UNIVERSITÄT
BAYREUTH

FAKULTÄT FÜR MATHEMATIK, PHYSIK UND INFORMATIK
MATHEMATISCHES INSTITUT

Adaptive Interpolation in der Risikobewertung mittels Stresstests

Diplomarbeit

von

Kathrin Maul

Datum: 30. Mai 2007

Themenstellung / Betreuung:
Prof. Dr. Lars Grüne

Externe Betreuung:
Jens Zahn (BayernLB)

Diese Arbeit wurde im Rahmen einer Projektarbeit im Risk Office der BayernLB, Abteilung Risikocontrolling Handelsbereiche, Team Methoden/Systeme (6531), erstellt.

Danksagung

An dieser Stelle möchte ich allen, die zum Gelingen dieser Diplomarbeit beigetragen haben, herzlich danken. Zunächst danke ich Herrn Prof. Dr. Lars Grüne für das Interesse an der Thematik, die hervorragende Betreuung sowie seine wertvollen Anregungen zur Umsetzung der Diplomarbeit.

Bedanken möchte ich mich außerdem bei den Mitarbeitern der BayernLB, allen voran bei den Herren Kai-Uwe Radde, Jens Zahn und Jürgen Menden: Herrn Radde für die Möglichkeit, meine Diplomarbeit in der BayernLB zu schreiben, und für viele interessante Diskussionen im Vorfeld, Herrn Zahn für seine stete Unterstützung bei der praktischen Umsetzung und sein Engagement als Betreuer der Arbeit und nicht zuletzt Herrn Menden für die zahlreichen Hilfestellungen bei allen in der Implementierung aufgetretenen Problemen.

Meinen Kommilitonen Jasmin Schierding und Karsten Hackler möchte ich nicht nur für das Korrekturlesen dieser Arbeit danken, sondern vor allem auch für die gemeinsame Zeit während unseres Studiums. Wann immer es nötig war, standen sie mir mit Rat und Tat – fachlich wie privat – zur Seite.

Mein besonderer Dank gilt auch meiner Familie und meinem Freund Christian Weitkuhn für die Unterstützung während meines gesamten Studiums.

Abstract

Abstract
This thesis presents an adaptive interpolation approach for the approximation of the portfolio function in a market risk environment. Using the space of multilinear functions for the approximation of the portfolio function local error estimates for the approximation error can be defined. Based on these estimates different refinement strategies for the generation of adaptive grids are developed. The approximate portfolio value results from multilinear interpolation. Implementational details are discussed and results are reported for a set of practical examples.

Keywords
stresstesting, multilinear interpolation, local error estimates, adaptive grid generation

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

„*Die Märkte können länger irrational bleiben, als du solvent.*“
John Maynard Keynes (1883 - 1946)
britischer Ökonom und Mathematiker

Mit der zunehmenden Vielfalt der gehandelten Finanzinstrumente und der Globalisierung der Kapitalmärkte eröffneten sich in den letzten Jahren neue Chancen für die Finanzinstitute. Um nachhaltig am Kapitalmarkt erfolgreich zu sein, ist jedoch die effiziente Messung und die zielgerichtete Steuerung der damit einhergehenden Risiken eine wesentliche Voraussetzung. Die Aufgabe des Stresstestings besteht in diesem Zusammenhang darin, die Auswirkungen von extremen Marktbewegungen auf das Gesamtportfolio eines Finanzinstituts zu analysieren und gegebenenfalls geeignete Gegenmaßnahmen einzuleiten. So wird die Risikotragfähigkeit des Portfolios zu jedem Zeitpunkt sichergestellt und auch schwerwiegende Marktkrisen werden dadurch ohne existentielle Probleme bewältigt. Bisher werden Stresstests hauptsächlich auf zwei verschiedene Arten durchgeführt. Bei der Szenario-Analyse wird das zu untersuchende Portfolio auf Basis eines historischen oder hypothetischen Krisenszenarios neu bewertet, während bei der Worst-Case-Analyse das Szenario mit den schlimmsten Folgen gesucht wird. Beide Ansätze haben jedoch den Nachteil, dass sie die Gewinn- und Verluststruktur des Portfolios nicht vollständig erfassen, weil sie nur bestimmte Szenarien betrachten und zudem keine Auskunft darüber geben, welcher Risikofaktor in welchem Maße für den auftretenden Wertverlust verantwortlich ist.

Genau an diesem Punkt setzt diese Arbeit an. Es wird ein adaptives Verfahren zur Risikobewertung eines Portfolios im Rahmen des Stresstestings entwickelt, das die Gewinn- und Verluststruktur des Portfolios nachbildet. Die Idee dieses Ansatzes besteht darin, die Portfoliofunktion auf einem Gitter durch eine geeignete Funktion, die leicht ausgewertet und effizient gespeichert werden kann, zu approximieren. Jeder Gitterpunkt stellt dabei eine andere Wertkombination der für das Portfolio relevanten Risikofaktoren dar. Im Verlauf des Algorithmus wird iterativ jedes Gitterelement anhand von geeigneten Testpunkten dahingehend überprüft, ob die Approximation bereits gut genug ist oder ob das Element weiter verfeinert werden

muss. Ziel ist es, das Gitter so lange zu verfeinern, bis der Portfoliowert innerhalb jedes Elements mit einer vorgegebenen Genauigkeit durch multilineare Interpolation bestimmt werden kann. Gewählt wurde dieser Ansatz, da damit die Gewinne und Verluste für alle potenziellen Marktbewegungen ermittelt werden können und so die Risikotragfähigkeit des Portfolios im Gegensatz zu den bisher verwendeten Methoden viel umfassender überprüft werden kann.

Stresstesting wird sowohl für das Marktrisiko als auch für das Kreditrisiko durchgeführt. Die hier vorgestellte Methode wurde speziell für Portfolios aus dem Marktrisikobereich entwickelt, kann aber nach Anpassung der Portfoliobewertung auch im Kreditrisikobereich angewandt werden.

Die Arbeit ist in drei Teile gegliedert. Der **erste Teil** befasst sich mit den finanzwirtschaftlichen Grundlagen des Risikocontrollings und der Bewertung eines Portfolios.

Kapitel 2 In Kapitel 2 wird zunächst der Begriff des *Stresstestings* eingeführt und in den Kontext des Risikomanagements eingeordnet. Danach werden kurz die regulatorischen Vorgaben, die ein Finanzinstitut bezüglich des Stresstestings zu beachten hat, erläutert. Zum Abschluss des Kapitels werden verschiedene Umsetzungsmöglichkeiten aufgezeigt und diskutiert.

Kapitel 3 Die in dieser Arbeit betrachtete *Portfoliofunktion* und deren Eigenschaften sind Gegenstand von Kapitel 3. Außerdem wird die weitere Vorgehensweise zur Approximation dieser Funktion erklärt.

Im **zweiten Teil** wird die Theorie zur Approximation der Portfoliofunktion, Fehlerschätzung und adaptiver Gittererzeugung behandelt.

Kapitel 4 Die *Approximation der Portfoliofunktion* wird in Kapitel 4 beschrieben. Da für die Portfoliofunktion im Allgemeinen keine geschlossene Formel existiert, wird sie mit Hilfe des Raums der multilinearen Funktionen auf einem mehrdimensionalen Gitter approximiert. Die grundlegende Eigenschaft dieser Funktionen ist, dass sie sich durch ihre Werte in den Eckpunkten des Gitters eindeutig identifizieren lassen. Der Wert der Portfoliofunktion innerhalb des Gitters kann dann durch multilineare Interpolation bestimmt werden.

Kapitel 5 In Kapitel 5 wird mit Hilfe der *Fehlerschätzung* gezeigt, dass die Diskretisierung mit Hilfe von multilinearen Funktionen tatsächlich eine Approximation der Portfoliofunktion darstellt. Dazu werden verschiedene Arten von Fehlerschätzern vorgestellt, die eine Abschätzung des Approximationsfehlers in jedem Gitterelement erlauben.

Kapitel 6 Basierend auf der Fehlerschätzung aus dem fünften Kapitel wird in Kapitel 6 ein Algorithmus zur *adaptiven Gittererzeugung* hergeleitet, bevor

verschiedene Strategien zur Verfeinerung des Gitters diskutiert werden. Der numerische Vergleich der Strategien erfolgt in Kapitel 8.

Schließlich folgt im **dritten Teil** mit der Implementierung der adaptiven Interpolation und der Diskussion der Ergebnisse der praktische Teil der Arbeit.

Kapitel 7 Die *Implementierung* des besprochenen Ansatzes ist Thema von Kapitel 7. Es werden die grundlegenden Routinen zur Steuerung des Algorithmus, zur Dateneingabe, zur Portfoliobewertung, zur Erzeugung und Verwaltung des Gitters und zur multilinearen Interpolation sowie die verwendete Datenstruktur und die Visualisierung der Ergebnisse beschrieben. Dabei wird auf Aufbau, Funktion sowie die benötigten Ein- und Ausgaben der einzelnen Module eingegangen.

Kapitel 8 *Beispiele und Ergebnisse* werden in Kapitel 8 vorgestellt. Anhand von drei Beispielen aus der Praxis wird das implementierte Verfahren getestet und numerische Ergebnisse ausgewertet. Insbesondere werden die Ergebnisse der in Kapitel 5 und 6 vorgestellten Strategien zur Fehlerschätzung und adaptiven Gitterverfeinerung miteinander verglichen. Das erste Beispiel zeigt die Entstehung des Gitters und der Approximation an einem gemischten Portfolio. Die Abhängigkeit eines Portfolios aus DAX-Optionen von den verschiedenen Risikofaktoren sowie die Unterschiede der verschiedenen Fehlerschätzungs- und Verfeinerungsstrategien veranschaulicht das zweite Beispiel. Im dritten Beispiel wird anhand eines Portfolios aus verschiedenen Optionen das Ergebnis des Algorithmus für verschiedene Fehlertoleranzen untersucht.

Kapitel 9 Zum Abschluss der Arbeit wird in Kapitel 9 anhand der Ergebnisse ein *Fazit* über Vorteile und mögliche Probleme des Verfahrens gezogen sowie ein *Ausblick* auf Weiterentwicklungs- und alternative Anwendungsmöglichkeiten gegeben.

Im **Anhang** finden sich Anmerkungen zur Produktbewertung sowie alle entwickelten Programme.

Anhang A Anhang A erläutert die *Bewertungsmodelle* der in dieser Arbeit verwendeten Produktgruppen Aktien, Zinsprodukte, Optionen und Fremdwährungspositionen.

Anhang B Im Anhang B ist der *C- und C++-Quelltext* des Algorithmus abgedruckt. Ein Teil des Programms basiert auf vorgefertigten Routinen zur Gitterverwaltung von Prof. Dr. Lars Grüne.

Anhang C Der in Anhang C abgedruckte MATLAB-*Quelltext* zur Visualisierung erlaubt unabhängig von der Dimension des Gitters eine dreidimensionale grafische

Darstellung der Portfoliofunktion sowie der Struktur des Gitters. Mit den Programmen wurden die Abbildungen in Kapitel 8 erstellt.

Anhang D In Anhang D wird ein Überblick über den *Inhalt der beiliegenden CD* gegeben.

Kapitel 2

Stresstesting

In diesem Kapitel wird der Begriff des Stresstestings eingeführt und regulatorische Vorgaben sowie deren Umsetzung in der Praxis diskutiert. Die Überlegungen hierzu gehen im Wesentlichen auf [15], [17] und [20] zurück. Aus den Schwächen der vorgestellten Stresstesting-Methoden ergibt sich ein neuer Ansatz, der in den folgenden Kapiteln beschrieben wird.

2.1 Begriffsdefinition und Einordnung ins Risikomanagement

Stresstesting ist neben der Value-at-Risk-Berechnung und dem Backtesting einer der drei Hauptbestandteile des Marktrisikocontrollings. Während der Value-at-Risk (VaR)[1] das Risiko eines Portfolios unter normalen Marktbedingungen abschätzt, versteht man unter Stresstesting die *Analyse der Auswirkungen außergewöhnlicher Marktschwankungen auf ein Portfolio*. Ziel ist es, die Höhe potenzieller Verluste bei krisenhaften Entwicklungen zu bestimmen, um bereits im Vorfeld geeignete Gegenmaßnahmen, wie beispielsweise das Reduzieren oder Hedgen von offenen Positionen, ergreifen zu können.

Es gibt mehrere Gründe, warum Value-at-Risk-Modelle nicht ausreichen, um die Risiken eines Portfolios möglichst vollständig erfassen und kontrollieren zu können. Ein entscheidender Nachteil ist, dass auf Basis des Value-at-Risk keine Aussagen über die potenzielle Höhe der Verluste in extremen Marktbedingungen getroffen werden können, was aber gerade für die Berechnung der Eigenkapitalunterlegung wichtig ist. Des Weiteren sind die Veränderungen von Marktparametern nicht normalverteilt, wie in Modellen zur Berechnung des Value-at-Risk häufig angenommen wird. Die Verteilungen haben vielmehr Fat Tails, d.h. Ereignisse aus den Randbereichen der Wahrscheinlichkeitsverteilung, die mit großen Verlusten verbunden sein können, treten viel häufiger auf, als es die Normalverteilung voraussagt. Andere Modelle, wie

[1] Der Value-at-Risk ist eine auf statistischen Annahmen beruhende Risikokennzahl, die den maximalen Wertverlust eines Portfolios angibt, der innerhalb eines gegebenen Zeitraums t (meist 1 oder 10 Tage) mit einer gegebenen Wahrscheinlichkeit p (meist 95% oder 99%) nicht überschritten wird.

zum Beispiel die Historische Simulation, beruhen zwar nicht auf der Normalverteilungsprämisse, greifen aber zur Berechnung des Value-at-Risk auf historische Zeitreihen zurück, die oft zu kurz[2] sind, um relevante extreme Marktbedingungen überhaupt in einem ausreichenden Maße zu beinhalten. Außerdem basieren einige Modelle zur Berechnung des Value-at-Risk auf gleichbleibenden Korrelationen der Risikofaktoren, die sich aber gerade bei außergewöhnlichen Marktbedingungen sehr stark im Vergleich zu normalen Marktbedingungen ändern können.

Die Beurteilung des Risikos allein auf Grundlage dieser mathematischen Kennzahl ist deshalb unzureichend und muss durch Stresstesting-Methoden ergänzt werden. Im Gegensatz zur Value-at-Risk-Berechnung ist Stresstesting weitgehend unabhängig von statistischen Annahmen. Der Schwerpunkt liegt in der Untersuchung der Randbereiche der Wahrscheinlichkeitsverteilung – den Fat Tails – und deren Konsequenzen. Dadurch liefert Stresstesting eine zum Value-at-Risk komplementäre Betrachtungsweise des Risikos. Es steht deshalb in keinster Weise in Konkurrenz zum Value-at-Risk-Konzept, sondern ergänzt es und ermöglicht im Idealfall ein vertieftes Verständnis und eine bessere Risikosteuerung von komplexen Portfolios.

Abbildung 2.1 zeigt in Anlehnung an [18] die Abgrenzung von Value-at-Risk und Stresstesting anhand der Gewinn- und Verlustverteilung des DJ EURO STOXX 50. Während die tatsächlichen Gewinne und Verluste unter normalen Marktbedingungen durch den auf Normalverteilung basierenden Value-at-Risk-Ansatz gut approximiert werden, unterschätzt die Normalverteilung gerade bei extremen Marktschwankungen die Häufigkeit der eingetretenen Gewinne und Verluste und sollte dort durch Stresstesting-Methoden ergänzt werden, um das Risiko vollständig zu erfassen.

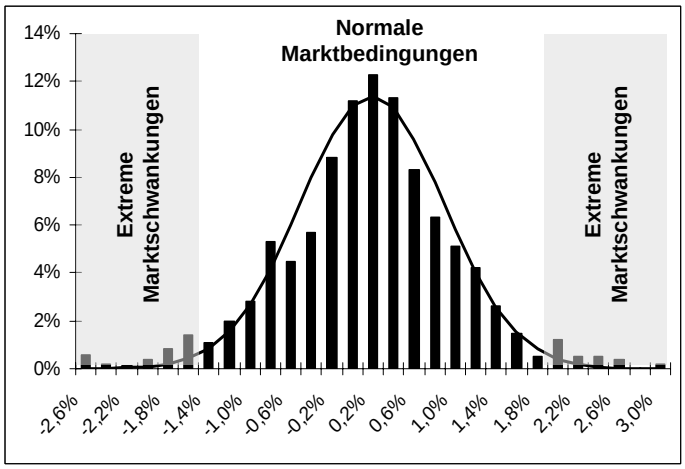

Abbildung 2.1: VaR versus Stresstesting

[2] meist im Bereich von 250 Handelstagen

2.2 Regulatorische Vorgaben

Von aufsichtsrechtlicher Seite werden Stresstests als notwendige Ergänzung und Kontrolle statistischer Modelle, wie etwa des Value-at-Risk, angesehen. Die Anforderung an Banken, Stresstests durchzuführen, stammt in Deutschland aus den Mindestanforderungen an das Risikomanagement [2] und für Banken mit internen Risikomodellen außerdem aus der Solvabilitätsverordnung [3]. Zusätzlich wird die Durchführung von Stresstests im Rahmen von Basel II [4] thematisiert.

Die *Mindestanforderungen an das Risikomanagement* (MaRisk) [2] sind die verbindliche Vorgabe der Bundesanstalt für Finanzdienstleistungsaufsicht (BaFin) für die Ausgestaltung des Risikomanagements in deutschen Kreditinstituten. Im Abschnitt AT 4.3.2 wird die Durchführung von Stresstests gefordert:

> „Die Risikosteuerungs- und controllingprozesse müssen gewährleisten, dass die wesentlichen Risiken frühzeitig erkannt, vollständig erfasst und in angemessener Weise dargestellt werden können. Wechselwirkungen zwischen den unterschiedlichen Risikoarten sollten berücksichtigt werden.
>
> Für die im Rahmen der Risikotragfähigkeit berücksichtigten Risiken sind regelmäßig angemessene Szenariobetrachtungen anzustellen.
>
> Die Geschäftsleitung hat sich in angemessenen Abständen über die Risikosituation und die Ergebnisse der Szenariobetrachtungen berichten zu lassen."

Auf komplexe Detailregelungen oder Festschreibungen wird bewusst verzichtet, um den Finanzinstituten genügend Spielraum für eine risikoadäquate und individuelle Umsetzung zu belassen. So hat sich in Bezug auf die Ausgestaltung noch kein konkreter, international anerkannter Standard etabliert. Wie eine Studie der Bundesbank zu den Methoden und Ergebnissen von Stresstests bei deutschen Banken [5] im Sommer 2004 zeigte, ist die Vielfalt der im Einsatz befindlichen Methoden und Stresstestszenarien sehr groß.

Die Durchführung von Stresstests ist außerdem ein qualitatives Kriterium zur Anwendung eines internen Modells für die Berechnung der Eigenkapitalunterlegung. Für die Genehmigung von internen Risikomodellen in den Banken präzisiert die BaFin ihre Anforderungen an das Stresstesting in § 317 (5) der *Solvabilitätsverordnung* (SolvV) [3]:

> „In dem Umfang und der Struktur des Geschäfts des Instituts angemessenen regelmäßigen zeitlichen Abständen, mindestens jedoch monatlich, sind mögliche außergewöhnlich große Wertverluste der in die modellmäßige Berechnung einbezogenen einzelnen Finanzinstrumente oder Finanzinstrumentsgruppen, die aufgrund von ungewöhnlich großen oder geringen Änderungen der wertbestimmenden Marktparameter und ihrer Zusammenhänge entstehen können, zu ermitteln (Krisenszenarien).
>
> Die Krisenszenarien sollen die Eigenarten der in die modellmäßige Berechnung einbezogenen Finanzinstrumentsgruppen angemessen berücksichtigen und die Zeitdauer widerspiegeln, die zur Absicherung und Steuerung von Risiken benötigt wird.
>
> Die Ermittlung der Wertverluste nach Satz 1 ist sowohl für die Gesamtheit als auch für vom Institut in angemessener Weise festgelegte Klassen von einzelnen Finanzinstrumenten und Finanzinstrumentsgruppen durchzuführen.

> Die Ergebnisse der Krisenszenarien sind der Beurteilung der Angemessenheit der Limite nach Absatz 6 zugrunde zu legen."

Die SolvV trat zum 1. Januar 2007 in Kraft und löste den bis dahin geltenden Grundsatz I über das Mindesteigenkapital der Institute ab. Etliche Passagen wurden direkt aus dem Grundsatz I übernommen, so auch der obige Abschnitt. Die Ausführungen des Grundsatzes I wurden 1997 in einer ersten Erläuterung zum Grundsatz I und 2000 in einer weiteren Erläuterung zum Grundsatz I [1] genauer spezifiziert. In den *Erläuterungen vom 29. Oktober 1997* wird zunächst näher auf die Ausgestaltung der Stresstests eingegangen:

> „Die Durchführung eines Krisenszenarios gliedert sich regelmäßig in die folgenden Schritte:
>
> a) Auswahl der zu berücksichtigenden Finanzinstrumente oder Portfolios,
> b) Festlegung der anzunehmenden Entwicklung der wertbestimmenden Marktparameter (Marktpreise, Volatilitäten, Korrelationen),
> c) Vergleich der ermittelten Wertveränderung mit einem zuvor festgelegten Vergleichsmaßstab.
>
> [...] Das Durchspielen verschiedener Annahmen über die mögliche Entwicklung der wertbestimmenden Marktparameter dient dazu, nicht unbedingt wahrscheinliche, aber wirtschaftlich relevante Konstellationen zu identifizieren, bei denen außergewöhnlich hohe Wertverluste auftreten können. Verlustträchtig können dabei nicht nur ungewöhnlich große Preisänderungen sein; gerade bei Optionsportfolios können auch nur geringe oder keine Preisänderungen oder ein Hin- und Herschwingen der Preise zu Verlusten führen. Auch bei diversifizierten Portfolios sind große Wertverluste nicht unbedingt nur bei extremen Marktpreisveränderungen zu finden. Ferner können bei besonderen Marktereignissen (zum Beispiel Krise eines Währungsregimes) ungewöhnliche Korrelationen zwischen verschiedenen Preisen auftreten. [...]
>
> Bei der Ausgestaltung der Krisenszenarien können verschiedene Ansätze gewählt werden:
>
> - ein eher pauschaler Ansatz besteht in der Analyse der Auswirkungen vorher größenmäßig meist intuitiv definierter Änderungen der Marktpreise (zum Beispiel Parallelverschiebung der Zinskurve um plus/minus 100 Basispunkte),
> - ein auf Erfahrungen früherer extremer Marktpreisänderungen basierender Ansatz spielt Worst-Case-Szenarien auf der Basis von Markt-Crashs durch,
> - ein verfeinerter Ansatz untersucht neben Erfahrungen aus der Vergangenheit auch künftig mögliche Krisen, die bislang noch nicht aufgetreten sind (oder aufgetreten sein konnten wie zum Beispiel Scheitern – oder Gelingen – der Europäischen Wirtschafts- und Währungsunion).
>
> Die genannten Methoden werden überwiegend als generelle Ansätze verwendet, indem auf der Basis zuvor definierter Marktbewegungen die Konsequenzen für das jeweilige institutsspezifische Portfolio analysiert werden. Demgegenüber gehen individuelle Ansätze den umgekehrten Weg, indem sie das jeweils aktuelle Portfolio im Hinblick auf die Frage untersuchen, welche Marktbewegungen zu hohen Verlusten führen können, anschließend diese für das Institut relevanten Marktbewegungen hinsichtlich ihrer Wahrscheinlichkeit oder Unwahrscheinlichkeit prüfen und daraus die Schlüsse für die notwendigen Entscheidungen ziehen."

In den *Erläuterungen zum Grundsatz I vom 20. Juli 2000* wird bezüglich Häufigkeit und Anwendungsebene von Stresstests weiter ausgeführt:

> „Die bisherigen Erfahrungen des Bundesaufsichtsamtes aus Modellprüfungen nach § 32 Grundsatz I haben die Notwendigkeit gezeigt, die Anforderung an die zeitliche Frequenz der Durchführung von Stress Tests zu präzisieren und eine Obergrenze für die zeitlichen Abstände festzulegen. Nach dem neuen Satz 1 sind Stress Tests in einem solchen zeitlichen Abstand durchzuführen, der den Risiken aus dem Umfang und der Struktur des Geschäftes bzw. eines Portfolios des Instituts angemessen ist. Insbesondere komplexe Optionspositionen dürften in aller Regel häufiger als nur einmal monatlich, gegebenenfalls sogar täglich, Stress Tests zu unterziehen sein. [...]
>
> Die bisherigen Erfahrungen des Bundesaufsichtsamtes haben die Notwendigkeit gezeigt, aus Gründen der Normenklarheit und Rechtssicherheit auch im Text des Grundsatzes I die Anforderung, Stress Tests auch auf Subportfolioebene durchzuführen, zu verankern. Dies ist notwendig, da nur bei der Durchführung von Stress Tests auf Subportfolioebene sinnvolle Aussagen über gegebenenfalls bestehende extreme Verlustmöglichkeiten möglich sind, die bei einer aggregierten Betrachtung auf der Ebene des Gesamtportfolios des Institutes verwischt werden oder durch andere, gegenläufige Effekte aus anderen Subportfolios kompensiert werden. Für die Erkennung und Analyse von Risiken und daraus folgend dem bewußten Umgang mit Risikopositionen ist es jedoch erforderlich, sich nicht mit einer saldierten und aggregierten Betrachtung zufriedenzustellen, sondern ins einzelne gehende Untersuchungen anzustellen.“

Der Basler Ausschuss für Bankenaufsicht führt als Begründung für die Notwendigkeit von Stresstests im Abschnitt B.5 der *Änderung der Eigenkapitalvereinbarung zur Einbeziehung der Marktrisiken* [4] folgendes an:

> „Die Banken, die ihre eigenen Modelle für die Berechnung der Eigenkapitalunterlegung für das Marktrisiko verwenden, müssen über ein systematisches und umfassendes Krisentestprogramm verfügen. Krisentests zur Ermittlung von Ereignissen oder Einflüssen, die bedeutende Auswirkungen in der Bank haben könnten, sind für die Banken ein Schlüsselelement zur Einschätzung ihrer Eigenkapitalsituation.
>
> Die Krisenszenarien einer Bank müssen eine Reihe von Faktoren einbeziehen, die zu ausserordentlichen Verlusten oder Gewinnen im Handelsbestand führen können oder die Risikokontrolle in diesem Bestand sehr erschweren. Zu diesen Faktoren gehören Ereignisse von geringer Wahrscheinlichkeit in allen bedeutenden Risikoarten, einschliesslich der verschiedenen Komponenten von Markt-, Kredit- und operationellem Risiko. Krisenszenarien müssen die Auswirkungen solcher Ereignisse auf Positionen beleuchten, die sowohl lineare als auch nicht lineare Preismerkmale aufweisen (d.h. Optionen und optionsähnliche Instrumente).
>
> Die Krisentests der Banken sollten sowohl quantitativer als auch qualitativer Natur sein und sowohl die Marktrisiko- als auch die Liquiditätsaspekte von Marktstörungen erfassen. Die quantitativen Kriterien sollten plausible Krisenszenarien bestimmen, mit denen sich die Banken konfrontiert sehen könnten. Mittels qualitativer Kriterien sind zwei wichtige Ziele der Krisentests hervorzuheben, nämlich abzuschätzen, ob die Eigenmittel einer Bank potenzielle grosse Verluste absorbieren könnten, und Massnahmen zu ermitteln, mit denen die Bank ihr Risiko vermindern und ihr Eigenkapital erhalten kann.“

Es sollen also plausible Krisenszenarien untersucht werden, die zu außerordentlichen Verlusten oder Gewinnen führen können. Weiter fordert der Basler Ausschuss für Bankenaufsicht die Berücksichtigung der Portfoliozusammensetzung bei der Wahl von Krisenszenarien:

> „Die Banken sollten die von der Aufsichtsinstanz festgelegten Krisenszenarien mit von ihnen selbst entwickelten Krisentests, die ihren speziellen Risikomerkmalen Rechnung tragen, kombinieren.“

In Bezug auf die Ausgestaltung der Stresstests werden sowohl historische Krisensimulationen als auch Sensitivitätsanalysen und die Suche nach Worst-Case-Szenarien gefordert:

> „Die Banken sollten ihre Portfolios einer Reihe von Krisensimulationen unterwerfen und die Ergebnisse der Aufsichtsinstanz mitteilen. Beispielsweise könnte das aktuelle Portfolio an früheren Perioden erheblicher Turbulenzen gemessen werden [...].
>
> Bei einer weiteren Art von Szenario würde die Sensitivität des Marktrisikos der Bank gegenüber Änderungen der angenommenen Volatilitäten und Korrelationen bewertet. Hierzu müsste die historische Schwankungsbandbreite der Volatilitäten und Korrelationen ermittelt werden, und die aktuellen Positionen der Bank müssten anhand der Extremwerte dieser Bandbreite bewertet werden. Angemessen zu berücksichtigen wären die heftigen Ausschläge, die bei erheblichen Marktturbulenzen manchmal innerhalb von wenigen Tagen eintraten.
>
> Neben den von den Aufsichtsinstanzen vorgegebenen Szenarien [...] sollte eine Bank auch ihre eigenen Krisentests für Situationen entwickeln, die sie angesichts der Zusammensetzung ihres Portfolios als die schlimmstmöglichen Fälle erachtet [...].“

Typische Kritikpunkte der Bankenaufsicht im Hinblick auf die implementierten Stresstests bei bereits durchgeführten Prüfungen von Finanzinstituten waren laut Reitz [17] die Szenarienauswahl und die Bewertungsmethoden. Die Stresstestszenarien waren oft zu pauschal gewählt und gingen nicht in ausreichendem Maße auf die Portfoliozusammensetzung ein. Außerdem fand oft keine vollständige Neubewertung des Portfolios statt. Gerade bei extremen Veränderungen von Risikofaktoren ist aber eine lineare oder quadratische Approximation von komplexen Finanzinstrumenten über Sensitivitäten, wie sie in Finanzinstituten zur Zeitersparnis gerne verwendet wird, nicht sinnvoll.

2.3 Umsetzung von Stresstests

Auch wenn von den Aufsichtsbehörden keine genauen Vorgaben zur Ausgestaltung von Stresstests gemacht werden, so liegt doch allen Stresstests die gleiche Struktur zugrunde. Sie untersuchen, wie stark sich der Wert eines Portfolios bei einem Schock in den Risikoparametern verändert, und tragen so zu einem verbesserten Verständnis des Risikoprofils bei. Die Umsetzung gliedert sich laut [18] meist in die folgenden drei Schritte:

1. Definition eines Stressszenarios und der entsprechenden Risikofaktoränderungen
2. Neubewertung des Portfolios unter Anwendung des Szenarios
3. Auswertung der Stresstesting-Ergebnisse

Die Schwierigkeit besteht in der Auswahl von plausiblen und für das jeweilige Portfolio relevanten Szenarien und der damit verbundenen Entwicklung der Risikofaktoren. Die Szenarien sollten in ihrer Gesamtheit so definiert sein, dass alle Teilbereiche des Risikoprofils eines Portfolios abgedeckt werden. Hierzu sind zahlreiche verschiedene Stressszenarien notwendig, wobei

jede Szenariosimulation eine Teilinformation über das Risikoprofil enthält. Im nächsten Schritt werden die einzelnen Produkte des Portfolios mit den entsprechend veränderten Risikofaktoren neu bewertet. Aufgrund der großen Veränderungen in den Risikofaktoren ist statt einer linearen oder quadratischen Approximation über Sensitivitäten eine exakte Neubewertung angebracht. Zuletzt werden die in den verschiedenen Szenarien auftretenden Gewinne und Verluste übersichtlich zusammengefasst. Für eine fundierte Analyse ist die Aufschlüsselung des Verlusts auf die einzelnen Risikofaktoren wichtig.

Eine systematische Übersicht über die möglichen Arten von Stresstests ist in Abbildung 2.2 gegeben. Im folgenden Abschnitt werden die üblichen Methoden des Stresstestings, nämlich

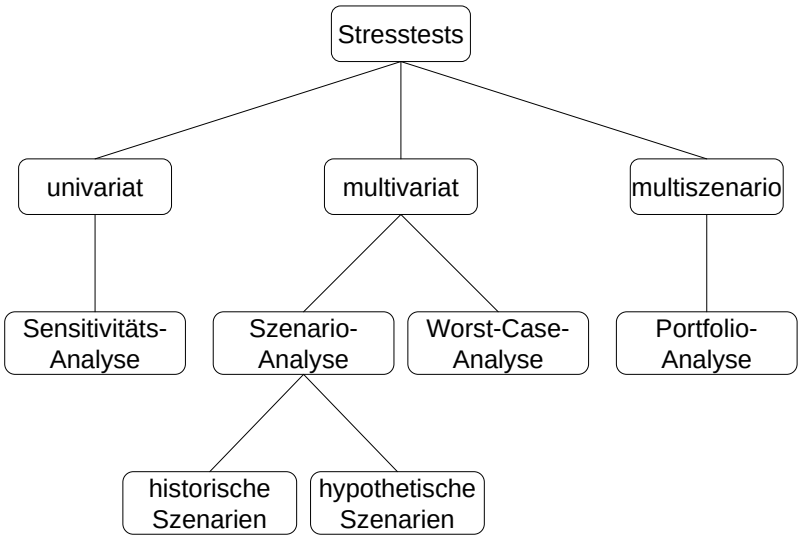

Abbildung 2.2: Stresstesting-Methoden

univariate und multivariate Stresstests, erläutert. Die in dieser Arbeit entwickelte Methode der Portfolioanalyse basiert im Gegensatz zu den anderen Verfahren auf der Auswertung mehrerer Szenarien und wird in den nächsten Kapiteln vorgestellt.

Hinsichtlich der Anzahl der geshifteten Risikofaktoren lässt sich zwischen univariaten und multivariaten Stresstests unterscheiden. *Univariate Stresstests* beruhen auf der Betrachtung eines einzelnen Risikofaktors und beurteilen den Effekt einer Auslenkung dieses Marktparameters auf den Portfoliowert. Sie sind in erster Linie dazu geeignet, Portfolios auf deren Sensitivitäten[3] hin zu untersuchen und werden deshalb auch als *Sensitivitätsanalysen* bezeichnet. Beispiele für Sensitivitätsanalysen wären eine Parallelverschiebung oder Drehung

[3] Die Sensitivität eines Portfolios gegenüber einem Risikofaktor gibt die Veränderung des Portfoliowerts in Geldeinheiten an, wenn sich der Risikofaktor um eine Einheit erhöht. Sie misst also die Stärke, mit der ein Risikofaktor auf ein Portfolio einwirkt.

von Zinskurven, Veränderungen von Aktienkursen oder etwa die Schwankung einer Volatilität. Ihr Vorteil ist, dass sie den spezifischen Einfluss einzelner Risikofaktoren von dem anderer Faktoren isolieren und so die Schwachstellen des Portfolios relativ genau identifizieren können. Problematisch ist dabei allerdings, dass diese Art von Stresstests die Korrelationen der Risikofaktoren untereinander völlig außer Acht lässt. Oft ist aber gerade die Kumulierung verschiedener Parameterveränderungen stabilitätsgefährdend für eine Bank, was bei einer isolierten Betrachtung der Sensitivitäten unter Umständen gar nicht auffallen würde. Da Krisen in der Realität typischerweise als Kombination aus vielen Parameterveränderungen auftreten, kann mit Hilfe von Sensitivitätsanalysen keine Aussage über die Risikotragfähigkeit eines Portfolios gemacht werden.

Realitätsnäher als univariate Stresstests sind die *multivariaten Stresstests*, bei denen mehrere Risikofaktoren simultan variiert werden und die daraus resultierenden Auswirkungen auf ein Portfolio analysiert werden. Sie berücksichtigen die Interaktion verschiedener Risikofaktoren und können dadurch das Risikoprofil auch größerer Portfolios besser erfassen. Es gibt zwei Arten von multivariaten Stresstests: die Szenario-Analysen sowie die Worst-Case-Analysen. Erstere untersuchen auf Basis vorab definierter Marktparameteränderungen den Einfluss wirtschaftlicher Krisen auf ein beliebiges Portfolio und bilden somit die Grundlage für Entscheidungen des Risikomanagements. Letztere prüfen ausgehend von einem bestimmten Portfolio, welche Marktpreisänderungen zu besonders hohen Verlusten führen können. Auf Basis dieser Informationen können dann die Wahrscheinlichkeiten für das Eintreten solcher Marktbewegungen ermittelt und die notwendigen Konsequenzen gezogen werden.

Zunächst soll die *Szenario-Analyse* näher betrachtet werden, bei der komplette Marktcrashs simuliert werden. Um die Auswirkungen außergewöhnlicher Marktereignisse auf ein Portfolio zu berechnen, müssen die Parameter, die dieses außergewöhnliche Ereignis ausmachen, festgelegt werden. Dies kann entweder auf Basis historischer Daten geschehen oder es können hypothetische Krisen, die auf potenziellen zukünftigen Marktentwicklungen beruhen, definiert werden.

Im Fall von *historischen Szenarien* wird ein besonderes Marktereignis, wie beispielsweise die Asienkrise von 1997, ausgewählt. Die Veränderungen der Risikofaktoren während dieses historischen Szenarios werden aus der gezielten Auswertung entsprechender Zeitreihen ermittelt und auf die aktuellen Werte der Risikofaktoren angewendet. Die Shifts sämtlicher Risikofaktoren stammen somit alle aus dem gleichen historischen Zeitraum. Mit den so veränderten Risikofaktoren wird das betrachtete Portfolio neu bewertet.

In vielen Banken wird eine Auswahl der folgenden historischen Krisenszenarien analysiert. In Tabelle 2.1 sind die beobachteten Veränderungen der Risikofaktorgruppen Aktien und Aktienindizes (EQU-Shifts in Prozent), Zinsraten (IR-Shifts in Basispunkten) und Wechselkurse (FX-Shifts in Prozent) für diese Krisenszenarien aufgelistet. Volatilitätsshifts werden in diesen Szenarien nicht betrachtet. Zur besseren Übersicht werden die Risikofaktoren einer Risikofaktorgruppe den Währungsräumen Europa Euroraum (EUR), Europa Rest (NEU), Amerika (USD), Asien (JPY) und Rest (XXX) zugeordnet, für die meist sowieso ein gemeinsamer

Shift festgelegt wird. Ist dies nicht der Fall wird die Schwankung eines Vertreters[4] zusammen mit dem Intervall, in dem die Shifts der restlichen Risikofaktoren liegen, angegeben. Jedoch kann das Ausmaß der Risikofaktorschwankungen je nach Art der Messung von Bank zu Bank stark variieren, so dass die Ergebnisse nicht wirklich vergleichbar sind.

- **Aktiencrash Oktober 1987**

  Der Aktiencrash im Oktober 1987 stellt für fast alle Aktienmärkte den größten relativen Verlust der neueren Börsengeschichte dar.

- **Zinsanstieg Februar 1990**

  Im Februar 1990 stiegen die Zinsen in Deutschland stark an. Die Aktienkurse gaben deutlich nach, während die Wechselkurse annähernd unverändert blieben. Bemerkenswert ist, dass sich die Korrelationen in dieser außergewöhnlichen Marktsituation anders verhalten als beim Aktiencrash 1987. Während 1987 zusammen mit den Aktienkursen auch die Zinsen sanken, stiegen nun die Zinsen, die Aktienkurse gingen jedoch zurück.

- **Golfkrieg 1990**

  Am 3. August 1990 brach durch die Invasion Iraks in Kuwait der Golfkrieg aus. Dies führte zu heftigen Erschütterungen auf den internationalen Finanzmärkten. Auch die Auswirkungen auf Zinsen und Währungen waren deutlich.

- **Asienkrise 1997**

  Auch die Asienkrise von 1997 stellt ein bekanntes Beispiel für eine außergewöhnliche Marktsituation dar. Dies gilt besonders für die sogenannten Tigerstaaten, deren Aktienmärkte einbrachen und deren Währungen deutlich an Wert verloren. Im Gegensatz zu den anderen Szenarien werden hier die Zinskurven für Regierungs- (GOV) und Nicht-Regierungsanleihen (SWAP) getrennt behandelt.

- **11. September 2001**

  Durch dieses Szenario werden erhebliche Kursschwankungen simuliert, die insbesondere Aktien von Versicherungsunternehmen, Banken, Fluggesellschaften und Reiseveranstaltern betrafen.

Bei der Verwendung historischer Szenarien ist darauf zu achten, dass sie im Zusammenhang mit den zu untersuchenden Portfolien sinnvoll verwendet werden können. Sollten sich im Portfolio beispielsweise überwiegend Finanzinstrumente befinden, die zu Zeiten des betrachteten Szenarios noch gar nicht existierten oder die auf ganz spezielle Kombinationen von Risikofaktoren reagieren, so ist die Aussagekraft willkürlich gewählter Szenarien natürlich nur begrenzt.

Zusätzlich zu historischen Szenarien können deshalb auch mögliche zukünftige Ereignisse und ihre voraussichtlichen Auswirkungen auf Aktienkurse, Zinsen, Wechselkurse und Volatilitäten betrachtet werden. Man spricht in diesem Zusammenhang von sogenannten *hypothetischen*

[4] Der Vertreter für Europa Euroraum ist EUR, für Europa Rest GBP, für Amerika USD und für Asien JPY. Für XXX gibt es keinen festen Vertreter.

Krise	**Währungs-raum**	**Auswirkungen auf die Risikofaktoren**		
		EQU-Shifts in %	**IR-Shifts in BP**	**FX-Shifts in %**
Aktiencrash 1987	EUR	-24 [-32; -10]	-30	—
	NEU	-29 [-29; -13]	-100	—
	USD	-24 [-36; -23]	-100	—
	JPY	-15 [-43; -15]	-100	—
	XXX	-22 [-24; -22]	-100	—
Zinsanstieg 1990	EUR	-6 [-6; +5]	[+30; +120]	—
	NEU	-2 [-9; +2]	[-10; +40]	—
	USD	-1 [-2; -1]	[-10; +40]	—
	JPY	-2 [-3; +7]	[-10; +40]	—
	XXX	-5	[-10; +40]	—
Golfkrieg 1990	EUR	-11 [-16; -2]	+20	—
	NEU	-5 [-7; -4]	+20	0
	CHF	-11	+30	-1
	USD	-5 [-7; -1]	+20	+1
	JPY	-16 [-20; -4]	+30	+4
	XXX	-5	+20	+1
Asienkrise 1997	EUR	-7 [-11; 0]	SWAP: -9; GOV: -14	—
	NEU	-6 [-12; -4]	SWAP: 0; GOV: -1	0
	CHF	-5	SWAP: +5; GOV: +5	-2
	USD	-3 [-8; -3]	SWAP: -17; GOV: -18	+3
	JPY	-4 [-12; -3]	SWAP: -4; GOV: -8	+2
	XXX	-9 [-9; -3]	SWAP: 0; GOV: -1	+3
11. September 2001	EUR	-16 [-46; -9]	—	—
	NEU	-13 [-49; -7]	—	-2
	USD	-12 [-70; -7]	—	-4
	JPY	-15 [-23; -9]	—	-4
	XXX	-13	—	-9

Tabelle 2.1: Historische Krisenszenarien

Krisenszenarien oder What-If-Szenarien. Diese bieten sich vor allem dann an, wenn besondere Eigenheiten eines Portfolios zu berücksichtigen sind oder spezielle Kombinationen von Risikofaktoren einem Stresstest unterzogen werden sollen. Im Gegensatz zu historischen Szenarien ist bei hypothetischen Szenarien darauf zu achten, dass die verschiedenen Kombinationen von Risikofaktoränderungen ökonomisch sinnvoll und plausibel sind. Sie sollten deshalb regelmäßig auf ihre Angemessenheit hin überprüft und gegebenenfalls verändert, entfernt oder durch neue Szenarien ergänzt werden. Zur Konstruktion von hypothetischen Szenarien gibt es verschiedene Vorgehensweisen. Oft wird bei der Auswahl der Risikofaktoren und der Größenordnung ihrer Auslenkung auf Expertenmeinung zurückgegriffen. Dabei besteht allerdings die Gefahr, dass sich in der Kombination aller festgelegten Risikofaktoränderungen willkürliche Korrelationsannahmen wiederfinden. Um dieses Problem zu umgehen, kann man auch hypothetische Stressszenarien mit Berücksichtigung empirischer Korrelationen konstruieren, so dass sie sich in Übereinstimmung mit den historisch beobachteten Abhängigkeiten zwischen den Risikofaktoren befinden. Hier lässt sich natürlich einwenden, dass die gewöhnlichen Korrelationen in Krisensituationen keine Gültigkeit mehr besitzen, sondern dass es im Gegenteil zu Korrelationszusammenbrüchen kommen kann. Um diese Schwierigkeit zu beheben, besteht die Möglichkeit, mit Hilfe von Verfahren wie Korrelationsschätzung, Monte-Carlo-Simulation oder Extremwerttheorie spezielle Stresskorrelationen zu schätzen. Weitere Information zu diesen Verfahren finden sich in Reitz [17]. Aber auch derartige Verfahren beruhen immer auf der Verwendung historischer Daten und tragen somit die Problematik in sich, dass künftige Stressszenarien am Markt nicht notwendigerweise so aussehen müssen, wie es in der Vergangenheit beobachtet wurde. Dennoch beziehen sie immerhin die empirisch beobachteten statistischen Eigenschaften der Risikofaktoren, insbesondere Korrelationen und Volatilitäten, mit in die Analyse ein und sind somit als realitätsnäher einzustufen als mehr oder weniger willkürlich festgelegte Szenarien.

Im Folgenden werden zwei mögliche hypothetische Krisenszenarien vorgestellt. Tabelle 2.2 zeigt wiederum die Auswirkungen der Szenarien auf die verschiedenen Risikofaktoren.

- **Euroschwäche**

  Dieses Szenario analysiert, was passiert, wenn der Euro gegenüber allen anderen Währungen an Wert verliert und gleichzeitig alle Zinskurven im Euroraum angehoben werden.

- **Flight-to-Quality**

  Dieses Szenario lehnt sich an eine schwere Krise der Finanzmärkte im Herbst 1998 an. In diesen Zeitraum fällt die Russlandkrise und die Beinahe-Pleite des Hedge Funds LTCM. Durch die allgemeine Unsicherheit während solcher Krisen werden viele Anleger dazu veranlasst, das Kapital in sichere Instrumente umzuschichten. Dadurch kommt es zu einer Aufweitung der Spreads zwischen Regierungs- (GOV) und Nicht-Regierungsanleihen (SWAP) sowie zu einem deutlichen Rückgang der Kurse an den internationalen Börsen.

Bei der Auswahl der Szenarien hat man die Wahl zwischen standardisierten und subjektiven Szenarien. Der Vorteil standardisierter Szenarien ist, dass sie leicht aggregierbar sind

Krise	Währungs-raum	Auswirkungen auf die Risikofaktoren		
		EQU-Shifts in %	**IR-Shifts in BP**	**FX-Shifts in %**
Euroschwäche	EUR	0	+25	—
	NEU	0	0	-5
	USD	0	0	-5
	JPY	0	0	-5
	XXX	0	0	-5
Flight-to-Quality	EUR	-10	SWAP: +50; GOV: 0	—
	NEU	-10	SWAP: +50; GOV: 0	0
	USD	-10	SWAP: +50; GOV: 0	-5
	JPY	-10	SWAP: +50; GOV: 0	0
	XXX	-10	SWAP: +50; GOV: 0	0

Tabelle 2.2: Hypothetische Krisenszenarien

und einen Vergleich der Ergebnisse über verschiedene Portfolios hinweg zulassen. Dem steht allerdings der Nachteil gegenüber, dass sie nicht unbedingt diejenigen Szenarien repräsentieren müssen, gegenüber denen das jeweils aktuelle Portfolio besonders sensitiv ist. Subjektive Stresstests hingegen haben den Vorteil, individuell an die Gegebenheiten des vorliegenden Portfolios angepasst werden zu können. Allerdings geht dies auf Kosten der Objektivität und Vergleichbarkeit.

Die zweite Variante der multivariaten Stresstests sind *Worst-Case-Analysen*. Der Unterschied zur Szenario-Analyse besteht darin, dass nicht alle Marktparameter simultan geändert werden, sondern jeweils nur eine Risikofaktorgruppe. Innerhalb einer solchen Gruppe sind alle Risikofaktoren zusammengefasst, welche die gleiche Risikoklasse (Aktien, Zinsen, Währungen und Volatilitäten) besitzen und aus dem gleichen Währungsraum stammen. Diese Aufteilung der Risikofaktoren ermöglicht eine systematische Untersuchung der Gewinn- und Verlustverteilung und ein besseres Verständnis des Risikos. Für jede Risikofaktorgruppe werden die zu simulierenden Schwankungen aus der Analyse historischer Zeitreihen ermittelt. Ausschlaggebend für die Auswahl ist aber nicht nur die Höhe der Risikofaktorschwankungen, sondern auch die Höhe des damit zusammenhängenden Portfolioverlusts. Dabei kann sich für jede Risikofaktorgruppe die Veränderung aus einem anderen historischen Zeitraum ergeben. Anschließend wird für jede Risikofaktorgruppe der Effekt der ermittelten Schwankung auf den Portfoliowert berechnet. Daraus ergibt sich eine Matrix aus potenziellen Portfoliogewinnen bzw. -verlusten, anhand der der Risikocontroller schnell die Risikofaktorgruppen identifizieren kann, in denen die größten Risiken liegen. Ziel der Worst-Case-Analyse ist es, das Szenario mit den größten Verlusten für das jeweilige Portfolio zu finden. Durch Szenario-Analyse allein ist das nicht möglich, da Krisenszenarien auf extremen Schwankungen in den Risikofaktoren basieren, die jedoch nicht zwangsläufig zu den höchsten Ausschlägen in der Portfoliofunktion führen müssen. Es kann durchaus Portfolios[5] geben, die auf geringe Marktschwankungen mit viel höheren Ausschlägen reagieren als auf große.

[5] Zum Beispiel Portfolios, die auf einer Straddle-Strategie basieren. Dabei wird auf extreme Marktschwankungen spekuliert und die größten Verluste entstehen, wenn sich der Markt überhaupt nicht verändert.

2.4 Bewertung der Stresstesting-Methoden

Stresstests sind in der Lage, einige der entscheidenden Defizite von Value-at-Risk-Methoden zu kompensieren. Sie beruhen weder auf statistischen Annahmen, wie zum Beispiel der Normalverteilung, noch auf zu kurzen Zeitreihen und sind daher geeignet, Verluste auch in sehr unwahrscheinlichen Marktsituationen abzuschätzen.

Nichtsdestotrotz haben die oben genannten Stresstest-Methoden aber auch ihre Nachteile. Ein Nachteil betrifft die Szenarienauswahl. Zum einen können Stresstests – egal, ob als Sensitivitäts-, Szenario- oder Worst-Case-Analyse – fatale Krisenszenarien übersehen, da sie immer nur eine bestimmte Anzahl von Szenarien analysieren, den möglichen Verlust also nur in wenigen Punkten des hochdimensionalen Raums der Szenarien bestimmen. Man kann daher nicht mit Sicherheit wissen, ob man tatsächlich die schlimmsten Szenarien gefunden hat. Zum anderen bleibt die Auswahl der Szenarien den Finanzinstituten weitestgehend selbst überlassen. Während es bei historischen Krisenszenarien noch relativ einfach ist, die Veränderungen der Risikofaktoren zu bestimmen, kommt es bei hypothetischen Krisenszenarien darauf an, möglichst plausible und für das Portfolio relevante Marktschwankungen zu schätzen – was auch immer im Ermessen des Risikocontrollers liegt. Aussagen über die Wahrscheinlichkeit, mit der ein bestimmtes Krisenszenario eintrifft, liefert der Stresstest nicht. Man kann deshalb weder die Vollständigkeit noch die Plausibilität des Stresstestings garantieren. Ein weiterer Nachteil ist, dass Szenario-Analysen, die den Wert mehrerer Risikofaktoren gleichzeitig verändern, keine Auskunft darüber geben, welcher Risikofaktor in welchem Maße für den auftretenden Portfolioverlust verantwortlich ist. Gegenmaßnahmen, wie ein Hedge oder das Reduzieren offener Positionen, können aber nur getroffen werden, wenn die Ursache für den Verlust eindeutig identifiziert ist.

In den nächsten Kapiteln wird deshalb ein Ansatz vorgestellt, der einige dieser Nachteile behebt und der in Abbildung 2.2 als Portfolioanalyse auf Basis mehrerer Szenarien bezeichnet wird. Der Hauptvorteil der hier entwickelten Methode besteht darin, dass sie in gewisser Weise alle oben aufgeführten Stresstesting-Methoden vereint. Die verschiedenen Szenarien oder Sensitivitätsanalysen müssen nicht mehr einzeln implementiert werden und auch die Suche nach einem Worst-Case-Szenario wird erheblich vereinfacht. Während diese Ansätze versuchen, das Risiko eines Portfolios anhand von Einzelszenarien einzuschätzen, untersucht die Portfolioanalyse systematisch die gesamte Funktion. Das hat den Vorteil, dass man nicht befürchten muss, Szenarien mit fatalen Folgen zu übersehen.

Kapitel 3

Die Portfoliofunktion

Nachdem im letzten Kapitel herkömmliche Stresstest-Methoden wie Sensitivitäts-, Szenario- und Worst-Case-Analyse vorgestellt und diskutiert wurden, soll nun ein alternativer Ansatz zur Analyse der Risikostruktur eines Portfolios hergeleitet werden. Statt das Portfolio, wie in Kapitel 2 erläutert, nur an vereinzelten, eher willkürlich gewählten Punkten auszuwerten, wird die gesamte Gewinn- und Verluststruktur nachgebildet. Dazu wird die Portfoliofunktion auf einem Gebiet Ω approximiert. Die folgende Definition spezifiziert zunächst den Begriff der Portfoliofunktion.

Definition 3.1 (Portfoliofunktion v)
Gegeben sei ein Portfolio aus m Finanzprodukten $p_1, \ldots, p_m$, $m \in \mathbb{N}$, deren Wert von insgesamt n Risikofaktoren $x^1, \ldots, x^n$, $n \in \mathbb{N}$, abhängt. Die Risikofaktoren können zu einem Vektor $x := (x^1, \ldots, x^n)^T \in \mathbb{R}^n$ zusammengefasst werden, der einen Marktzustand beschreibt. Der Wert des Portfolios im Marktzustand x ist dann durch die Portfoliofunktion

$$v : \mathbb{R}^n \to \mathbb{R}, \qquad v(x) = v(x^1, \ldots, x^n) \tag{3.1}$$

gegeben. Mit x_{act} wird der Vektor der aktuellen Werte der Risikofaktoren, also der momentane Marktzustand, bezeichnet. Ebenso gibt

$$v_{act} := v(x_{act})$$

den aktuellen Wert des Portfolios an.

Als Portfolio kommt das gesamte Handelsbuch der Bank oder auch ein Teilportfolio davon, zum Beispiel auf der Ebene einzelner Geschäftsbereiche, Händler oder Instrumente, in Frage. Je nach Art und Anzahl der darin enthaltenen Produkte kann die Funktion $v(x)$ sehr kompliziert aussehen oder gar nicht mehr explizit darstellbar sein. Entsprechend aufwendig ist dann auch die Auswertung und Speicherung von v.

Bemerkung 3.2 (Darstellung von v)
Wie die Funktion v genau aussieht, hängt von der Zusammensetzung des Portfolios ab. Im Prinzip ist v die Summe der Bewertungsfunktionen aller im Portfolio enthaltenen Produkte $p_1, \ldots, p_m$:

$$v(x) = \sum_{j=1}^{m} p_j(x)$$

Der Wert $p_j(x)$ mancher Produkte kann allerdings nicht durch eine Bewertungsfunktion, sondern nur in einem Bewertungsprozess[6] bestimmt werden. Im Allgemeinen kann deshalb auch v nicht als explizite Formel angegeben werden.

Die Idee dieser Arbeit besteht nun darin, die Portfoliofunktion $v(x)$ durch eine Funktion $\tilde{v}(x)$ zu approximieren, die leicht auszuwerten ist und für spätere Analysen effizient gespeichert werden kann. Dazu wird der Definitionsbereich von v auf eine kompakte Menge

$$\Omega = [a^1, b^1] \times \ldots \times [a^n, b^n] \subset \mathbb{R}^n$$

eingeschränkt, wobei jedes Intervall $[a^i, b^i]$ mit $x^i_{act} \in [a^i, b^i]$ diejenigen Werte definiert, die der Risikofaktor x^i im Rahmen der Analyse annehmen kann. Die Größe dieser Intervalle ergibt sich aus der Analyse historischer Zeitreihen, aus regulatorischen Vorgaben oder aus Erfahrungswerten und hängt davon ab, über welchen Zeitraum die maximalen Wertveränderungen berücksichtigt werden sollen. Je länger der betrachtete Zeitraum, desto größer ist auch die potenzielle Abweichung vom aktuellen Wert x^i_{act}. Üblicherweise geht man je nach Verwendungszweck entweder von einer Ein-Tages- oder einer Zehn-Tages-Haltedauer aus. Während die Berücksichtigung von Ein-Tages-Veränderungen leicht zu einer Überschätzung des tatsächlichen Risikos führen kann, birgt die Annahme einer Zehn-Tages-Haltedauer die Gefahr, dass lediglich schockartige Marktveränderungen, nicht jedoch schleichende Krisen berücksichtigt werden.

Die verschiedenen Risikofaktoren, die bei der Bewertung eines Portfolios aus dem Marktrisikobereich eine Rolle spielen können, lassen sich den Risikokategorien Aktien und Aktienindizes, Zinskurven, Wechselkurse und Volatilitäten zuordnen. Je nach Art und Anzahl der im Portfolio enthaltenen Produkte kann die Menge der relevanten Risikofaktoren und damit die Dimension im Algorithmus sehr groß sein. Um den numerischen Aufwand der Approximation, der exponentiell mit der Anzahl der unabhängig geshifteten Risikofaktoren wächst, zu reduzieren, beschränkt man sich bei der Bestimmung der Intervalle $[a^i, b^i]$ auf eine Auswahl[7] derjenigen Risikofaktoren, die den Portfoliowert signifikant beeinflussen.[8] Die restlichen Risikofaktoren, die auch zur Bewertung der Produkte nötig sind, werden entweder auf einen dieser Risikofaktoren geschlüsselt und jeweils um denselben Betrag geshiftet oder nicht verändert.

[6] Ein Beispiel für einen derartigen Bewertungsprozess ist die Bepreisung von exotischen Optionen durch Monte-Carlo-Simulation.

[7] In den numerischen Tests hat sich eine Anzahl von bis zu 10 Risikofaktoren als geeignet erwiesen.

[8] Dies könnte beispielsweise im Rahmen einer Hauptkomponentenanalyse untersucht werden.

Je nach Bedarf könnte man Aktien und Volatilitäten zu bestimmten Märkten oder Indizes, wie zum Beispiel dem DAX, zusammenfassen. Andere Risikofaktoren wie Wechselkurse und Zinsen oder auch bereits gebündelte Indizes könnte man nach Währungsräumen sortieren. Hier wäre eine mögliche Unterteilung in Europa Euroraum, Europa Rest, Amerika, Asien und Rest denkbar. Statt jeden Risikofaktor unabhängig von den anderen zu variieren, wird für jede dieser Risikofaktorgruppen ein gemeinsamer Shift festgelegt und so die Dimension des Algorithmus reduziert. Dieses Verfahren wird auch als Mapping bezeichnet. Man muss sich allerdings dessen bewusst sein, dass durch die Zusammenfassung von Risikofaktoren implizit Korrelationen festlegt werden, die sich in Krisensituationen grundlegend von den Verhältnissen in normalen Marktbedingungen unterscheiden können.

Bei der Festlegung der Risikofaktoren kommt den Zinsen eine Sonderrolle zu. Während die meisten Risikofaktoren aus einem einzigen Wert bestehen, der entweder fallen oder steigen kann, ist die Lage bei Zinskurven komplizierter, weil es abhängig von der Laufzeit mehrere Stützstellen gibt, die bei einer Krise unterschiedlich große Ausschläge aufweisen können. Neben Parallelverschiebungen über alle Laufzeiten hinweg können auch Drehungen der Zinsstrukturkurve und Ausschläge im kurz-, mittel- oder langfristigen Bereich ein Risiko für das Portfolio darstellen und sollten in die Analyse einbezogen werden. Dazu wird die Zinskurve in einen kurzfristigen, mittelfristigen und langfristigen Bereich[9] unterteilt und für jeden Bereich ein eigener Shift definiert. Die Stützstellen innerhalb eines Bereiches werden jedoch immer um denselben Betrag geshiftet. Will man zusätzlich den Verlust messen, der aus einer Änderung des Creditspreads entsteht, muss man in jeder Währung mindestens zwei Zinskurven – eine Swap-Kurve und eine Government-Kurve – betrachten, die sich in unterschiedliche Richtungen bewegen können.

Die Wertänderungen oder Shifts s^i der einzelnen Risikofaktoren können entweder als relative Änderung

$$s^i = \frac{x^i}{x^i_{act}} - 1 \quad \text{für } i = 1, \ldots, n$$

oder als absolute Änderung

$$s^i = x^i - x^i_{act} \quad \text{für } i = 1, \ldots, n$$

angegeben werden. Allgemein ist es üblich, Shifts für Aktien- und Wechselkurse sowie für Volatilitäten als Relativveränderungen in Prozentpunkten anzugeben und Shifts für Zinsen als Absolutveränderungen in Basispunkten. Der Grund dafür ist, dass eine Multiplikation der gesamten Zinskurve mit einer relativen Änderung nicht zu einem reinen Parallelshift führen würde, sondern zusätzlich eine Drehung der Zinskurve verursachen würde. Um auch Parallelverschiebungen von Zinskurven realisieren zu können, müssen daher absolute Änderungen angegeben werden.

[9] Analog zur Studie [5] der Bundesbank sollen im kurzfristigen Bereich Stützstellen bis zu 3 Monaten, im mittelfristig Bereich Stützstellen von mehr als 3 Monaten bis zu 5 Jahren und im langfristigen Bereich Stützstellen von mehr als 5 Jahren zusammengefasst werden.

Will man das Portfolio im Marktzustand $x \in \Omega$ bewerten, müssen die Preise der darin enthaltenen Produkte $p_1, \ldots, p_m$ berechnet werden. Normalerweise geschieht dies direkt in den Front-Office-Systemen der Banken. Das hier entwickelte Programm dient jedoch nur als Pilotprogramm und kann daher nicht auf diese Bewertungsroutinen zurückgreifen. Um dennoch Portfolien bewerten zu können, wurden eigene Bewertungsroutinen für die Produktgruppen Aktien, Zinsprodukte, Optionen und Fremdwährungsprodukte implementiert. Die theoretischen Modelle dazu finden sich in Anhang A.

Schwankungen des Portfoliowertes stammen entweder von Veränderungen der Marktpreise, die den Wert der einzelnen Produkte beeinflussen, oder von Variationen der Positionen selbst. Das Stresstesting soll jedoch auf Grundlage statischer Portfolios durchgeführt werden, so dass die Analyse auf Marktpreisschwankungen beschränkt bleibt.

Nun könnte man theoretisch die Wertveränderung

$$\Delta v = v(x) - v_{act}$$

an jedem Punkt $x \in \Omega$ berechnen und damit die Gewinn- und Verluststruktur des gesamten Portfolios analysieren. Da die Funktionsvorschrift im Allgemeinen aber nur sehr schwierig auszuwerten ist und darüber hinaus nicht für unendlich viele Punkte berechnet werden kann, muss man sich auf einen endlichdimensionalen Funktionenraum einschränken, mit dessen Hilfe die Portfoliofunktion $v(x)$ approximiert werden kann. Bevor in den nächsten drei Kapiteln die theoretischen Grundlagen zur adaptiven Approximation von $v(x)$ erläutert werden, soll noch eine Regularitätseigenschaft von $v(x)$, die für eine spätere Fehlerschätzung benötigt wird, gezeigt werden.

Satz 3.3 (Lipschitz-Stetigkeit von v)
Sei Ω eine Teilmenge von $\mathbb{R}^n$ und $v : \Omega \to \mathbb{R}$ die Funktion aus (3.1)*. Dann ist v für die in dieser Arbeit betrachteten Produkte Lipschitz-stetig mit Lipschitz-Konstante $L > 0$, d.h.*

$$\|v(x) - v(y)\| \leq L \|x - y\| \quad \forall\, x, y \in \Omega.$$

Beweis:
Als Portfoliofunktion ist v die Summe der einzelnen Bewertungsfunktionen ihrer Produkte. Die in dieser Arbeit betrachteten Bewertungsfunktionen für Aktien, Zinsprodukte, Plain-Vanilla-Optionen und Fremdwährungspositionen sind Lipschitz-stetig, weshalb v als die Summe von Lipschitz-stetigen Funktionen auch selbst Lipschitz-stetig ist. □

Kapitel 4

Approximation der Portfoliofunktion

In den folgenden Kapiteln werden die theoretischen Grundlagen zur Approximation der Portfoliofunktion v aus Definition 3.1 dargestellt, die auf dem Gebiet

$$\Omega = [a^1, b^1] \times \ldots \times [a^n, b^n] \subset \mathbb{R}^n$$

berechnet werden soll. Die Idee der Approximation besteht darin, die Portfoliofunktion v auf einem Gitter durch eine Funktion zu diskretisieren, die leicht auszuwerten und effizient zu speichern ist und mit deren Hilfe der approximative Wert von v auf ganz Ω durch eine geeignete Interpolationsvorschrift bestimmt werden kann. Um bei der numerischen Approximation eine vorgegebene Genauigkeit sicherzustellen und gleichzeitig den Rechenaufwand möglichst gering zu halten, kommen zwei verschiedene Interpolationsmethoden in Betracht. Zum einen kann die Funktion durch Interpolationstechniken höherer Ordnung, wie zum Beispiel Polynom- oder Splineinterpolation, approximiert werden. Dabei sind oft zusätzliche Informationen über die zu approximierende Funktion, wie zum Beispiel Ableitungen, und eine geschickte Wahl der Stützstellen von Vorteil. Zum anderen kann multilineare Interpolation mit Methoden zur adaptiven Gittererzeugung kombiniert werden. Dies hat den Vorteil, dass die approximierende Funktion auch in höheren Dimensionen mit relativ geringem Aufwand berechnet und ausgewertet werden kann. Durch die adaptive Gittererzeugung, die ursprünglich aus der Theorie der partiellen Differentialgleichungen stammt, werden im Gegensatz zu äquidistanten Gittern nur dort weitere Stützstellen erzeugt, wo die Funktion kompliziert und der lokale Fehler der Approximation noch zu groß ist, was sich positiv auf den Rechenaufwand auswirkt. Letztere Methode soll in dieser Arbeit untersucht und implementiert werden.

Zunächst wird nun der Begriff des Gitters eingeführt, das eine Unterteilung von Ω in eine endliche Anzahl P von Quadern Q_j und N Knotenpunkten $\hat{x}_l$ vornimmt. Gegenüber einer Triangulierung, wie sie zum Beispiel in Grüne [6] für die Approximation der Hamilton-Jacobi-Bellman Gleichung diskutiert wird, hat ein Rechtecksgitter den Vorteil, dass eine Verfeinerung in die einzelnen Koordinatenrichtungen leichter implementiert werden kann und der Quader, in dem das Portfolio ausgewertet werden soll, einfach durch Koordinatenvergleich bestimmt

werden kann. Da außerdem das Gebiet Ω in diesem Anwendungsbereich sowieso quaderförmig ist, entstehen dadurch keine weiteren Probleme. Auf dem Gitter wird später ein endlichdimensionaler Funktionenraum definiert, der zur Approximation von v verwendet werden soll. Die Funktionen dieses Funktionenraums zeichnen sich durch gewisse Eigenschaften aus, die direkt zur Interpolationsvorschrift führen. Abschließend wird ein Regularitätsresultat für die approximierende Funktion gezeigt. Dieses Kapitel orientiert sich an Grüne [8] und [11, Abschnitt 3.2].

Definition 4.1 (Gitter Γ)
Sei $\Omega \subset \mathbb{R}^n$ gegeben durch $\Omega = [a^1, b^1] \times \ldots \times [a^n, b^n]$ mit Werten $a^i < b^i$ für $i = 1, \ldots, n$. Es seien Elemente

$$Q_j := \{x = (x^1, \ldots, x^n)^T \in \mathbb{R}^n \,|\, x^i \in [a_j^i, b_j^i] \text{ für alle } i = 1, \ldots, n\}$$

definiert, die auch als Quader bezeichnet werden. Die Knoten oder Eckpunkte $\hat{x}_{j_k}$, $k = 1, \ldots, 2^n$ eines Elements Q_j sind gegeben durch die Menge

$$nodes(Q_j) := \{x \in \mathbb{R}^n \,|\, x^i = a_j^i \text{ oder } x^i = b_j^i \text{ für alle } i = 1, \ldots, n\}.$$

Ein n-dimensionales Gitter Γ auf Ω ist nun eine endliche Menge von Quadern $Q_j, j = 1, \ldots, P$, so dass

$$\bigcup_{j=1}^{P} Q_j = cl\Omega$$

und

$$int\, Q_j \cap int\, Q_k = \emptyset \qquad \forall\ j, k = 1, \ldots, P,\ j \neq k.$$

Die Knoten $\hat{x}_l$ von Γ sind gegeben durch

$$nodes(\Gamma) := \bigcup_{j=1,\ldots,P} nodes(Q_j),$$

wobei N die Anzahl dieser Knoten sei.

Der Durchmesser k gibt die Feinheit des Gitters Γ an.

Definition 4.2 (Durchmesser k)
Sei Γ ein Gitter auf Ω mit P Quadern Q_j. Der Wert

$$k_j := \sqrt{(k_j^1)^2 + \ldots + (k_j^n)^2} \quad \text{mit } k_j^i := b_j^i - a_j^i$$

bezeichnet den Durchmesser des Quaders Q_j. Der maximale Durchmesser der Quader

$$k := \max_{j=1,\ldots,P} k_j$$

gibt den Durchmesser bzw. die Feinheit des Gitters Γ an.

Am Übergang zu verfeinerten Gebieten können irreguläre Knoten entstehen.

Definition 4.3 (reguläre und irreguläre Knoten)
Ein Knoten $\hat{x}_l$ heißt regulär, falls $\hat{x}_l \in nodes(Q_j)$ für alle j mit $\hat{x}_l \in Q_j$, sonst irregulär. Mit $reg(\Gamma)$ wird die Menge der regulären Knoten von Γ bezeichnet.

Abbildung 4.1 zeigt ein zweidimensionales Gitter mit irregulären Knoten.

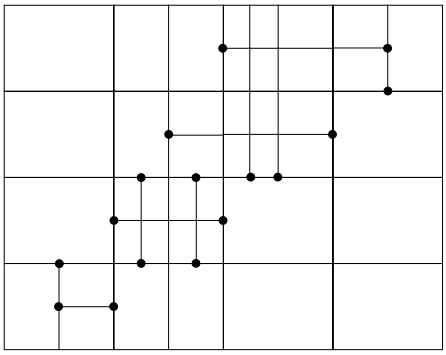

Abbildung 4.1: Gitter in 2d mit irregulären Knoten

Das Ziel der Approximation besteht nun darin, auf dem Gitter Γ eine Interpolationsfunktion v_Γ zu bestimmen, die leicht auszuwerten und effizient zu speichern ist, und für die an allen Knoten $\hat{x}_1, \ldots, \hat{x}_N$ von Γ die Gleichung

$$v_\Gamma(\hat{x}_l) = v(\hat{x}_l) \quad \text{für } l = 1, \ldots, N$$

gilt. Der Wert der Portfoliofunktion $v(x)$ innerhalb eines Quaders kann dann mit Hilfe von $v_\Gamma(x)$ approximiert werden. Der endlichdimensionale Funktionenraum, der zur Approximation der Portfoliofunktion verwendet wird, ist folgendermaßen definiert:

Definition 4.4 (multilinearer Funktionenraum $\mathcal{W}$)
Der Raum der multilinearen Funktionen auf Ω bezüglich eines Gitters Γ ist definiert durch

$$\mathcal{W} := \{w \in C(cl\Omega) \,\big|\, w(x + \alpha e^i) \text{ ist auf jedem } Q_j \text{ für alle } i \text{ linear in } \alpha\},$$

wobei e^i, $i = 1, \ldots, n$, die Basisvektoren des $\mathbb{R}^n$ sind und $C(cl\Omega)$ den Raum der auf dem Abschluss von Ω stetigen Funktionen darstellt.

Die Funktionen von $\mathcal{W}$ zeichnen sich durch folgende Eigenschaften aus.

Lemma 4.5 (wichtige Eigenschaften von $\mathcal{W}$)

(i) Jede Funktion $w \in \mathcal{W}$ ist eindeutig durch ihre Werte $w(\hat{x}_l)$ in den regulären Knotenpunkten des Gitters Γ bestimmt.

(ii) Für ein Rechteck Q_j von Γ mit Knoten $\hat{x}_{j_k} = (\hat{x}^1_{j_k}, \ldots, \hat{x}^n_{j_k})^T$, $k = 1, \ldots, 2^n$, und Koordinaten $a^i_j < b^i_j$, $i = 1, \ldots, n$, ist der Wert $w(x)$ für $x = (x^1, \ldots, x^n)^T \in Q_j$ durch die multilineare Interpolation

$$w(x) = \sum_{k=1}^{2^n} \mu_{j_k}(x) w(\hat{x}_{j_k}) \quad \text{für } x \in Q_j \tag{4.1}$$

mit

$$\mu_{j_k}(x) = \prod_{i=1}^{n} g_{j_k i}(x^i) \tag{4.2}$$

und

$$g_{j_k i}(x^i) = \begin{cases} (b^i_j - x^i)/(b^i_j - a^i_j) & \text{falls } \hat{x}^i_{j_k} = a^i_j \\ (x^i - a^i_j)/(b^i_j - a^i_j) & \text{falls } \hat{x}^i_{j_k} = b^i_j \end{cases} \tag{4.3}$$

gegeben. Insbesondere gilt hierbei $\mu_{j_k}(x) \geq 0$ für $k = 1, \ldots, 2^n$ und $\sum_{k=1}^{2^n} \mu_{j_k}(x) = 1$.

Jede Funktion $w \in \mathcal{W}$ kann man also anhand ihrer Werte $w(\hat{x}_l)$ in den regulären Knotenpunkten des Gitters eindeutig identifizieren. Somit ist der Funktionenraum $\mathcal{W}$ ein N-dimensionaler Vektorraum über $\mathbb{R}$, wobei N die Anzahl der regulären Knoten $\hat{x}_l$ des Gitters ist.

Beispiel 4.6 (Berechnung von $w(x)$)
Sei Q_j ein Element eines zweidimensionalen Gitters Γ mit

$$\begin{aligned} a^1_j &= 0 \qquad b^1_j = 4 \\ a^2_j &= 0 \qquad b^2_j = 3, \end{aligned}$$

wie in Abbildung 4.2 gezeigt. Die Funktion $w \in \mathcal{W}$ sei auf Q_j durch folgende Werte an den Eckpunkten $\hat{x}_{j_k}$, $k = 1, \ldots, 4$, festgelegt:

$$\begin{aligned} \hat{x}_{j_1} &= (0,0)^T, \quad w(\hat{x}_{j_1}) = 0 \\ \hat{x}_{j_2} &= (4,0)^T, \quad w(\hat{x}_{j_2}) = 40 \\ \hat{x}_{j_3} &= (0,3)^T, \quad w(\hat{x}_{j_3}) = 30 \\ \hat{x}_{j_4} &= (4,3)^T, \quad w(\hat{x}_{j_4}) = 70. \end{aligned}$$

Berechnet werden soll nun der Wert von w im Punkt

$$x = (3,2)^T.$$

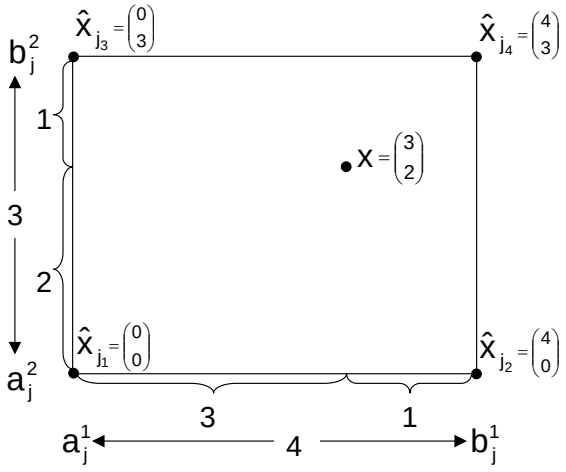

Abbildung 4.2: Quader im Beispiel 4.6

Dazu müssen zunächst die $\mu_{j_k}(x)$ aus (4.2) und (4.3) ermittelt werden:

$$\begin{aligned}
\mu_{j_1}(x) = g_{j_1 1}(x^1) \cdot g_{j_1 2}(x^2) &= \frac{1}{4} \cdot \frac{1}{3} = \frac{1}{12} \\
\mu_{j_2}(x) = g_{j_2 1}(x^1) \cdot g_{j_2 2}(x^2) &= \frac{3}{4} \cdot \frac{1}{3} = \frac{1}{4} \\
\mu_{j_3}(x) = g_{j_3 1}(x^1) \cdot g_{j_3 2}(x^2) &= \frac{1}{4} \cdot \frac{2}{3} = \frac{1}{6} \\
\mu_{j_4}(x) = g_{j_4 1}(x^1) \cdot g_{j_4 2}(x^2) &= \frac{3}{4} \cdot \frac{2}{3} = \frac{1}{2}.
\end{aligned}$$

Jetzt kann der Wert von $w(x)$ mit Hilfe der bilinearen Interpolation (4.1) berechnet werden:

$$\begin{aligned}
w(x) &= \sum_{k=1}^{4} \mu_{j_k}(x) w(\hat{x}_{j_k}) \\
&= \frac{1}{12} \cdot 0 + \frac{1}{4} \cdot 40 + \frac{1}{6} \cdot 30 + \frac{1}{2} \cdot 70 \\
&= 50
\end{aligned}$$

Man beachte, dass die Werte $w(\hat{x}_l)$ in den Eckpunkten durch die bilineare Funktion $f(x) = 10(x^1 + x^2)$ gegeben sind. Eine Auswertung der Funktion f an der gewünschten Stelle $x = (3, 2)^T$ liefert

$$f(x) = 10(3 + 2) = 50.$$

Das Ergebnis der multilinearen Interpolation stimmt also exakt mit dem Funktionswert überein. Natürlich gilt dies nur für multilineare Funktionen. □

Um die Existenz einer stetigen Funktion $w \in \mathcal{W}$ sicherzustellen, müssen einige Bedingungen beachtet werden, die an dieser Stelle erläutert werden sollen, auch wenn in diesem An-

wendungsbereich zur approximativen Auswertung der Portfoliofunktion eine stetige Funktion nicht unbedingt nötig ist. Zum einen dürfen in (4.1) zur Berechnung von $w(x)$ keine irregulären Knoten verwendet werden. Der Wert der Portfoliofunktion in den irregulären Knoten kann zum Schluss durch die Werte in den regulären Nachbarknoten interpoliert werden. Zum anderen muss folgende Regularitätsbedingung erfüllt sein.

Bemerkung 4.7 (Regularitätsbedingung für Γ)
Für Dimensionen $n \geq 3$ muss ein Gitter Γ gewisse Regularitätsbedingungen erfüllen, um die Existenz einer stetigen Funktion $w \in \mathcal{W}$ für beliebig vorgegebene Werte in den regulären Knoten sicherzustellen. Eine ausreichende Regularitätsbedingung für diesen Zweck ist gegeben durch

$$(nodes(Q_j) \cup nodes(Q_k)) \cap (Q_j \cap Q_k) \subset nodes(Q_l) \text{ für } l = j \text{ oder } l = k$$

für alle $j, k = 1, \ldots, P$ mit $j \neq k$. Für zwei angrenzende Elemente muss es also immer ein gemeinsames feineres Element geben.
Außerdem ist es von Vorteil, große Unterschiede bei der Größe von zwei angrenzenden Elementen zu vermeiden, um die Anzahl der irregulären Knoten zu reduzieren.

Beispiel 4.8 (Verletzung der Regularitätsbedingung)
Abbildung 4.3 zeigt eine Situation, in der obige Regularitätsbedingung nicht erfüllt ist. Infolgedessen kommt es bei der Berechnung des Werts im mittleren Knotenpunkt der angrenzenden Flächen zu Überschneidungen: Im linken Quader würde man den Wert aus den beiden horizontalen Nachbarknoten interpolieren, während man im rechten Quader die vertikalen Nachbarknoten benutzen würde. Im Allgemeinen stimmen diese beiden Ergebnisse aber nicht überein, so dass die Stetigkeit von w nicht mehr gewährleistet ist. □

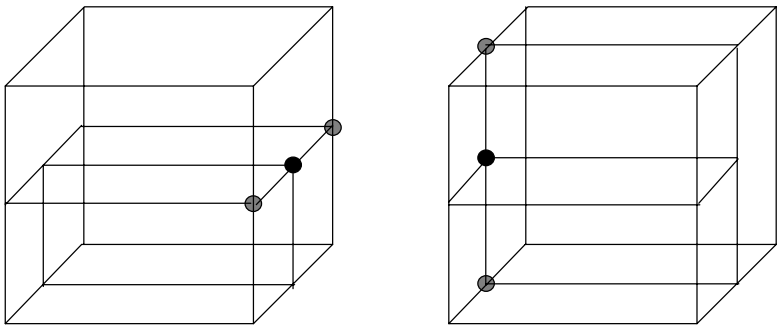

Abbildung 4.3: Verletzung der Regularitätsbedingung

Mit dem Raum der multilinearen Funktionen $\mathcal{W}$ kann nun eine endlichdimensionale Appro-

ximation $v_\Gamma \in \mathcal{W}$ von v definiert werden. Für ein Gitter Γ mit regulären Knoten $\hat{x}_l$ sei

$$\begin{aligned} v_\Gamma(\hat{x}_l) &:= v(\hat{x}_l) \qquad \text{für alle regulären Knoten } \hat{x}_l \text{ von } \Gamma \\ v_\Gamma(x) &= \sum_{k=1}^{2^n} \mu_{j_k}(x) v_\Gamma(\hat{x}_{j_k}) \qquad \text{für } x \in Q_j. \end{aligned} \tag{4.4}$$

Der Wert $v_\Gamma(x)$ innerhalb eines Rechtecks kann also durch multilineare Interpolation aus den Eckpunkten des x enthaltenden Quaders gewonnen werden. Zur approximativen Berechnung von $v(x)$ genügt es daher, die Werte von v in den Eckpunkten des Gitters zu berechnen.

Bevor nun im nächsten Kapitel der Fehler dieser Approximation untersucht wird, soll eine Regularitätseigenschaft von v_Γ überprüft werden. Dazu wird folgendes Lemma benötigt.

Lemma 4.9 (Regularitätseigenschaft von $w \in \mathcal{W}$)
Sei $w \in \mathcal{W}$ eine beliebige Funktion auf einem Gitter Γ. Dann gilt die Abschätzung

$$\frac{|w(x) - w(y)|}{\|x - y\|_1} \leq \max_{\hat{x},\hat{y} \in nodes(\Gamma),\ \hat{x} \neq \hat{y}} \frac{|w(\hat{x}) - w(\hat{y})|}{\|\hat{x} - \hat{y}\|_1}$$

für alle $x, y \in \Omega$ mit $x \neq y$. $\|\cdot\|_1$ bezeichnet die 1-Norm mit $\|x\|_1 := \sum_{i=1}^n |x^i|$.

Beweis:
Der Beweis folgt direkt aus der Definition von $\mathcal{W}$. □

Basierend auf diesem Lemma kann nun ein Lipschitz-Ergebnis für die Approximation v_Γ bewiesen werden.

Proposition 4.10 (Lipschitz-Stetigkeit von v_Γ)
Sei Γ ein Gitter und betrachte die Funktionen $v_\Gamma \in \mathcal{W}$ definiert durch (4.4). Dann ist v_Γ Lipschitz-stetig, d.h. es gilt die Abschätzung

$$|v_\Gamma(x) - v_\Gamma(y)| \leq L \|x - y\|_1$$

für alle $x, y \in \Omega$, wobei L die Lipschitz-Konstante von v aus Satz 3.3 ist.

Beweis:
Da v_Γ eine Funktion aus $\mathcal{W}$ ist, die an den Eckpunkten $\hat{x}_l$ mit v übereinstimmt, folgt mit Hilfe von Lemma 4.9

$$\begin{aligned} \frac{|v_\Gamma(x) - v_\Gamma(y)|}{\|x - y\|_1} &\leq \max_{\hat{x},\hat{y} \in nodes(\Gamma),\ \hat{x} \neq \hat{y}} \frac{|v_\Gamma(\hat{x}) - v_\Gamma(\hat{y})|}{\|\hat{x} - \hat{y}\|_1} \\ &= \max_{\hat{x},\hat{y} \in nodes(\Gamma),\ \hat{x} \neq \hat{y}} \frac{|v(\hat{x}) - v(\hat{y})|}{\|\hat{x} - \hat{y}\|_1} \leq L \end{aligned}$$

und daraus nach Multiplikation mit $\|\hat{x} - \hat{y}\|_1$ die gewünschte Behauptung. □

Kapitel 5

Fehlerschätzung

Nachdem im letzten Kapitel ein geeigneter Funktionenraum $\mathcal{W}$ zur Approximation von v betrachtet wurde, soll nun der Fehler, der durch diese Approximation entsteht, untersucht werden. Ziel ist es, auf Basis des Fehlers ein lokal verfeinertes Gitter zu konstruieren, auf dem eine bessere Approximation möglich ist.

Definition 5.1 (Approximationsfehler)
Sei ein Gitter Γ auf Ω mit Elementen Q_j, $j = 1, \ldots, P$, und eine Approximation $v_\Gamma \in \mathcal{W}$ von v gegeben. Für jedes $x \in \Omega$ wird der punktweise lokale Fehler der Approximation durch

$$\eta(x) := |v(x) - v_\Gamma(x)|$$

definiert.

$$\eta_j := \sup_{x \in Q_j} \eta(x)$$

bezeichnet den lokalen Fehler auf einem Element Q_j,

$$\eta := \max_{j=1,\ldots,P} \eta_j = \|v - v_\Gamma\|_\infty$$

den globalen Approximationsfehler auf Ω.

Natürlich ist es nicht möglich, die Differenz $\|v - v_\Gamma\|_\infty$ ohne Kenntnis von v exakt anzugeben. Über geeignete Größen kann der Fehler aber zumindest abgeschätzt werden. Dies kann entweder a-priori, d.h. ohne Kenntnis der Approximation v_Γ, geschehen oder a-posteriori unter Verwendung von v_Γ.

5.1 A-priori Fehlerschätzer

Um sicherzustellen, dass die lokalen Fehler η_j, $j = 1, \ldots, P$, im Laufe der adaptiven Verfeinerung schließlich gegen Null konvergieren und eine gute Approximation der Portfoliofunktion v erreicht wird, soll nun eine Beziehung zwischen der Größe des Elements Q_j und der Größe des lokalen Fehlers η_j hergeleitet werden. Dieser Abschnitt orientiert sich an Grüne [11, Abschnitt 3.2.3].

Zunächst wird die Projektion einer beliebigen Funktion nach $\mathcal{W}$ betrachtet.

Definition 5.2 (Projektion)
Für eine Funktion $q : \Omega \to \mathbb{R}$ und ein Gitter Γ mit regulären Knotenpunkten $\hat{x}_l$, $l = 1, \ldots, N$, bezeichnet $\pi_{\mathcal{W}} q$ die eindeutige Funktion $w \in \mathcal{W}$ mit

$$w(\hat{x}_l) = q(\hat{x}_l) \text{ für alle } l = 1, \ldots, N.$$

Das folgende Lemma gibt Auskunft über den dabei entstehenden Projektionsfehler, der auch Interpolationsfehler genannt wird.

Lemma 5.3 (Interpolationsfehler)
Auf einem Gitter Γ mit P Elementen $Q_1, \ldots, Q_P$ gilt für eine Lipschitz-stetige Funktion $q : \Omega \to \mathbb{R}$ mit Lipschitz-Konstante L_q die Abschätzung

$$\sup_{x \in Q_j} |q(x) - \pi_{\mathcal{W}} q(x)| \leq L_q k_j \text{ für alle } j = 1, \ldots, P$$

mit dem Wert k_j aus Definition 4.2.

Beweis:
Sei $x \in Q_j$ ein beliebiger Punkt und seien $\hat{x}_{j_1}, \ldots, \hat{x}_{j_{2^n}}$ die Eckpunkte dieses Quaders. Dann gilt $\|x - \hat{x}_{j_i}\| \leq k_j$ für $i = 1, \ldots, 2^n$ und somit $|q(x) - q(\hat{x}_{j_i})| \leq L_q k_j$. Mit Lemma 4.5 folgt

$$\begin{aligned}
|q(x) - \pi_{\mathcal{W}} q(x)| &= \left| q(x) - \sum_{i=1}^{2^n} \mu_{j_i}(x) q(\hat{x}_{j_i}) \right| \\
&= \left| \sum_{i=1}^{2^n} \mu_{j_i}(x) q(x) - \sum_{i=1}^{2^n} \mu_{j_i}(x) q(\hat{x}_{j_i}) \right| \\
&\leq \sum_{i=1}^{2^n} \mu_{j_i}(x) \, |q(x) - q(\hat{x}_{j_i})| = \sum_{i=1}^{2^n} \mu_{j_i}(x) L_q k_j = L_q k_j,
\end{aligned}$$

wobei im zweiten und im letzten Schritt ausgenutzt wurde, dass $\sum_{i=1}^{2^n} \mu_{j_i}(x) = 1$. Da x beliebig gewählt war, gilt die Abschätzung auch für das Supremum. □

Daraus lässt sich nun eine Beziehung zwischen der Größe des Elements Q_j und dem lokalen Approximationsfehler η_j herleiten.

Lemma 5.4 (A-priori Fehler)
Gegeben sei die Portfoliofunktion $v : \Omega \to \mathbb{R}$ aus Definition 3.1 mit Lipschitz-Konstante L und ihre Approximation v_Γ. Dann entsteht durch die Approximation in jedem Quader Q_j des Gitters Γ der a-priori Fehler

$$\eta_j \leq Lk_j$$

mit dem Wert k_j aus Definition 4.2.

Beweis:
v_Γ ist laut (4.4) die Projektion von v nach $\mathcal{W}$. Das gewünschte Resultat folgt sofort aus Lemma 5.3, wenn man die Lipschitz-stetige Funktion q durch v und ihre Projektion $\pi_{\mathcal{W}} q$ durch v_Γ ersetzt. □

5.2 A-posteriori Fehlerschätzer

Die Abschätzung für den Approximationsfehler η_j in Lemma 5.4 ist eine sogenannte a-priori Abschätzung, da sie ohne Kenntnis von v_Γ auskommt. Nun soll der Fehler a-posteriori, d.h. unter Einbeziehung von v_Γ, abgeschätzt werden. Aufbau und Argumentation dazu basieren auf Grüne [8] und [11, Abschnitt 4.4].

Formal ist ein Fehlerschätzer durch die folgende Definition gegeben.

Definition 5.5 (A-posteriori Fehlerschätzer)
Betrachte ein Gitter Γ mit P Elementen $Q_1, \ldots, Q_P$. Ein lokaler a-posteriori Fehlerschätzer für den Approximationsfehler $\eta = \|v - v_\Gamma\|_\infty$ aus Definition 5.1 ist eine Menge von Werten $\widetilde{\eta}_1, \ldots, \widetilde{\eta}_P$, für die gilt:
Es gibt vom Gitter Γ unabhängige Konstanten $C_1, C_2 > 0$, so dass für den Wert $\widetilde{\eta} := \max_{j=1,\ldots,P} \widetilde{\eta}_j$ die Abschätzungen

$$C_1 \widetilde{\eta} \leq \eta \leq C_2 \widetilde{\eta} \tag{5.1}$$

gelten. Man sagt auch, dass der Fehlerschätzer effizient und zuverlässig ist.
Gilt darüber hinaus die Abschätzung

$$C_1 \widetilde{\eta}_j \leq \eta_j, \tag{5.2}$$

so heißt der Fehlerschätzer lokal effizient.

Die obere Schranke in (5.1) stellt sicher, dass kleine Fehlerschätzer auch wirklich kleine globale Fehler implizieren und garantiert damit die Zuverlässigkeit des Fehlerschätzers. Insbesonde-

re kann der globale Fehler durch Verfeinerung der Elemente mit großem Fehlerschätzer, d.h. durch Verringerung des Fehlerschätzers, reduziert werden. Andererseits gewährleistet die untere Schranke die Effizienz, da ein großer Fehlerschätzer immer mit einem großem Fehler einhergeht, der Fehler also nicht überschätzt wird.

Da auf Basis des Fehlerschätzers ein lokal verfeinertes Gitter Γ' konstruiert werden soll, das eine bessere Approximation von v ermöglicht, ist besonders die lokale Effizienz (5.2) wichtig, die bei großem Fehlerschätzer $\widetilde{\eta}_j$ einen großen Fehler in der Nähe impliziert. Es liegt also nahe, die Regionen mit großen $\widetilde{\eta}_j$ weiter zu verfeinern, während die Approximation in Regionen mit kleinen $\widetilde{\eta}_j$ gleich bleibt. Dies führt zur sogenannten adaptiven Gittererzeugung, bei der ein Gitter solange lokal verfeinert wird, bis der globale Fehler $\|v - v_\Gamma\|_\infty$ kleiner als eine vorgegebene Fehlertoleranz *tol* ist.

5.2.1 Konstruktion unter Verwendung der Portfoliofunktion

Die Idee zur Konstruktion des Fehlerschätzers $\widetilde{\eta}_1, \ldots, \widetilde{\eta}_P$ besteht nun darin, dass der lokale Fehler η_j in der Praxis nicht exakt berechnet werden kann, da in jedem Quader unendlich viele Punkte ausgewertet werden müssten. Stattdessen wird der Fehler in jedem Quader an einer endlichen Anzahl von Punkten $\widetilde{x}_{j_k}$, $k = 1, \ldots, p$, der Testpunktmenge $tp(Q_j) \subset Q_j$ ausgewertet und als Maximum über diese Werte approximiert:

$$\eta_j = \sup_{x \in Q_j} |v(x) - v_\Gamma(x)| \approx \max_{k=1,\ldots,p} |v(\widetilde{x}_{j_k}) - v_\Gamma(\widetilde{x}_{j_k})| \tag{5.3}$$

Definition 5.6 (Fehlerschätzer $\widetilde{\eta}_j$)
Betrachte die Testpunktmengen $tp(Q_j)$ für $j = 1, \ldots, P$ und die Funktion $\widetilde{\eta} : tp(Q_j) \to \mathbb{R}_0^+$, gegeben durch

$$\widetilde{\eta}(\widetilde{x}_{j_k}) := |v(\widetilde{x}_{j_k}) - v_\Gamma(\widetilde{x}_{j_k})| .$$

Durch die Werte

$$\widetilde{\eta}_j := \max_{k=1,\ldots,p} \widetilde{\eta}(\widetilde{x}_{j_k})$$

für $j = 1, \ldots, P$ ist dann ein Fehlerschätzer im Sinn von Definition 5.5 definiert.

Das folgende Lemma zeigt, dass die oben geforderten Eigenschaften eines lokalen Fehlerschätzers erfüllt sind und die Auswertung in endlich vielen Testpunkten den ursprünglichen Fehler η_j in der Tat gut approximiert.

Lemma 5.7 (Fehlerschätzer $\widetilde{\eta}_j$)
Für jedes Q_j betrachte eine endliche Testpunktmenge $tp(Q_j)$ mit $\widetilde{x}_{j_k} \in tp(Q_j)$, $k = 1, \ldots, p$, für die für den nichtsymmetrischen Hausdorff-Abstand H^* zwischen Q_j und $tp(Q_j)$

$$H^*(Q_j, tp(Q_j)) = \sup_{x \in Q_j} \min_{k=1,\ldots,p} \|x - \widetilde{x}_{j_k}\| \leq \alpha$$

für ein $\alpha > 0$ gilt. Der Fehlerschätzer aus Definition 5.6 genügt den Ungleichungen

$$\widetilde{\eta} \leq \eta \leq \widetilde{\eta} + 2L\alpha,$$

wobei $\widetilde{\eta} = \max\limits_{j=1,\ldots,P} \widetilde{\eta}_j$ und L die Lipschitz-Konstante von v ist. Darüber hinaus gilt lokal

$$\widetilde{\eta}_j \leq \eta_j \quad \forall j = 1, \ldots, P.$$

Beweis:
Die Ungleichung für $\widetilde{\eta}_j$ folgt aus der Tatsache, dass $\widetilde{\eta}_j$ nur auf einer Teilmenge von Q_j berechnet wird:

$$\widetilde{\eta}_j = \max_{k=1,\ldots,p} \eta(\widetilde{x}_{j_k}) \leq \sup_{x \in Q_j} \eta(x) = \eta_j.$$

Diese Abschätzung gilt für beliebiges $j = 1, \ldots, P$ und damit auch für das Maximum über j

$$\widetilde{\eta} = \max_{j=1,\ldots,P} \widetilde{\eta}_j \leq \max_{j=1,\ldots,P} \eta_j = \eta,$$

womit die erste Ungleichung für $\widetilde{\eta}$ gezeigt ist.

Für die zweite Abschätzung wird die Lipschitz-Stetigkeit von v und v_Γ ausgenutzt. Es gilt

$$\begin{aligned}
\eta_j(x) &= |v(x) - v_\Gamma(x)| \\
&= |v(x) - v(\widetilde{x}_{j_k}) + v(\widetilde{x}_{j_k}) - v_\Gamma(\widetilde{x}_{j_k}) + v_\Gamma(\widetilde{x}_{j_k}) - v_\Gamma(x)| \\
&\leq |v(x) - v(\widetilde{x}_{j_k})| + |v(\widetilde{x}_{j_k}) - v_\Gamma(\widetilde{x}_{j_k})| + |v_\Gamma(\widetilde{x}_{j_k}) - v_\Gamma(x)| \\
&\leq L \|x - \widetilde{x}_{j_k}\| + \widetilde{\eta}_j(\widetilde{x}_{j_k}) + L \|x - \widetilde{x}_{j_k}\| \\
&\leq \widetilde{\eta}_j(\widetilde{x}_{j_k}) + 2L\alpha.
\end{aligned}$$

Durch Bilden des Maximums über Ω ergibt sich

$$\eta = \sup_{x \in \Omega} \eta_j(x) \leq \max_{j=1,\ldots,P} \max_{k=1,\ldots,p} \widetilde{\eta}_j(\widetilde{x}_{j_k}) + 2L\alpha = \widetilde{\eta} + 2L\alpha$$

und daraus die gewünschte Behauptung. □

Die Konstante $C_2 = 2L\alpha$ für die obere Abschätzung von η ist zwar additiv und nicht, wie in Definition 5.5 gefordert, multiplikativ. Im Prinzip kann sie jedoch beliebig klein gemacht werden und schränkt deshalb nicht die Zuverlässigkeit des Fehlerschätzers $\widetilde{\eta}_j$ ein.

Je gleichmäßiger und dichter die Testpunkte $\widetilde{x}_{j_k} \in tp(Q_j)$ in den Quadern Q_j verteilt sind, desto kleiner ist α und desto besser ist auch die Approximation (5.3) von η_j. Mit einer sorgfältigen Wahl der Testpunkte, die im nächsten Kapitel besprochen wird, lässt sich der Fehler effizient messen. Auf sehr großen Elementen Q_j kann durch diese Vorgehensweise der exakte Fehler allerdings auch unterschätzt werden. Es ist deshalb ratsam, das Anfangsgitter Γ_0 nicht zu grob zu wählen.

Ein Nachteil dieser Art der Fehlerschätzung ist, dass die Portfoliofunktion an jedem Testpunkt des Quaders neu ausgewertet werden muss, was gerade in höheren Dimensionen beträchtlichen Aufwand verursachen kann. Deutlich wird dies, wenn man die Anzahl der Testpunkte pro Quader betrachtet, die in Tabelle 6.1 für verschiedene Dimensionen n aufgelistet ist.

5.2.2 Konstruktion ohne Verwendung der Portfoliofunktion

Ein alternativer Ansatz zur Konstruktion eines Fehlerschätzers, der dieses Problem zu beheben versucht, besteht nun darin, den Fehler nur aus Werten zu berechnen, die sowieso schon in der Approximation v_Γ vorhanden sind bzw. ohne großen Aufwand aus dieser berechnet werden können, so dass kein zusätzlicher Bewertungsaufwand entsteht. Die grundlegende Idee hierzu geht auf Grüne [9, Abschnitt 5.5] zurück.

Es wird folgende Annahme gemacht:

Annahme 5.8 (Folge der Approximationsfehler)
Sei Γ_i, $i = 0, 1, 2, \ldots$, eine Folge von Gittern mit nodes$(\Gamma_{i-1}) \subset$ nodes(Γ_i), die iterativ aus der lokalen Verfeinerung des Anfangsgitters Γ_0 entstanden sind. Es wird angenommen, dass der Approximationsfehler $\eta^{\Gamma_{i-1}}(x)$ durch eine Verfeinerung von Γ_{i-1} zu Γ_i reduziert wird, d.h.

$$\eta^{\Gamma_i}(x) < \eta^{\Gamma_{i-1}}(x).$$

Die Annahme bedeutet, dass der Übergang von Γ_{i-1} zu einem feineren Gitter Γ_i zu einer genaueren Approximation v_{Γ_i} und damit zu einem kleineren Fehler führt. Dies ist nach Lemma 5.4, das den lokalen Fehler in Abhängigkeit von der Größe eines Quaders schätzt, plausibel.

Die Konstruktion eines Fehlerschätzers $\widetilde{\eta}^{\Gamma_i}$ für den Fehler η^{Γ_i} beruht nun auf einem Vergleich der Approximationen v_{Γ_i} und $v_{\Gamma_{i-1}}$. Relativ zu der groben Approximation $v_{\Gamma_{i-1}}$ ist die genauere Approximation v_{Γ_i} annähernd exakt, so dass für den exakten Fehler $\eta^{\Gamma_{i-1}}(x)$ die Approximation

$$\eta^{\Gamma_{i-1}}(x) = \left|v_{\Gamma_{i-1}}(x) - v(x)\right| \approx \left|v_{\Gamma_{i-1}}(x) - v_{\Gamma_i}(x)\right| \tag{5.4}$$

gelten sollte. Leider ist es mit dieser Idee nicht möglich, einen Fehlerschätzer für den aktuellen Fehler η^{Γ_i} zu bestimmen, da man dazu die Approximation $v_{\Gamma_{i+1}}$ benötigen würde, die aber noch nicht berechnet ist. Man begnügt sich daher mit einem Schätzer für den Fehler des Vorgängergitters Γ_{i-1}.

Definition 5.9 (Fehlerschätzer $\widetilde{\eta}_j^{\Gamma_{i-1}}$)
Betrachte die Funktion $\widetilde{\eta}^{\Gamma_{i-1}} : \Omega \to \mathbb{R}_0^+$, gegeben durch

$$\widetilde{\eta}^{\Gamma_{i-1}}(x) := \left|v_{\Gamma_{i-1}}(x) - v_{\Gamma_i}(x)\right|.$$

Basierend auf $\widetilde{\eta}^{\Gamma_{i-1}}(x)$ ist ein lokaler Fehlerschätzer mittels

$$\widetilde{\eta}_j^{\Gamma_{i-1}} := \sup_{x \in Q_j} \widetilde{\eta}^{\Gamma_{i-1}}(x)$$

für $j = 1, \ldots, P$ definiert.

Das folgende Lemma zeigt, dass sich der Fehlerschätzer $\widetilde{\eta}_j^{\Gamma_{i-1}}$ formal in den Rahmen von Definition 5.5 fassen lässt.

Lemma 5.10 (Fehlerschätzer $\widetilde{\eta}_j^{\Gamma_{i-1}}$)
Es gelte Annahme 5.8 für $\eta^{\Gamma_{i-1}}(x)$ und $\eta^{\Gamma_i}(x)$, genauer existiere $\alpha < 1$, so dass die Ungleichung

$$\eta^{\Gamma_i}(x) \leq \alpha \eta^{\Gamma_{i-1}}(x)$$

für alle $x \in \Omega$ gilt. Für den Fehlerschätzer aus Definition 5.9 gelten dann die Ungleichungen

$$\frac{1}{1+\alpha} \widetilde{\eta}^{\Gamma_{i-1}} \leq \eta^{\Gamma_{i-1}} \leq \frac{1}{1-\alpha} \widetilde{\eta}^{\Gamma_{i-1}}$$

mit $\widetilde{\eta}^{\Gamma_{i-1}} = \max\limits_{j=1,\ldots,P} \widetilde{\eta}_j^{\Gamma_{i-1}}$. Darüber hinaus gilt

$$\frac{1}{1+\alpha} \widetilde{\eta}_j^{\Gamma_{i-1}} \leq \eta_j^{\Gamma_{i-1}}.$$

Beweis:
Aus der Definition von $\widetilde{\eta}^{\Gamma_{i-1}}(x)$ ergibt sich mit Hilfe der Dreiecksungleichung

$$\begin{aligned} \widetilde{\eta}^{\Gamma_{i-1}}(x) &= \left| \left(v_{\Gamma_{i-1}}(x) - v(x) \right) - \left(v_{\Gamma_i}(x) - v(x) \right) \right| \\ &\leq \eta^{\Gamma_{i-1}}(x) + \eta^{\Gamma_i}(x). \end{aligned}$$

Andererseits folgt aus $\eta^{\Gamma_i} < \eta^{\Gamma_{i-1}}$

$$\begin{aligned} \widetilde{\eta}^{\Gamma_{i-1}}(x) &= \left| \left(v_{\Gamma_{i-1}}(x) - v(x) \right) - \left(v_{\Gamma_i}(x) - v(x) \right) \right| \\ &\geq \eta^{\Gamma_{i-1}}(x) - \eta^{\Gamma_i}(x). \end{aligned}$$

Zusammen erhält man mit $\alpha \geq \frac{\eta^{\Gamma_i}(x)}{\eta^{\Gamma_{i-1}}(x)}$

$$\begin{aligned} \widetilde{\eta}^{\Gamma_{i-1}}(x) &\leq \eta^{\Gamma_{i-1}}(x) + \eta^{\Gamma_i}(x) \\ &= \left(1 + \frac{\eta^{\Gamma_i}(x)}{\eta^{\Gamma_{i-1}}(x)} \right) \eta^{\Gamma_{i-1}}(x) \\ &\leq (1+\alpha) \eta^{\Gamma_{i-1}}(x) \end{aligned} \tag{5.5}$$

und

$$\begin{aligned}(1-\alpha)\eta^{\Gamma_{i-1}}(x) &\leq \left(1-\frac{\eta^{\Gamma_i}(x)}{\eta^{\Gamma_{i-1}}(x)}\right)\eta^{\Gamma_{i-1}}(x)\\ &= \eta^{\Gamma_{i-1}}(x)-\eta^{\Gamma_i}(x)\\ &\leq \widetilde{\eta}^{\Gamma_{i-1}}(x).\end{aligned} \tag{5.6}$$

Aus der Ungleichung (5.5) ergibt sich

$$\frac{1}{1+\alpha}\widetilde{\eta}^{\Gamma_{i-1}}(x) \leq \eta^{\Gamma_{i-1}}(x)$$

und daraus durch Bilden des Supremums über $x \in Q_j$ die Abschätzung für $\widetilde{\eta}_j^{\Gamma_{i-1}}$.

Die erste Abschätzung für $\widetilde{\eta}^{\Gamma_{i-1}}$ folgt sofort aus diesem Resultat durch Maximumsbildung über $i = 1, \ldots, P$.

Die zweite Ungleichung für $\widetilde{\eta}^{\Gamma_{i-1}}$ erhält man aus (5.6)

$$\eta^{\Gamma_{i-1}}(x) \leq \frac{1}{1-\alpha}\widetilde{\eta}^{\Gamma_{i-1}}(x)$$

durch den Übergang zum Supremum über alle $x \in \Omega$. □

Diese Approximation 5.4 von $\eta^{\Gamma_{i-1}}(x)$ hat den Vorteil, dass man den Fehlerschätzer ohne Auswertung der Portfoliofunktion v berechnen kann. Auch wenn statt des aktuellen Fehlers nur der Fehler für $v_{\Gamma_{i-1}}$ geschätzt wird, so kann auf seiner Basis trotzdem eine sinnvolle und effiziente Verfeinerungsstrategie implementiert werden. Allerdings ist es nicht möglich, das Supremum $\sup_{x\in Q_j} \widetilde{\eta}^{\Gamma_{i-1}}(x)$ exakt zu berechnen, da die Funktion $\widetilde{\eta}^{\Gamma_{i-1}}(x)$ dazu an unendlich vielen Punkten ausgewertet werden müsste. Analog zu vorher kann aber auch hier der Wert des Fehlerschätzers $\widetilde{\eta}^{\Gamma_{i-1}}$ durch Auswertung über eine Menge von geeigneten Testpunkten approximativ bestimmt werden.

Die Fehlerschätzer können nun numerisch berechnet werden und dienen als Kriterium für verschiedene adaptive Verfeinerungsstrategien, welche zusammen mit der Verteilung der Testpunkte Thema des nächsten Kapitels sind.

Kapitel 6

Adaptive Gittererzeugung

Die Portfoliofunktion v kann nun auf einem Gitter Γ durch eine multilineare Funktion v_Γ aus dem endlichdimensionalen Funktionenraum $\mathcal{W}$ approximiert werden. Wie Lemma 5.4 gezeigt hat, hängt der Fehler der Approximation auch von der Größe des Gitterelements ab. Je kleiner der Quader, auf dem die Approximation v_Γ berechnet wird, desto genauer ist auch die Approximation. Die Idee der adaptiven Gittererzeugung ist es nun, die Größe der Quader Q_j abhängig von v möglichst effizient zu bestimmen, sie also angepasst an v zu wählen. Dies soll ohne vorherige aufwendige Analyse von v und ohne weitere Informationen, wie zum Beispiel Ableitungen, passieren. Vielmehr wird anhand eines Fehlerschätzers $\widetilde{\eta}_j$ entschieden, ob ein Quader verfeinert werden soll oder nicht. So wird sichergestellt, dass Regionen mit großen $\widetilde{\eta}_j$ so lange verfeinert werden, bis der globale Fehlerschätzer $\widetilde{\eta}$ unter eine vorgegebene Toleranzschranke *tol* fällt, während die Approximation in Regionen mit kleinen $\widetilde{\eta}_j$ gleich bleibt. Das Kapitel ist an Grüne [8] und [12] angelehnt.

Der Algorithmus zur adaptiven Gittererzeugung sieht folgendermaßen aus: Nachdem das gewünschte Approximationsgebiet Ω bestimmt ist, auf dem die Portfoliofunktion v mit einer vorgegebenen Genauigkeit *tol* approximiert werden soll, wird Ω in ein grobes Anfangsgitter Γ_0 mit P_0 Quadern Q_j und N_0 Knotenpunkten $\hat{x}_l$ unterteilt. Auf diesem Gitter wird die erste Approximation v_{Γ_0} berechnet, indem v an jedem Knotenpunkt $\hat{x}_l$ bewertet wird und $v_{\Gamma_0}(\hat{x}_l) = v(\hat{x}_l)$ für $l = 1, \ldots, N_0$ gesetzt wird. Da die Funktion $v_{\Gamma_0} \in \mathcal{W}$ eindeutig durch die Werte in den Knotenpunkten des Gitters Γ_0 bestimmt ist, reicht es aus, für jeden Quader Q_j die Werte $v_{\Gamma_0}(\hat{x}_{j_k}), k = 1, \ldots, 2^n$, der Approximation an dessen Knotenpunkten zu speichern. Zu Beginn der Iteration wird der Status aller Quader Q_j auf 2 gesetzt. Ein Status von 2 bedeutet dabei, dass die Approximation auf dem Quader weiter überprüft werden muss. Um das Gitter nun adaptiv verfeinern zu können, muss der Fehler $\eta_j^{\Gamma_i}$, der durch die Approximation v_{Γ_i} in jedem Quader Q_j von Γ_i entsteht, abgeschätzt werden. Dazu werden der Reihe nach alle Quader, deren Status 2 ist, durchlaufen und der gewählte Fehlerschätzer $\widetilde{\eta}_j^{\Gamma_i}$ an einer Reihe von Testpunkten $\widetilde{x}_{j_k}$ für $k = 1, \ldots, p$ ausgewertet. Stellt sich heraus, dass in einem Quader $\max_{k=1,\ldots,p} \widetilde{\eta}_j^{\Gamma_i}(\widetilde{x}_{j_k}) < tol$, so wird die Approximation v_{Γ_i} auf diesem Quader für gut

befunden und sein Status auf 1 gesetzt. Das bedeutet, dass der Quader nicht weiter verfeinert werden muss und auch in den darauffolgenden Iterationen nicht weiter untersucht wird. Sind alle Quader des Gitters Γ_i durchlaufen, so werden nun die Quader mit Status 2 verfeinert. Formal ist die Verfeinerung eines Quaders folgendermaßen definiert.

Definition 6.1 (Verfeinerung eines Quaders)
Sei Q_j ein Element eines Gitters Γ mit Koordinaten $a_j^i < b_j^i$, $i = 1, \ldots, n$. Dann besteht die Verfeinerung $ref(Q_j, i_0)$ von Element Q_j in Koordinatenrichtung e^{i_0}, $i_0 \in \{1, \ldots, n\}$, aus den zwei Elementen Q_{j1} und Q_{j2} mit den Koordinaten

$$Q_{j1}: \quad \begin{array}{ll} a_{j1}^i = a_j^i & \text{für } i = 1, \ldots, n \\ b_{j1}^i = b_j^i & \text{für } i = 1, \ldots, n \text{ mit } i \neq i_0 \\ b_{j1}^{i_0} = a_j^{i_0} + (b_j^{i_0} - a_j^{i_0})/2 & \end{array}$$

und

$$Q_{j2}: \quad \begin{array}{ll} b_{j2}^i = b_j^i & \text{für } i = 1, \ldots, n \\ a_{j2}^i = a_j^i & \text{für } i = 1, \ldots, n \text{ mit } i \neq i_0 \\ a_{j2}^{i_0} = a_j^{i_0} + (b_j^{i_0} - a_j^{i_0})/2. & \end{array}$$

$ref(Q_j)$ bezeichnet die Verfeinerung in alle Koordinatenrichtungen, die für einen n-dimensionalen Quader aus 2^n neuen Elementen besteht. Die Verfeinerung des ganzen Gitters wird mit $ref(\Gamma)$ bezeichnet und die m-te Verfeinerung mit $ref^m(\Gamma)$, wobei

$$\begin{aligned} ref^1(\Gamma) &= ref(\Gamma) \\ ref^{m+1}(\Gamma) &= ref(ref^m(\Gamma)). \end{aligned}$$

Die Verfeinerung eines Quaders erfolgt also iterativ durch gleichmäßige Teilung in je eine Koordinatenrichtung, so dass zwei neue Elemente entstehen. Auf diesen beiden kleineren Elementen soll nun eine bessere Approximation von v berechnet werden. An den neu entstandenen Knotenpunkten wird $v_{\Gamma_{i+1}}$ durch Auswertung von v neu berechnet. Die Werte von $v_{\Gamma_{i+1}}$ an den Knotenpunkten, die vom Vaterquader vererbt wurden, können übernommen werden. Auf Quadern mit Status 1, die nicht weiter verfeinert werden, bleibt die Approximation gleich. Das Ergebnis der Verfeinerung aller Quader mit Status 2 ist ein lokal verfeinertes Gitter Γ_{i+1}. Diese Prozedur wird so lange iterativ wiederholt bis $\widetilde{\eta}_j^{\Gamma_i} < tol$ für alle $j = 1, \ldots, P_i$. Hat am Ende einer Iteration kein Quader mehr Status 2, so ist die Approximation v_{Γ_i} von v auf ganz Ω hinreichend gut und der Algorithmus kann gestoppt werden. Alternativ zur Angabe einer Fehlertoleranz *tol* könnte auch eine maximale Verfeinerungstiefe oder eine maximale Anzahl von Elementen als Abbruchkriterium dienen.

Diese Vorgehensweise führt auf folgenden Algorithmus, der die Basis für die in den nächsten Abschnitten behandelten Verfeinerungsstrategien ist.

Algorithmus 6.2 (adaptive Gittererzeugung)
Eingabe: *Approximationsgebiet* Ω, *Fehlertoleranz* $tol > 0$

(1) *Erzeuge Anfangsgitter* Γ_0 *auf* Ω
Berechne Approximation v_{Γ_0}
Setze den Status aller Quader auf 2
Setze $i := 0$

(2) *Durchlaufe alle Quader* Q_j *des Gitters* Γ_i *mit Status = 2*
Durchlaufe alle Testpunkte $\widetilde{x}_{j_k}$ *des Quaders* Q_j
Berechne den Fehlerschätzer $\widetilde{\eta}_j^{\Gamma_i}(\widetilde{x}_{j_k})$
Falls $\max\limits_{k=1,\dots,p} \widetilde{\eta}_j^{\Gamma_i}(\widetilde{x}_{j_k}) < tol$, *setze Status = 1*

(3) *Falls alle Quader Status 1 haben, STOP*

(4) *Verfeinere alle Quader mit Status 2 zu neuem Gitter* Γ_{i+1}

(5) *Berechne Approximation* $v_{\Gamma_{i+1}}$

(6) *Setze* $i := i + 1$
Gehe zu (2)

Ausgabe: *Gitter* Γ_i *mit Approximation* v_{Γ_i}

6.1 Verfeinerung in alle Koordinatenrichtungen

Die zunächst einfachste Art der Verfeinerung besteht darin, die Elemente mit Status 2 immer in eine Koordinatenrichtung zu verfeinern, wobei sich die Richtungen zyklisch abwechseln, in 3d also x - y - z - x - y - .... Die Verfeinerungsrichtung $dir \in \{1, \dots, n\}$ ergibt sich dabei aus dem Iterationsindex i und der Dimension n mit Hilfe der Modulo-Rechnung

$$dir = i \ mod \ n. \tag{6.1}$$

Theoretisch könnte man auch in jeder Iteration in alle Koordinatenrichtungen unterteilen, so dass aus einem Quader nicht 2, sondern 2^n Teilquader entstehen, und dabei dasselbe Resultat erhalten. Es kann aber passieren, dass man öfter verfeinert als nötig und damit die Approximationsfunktion unnötig „aufbläst“.

Wie im letzten Kapitel bereits angedeutet wurde, kann die Performance des Algorithmus durch eine effiziente Wahl der Testpunkte optimiert werden. Wird der a-posteriori Fehlerschätzer nach Definition 5.6 unter Verwendung von v berechnet, so muss die Portfoliofunktion v an allen Testpunkten des aktuellen Gitters Γ ausgewertet werden. Die Idee ist nun, die Testpunktmenge $tp(Q_j)$ eines Quaders Q_j so zu wählen, dass sie die Eckpunkte für eine mögliche Verfeinerung $ref(Q_j)$ bereits beinhaltet, also

$$(nodes(ref(Q_j)) \backslash nodes(Q_j)) \subseteq tp(Q_j).$$

Der Wert von v an diesen Testpunkten kann dann für die Berechnung der neuen Approximation $v_{ref(\Gamma)}$ wiederverwendet werden. In der numerischen Praxis haben sich für zwei- und dreidimensionale Gitter die in Abbildung 6.1 angezeigten Punkte als geeignet erwiesen. Dieses Muster lässt sich leicht auf höhere Dimensionen verallgemeinern. Durch diese Wahl können die

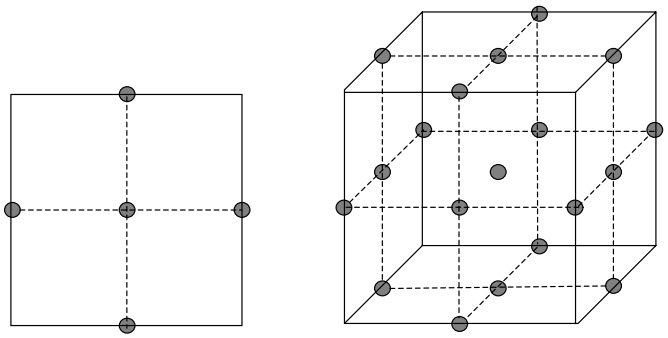

Abbildung 6.1: Testpunktmengen bei Verfeinerung in alle Koordinatenrichtungen in 2d und 3d

Werte der Portfoliofunktion, die für die Auswertung des Fehlerschätzers $\widetilde{\eta}_j$ an den Testpunkten berechnet wurden, für die nächste Iteration als Werte an den Gitterpunkten weiterverwendet werden. Jedoch wächst die Anzahl der Testpunkte pro Quader mit der Dimension n enorm an, wie Tabelle 6.1 zeigt.

n	2	3	4	5	6	7	8
Testpunkte pro Quader	5	19	65	211	665	2059	6305

Tabelle 6.1: Anzahl der Testpunkte pro Quader

Durch die Auswertung der Portfoliofunktion v an jedem Testpunkt des Quaders kann gerade bei höheren Dimensionen erheblicher Aufwand entstehen. Eine geeignete Alternative ist deshalb die Berechnung des Fehlerschätzers nach Definition 5.9. Dazu wird die aktuelle Approximation v_{Γ_i} mit der vorherigen Approximation $v_{\Gamma_{i-1}}$ verglichen, ohne v selbst auswerten zu müssen. Der Trick dabei ist, dass die Testpunkte des Vorgängerquaders ja gerade die neuen Eckpunkte der Verfeinerung sind, an denen der Wert von v schon exakt berechnet wurde. Der Fehlerschätzer wird also in den Testpunkten des Gitters Γ_{i-1} durch Vergleich der alten Approximation $v_{\Gamma_{i-1}}$ mit der aktuellen Approximation v_{Γ_i} bestimmt. Wurde immer nur in eine Koordinatenrichtung verfeinert, so muss man jedoch in der Konstruktion des Gitters nicht nur einen Schritt, sondern n Schritte zurückgehen.

6.2 Anisotrope Verfeinerung

Unter anisotroper Verfeinerung versteht man die richtungsabhängige Zerlegung eines Gitters, d.h. ein Quader wird nicht in alle Koordinatenrichtungen nacheinander, sondern abhängig vom Wert des Fehlerschätzers nur in bestimmte Koordinatenrichtungen unterteilt. Da die Portfoliofunktion bezüglich der Risikofaktoren unterschiedliche Sensitivitäten aufweist, auf Veränderungen also unterschiedlich stark reagiert, ist es sinnvoll, manche Koordinatenrichtungen feiner zu unterteilen als andere. Durch die anisotrope Verfeinerung kann das Gitter also besser auf die Struktur der Portfoliofunktion abgestimmt werden.

Der Fehlerschätzer wird dafür nicht wie bei der Verfeinerung in alle Koordinatenrichtungen auf dem ganzen Quader, sondern in jeder Iteration nur für eine Koordinatenrichtung $dir \in \{1, \ldots, n\}$ berechnet. Die Richtung dir ergibt sich analog zu (6.1) aus dem Iterationsindex i und der Dimension n mittels

$$dir = i \ mod \ n.$$

Für jede Richtung dir wird eine eigene Testpunktmenge $tp_{dir}(Q_j)$ betrachtet, in der unter anderem die potenziellen neuen Knoten, die zum Gitter dazukommen, falls das Element Q_j in Koordinatenrichtung e^{dir} unterteilt wird, enthalten sind. Wählt man diese Testpunktmengen analog zur Verfeinerung in alle Koordinatenrichtungen, so ergeben sich für 2d-Elemente die in Abbildung 6.2 und für 3d-Elemente die in Abbildung 6.3 gezeigten Punkte. Im Laufe der Arbeit hat sich jedoch herausgestellt, dass durch eine Auswertung des Fehlerschätzers auf einer Testpunktmenge, die nur aus den potenziellen neuen Knoten besteht, besonders gute und schnelle Ergebnisse erzielt werden, da der Einfluss der einzelnen Risikofaktoren auf das Portfolio so besser isoliert werden kann. In den Abbildungen 6.2 und 6.3 sind diese Punkte schwarz eingezeichnet. In der Auswertung in Kapitel 8 wurden deshalb diese Testpunktmengen benutzt.

Definiert man nun den Fehler für die verschiedenen Koordinatenrichtungen eines Elements durch

$$\widetilde{\eta}_{dir,j} := \max_{x \in tp_{dir}(Q_j)} \widetilde{\eta}_j(x),$$

so wird der Quader Q_j nacheinander in alle Richtungen e^{dir} mit $\widetilde{\eta}_{dir,j} > tol$ verfeinert.

Analog zur Verfeinerung in alle Koordinatenrichtungen lässt sich auch diese Verfeinerungsstrategie sowohl mit der Fehlerschätzung nach Definition 5.6 unter Verwendung von v als auch mit der Fehlerschätzung nach Definition 5.9 ohne Verwendung von v kombinieren.

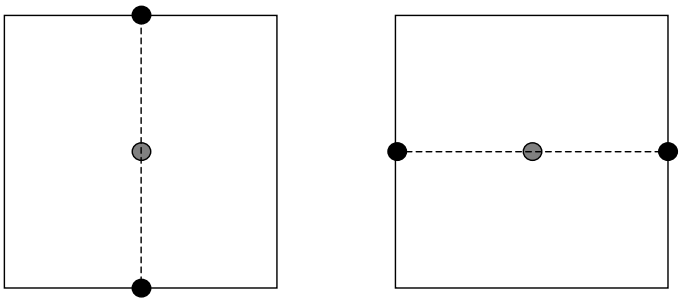

Abbildung 6.2: Testpunktmengen bei anisotroper Verfeinerung in 2d

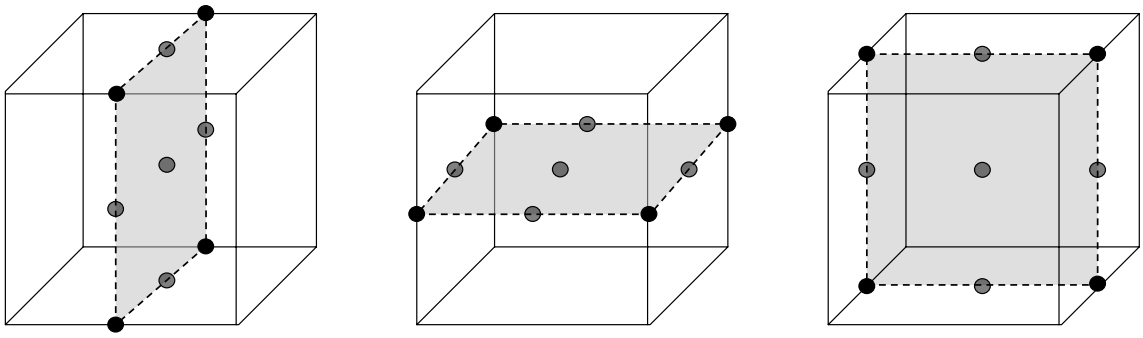

Abbildung 6.3: Testpunktmengen bei anisotroper Verfeinerung in 3d

Kapitel 7

Implementierung

In diesem Kapitel soll nun die Implementierung der in den letzten drei Kapiteln diskutierten Approximation der Portfoliofunktion mit Hilfe von adaptiver Gittererzeugung beschrieben werden. Der Algorithmus 6.2 wurde in C bzw. C++ realisiert und ist im Anhang B abgedruckt. Für die grafische Darstellung der Portfoliofunktion und der Gitterstruktur wurden MATLAB-Routinen verwendet, die sich im Anhang C befinden.

7.1 Steuerung des Algorithmus

Gesteuert wird der Algorithmus durch die Funktion `int main(int argc, char **argv)`, die im Modul `main.cpp` enthalten ist. Der Aufruf des Programms erfolgt mit den Parametern

`products`	Die Datei `products` enthält Informationen zu den im Portfolio enthaltenen Produkten.
`riskfactors`	In der Datei `riskfactors` sind die Daten der Risikofaktoren gespeichert.
`shifts`	Die Angaben zu den Veränderungen der Risikofaktoren befinden sich in der Datei `shifts`.
`ref_type`	`ref_type` gibt die Art der Verfeinerung an, wobei `0` eine Verfeinerung in alle Koordinatenrichtungen nach Abschnitt 6.1 und `1` eine anisotrope Verfeinerung nach Abschnitt 6.2 bewirkt.
`err_type`	`err_type` gibt die Art der Fehlerschätzung an. Wird dieser Wert auf `0` gesetzt, so wird die Fehlerschätzung unter Verwendung von v auf dem aktuellen Gitter nach Abschnitt 5.2.1 durchgeführt. Ist er dagegen `1`, so wird der Fehlerschätzer ohne Verwendung von v auf dem Vorgängergitter nach Abschnitt 5.2.2 berechnet.

`approx_type`	`approx_type` gibt die Art der Approximation an. Bei `0` wird die Portfoliofunktion selbst approximiert, bei `1` deren Gewinn- und Verlustverteilung.
`tol`	`tol` gibt die gewünschte Fehlertoleranz an, mit der die Approximation berechnet werden soll.

- `int main(int argc, char **argv)`

  Zu Beginn des Algorithmus werden die Produktinformationen, Marktdaten und Shifts eingelesen und die Bewertung des Portfolios vorbereitet. Dazu werden jedem Produkt die entsprechenden Risikofaktoren und jedem Risikofaktor der passende Shift zugeordnet. Sollten für ein Produkt die passenden Risikofaktoren nicht gefunden werden, wird es später bei der Bewertung des Portfolios nicht berücksichtigt. Risikofaktoren, für die kein passender Shift angegeben ist, werden später nicht verändert.
  Anschließend wird ein Anfangsgitter Γ_0 auf dem durch die Shifts bestimmten Gebiet Ω erzeugt und die Werte von v_{Γ_0} an den Knotenpunkten berechnet. Im weiteren Verlauf des Algorithmus wird iterativ mit Hilfe des Fehlerschätzers und der adaptiven Verfeinerung eine Approximation v_{Γ_i} von v berechnet, die überall in Ω der Fehlertoleranz `tol` genügt.
  In jeder Iteration werden Angaben zur Quaderanzahl, zum Wert des globalen Fehlerschätzers und zur Rechenzeit automatisch in die Datei

  `info_ref_type_err_type_approx_type_tol.txt`

  geschrieben. Die Approximation auf dem Gitter wird zum Schluss mit Hilfe der Funktion `write_grid` in die Datei

  `grid_ref_type_err_type_approx_type_tol.txt`

  ausgegeben.

7.2 Dateneingabe

Die Dateneingabe besteht aus den drei Dateien `products`, `riskfactors` und `shifts`, die alle benötigten Informationen zur Bewertung des gewünschten Portfolios enthalten. Das Einlesen der Daten erfolgt über nachstehende Funktionen, die sich ebenfalls im Modul `main.cpp` befinden und zu Beginn des Algorithmus von `int main(int argc, char **argv)` aufgerufen werden:

- `void get_products(char *products)`

  Liest die Produktinformationen aus der Datei `products` ein.

- `void get_riskfactors(char *riskfactors)`

  Liest die Daten der Risikofaktoren aus der Datei `riskfactors` ein.

- `void get_shifts(char *shifts)`

  Liest die Angaben zu den Veränderungen der Risikofaktoren aus der Datei `shifts` ein.

Die Informationen zu den Produkten, den Marktdaten und den Shifts werden dem Programm in Form von `.txt`-Dateien übergeben. Dabei muss eine gewisse Struktur eingehalten werden, die im Folgenden beschrieben wird.

7.2.1 Eingabe der Produktinformationen

	type	ccy_id	s_id	n
Aktie	1	USD	DJ	500

	type	ccy_id	ir_id	frn	[frnt]	c[1]	...	c[k]	t[1]	...	t[k]
Zinsprodukt	2	USD	USDSWAP	1	0.5	1000	...	2000	0.1	...	10

	type	ccy_id	u_id	sigma_id	r_id	k	mat	cp	ls
Option	3	EUR	DAX	DAX	EURSWAP	7500.0	1.5	1	1

Tabelle 7.1: Eingabe der Produktinformationen

Bei der Eingabe der Produktinformationen wird zwischen Aktien, Zinsprodukten und Optionen unterschieden. Für jedes Produkt ist in der Datei `products` eine eigene Zeile angelegt. In Tabelle 7.1 sind die zur Bewertung benötigten Angaben mit Beispielen belegt. Bei jedem Produkt wird zunächst der Typ `type` eingelesen, anhand dessen später entschieden wird, welche weiteren Parameter eingelesen werden sollen. Dabei zeigt der Wert `1` an, dass es sich um eine Aktie handelt, der Wert `2` initialisiert ein Zinsprodukt und der Wert `3` steht für eine Option. Weiterhin wird für jedes Produkt dessen Währung `ccy_id` angegeben, bevor die produktspezifischen Daten eingelesen werden. Für Aktien sind dies der Name der Aktie bzw. des Aktienindex `s_id` und die Anzahl der Aktien `n`. Bei Zinsprodukten wird zunächst der Name der zugrunde liegenden Zinskurve `ir_id` übergeben und ob die Position eine Floating-Rate-Note `frn` ist (`1`) oder nicht (`0`). Handelt es sich um eine Floating-Rate-Note, so wird zusätzlich ein optionaler Parameter `frnt` eingelesen, der angibt, ab welcher Laufzeit die Zinskurve flexibel ist. Es folgen die Zinszahlungen `c[1],...,c[k]`, deren Anzahl flexibel ist, und deren Fälligkeit `t[1],...,t[k]` in Jahren. Für die Bewertung von Optionen sind die fünf Parameter Name des Underlyings `u_id`, Name der Volatilität `sigma_id`, Name des sicheren Zinssatzes `r_id`, Strike `k` und Laufzeit `mat` von Bedeutung, die in dieser Reihenfolge eingelesen werden. Zusätzlich muss durch den Parameter `cp` angegeben sein, ob es sich um einen Call (`1`) oder einen Put (`0`) handelt, und durch den Parameter `ls`, ob sich die Bank in der Long- (`1`) oder in der Short-Position (`0`) befindet.

7.2.2 Eingabe der Marktdaten

	type	id	map_id	val
Aktienkurs	1	DJ	DAX	12075.96
Wechselkurs	3	USD_EUR	USD_EUR	0.75528
Volatilität	4	DJ	DAX	0.21

	type	id	mapst_id	mapmt_id	maplt_id	r[1] ... r[k]	t[1] ... t[k]
Zinskurve	2	SWAP	SWAP3m	SWAP5y	SWAP10y	0.02153 ... 0.04285	0.1 ... 30

Tabelle 7.2: Eingabe der Marktdaten

Um die drei oben genannten Produktarten zu bewerten, werden grundsätzlich die vier Risikofaktorgruppen Aktie bzw. Aktienindex, Zinskurve, Wechselkurs und Volatilität benötigt. In der Datei `riskfactors` wird wiederum jeder Risikofaktor in einer eigenen Zeile übergeben. In Tabelle 7.2 wird die Struktur der Risikofaktoren aufgezeigt. `type` legt die Art des Risikofaktors fest. Dabei steht `1` für Aktien- bzw. Aktienindexkurs, `2` für Zinskurve, `3` für Wechselkurs und `4` Volatilität. Je nach Art des Risikofaktors werden verschiedene Werte benötigt. Zunächst wird jedoch für jeden Risikofaktor dessen Name `id` eingelesen. Anschließend folgt der Name `map_id` des Risikofaktors, auf den er bei der Auswahl der Shifts gemappt werden soll. Zinskurven nehmen hier eine Sonderstellung ein, da die kurzfristigen, die mittelfristigen und die langfristigen Bereiche unabhängig voneinander geshiftet werden können und daher die drei Namen `mapst_id, mapmt_id` und `maplt_id` übergeben werden müssen. Bei Aktienkursen, Wechselkursen und Volatilitäten wird zuletzt der aktuelle Wert `val` des Risikofaktors eingelesen, während bei Zinsen eine ganze Zinskurve `r[1],...,r[k]` zusammen mit den Laufzeiten `t[1],...,t[k]` benötigt wird.

7.2.3 Eingabe der Shifts

	type	id	mod	xu	xo
Aktienkurs	1	DAX	1	-0.3	0.3
Zinskurve	2	USDSWAP	0	-0.2	0.3
Wechselkurs	3	USD_EUR	1	-0.15	0.2
Volatilität	4	DAX	1	-0.4	0.4

Tabelle 7.3: Eingabe der Shifts

Tabelle 7.3 zeigt beispielhaft die Eingabe der Shifts. Auch hier steht jeder Shift in einer eigenen Zeile. Der erste Eintrag `type` identifiziert die Art des zu verändernden Risikofaktors. 1 steht dabei für Aktien- bzw. Aktienindexkurs, 2 für Zinskurve, 3 für Wechselkurs und 4 für Volatilität. Der Name `id` des jeweiligen Risikofaktors steht an zweiter Stelle, während `mod` angibt, ob die Veränderung absoluter (0) oder relativer (1) Natur ist. Dementsprechend wird

zum Schluss das absolute oder relative Intervall `[xu; xo]`, innerhalb dessen ein Risikofaktor geshiftet werden kann, eingelesen.

7.3 Produktbewertung

Die Routinen zur Produktbewertung befinden sich im Modul `valuation.cpp` bzw. `valuation.h`.

- `void assign_rf(void)`

  Ordnet jedem Produkt die richtigen Risikofaktoren zu. Dabei wird der Index des jeweiligen Risikofaktors beim Produkt gespeichert, so dass später eine schnellere Bewertung möglich ist. Wird der passende Risikofaktor nicht gefunden, so wird der Index auf `-1` gesetzt. Zusätzlich erhält der Parameter `flag`, der für jedes Produkt angibt, ob alle zugehörigen Risikofaktoren zugeordnet werden konnten, den Wert `0`.

- `void assign_shift(void)`

  Sucht für jeden Risikofaktor den zugehörigen Shift heraus und speichert dessen Index in einer eigens dafür angelegten Variablen. Einer Zinskurve können drei verschiedene Shifts für den kurz-, mittel- und langfristigen Bereich zugeordnet werden. Wird der passende Shift nicht gefunden oder soll der Risikofaktor gar nicht geshiftet werden, so wird der Index auf `-1` gesetzt.

- `double get_rf(double rf_act, int i, double *x)`

  Gibt den Wert des Risikofaktors `rf_act` nach Berücksichtigung des Shifts `i` im Marktzustand `x` zurück. Möglich ist eine relative oder eine absolute Veränderung. Ist `i` negativ, so wird der Risikofaktor nicht geshiftet.

- `double get_fx(double value, int i, double *x)`

  Prüft, ob `value` bereits in `EUR` ist. Falls nicht, wird der Wert in `EUR` unter Berücksichtigung der Wechselkursveränderung `i` im Marktzustand `x` zurückgegeben.

- `double stock_val(int i, double *x)`

  Berechnet den Wert der Aktienposition `i` im Marktzustand `x`.

- `double interest_val(int i, double *x)`

  Berechnet den Wert der Zinsposition `i` im Marktzustand `x` durch Abdiskontierung. Findet sich zu einer Zinszahlung in der zugeordneten Zinskurve keine Stützstelle mit gleicher Laufzeit, so wird zwischen den beiden angrenzenden Stützstellen linear interpoliert. Entstehen durch die Veränderung der Zinskurve negative Zinssätze, so werden diese automatisch auf 0 gesetzt.

- `double option_val(int i, double *x)`

  Berechnet den Wert der europäischen Option `i` im Marktzustand `x` mit Hilfe der Black-Scholes-Formel.

- `double portfolio_val(double *x)`

  Gibt den exakten Wert des Portfolios im Marktzustand `x` zurück. Dazu werden alle Produkte durchlaufen und die dazugehörigen Bewertungsfunktionen aufgerufen. Ist ein Produkt unbekannt oder konnten nicht alle benötigten Risikofaktoren zugeordnet werden, so wird der Portfoliowert ohne dieses Produkt berechnet und eine Warnung ausgegeben.

- `double portfolio_approx(cube *that, double *x, double *xl, double *xw)`

  Gibt den approximativen Wert des Portfolios im Marktzustand `x` zurück, der durch multilineare Interpolation aus den Werten an den Eckpunkten des Elements `that`, in dem der Punkt `x` liegt, berechnet wird.

7.4 Gittererzeugung und -verwaltung

Die Routinen zur Gittererzeugung und -verwaltung sind im Modul `grid.c` bzw. `grid.h` enthalten. Bei der Implementierung konnte auf bereits vorhandene Routinen von Prof. Dr. Lars Grüne zurückgegriffen werden. Diese wurden teilweise abgeändert und ergänzt, um der veränderten Aufgabenstellung Rechnung zu tragen. Im Folgenden werden die Routinen einzeln beschrieben. Diese Dokumentation ist zum Teil dem Anhang von [10] entnommen.

- `double make_grid(double *xu, double *xo, int dim, int n)`

  Erzeugt ein Gitter im $\mathbb{R}^{dim}$ auf der quaderförmigen Menge Ω mit „linker unterer“ Ecke `xu` und „rechter oberer“ Ecke `xo`. In jeder Koordinatenrichtung werden `n` Quader erzeugt, also `n`·`dim` Quader insgesamt. Falls `n` keine Zweierpotenz ist, wird automatisch abgerundet. Die Werte der Approximation an den Knotenpunkten werden berechnet und den Quadern zugewiesen. Als Rückgabewert gibt die Funktion den Durchmesser der Elemente in der 2–Norm zurück. Jedes Element wird mit Status 2 vorbelegt.

- `int write_grid(char *file)`

  Schreibt das aktuelle Gitter in die Datei mit dem Namen `file`. Nach einer Kommentarzeile mit der Dimension des Gitters und den Namen der Risikofaktoren wird für jedes Element eine Zeile mit linker unterer und rechter oberer Ecke, den Werten der Approximation an den Eckpunkten sowie dem Status ausgegeben. Das Gitter und die approximierte Funktion können mit Hilfe der Datei durch die unten beschriebenen MATLAB–Routinen grafisch dargestellt werden.

- `int first_cell(void)`

  Setzt den internen Zeiger auf das erste Element des Gitters. Gibt `-1` zurück, falls noch kein Gitter erzeugt wurde, sonst `0`. Das Element, auf das der interne Zeiger zeigt, wird im Folgenden als aktuelles Element bezeichnet.

- `int next_cell(void)`

  Setzt den internen Zeiger auf das nächste Element des Gitters. Gibt eine fortlaufende positive Nummer, den Index des Elements, zurück und `-1`, falls das letzte Element des Gitters bereits erreicht war.

- `double get_err(double *val_sep)`

  Gibt den Wert des Fehlerschätzers auf dem aktuellen Element zurück. Je nach Art der Fehlerschätzung werden dazu entweder die Testpunkte des aktuellen Gitters erzeugt und der Fehler unter Verwendung der Portfoliofunktion ausgewertet oder der Fehler an den Testpunkten des Vorgängergitters ohne Verwendung der Portfoliofunktion berechnet. Wird im Zuge der Fehlerschätzung der Wert des Portfolios an einem potenziellen neuen Knoten berechnet, so wird der Wert in `val_sep` gespeichert.

- `void set_err(cube *that, double err)`

  Setzt den Fehler des Elements `that` auf `err`.

- `double get_newval(cube *that, double *xl, double *xw, double *val)`

  Speichert die Werte der Portfoliofunktion an den Punkten, die bei einer Verfeinerung des Elements `that` in die aktuelle Verfeinerungsrichtung zur Approximation dazukommen, in `val` und gibt den Approximationsfehler in diesen Punkten zurück.

- `void refine_grid(void)`

  Verfeinert jeden Quader des Gitters mit Status 2. Hierbei wird immer in eine Koordinatenrichtung verfeinert. Je nach Art der Verfeinerung wechseln sich die Richtungen zyklisch ab oder werden durch den Wert des Fehlerschätzers in den einzelnen Richtungen bestimmt.

- `void get_corner(int i, double *x)`

  Gibt den `i`–ten Eckpunkt des aktuellen Elements aus. Der Index `i` muss zwischen `0` und $2^{dim} - 1$ liegen, die Variable `x` muss entsprechend der Dimension deklariert sein.

- `void get_testpoint(int i, double *x)`

  Gibt den `i`–ten Testpunkt (von insgesamt `NUM_TP`) des aktuellen Elements aus. Der Index `i` muss zwischen `0` und `NUM_TP - 1` liegen, die Variable `x` muss entsprechend der Dimension deklariert sein. Die Testpunktanzahl `NUM_TP` wird dabei durch die Art der Verfeinerung bestimmt.

- `int set_status(int status)`

  Setzt den Status des aktuellen Elements bzw. bei anisotroper Verfeinerung des Status der aktuellen Verfeinerungsrichtung auf `status`. Dieser Wert muss dabei ≥ 0 sein. Anschließend wird bei anisotroper Verfeinerung der Status des Elements, der sich als maximaler Status aller Verfeinerungsrichtungen ergibt, aktualisiert. Gibt bei Erfolg `0` zurück, sonst `-1`.

- `int get_status(void)`

  Gibt den Status des aktuellen Elements zurück bzw. `-1`, falls der interne Zeiger nicht gesetzt wurde.

- `int get_status_dir(void)`

  Gibt bei anisotroper Verfeinerung den Status der aktuellen Verfeinerungsrichtung, ansonsten den Status des aktuellen Elements zurück bzw. `-1`, falls der interne Zeiger nicht gesetzt wurde.

- `void set_refdir(cube *that, int dir)`

  Setzt die Verfeinerungsrichtung des Elements `that` auf `dir`.

- `unsigned char get_refdir(cube *that)`

  Gibt die Verfeinerungsrichtung des Elements `that` zurück bzw. `0`, falls das Element noch gar nicht verfeinert wurde.

7.5 Nachträgliche Auswertung der Approximation

Wurde die Approximation v_Γ auf dem Gitter Γ berechnet und wie oben beschrieben mit der Anweisung `write_grid(,,file“)` in der Datei `file` gespeichert, so kann der approximative Portfoliowert mit Hilfe von `post_eval.cpp` auch nachträglich an jedem beliebigen Punkt `x` in Ω berechnet werden. `post_eval.cpp` funktioniert im Prinzip wie die multilineare Interpolation, die auch während der adaptiven Gittererzeugung zum Einsatz kommt, ist aber ein eigenständiges Programm zur nachträglichen Auswertung des Portfolios. Der Programmaufruf erfolgt mit folgenden Parametern:

`file`	In der Datei `file` ist die Approximation auf dem Gitter gespeichert. Sie wurde im Algorithmus durch den Befehl `write_grid(,,file“)` erzeugt und enthält außerdem Informationen zur Dimension des Gitters sowie den Namen der Risikofaktoren.
`x[1],...,x[dim]`	Punkt, an dem der approximative Portfoliowert berechnet werden soll.

- `double portfolio_approx(double *x, double *xu, double *xo, double *val, int dim)`

Gibt den approximativen Wert des Portfolios im Marktzustand `x` zurück, der durch multilineare Interpolation aus den Werten an den Eckpunkten des Elements, in dem der Punkt `x` liegt, berechnet wird.

- `int main(int argc, char **argv)`

  Sucht aus der übergebenen Datei `grid` das Element heraus, das `x` enthält und gibt den approximativen Wert des Portfolios an der Stelle `x` aus.

7.6 Datenstruktur

Nachdem die einzelnen Module der Implementierung beschrieben wurden, soll kurz die Datenstruktur erläutert werden. Ausgehend von einem äquidistanten Anfangsgitter Γ_0 wird jede Verfeinerung intern in einem hierarchischen binären Baum gespeichert. Das tatsächlich verwendete Gitter ist dabei immer die Verfeinerung auf der untersten Ebene. Es besteht also gerade aus denjenigen Elementen, die die sogenannten Blätter des binären Baums bilden. Abbildung 7.1 stellt diese Datenstruktur schematisch dar, das aktuelle Gitter besteht hier aus den drei Elementen Q_{100}, Q_{101} und Q_{11}. Weiterführende Informationen zu dieser Datenstruktur finden sich in Grüne, Metscher, Ohlberger [7].

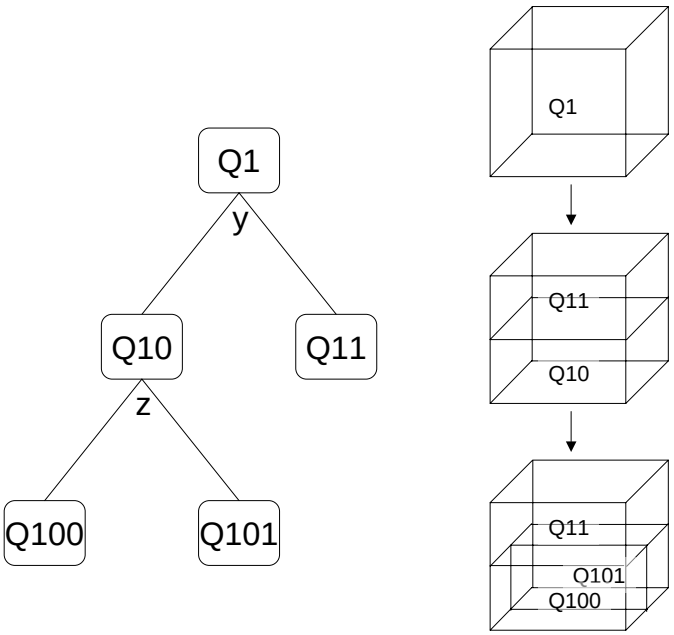

Abbildung 7.1: Schematische Darstellung der Baumstruktur

Speziell im Hinblick auf die flexible Verfeinerung der Elemente ist diese Datenstruktur sinnvoll, da sie einen schnellen Zugriff auf die einzelnen Quader erlaubt, ohne die Koordinaten

der Eckpunkte bei jedem Quader mit abzuspeichern. Diese Informationen werden gegebenenfalls rekursiv rekonstruiert, während der Baum durchlaufen wird. Dadurch kann der adaptive Verfeinerungsalgorithmus aus Kapitel 6 effizient implementiert werden. Der Vorteil dieser Speicherung liegt also im geringen Speicherplatzbedarf sowie in der effizienten Umsetzung der Routinen.

Bei der Verwendung der Routinen ist diese Datenstruktur unsichtbar, sie erklärt aber die auf den ersten Blick etwas ungewöhnliche Handhabung der Routinen, die sich einfach daher ergibt, dass die Quader nicht in einem Array, sondern in einem Baum abgespeichert sind.

7.7 Visualisierung

Zur grafischen Darstellung der Ergebnisse des Algorithmus können die MATLAB-Routinen `surfaceplot.m` und `gridplot.m` verwendet werden. Mit ihnen wurden auch die Abbildungen aus Kapitel 8 erzeugt.

7.7.1 Darstellung der Portfoliofunktion

Mit Hilfe des Programms `surfaceplot.m` kann die Approximation der Portfoliofunktion grafisch dargestellt werden. Da die Funktion auf einem mehrdimensionalen Raum $\Omega \subset \mathbb{R}^n$ definiert ist, ist eine komplette Darstellung in Abhängigkeit aller Risikofaktoren nicht möglich. Die Portfoliofunktion kann immer nur in Abhängigkeit von zwei Risikofaktoren geplottet werden, während die Werte der restlichen Risikofaktoren festgeschrieben werden. An das Programm müssen folgende Parameter übergeben werden:

`file`	Die Datei `file` enthält in einer Kommentarzeile am Anfang die Dimension und die Namen der Risikofaktoren. Danach werden für jedes Element des Gitters die linke untere und die rechte obere Ecke, die Portfoliowerte an den Knotenpunkten sowie der Status ausgegeben. Sie wird durch den Befehl `write_grid(„file“)` im Algorithmus automatisch erzeugt.
`x`	erster variabler Risikofaktor
`y`	zweiter variabler Risikofaktor
`varargin`	`varargin` besteht aus zwei Arrays variabler Länge, wobei im ersten Array alle Risikofaktoren aufgelistet sind, die nicht variiert werden. Im zweiten Array werden die Werte der Risikofaktoren übergeben und zwar in derselben Reihenfolge wie im ersten Array. Sind nur zwei Risikofaktoren vorhanden, bleiben die Arrays leer.

Der Aufruf `surfaceplot('approx.txt',3,5,[1,2,4],[0.1,200,-0.3])` generiert einen Plot der in `approx.txt` gespeicherten Portfoliofunktion in Abhängigkeit von den Risikofaktoren `3` und `5`, wobei Risikofaktor `1` den Wert `0.1`, Risikofaktor `2` den Wert `200` und Risikofaktor `4`

den Wert `-0.3` annimmt.

7.7.2 Darstellung des Gitters

Das Gitter, das im Rahmen der Approximation adaptiv verfeinert wurde, kann mit Hilfe der Routine `gridplot.m` zwei- bzw. dreidimensional dargestellt werden. Ist das Gitter höherdimensional, werden für die restlichen Risikofaktoren feste Werte angegeben. Das Programm ist eine Erweiterung der zwei Programme `gridplot2d.m` und `gridplot3d.m`, die in [10] speziell für zwei- und dreidimensionale Gitter entwickelt wurden.

Der Aufruf erfolgt analog zu vorher mit den Parametern

`file`	Die Datei `file` enthält in einer Kommentarzeile am Anfang die Dimension und die Namen der Risikofaktoren. Danach werden für jedes Element des Gitters die linke untere und die rechte obere Ecke, die Portfoliowerte an den Knotenpunkten sowie der Status ausgegeben. Sie wird durch den Befehl `write_grid(„file“)` im Algorithmus automatisch erzeugt.
`x`	erster variabler Risikofaktor
`y`	zweiter variabler Risikofaktor
`z`	dritter variabler Risikofaktor
`varargin`	`varargin` besteht aus zwei Arrays variabler Länge, wobei im ersten Array alle Risikofaktoren aufgelistet sind, die nicht variiert werden. Im zweiten Array werden die Werte der Risikofaktoren übergeben und zwar in derselben Reihenfolge wie im ersten Array. Sind nur zwei oder drei Risikofaktoren vorhanden, bleiben die Arrays leer.

Die Anweisung `gridplot('approx.grd',3,5,1,[2,4],[200,-0.3])` erzeugt einen Plot des dreidimensionalen Gitters an der Stelle, an der Risikofaktor `2` den Wert `200` und Risikofaktor `4` den Wert `-0.3` annimmt. Ist das Gitter nur zweidimensional, so wird der dritte variable Risikofaktor `z=0` gesetzt. Elemente mit Status 2 werden rot und die restlichen Elemente durchsichtig gezeichnet.

Kapitel 8

Beispiele und Ergebnisse

In diesem Kapitel soll nun die in Kapitel 7 vorgestellte Implementierung anhand von drei Beispielen aus der Praxis getestet werden. Die folgenden Ergebnisse wurden mit Hilfe der in Anhang B abgedruckten Programme in C und C++ berechnet und durch die MATLAB-Routinen aus Anhang C grafisch dargestellt. Zur Auswertung wurde ein PC mit einem AMD Athlon-Prozessor mit 2600 MHz und 512 MByte RAM, betrieben von einem Windows XP-System, benutzt.

8.1 Gemischtes Portfolio

Im ersten Beispiel soll die Entstehung des Gitters im Laufe des Algorithmus untersucht werden. Betrachtet wird ein gemischtes Portfolio aus Aktien, Optionen und einem Zinsprodukt. Zur Bewertung werden drei Risikofaktoren benötigt, der Aktienindex EUROSTOXX, dessen Volatilität EUROSTOXX_VOL sowie die Zinskurve EURSWAP. Um das Gitter besser darstellen zu können, werden aber nur zwei Risikofaktoren innerhalb der folgenden Intervalle verändert, so dass ein zweidimensionales Approximationsgebiet Ω entsteht:

EUROSTOXX:	[-30 %; +30 %]
EURSWAP:	[-250 BP; +250 BP]

Die Approximation der Portfoliofunktion soll nun auf Ω mit einer Genauigkeit von 0.0005 berechnet werden. Der Fehlerschätzer, auf dessen Basis das Gitter adaptiv verfeinert werden soll, wird, wie in Abschnitt 5.2.1 beschrieben, unter Verwendung von v auf dem jeweils aktuellen Gitter ausgewertet (err_type = 0).

In den Abbildungen 8.1 - 8.4 ist die Approximation der Portfoliofunktion und das jeweilige Gitter zu verschiedenen Zeitpunkten im Algorithmus aufgezeigt. Ein rotes Gitterelement bedeutet dabei, dass das Element weiter untersucht werden muss, während fertige Elemente

weiß gezeichnet werden. Um den Unterschied zwischen den verschiedenen Verfeinerungsstrategien zu verdeutlichen, ist jeweils das Ergebnis für äquidistante Verfeinerung sowie für adaptive Verfeinerung in alle Koordinatenrichtungen (ref_type = 0) und anisotrope Verfeinerung (ref_type = 1) dargestellt.

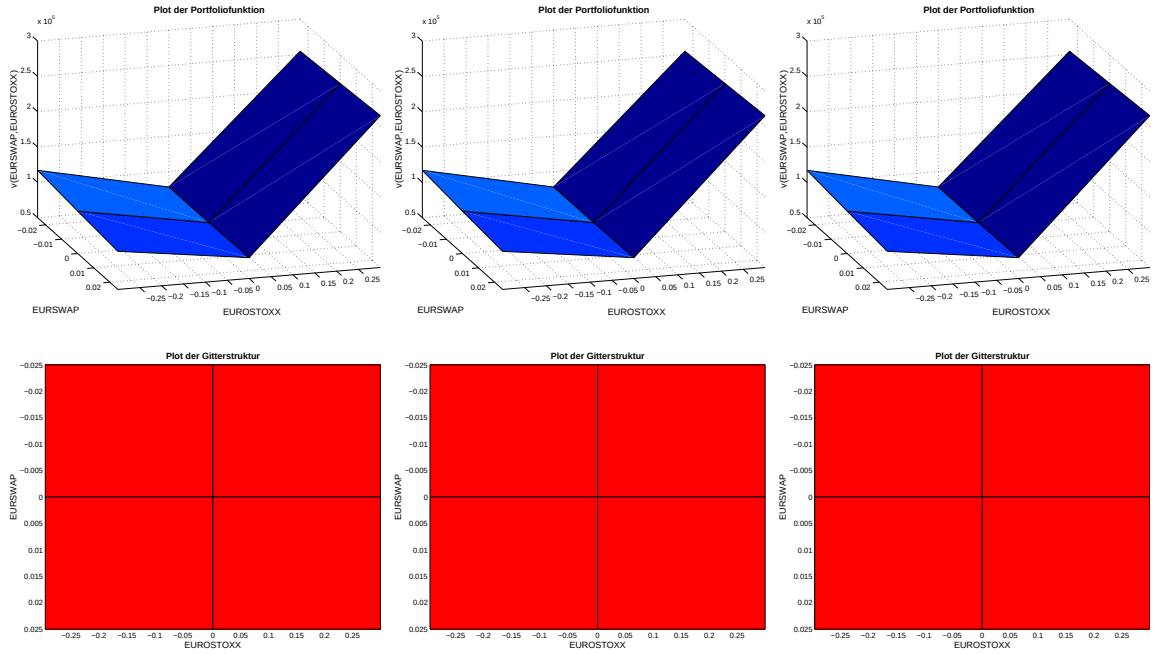

Abbildung 8.1: Approximation und Gitter für Beispiel 8.1 bei äquidistanter Verfeinerung (links) und adaptiver Verfeinerung mit ref_type = 0 (Mitte) bzw. ref_type = 1 (rechts) zu Beginn der Iteration

Das Anfangsgitter besteht bei allen Verfeinerungsstrategien aus vier Quadern, so dass sich zu Beginn der Iteration noch kein Unterschied ergibt. Auch nach vier Iterationen, d.h. nach zwei Durchgängen in jeder Dimension, zeigt sich lediglich bei der anisotropen Verfeinerung eine Veränderung gegenüber den anderen beiden Methoden. Das Gitter ist in Richtung des EURSWAP weniger fein, da das Portfolio auf diesen Risikofaktor weniger sensitiv reagiert. Ein Unterschied zwischen äquidistanter Verfeinerung und adaptiver Verfeinerung in alle Koordinatenrichtungen zeigt sich erst ab der achten Iteration. Bei letzterer Verfeinerungsmethode wurden bereits die Randbereiche für gut befunden und müssen nicht weiter unterteilt werden, während bei äquidistanter Verfeinerung alle Quader solange verfeinert werden bis der Fehlerschätzer überall geringer als die Toleranzschranke ist. Dieser Effekt wird auch im Endergebnis deutlich.

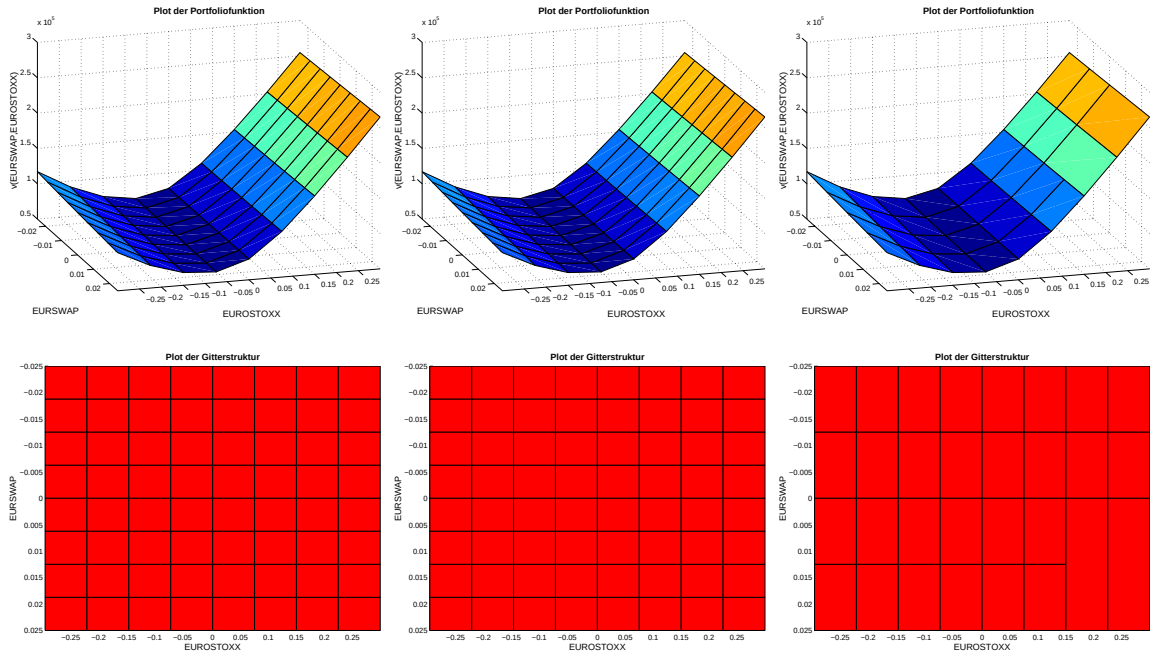

Abbildung 8.2: Approximation und Gitter für Beispiel 8.1 bei äquidistanter Verfeinerung (links) und adaptiver Verfeinerung mit ref_type = 0 (Mitte) bzw. ref_type = 1 (rechts) nach 4 Iterationen

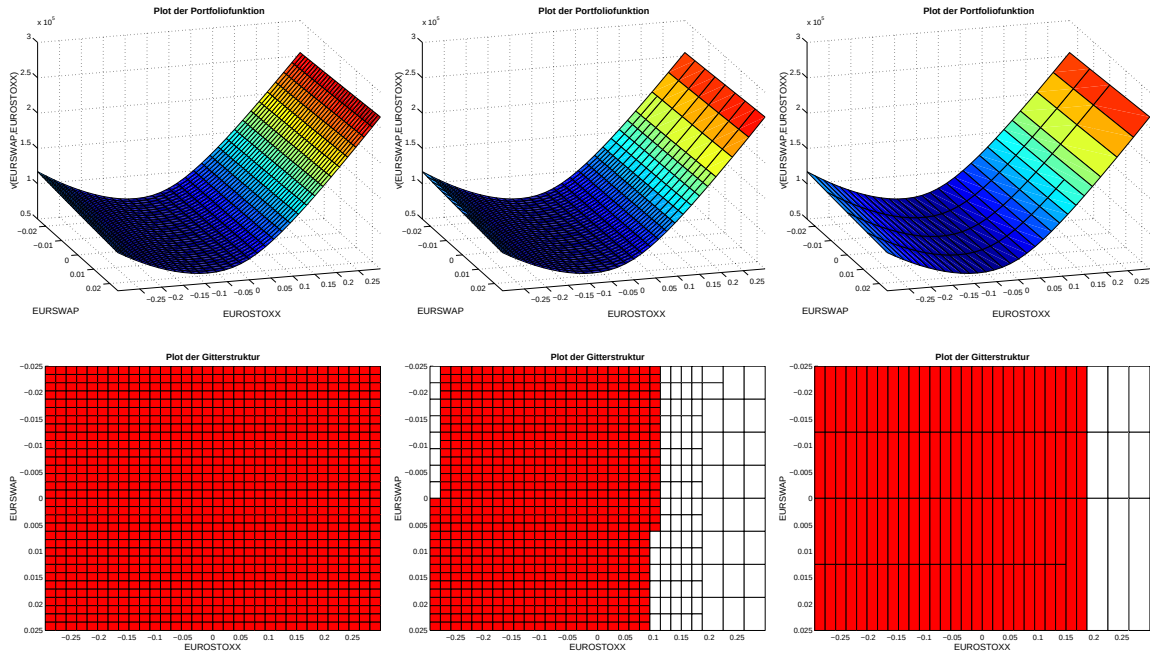

Abbildung 8.3: Approximation und Gitter für Beispiel 8.1 bei äquidistanter Verfeinerung (links) und adaptiver Verfeinerung mit ref_type = 0 (Mitte) bzw. ref_type = 1 (rechts) nach 8 Iterationen

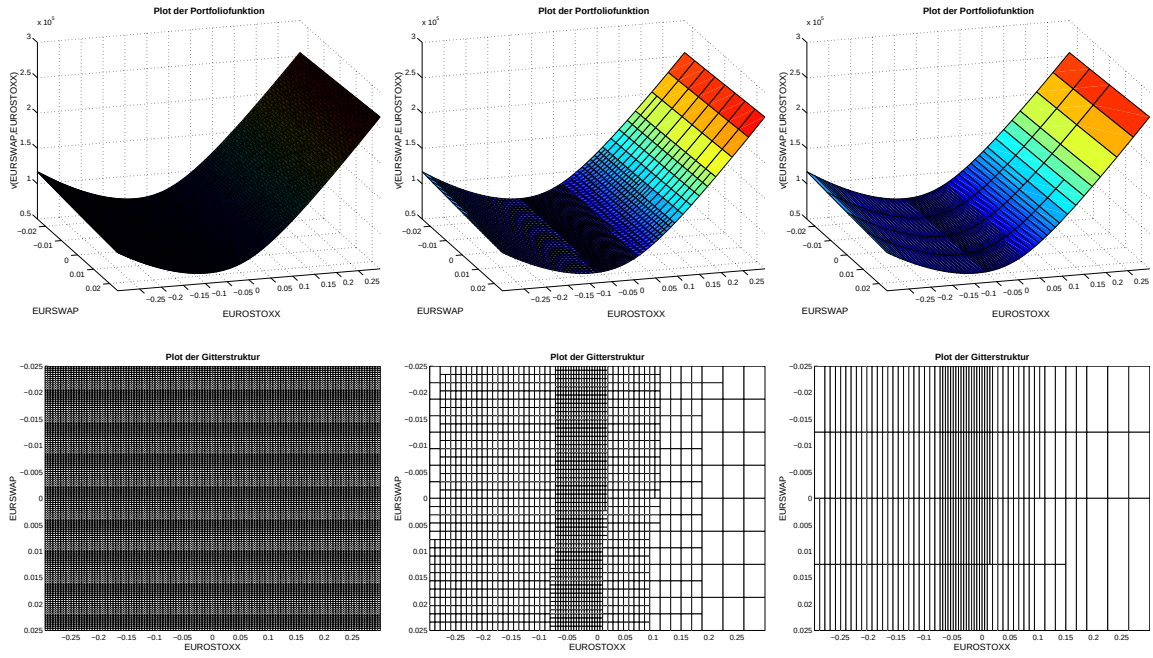

Abbildung 8.4: Approximation und Gitter für Beispiel 8.1 bei äquidistanter Verfeinerung (links) und adaptiver Verfeinerung mit ref_type = 0 (Mitte) bzw. ref_type = 1 (rechts) nach Ende der Iteration

Tabelle 8.1 zeigt die Anzahl der Quader, den globalen Fehlerschätzer sowie die Rechenzeit nach jeder Iteration für die drei Verfeinerungsstrategien. Wie erwartet werden bei der adaptiven Verfeinerung viel weniger Quader erzeugt und eine geringere Rechenzeit benötigt als bei der äquidistanten Verfeinerung. Bemerkenswert ist aber auch der Unterschied innerhalb der adaptiven Methoden zwischen der Verfeinerung in alle Koordinatenrichtungen und anisotroper Verfeinerung. Dass die anisotrope Verfeinerung mit einem Bruchteil der Quader und der Rechenzeit auskommt, liegt vor allem daran, dass die Quader vor allem in diejenigen Richtungen verfeinert werden, auf die das Portfolio besonders sensitiv reagiert, in diesem Fall der EUROSTOXX. Ein weiterer Grund ist, dass pro Quader weniger Testpunkte ausgewertet werden müssen.

	äquidistante Verfeinerung			**adaptive Verfeinerung mit ref_type = 0**			**adaptive Verfeinerung mit ref_type = 1**		
Iteration	#Quader	Fehler-schätzer	Rechen-zeit	#Quader	Fehler-schätzer	Rechen-zeit	#Quader	Fehler-schätzer	Rechen-zeit
0	4	0.3490	0.06	4	0.3490	0.06	4	0.3490	0.06
1	8	0.1348	0.11	8	0.1348	0.11	8	0.0018	0.07
2	16	0.1339	0.24	16	0.1339	0.21	15	0.1338	0.09
3	32	0.0397	0.39	32	0.0397	0.37	30	0.0005	0.13
4	64	0.0394	0.69	64	0.0394	0.65	30	0.0393	0.18
5	128	0.0102	1.27	128	0.0102	1.23	60	0.0000	0.18
6	256	0.0101	2.45	233	0.0101	2.18	60	0.0101	0.24
7	512	0.0026	4.73	441	0.0026	4.04	111	0.0000	0.24
8	1024	0.0026	9.29	779	0.0026	7.01	111	0.0025	0.35
9	2048	0.0006	18.32	1445	0.0006	12.84	195	0.0000	0.35
10	4096	0.0006	36.32	1762	0.0006	15.61	195	0.0006	0.51
11	8192	0.0002	72.19	2373	0.0002	20.96	234	0.0000	0.51
12	—	—	—	—	—	—	234	0.0002	0.58

Tabelle 8.1: Auswertung für Beispiel 8.1 bei äquidistanter Verfeinerung (links) und adaptiver Verfeinerung mit ref_type = 0 (Mitte) bzw. ref_type = 1 (rechts)

8.2 DAX-Optionen

Als nächstes wird ein Portfolio aus acht Optionen auf den DAX betrachtet. Als Risikofaktoren gehen der Aktienindex DAX, dessen Volatilität DAX_VOL und die Zinskurve EURSWAP ein. Die verwendeten Shifts lauten:

DAX:	[-30 %; +30 %]
EURSWAP:	[-300 BP; +300 BP]
DAX_VOL:	[-50 %; +50 %]

Die Fehlertoleranz wurde auf 0.01 gesetzt. Im Gegensatz zum ersten Beispiel soll nun nicht die Portfoliofunktion selbst, sondern die Gewinn- und Verluststruktur des Portfolios approximiert werden.

Abbildung 8.5 zeigt zunächst die Approximation an verschiedenen Stellen im dreidimensionalen Gebiet Ω bei äquidistanter Verfeinerung des Gitters. Die Anzahl der Quader, der Fehlerschätzer und die Rechenzeit nach jeder Iteration der Verfeinerung werden in Tabelle 8.2 aufgelistet.

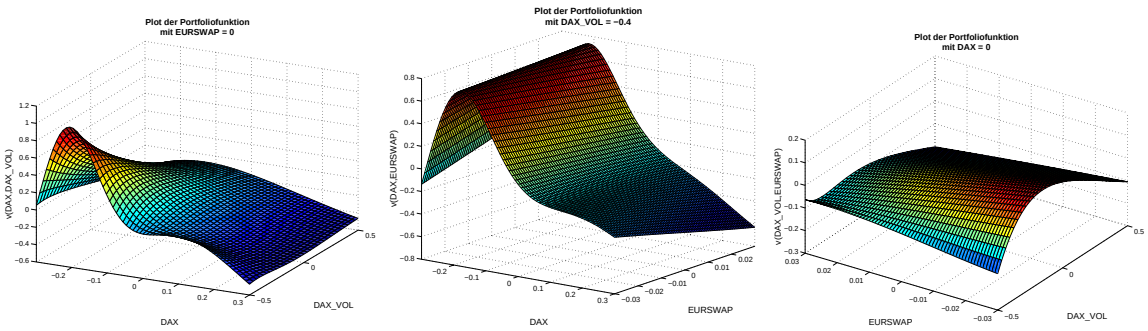

Abbildung 8.5: Approximation für Beispiel 8.2 bei äquidistanter Verfeinerung

	äquidistante Verfeinerung		
Iteration	#Quader	Fehlerschätzer	Rechenzeit
0	8	1.0993	0.01
1	16	0.5773	0.02
2	32	0.5797	0.03
3	64	0.5797	0.05
4	128	0.1637	0.10
5	256	0.1637	0.20
6	512	0.1637	0.41
7	1024	0.0448	0.76
8	2048	0.0445	1.46
9	4096	0.0445	2.87
10	8192	0.0113	5.63
11	16384	0.0112	11.13
12	32768	0.0112	22.19
13	65536	0.0029	44.66

Tabelle 8.2: Auswertung für Beispiel 8.2 bei äquidistanter Verfeinerung

In diesem Beispiel sollen die Unterschiede zwischen den beiden Verfeinerungsstrategien aus Kapitel 6 (ref_type = 0 und ref_type = 1) und den zwei unterschiedlichen Arten der Fehlerschätzung, wie sie in Kapitel 5 besprochen wurden (err_type = 0 und err_type = 1), explizit herausgestellt werden. Dazu werden diese beiden Parameter auf unterschiedliche Art

und Weise miteinander kombiniert, so dass daraus jedes Mal ein neuer adaptiver Algorithmus resultiert. Die Abbildungen 8.6 und 8.7 zeigen die Approximationen bei Fehlerschätzung unter Verwendung von v (err_type = 0), jedoch einmal mit Verfeinerung in alle Koordinatenrichtungen (ref_type = 0) und einmal mit anisotroper Verfeinerung (ref_type = 1). Die Approximation an die Gewinn- und Verlustverteilung wird wie auch schon bei der äquidistanten Verfeinerung jeweils in Abhängigkeit von zwei Risikofaktoren dargestellt, während der dritte Risikofaktor festgeschrieben wird. Die dreidimensionalen Gitter, die bei diesen beiden Approximationsvarianten entstehen, finden sich in Abbildung 8.8. In Tabelle 8.3 werden die Ergebnisse aufgeführt.

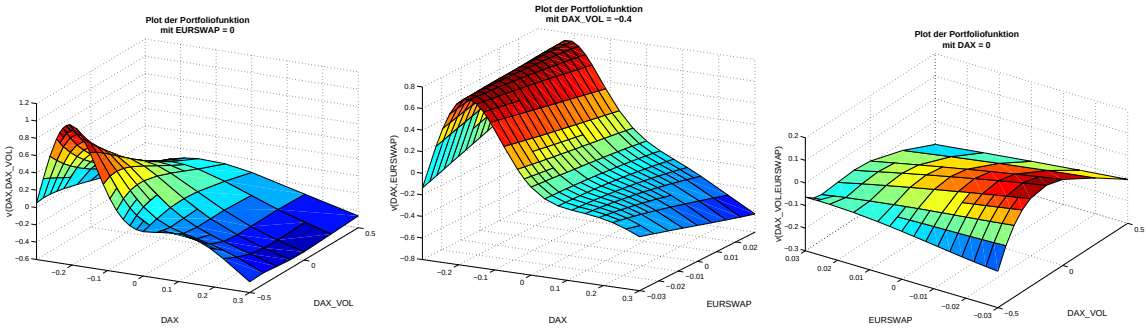

Abbildung 8.6: Approximation für Beispiel 8.2 bei adaptiver Verfeinerung mit ref_type = 0 und err_type = 0

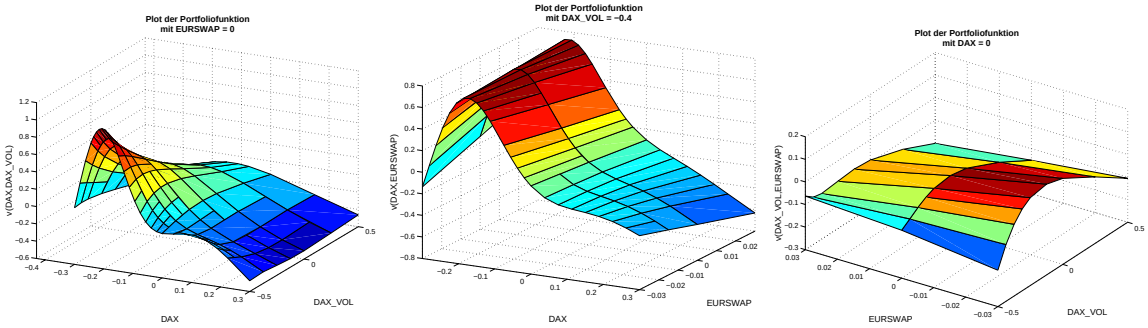

Abbildung 8.7: Approximation für Beispiel 8.2 bei adaptiver Verfeinerung mit ref_type = 1 und err_type = 0

Wie auch schon im ersten Beispiel zeigt sich, dass die anisotrope Verfeinerung weitaus weniger Quader und Rechenzeit benötigt als die adaptive Verfeinerung in alle Richtungen, um die Approximation auf ganz Ω mit der derselben Genauigkeit zu berechnen, weil die Quader nur in diejenigen Richtungen unterteilt werden, in denen ein großer Fehler auftritt. Beide adaptiven Ansätze sind jedoch der äquidistanten Methode in Bezug auf Speicherplatzbedarf und Rechenaufwand weit überlegen.

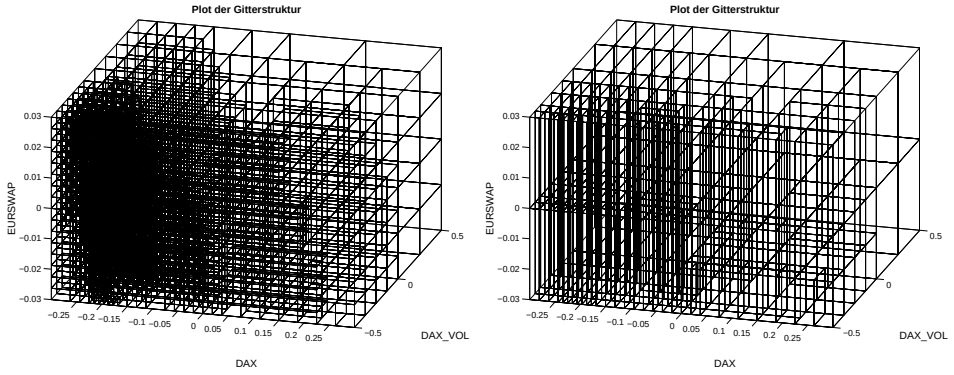

Abbildung 8.8: Gitter für Beispiel 8.2 bei adaptiver Verfeinerung mit ref_type = 0 (links) bzw. ref_type = 1 (rechts) und err_type = 0

	adaptive Verfeinerung mit ref_type = 0 und err_type = 0			**adaptive Verfeinerung mit ref_type = 1 und err_type = 0**		
Iteration	#Quader	Fehlerschätzer	Rechenzeit	#Quader	Fehlerschätzer	Rechenzeit
0	8	1.0993	0.00	8	1.0993	0.00
1	16	0.5773	0.01	16	0.0168	0.00
2	32	0.5797	0.03	18	0.1180	0.00
3	64	0.5797	0.06	36	0.5749	0.01
4	120	0.1637	0.11	68	0.0044	0.01
5	203	0.1637	0.18	68	0.0606	0.01
6	366	0.1637	0.30	105	0.1637	0.02
7	601	0.0448	0.48	160	0.0000	0.02
8	775	0.0445	0.60	160	0.0210	0.03
9	1114	0.0445	0.83	197	0.0444	0.04
10	1496	0.0113	1.09	241	0.0000	0.04
11	1535	0.0112	1.12	241	0.0063	0.05
12	1610	0.0112	1.18	241	0.0111	0.06
13	1685	0.0029	1.23	248	0.0000	0.06
14	—	—	—	248	0.0000	0.06
15	—	—	—	248	0.0028	0.06

Tabelle 8.3: Auswertung für Beispiel 8.2 bei adaptiver Verfeinerung mit ref_type = 0 (links) bzw. ref_type = 1 (rechts) und err_type = 0

In den folgenden Abbildungen 8.9 und 8.10 sollen nun diese beiden Verfeinerungsstrategien mit der zweiten Art der Fehlerschätzung (err_type = 1) kombiniert werden. Der Fehlerschätzer wird nun nicht mehr auf dem aktuellen Gitter ausgewertet, indem der exakte Portfoliowert an einer Reihe von Testpunkten mit dem approximierten Wert verglichen wird, sondern er ergibt sich aus dem Vergleich der Approximationen auf den Vorgängergittern. Im Gegensatz zur ersten Art der Fehlerschätzung braucht nun v nicht mehr ausgewertet werden, was sich positiv auf die Rechenzeit auswirken sollte. Abbildung 8.11 zeigt wieder die beiden Gitter, die im Laufe der Approximation entstehen.

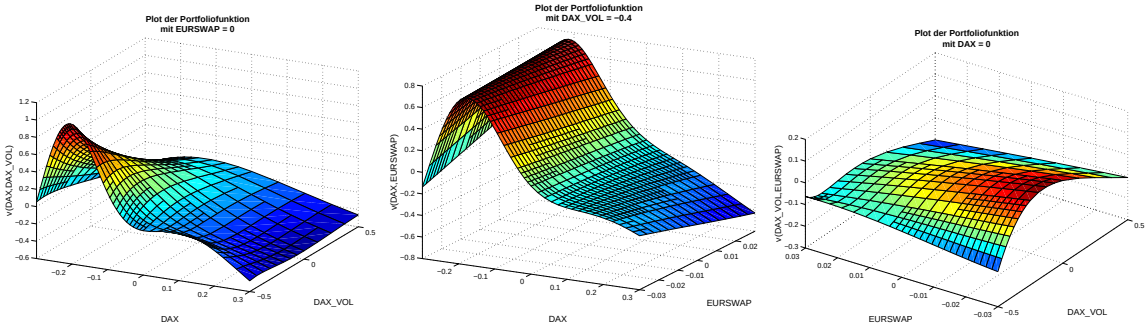

Abbildung 8.9: Approximation für Beispiel 8.2 bei adaptiver Verfeinerung mit ref_type = 0 und err_type = 1

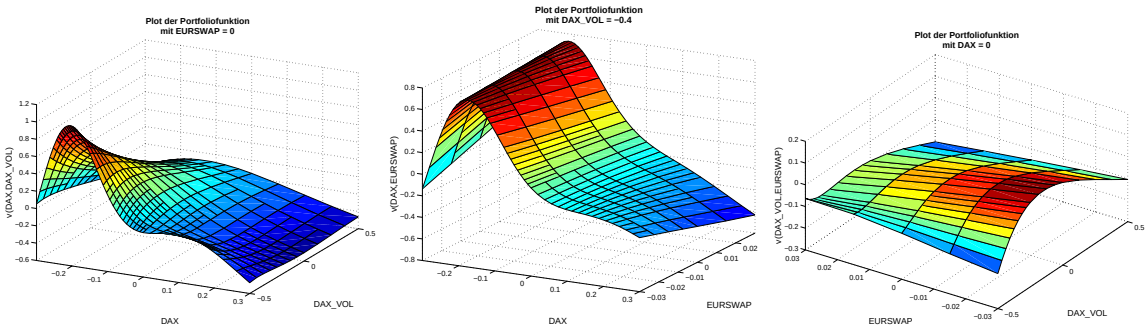

Abbildung 8.10: Approximation für Beispiel 8.2 bei adaptiver Verfeinerung mit ref_type = 1 und err_type = 1

Die in Tabelle 8.4 aufgeführte Auswertung der beiden Varianten bestätigt zunächst, dass durch diese Art der Fehlerschätzung ein geringerer Rechenaufwand entsteht. Trotz der viel größeren Anzahl der Quader, die sich dadurch erklären lässt, dass in jeder Iteration nicht der aktuelle Fehler, sondern der Fehler des Vorgängergitters, auf dem die Approximation noch nicht so gut war, berechnet wird, benötigt der Algorithmus bei der Verfeinerungsvariante ref_type = 0 weniger Rechenzeit. Bei der anisotropen Verfeinerung dagegen ist der Algorithmus mit Fehlerschätzung auf dem aktuellen Gitter schneller. Dies kommt daher, dass bei dieser Variante nur diejenigen Punkte als Testpunkte verwendet werden, die bei einer potenziellen Verfeinerung sowieso zum Gitter dazukommen. An diesen Punkten muss die Portfoliofunktion auch bei der Fehlerschätzung ohne Verwendung von v exakt ausgewertet werden, falls der Quader verfeinert wird.

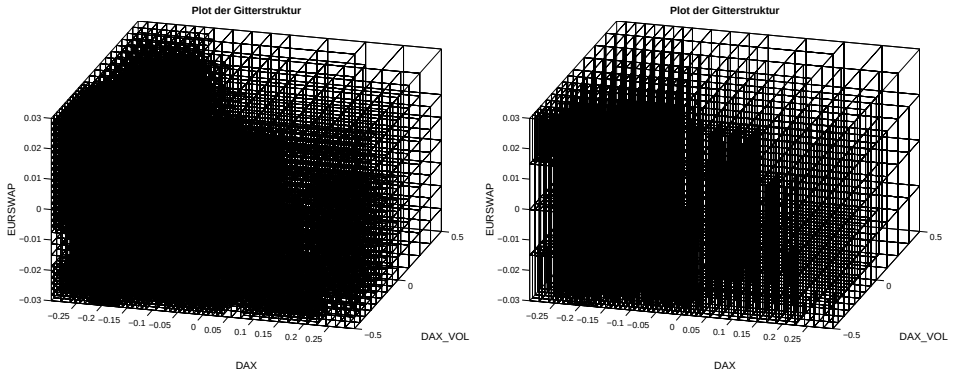

Abbildung 8.11: Gitter für Beispiel 8.2 bei adaptiver Verfeinerung mit ref_type = 0 (links) bzw. ref_type = 1 (rechts) und err_type = 1

	adaptive Verfeinerung mit ref_type = 0 und err_type = 1			**adaptive Verfeinerung mit ref_type = 1 und err_type = 1**		
Iteration	#Quader	Fehlerschätzer	Rechenzeit	#Quader	Fehlerschätzer	Rechenzeit
0	8	0.2384	0.00	8	0.2384	0.00
1	16	1.0993	0.00	16	0.0182	0.00
2	32	1.0993	0.01	32	0.2030	0.01
3	64	1.0993	0.01	64	1.0993	0.01
4	128	0.5786	0.02	128	0.0168	0.01
5	254	0.5786	0.04	144	0.1180	0.03
6	506	0.5786	0.06	284	0.5786	0.04
7	954	0.1637	0.09	538	0.0044	0.04
8	1568	0.1637	0.15	538	0.0629	0.06
9	2796	0.1637	0.25	822	0.1637	0.10
10	4620	0.0444	0.32	1242	0.0000	0.10
11	5880	0.0444	0.44	1242	0.0215	0.12
12	8400	0.0444	0.59	1510	0.0444	0.15
13	11416	0.0112	0.62	1804	0.0000	0.15
14	11566	0.0112	0.64	1804	0.0063	0.15
15	11866	0.0112	0.68	1804	0.0111	0.15
16	12466	0.0028	0.69	1826	0.0000	0.15
17	—	—	—	1826	0.0000	0.15
18	—	—	—	1826	0.0028	0.15

Tabelle 8.4: Auswertung für Beispiel 8.2 bei adaptiver Verfeinerung mit ref_type = 0 (links) bzw. ref_type = 1 (rechts) und err_type = 1

8.3 Portfolio mit gemischten Optionen

In den letzten beiden Beispielen hat sich die Algorithmusvariante mit anisotroper Verfeinerung (ref_type = 1) und Fehlerschätzung unter Verwendung von v (err_type = 0) als sehr erfolgreich erwiesen. Deshalb soll nun in einem abschließenden Beispiel die Performance dieser Variante für ein höherdimensionales Approximationsgebiet und unterschiedliche Fehlertoleranzen *tol* untersucht werden. Das Portfolio besteht aus Optionen auf verschiedene Underlyings, die jedoch nicht alle unabhängig voneinander geshiftet werden sollen. Vielmehr werden sie je nach Zugehörigkeit auf die beiden Aktienindizes EUROSTOXX und DAX gemappt. Es ergibt sich ein fünfdimensionales Approximationsgebiet Ω, das durch die folgenden Intervalle begrenzt wird:

EUROSTOXX:	[-30 %; +30 %]
DAX:	[-30 %; +30 %]
EURSWAP:	[-150 BP; +150 BP]
EUROSTOXX_VOL:	[-25 %; +25 %]
DAX_VOL:	[-20 %; +20 %]

In den drei Abbildungen 8.12 – 8.14 werden Ausschnitte der Gewinn- und Verluststruktur dieses Portfolios für unterschiedliche Fehlertoleranzen dargestellt. Je höher die Genauigkeit, desto feiner werden die Quader vor allem in Richtung der beiden Aktienindizes unterteilt, da sie den Portfoliowert am stärksten beeinflussen. Dagegen wird in Richtung der Volatilitäten und der Zinskurve weitaus weniger unterteilt. Daran ändert auch eine höhere Genauigkeit kaum etwas. Tabelle 8.5 fasst den Ablauf der Approximation zusammen und stellt die Anzahl der Quader, den Fehlerschätzer und die Rechenzeit nach jeder Iteration für die unterschiedlichen Fehlertoleranzen einander gegenüber.

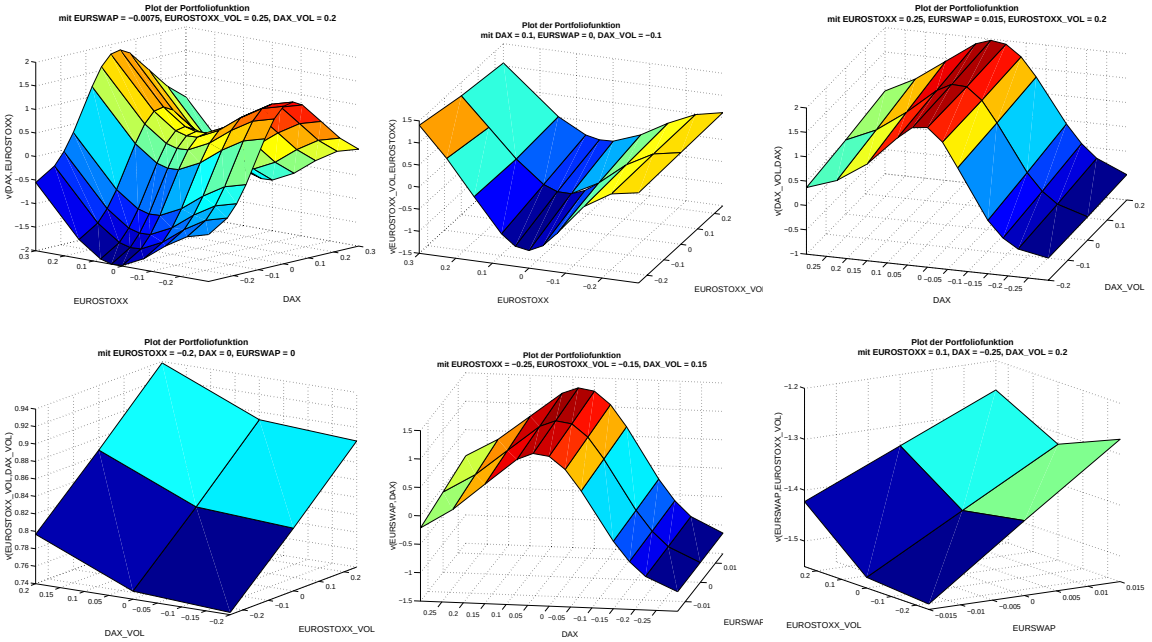

Abbildung 8.12: Approximation für Beispiel 8.3 mit tol = 0.05

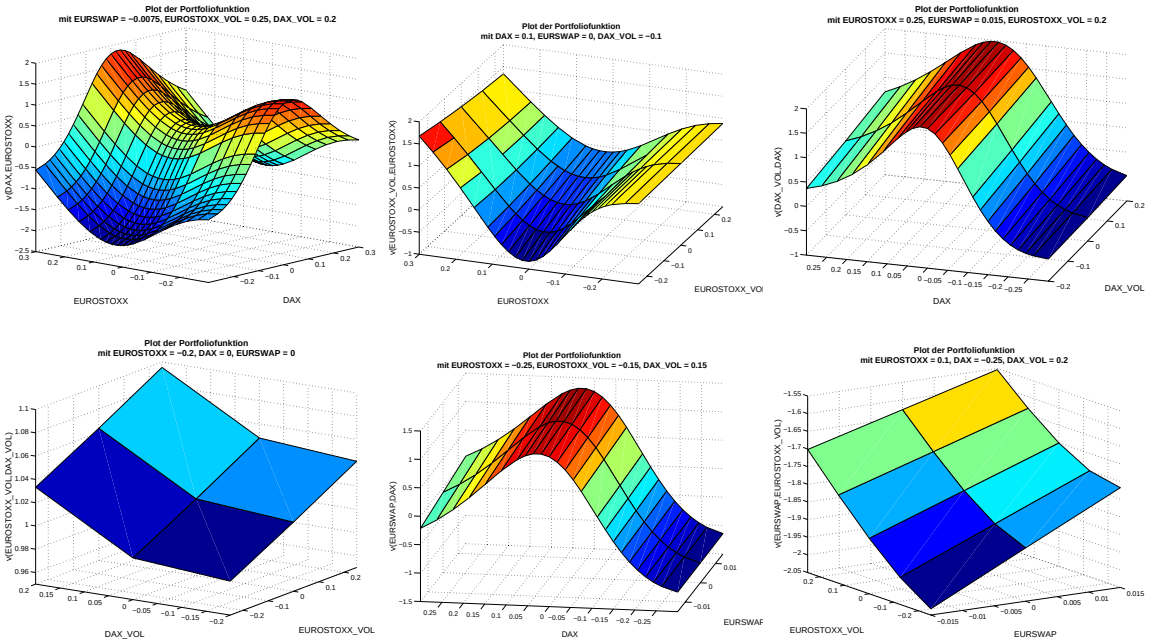

Abbildung 8.13: Approximation für Beispiel 8.3 mit tol = 0.01

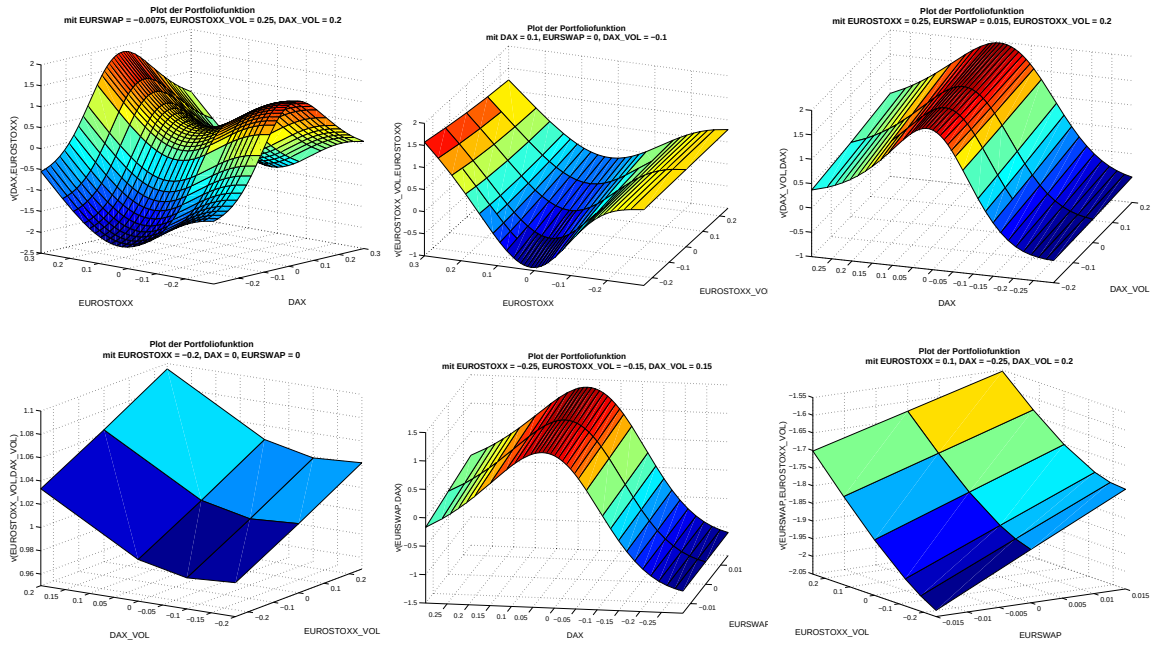

Abbildung 8.14: Approximation für Beispiel 8.3 mit tol = 0.005

	tol = 0.05			**tol = 0.01**			**tol = 0.005**		
Iteration	#Quader	Fehler-schätzer	Rechen-zeit	#Quader	Fehler-schätzer	Rechen-zeit	#Quader	Fehler-schätzer	Rechen-zeit
0	32	0.3710	0.10	32	0.3710	0.13	32	0.3710	0.11
1	64	0.5711	0.19	64	0.5711	0.21	64	0.5711	0.20
2	128	0.0011	0.34	128	0.0011	0.37	128	0.0011	0.35
3	128	0.0196	0.50	128	0.0196	0.53	128	0.0196	0.51
4	128	0.0061	0.66	216	0.0061	0.79	224	0.0061	0.79
5	128	0.2486	0.83	216	0.2486	1.06	280	0.2486	1.15
6	240	0.2001	1.14	432	0.2001	1.59	560	0.2001	1.90
7	480	0.0000	1.14	864	0.0000	1.59	1120	0.0000	1.91
8	480	0.0000	1.15	864	0.0056	2.46	1120	0.0056	3.14
9	480	0.0000	1.15	864	0.0000	2.46	1200	0.0016	3.79
10	480	0.0941	1.73	864	0.0941	3.54	1200	0.0941	5.32
11	576	0.1139	2.45	1568	0.1139	5.51	2300	0.1139	8.16
12	792	0.0000	2.45	3136	0.0000	5.52	4600	0.0000	8.18
13	792	0.0000	2.46	3136	0.0000	5.53	4600	0.0015	9.01
14	792	0.0000	2.46	3136	0.0000	5.53	4600	0.0000	9.01
15	792	0.0263	2.78	3136	0.0263	9.03	4600	0.0263	14.74
16	792	0.0372	3.33	4320	0.0372	14.35	7200	0.0372	23.77
17	—	—	—	6480	0.0000	14.37	12780	0.0000	23.81
18	—	—	—	6480	0.0000	14.37	12780	0.0000	23.82
19	—	—	—	6480	0.0000	14.38	12780	0.0000	23.83
20	—	—	—	6480	0.0067	18.76	12780	0.0067	35.22
21	—	—	—	6480	0.0098	24.19	13632	0.0098	50.21
22	—	—	—	—	—	—	17088	0.0000	50.24
23	—	—	—	—	—	—	17088	0.0000	50.26
24	—	—	—	—	—	—	17088	0.0000	50.27
25	—	—	—	—	—	—	17088	0.0017	53.07
26	—	—	—	—	—	—	17088	0.0025	61.61

Tabelle 8.5: Auswertung für Beispiel 8.3 mit tol = 0.05 (links), tol = 0.01 (Mitte) und tol = 0.005 (rechts)

Kapitel 9

Fazit und Ausblick

Diese Arbeit untersucht die Approximation einer Portfoliofunktion v durch adaptive Gittererzeugung und multilineare Interpolation. Die Beispiele in Kapitel 8 zeigen, dass durch die adaptiven Verfeinerungsstrategien, besonders aber durch die anisotrope Verfeinerung kombiniert mit der Fehlerschätzung unter Verwendung von v, eine gute Approximation auch ohne Kenntnis der kritischen Regionen, das heißt ohne vorher die Struktur des Portfolios analysiert zu haben, erreicht wird. Im Gegensatz zur äquidistanten Methode bleibt dabei die Effizienz des Algorithmus gewahrt, denn es werden nur diejenigen Regionen verfeinert, in denen der Fehler groß ist.

Im Vergleich zu den in Kapitel 2 beschriebenen Stresstesting-Methoden hat dieser Ansatz zum einen den Vorteil, dass die Gewinn- und Verluststruktur des Portfolios vollständig erfasst wird. So können keine Sicherheitslücken durch zu grobe Szenarienauswahl übersehen werden und auch das Worst-Case-Szenario, das die Krise mit dem größten Verlust angibt, lässt sich nun mühelos durch Minimierung der approximierten Portfoliofunktion ermitteln. Zum anderen kann man durch gezielte Auswertungen leicht herausfinden, in welchem Maße jeder einzelne Risikofaktor für den jeweiligen Verlust verantwortlich ist. Das ist vor allem dann wichtig, wenn geeignete Gegenmaßnahmen bestimmt werden sollen.

Ein Nachteil der hier betrachteten Methode ist der relativ große Aufwand, der durch die Neubewertung des Portfolios in jedem Punkt des Gitters und gegebenenfalls an allen Testpunkten entsteht. Im Vergleich zu den in Kapitel 2 beschriebenen Methoden, wo in jedem Stresstesting-Durchlauf nur eine bestimmte Anzahl[10] von Szenarien ausgewertet wird, kann es hier – je nach gewünschter Genauigkeit und zugrunde liegender Portfoliofunktion – nötig sein, ein Vielfaches an Szenarien zu berechnen. Hier könnte eventuell eine Kombination der adaptiven Gittererzeugung mit Interpolationstechniken höherer Ordnung anstelle von multilinearer Interpolation Abhilfe schaffen. Von einer Interpolation höherer Ordnung verspricht man sich höhere Genauigkeit pro Schritt und dadurch wiederum eine geringere Anzahl von Portfolioauswertungen.

[10] je nach Finanzinstitut etwa im Bereich von 50 - 100 Szenarien für Sensitivitäts-, Szenario- und Worst-Case-Analysen

Inwiefern die Beschaffenheit eines Marktrisikoportfolios dafür geeignet ist, müsste natürlich erst noch untersucht werden. Zu klären wäre zudem die Verknüpfung von mehrdimensionaler Interpolation höherer Ordnung mit den Gitterpunkten als Stützstellen. Um zusätzlich die Dimensionalität des Algorithmus und damit die Anzahl der zu bewertenden Szenarien möglichst gering zu halten, könnte man vielleicht auch schon im Vorfeld des Stresstests im Rahmen einer Hauptkomponentenanalyse klären, welche Risikofaktoren in erster Linie für das Risiko eines Portfolios verantwortlich sind.

Eine alternative Anwendungsmöglichkeit des entwickelten Algorithmus liegt in der Produktbewertung. Gerade im Investment Banking sind schnelle Produktanalysen gefordert. Für einige Produkte, wie beispielsweise exotische Optionen, existieren aber nur sehr komplizierte und langsame Pricingalgorithmen. Die Idee besteht nun darin, den Algorithmus für jedes gewünschte Produkt über Nacht laufen zu lassen und am Tag nur noch innerhalb des Gitters zu interpolieren. Da hier an jedem Gitterpunkt nur ein Produkt bewertet werden muss und die Anzahl der eingehenden Risikofaktoren nicht so hoch ist, sollte der Aufwand im Vergleich zur Anwendung im Stresstesting wesentlich geringer ausfallen. Außerdem bewegen sich die Werte der Risikofaktoren während eines Tages unter normalen Marktverhältnissen innerhalb von wesentlich kleineren Intervallen, als für das ursprüngliche Anwendungsgebiet, das Stresstesting, benötigt, so dass das Approximationsgebiet Ω viel kleiner ist und man bei gleicher Iterationsanzahl eine wesentlich genauere Approximation erwarten darf. Ist das Gitter einmal berechnet, kann der approximative Wert des Produkts sehr schnell mittels der beschriebenen multilinearen Interpolation berechnet werden. Zu untersuchen wäre natürlich noch der Zeitaufwand, den der Algorithmus für jedes einzelne Produkt benötigt. Da an jedem Gitterpunkt exakt bewertet wird, hängt es von den ursprünglichen Bewertungsroutinen und der gewünschten maximalen Fehlertoleranz ab, ob ein Durchlauf des Algorithmus für viele Produkte überhaupt praktikabel ist. Außerdem kann der Algorithmus trotz des zu erwartenden geringen Approximationsfehlers natürlich nicht die genaue Bewertung ersetzen, sondern ist vielmehr als eine Art Entscheidungshilfe zu betrachten.

Anhang A

Bewertungsmodelle

Um den Wert des betrachteten Portfolios berechnen zu können, wurden eigene Bewertungsfunktionen für Aktien, Zinsprodukte, Optionen und Fremdwährungsprodukte implementiert. Hier finden sich die theoretischen Modelle, die der Implementierung zugrunde liegen. Für detaillierte Informationen und Hintergründe vergleiche Hull [13] und Ross [19].

A.1 Aktien

Der Wert p einer Aktienposition ergibt sich aus der Anzahl n und dem Aktienkurs s, der als Risikofaktor gegeben ist. Die Bewertungsformel lautet daher

$$p = n \cdot s.$$

A.2 Zinsprodukte

Der Begriff Zinsprodukt kann relativ weit gefasst sein. Darunter zählen vor allem klassische Bonds mit fixem Zinssatz, Floating Rate Notes, aber auch Swaps und exotischere Zinsprodukte. Hinsichtlich ihrer Anpassungsfähigkeit an auftretende Marktzinsänderungen lassen sich Zinsprodukte aller Art in zwei Gruppen einteilen. Festzinsgeschäfte umfassen sämtliche Positionen, die für ihre gesamte Laufzeit einen fest vereinbarten und in seiner Höhe konstanten Zinssatz aufweisen. Variable Zinssätze hingegen besitzen entweder keine Zinsbindungsdauer oder aber nur eine sehr kurze. Dadurch sind diese Geschäfte teilweise bzw. voll zinsreagibel.

Die meisten dieser Produkte haben mehr oder weniger regelmäßige Zahlungsströme $c(t_i)$ zu verschiedenen Zeitpunkten t_i, $i = 1, \ldots, n$, gemeinsam, wodurch sie relativ einfach und übergreifend bewertet werden können. Zur Wertermittlung müssen diese Zahlungsströme lediglich mit dem entsprechenden Zinssatz $r(t_i)$ zur Zeit t_i abgezinst werden, wobei die Zahlungsströme

der Positionen, die sich auf denselben Zinssatz und denselben Zeitraum beziehen, zusammengefasst werden können. Da die Zinskurve r nur für bestimmte Stützstellen t_j, $j = 1, \ldots, m$, gegeben ist, wird bei Zahlungen zu einem Zeitpunkt t_i, der zwischen zwei Stützstellen t_j und t_{j+1} liegt, linear interpoliert, um den passenden Zinssatz $r(t_i)$ zu ermitteln:

$$r(t_i) = r(t_j) + \frac{t_i - t_j}{t_{j+1} - t_j} \left(r(t_{j+1}) - r(t_j) \right).$$

Der Wert p des Zinsproduktes ergibt sich dann durch Abzinsung der Zahlungsströme $c(t_i)$ mit den Zinsraten $r(t_i)$, wobei t_i die Restlaufzeit in Jahren angibt.

$$p = \sum_{i=1}^{n} c(t_i) \cdot e^{-r(t_i)t_i}.$$

A.3 Optionen

Eine Option verleiht ihrem Inhaber das Recht, ein Underlying zu einem bestimmten Zeitpunkt (europäische Option) bzw. innerhalb einer bestimmten Zeitspanne (amerikanische Option) zu einem vorher vereinbarten Ausübungspreis zu kaufen (Call-Option) oder zu verkaufen (Put-Option).

Die Bewertung von Optionen ist ein sehr umfangreiches und kompliziertes Gebiet der Finanzmathematik. Nicht für alle Optionstypen existieren überhaupt explizite Bewertungsformeln wie die Black-Scholes-Formel. Einige exotische Optionen können nur mit Hilfe von Bewertungsprozessen, wie zum Beispiel der Monte-Carlo-Simulation, bewertet werden. Eine beliebte Methode, um den Wert von vielen verschiedenen Optionstypen anzunähern, ist wegen ihrer Schnelligkeit die Taylorapproximation mit den Options-Griechen Delta, Gamma, Vega, Rho und Tau. Die Griechen sind Sensitivitäten des Optionswerts gegenüber kleinen Veränderungen der eingehenden Risikofaktoren. Für große Veränderungen der Risikofaktoren, wie sie im Stresstesting auftreten, verliert diese lineare Approximation jedoch immer mehr ihre Gültigkeit und ist daher nicht zweckmäßig. Stattdessen wurde in dieser Arbeit die Black-Scholes-Formel für europäische Optionen implementiert. Die Herleitung dieser Formel sowie Weiterentwicklungen und alternative Bewertungsmethoden finden sich in Hull [13].

Der Preis einer Option hängt von folgenden Risikofaktoren ab:

- Preis des Underlyings s
- Ausübungspreis bzw. Strike k
- Restlaufzeit bis zum Ausübungsdatum t
- Volatilität des Underlyings σ

- risikoloser Zinssatz r

Der Wert von europäischen Calls und Puts ergibt sich unter der Annahme, dass keine Dividenden gezahlt werden, aus der Black-Scholes-Formel

$$\begin{aligned} c &= s \cdot N(d_1) - ke^{-rt} \cdot N(d_2) \qquad \text{für Calls} \\ p &= -s \cdot N(-d_1) + ke^{-rt} \cdot N(-d_2) \qquad \text{für Puts,} \end{aligned}$$

wobei

$$\begin{aligned} d_1 &= \frac{\ln\left(\frac{s}{k}\right) + \left(r + \frac{\sigma^2}{2}\right) t}{\sigma\sqrt{t}} \\ d_2 &= \frac{\ln\left(\frac{s}{k}\right) + \left(r - \frac{\sigma^2}{2}\right) t}{\sigma\sqrt{t}} \\ &= d_1 - \sigma\sqrt{t} \end{aligned}$$

und $N(y)$ durch die Verteilungsfunktion der Standard-Normalverteilung

$$N(y) = \int_{-\infty}^{y} \frac{1}{\sqrt{2\pi}} \exp\left(\frac{-x^2}{2}\right) dx$$

gegeben ist.

A.4 Fremdwährungspositionen

Bei Positionen in Fremdwährung müssen die berechneten Produktwerte p noch mit dem für das Szenario relevanten Wechselkurs in Euro umgewandelt werden. Da man in Europa immer Währungsrisiken gegenüber der Währung EUR betrachtet, ist die Konvention wichtig, in welcher die Wechselkurse angegeben sind. In dieser Arbeit wird folgende Schreibweise verwendet:

$$x = \frac{XXX}{EUR}$$

Ein positiver Shift des Risikofaktors x, das heißt ein steigender Kurs, bedeutet, dass die Währung XXX an Wert verliert und der EUR an Wert gewinnt. Positive Shifts stellen also Eurostärke dar, negative Euroschwäche.

Der Wert einer Fremdwährungsposition p wird folgendermaßen in Euro umgerechnet

$$p_{EUR} = p \cdot \frac{1}{x}.$$

Anhang B

C– und C++–Quelltext

Der Vollständigkeit halber sind die in Kapitel 7 beschriebenen Routinen zur Approximation der Portfoliofunktion auf den folgenden Seiten abgedruckt. Die Funktionen im Modul `grid.c` stammen dabei zu einem großen Teil aus einem bereits implementierten Modul zur Gittererzeugung und -verwaltung von Prof. Dr. Lars Grüne. Die Funktionen zur Optionsbewertung sind in Anlehnung an [14] und [16] entstanden. Für genauere Informationen zu Funktion und Aufruf der Programme sei auf Kapitel 7 verwiesen.

B.1 Modul `main.cpp`

```
/*****************************************************************************
MODULE: main.cpp
PURPOSE: functions for loading portfolio, market and scenario information
         and controlling the adaptive grid generation
*****************************************************************************/

#include <iostream>
#include <fstream>
#include <cmath>
#include <time.h>
#include "valuation.h"
extern "C"{
#include "grid.h"
#include "defs.h"
}

using namespace std;

struct product *p = NULL;
struct riskfactor *rf = NULL;
struct shift *s = NULL;
int num_p;
int num_rf;
int dim;
int ref_type;
int err_type;
```

```
int approx_type;
int refdir;
double xu[MAXDIM], xo[MAXDIM];
double val_act;
extern "C"{
extern cube *acube;
extern qgrid agrid;
extern double axl[MAXDIM], axw[MAXDIM];
}


void get_products(char *products)  //read product information
{
  int i, j, k;
  char e;
  ifstream file, putback;

  file.open(products);
  if(!file)
  {
    printf("Could not read file %s\n", products);
    exit(0);
  }

  file >> num_p;  //number of products
  p = new product[num_p];

  for(i=0; i<num_p; i++)
  {
    file >> p[i].type >> p[i].ccy_id;
    p[i].ccy_idx = -1;
    switch(p[i].type)
    {
      case 1:  //stock
      file >> p[i].s_id >> p[i].n;
      p[i].s_idx = -1;
      break;

      case 2:  //ir-product
      file >> p[i].ir_id >> p[i].frn;
      if(p[i].frn==1)  //floating rate note
      {
        file >> p[i].frnt;  //beginning of floating rate
      }
      k = 0;
      file >> e;
      while(e != '|')
      {
        file.putback(e);
        file >> p[i].c[k];
        file >> e;
        k++;
      }
      p[i].num = k;
      for(j=0; j<p[i].num; j++)
      {
        file >> p[i].t[j];
      }
      p[i].ir_idx = -1;
      break;

      case 3:  //option
```

```
      file >> p[i].u_id >> p[i].sigma_id >> p[i].r_id
           >> p[i].k >> p[i].mat  >> p[i].cp >> p[i].ls;
      p[i].u_idx = -1;
      p[i].sigma_idx = -1;
      p[i].r_idx = -1;
      break;
    break;
    }
  }

  file.close();
}


void get_riskfactors(char *riskfactors)  //read riskfactor information
{
  int i, j, k;
  char e;
  ifstream file, putback;

  file.open(riskfactors);
  if(!file)
  {
    printf("Could not read file %s\n", riskfactors);
    exit(0);
  }

  file >> num_rf;  //number of riskfactors
  rf = new riskfactor[num_rf];

  for(i=0; i<num_rf; i++)
  {
    file >> rf[i].type >> rf[i].id;
    if(rf[i].type==2)  //ir
    {
      file >> rf[i].mapst_id >> rf[i].mapmt_id >> rf[i].maplt_id;
      k = 0;
      file >> e;
      while(e!='|')
      {
        file.putback(e);
        file >> rf[i].r[k];
        file >> e;
        k++;
      }
      for(j=0; j<k; j++)
      {
        file >> rf[i].t[j];
      }
      rf[i].mapst_idx = -1;
      rf[i].mapmt_idx = -1;
      rf[i].maplt_idx = -1;
    }
    else  //stock, fx, vola
    {
      file >> rf[i].map_id >> rf[i].val;
      rf[i].map_idx = -1;
    }
  }

  file.close();
}
```

```
void get_shifts(char *shifts)  //read shift information
{
  int i;
  ifstream file;

  file.open(shifts);
  if(!file)
  {
    printf("Could not read file %s\n", shifts);
    exit(0);
  }

  file >> dim;  //number of shifts
  s = new shift[dim];

  for(i=0; i<dim; i++)
  {
    file >> s[i].type >> s[i].id >> s[i].mod >> xu[i] >> xo[i];
  }

  file.close();
}


int main(int argc, char **argv)
{
  int i, iteration=0, index, refine;
  double err_cube, err_grid, err_new, tol;
  double x_act[MAXDIM]={0.0}, val_sep[1<<(MAXDIM-1)]={0.0};
  char approx[30], info[30];
  clock_t t0=clock();
  static FILE *infofile;

  if(argc!=8)
  {
    cout << "Usage: " << argv[0] << endl;
    cout << "\t products: file products contains portfolio data" << endl;
    cout << "\t riskfactors: file riskfactors contains specification of riskfactors"
         << endl;
    cout << "\t shifts: file shifts contains scenario information" << endl;
    cout << "\t ref_type: type of refinement (0/1)" << endl;
    cout << "\t err_type: type of error estimation (0/1)" << endl;
    cout << "\t approx_type: type of approximation (0/1)" << endl;
    cout << "\t tol: error tolerance" << endl;
    exit(0);
  }

  ref_type = atoi(argv[4]);  //0=refinement in all directions
                             //1=anisotropic refinement
  err_type = atoi(argv[5]);  //0=error estimation on current grid
                             //1=error estimation on previous grid
  approx_type = atoi(argv[6]);  //0=approximation of portfolio function
                                //1=approximation of profit and loss
  tol = atof(argv[7]);  //error tolerance
  get_products(argv[1]);  //read product information
  get_riskfactors(argv[2]);  //read riskfactor information
  get_shifts(argv[3]);  //read shift information
  assign_rf();  //prepare for valuation
  assign_shifts();  //prepare for valuation
  sprintf(approx, "approx[%c]_%s_%s_%s_%s.txt",
```

```
  argv[1][5], argv[4], argv[5], argv[6], argv[7]);
sprintf(info, "info[%c]_%s_%s_%s_%s.txt",
  argv[1][5], argv[4], argv[5], argv[6], argv[7]);
infofile = fopen(info, "wt");

if(infofile==NULL)
{
  fprintf(stderr, "Could not write file %s\n", infofile);
  return 1;
}
fprintf(infofile, "Iteration    #Quader    Fehlerschaetzer    Rechenzeit\n");

val_act = portfolio_val(x_act);  //current portfolio value
make_grid(xu, xo, dim, 2);  //step 1 of the algorithm

do  //iteration begins
{
  refdir = iteration % dim;  //direction of refinement
  index = first_cell();
  refine = 0;
  err_grid = 0.0;

  do  //step 2 of the algorithm
  {
    if(get_status_dir()==2)  //look at cubes with status 2
    {
      err_cube = get_err(val_sep);  //calculate error estimate
      if(err_cube<tol)  //cube is sufficiently refined
      {
        set_status(1);
      }
      else  //cube needs to be refined
      {
        if(ref_type==1)
        {
          set_refdir(acube, refdir);
        }
        if(err_type==1)
        {
          err_new = get_newval(acube, axl, axw, val_sep);
          set_err(acube, err_new);
        }
        if((acube->val_new = DOALLOC(1 << (dim - 1),double))==NULL)
        {
          printf("Storage failed\n");
          exit(0);
        }
        for(i=0; i<(1<<dim-1); i++)
        {
          acube->val_new[i] = val_sep[i];  //assign new values
        }
      }
      err_grid = MAX(err_grid,err_cube);
    }

    if(get_status()==2)  //status in direction other than refdir may still be 2
    {
      refine ++;
    }
    index = next_cell();

  }while(index!=-1);  //last cube is reached
```

```
    fprintf(infofile, "%5d %12d %15.4f %15.2f\n", iteration, agrid.max_simplex,
      err_grid, ((clock() - t0)/(double) CLOCKS_PER_SEC));
    refine_grid();  //step 4 & 5 of the algorithm
    iteration++;

  }while(((refine > 0) && (iteration < MAXITER)));  //step 6 of the algorithm

  fclose(infofile);
  write_grid(approx);  //write approximation in file approx

  delete []p;
  delete []rf;
  delete []s;

}
```

B.2 Modul `defs.h`

```
/*****************************************************************************
HEADER: defs.h
PURPOSE: standard definitions file
*****************************************************************************/

#ifndef __DEFS__
#define __DEFS__

#include <malloc.h>

extern int ref_type;


#define MAXITER 50
#define MAXDIM 10
#define NOLEAF 100
#define ISLEAF(that) ((that)->status<NOLEAF)
#define NUM_N (1<<dim)  //number of nodes per cubic element
#define NUM_TP ((ref_type) ? ((int)(pow((double) 3,(double) (dim-1)))) : \
((int)(pow((double) 3,(double) dim))))  //number of testpoints per cubic element
#define MAX(x, y) ((x) < (y) ? (y) : (x))
#define DOALLOC(nitems, type) ((type*) malloc((size_t) (nitems)*sizeof(type)))
#define UNALLOC(ptr) free((void*) (ptr))


struct product  //struct for portfolio data
{
  unsigned int type;  //1=stock, 2=ir, 3=option
  char ccy_id[20];  //currency
  int ccy_idx;  //index of fx-rf
  int flag;  //riskfactors found
  union
  {
    struct  //stock
    {
      char s_id[20];  //underlying
      int s_idx;  //index of underlying-rf
      double n;  //number of stocks
```

```
    };
    struct  //ir-product
    {
      char ir_id[20];  //interest curve
      int ir_idx;  //index of ir-rf
      unsigned int frn;  //0=fixed rate, 1=floating rate
      double frnt;  //beginning of floating rate
      double c[50];  //coupon
      double t[50];  //maturity in years
      int num;  //number of coupons
    };
    struct  //option
    {
      char u_id[20];  //underlying
      char sigma_id[20];  //volatility
      char r_id[20];  //discount-rate
      int u_idx;  //index of underlying-rf
      int sigma_idx;  //index of volatility-rf
      int r_idx;  //index of discount-rate-rf
      double k;  //strike
      double mat;  //maturity in years
      unsigned int cp;  //0=put, 1=call
      unsigned int ls;  //0=short position, 1=long position
    };
  };
};

struct riskfactor  //struct for riskfactor information
{
  int type;  //1=stock, 2=ir, 3=fx, 4=vola
  char id[20];  //name
  union
  {
    struct  //stock, fx, vola
    {
      double val;  //current value
      char map_id[20];  //shift
      int map_idx;  //index of shift
    };
    struct  //ir
    {
      char mapst_id[20];  //shift for short-term ir
      int mapst_idx;  //index of shift for short-term ir
      char mapmt_id[20];  //shift for middle-term ir
      int mapmt_idx;  //index of shift for middle-term ir
      char maplt_id[20];  //shift for long-term ir
      int maplt_idx;  //index of shift for long-term ir
      double r[50];  //interest curve
      double t[50];  //maturity in years
    };
  };
};

struct shift  //struct for shift information
{
  int type;  //1=stock, 2=ir, 3=fx, 4=vola
  char id[20];  //name
  int mod;  //0=absolute change, 1=relative change
};

typedef struct scube
{
```

```
  struct scube *upper;  //pointer to father element
  unsigned char status;  //1=done, 2=to be checked
  int status_dir[MAXDIM];  //status of refinement in direction dir
  union
  {
    struct scube **lower;  //pointer to son element
    int *nodes;  //nodes
  } l;
  double **val;  //pointer to values in nodes
  double *val_new;  //pointer to values for refinement
  unsigned char refdir;  //direction of refinement
  double err;  //total error of approximation in element
} cube;

typedef struct sqgrid
{
  cube *root;  //pointer to root element
  double ql[MAXDIM];  //lower left corner of root element
  double qx[MAXDIM];  //width of root element
  int max_simplex;  //number of leaf elements in grid
  int max_vertex;  //number of vertices in grid
  double val[1 << MAXDIM];  //portfolio values in nodes of root element
} qgrid;


#endif
```

B.3 Modul `grid.c`

```
/*****************************************************************************
MODULE: grid.c
PURPOSE: functions for cubic grid handling and adapting
*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include "grid.h"

extern dim;
extern int ref_type;
extern int err_type;
extern int approx_type;
extern int refdir;
extern double val_act;
extern struct shift *s;
extern double portfolio_val(double *shift);
extern double portfolio_approx(cube *that, double *x, double *xl, double *xw);
qgrid agrid;
cube *acube = NULL;
static int aindex;
static int aexists = 0;
double axw[MAXDIM], axl[MAXDIM];
static unsigned char szdir = 0;
static FILE *gridfile;
```

```
double make_grid(double *xu, double *xo, int dim, int n)
{  //creates start grid from lo to hi with n columns
  aexists = 1;

  return do_qgrid_refd(&agrid, xu, xo, n);
}


void refine_grid(void)
{  //refines current grid
  szdir = 1 << refdir;
  qadapt(&agrid);
}


int first_cell(void)
{  //sets pointer to the first element of the current grid
  cube *that;
  int i;
  unsigned char dir;

  if(!aexists)  //no grid found
    return -1;
  that = agrid.root;  //start with root element
  for(i=0; i<dim; i++)
  {
    axw[i] = agrid.qx[i];
    axl[i] = agrid.ql[i];
  }
  while(!ISLEAF(that))  //go down refinement tree to first leaf
  {
    dir = get_refdir(that);
    for(i=0; i<dim; i++)
    {
    if(dir==i)
      axw[i] /= 2.0;  //calculate width of element
    }
    that = that->l.lower[0];
  }
  acube = that;
  aindex = 0;

  return 0;
}


cube *next_cell_rec(cube *that)
{  //recursive help function for next_cell
  int i;
  unsigned char dir;
  cube *unext;

  if(that->upper == NULL)  //root element
    return NULL;
  dir = get_refdir(that->upper);

  if(get_pos(that)==0)  //left son element
  {
    for(i=0; i<dim; i++)
    {
      if(dir==i)
```

```
      axl[i] += axw[i];
    }
    unext = that->upper->l.lower[1];  //next cell is right son element
  }
  else  //right son element
  {
    for(i=0; i<dim; i++)
    {
      if(dir==i)
      {
      axl[i] -= axw[i];
      axw[i] *= 2.0;
      }
    }
    unext = next_cell_rec(that->upper);
  }
  if(unext == NULL)
    return NULL;

  while(unext->l.lower!=NULL)  //go down refinement tree
  {
    dir = get_refdir(unext);
    for(i=0; i<dim; i++)
    {
      if(dir==i)
      axw[i] /= 2.0;
    }
    unext = unext->l.lower[0];
  }

  return unext;
}


int next_cell(void)
{  //sets pointer to the next element of the current grid
  acube = next_cell_rec(acube);

  if(acube==NULL)
    return -1;
  else
    return ++aindex;
}


void get_corner(int i, double *x)
{  //returns corner i of current element
  get_coord(i, axl, axw, x);
}


static int write_grid_that(qgrid *tg, cube *that, double *xl, double *xw)
{  //help function for write_grid
  int k;
  double x[MAXDIM];

  if(!ISLEAF(that))
    return 0;

  get_coord(0, xl, xw, x);
  for(k=0; k<dim; k++)
  {
```

```
    fprintf(gridfile, "%12.8f ", x[k]);  //lower left corner
  }
  fprintf(gridfile, "   ");
  get_coord((NUM_N - 1), xl, xw, x);
  for(k=0; k<dim; k++)
  {
    fprintf(gridfile, "  %12.8f ", x[k]);  //upper right corner
  }
  fprintf(gridfile, "   ");
  for(k=0; k<NUM_N; k++)
  {
    fprintf(gridfile, "  %12.8f ", *that->val[k]);  //values on corners
  }
  fprintf(gridfile, "  %d\n", that->status);  //status

  return 1;
}


int write_grid(char *filename)
{  //writes current grid and approximation in file filename
  int i;

  gridfile = fopen(filename, "wt");
  if(gridfile==NULL)
  {
    fprintf(stderr, "Could not write file %s\n", filename);
    return 1;
  }
  fprintf(gridfile, "%% %d", dim);  //dimension
  for(i=0; i<dim; i++)
  {
    fprintf(gridfile, " %s ", s[i].id);  //names of riskfactors
  }
  fprintf(gridfile, "\n");
  all_cubes(&agrid, write_grid_that);
  fclose(gridfile);

  return 0;
}


double get_err(double *val_sep)
{  //returns error estimation of current element
  int t=0, i;
  double err=0.0, value, approx, tp[dim];
  cube *father;

  if(err_type==0)  //error estimation on current grid
  {
    if(ref_type==1)  //only new nodes as testpoints
    {
      err = get_newval(acube, axl, axw, val_sep);
    }
    else  //3 testpoints per dim, also possible for ref_type=1
    {
      for(i=0; i<NUM_TP; i++)  //run through all testpoints
      {
        get_testpoint(i,tp);
        approx = portfolio_approx(acube, tp, axl, axw);  //approximated value
        if(approx_type==0)
        {
```

```
          value = portfolio_val(tp);  //exact value
          err =  MAX(err,fabs((value - approx)/value));
        }
        else if(approx_type==1)
        {
          value = ((portfolio_val(tp) - val_act) / fabs(val_act));  //exact value
          err =  MAX(err,fabs(value - approx));
        }
        if(i==get_newval_idx(t))
        {
          val_sep[t] = value;  //assign potential new values
          t++;
        }
      }
    }
  }
  else if(err_type==1)  //error estimation on previous grid
  {
    father = acube;
    if(ref_type==0)  //refinement in all directions
    {
      for(i=0; i<dim; i++)
      {
        err = MAX(err, father->upper->err);
        father = father->upper;
      }
    }
    else if(ref_type==1)  //anisotropic refinement
    {
      for(i=0; i<dim; i++)
      {
        if(father->upper->refdir==refdir)
        {
          err = father->upper->err;
          break;
        }
        else
          father = father->upper;
      }
    }
  }

  return err;
}


int set_status(int status)
{  //sets status of current element to status
  int i, state=0;

  if(acube==NULL)
    return -1;
  else
  {
    if(ref_type==1)  //anisotropic refinement
    {
      acube->status_dir[refdir] = (unsigned char) status;
      for(i=0; i<dim; i++)
      {
        state = MAX(state, acube->status_dir[i]);
      }
      acube->status = (unsigned char) state;  //update status
```

```
    }
    else
    {
      acube->status = (unsigned char) status;
    }
  }

  return 0;
}


int get_status(void)
{  //returns status of current element
  if(acube==NULL)
    return -1;
  else
    return (int) acube->status;
}


int get_status_dir(void)
{  //returns status_dir of current element
  int i, status=0;

  if(acube==NULL)
    return -1;
  else
  {
    if(ref_type==1)  //anisotropic refinement
    {
      return (int) acube->status_dir[refdir];
    }
    else
    {
      return (int) acube->status;
    }
  }
}


void set_refdir(cube *that, int dir)
{  //sets refinement direction of cube that to dir
  that->refdir = (unsigned char) dir;
}


unsigned char get_refdir(cube *that)
{  //returns refinement direction of cube that
  int cnt;

  if(ISLEAF(that)) return 0;  //cube not yet refined
  if(ref_type==1)
  {
    return (unsigned char) that->refdir;
  }
  else
  {
    cnt = that->status-NOLEAF;
    return (unsigned char)(cnt % dim);
  }
}
```

```
unsigned char get_rdir(cube *that)
{  //returns (1<<refinement direction) of cube that
  int cnt;

  if(ISLEAF(that)) return 0;  //cube not yet refined

  if(ref_type==1)
  {
    return (unsigned char) (1 << that->refdir);
  }
  else
  {
    cnt = that->status-NOLEAF;
    return (unsigned char)(1 << (cnt % dim));
  }
}


void get_testpoint(int i, double *x)
{  //stores testpoint i of current element in x
  if(i>=NUM_TP)
  {
    i = NUM_TP - 1;
  }
  get_tcoord(i, axl, axw, x);
}


double get_newval(cube *that, double *xl, double *xw, double *val_sep)
{  //stores values for refinement in val_sep and returns error of this refinement
  int i;
  double x[dim], approx, err=0.0;

  for(i=0; i<(1<<(dim-1)); i++)
  {
    get_sepcoord(i,that, xl,xw,x);
    approx = portfolio_approx(that, x, xl, xw);
    if(approx_type==0)
    {
      val_sep[i] = portfolio_val(x);
      err = MAX(err, fabs((val_sep[i] - approx)/val_sep[i]));
    }
    else if(approx_type==1)
    {
      val_sep[i] = ((portfolio_val(x)- val_act) / fabs(val_act));
      err = MAX(err, fabs(val_sep[i] - approx));
    }
  }

return err;
}


void set_err(cube *that, double err)
{  //sets err of cube that to err
  that->err = err;
}


int get_newval_idx(int t)
{  //returns index of next new value
```

```
  int c=1, sum=0, i, tmp;

  if(ref_type==0)
  {
    tmp = refdir;
  }
  else if(ref_type==1)
  {
    tmp =  dim-1;
  }

  for(i=0; i<dim; i++)
  {
    if(tmp!=i)
    {
      if((1 << c - 1) & t)
      {
        sum += 2 * (int) pow(3., (double) i);
      }
      c++;
    }
    else
    {
      if(ref_type==0)
      {
        sum += (int) pow(3., (double) i);
      }
    }
  }

  return sum;
}


int get_pos(cube *that)
{  //returns if cube that is left or right son
  if(that->upper==NULL)
    return 0;

  else if(that==that->upper->l.lower[0])
    return 0;  //left son element

  else
    return 1;  //right son element
}


void get_posinfo(int ipos, int pos[])
{  //stores position of testpoint ipos in pos
  int i, dir;

  dir = refdir;
  for(i=0; i<dim; i++)
  {
    if((ref_type==1) && (dir==i))
    {
      pos[i] = 1;
    }
    else
    {
      pos[i] = ipos % 3;
      ipos = (int) ipos / 3;
```

```
    }
  }
}


void get_sepposinfo(int ipos, cube *that, int pos[])
{  //stores position of new node ipos in pos
  int i, tmp=0;

  for(i=0; i<dim; i++)
  {
    if(refdir==i)
    {
      pos[i] = 1;
      tmp = 1;
    }
    else
    {
      pos[i] = 2 * (ipos & (1<<(i-tmp))) / (1<<(i-tmp));
    }
  }
}


void get_coord(int i, double *xl, double *xw, double *coord)
{  //stores coordinates of corner i in coord
  int j;

  for(j=0; j<dim; j++)
  {
    coord[j] = ((i & (1 << j)) ? xl[j] + xw[j] : xl[j]);
  }
}


void get_tcoord(int i, double *xl, double *xw, double *coord)
{  //stores coordinates of testpoint i in coord
  int j;
  int pos[MAXDIM];

  get_posinfo(i, pos);

  for(j=0; j<dim; j++)
  {
    coord[j] = xl[j] + (double) pos[j] * xw[j] / 2.0;
  }
}


void get_sepcoord(int i, cube *that, double *xl, double *xw, double *coord)
{  //stores coordinates of new node i in coord
  int j;
  int pos[MAXDIM];

  get_sepposinfo(i, that, pos);

  for(j=0; j<dim; j++)
  {
    coord[j] = xl[j] + (double) pos[j] * xw[j] / 2.0;
  }
}
```

```
int refine_that(qgrid *tg, cube *that, int virtual, unsigned char refdir)
{  //refines cube that, does all allocation
  int i, j, rdir;

  if(!ISLEAF(that))
  {
    fprintf(stderr, "Attempted to refine refined cube\n");
    return -1;
  }
  else
  {
    if(that->l.nodes != NULL)
      UNALLOC(that->l.nodes);
    that->l.lower = DOALLOC(2, cube *);  //initialize new cubes

    for(i=0; i<2; i++) //allocation
    {
      that->l.lower[i] = DOALLOC(1, cube);
      that->l.lower[i]->upper = that;
      that->l.lower[i]->l.nodes = NULL;
      that->l.lower[i]->val = DOALLOC(NUM_N, double *);
      that->l.lower[i]->status = that->status;
      if(ref_type==1)
      {
        for(j=0; j<dim; j++)
        {
          that->l.lower[i]->status_dir[j]=that->status_dir[j];
        }
      }
    }

    if(that->upper==NULL)
    {
      that->status = NOLEAF;
    }
    else  //update status
      (that->status = that->upper->status + 1);

    rdir = 1<<refdir;

    for(i=0; i<2; i++)
    {
      for(j=0; j<NUM_N; j++)  //assign values in new nodes
      {
      that->l.lower[i]->val[j] = ((i * rdir) ^ (j & rdir)) ?
        &that->val_new[j - ((int)(j /(2 * rdir)) + 1 - i) * rdir] : that->val[j];
      }
    }

    if(that->val!=NULL)
      UNALLOC(that->val);
    tg->max_simplex++;  //number of leaf elements
  }

  return 0;
}


static int refine_refdir_that(qgrid *tg, cube *that, double *xl, double *xw)
{  //refines cube that in direction refdir
  int i;
```

```
  double val[1<<(dim-1)], err;

  if(ISLEAF(that))
  {
    if(ref_type==1)
    {
      set_refdir(that, refdir);
    }
    err = get_newval(that, xl, xw, val);  //get values on new nodes
    if(err_type==1)
    {
      set_err(that, err);
    }
    if((that->val_new=DOALLOC(1 << (dim - 1),double))==NULL)
    {
      printf("Storage failed\n");
      exit(0);
    }
    for(i=0; i<(1<<dim-1); i++)
    {
      that->val_new[i] = val[i];
    }
    refine_that(tg, that, 0, refdir);
  }

  return 0;
}


static void refine_all(qgrid *tg, int numref)
{  //refines grid tg in all directions
  int i, j;

  for(j=0; j<numref; j++)
  {
    for(i=0; i<dim; i++)
    {
      refdir = i;
      all_cubes(tg, refine_refdir_that);
    }
  }
}


double do_qgrid_refd(qgrid *tg, double *lo, double *hi, int n)
{  //creates a basic cubic grid from lo to hi with n columns
  int i, numref;
  double k=0;

  numref = 0;
  while(n>1)
  {
    numref++;
    n /= 2;
  }
  k = do_qgrid(tg, lo, hi);
  k /= pow(2, numref);
  refine_all(tg, numref);

  return k;
}
```

```
double do_qgrid(qgrid *tg, double *lo, double *hi)
{  //creates a basic cubic grid from lo to hi with 1 column, does all allocation
  int i;
  double k=0, x[dim];

  tg->max_simplex = 1;
  for(i=0; i<dim; i++)
  {
    tg->qx[i] = (hi[i] - lo[i]);  //width of root element
    tg->ql[i] = lo[i];  //lower left corner of root element
    k += (tg->qx[i]) * (tg->qx[i]);
  }

  tg->root = DOALLOC(1, cube);  //initialize cube

  tg->root->l.nodes = NULL;  //allocation
  tg->root->upper = NULL;
  tg->root->status = 2;
  if(ref_type==1)
  {
    for(i=0; i<dim; i++)
    {
      tg->root->status_dir[i]=2;
    }
  }

  tg->root->val = DOALLOC(NUM_N, double *);

  for(i=0; i<NUM_N; i++)  //assign values in nodes
  {
    get_coord(i,tg->ql,tg->qx,x);
    if(approx_type==0)
    {
      tg->val[i] = portfolio_val(x);
    }
    else if(approx_type==1)
    {
      tg->val[i] = ((portfolio_val(x) - val_act) / fabs(val_act));
    }
    tg->root->val[i] = &tg->val[i];
  }
  tg->max_vertex = 0;

  return sqrt(k);
}


static int rec_cubes(qgrid *tg, int (*todo)(qgrid *, cube *, double *, double *),
          cube *that, double *xl, double *xw)
{  //recursive help function for all_cubes
  int i, j, dir;
  double xwn[MAXDIM], xln[MAXDIM];
  int retval=0;

  if(!ISLEAF(that))  //go down refinement tree until leaf is found
  {
    dir=get_refdir(that);
    for(i=0; i<2; i++)
    {
      for(j=0; j<dim; j++)
      {
```

```
      if(dir==j)
      {
        xwn[j] = xw[j] / 2.0;
        xln[j] = xl[j] + (double) i *xwn[j];
        }
        else
      {
        xwn[j] = xw[j];
        xln[j] = xl[j];
      }
      }
      if(dim==1)
      xwn[1] = xln[1] = 0.0;
      retval += rec_cubes(tg, todo, that->l.lower[i], xln, xwn);
    }
  }

  return (retval + (*todo)(tg, that, xl, xw));
}


int all_cubes(qgrid *tg, int (*todo)(qgrid *, cube *, double *, double *))
{  //this function runs bottom-up through all cubes and lets todo act on every cube
  double xl[MAXDIM];
  int i, retval=0;

  retval += rec_cubes(tg, todo, tg->root, tg->ql, tg->qx);

  if(retval<0)
    return retval;

  return retval;
}


static int rec_cubes_nocoord(qgrid *tg, int (*todo)
           (qgrid *, cube *, double *, double *), cube *that)
{  //recursive help function for all_cubes_nocoord
  int i, j;
  int retval=0;

  if(!ISLEAF(that))  //go down refinement tree until leaf is found
  {
    for(i=0; i<2; i++)
      retval += rec_cubes_nocoord(tg, todo, that->l.lower[i]);
  }

  return (retval + (*todo)(tg, that, NULL, NULL));
}


int all_cubes_nocoord(qgrid *tg, int (*todo)(qgrid *, cube *, double *, double *))
{  //this function runs bottom-up through all cubes and lets todo act on every cube
   //this version does not compute coordinates
  int i, retval=0;

  retval += rec_cubes_nocoord(tg, todo, tg->root);
  if(retval<0)
    return retval;

  return retval;
}
```

```
static int qadapt_that(qgrid *tg, cube *that, double *xl, double *xw)
{
  if((!ISLEAF(that)) || (get_rdir(that)==szdir))
    return 0;

  if(ref_type==1)  //anisotropic refinement
  {
    if(that->status_dir[refdir]==2)
    {
      refine_that(tg, that, 0, refdir);
    }
  }
  else  //refinement in all directions
  {
    if(that->status==2)
    {
      refine_that(tg, that, 0, refdir);
    }
  }

  return 0;
}


void qadapt(qgrid *tg)
{  //head function for adapting according to the error estimates
  if(szdir!=0)
  {
    all_cubes_nocoord(tg, qadapt_that);
  }
}
```

B.4 Modul grid.h

```
/*****************************************************************************
HEADER: grid.h
PURPOSE: functions for cubic grid handling and adapting
*****************************************************************************/

#include "defs.h"


double make_grid(double *xu, double *xo, int dim, int n);

void refine_grid(void);

int first_cell(void);

int next_cell(void);

void get_corner(int i, double *x);

int write_grid(char *filename);

double get_err(double *val_sep);
```

```
int set_status(int status);
int get_status(void);
int get_status_dir(void);
void set_refdir(cube *that, int dir);
unsigned char get_refdir(cube *that);
unsigned char get_rdir(cube *that);
void get_testpoint(int i, double *x);
double get_newval(cube *that, double *xl, double *xw, double *val_sep);
void set_err(cube *that, double err);
int get_newval_idx(int t);
int get_pos(cube *that);
void get_posinfo(int ipos, int pos[]);
void get_sepposinfo(int ipos, cube *that, int pos[]);
void get_coord(int i, double *xl, double *xw, double *coord);
void get_tcoord(int i, double *xl, double *xw, double *coord);
void get_sepcoord(int i, cube *that, double *xl, double *xw, double *coord);
int refine_that(qgrid *tg, cube *that, int nvirtual, unsigned char refdir);
double do_qgrid_refd(qgrid *tg, double *lo, double *hi, int n);
double do_qgrid(qgrid *tg, double *lo, double *hi);
int all_cubes(qgrid *tg, int (*todo)(qgrid *, cube *, double *, double *));
int all_cubes_nocoord(qgrid *tg, int (*todo)(qgrid *, cube *, double *, double *));
void qadapt(qgrid *tg);
```

B.5 Modul `valuation.cpp`

```
/*************************************************************************
MODULE: valuation.cpp
PURPOSE: exact valuation and approximation of the portfolio value
*************************************************************************/

#include <iostream>
#include <cmath>
#include "defs.h"

using namespace std;
```

```
extern int dim;
extern int num_p;
extern int num_rf;
extern struct product *p;
extern struct riskfactor *rf;
extern struct shift *s;


int get_rf_idx(int type, string id, int *flag)  //look for corresponding riskfactor
{
  int i;

  for(i=0; i<num_rf; i++)  //scan riskfactors
  {
    if((type==rf[i].type) && (id==rf[i].id))
    {
      return i;  //corresponding riskfactor found
    }
  }
  *flag = 0;  //valuation of product not possible

  return -1;  //no corresponding riskfactor found
}


void assign_rf(void) //allocate corresponding riskfactors to products
{
  int i;
  char tmp[20];

  for(i=0; i<num_p; i++)
  {
    p[i].flag = 1;
    if(strcmp(p[i].ccy_id, "EUR"))  //currency
    {
      strcpy(tmp, p[i].ccy_id);
      strcat(tmp,"\\_EUR");
      p[i].ccy_idx = get_rf_idx(3, tmp, &p[i].flag);
    }

    if(p[i].type==1)  //stock
    {
      p[i].s_idx = get_rf_idx(1, p[i].s_id, &p[i].flag);
    }
    else if(p[i].type==2)  //ir-product
    {
      p[i].ir_idx = get_rf_idx(2, p[i].ir_id, &p[i].flag);
    }
    else if(p[i].type==3)  //option
    {
      p[i].u_idx = get_rf_idx(1, p[i].u_id, &p[i].flag);
      p[i].sigma_idx = get_rf_idx(4, p[i].sigma_id, &p[i].flag);
      p[i].r_idx = get_rf_idx(2, p[i].r_id, &p[i].flag);
    }

    if(p[i].flag==0)  //not all corresponding riskfactors allocated
    {
      printf("Product %d cannot be valuated\n", i);
    }
  }
}
```

```
int get_shift_idx(int type, string id)  //look for corresponding shift
{
  int i;

  for(i=0; i<dim; i++)  //scan shifts
  {
    if((type==s[i].type) && (id==s[i].id))
    {
      return i;  //corresponding shift found
    }
  }

  return -1;  //no corresponding shift found, riskfactor will not be shifted
}


void assign_shifts(void)  //allocate corresponding shifts to riskfactors
{
  int i;

  for(i=0; i<num_rf; i++)
  {
    if(rf[i].type==2)  //ir
    {
      rf[i].mapst_idx = get_shift_idx(rf[i].type, rf[i].mapst_id);
      rf[i].mapmt_idx = get_shift_idx(rf[i].type, rf[i].mapmt_id);
      rf[i].maplt_idx = get_shift_idx(rf[i].type, rf[i].maplt_id);
    }
    else  //stock, fx, vola
    {
      rf[i].map_idx = get_shift_idx(rf[i].type, rf[i].map_id);
    }
  }
}


double get_rf(double rf_act, int i, double *x)  //calculate shifted rf-value
{
  double rf;

  if(i<0)  //no shift
  {
    return rf_act;
  }
  if(s[i].mod==0)  //absolute change
  {
    rf = rf_act + x[i];
  }
  else if(s[i].mod==1)  //relative change
  {
    rf = rf_act * (1 + x[i]);
  }

  return rf;
}


double get_fx(double value, int i, double *x)  //calculate value in EUR
{
  double fx, value_fx;
```

```
  if(i<0)  //value already in EUR
  {
    return value;
  }
  fx = get_rf(rf[i].val, rf[i].map_idx, x);
  value_fx = value * fx;

  return value_fx;
}


double stock_val(int i, double *x)  //valuation of stocks
{
  double s, value;

  s = get_rf(rf[p[i].s_idx].val, rf[p[i].s_idx].map_idx, x);
  value = p[i].n * s;
  value = get_fx(value, p[i].ccy_idx, x);

  return value;
}


double interest_val(int i, double *x)  //valuation of ir-products
{
  int j, k;
  double r, value=0.0;

  for(k=0; k<p[i].num; k++)
  {
    j = 0;
    while(p[i].t[k]>=rf[p[i].ir_idx].t[j])
    {
      j++;
    }
    //linear interpolation in interest rates if necessary
    r = rf[p[i].ir_idx].r[j-1] + ((p[i].t[k]-rf[p[i].ir_idx].t[j-1]) /
       (rf[p[i].ir_idx].t[j] - rf[p[i].ir_idx].t[j-1]) * (rf[p[i].ir_idx].r[j] -
        rf[p[i].ir_idx].r[j - 1]));
    if((p[i].frn==0) || ((p[i].frn==1) && (p[i].t[k]<=p[i].frnt)))
    {
      if(p[i].t[k]<=0.25)
      {
        r = get_rf(r, rf[p[i].ir_idx].mapst_idx, x);
      }
      else if((p[i].t[k]>5.0))
      {
        r = get_rf(r, rf[p[i].ir_idx].maplt_idx, x);
      }
      else
      {
        r = get_rf(r, rf[p[i].ir_idx].mapmt_idx, x);
      }
    }
    r = MAX(0.0, r);  //interest rates may not be negative
    value += p[i].c[k] / exp(r * p[i].t[k]);
  }
  value = get_fx(value, p[i].ccy_idx, x);

  return value;
}
```

```
double N(const double y)  //normal distribution
{
  if(y>6.0)
  {
    return 1.0;
  }
  if(y<-6.0)
  {
    return 0.0;
  }
  double b1 = 0.31938153;
  double b2 = -0.356563782;
  double b3 = 1.781477937;
  double b4 = -1.821255978;
  double b5 = 1.330274429;
  double p = 0.2316419;
  double c2 = 0.3989423;
  double a = fabs(y);
  double t = 1.0 / (1.0 + a * p);
  double b = c2 * exp((-y) * (y/2.0));
  double n = ((((b5 * t + b4) * t + b3) * t + b2) * t + b1) * t;
  n = 1.0 - b * n;
  if(y<0.0)
  {
    n = 1.0 - n;
  }

  return n;
}


double option_val(int i, double *x)  //valuation of options via black-scholes
{
  int j;
  double value, d1, d2, s, sigma, r;

  s = get_rf(rf[p[i].u_idx].val, rf[p[i].u_idx].map_idx, x);
  sigma = get_rf(rf[p[i].sigma_idx].val, rf[p[i].sigma_idx].map_idx, x);

  j = 0;
  while(p[i].mat>=rf[p[i].r_idx].t[j])
  {
    j++;
  }
   //linear interpolation in interest rates if necessary
  r = rf[p[i].r_idx].r[j-1] + ((p[i].mat-rf[p[i].r_idx].t[j-1])/(rf[p[i].r_idx].t[j] -
      rf[p[i].r_idx].t[j-1]) * (rf[p[i].r_idx].r[j]-rf[p[i].r_idx].r[j - 1]));

  if(p[i].mat<=0.25)  //short-term
  {
    r = get_rf(r, rf[p[i].r_idx].mapst_idx, x);
  }
  else if((p[i].mat>5.0))  //long-term
  {
    r = get_rf(r, rf[p[i].r_idx].maplt_idx, x);
  }
  else  //middle-term
  {
    r = get_rf(r, rf[p[i].r_idx].mapmt_idx, x);
  }
```

```
  r = MAX(0.0, r);  //interest rates may not be negative

  d1=(log(s/p[i].k)+r*p[i].mat) / (sigma*sqrt(p[i].mat)) + 0.5*sigma*sqrt(p[i].mat);
  d2=d1 - (sigma * sqrt(p[i].mat));

  if(p[i].cp==1)  //call
  {
    value = s * N(d1) - p[i].k * exp(-r*p[i].mat) * N(d2);
  }
  else  //put
  {
    value = -s * N(-d1) + p[i].k * exp(-r*p[i].mat) * N(-d2);
  }
  if(p[i].ls==0)  //short position
  {
    value = value * (-1);
  }
  value = get_fx(value, p[i].ccy_idx, x);

 return value;
}


extern "C" double portfolio_val(double *x)  //exact valuation in x
{
  int i;
  double value=0.0;

  for(i=0; i<num_p; i++)  //run through all products
  {
    if(p[i].flag !=0)  //all riskfactors assigned
    {
      if(p[i].type==1)  //stock
      {
        value += stock_val(i, x);
      }
      else if(p[i].type==2)  //ir-product
      {
        value += interest_val(i, x);
      }
      else if(p[i].type==3)  //option
      {
        value += option_val(i, x);
      }
      else  //product not known
      {
        printf("Product %d not known\n", i);
      }
    }
  }

  return value;
}


extern "C" double portfolio_approx(cube *that, double *x, double *xl, double *xw)
//approximation in x
{
  int i, j;
  double y[dim], coord[NUM_N], value=0.0;

  for(i=0; i<dim; i++)  //calculate parameters
```

```
  {
    y[i] = (x[i] - xl[i]) / xw[i];
  }
  for(i=0; i<NUM_N; i++)
  {
    coord[i] = 1;
    for(j=0; j<dim; j++)
    {
      coord[i] *= ((i & (1 << j))) ? y[j] : (1.0 - y[j]);
    }
  }
  for(i=0; i<NUM_N; i++)  //multilinear interpolation in nodes of element
  {
    value += coord[i] * *that->val[i];
  }

  return value;
}
```

B.6 Modul `valuation.h`

```
/*****************************************************************************
HEADER: valuation.h
PURPOSE: exact valuation and approximation of the portfolio value
*****************************************************************************/

extern "C"{
#include "grid.h"
}

using namespace std;


int get_rf_idx(int type, string id, int *flag);

void assign_rf(void);

int get_shift_idx(int type, string id);

void assign_shifts(void);

double get_rf(double rf_act, int i, double *x);

double get_fx(double value, int i, double *x);

double stock_val(int i, double *x);

double interest_val(int i, double *x);

double N(const double y);

double option_val(int i, double *x);

extern "C" double portfolio_val(double *x);

extern "C" double portfolio_approx(cube *that, double *x, double *xl, double *xw);
```

B.7 Modul post_eval.cpp

```
/*****************************************************************************
MODULE: post_eval.cpp
PURPOSE: additional evaluation of the approximate portfolio value in grid
*****************************************************************************/

#include <iostream>
#include <fstream>

using namespace std;


double portfolio_approx(double *x, double *xu, double *xo, double *val, int dim)
//approximation in x
{
  int i, j;
  double y[dim], coord[1<<dim], value=0.0;

  for(i=0; i<dim; i++)  //calculate parameters
  {
    y[i] = (x[i] - xu[i]) / (xo[i] - xu[i]);
  }
  for(i=0; i<(1<<dim); i++)
  {
    coord[i] = 1;
    for(j=0; j<dim; j++)
    {
      coord[i] *= ((i & (1 << j))) ? y[j] : (1.0 - y[j]);
    }
  }
  for(i=0; i<(1<<dim); i++)  //multilinear interpolation in nodes of element
  {
    value += coord[i] * val[i];
  }

  return value;
}


int main(int argc, char **argv)
{
  int i, dim, status, tmp;
  double approx;
  char z;
  ifstream file;

  file.open(argv[1]);  //read grid information
  if(!file)
  {
    cout << "Could not read file!" << endl;
    exit(0);
  }
  file >> z >> dim;

  double x[1<<dim], xu[dim], xo[dim], val[1<<dim];
  string name[dim];

  for(i=0; i<dim;i++)
  {
```

```
    file >> name[i];
  }

  if(argc!=dim+2)
  {
    cout << "Usage: " << argv[0] << endl;
    cout << "\t approx: file approx contains approximation and grid data" << endl;
    cout << "\t x[1],...,x[dim]: point at which portfolio value should be approximated"
         << endl;
    exit(0);
  }
  else
  {
    for(i=0; i<dim; i++)
    {
      x[i] = atof(argv[i + 2]);
    }
  }

  do  //run through cubes
  {
    if(file.eof())  //reached end of file
    {
      cout << "x outside grid" << endl;
      return -1;
    }

    for(i=0; i<dim; i++)
    {
      file >> xu[i];  //lower left corner
    }
    for(i=0; i<dim; i++)
    {
      file >> xo[i]; //upper right corner
    }
    for(i=0; i<(1<<dim); i++)
    {
      file >> val[i];  //portfolio values
    }
    file >> status;  //status
    tmp = 0;
    for(i=0; i<dim; i++)
    {
      if((x[i] < xu[i]) || (x[i] > xo[i]))  //x not in this cube
      {
      tmp++;
      }
    }
  }while(tmp!=0);  //cube not yet found

  file.close();

  approx = portfolio_approx(x, xu, xo, val, dim);

  cout << "Portfolio value or change in portfolio value at point x=(";
  for(i=0; i<dim-1; i++)
  {
    cout << x[i] << ",";
  }
  cout << x[dim-1] << ") is " << approx << endl;

}
```

Anhang C

MATLAB-Quelltext

An dieser Stelle sind die MATLAB-Funktionen `surfaceplot.m` und `gridplot.m` zur grafischen Darstellung der Portfoliofunktion und des Gitters abgedruckt. Für genauere Information zu Funktion und Aufruf der Programme sei auf Kapitel 7 verwiesen.

C.1 Funktion `surfaceplot.m`

```
function surfaceplot(file,x,y,varargin)
%draws a plot of the approximation function

%read dimension and names of riskfactors
[dim] = textread(file,'%*s %d',1);
[R] = textread(file,'%s',dim+2);
for i=1:dim
    rf(i)=R(i+2);
end

%read file
f = load(file);
sf = size(f);

cube = ones(1,dim)*2;
dimarray = cell(length(varargin{1}),1);

figure;
grid on;
hold on;

%assign axes
miny = min([f(:,x);f(:,x+dim)]);
maxy = max([f(:,x);f(:,x+dim)]);
minx = min([f(:,y);f(:,y+dim)]);
maxx = max([f(:,y);f(:,y+dim)]);
axis([minx,maxx,miny,maxy]);

%plot
for i=1:sf(1) %run through all cubes
    %read axes and function values
    a=[f(i,x),f(i,x+dim)];
```

```
    b=[f(i,y),f(i,y+dim)];
    for j=1:2^dim
        val(j)=f(i,2*dim+j);
    end
    %permutation of the function value matrix, so that dimension x and y can be displayed
    valmat=reshape(val,cube);
    c=permute(valmat,[x y varargin{1}]);
    tmp=0;
    %check if cube lies in specified area
    for k=1:length(varargin{1})
        if (varargin{2}(k)>f(i,varargin{1}(k))& varargin{2}(k)<=f(i,varargin{1}(k)+dim))
            dimarray{k}=2;
        else
        dimarray{k}=3;
        tmp=tmp+1;
        end
    end
    %plot function on cube
    if tmp==0
        surf(b,a,c(:,:,dimarray{:,1}));
    end
end

%window settings
S='Plot der Portfoliofunktion';
T= 'mit ';
for i=1:length(varargin{1})
    T=[T char(rf(varargin{1}(i))) ' = ' num2str(varargin{2}(i))];
    if i<length(varargin{1})
        T=[T ', '];
    end
end
if dim==2
    title(S,'Fontsize',12,'FontWeight','bold');
else
    title({S; T},'Fontsize',12,'FontWeight','bold');
end
xlabel(rf(y),'Fontsize',12);
ylabel(rf(x),'Fontsize',12);
zlabel(['v(' char(rf(y)) ',' char(rf(x)) ')'],'Fontsize',12);
view(-37.5,30);
hold off;
```

C.2 Funktion `gridplot.m`

```
function gridplot(file,x,y,z,varargin)
%draws a plot of the grid used for the approximation

%read dimension and names of riskfactors
[dim] = textread(file,'%*s %d',1);
[R] = textread(file,'%s',dim+2);
for i=1:dim
    rf(i)=R(i+2);
end

%read file
f  = load(file);
```

```
sf = size(f);

p1 = [0,1,1,0;...
      0,1,1,0;...
      0,0,0,0;...
      1,1,1,1;...
      0,1,1,0;...
      1,0,0,1];
p2 = [0,0,0,0;...
      1,1,1,1;...
      0,0,1,1;...
      0,0,1,1;...
      0,0,1,1;...
      0,0,1,1];
p3 = [0,0,1,1;...
      0,0,1,1;...
      1,0,0,1;...
      0,1,1,0;...
      0,0,0,0;...
      1,1,1,1];

figure;
hold on;

minx = min([f(:,x);f(:,x+dim)]);
maxx = max([f(:,x);f(:,x+dim)]);
miny = min([f(:,y);f(:,y+dim)]);
maxy = max([f(:,y);f(:,y+dim)]);


%plot
if dim>2 %three-dimensional plot of the grid if dim>2

%assign axes
minz = min([f(:,z);f(:,z+dim)]);
maxz = max([f(:,z);f(:,z+dim)]);
axis([minx,maxx,miny,maxy,minz,maxz]);

for i=1:sf(1) %run through all cubes
    %check if cube lies in specified area
    tmp=0;
    for k=1:length(varargin{1})
        if ~(varargin{2}(k)==f(i,varargin{1}(k)) |...
            varargin{2}(k)==f(i,varargin{1}(k)+dim))
            tmp=tmp+1;
        end
    end
    %plot cube
    if tmp==0
        for j=1:6
            a=f(i,dim.*p1(j,:)+x);
            b=f(i,dim.*p2(j,:)+y);
            c=f(i,dim.*p3(j,:)+z);
            if (f(i,(dim*2+2^dim+1))==1) %cube sufficiently refined
                plot3(a,b,c,'k');
            elseif (f(i,(dim*2+2^dim+1))==2) %cube not sufficiently refined
                plot3(a,b,c,'r');
            end
        end
    end
end
```

```
else %two-dimensional plot of the grid if dim=2

%assign axes
axis([minx,maxx,miny,maxy]);

for i=1:sf(1) %run through all cubes
    %plot
    a=[f(i,1), f(i,3), f(i,3), f(i,1)];
    b=[f(i,2), f(i,2), f(i,4), f(i,4)];
    if (f(i,9)==1) %cube sufficiently refined
        patch(a,b,'w');
    elseif (f(i,9)==2) %cube not sufficiently refined
        patch(a,b,'r');
    end
end

end


%window setting
S='Plot der Gitterstruktur';
T= 'mit ';
for i=1:length(varargin{1})
    T=[T char(rf(varargin{1}(i))) ' = ' num2str(varargin{2}(i))];
    if i<length(varargin{1})
        T=[T ', '];
    end
end
if dim<=3
    title([S],'Fontsize',12,'FontWeight','bold');
else
    title({S; T},'Fontsize',12,'FontWeight','bold');
end
xlabel(rf(x),'Fontsize',12);
ylabel(rf(y),'Fontsize',12);
zlabel(rf(z),'Fontsize',12);
view(-37.5,30);
hold off;
```

Anhang D

Inhalt der beiliegenden CD

Auf der beiliegenden CD sind folgende Verzeichnisse enthalten:

DIPLOMARBEIT
Das Verzeichnis *Diplomarbeit* enthält eine pdf-Version dieser Arbeit sowie ein Unterverzeichnis *Abbildungen*, in dem alle in der Arbeit erscheinenden Abbildungen im eps-Format sowie im pdf-Format gespeichert sind. Die Bezeichnung der Abbildungen orientiert sich an deren Nummerierungen im Text.

BEISPIELE
Die Daten, Abbildungen und Ergebnisse zu den in Kapitel 8 untersuchten Beispielen sind im Verzeichnis *Beispiele* abgelegt. Für jedes Beispiel `i` wurde ein eigenes Unterverzeichnis mit dem Namen der im Text verwendeten Überschrift erstellt, das folgende Dateitypen beinhaltet:

`prod[i].txt`	Eingabedatei mit Portfoliodaten
`rf[i].txt`	Eingabedatei mit Risikofaktorwerten
`shifts[i].txt`	Eingabedatei mit Shifts
`approx[i]_ref_type_err_type_approx_type_tol.txt`	Ausgabedatei mit Approximation
`info[i]_ref_type_err_type_approx_type_tol.txt`	Ausgabedatei zum Programmverlauf
`surface[i]_ref_type_err_type_approx_type_tol.eps`	Plot der Portfoliofunktion
`grid[i]_ref_type_err_type_approx_type_tol.eps`	Plot des Gitters

IMPLEMENTIERUNG
Das Verzeichnis *Implementierung* enthält folgende C- und C++-Programme:

`main.cpp`	Steuerung des Algorithmus
`defs.h`	Headerdatei mit grundlegenden Definitionen
`grid.c`	Implementierung der Gitterverwaltung
`grid.h`	Headerdatei für die Gitterverwaltung

`valuation.cpp`	Implementierung der Produktbewertung und der multilinearen Interpolation
`valuation.h`	Headerdatei für die Produktbewertung und die multilineare Interpolation
`approx`	Ausführbare Datei zur Approximation der Portfoliofunktion
`post_eval.cpp`	Nachträgliche Auswertung der Portfolioapproximation
`post_eval`	Ausführbare Datei zur nachträglichen Auswertung der Portfolioapproximation

Für weitere Informationen bezüglich der einzelnen Files sei auf Kapitel 7 und Anhang B verwiesen.

VISUALISIERUNG

Die in Abschnitt 7.7 besprochenen und in Anhang C abgedruckten MATLAB-Programme zur grafischen Darstellung finden sich im Verzeichnis *Visualisierung*:

`surfaceplot.m`	Visualisierung der Portfoliofunktion
`gridplot.m`	Visualisierung des Gitters

Notation

Die folgende Übersicht enthält einige wichtige Bezeichnungen und Notationen, die im Laufe der vorliegenden Arbeit regelmäßig vorkommen. Für weiterführende Erläuterungen zu den Ausdrücken sei auf die entsprechenden Kapitel verwiesen.

Ω	Approximationsgebiet
Γ	Gitter
Q	n-dimensionaler Quader
x_i	Wert des Risikofaktors i
s_i	Shift des Risikofaktors i
x	Marktzustand
x_{act}	aktueller Marktzustand
$\hat{x}$	Knotenpunkt des Gitters Γ
$\widetilde{x}$	Testpunkt des Gitters Γ
$nodes(\Gamma)$	Menge der Knotenpunkte des Gitters Γ
$reg(\Gamma)$	Menge der regulären Knotenpunkte des Gitters Γ
$tp(\Gamma)$	Menge der Testpunkte des Gitters Γ
$ref(\Gamma)$	Verfeinerung des Gitters Γ
$v(x)$	Portfoliofunktion
v_{act}	aktueller Wert des Portfolios
$p_i(x)$	Wert des Produkts i
$v_\Gamma(x)$	Approximation der Portfoliofunktion
$\eta(x)$	Approximationsfehler
$\widetilde{\eta}(x)$	Fehlerschätzer
$\mathbb{R}$	Raum der reellen Zahlen
$\mathbb{R}^n$	Raum der n-dimensionalen reellen Zahlen
$\mathcal{W}$	Raum der multilinearen Funktionen auf Ω
$C(A)$	Raum der auf der Menge A stetigen Funktionen
$int(A)$	Inneres der Menge A
$cl(A)$	Abschluss der Menge A
$\|a\|$	Absolutwert der Zahl a

$\|x\|$	euklidische Norm des Vektors x
$\|x\|_1$	Betragssummennorm des Vektors x
$\|x\|_\infty$	Maximumsnorm des Vektors x
$H^*(A, B)$	nichtsymmetrischer Hausdorff-Abstand zwischen den Mengen A und B

Literaturverzeichnis

[1] BaFin (2000), *Grundsatz I über die Eigenmittel der Institute*, Bekanntmachung vom 20. Juli 2000, BAnz. Nr. 160 vom 25. August 2000.

[2] BaFin (2006a), *Rundschreiben 18/2005, Mindestanforderungen an das Risikomanagement*, Fassung vom 17.08.2006.

[3] BaFin (2006b), *Verordnung über die angemessene Eigenmittelausstattung von Instituten, Institutsgruppen und Finanzholding-Gruppen (Solvabilitätsverordnung)*, BGBl. I S. 2926.

[4] Basler Ausschuss für Bankenaufsicht (1996), *Änderung der Eigenkapitalvereinbarung zur Einbeziehung der Marktrisiken*, BGBl. I S. 2926.

[5] Deutsche Bundesbank (2004), *Stresstests bei deutschen Banken – Methoden und Ergebnisse*, Monatsbericht Oktober 2004.

[6] L. Grüne (1997), *An Adaptive Grid Scheme for the discrete Hamilton-Jacobi-Bellman Equation*, in: Numerische Mathematik, Volume 75, No. 3, S. 319 - 337, Springer Verlag.

[7] L. Grüne, M. Metscher, M. Ohlberger (1999), *On Numerical Algorithm and Interactive Visualization for Optimal Control Problems*, in: Computing and Visualization in Science, Volume 1, No. 4, S. 221 - 229, Springer Verlag.

[8] L. Grüne (2001), *Adaptive grid generation for evolutive Hamilton-Jacobi-Bellman equations*, in: M. Falcone, Ch. Makridakis (Hrsg.), Numerical Methods for Viscosity Solutions and Applications, World Scientific, S. 153 - 172.

[9] L. Grüne (2003), *Numerische Mathematik I*, Vorlesungsskript, Universität Bayreuth, WS 2002/2003, http://www.uni-bayreuth.de/departments/math/∼lgruene/numerik0203/.

[10] L. Grüne (2004a), *Numerik Dynamischer Systeme*, Vorlesungsskript, Universität Bayreuth, WS 2003/2004, http://www.uni-bayreuth.de/departments/math/∼lgruene/numdyn0304/.

[11] L. Grüne (2004b), *Numerische Dynamik von Kontrollsystemen*, Vorlesungsskript, Universität Bayreuth, SS 2004, http://www.uni-bayreuth.de/departments/math/∼lgruene/ndks04/.

[12] L. Grüne, W. Semmler (2004c), *Using Dynamic Programming with Adaptive Grid Scheme for Optimal Control Problems in Economics*, in: J. Bullard et al. Hrsg., Journal of Economic Dynamics and Control, Volume 28, S. 2427 - 2456, Verlag Elsevier.

[13] J. C. Hull (2005), *Options, Futures and other Derivatives*, 6. englische Auflage, Prentice Hall.

[14] B. A. Odegaard (2007), *Financial Numerical Recipes in C++*, http://finance-old.bi.no/~bernt/gcc_prog/recipes/index.html.

[15] Österreichische Nationalbank (1999), *Durchführung von Krisentests*, Leitfadenreihe zum Marktrisiko, Band 5.

[16] W. H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling (1993), *Numerical Recipes in C: The Art of Scientific Computing*, 2. englische Auflage, Cambridge University Press.

[17] S. Reitz (2006), *Stresstests*, in: A. Becker, W. Gruber und D. Wohlert (Hrsg.), Handbuch MaRisk, Mindestanforderungen an das Risikomanagement in der Bankpraxis, S. 571 - 589, Fritz Knapp Verlag.

[18] Riskmetrics Group (1999), *Risk Management – A Practical Guide*.

[19] S. M. Ross (2003), *An Elementary Introduction to Mathematical Finance*, 2. englische Auflage, Cambridge University Press.

[20] C. Schirsch et al. (2003), *Fachkonzept Stresstesting*, Internes Dokument der BayernLB.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

München, den 30. Mai 2007

.....
Kathrin Maul