x_{j,l}\right)^2\right]$$

$$\log\left(c_{erw}\right) = -\frac{1}{2}e^{\theta}\sum_{l=1}^{l<k}\left(E\left[x_{i,l}^2\right] - E\left[2x_{i,l}x_{j,l}\right] + E\left[x_{j,l}^2\right]\right)$$

Nimmt man nun an, dass die Zufallsvariablen unabhängig sind, gilt $E\left[2x_{i,l}x_{j,l}\right] = 0$

$$\log\left(c_{erw}\right) = -\frac{1}{2}e^{\theta}\sum_{l=1}^{l<k}\left(E\left[x_{i,l}^2\right] + E\left[x_{j,l}^2\right]\right)$$

Die Varianz einer Zufallsvariable X ist definiert durch $\text{var}\left(X\right) = E\left[X^2\right] - E\left[X\right]^2$. Die im Modell verwendeten Daten werden grundsätzlich auf einen Erwartungswert von Null und eine Standardabweichung von Eins normiert.

$$E\left[X\right] = 0$$

$$\text{var}\left[X\right] = 1$$

Daraus ergibt sich folgende Formel für das verwendete Modell für die Varianz der Stützstellen:

$$\text{var}\left[X\right] = E\left[X^2\right]$$

Also

$$E\left[x_{i,l}^2\right] = \text{var}\left[x_{i,l}^2\right] = 1$$

und analog dazu:

$$E\left[x_{j,l}^2\right] = \text{var}\left[x_{j,l}^2\right] = 1$$

Daraus folgt:

$$\log\left(c_{erw}\right) = -\frac{1}{2}e^{\theta}\sum_{l=1}^{l<k}(1+1)$$

$$\log\left(c_{erw}\right) = -\frac{1}{2}e^{\theta}2k$$

$$\log\left(-\frac{\log\left(c_{erw}\right)}{k}\right) = \theta \tag{5.1}$$

Mit dieser Formel hat man nun eine Möglichkeit die Hyperparameter zu schätzen. Aufgrund der vielen Annahmen und Vereinfachungen ist dieses Verfahren als heuristisch einzustufen. Die Hyperparameter haben dann allerdings alle denselben Initialwert. Zudem ist der erwartete Korrelationswert c_{erw} unbekannt, dies kann leicht durch einfaches Ausprobieren gelöst werden, da der Wertebereich bekannt ist. Man würde also c_{erw} von 0 bis 1 variieren, damit einen Hyperparameter erhalten und mit diesem Hyperparameter die Likelihood Funktion berechnen. Letztlich wählt man den Hyperparameter, welcher den besten Likelihood Wert aufweist.

Reduktion der Stützstellen

Eine weitere Möglichkeit eine Initialisierung für die Hyperparameter zu finden ist, diese zufällig zu Erzeugen und die entsprechende Likelihood Funktion zu berechnen. Es würden die Hyperparameter gewählt, welche die beste Likelihood Funktion haben. Die zufällige Erzeugung ist extrem zeitaufwendig, da für jeden Satz zufälliger Hyperparameter die Likelihood Funktion ausgewertet werden muss. Um diesen Aufwand zu reduzieren, kann man einfach Stützstellen weglassen. Die Likelihood Funktion sollte sich im Vergleich zumindest ähnlich verhalten. Statt einer zufälligen Veränderung der Hyperparameter kann man auch ein Minimierungsverfahren mit reduzierter Stützstellenzahl verwenden. Dies wurde im Code auch umgesetzt. Die möglichen Minimierungsverfahren sind dieselben wie sie für das eigentliche Training verwendet werden und werden im nächsten Abschnitt beschrieben.

Reduziert man die Anzahl der Stützstellen wird die Korrelationsmatrix dementsprechend kleiner. Dadurch sinkt der Aufwand für die Invertierung und die Matrix Multiplikationen erheblich. Tests zeigten, dass die Initialisierung durch solch ein Verfahren zwar langsamer ist, allerdings konvergiert das Minimierungsverfahren durch die bessere Initialisierung deutlich schneller. Da bei dem Minimierungsverfahren wieder die volle Anzahl der Stützstellen notwendig ist und zusätzlich noch die Ableitungen der

Korrelationsmatrix berechnet werden müssen, bietet dieses Verfahren durch die bessere Initialisierung für das gesamte Training betrachtet eine deutliche Beschleunigung.

Allerdings konnte auch beobachtet werden, dass die Initialisierung häufiger zu lokalen Minima führt. Eine Begründung für dieses Verhalten wurde noch nicht gefunden und sollte weitergehend untersucht werden.

5.3 Minimierungsverfahren

Innerhalb des Kriging Modells wurden zwei verschiedene mehrdimensionale Minimierungsverfahren eingesetzt. Beide Verfahren waren bereits in einer institutseigenen Software Bibliothek verfügbar.

Minimierungsverfahren angelehnt an Resilient Backpropagation

Das erste hier verwendete Minimierungsverfahren ist angelehnt an ein Trainingsverfahren für Neuronale Netzwerke, genannt RPROP (Resilient Backpropagation) [RB93, HS11] und ist ein Verfahren erster Ordnung. Besonderheit des Verfahrens ist, dass es nur das Vorzeichen der partiellen Ableitungen verwendet und nicht den Wert selbst.

Die Änderung der Hyperparameter θ_i für den nächsten Iterationsschritt $t+1$ ergibt sich aus der Schrittweite γ_i. Diese wird für jeden Hyperparameter einzeln bestimmt und in jeder Iteration geändert. Die Änderung hängt nur von dem Vorzeichen der entsprechenden partiellen Ableitung $\frac{\partial f}{\partial \theta_i}$ zum Zeitpunkt t der zu minimierenden Funktion f ab.

$$\theta_i^{t+1} = \theta_i^t - \gamma_i^t \text{sgn}\left(\left(\frac{\partial f}{\partial \theta_i}\right)^t\right)$$

Die Schrittweite wird in jedem Iterationsschritt für jeden Hyperparameter einzeln angepasst. Dies wird über zwei Multiplikatoren erzielt $\eta^+ = \{\eta^+ \in \mathbb{R} | 1 < \eta^+\}$ und $\eta^- = \{\eta^- \in \mathbb{R} | 1 > \eta^-\}$. Ist die entsprechende partielle Ableitung aus dem letzten Schritt multipliziert mit dem jetzigen Schritt größer als Null, wird die Schrittweite erhöht, indem die Schrittweite γ_i^t multipliziert wird mit η^+. Wenn die partielle Ableitung aus dem letzten Schritt multipliziert mit dem jetzigen Schritt kleiner als Null ist, dann wird die Schrittweite verkleinert durch Multiplikation mit η^-. Für die Schrittweite wird zudem eine Unter- und Obergrenze $(\gamma_{min}, \gamma_{max})$ festgelegt.

$$\gamma_i^{t+1} = \begin{cases} \min\left(\gamma_i^t \eta^+, \gamma_{max}\right) & wenn \left(\frac{\partial f}{\partial \theta_i}\right)^t \left(\frac{\partial f}{\partial \theta_i}\right)^{t-1} > 0 \\ \max\left(\gamma_i^t \eta^-, \gamma_{min}\right) & wenn \left(\frac{\partial f}{\partial \theta_i}\right)^t \left(\frac{\partial f}{\partial \theta_i}\right)^{t-1} < 0 \\ \gamma_i^t & sonst \end{cases}$$

Bei sehr flachen Bereichen der zu minimierenden Funktion, wo die partiellen Ableitungen nur sehr klein sind, würden andere Gradientenverfahren nur sehr langsam

bis gar nicht mehr vorwärts kommen. Da dieses Verfahren allerdings die Größe der Gradienten überhaupt nicht berücksichtigt, kann dies nicht passieren. Das ist bei der Likelihood Funktion von besonderem Vorteil, da diese bereits durch Ihre Definition sehr viele flache Gebiete aufweist.

Quasi Newton

Das zweite implementierte Minimierungsverfahren ist ein Verfahren höherer Ordnung namens Quasi Newton. Basis für diese Art der mehrdimensionalen Minimierung ist eine Taylor Approximation zweiten Grades, wobei t der Iterationsschritt ist und $\mathbf{H}$ die Hesse Matrix:

$$f\left(\vec{\theta}\right) \approx f\left(\vec{\theta}_t\right) + \left(\vec{\theta} - \vec{\theta}_t\right)^T \nabla f\left(\vec{\theta}_k\right) + \frac{1}{2}\left(\vec{\theta} - \vec{\theta}_t\right)^T \mathbf{H}\left(\vec{\theta}_t\right)\left(\vec{\theta} - \vec{\theta}_t\right)$$

Die entsprechende Ableitung dieser Funktion muss im Minimum oder Maximum der Funktion Null ergeben:

$$\nabla f\left(\vec{\theta}\right) \approx \nabla f\left(\vec{\theta}_t\right) + \mathbf{H}\left(\vec{\theta}_t\right)\left(\vec{\theta} - \vec{\theta}_t\right) = 0$$

Besonderheit bei der Quasi Newton Methode ist, dass die Hesse Matrix $\mathbf{H}$, nicht direkt berechnet werden muss, sondern sukzessive über die Gradienten angenähert wird. Vorteil des Verfahrens ist, dass es deutlich schneller konvergiert als das bereits vorgestellte Verfahren erster Ordnung. Allerdings ist es weniger robust und kann in flachen Gebieten der Funktion langsam bis gar nicht konvergieren. Die exakte Umsetzung des Algorithmus und weitere Details können in [Pre07, GMW81, Gil07] gefunden werden.

5.4 Softwaretechnische Umsetzung

Klasse für die Steuerung des Trainings

In diesem Abschnitt soll der Ablauf und die dazugehörige Klasse für ein Training eines Kriging Modells erklärt werden. Das UML Diagramm 5.1 zeigt die Klasse Trainer, welche das Training steuern und verwalten soll.

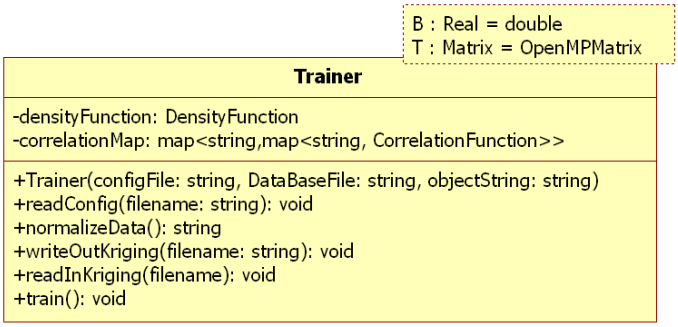


Abbildung 5.1: UML Diagramm der Trainer Klasse, welche das Training des Kriging Modells steuert

Die Methoden und auch Attribute der Trainer Klassen sollen am Ablauf des Trainings erläutert werden. Ein Training besteht im Wesentlichen aus den folgenden Schritten:

1. Der erste Schritt besteht aus der Erzeugung eines Trainer Objekts. Dem Konstruktor müssen drei Parameter übergeben werden: Der Name der Konfigurationsdatei (configFile), der Name der Datenbankdatei (DataBaseFile) und die zu trainierende Funktion. Der Parameter objectString gibt an, welcher der Funktionswerte aus der Datenbankdatei verwendet werden soll, diese kann zu einem Variablensatz mehrere verschiedene Funktionen beinhalten. Innerhalb des Konstruktors werden dann einige Schritte ausgeführt, um das Kriging Modell zu initialisieren.

   (a) Einlesen der Datenbankdatei.

   (b) Einlesen der Konfigurationsdatei durch die Methode readConfig(), an dieser Stelle werden auch die zu verwendenden Korrelationsfunktionen gesetzt (Attribut correlationMap, siehe Kapitel 3.3). Zudem wird ein DensityFunktion Objekt erzeugt und in dem Attribut densityFunction gespeichert, siehe Kapitel 4.2.

   (c) Normalisierung der Stützstellen und der dazugehörigen Funktionswerte (mit der Methode normalizeData())

   (d) Initialisierung der Hyperparameter, siehe Kapitel 5.2.

2. Starten der train() Methode des Trainer Objekts.

   (a) Erzeugung eines Minimierer Objekts, je nach gewähltem Minimierer Typ. Die entsprechende Klassenstruktur wird im nächsten Abschnitt behandelt

   (b) Starten des Minimierers

3. Nach erfolgreichem Training wird eine XML Datei geschrieben, in der im Wesentlichen alle Ergebnisse des Trainings stehen. Es werden die gefundenen Hyperparameter, die Korrelationsmatrix, einige Vektoren usw. gespeichert um bei einer späteren Vorhersage diese Werte nicht mehr berechnen zu müssen. Diese XML Datei beinhaltet also ein fertiges Kriging Modell.

Klassenstruktur zur Steuerung der Minimierungsverfahren

In Abbildung 5.2 wird das UML Diagramm der Klassenstruktur für die Minimierungsverfahren gezeigt. Es gibt eine abstrakte Superklasse MinimizerInterface, welche hier als Interface zu verstehen ist. Diese schreibt die notwendigen Methoden für die Subklassen vor. Die Subklassen sollen dann die konkreten Minimierungsverfahren realisieren. Die einzige öffentliche Methode callMinimizer ist dazu da, um von außen den entsprechenden Minimierer aufzurufen und die Minimierung zu starten. In der Methode function muss die zu minimierende Funktion berechnet werden. Die Rückgabe des berechneten Funktionswertes wird über eine Referenz des Parameters functionValue gemacht, eine Referenz wird aus Performancegründen verwendet. Der Parameter variables vom Typ vector, soll die entsprechenden Variablen beinhalten. Zu diesen Variablen wird dann der Funktionswert berechnet. Wie die Berechnung vor sich geht und was genau berechnet wird, ist den einzelnen Subklassen überlassen, diese müssen sich nur an die Interface Spezifikation halten.

Die Methode functionDerivative soll den Gradienten des Funktionswertes abgeleitet nach den Variablen berechnen. Der Gradient wird über die Referenz auf den Parameter derivatives zurückgegeben. Zusätzlich soll es möglich sein, Nebenbedingungen für die Minimierung vorzugeben. Dies wird über die Methode constraintFunction umgesetzt. Die Nebenbedingung muss so formuliert werden, dass diese bei einem Wert größer oder gleich Null eingehalten wird und unter Null nicht eingehalten wird. Zudem muss der Gradient der einzelnen Nebenbedingungsfunktionen bereitgestellt werden und zwar über die Methode constraintFunctionDerivative.

Um die Konvergenz des Verfahrens festzustellen, wird die Methode convergenceCheck verwendet. Der Methode müssen drei Parameter übergeben werden, der erste Parameter ist ein vector mit den Funktionswerten der bisher durchgeführten Iterationen. Der zweite Parameter ist ein mehrdimensionaler vector, welcher alle Variablenwerte der bisherigen Iterationen beinhaltet und der letzte Parameter der Methode ist die Nummer der aktuellen Iteration. Das entsprechende Konvergenzkriterium muss dann innerhalb der Funktion umgesetzt werden. Beispielsweise könnte man das Verfahren als konvergiert ansehen, wenn die Funktions- und Parameterwerte sich seit einigen Iterationen nicht mehr verändert haben oder die Veränderung unterhalb einer bestimmten Schwelle liegt.

Die Methode saveFunction stellt eine Art von Rettungsfunktion dar. Diese soll aufgerufen werden, wenn das Verfahren in irgendeiner Form numerisch instabil wird. Beispielsweise könnte eine einfache Maßnahme sein, die Funktionswerte zufällig zu verändern, in der Hoffnung auf ein anderes (globaleres) Minimum zu treffen.

Die jeweiligen Subklassen implementieren in diesem Fall noch ein Attribut vom Typ DensityFunction, da diese die LikeliHood Funktion minimieren sollen und die Klasse DensityFunction alle nötigen Methoden für die Berechnung dieser liefert, siehe Kapitel 4.

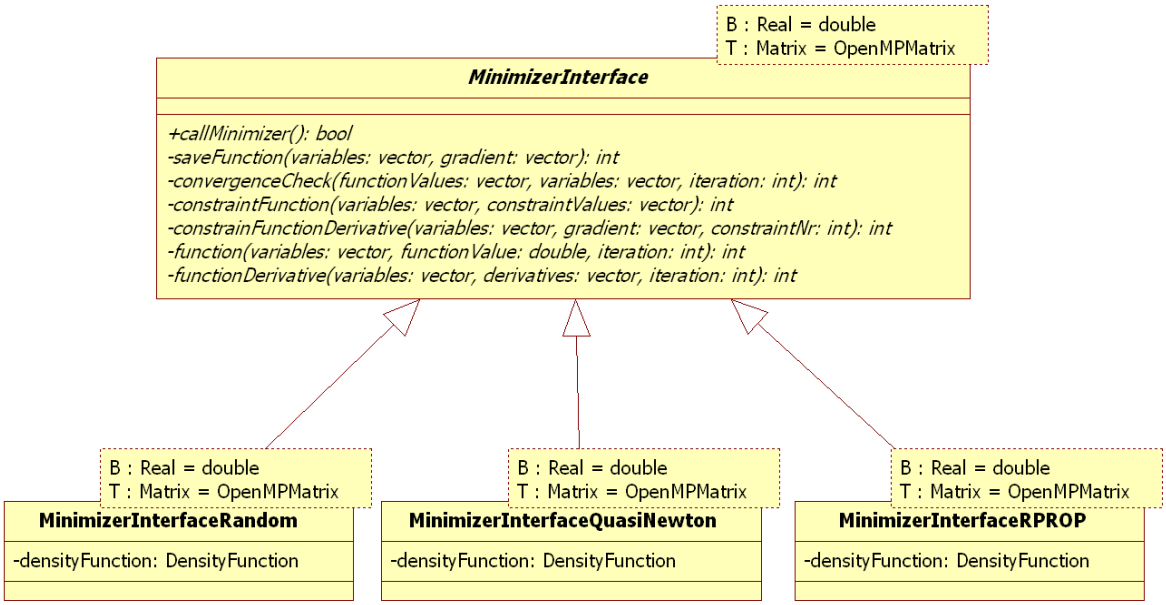


Abbildung 5.2: UML Diagramm der Klassenstruktur der Minimierungsalgorithmen

Das folgende Listing zeigt die Umsetzung der Methode function in der Subklasse MinimizerInterfaceRPROP. Die Methode ist in diesem Fall vereinfacht dargstellt, einige Ausgabefunktionen wurden aus Platzgründen entfernt. Die wichtigen Teile der Methode sind allerdings unverändert. Der Methodenkopf entspricht dem aus dem UML Diagramm. In Zeile 5 werden die aktuellen Variablen (in diesem Fall die Hyperparameter) dem Objekt densityFunction zu Berechnung der Likelihood Funktion übergeben. In den Zeilen 7-18 wird dann die eigentliche Likelihood Funktion innerhalb eines try-catch Blocks berechnet, um eventuelle Exceptions fangen zu können. Wird eine Exception geworfen, so wird eine Fehlermeldung ausgegeben und eine -1 als zurückgegeben. Zusätzlich kann wie in Zeile 15 die entsprechende Korrelationsmatrix bei Auftreten einer Exceptions ausgegeben werden, in diesem Fall wird die Matrix in eine Datei geschrieben.

```
1  int MinimizerInterfaceRPROP<T,B>::function( vector<double> &hyperparameter,
2                                              double &likelihood,
3                                              size_t *iteration){
4
5      densityFunction->setAllHyperparameter(hyperparameter);
6
7      try{
8          likelihood = densityFunction->calcDensity();
9      }
10     catch(InvCholNotIdentity &e){
11         cout <<"RPROP Func (InvCholNotIdentity):"<<endl;
12         return -1;
13     }
14     catch(ChodecNotPosDef &Exception){
15         densityFunction->getCorrelationMatrixRef().saveAsAscii("CorrMatFailed_sav");
16         cout <<"RPROP Func (ChodecNotPosDef):"<<endl;
17         return -1;
18     }
19
20 }
```

Ein zusätzliches Problem bei der Implementierung der Minimierungsverfahren war, dass diese in einer externen Bibliothek in Form von C Funktionen vorlagen und diese

Funktionen Funktionspointer als Parameter erwarten. Die zu übergebenden Funktionen entsprechen den Funktionen aus der MinimizerInterface Klasse, also z.B. function oder functionDerivative.

```
quasiNewton(nrHyperparam,
            nrConstraints,
            variables,
            function(),
            functionDerivatives(),
            constraintFunction(),
            constrainFunctionDerivative(),
            convergenceCheck())

rprop(      nrHyperparam,
            nrConstraints,
            variables,
            variablesLowerLimit,
            variablesUpperLimit,
            function(),
            functionDerivatives(),
            constraintFunction(),
            constrainFunctionDerivative(),
            convergenceCheck())
```

Die Schwierigkeit ergibt sich in diesem Fall dadurch, dass die Funktionspointer in C++ eine Zuordnung zu dem entsprechenden Objekt zu dem die Funktionen gehören, benötigen. Das folgende Listing soll das Problem verdeutlichen:

```
1 MinimizerInterfaceRPROP testObject;
2 int (MinimizerInterfaceRPROP::*ptr2)(vector<double> &, double &, size_t *) =
3                                          &MinimizerInterfaceRPROP::function;
4 (testObject.*ptr2)(variables, functionValue, iteration);
```

In Zeile 1 wird ein Test Objekt vom Typ MinimizerInterfaceRPROP erzeugt (der Konstruktoraufruf wurde hier absichtlich vereinfacht). In den Zeilen 2-3 wird ein Funktionspointer namens ptr2 erzeugt, dieser zeigt auf eine Methode der Klasse MinimizerInterfaceRPROP mit den entprechenden Parametern der Methode function. Zusätzlich erfolgt in diesen Zeilen eine Zuweisung des Pointers der Methode durch *"&MinimizerInterfaceRPROP::function"*.

In Zeile 4 wird ein beispielhafter Aufruf des Funktionspointers auf dem Objekt testObject gemacht. Dieses Beispiel würde so funktionieren. Das eigentliche Problem besteht aber nun darin, dass man in Zeile 2 statt der Subklasse MinimizerInterfaceRPROP die abstrakte Klasse MinimizerInterface verwenden möchte. Da diese Funktionspointer Parameter einer Funktion darstellen, wäre dieses Verhalten sehr wichtig, weil so alle Methodenpointer der Subtypen von MinimizerInterface angenommen werden würden. Andernfalls müsste man die Funktion rprop oder quasiNewton für jeden Subtypen von MinimizerInterface neu implementieren. Leider sind die Möglichkeiten polymorpher Programmierung in C++ stark begrenzt und solch ein Konstrukt wird von der Sprache nicht unterstützt.

Um dieses Problem zu umgehen, werden Funktionsobjekte [Dou04] der Boost Bibliothek verwendet. Mit dieser Bibliothek ist es möglich, die entsprechenden Methoden als Objekt an die entsprechenden externen Funktionen (z.B. rprop und quasiNewton) zu übergeben. Das folgende Listing soll die prinzipielle Funktionsweise von Boost Funktionsobjekten erklären:

```
1  class X {
2  public:
3   int foo(int);
4  };
5
6  boost::function<int (X*, int)> f;
7
8  X x;
9
10 f = &X::foo;
11 f(&x, 10);
```

In diesem Beispiel soll ein Funktionsobjekt der Methode foo der Klasse X erzeugt werden. Zu diesem Zweck wird ein Funktionsobjekt in Zeile 6 initialisiert, wobei innerhalb der eckigen Klammern zuerst der Rückgabewert int und danach die Parameter der Funktion (X*, int) übergeben werden. Der Parameter X* muss vorhanden sein, da innerhalb C++ der erste Parameter einer Methode immer das Objekt selbst ist. Im Normalfall wird dies jedoch automatisch umgesetzt und ist daher unsichtbar für den Programmierer. Die Zuweisung der Methode auf das Funktionsobjekt erfolgt dann in Zeile 10. Der Aufruf des Funktionsobjekts erfolgt nach normaler C++ Syntax, siehe Zeile 11.

Für die Klasse MinimizerInterfaceRPROP würde ein solches Funktionsobjekt wie folgt aussehen:

```
boost::function<int (MinimizerInterfaceRPROP*, vector<double> &, double &, size_t *)> fPointer;
```

Das Problem dass das Funktionsobjekt unabhängig vom Subtyp der Klasse MinimizerInterface sein soll, bleibt allerdings bestehen. Um dies nun zu umgehen, kann man boost::bind verwenden. Mit dieser Funktionalität ist es möglich, Parameter von Funktionen zu verändern.

Das nächste Listing soll dies verdeutlichen, es handelt sich hier um ein stark vereinfachtes Beispiel um die grundlegende Funktionalität zu erklären.

```
1  void external_rprop(boost::function<int (vector<double> &, double &, size_t *)> fPointer){...}
2
3  class MinimizerInterfaceRPROP: public MinimizerInterface{
4  public:
5      void callMinimizer(){
6          boost::function<int (vector<double> &, double &, size_t *)> fPointer;
7          fPointer = boost::bind(&MinimizerInterfaceRPROP::function, (*this), _1, _2, _3);
8          external_rprop(fPointer);
9      }
10 private:
11     int function(vector<double> &vars, double &f, size_t *it) {...}
12 }
```

In Zeile 1 ist eine Funktion definiert, welche einer externen Bibliotheksfunktion entspricht, beispielsweise einem externen RPROP Algorithmus. Dieser Algorithmus benötigt nun ein Funktionsobjekt, mit dem er die zu minimierende Funktion berechnen kann. Wie man sehen kann, benötigt das hier definierte Funktionsobjekt als ersten Parameter nicht mehr das aufrufende Objekt selbst.

Analog zum Originalcode wird als nächstes ist die Subklasse MinimizerInterfaceRPROP definiert, welche Subklasse der abstrakten Klasse MinimizerInterface ist. Diese Klasse besitzt nun eine öffentliche Methode namens callMinimizer. Diese Methode soll von irgendeinem Clienten ausgeführt werden können, um den Minimierungsalgorithmus starten.

Der erste Schritt innerhalb der Methode callMinimizer ist die Erzeugung eines boost Funktionsobjekts, auch hier ist der erste Parameter nicht mehr das aufrufende Objekt selbst (also MinimizerInterfaceRPROP*). In Zeile 7, wird nun ein Funktionsobjekt mit boost bind erzeugt. Mit bind ist es Möglich, die Methodenparameter zu verändern. Dies wird dazu verwendet das Funktionsobjekt quasi unabhängig von der aufgerufenen Klasse zu machen. Dem ersten Parameter für bind wird der Funktionspointer übergeben, der zweite Parameter ist das Objekt selbst. Mit boost bind ist es nun möglich, das Objekt einfach standardmäßig über den this Zeiger fest zu binden. Dieser taucht im kreierten Funktionsobjekt nicht mehr auf und man hat die gewünschte Unabhängigkeit erreicht. Die nächsten drei Parameter _1,_2,_3 sind Platzhalter für die später nötigen Parameter des Funktionsobjekts (also vector<double> &, double &, size_t *).

In der nächsten Zeile wird das Funktionsobjekt an die externe Bibliotheksfunktion übergeben und diese kann die Funktion nun nach belieben verwenden.

6 Validierung

In diesem Kapitel soll das entwickelte Kriging Modell validiert werden. Dazu wird zuerst eine in der Optimierung übliche Testfunktion verwendet. Mit dieser Funktion wird zufällig eine Datenbasis generiert, welche Stützstellen und deren Ableitungen enthält. Auf Basis dieser Daten wird ein Kriging Modell erzeugt und mit der Testfunktion verglichen. Zusätzlich soll ein Geschwindigkeitsvergleich zwischen dem alten Kriging Verfahren und dem neuen Kriging Verfahren angestellt werden.

Da das neue Kriging Modell verschiedene Initialisierungsmöglichkeiten und Minimierungsverfahren bietet, sollen diese hier ebenfalls getestet und verglichen werden, insbesondere in Bezug auf Geschwindigkeit und Genauigkeit.

Außerdem konnte im Rahmen einer Projektarbeit eines Studenten der Universität der Bundeswehr München das entwickelte Gradient Enhanced Kriging Verfahren bereits erfolgreich verwendet und validiert werden, weitere Informationen gibt es in [Eif13].

6.1 Validierung

In diesem Abschnitt soll das neue Verfahren anhand einer für Optimierungsverfahren üblichen Testfunktion [ZDT00] getestet werden. Die Testfunktion $f(\vec{x})$ mit der Variable $\vec{x} \in \mathbb{R}^{\mathrm{k}}$ wird wie folgt berechnet:

$$f(\vec{x}) = h(\vec{x}) * g(\vec{x}) \tag{6.1}$$

$$g(\vec{x}) = \left(1 + \frac{9}{k-1}\sum_{i=2}^{k}(x_i - 0.5)^2\right)$$

$$h(\vec{x}) = \left(2 - \sqrt{\frac{x_1}{g}}\right) - \frac{x_1}{g}\sin(10\pi x_1)$$

In Abbildung 6.1 ist die Testfunktion dargestellt, es wurden zwei Variablen $\vec{x} = \left\{\vec{x} \in \mathbb{R}^k | k = 2\right\}$ verwendet. Bei der Funktion handelt es sich um einen Paraboloid mit überlagerter Sinusschwingung.

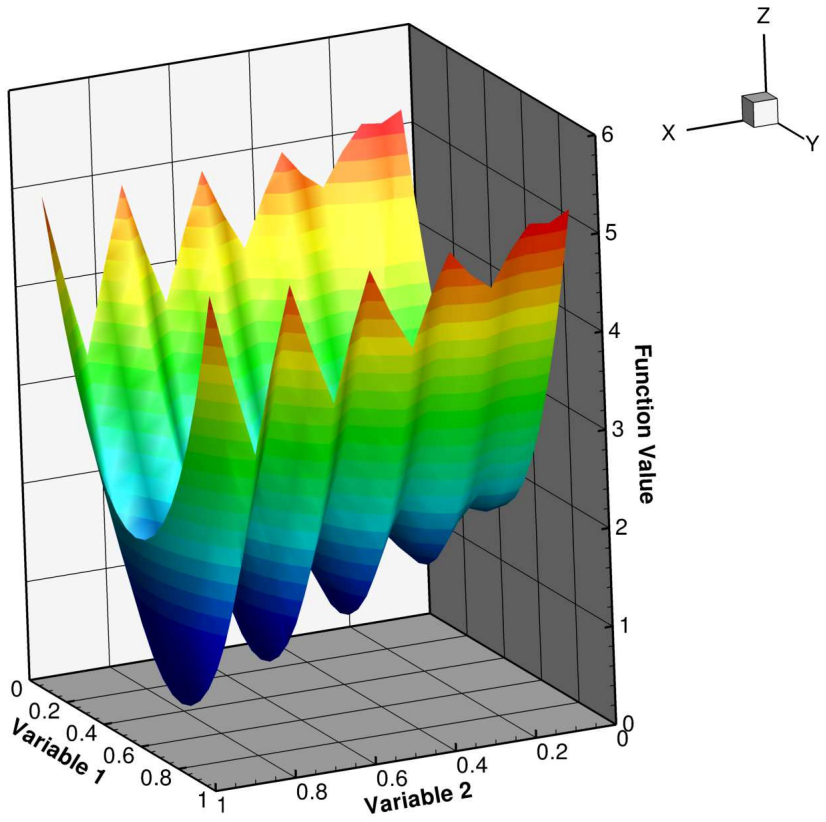
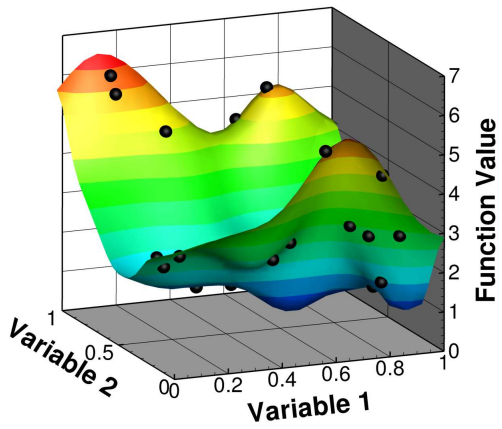


Abbildung 6.1: Plot der Testfunktion mit zwei Variablen

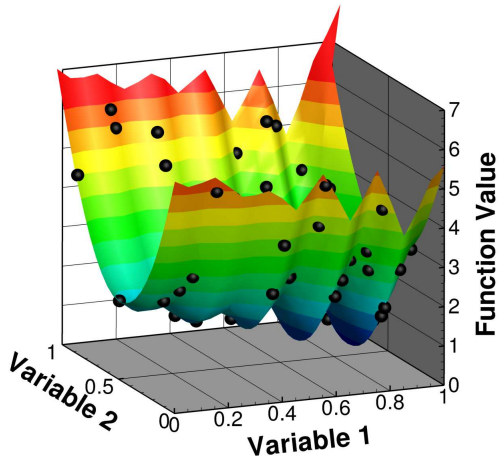
Als erstes soll das neue Kriging Verfahren anhand dieser Testfunktion validiert werden. Hierfür werden zufällig Stützstellen erzeugt, entsprechende Datenbasen generiert und ein Ordinary Kriging Modell damit trainiert. In Abbildung 6.2 sind die Ergebnisse der verschiedenen Testfunktionen gezeigt. Es wurden jeweils verschieden viele Stützstellen für das Training der Modelle verwendet, welche als schwarze Punkte eingezeichnet sind. Die Farben der Fläche stellen den Funktionswert der Testfunktion dar. Bei 25 Stützstellen (Abbildung 6.2a), welches die geringste Anzahl ist, wird die Funktion nur ansatzweise wiedergegeben, die vorhandene Information ist noch nicht ausreichend. Der grundsätzliche Verlauf der Funktion und einzelne Minima können aber erkannt werden. Bei Erhöhung der Anzahl auf 50 Stützstellen wird die Funktion bereits sehr gut wiedergegeben, bei einer weiteren Steigerung verbessert sich das Bild stetig. Das Ergebnis scheint dahingehend plausibel zu sein.

Zusätzlich soll die Annahme überprüft werden, ob das Modell gegen einen konstanten Erwartungswert läuft. Der Erwartungswert sollte β aus Kapitel 2 entsprechen. Um dies zu testen, wurde ein Kriging Modell mit 500 Stützstellen trainiert und das β aus dem Modell ausgelesen und sollte für das hier verwendete Modell $\beta = 9.15$ betragen. Die Funktion wurde wieder mit dem Kriging Verfahren vorhergesagt, allerdings in einem deutlich weiteren Bereich. Dieser Bereich überschreitet die Trainingsdaten und das Verfahren sollte, da es keine anderen Daten in diesem Bereich hat, auf den berechneten Erwartungswert $\beta = 9.15$ extrapolieren.

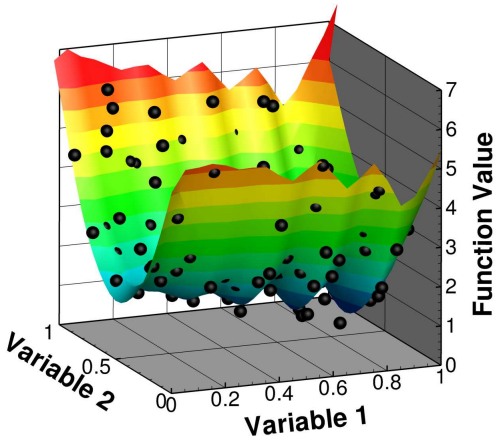
Abbildung 6.3 zeigt das entsprechende Kriging Modell. Der Bereich, in dem Stütz-



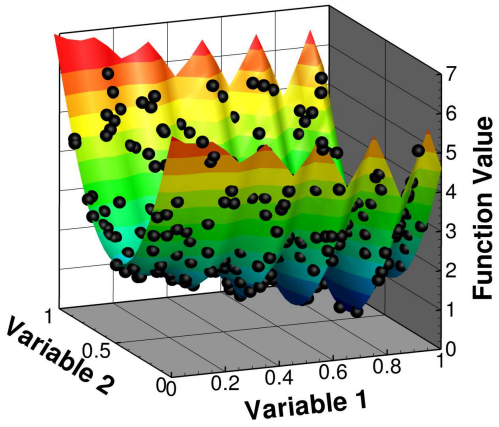
(a) Ordinary Kriging mit 25 Stützstellen



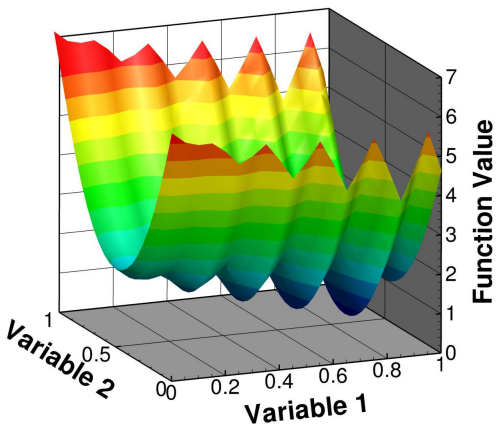
(b) Ordinary Kriging mit 50 Stützstellen



(c) Ordinary Kriging mit 100 Stützstellen



(d) Ordinary Kriging mit 250 Stützstellen



(e) Echte Testfunktion

Abbildung 6.2: Vergleich zwischen den Kriging Modellen der Testfunktion mit verschieden vielen Stützstellen und der realen Funktion. Die Farben entsprechen dem Funktionswert selbst und die schwarzen Punkte stellen die Stützstellen dar.

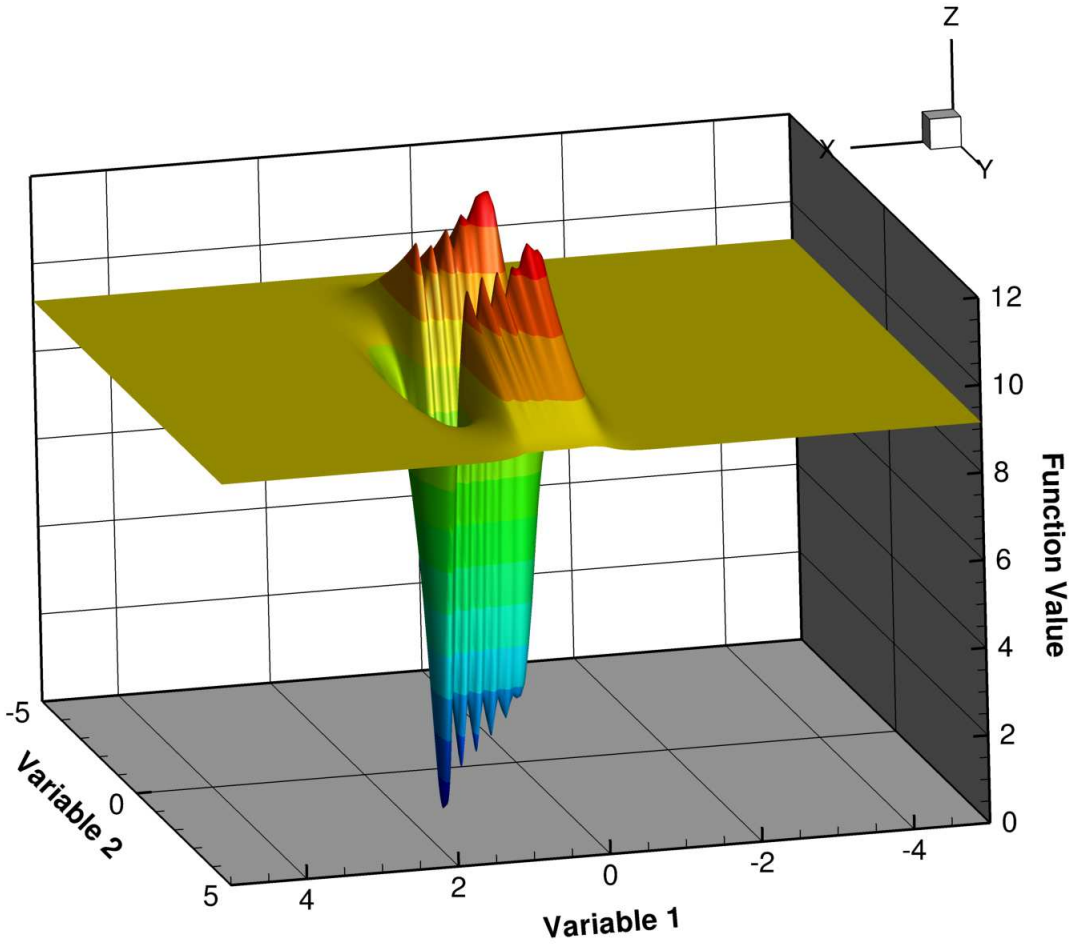
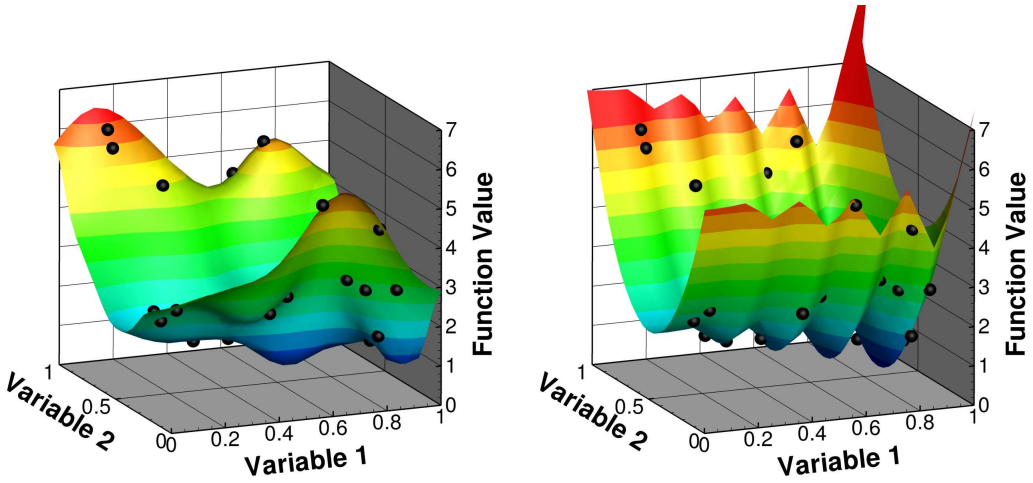


Abbildung 6.3: Extrapolationsverhalten des Ordinary Kriging Modells der Testfunktion.

stellen vorhanden sind, liegt im Intervall $\vec{x} = \left\{\vec{x} \in \mathbb{R}^k | 0 \leq x_{0..k} \leq 1\right\}$. Außerhalb des Bereichs läuft die angezeigte Funktion auf den erwarteten Wert.

Neben dem Ordinary Kriging Verfahren soll das implementierte Gradient Enhanced Verfahren getestet werden. Um dies zu bewerkstelligen, wurde die Testfunktion mit 25 Stützstellen (siehe Abbildung 6.2a) verwendet. Zusätzlich zu den 25 Stützstellen, wurden noch die Gradienten an den Punkten vorgegeben. Die Ableitung der Funktion $f(\vec{x})$ wird hier aus Platzgründen nicht gezeigt. Durch die zusätzliche Gradienteninformation ist zu erwarten, dass die Funktion deutlich besser dargestellt werden kann. In Abbildung 6.4 sind die Ergebnisse der beiden Kriging Modelle dargestellt. Die Stützstellen sind bei beiden Modellen dieselben und wie erwartet, sieht das Gradient Enhanced Kriging mit 25 Stützstellen der originalen Funktion deutlich ähnlicher und der Fehler wurde stark reduziert.

Die Verfahren konnten somit alle durch einen ersten Test erfolgreich validiert werden. Außerdem konnte das Verfahren innerhalb einer externen Projektarbeit erfolgreich validiert werden, siehe [Eif13].



(a) Ordinary Kriging mit 25 Stützstellen

(b) Gradient Enhanced Kriging der Testfunktion mit 25 Stützstellen, an jeder Stützstelle ist zusätzlich ein Gradient vorgegeben.

Abbildung 6.4: Vergleich zwischen Ordinary und Gradient Enhanced Kriging

6.2 Benchmark

Im folgenden Abschnitt soll ein Benchmark des Kriging Modells durchgeführt werden. Es sollen hierbei die Trainingszeiten des alten Codes mit dem neuen Code verglichen werden. Zuerst soll ein Vergleich zwischen den Ordinary Kriging Verfahren und danach zwischen den Gradient Enhanced Verfahren durchgeführt werden. Zusätzlich dazu sollen bei dem neuen Verfahren die verschiedenen Trainingsalgorithmen und Initialisierungen getestet werden.

Ordinary Kriging Benchmark

Es soll zuerst das Ordinary Kriging Verfahren getestet und die Laufzeit für das Training mit dem alten Code verglichen werden. Die zu interpolierende Funktion ist dieselbe wie im Abschnitt vorher (siehe Gleichung 6.1), allerdings wurden für diesen Test 80 Stützstellen zufällig erzeugt und die Stützstellen hatten 11 Variablen $\vec{x} = \left\{\vec{x} \in \mathbb{R}^k | k = 11\right\}$. Daher lässt sich diese grafisch auch nicht mehr darstellen. Es wurden mehr Variablen verwendet, um die Interpolation komplexer und somit zeitaufwendiger zu machen. Dies ermöglicht einen einfacheren Vergleich, da so die reine Rechenzeit im Vergleich zum Overhead (In- und Output usw.) deutlich höher ist. Schwankungen durch bspw. Systemprozesse, welche auf das Dateisystem zugreifen, sollten so verringert werden. Gerechnet wurde der Test auf einem Intel(R) Xeon(R) CPU E5640 mit 8 Kernen bei jeweils 2,67 GHz, wobei nur 6 Kerne verwendet wurden, um dem System noch genügend Rechenleistung übrig zu lassen.

Um den alten Code mit dem neuen vergleichen zu können, wurde dieselbe Initialisierung und dasselbe Minimierungsverfahren (siehe Kapitel 5.3) verwendet. Zusätzlich wurde der Test bei dem neuen Code mit dem QuasiNewton Trainingsverfahren (siehe

Minimierer	Zeit neuer Code	Zeit alter Code
RPROP	2.0 s	3.9 s
QN	0.52 s	-

Tabelle 6.1: Vergleich der Trainingszeit zwischen altem und neuem Code für ein Ordinary Kriging der Testfunktion mit 11 Variablen

Kapitel 5.3) gerechnet. Da der alte Code diese Verfahren nicht unterstützt, konnte hier kein Vergleich angestellt werden. Nach erfolgreichem Training wurden alle Ergebnisse mit der Originalfunktion verglichen und der Fehler zwischen Modell und echter Funktion war grundsätzlich für beide Codes ähnlich. Eine Vergleichbarkeit der Ergebnisse wurde damit sichergestellt.

Tabelle 6.1 zeigt die Ergebnisse. In der ersten Spalte wird die Art des Minimierers angegeben, wobei RPROP und Quasi Newton den Verfahren aus Abschnitt 5.3 entsprechen. Die zweite Spalte stellt die Zeit in Sekunden für ein komplettes Training des neuen Codes dar und die dritte Spalte die Zeit, die der alte Code benötigt hat, um dasselbe Ergebnis zu erreichen. Vergleich man nun den alten mit dem neuen Code, so kann man sehen, dass der neue Code in etwas doppelt so schnell war. Verwendet man bei dem neuen Code das Quasi Newton Trainingsverfahren, stellt sich sogar eine 7,5-fache Beschleunigung dar. Beim Ordinary Kriging Verfahren konnte also eine deutliche Beschleunigung erreicht werden. Welche Teile des Codes diese Beschleunigung verursachen, konnte im Nachhinein kaum festgestellt werden, da durch die komplette Neuentwicklung zu viele strukturelle Unterschiede zwischen den Codes bestehen. Einige grundsätzlichen Unterschiede können aber genannt werden:

Bei der Entwicklung des neuen Codes wurde sehr stark auf minimierte Speicherzugriffe und effiziente CPU Befehle zurückgegriffen. Außerdem konnten viele zeitkritische Strukturen und Operationen gekapselt werden. Dadurch mussten diese nur an einer Stelle im Code verändert bzw. optimiert werden. Im alten Code wurden mehr Code Redundanzen festgestellt, einige redundante Teile waren weniger stark optimiert als andere. Die Thread Parallelisierung wurde im neuen Code grundsätzlich in der Matrix Klasse umgesetzt. Innerhalb des alten Codes wurde die Parallelisierung an anderen Stellen vorgenommen. Möglicherweise konnten dadurch die vorhandenen Kerne besser genutzt werden.

Gradient Enhanced Kriging Benchmark

In diesem Abschnitt soll ein Benchmark des Gradient Enhanced Kriging Verfahrens erstellt werden. Getestet wird dieselbe Funktion wie im Ordinary Kriging Benchmark, allerdings mit dem Unterschied, dass an jeder Stützstelle der Gradient als Zusatzinformation enthalten ist. Tabelle 6.2 zeigt die Ergebnisse der Rechnungen. Dargestellt ist der Vergleich der Dauer für das Training zwischen altem Code und neuem Code. Grundsätzlich ist die Dauer für das Training deutlich höher mit Gradienten als ohne, dies kommt hauptsächlich durch die größere Korrelationsmatrix.

Code	Time
Alter Code	91.99 s
Neuer Code	50.82 s

Tabelle 6.2: Vergleich Gradient Enhanced Kriging zwischen altem und neuem Code

Der neue Code benötigt ca. die Hälfte der Zeit für das Training bei selber Initialisierung und demselben Trainingsverfahren.

Folgend sollen die verschiedenen Minimierungsverfahren und Initialisierungen miteinander verglichen werden. In Tabelle 6.3 sind die Ergebnisse der Messung.

Minimierer	Initialisierung	Stützstellen Init.	Iterationen Init.	Zeit	Zeit Init.
Alter Code	Const			91.99 s	-
QN	Const			19.48 s	0.091 s
QN	Random	10	400	18.64 s	1.15 s
QN	RPROP	15	200	16.18 s	0.19 s
QN	QN	30	20	16.97 s	1.64 s
RPROP	Const			50.82 s	0.18 s
RPROP	Random	10	400	44.23 s	1.31 s
RPROP	RPROP	15	200	42.25 s	2.02 s
RPROP	QN	30	20	34.94 s	4.25 s

Tabelle 6.3: Verschiedene Trainingszeiten für ein Gradient Enhanced Kriging der Testfunktion mit 11 Variablen und verschiedenen Initialisierungen und Trainingsmethoden. Alle Zeiten wurden mit dem neuen Code gemessen.

Die erste Spalte beschreibt hier das eigentliche Minimierungsverfahren, welches nach der Initialisierung mit voller Anzahl der Stützstellen verwendet wird. Die zweite Spalte beschreibt das Verfahren für die Initialisierung. Const beschreibt das Verfahren aus Kapitel 5.2 und entspricht dem Initialisierungsverfahren, welches im alten Code verwendet wurde. Die anderen Verfahren basieren grundsätzlich auf einer temporären Reduktion der Stützstellen (siehe Kapitel 5.2) mit darauf folgender Minimierung. Random würde die Hyperparameter zufällig setzen, die Stützstellen reduzieren und den entsprechenden Likelihood Wert berechnen. Die Hyperparameter, welche den niedrigsten Likelihood Wert haben, werden als Initialisierung für das eigentliche Minimierungsverfahren verwendet. RPROP und QN würden dann im Initialisierungsschritt eine entsprechende Minimierung mit reduzierter Stützstellenanzahl durchführen.

Die Spalten 3 und 4 geben an, wie viele Stützstellen für die Initialisierung verwendet und wie viele Intitalisierungsiterationen durchgeführt worden sind. In den Spalten 5 und 6 ist dann die gemessene Gesamtzeit für das Training und die Initialisierungszeit angegeben.

Das Ergebnis des alten Codes ist in der ersten Zeile als Referenzwert angegeben. Man sieht sehr schnell, dass grundsätzlich die Rechnungen mit Quasi Newton als Minimierungsverfahren am schnellsten konvergiert sind. Dieses Ergebnis war auch zu erwarten.

Alle hier erwähnten Verfahren boten eine bessere Initialisierung als die konstante Schätzung der Hyperparameter, dies wird in der kürzeren Trainingszeit deutlich.

Die Inititialisierung durch das zufällige Bestimmen der Hyperparameter bei reduzierter Stützstellenanzahl bringt einen leichten Geschwindigkeitsvorteil. Dieser kann aufgrund des Verfahrens aber zufällig variieren. Vorteil des Verfahrens ist, dass die Initialisierung eine globale Suche darstellt und somit unter Umständen andere Minima erreichen kann.

Die Verwendung des Quasi Newton oder RPROP Verfahrens für die Initialisierung brachte bei anschließender Minimierung mit dem Quasi Newton Verfahren nahezu keine Vorteile, aber auch keine Nachteile.

Verwendete man für die anschließende Minimierung allerdings das RPROP Verfahren, so konnten sehr große Unterschiede festgestellt werden. Setzte man vor der eigentlichen Minimierung eine Minimierung mit dem RPROP bei reduzierter Stützstellenzahl an, konnte die benötigte Gesamtzeit um ca. 17 % verkürzt werden. Verwendete man für die Initialisierung das Quasi Newton Verfahren, so konnte die Gesamtzeit um ca. 31 % reduziert werden.

7 Fazit und Ausblick

Ziel dieser Arbeit war es, ein Gradient Enhanced Kriging Modell in einer objektorientierten Programmiersprache umzusetzen. Das Modell sollte für zukünftige Anwendungen möglichst modular aufgebaut sein. Besonders wichtig war es, die Korrelationsfunktionen, welche für die Aufstellung der Korrelationsmatrix verwendet werden, austauschbar zu machen. Theoretisch sollte es möglich sein, für jeden Eintrag in der Korrelationsmatrix eine eigene Korrelationsfunktion nutzen zu können. Dieses Ziel konnte mithilfe von polymorpher Programmierung erreicht werden.

Zusätzlich sollte es möglich sein, für das Training verschiedene Minimierungsalgorithmen verwenden zu können. Dies wurde ebenfalls erreicht und es wurde ein zusätzliches Minimierungsverfahren höherer Ordnung verwendet (Quasi Newton), welches die Trainingszeit stark reduziert. Bei dieser Art von Minimierungsalgorithmus musste sichergestellt werden, dass die Hyperparameter nicht negativ werden können. Eine einfache Begrenzung während des Trainings war hier nicht ausreichend, daher wurde die verwendete Korrelationsfunktion so umformuliert, dass negative Hyperparameter ohne Probleme verwendet werden konnten.

Die Initialisierung der Hyperparameter für das Training des Modells war ebenfalls ein wichtiger Punkt. Es wurden hier verschiedene Möglichkeiten implementiert und ein Benchmark für die verschiedenen Initialisierungen durchgeführt.

Der Code wurde grundsätzlich geschwindigkeitsoptimiert. Es wurden die Speicherzugriffe und CPU Befehle minimiert. Der Code ist komplett Thread parallelisiert und bietet Schnittstellen zu einer späteren prozessweiten Parallelisierung. Um die Möglichkeiten moderner CPUs zu nutzen, wurden zudem SSE Befehle verwendet, was einen Geschwindigkeitszuwachs von ca. 20 % brachte. Die Optimierungen wurden am Ende auch in der Gesamtheit mit einem alten Kriging Code verglichen und brachten eine ungefähr doppelt so hohe Geschwindigkeit bei gleichen Einstellungen. Wählte man zusätzlich noch einen Minimierungsalgorithmus höherer Ordnung lag der Geschwindigkeitszuwachs ungefähr bei einem Faktor von 6.

Im Rahmen einer Projektarbeit [Eif13] eines Studenten der Universität der Bundeswehr München wurde das hier entwickelte Gradient Enhanced Kriging Verfahren bereits erfolgreich verwendet und validiert.

Zukünftig soll das Co-Kriging Verfahren implementiert werden. Dieses ermöglicht es Trainingsdaten verschiedener Güte in einem Modell zu verarbeiten. Außerdem soll das Modell in den automatisierten Optimierer AutoOpti eingebaut und dann innerhalb zukünftiger Optimierungen genutzt werden.

Die Matrix Klasse, welche hier verwendet und erweitert wurde, soll um eine Subklasse zur Berechnung einiger wichtiger Matrix Operationen auf einer GPU (Graphics Processing Unit) erweitert werden. Durch Portierung einzelner Operationen auf die GPU erhofft man sich eine deutliche Beschleunigung der Rechenzeit.

Literaturverzeichnis

[Aul12] AULICH, Marcel: Erstellung und Verwendung von Ersatzmodellen. In: *Internes Paper* (2012)

[Brä04] BRÄUNLING, W.J.: *Flugzeugtriebwerke: Grundlagen, Aero-Thermodynamik, Kreisprozesse, Thermische Turbomaschinen, Komponenten- und Emissionen*. Springer, 2004 (VDI-Buch Series). – ISBN 9783540405894

[BS08] BRONSTEJN, I.N. ; SEMENDJAJEW, K.A.: *Taschenbuch der Mathematik*. Harri Deutsch, 2008. – ISBN 9783817120079

[Cum04] CUMPSTY, N.: *Compressor Aerodynamics*. Krieger Publishing Company, 2004

[D+09] DELFINER, Pierre u. a.: *Geostatistics: modeling spatial uncertainty*. Bd. 497. Wiley-Interscience, 2009

[Dou04] DOUGLAS, Gregor: *Boost Function*. `http://www.boost.org/`. Version: July 2004

[Eif13] EIFINGER, P.: Validierung eines Gradient Enhanced Kriging Verfahrens anhand einer Parameterstudie am Profilschnitt eines Fans. In: *Projektarbeit* (2013)

[Fog13] FOG, Agner: *Software optimization resources*. `http://www.agner.org/`. Version: 2013

[FSK07] FORRESTER, Alexander I. ; SÓBESTER, András ; KEANE, Andy J.: Multi-fidelity optimization via surrogate modelling. In: *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Science* 463 (2007), Nr. 2088, S. 3251–3269

[Gil07] GILL, Phil: Numerical linear algebra and optimization. (2007)

[GMW81] GILL, Philip E. ; MURRAY, Walter ; WRIGHT, Margaret H.: Practical optimization. (1981)

[HGZ09] HAN, Zhong-Hua ; GÖRTZ, Stefan ; ZIMMERMANN, Ralf: On improving efficiency and accuracy of variable-fidelity surrogate modeling in aero-data for loads context. In: *Proceeings of European Air and Space Conference*, 2009

[HS11] HELBIG, H. ; SCHERER, A.: Neuronale Netze. In: *Vorlesungsskript* (2011)

[K.A11] K.A.: *Processors - Define SSE2,SSE3 and SSE4*. http://www.intel.com/support/processors/sb/CS-030123.htm. Version: 10 2011

[Kri53] KRIGE, Daniel G.: A statistical approach to some basic mine valuation problems on the witwatersrand. (1953)

[Krü12] KRÜGER, Franziska: *Entwicklung von parallelisierbaren Gradienten-basierten Verfahren zur automatisierten, Ersatzmodell-gestützten Optimierung unter Nebenbedinungen für CFD-FEM-Verdichterdesign*, Diplomarbeit, 2012

[Lon08] LONGLEYS, Dr. J.: *Cambridge Turbomachinary Course Vol. 1*. 2008

[Mac91] MACKKAY, D.J.C.: *Bayesian Methods for Adaptive Models*, Citeseer, Diss., 1991

[Mat63] MATHERON, Georges: Principles of geostatistics. In: *Economic geology* 58 (1963), Nr. 8, S. 1246–1266

[N.A11] N.A.: *Flightpath 2050 Europes Vision for Aviation*. http://www.acare4europe.com/. Version: 2011

[Pla50] PLACKETT, Ronald L.: Some theorems in least squares. In: *Biometrika* 37 (1950), Nr. 1/2, S. 149–157

[Pre07] PRESS, William H.: *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007

[RB93] RIEDMILLER, Martin ; BRAUN, Heinrich: A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In: *Neural Networks, 1993., IEEE International Conference on* IEEE, 1993, S. 586–591

[Rec94] RECHENBERG, Ingo: Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. frommann-holzbog, Stuttgart, 1973. In: *Step-Size Adaptation Based on Non-Local Use of Selection Information. In Parallel Problem Solving from Nature (PPSN3)* (1994)

[SFK12] STEIMANN, Prof. ; FRENKEL, M. ; KELLER, D.: Moderne Programmiertechniken und Methoden. (2012)

[She68] SHEPARD, Donald: A two-dimensional interpolation function for irregularly-spaced data. In: *Proceedings of the 1968 23rd ACM national conference* ACM, 1968, S. 517–524

[Tho06] THORNBURG, Harvey: Autoregressive Modeling: Elementary Least-Squares Methods. In: *Center for Computer Research in Music and Acoustics (CCRMA) Department of Music, Stanford University Stanford, California 94305* (2006)

[Voß08] VOSS, C.: *Automatische Optimierung von Verdichterschaufeln*. Abschlussbericht zum AG-Turbo COOREFF-T Teilvorhaben 1.1.1 des Verbundprojektes CO2-Reduktion durch Effizienz, 2008. – Förderkennzeichen 0327713 B

[Voß10] VOSS, C.: Multi-objective automated compressor optimization using a combines CFD-FEM process chain. In: *Eccomas* (2010)

[ZDT00] ZITZLER, Eckart ; DEB, Kalyanmoy ; THIELE, Lothar: Comparison of multiobjective evolutionary algorithms: Empirical results. In: *Evolutionary computation* 8 (2000), Nr. 2, S. 173–195