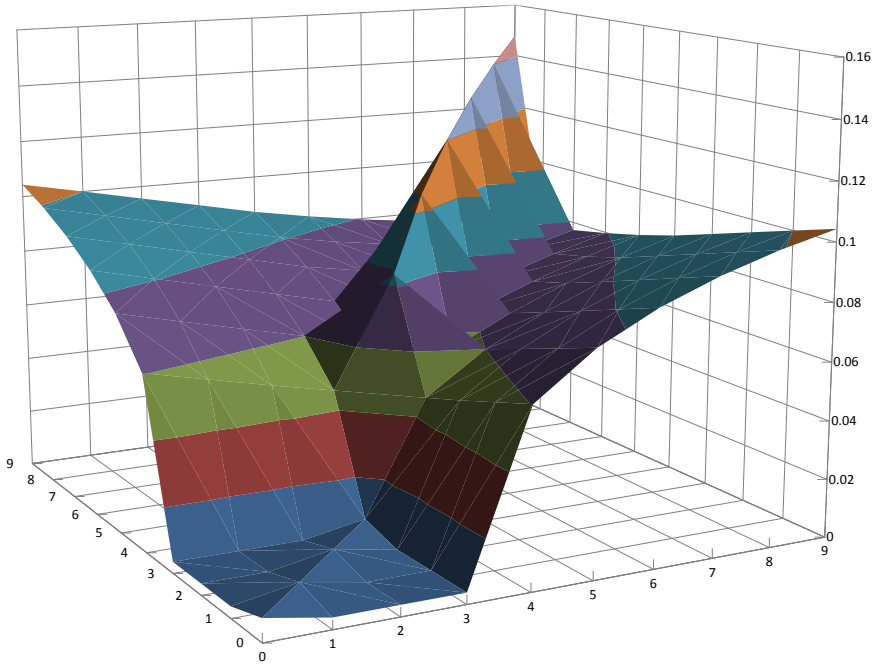
Innovative Techniken und Algorithmen im Bereich Computational-Finance und Risikomanagement

Liang, Qian TU Kaiserslautern



1. Gutachter: **Prof. Dr. Ralf Korn**
2. Gutachter: **Prof. Dr. Thomas Gerstner**

Innovative Techniken und Algorithmen im Bereich Computational-Finance und Risikomanagement

Liang, Qian

Vom Fachbereich Mathematik der Technischen Universität Kaiserslautern zur Verleihung des akademischen Grades Doktor der Naturwissenschaften (Doctor rerum naturalium, Dr. rer. nat.) genehmigte Dissertation.

1.Gutachter: Prof. Dr. Ralf Korn
2.Gutachter: Prof. Dr. Thomas Gerstner

Datum der Disputation: 05. Juni 2012

D 386

Für Meine Familie

Jinna

Du bist der Sonnenschein in meinem Leben
das einzige Wichtige auf der Welt für mich
meine ganze Liebe möchte ich dir immer geben

Ruige und Ruifei

Ihr seid das größte Geschenk das ich je bekommen hab
das Beste was das Leben mir bisher gab
Ich liebe euch

Vorwort und Danksagung

Die vorliegende Dissertation entstand im Zeitraum von August 2008 bis Juni 2012 in der Arbeitsgruppe Finanzmathematik an der TU Kaiserslautern.

Der Titel der Dissertation lautet "*Innovative Techniken und Algorithmen im Bereich Computational-Finance und Risikomanagement*". Sie besteht aus zwei aktuellen und interessanten Themen, die voneinander unabhängig sind.

Das erste Thema, "*Flexible Algorithmen zur Bewertung komplexer Optionen mit mehreren Eigenschaften mittels der funktionalen Programmiersprache Haskell*", gehört zum Projekt *Composable Derivative Contracts (ComDeCo)* des *Center for Mathematical and Computational Modelling* $(CM)^2$. Hierbei handelt es sich um ein interdisziplinäres Projekt in Kooperation mit der Arbeitsgruppe Softwaretechnik, in dem wir eine wissenschaftliche Brücke zwischen der Optionsbewertung und der funktionalen Programmierung schlugen.

Mit dem zweiten Thema, "*Monte-Carlo-Simulation des Deltas und (Cross-)Gammas von Bermuda-Swaptions im LIBOR-Marktmodell*", das auf meiner Diplomarbeit [24] basiert, wurde eine neue Problemstellung, die Gamma-Matrix einer Bermuda-Swaption genau zu berechnen, erfolgreich bearbeitet.

An dieser Stelle möchte ich mich noch bei all denen bedanken, die mich auf vielfältige Weise bei der Fertigstellung der Dissertation unterstützten:

Zu aller erst möchte ich meinem Doktorvater Herrn Prof. Dr. Ralf Korn für die Überlassung des interessanten Themas, die Fachgespräche und Anregungen sowie für die geduldige Betreuung bei der Zusammenstellung dieser Dissertation danken. Er stand mir immer mit seinem Fachwissen zur Seite und opferte viel freie Zeit für unsere wissenschaftlichen Veröffentlichungen.

Für die bereitwillige Übernahme des Zweitgutachtens bedanke ich mich sehr bei Herrn Prof. Dr. Thomas Gerstner von der Johann Wolfgang Goethe-Universität, Frankfurt am Main.

Ein besonders herzlicher Dank gilt meinen Eltern, Schwiegereltern und meiner Frau, die mich sowohl während meines Studiums als auch während meiner Promotion in Deutschland sehr unterstützt und alle auf ihre Weise zum Gelingen dieser Dissertation beigetragen haben.

Danke sage ich auch Herrn Prof. Dr. Arnd Poetzsch-Heffter und Herrn Jean-Marie Gaillourdet für die Zusammenarbeit und Kooperation bei dem ersten Teil der Dissertation, vor allem im Bezug auf die funktionale Programmierung.

Herrn Prof. Dr. Matthias Leclerc von der KfW Bankengruppe und Herrn Dr. Ingo Schneider von der DekaBank, Frankfurt am Main, danke ich für ihre stets Unterstützung meiner Promotion und ihre Interesse an meiner Forschung. Herr Dr. Ingo Schneider bot mir mehrmals neue wissenschaftliche Artikel und Ideen, die hilfreich für den zweiten Teil der Dissertation waren.

Ein ganz persönlicher Dank geht an meinen Bürokollegen Herrn Dr. Michael Busch für seine stete Hilfsbereitschaft, die angenehme Atmosphäre in unserem Büro und die daraus erwachsene Freundschaft, auch zur Familie Busch.

Neben Prof. Dr. Ralf Korn und Dr. Michael Busch bedanke ich mich ebenso herzlich bei den anderen aktuellen und ehemaligen Mitarbeitern der Arbeitsgruppe Finanzmathematik, Christoph Belak, Dr. Peter Diesinger, Alona Futorna, Dr. Melanie Hollstein, Alexandra Kochendörfer, Anton Kostiuk, Dr. Henning Marxen, Yaroslav Melnyk, Dr. Stefanie Müller, Prof. Dr. Jörn Sass, Prof. Dr. Frank Thomas Seifried, Martin Smaga, Songyin Tang, Dr. Nicole Tschauder und unsere Sekretärin Frau Cornelia Türk, für ihre Unterstützung und Hilfestellung sowie die stets nette und kollegiale Atmosphäre.

Ich kann an dieser Stelle nicht alle Personen aufzählen, die mir während der Dissertation durch konstruktive Kritik, motivierende Gespräche und Ablenkung geholfen haben. Dazu zählt insbesondere auch mein Freundeskreis. All diesen Personen sei nochmals ganz herzlich gedankt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Teil I.

Flexible Algorithmen zur Bewertung komplexer Optionen mit mehren Eigenschaften mittels der funktionalen Programmiersprache Haskell

1. Einführung in Teil I

Banken, Versicherungen und andere Finanzdienstleister verwenden heutzutage immer neuere und komlexere Finanzderivate. Die Unternehmen setzen sie vor allem für die Zwecke Absicherung, Spekulation und Arbitrage ein. Entsprechend entstehen immer mehr Algorithmen im Bereich Finanzmathematik, um die Finanzderivate so präzise und effizient wie möglich zu bewerten. Allerdings beschränken sich die meisten solcher Algorithmen auf einen Typ oder eine Klasse von Finanzderivaten. Will man ein neuartiges Finanzderivat bewerten, wird oft ein neuer, individueller Algorithmus entworfen. Die Herausforderung unseres Projekts besteht darin, einen innovativen Algorithmus zu entwickeln, damit man so viele Finanzderivate wie möglich bewerten kann, idealerweise auch Finanzderivate von unbekanntem Typ. Ein wichtiger Aspekt des Projekts ist die funktionale Denkweise. D. h. wir formulieren den Algorithmus mithilfe einer funktionalen Programmiersprache. In unserem Projekt wird `Haskell` verwendet. Für eine kurze Vorstellung von `Haskell` und der funktionalen Programmierung formulieren wir ein spezielles Kapitel im Anhang A.

Das erste Thema der Dissertation ist Teil des Projekts *Composable Derivative Contracts (ComDeCo)* des *Center for Mathematical and Computational Modelling* $(CM)^2$. Unser Kooperationspartner ist die Arbeitsgruppe Softwaretechnik an der Technischen Universität Kaiserslautern. Im Rahmen des Projekts *Composable-Derivative-Contracts* wird ein neuer generischer Ansatz zur Modellierung und Bewertung von Finanzderivaten entwickelt. Mittels der Bausteine eines funktionalen Frameworks lässt sich dabei eine große Klasse von Finanzderivaten beschreiben. Die zentrale Aufgabe besteht darin, neuartige mathematische Algorithmen im Rahmen des funktionalen Frameworks zu entwickeln, mit denen sich eine enorme Anzahl von Finanzderivaten bewerten lässt.

Die Grundidee und das Grundkonzept unseres Projekts stammen aus den Pionierarbeiten von Peyton-Jones, Eber und Seward, deren Ansatz einen vernünftigen Weg zur Beschreibung und Bewertung von finanziellen Kontrakten bietet. Ihre Artikel [30] und [31] geben einen Überblick über diesen Ansatz und seine Vorteile, zusammen mit einer einfachen Umsetzung des grundlegenden Konzepts, das in der funktionalen Programmiersprache `Haskell` präsentiert ist. Im Kapitel 3 der Dissertation werden die Beiträge von Peyton-Jones, Eber und Seward in Bezug auf unser Projekt diskutiert.

Unser Projekt besteht aus zwei Teilprojekten: Das eindimensionale Projekt und das multidimensionale Projekt, die wir jeweils im Kapitel 4 und Kapitel 5 vorstellen werden. Im eindimensionalen Projekt entwerfen wir ein Konzept zur Konstruktion von Single-Asset-Optionen, in dem es eine Reihe von grundle-

genden Konstruktoren gibt, mit denen man verschiedene Single-Asset-Optionen kombinieren kann. Im Rahmen des Konzepts entwickeln wir einen allgmeinen Algorithmus, durch den die aus den Konstruktoren kombinierten Single-Asset-Optionen bewertet werden können. So auch im multidimensionalen Projekt.

Der mathematische Aspekt unseres Projekts besteht in der Entwicklung neuer Konzepte zur Preisberechnung sowohl für Single-Asset-Optionen als auch für Multi-Asset-Optionen. Diese Konzepte basieren unter anderem auf dem Binomialmodell, welches im Bereich der Optionsbewertung in den letzten Jahren eine weite Verbreitung fand. Der Algorithmus zur Bewertung von Single-Asset-Optionen ist eine Kombination von mehreren sorgfältig ausgewählten numerischen Methoden, die auf dem eindimensionalen Binomialmodell basieren. Diese Kombination ist nicht trivial, sondern entwikelt sich nach bestimmten Regeln und ist eng mit den grundlegenden Konstruktoren verknüpft. Der zur Bewertung von Multi-Asset-Optionen vorgeschlagene Algorithmus entspricht der von Korn und Müller [20] entwickelten multidimensionalen Binomialmethode. Die theoretischen Grundlagen der Finanzderivate, des Binomialmodells und alle auf unser Projekt anzuwendende numerische Methoden diskutieren wir in Kapitel 2.

2. Finanzmathematische Grundlagen

Die Finanzmathematik ist eine Disziplin der angewandten Mathematik, die sich mit Themen aus dem für Finanzdienstleister, wie etwa Banken oder Versicherungen, relevanten Bereich beschäftigt. Im engeren Sinne wird mit Finanzmathematik meist die bekannteste Unterdisziplin, die Bewertungstheorie, bezeichnet, d. h. die Ermittlung theoretischer fairer Preis von Finanzderivaten. Ein wichtiges Axiom der Bewertungstheorie ist das der Arbitragefreiheit, also der Abwesenheit jeder Möglichkeit zur Arbitrage. In einem arbitragefreien, vollständigen Markt gibt es ein eindeutig bestimmtes, äquivalentes Martingalmaß und daher ist der Preis jedes Finanzderivats eindeutig festgelegt. Das bekannteste Ergebnis der Bewertungstheorie ist das Black-Scholes-Modell, das von Black und Scholes [3] im Jahre 1973 entwikelt wurde und geschlossene Bewertungsformeln besitzt. Das Black-Scholes-Modell ist ein kontinuierliches Modell, in dem der Aktienkursverlauf mittels einer geometrischen Brownschen Bewegung modelliert wird. Es entwickelte sich sehr schnell zum Standardmodell für die analytische Bewertung von einfachen Finanzderivaten und gilt als ein Meilenstein der Finanzmathematik.

In unserer Arbeit beschäftigen wir uns mit dem Binomialmodell, das eine diskrete Approximation des Black-Scholes-Modells darstellt. Dabei wird der kontinuierliche Prozess der geometrischen Brownschen Bewegung ersetzt durch einen diskreten stochastischen Prozess, den Binomialprozess. Aufgrund der Intuitivität, der Einfachheit der Implementierung und der hohen Flexibilität des Binomialmodells sind die darauf basierenden numerische Methoden zur Bewertung von Finanzderivaten in der Praxis weit verbreitet. Darüber hinaus erweist es sich als äußerst flexibel bzgl. der zu bewertenden Finanzderivate.

In diesem Kapitel werden wir zunächst die Finanzderivate kurz einführen. Dann erklären wir das allgemeine Prinzip des Binomialmodells. Dazu zählen die Konstruktion von Binomialbäumen, der Ablauf der darauf basierenden Methode mit ihrer Implementierung sowie die Beziehungen der benötigten Parameter untereinander. Im weiteren Verlauf stellen wir ein eindimensionales Binomialmodell, das CRR-Modell, und seine Anwendungen auf verschiedene Arten von Finanzderivaten vor. Schließlich widmen wir uns einem innovationen multidimensionalen Binomialmodell, welches von Korn und Müller [20] im Jahr 2009 entwickelt wurde. Die Ausarbeitung dieses Kapitels orientiert sich hauptsächlich an den Bücher von Hoek und Elliott [13], Hull [14] sowie R. Korn und E. Korn [17].

2.1. Einführung der Finanzderivate

In den letzten dreißig Jahren haben Finanzderivate in der Finanzwelt immer größere Bedeutung erlangt. Futures und Optionen werden heute intensiv an den Börsen gehandelt. Außerhalb der Börsen werden viele verschiedene Arten von Forward-Kontrakten, Swaps, Optionen und anderen Finanzderivaten regelmäßig durch Finanzinstitute, Fondsmanager und Finanzmanager auf dem Over-the-Counter-Markt (OTC-Markt) gehandelt. Finanzderivate bieten ein breites Anwendungsspektrum in der Finanzwelt und werden vor allem für die folgenden Zwecke eingesetzt:

- **Absicherung** (Hedging): Absicherung bestehender oder geplanter Positionen gegenüber Markt- und Ausfallrisiken.
- **Spekulation**: Nutzung von Preisunterschieden zur Erzielung eines Gewinns aufgrund von Markterwartungen. Durch den Einsatz von Finanzderivaten kann ein hohes Volumen mit relativ geringem Kapitaleinsatz bewegt werden. Dies nennt man Hebelwirkung.
- **Arbitrage**: Nutzung von Preisdifferenzen zwischen verschiedenen Märkten und Produkten zur gleichen Zeit, insbesondere zwischen Kassa- und Derivatemarkt.

Was versteht man genau unter Finanzderivaten? Allgemein ist ein Finanzderivat ein finanzieller Kontrakt, dessen Wert an seiner Fälligkeit oder seinem Verfallsdatum durch den Wert (oder die Werte) seines Basiswertes oder Underlyings (z. B. Aktien, Aktienindex, Zins, Kredit, Währung, Energiepreis, Wetter u. s. w.) zur Fälligkeit oder bis zur Fälligkeit eindeutig bestimmt wird. Es gibt grob gesagt drei Klassen von Finanzderivaten:

- **Option**: Eine Option ist ein finanzieller Kontrakt, der dem Käufer oder Inhaber der Option bis zur ihre Fälligkeit das Recht, aber nicht die Verpflichtung einräumt, eine bestimmte Menge eines bestimmten Basiswertes zu einem im voraus festgesetzten Ausübungspreis zu kaufen oder verkaufen.
- **Forward-Kontrakt** und **Future**: Ein Forward-Kontrakt ist ein finanzieller Kontrakt mit der Vereinbarung zwischen zwei Geschäftspartnern, einen Basiswert untereinander bei Fälligkeit zu einem bestimmten Ausübungspreis zu kaufen oder verkaufen. Der Unterschied zur Option ist, dass der Basiswert geliefert und bezahlt werden muss. Ein Future ist im wesentlichen ein an den Börsen standardisierter Forward-Kontrakt. Der Wert eines Futures wird täglich berechnet und von den Vertragsparteien ausgeglichen.
- **Swap**: Bei dem finanziellen Kontrakt Swap tauschen zwei Geschäftspartnern Zahlungsströme zu mehreren vorzeitig festgelegten Zeitpunkten, die durch eine vorgegebene Formel bestimmt werden.

In unserer Arbeit werden wir uns im wesentlichen auf Optionen konzentrieren. Wir unterteilen die Optionen in drei Schichten: Die Plain-Vanilla-Optionen, die pfadabhängigen Optionen und die Multi-Asset-Optionen (Korrelationsoptionen) und werden diese in ihren Einzelheiten präsentieren. Pfadabhängige und Multi-Asset-Optionen werden allgemein unter dem Begriff der exotischen Optionen zusammengefasst.

2.1.1. Plain-Vanilla-Optionen

Als Plain-Vanilla-Option bezeichnet man die von ihrer Struktur her einfachsten Optionen mit den Varianten europäische bzw. amerikanische Call- und Put-Option, die eindeutig definierte, standardisierte Eigenschaften besitzen und akiv im Derivatmarkt gehandelt werden. Im Folgenden sind die jeweiligen Definitionen und Notationen aufgeführt:

- Eine europäische Call- bzw. Put-Option ist ein Finanzderivat mit der Vereinbarung, dass der Käufer der Option zur Fälligkeit T das Recht, aber nicht die Pflicht hat, einen Basiswert zu einem bestimmten Ausübungspreis K vom Verkäufer der Option zu kaufen (Call) bzw. an diesen zu verkaufen (Put).
- Bei einer amerikanischen Call- bzw. Put-Option hat der Käufer der Option jederzeit bis zur Fälligkeit T das Recht, nicht aber die Pflicht, einen Basiswert zu einem bestimmten Ausübungspreis K vom Verkäufer der Option zu kaufen (Call) bzw. an diesen zu verkaufen (Put).

In der Finanzmathematik sind meist die Auszahlungsfunktionen der Optionen von Interesse. So sind die Auszahlungsfunktionen aus Sicht des Inhabers einer Plain-Vanilla-Option wie folgt gegeben:

- Auszahlungsfunktion einer europäische Call-Option mit Fälligkeit T:
$$(S(T) - K)^+ = \max(S(T) - K, 0).$$
- Auszahlungsfunktion einer europäische Put-Option mit Fälligkeit T:
$$(K - S(T))^+.$$
- Auszahlungsfunktion einer amerikanischen Call-Option zum optimalen Ausübungszeitpunkt $t^* \leq T$:
$$(S(t^*) - K)^+.$$
- Auszahlungsfunktion einer amerikanischen Put-Option zum optimalen Ausübungszeitpunkt $t^* \leq T$:
$$(K - S(t^*))^+.$$

Dabei bezeichnet $S(t)$ den Kurs des zugrunde liegenden Basiswertes zur Zeit t.

2.1.2. Exotische Optionen

Ein besonder Aspekt des Derivatmarktes sind die nicht-standardisierten Produkte, die im Rahmen des Financial-Engineerings geschaffen wurden. Zu den Produkten gehören die exotischen Optionen. Sie sind Finanzderivate, die von Plain-Vanilla-Optionen abgeleitet sind und im Allgemeinen kompliziertere Auszahlungsfunktionen als die vergleichbaren Plain-Vanilla-Optionen besitzen. Wir können exotische Optionen in die folgenden Klassen einteilen: Pfadunabhängige Optionen und pfadabhängige Optionen oder Single-Asset-Optionen und Multi-Asset-Optionen. Im Gegensatz zu den pfadunabhängigen Optionen hängt die Auszahlung einer pfadabhängigen Option nicht nur vom Kurs des Basiswertes zur Fälligkeit, sondern vom gesamten Kursverlauf des Basiswertes ab. Und im Gegensatz zu den Single-Asset-Optionen ist eine Multi-Asset-Option eine Option, die nicht nur auf einem Basiswert, sondern auf mehreren Basiswerten basiert. Von der Vielzahl der exotischen Optionen stellen wir in diesem Unterabschnitt nur einige wichtige Vertreter vor, da der Phantasie zur Konstruktion von exotischen Optionen keine Grenzen gesetzt sind. In dem Buch von Zhang [35] ist eine größe Auswahl exotischer Optionen aufgelistet.

Pfadunabhängige Single-Asset-Optionen

Digitaloption Eine Digitaloption wird wertlos, wenn der Kurs des Basiswertes zur Fälligkeit T eine festgelegte Schranke H über- oder unterschreitet. Die Auszahlungsfunktionen der Digitaloptionen zur Fälligkeit T sind durch

$$\begin{aligned} \text{Digital-Call:} \quad & \mathbf{1}\{S(T) > H\}, \\ \text{Digital-Put:} \quad & \mathbf{1}\{S(T) < H\} \end{aligned}$$

gegeben, wobei

$$\mathbf{1}\{A\} = \begin{cases} 1 & : \quad A = wahr \\ 0 & : \quad A = falsch \end{cases}$$

die Indikatorfunktion ist.

Gap-Option Gap-Optionen sind eng verwandt mit den Digitaloptionen. Ihre Auszahlungsfunktionen zur Fälligkeit T sind durch

$$\begin{aligned} \text{Gap-Call:} \quad & (S(T) - K) \cdot \mathbf{1}\{S(T) \geq H\}, \\ \text{Gap-Put:} \quad & (K - S(T)) \cdot \mathbf{1}\{H \geq S(T)\} \end{aligned}$$

gegeben, wobei im Allgemeinen $K \neq H$ gilt. Für $K = H$ sind sie mit den europäischen Optionen identisch.

Zusammengesetzte Option Mit dem Kauf einer zusammengesetzten Option bekommt man das Recht, zur Fälligkeit T eine andere Option mit der Fälligkeit $T_1 \geq T$ zum Ausübungspreis K zu kaufen bzw. zu verkaufen. Es gibt vier Fälle, deren Auszahlungsfunktionen zur Zeit T wie folgt dargestellt sind:

$$\begin{aligned}
\text{Call-auf-Call:} &\quad \left(V_T\left[(S(T_1)-K_1)^+\right]-K\right)^+,\\
\text{Call-auf-Put:} &\quad \left(V_T\left[(K_1-S(T_1))^+\right]-K\right)^+,\\
\text{Put-auf-Call:} &\quad \left(K-V_T\left[(S(T_1)-K_1)^+\right]\right)^+,\\
\text{Put-auf-Put:} &\quad \left(K-V_T\left[(K_1-S(T_1))^+\right]\right)^+,
\end{aligned}$$

wobei die auftretenden europäischen Call- und Put-Optionen jeweils einen Ausübungspreis K_1 haben und $V_T[f]$ den diskontierten Wert von Auszahlung f zur Zeit T bezeichnet.

Wahl-Option Bei der Wahl-Option kann man zur Fälligkeit T entscheiden, entweder einen europäischen Call mit der Fälligkeit $T_1 \geq T$ und dem Ausübungspreis K_1 oder einen europäischen Put mit der Fälligkeit $T_2 \geq T$ und dem Ausübungspreis K_2 zu erhalten. Ihre Auszahlungsfunktion zur Zeit T lautet:

$$\max\left(V_T\left[(S(T_1)-K_1)^+\right], V_T\left[(K_2-S(T_2))^+\right]\right).$$

Pfadabhängige Single-Asset-Optionen

Bermuda-Option Unter einer Bermuda-Option versteht man eine modifizierte amerikanische Option, dabei kann sich die vorzeitige Ausübung auf bestimmte Termine $\{t_1, t_2, \ldots, t_n\} \subseteq [0,T]$ beschränken. Ihre Auszahlungsfunktionen zum Ausübungszeitpunkt t sind:

$$\begin{aligned}
\text{Bermuda-Call:} &\quad \left(\max_{t\in\{t_1,\ldots,t_n\}} S(t)-K\right)^+,\\
\text{Bermuda-Put:} &\quad \left(K-\min_{t\in\{t_1,\ldots,t_n\}} S(t)\right)^+.
\end{aligned}$$

Barriere-Option Barriere-Optionen werden unterschieden in Knock-Out-Optionen und Knock-In-Optionen. Das Optionsrecht einer Knock-Out-Option erlischt, sobald der Kurs des Basiswertes eine bestimmte Barriere H erreicht. Im Gegensatz dazu löst eine Knock-In-Option erst dann aus, wenn der Kurs des Basiswertes die Barriere H trifft. Knock-Out-Optionen werden ferner in Down-and-Out-, Up-and-Out- und Double-Knock-Out-Optionen unterteilt. In gleicher

Weise können die Knock-In-Optionen in Down-and-In-, Up-and-In- und Double-Knock-In-Optionen untergliedert werden. Ihre Auszahlungsfunktionen zur Fälligkeit T im europäischen Fall sind beispielsweise:

$$\begin{aligned}
\text{Down-and-Out-Call:} \quad & (S(T)-K)^+ \cdot \mathbf{1}\left\{\min_{t\in[0,T]} S(t) > H\right\},\\
\text{Up-and-Out-Call:} \quad & (S(T)-K)^+ \cdot \mathbf{1}\left\{\max_{t\in[0,T]} S(t) < H\right\},\\
\text{Down-and-In-Call:} \quad & (S(T)-K)^+ \cdot \mathbf{1}\left\{\min_{t\in[0,T]} S(t) \leq H\right\},\\
\text{Up-and-In-Call:} \quad & (S(T)-K)^+ \cdot \mathbf{1}\left\{\max_{t\in[0,T]} S(t) \geq H\right\},\\
\text{Down-and-Out-Put:} \quad & (K-S(T))^+ \cdot \mathbf{1}\left\{\min_{t\in[0,T]} S(t) > H\right\},\\
\text{Up-and-Out-Put:} \quad & (K-S(T))^+ \cdot \mathbf{1}\left\{\max_{t\in[0,T]} S(t) < H\right\},\\
\text{Down-and-In-Put:} \quad & (K-S(T))^+ \cdot \mathbf{1}\left\{\min_{t\in[0,T]} S(t) \leq H\right\},\\
\text{Up-and-In-Put:} \quad & (K-S(T))^+ \cdot \mathbf{1}\left\{\max_{t\in[0,T]} S(t) \geq H\right\}.
\end{aligned}$$

Lookback-Option Bei Lookback-Optionen hängen die Auszahlungsfunktionen vom Minimum und Maximum des Kurses des Basiswertes während der gesamten Optionslaufzeit mit Fälligkeit T ab. Es gibt im Allgemeinen vier Typen von Lookback-Optionen. Ihre Auszahlungen zur Zeit T im europäischen Fall sind jeweils:

$$\begin{aligned}
\text{Fixed-Lookback-Call:} \quad & \left(\max_{t\in[0,T]} S(t) - K\right)^+,\\
\text{Fixed-Lookback-Put:} \quad & \left(K - \min_{t\in[0,T]} S(t)\right)^+,\\
\text{Floating-Lookback-Call:} \quad & S(T) - \min_{t\in[0,T]} S(t),\\
\text{Floating-Lookback-Put:} \quad & \max_{t\in[0,T]} S(t) - S(T).
\end{aligned}$$

Asiatische Option Asiatische Optionen sind Optionen, bei denen die Auszahlung vom durchschnittlichen Kurs des Basiswertes während der Optionslaufzeit abhängt. Ihre Auszahlungen zur Fälligkeit T im europäischen Fall betragen:

$$\text{Asiatische Call-Option:} \quad \left(\tfrac{1}{T}\int_0^T S(t)dt - K\right)^+,$$
$$\text{Asiatische Put-Option:} \quad \left(K - \tfrac{1}{T}\int_0^T S(t)dt\right)^+.$$

Pfadunabhängige Multi-Asset-Optionen

Basket-Option Bei der Basket-Option handelt es sich angeblich um die populärste Multi-Asset-Option. Sie ist eine Option, deren Auszahlung vom Wert eines Portfolios mit M Basiswerten abhängig ist. Ihre Auszahlungsfunktionen zur Fälligkeit T sind beispielsweise:

$$\text{Basket-Call:} \quad \left(\sum_{i=1}^{M} \omega_i S_i(T) - K\right)^+,$$
$$\text{Basket-Put:} \quad \left(K - \sum_{i=1}^{M} \omega_i S_i(T)\right)^+,$$

wobei

$$\sum_{i=1}^{M} \omega_i = 1$$

gilt.

Produktoption Die Auszahlungsfunktionen einer Produktoption mit M Basiswerten zur Fälligkeit T sind:

$$\text{Produkt-Call:} \quad \left(\left(\prod_{i=1}^{M} S_i(T)\right)^{\frac{1}{M}} - K\right)^+,$$
$$\text{Produkt-Put:} \quad \left(K - \left(\prod_{i=1}^{M} S_i(T)\right)^{\frac{1}{M}}\right)^+.$$

Pfadabhängige Multi-Asset-Optionen

Rainbow-Barriere-Option Die Rainbow-Barriere-Option ist eine normale Multi-Asset-Option mit zusätzlichen Ausübungskriterien, deren Wert davon abhängt, ob der Kurs von einem oder einigen Basiswerten vorher festgelegte Schranken in der Zeit bis zur Fälligkeit T über- bzw. unterschreitet. So kann die Option durchaus auch wertlos werden. Beispielsweise ist die Auszahlungsfunktion einer Rainbow-Down-and-Out zur Zeit T gegeben durch:

$$\max\{S_1(T), \ldots, S_M(T)\} \cdot \mathbf{1}\left\{\bigwedge_{i \in I \subseteq \{1,\ldots,M\}} \left(\min_{t\in[0,T]} S_i(t) > H_i\right)\right\}.$$

2.2. Eindimensionales Binomialmodell

In der Finanzwelt sind Optionen und andere Finanzderivate in den letzten Jahren zu einem unentbehrlichen Werkzeug zur Kontrolle und Absicherung von Risiken geworden. Das herausfordernde Problem ist die Ermittlung der fairen Preise von Optionen, die auf modernen mathemathischen Methoden basiert. Zur Bewertung einfacher Optionen trägt die Black-Scholes-Formel bei, die aus dem Black-Scholes-Modell stammt. Für komplexere Optionen existieren jedoch keine geschlossenen Bewertungsformeln mehr und ihre fairen Preise müssen numerisch gelöst werden. Die Problemstellung, die mathematische Modellierung und die numerische Simulation von Optionen, auf denen unser eindimensionales Projekt basiert, werden in diesem Abschnitt ausführlich behandelt. Dabei beschränken wir uns auf ein zeitdiskretes Modell, das Binomialmodell. Die entsprechende numerische Simulation ist der sogenannte Binomialbaum und die korrespondierende numerische Methode heißt Binomialmethode.

Der Binomialbaum bzw. die Binomialmethode ist ein nützliches und weit verbreitetes Verfahren, um Optionen und andere Finanzderivate zu bewerten. Darunter versteht man eine Darstellung, die verschiedenen Pfade aufzeigt, in denen der Kurs des Basiswertes einem Random Walk folgt. In jedem Simulationszeitschritt gibt es eine gewisse Wahrscheinlichkeit dafür, dass sich der Kurs um einen bestimmten Prozentsatz aufwärts bewegt, und eine gewisse Wahrscheinlichkeit dafür, dass er sich um einen bestimmten Prozentsatz abwärts bewegt. Die Approximationsmethode mittels Binomialbäumen lässt sich durch den zentralen Grenzwertesatz motivieren. Genauer: Für die Grenzbetrachtung unendlich kleiner Zeitschritte führt dieses Modell zur Annahme der Lognormalverteilung der Kurse des Basiswertes, welche dem Black-Scholes-Modell zugrunde liegt.

Bevor wir den Binomialbaum explizit vorstellen, fassen wir die Voraussetzungen und Einflussgrößen für die Bewertung einer Option zusammen:

- Der Finanzmarkt sei arbitragefrei und vollständig, habe das äquivalenten Martingalmaß (risikoneutralen Maß) P und den kontinuierlichen, jährlichen, risikolosen Zinssatz r.
- Der Basiswert $S(t)$ mit dem Anfangskurs $S(0)$.
- Die Volatilität (Das Schwankungsmaß) σ vom Kurs des Basiswertes.
- Die stetige Dividendenrendite g des Basiswertes.
- Die Zeitperiode Δt und die Simulationszeitschritte $\{t_0 = 0, t_1, \ldots, t_N = T\}$ mit $t_i = i\Delta t$ für $i = 0, \ldots, N$.

Der Binomialbaum beinhaltet die Unterteilung der Laufzeit T einer Option in eine große Anzahl N von kleinen Zeitperioden der Länge Δt. Es wird angenommen, dass sich der Kurs des Basiswertes in jeder Zeitperiode $[t_i, t_{i+1}]$ für

$i = 0, \ldots, N-1$ von seinem Anfangskurs $S(t_i)$ zu einem der beiden neuen Kurse $uS(t_i)$ oder $dS(t_i)$ entwickelt. Dieser Ansatz wird in Abbildung 2.1 gezeigt. Im All-

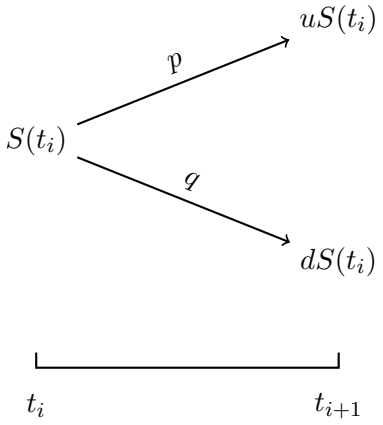


Abbildung 2.1.: Kursbewegungen des Basiswertes über einer Zeitperiode Δt nach dem Binomialmodell

gemeinen gilt $u > 1$ und $0 < d < 1$. Die Bewegung von $S(t_i)$ nach $uS(t_i)$ ist daher eine Aufwärtsbewegung und die Bewegung von $S(t_i)$ nach $dS(t_i)$ eine Abwärtsbewegung. Die Wahrscheinlichkeit der Aufwärtsbewegung wird mit p bezeichnet. Die Wahrscheinlichkeit der Abwärtsbewegung ist q mit $q = 1-p$. Die Parameter p, q, u und d müssen korrekte Werte für den Mittelwert und die Varianz der Kursveränderungen des Basiswertes während einer Zeitperiode Δt erzeugen. Da wir in einem risikoneutralen Markt arbeiten, entspricht die erwartete Rendite eines Basiswertes dem risikolosen Zinssatz r. Wenn der Basiswert noch eine stetige Dividendenrendite g liefert, muss seine erwartete Rendite dann $r - g$ betragen. Das bedeutet, dass der Erwartungswert vom Kurs des Basiswertes am Ende einer Zeitperiode $[t_i, t_{i+1}]$ für $i = 0, \ldots, N-1$ durch $\mathbb{E}\left(S(t_i)e^{(r-g)\Delta t}\right)$ gegeben ist, wobei $S(t_i)$ der Kurs zum Beginn der Zeitperiode ist. Das Prinzip der risikoneutralen Bewertung ist das Schlüsselelement bei der Verwendung von Binomialbäumen. Daraus können wir die folgenden Beziehungen der Parameter erhalten:

$$p = \frac{e^{(r-g)\Delta t} - d}{u - d}$$

und

$$q = \frac{u - e^{(r-g)\Delta t}}{u - d}.$$

Außerdem gilt wegen der Arbitragefreiheit:

$$d < e^{(r-g)\Delta t} < u.$$

Durch den vorgenannten Prozess kann sich der gesamte Binomialbaum vom Kurs eines Basiswertes vom Zeitpunkt 0 bis T entwickeln. Wir nennen diesen

Binomialbaum den S-Baum, welcher wie in Abbildung 2.2 aussieht. Dabei be-

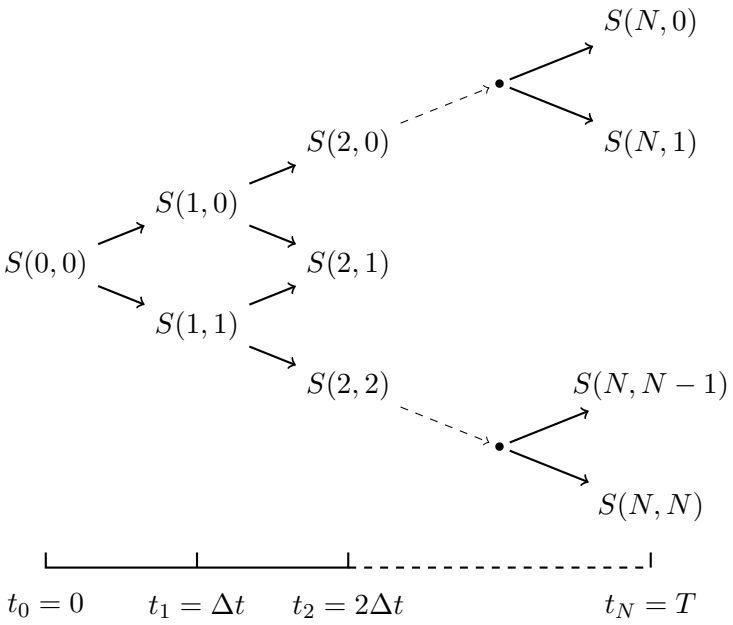


Abbildung 2.2.: Binomialbaum vom Kurs des Basiswertes (S-Baum)

zeichnet der Knoten $S(i,j)$ für $i = 0,\ldots,N$ und $j = 0,\ldots,i$ den Kurs des Basiswertes nach i Zeitperioden mit $i-j$ Aufwärtsbewegungen und j Abwärtsbewegungen. Hierdurch können alle Knoten des S-Baumes durch den Algorithmus 1 berechnet werden. Dieser Binomialbaum ist offenbar rekombinierbar, was die numerische Handhabung wesentlich vereinfacht.

Algorithmus 1: S-Baum einer Option

Eingabe : $S(0)$, u, d, N
Ausgabe : Alle Knoten des S-Baumes

```
for i = 0 to N do
    for j = 0 to i do
        S(i,j) ⟵ S(0)u^{i-j}d^j
    end
end
```

Bei der Bewertung einer Option interessieren wir uns nicht nur für den Kurs des Basiswertes, sondern auch für die Auszahlungen der Option, welche von der Auszahlungsfunktion und den Auszahlungsterminen abhängen. Der entsprechende Auszahlungsbaum wird C-Baum (siehe Abbildung 2.3) genannt. Dabei bezeichnet $C(i,j)$ für $i = 0,\ldots,N$ und $j = 0,\ldots,i$ die Auszahlung bzgl. des Kurses $S(i,j)$. Die Werte der Knoten des C-Baumes werden durch den Algorithmus 2 berechnet, wobei f die Auszahlungsfunktion, $\perp$ den Knoten ohne Auszahlung und

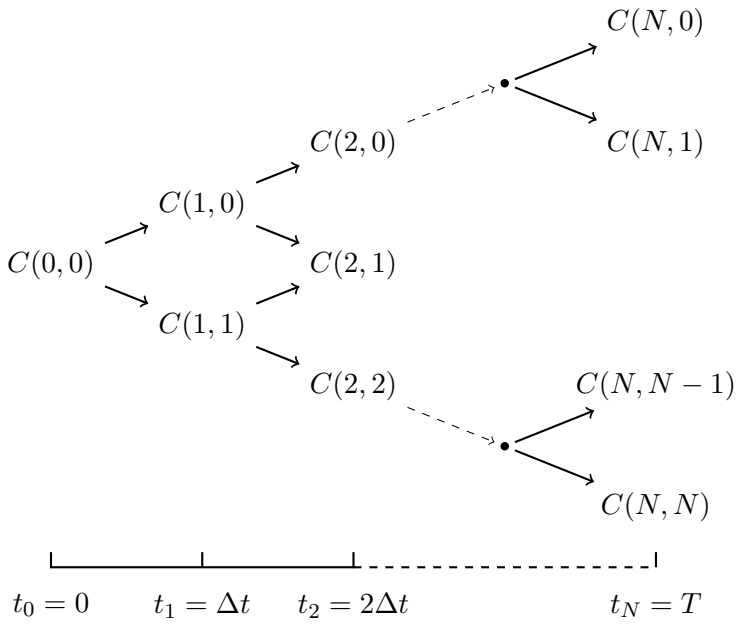


Abbildung 2.3.: Binomialbaum der Auszahlungen (C-Baum)

T_A die Menge aller Auszahlungstermine bezeichnet. Wir bemerken, dass für eine europäische Option $T_A = \{t_N\}$, für eine amerikanische Option $T_A = \{t_0, \ldots, t_N\}$ und für eine bermudische Option $T_A \subsetneq \{t_0, \ldots, t_N\}$ gilt.

Algorithmus 2: Vom S-Baum zum C-Baum einer Option

Eingabe : S-Baum, N, f, T_A
Ausgabe : Alle Knoten des C-Baumes

```
for i = 0 to N do
    if t_i ∈ T_A then
        for j = 0 to i do
            C(i,j) ⟵ f(S(i,j))
        end
    else
        for j = 0 to i do
            C(i,j) ⟵ ⊥
        end
    end
end
```

Mithilfe des S-Baumes und des C-Baumes einer Option können wir ihren fairen Preis vom Zeitpunkt T bis zum Zeitpunkt 0 rückwärts rekursiv berechnen. Dabei entsteht wieder ein Binomialbaum von den inneren Werten der Option bezogen auf alle Zeitschritte. Diesen Binomialbaum nennen wir den V-Baum (siehe Abbildung 2.4), wobei $V(0,0)$ dem Preis der Option zum Zeitpunkt 0 ent-

spricht. Im Gegensatz zum S-Baum und C-Baum wird der V-Baum rückwärtig berechnet. Die Berechnungsmethoden für den V-Baum sind wegen der vielfältigen Arten von Optionen unterschiedlich. In den folgenden Unterabschnitten werden wir die Berechnungsmethoden für ausgewählte fundamentale Optionstypen detailliert vorstellen.

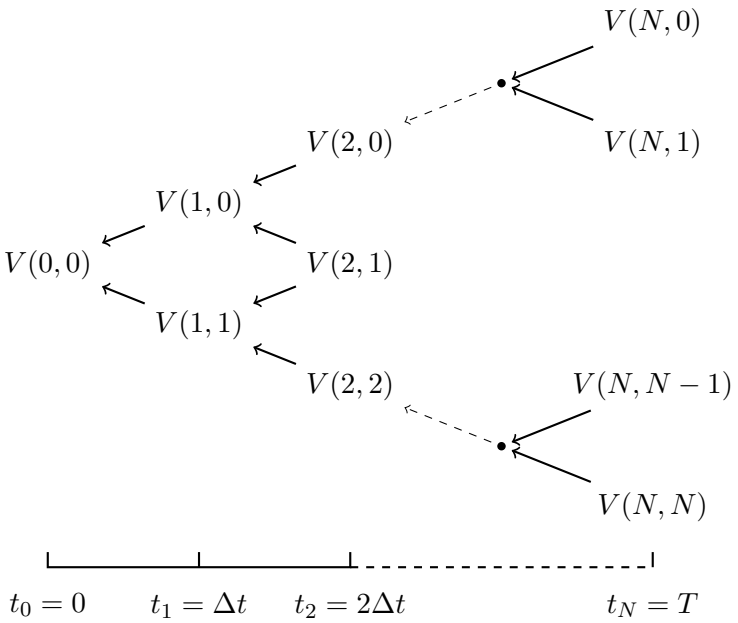


Abbildung 2.4.: Binomialbaum der Werte (V-Baum)

Zusammenfassend besteht die Binomialmethode zur Bewertung einer Option aus zwei Phasen, dem Vorwärts- und der Rückwärtsphase, und drei Binomialbäumen, dem S-Baum, dem C-Baum und dem V-Baum. In der Vorwärtsphase berechnen wir den S-Baum und den C-Baum. Und in der Rückwärtsphase wird der V-Baum rekursiv berechnet. Der Preis einer Option, den wir auf diesem Weg erhalten, ist nicht nur korrekt im risikoneutralen Markt, sondern auch im realen Markt.

2.2.1. Das CRR-Modell

Bislang haben wir schon einen ersten Blick auf die allgemeinen Binomialbäume und ihre Beziehung zu einem wichtigen, als risikoneutrale Bewertung bekannten Prinzip werfen. Der konkrete Ansatz, den wir für unser eindimensionales Projekt wählen, ist das sogenannte CRR-Modell, das Cox, Ross und Rubinstein [7] im Jahre 1979 veröffentlicht haben. Gemäß ihrer Idee zum Binomialmodell wird eine spezielle Gleichung zur Herleitung der Faktoren u und d gewählt, sie ist:

$$u \cdot d = 1.$$

Aus dem Prinzip der risikoneutralen Bewertung folgt dann die genauen Darstellungen von u und d:

$$\begin{aligned} u &= e^{\sigma\sqrt{\Delta t}}, \\ d &= e^{-\sigma\sqrt{\Delta t}}. \end{aligned}$$

Damit kann sich der S-Baum einer Option im CRR-Modell mittels des Algorithmus 3 entwickeln. Dabei wird der Parameter d durch Faktor $1/u$ ersetzt.

Algorithmus 3: S-Baum im CRR-Modell

Eingabe : $S(0)$, u, N
Ausgabe : Alle Knoten des S-Baumes im CRR-Modell

```
1 for i = 0 to N do
      for j = 0 to i do
          S(i, j) ⟵ S(0)u^{i−2j}
      end
  end
```

Der Ansatz von Cox, Ross und Rubinstein [7] spiegelt eine Symmetrie zwischen Aufwärtsbewegung und Abwärtsbewegung des Kurses des Basiswertes wider. Sei $S(t_i) = S(0)u^k$ der Kurs zum Zeitpunkt t_i. Dann führt eine Aufwärtsbewegung des Kurses um den Faktor u in Zeitschritt t_{i+1}, gefolgt von einer Abwärtsbewegung um den Faktor $d = 1/u$ in Zeitschritt t_{i+2} oder umgekehrt, zum Kurs $S(t_{i+2}) = S(t_i)$. Somit wiederholt sich der gleiche Kurs $S(t_i)$ nach jeweils zwei Zeitperioden. Die entsprechenden Knoten im Binomialbaum liegt auf der gleichen Ebene mit dem Wert $S(0)u^k$, die wir die Wertebene k nennen. Ein Binomialbaum im CRR-Modell mit N Zeitperioden hat deshalb $2N + 1$ Wertebenen (vgl. die Abbildung 2.5), nämlich die Wertebene N bis $-N$. Das ist eine wichtige Eigenschaft des CRR-Modells, damit sich so viele Arten von Optionen wie möglich bewerten lassen können.

Weitere Details über die Verwendung numerischer Verfahren zur Optionsbewertung basierend auf dem CRR-Modell finden sich in den folgenden Unterabschnitten.

2.2.2. Bewertung von europäischen Optionen

Wir beginnen mit der Bewertung der einfachsten Option, nämlich der europäischen Option. Wenn der Basiswert $S(t)$ sich gemäß dem Binomialmodell verhält, so ergibt sich der Wert $V(i, j)$ für alle $i = N - 1, \ldots, 0$ und $j = 0, \ldots, i$ im V-Baum bei risikoneutraler Bewertung durch die Diskontierung der Summe aus den unter dem risikoneutralen Maß gewichteten Werten der Option in den Knoten $V(i + 1, j)$ und $V(i + 1, j + 1)$, genauer gilt:

$$V(i, j) = e^{-(r-g)\Delta t}(pV(i + 1, j) + qV(i + 1, j + 1)). \tag{2.1}$$

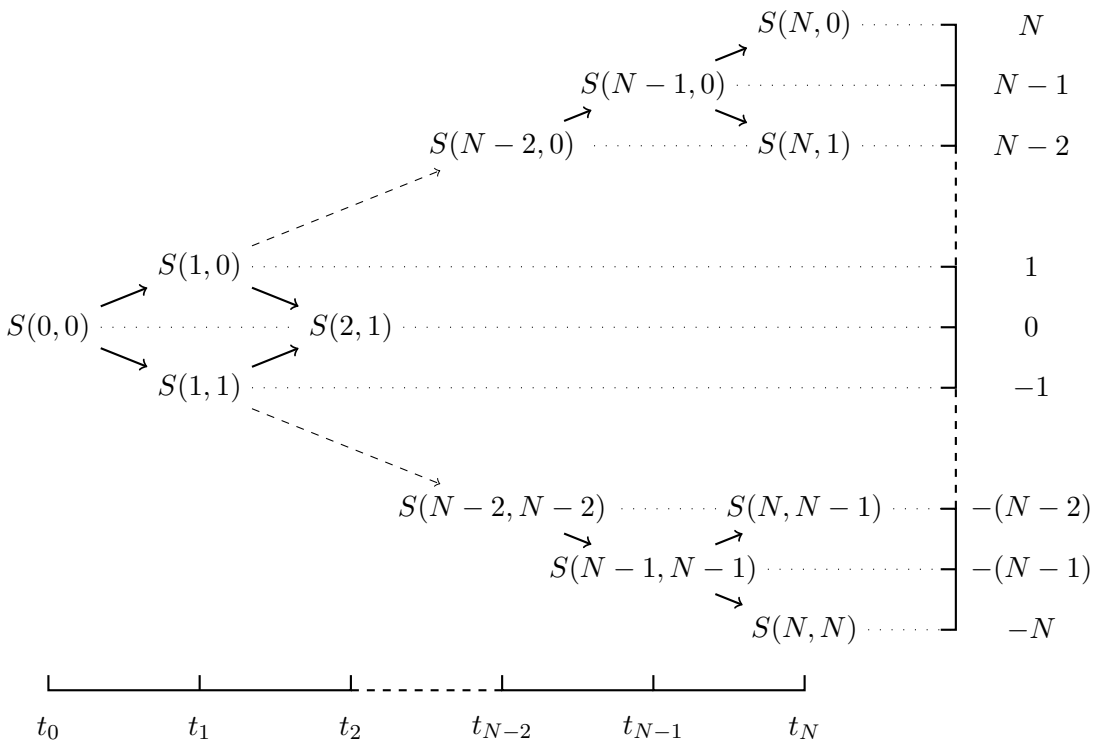


Abbildung 2.5.: S-Baum im CRR-Modell mit den Wertebenen

Basierend auf Formel (2.1) können wir eine europäische Option mittels des Algorithmus 4 bewerten.

Algorithmus 4: V-Baum einer europäischen Option

Eingabe : C-Baum, p, q, r, g, Δt, N
Ausgabe : Alle Knoten des V-Baumes

1 `// Initialisierung zum Zeitpunkt` T_N
for $j = 0$ **to** N **do**
  $V(N,j) \longleftarrow C(N,j)$
end

2 `// Diskontierung vom Zeitpunkt` T_N `zum Zeitpunkt` T_0
for $i = N-1$ **to** 0 **do**
  for $j = 0$ **to** i **do**
    $V(i,j) \longleftarrow e^{-(r-g)\Delta t}(pV(i+1,j) + qV(i+1,j+1))$
  end
end

2.2.3. Bewertung von amerikanischen Optionen

Algorithmus 4 lässt sich nun leicht auf eine amerikanische Option verallgemeinern (siehe Algorithmus 5). Die Methode besteht darin, sich vom Ende bis zum Anfang des Binomialbaumes zurückzuarbeiten und an jedem Knoten zu überprüfen, ob eine vorzeitige Ausübung sinnvoll ist. Der Wert der Option an den Knoten im Zeitschritt N ist derselbe wie der einer europäischen Option. An Knoten in jedem früheren Zeitschritt ist der Wert der Option das Maximum aus dem diskontierten risikoneutralen Auszahlungswert und der Auszahlung aus der vorzeitigen Ausübung. Wir beachten, dass wir hier nicht kontinuierlich prüfen, ob die vorzeitige Ausübung sinnvoll ist, sondern nur zu den durch das Binomialmodell vorgegebenen diskreten Zeitschritten. Diese Methode kann sich direkt auf die bermudischen Optionen anwenden lassen, dabei beschränken wir den obengenannten Vergleichsprozess nur auf die vorgegebenen Auszahlungstermine T_A.

Algorithmus 5: V-Baum einer amerikanischen Option

Eingabe : C-Baum, p, q, r, g, Δt, N
Ausgabe : Alle Knoten des V-Baumes

1 // Initialisierung zum Zeitpunkt T_N
for $j = 0$ **to** N **do**
  $V(N, j) \longleftarrow C(N, j)$
end

2 // Diskontierung vom Zeitpunkt T_N zum Zeitpunkt T_0
for $i = N - 1$ **to** 0 **do**
  for $j = 0$ **to** i **do**
    $V(i, j) \longleftarrow e^{-(r-g)\Delta t}(p \cdot V(i+1, j) + q \cdot V(i+1, j+1))$
    if $V(i, j) < C(i, j)$ **then**
      $V(i, j) \longleftarrow C(i, j)$
    end
  end
end

Wenn wir den Algorithmus 2 für den C-Baum nachvollziehen, sehen wir, dass der C-Baum schon alle Informationen von den vorgegebenen Auszahlungsterminen T_A enthält. Dabei werden die Knoten des C-Baumes in allen Terminen, bei denen die zugrunde liegende Option keine Auszahlungen hat, durch das Symbol "$\perp$" ersetzt. Dann können wir anhand des C-Baumes den V-Baum direkt berechnen, ohne dabei zu berücksichtigen, ob die Option vom europäischen, amerikanischen oder bermudischen Ausübungstyp ist. Solch ein allgemeines Verfahren formulieren wir im Algorithmus 6:

Algorithmus 6: V-Baum einer Option auf den normalen Basiswert

Eingabe : C-Baum, p, q, r, g, Δt, N
Ausgabe : Alle Knoten des V-Baumes

```
1 // Initialisierung zum Zeitpunkt T_N
  for j = 0 to N do
  |  V(N,j) ⟵ C(N,j)
  end
2 // Diskontierung vom Zeitpunkt T_N zum Zeitpunkt T_0
  for i = N − 1 to 0 do
  |  for j = 0 to i do
  |  |  V(i,j) ⟵ e^{−(r−g)Δt}(p · V(i+1,j) + q · V(i+1,j+1))
  |  |  if C(i,j) ≠ ⊥ then
  |  |  |  V(i,j) ⟵ max(V(i,j), C(i,j))
  |  |  end
  |  end
  end
```

Algorithmus 6 ist offenbar umfassender als die Algorithmen 4 und 5. Im Rest des Kapitels wollen wir alle Algorithmen mithilfe des C-Baumes wie in Algorithmus 6 beschreiben, ohne nach der Ausübungsart einer Option zu unterscheiden.

2.2.4. Bewertung von Optionen mit Barrieren

Wie wir schon erwähnt haben, kommen Barriere-Optionen hauptsächlich in zwei Formen vor, und zwar als Knock-Out-Optionen und als Knock-In-Optionen. Darüber hinaus kann der Verfall des Optionsrechts zu einer Rückvergütung (Rebate) rbt führen. Diese kann bei Knock-Out-Optionen zum Zeitpunkt des Berührens der Barriere H erfolgen. Bei Knock-In-Optionen ist die Rückvergütung rbt nur zur Fälligkeit T möglich, nämlich in dem Fall, dass eine Option endgültig nicht ausgelöst wird. Die Binomialmethode zur Bewertung von Optionen mit Barrieren wird dementsprechend in dem Knock-Out- und Knock-In-Fall unterteilt.

Knock-Out-Fall

Wir diskutieren zuerst den Knock-Out-Fall anhand einer europäischen Down-and-Out-Option mit der unteren Barriere H und Rückvergütung rbt. Dabei wird der Wert der Down-and-Out-Option in allen Knoten des V-Baumes zur Zeit t_N initialisiert. Dann erfolgt die Bewertung in einem Knoten $V(i,j)$ rekursiv je nachdem, ob der Kurs des Basiswertes in dem entsprechenden Knoten $S(i,j)$ im S-Baum über- oder unterhalb der Barriere H liegt. Im ersten Fall ergibt sich der Wert der Down-and-Out-Option im Knoten $V(i,j)$ durch die diskontierte Formel (2.1). Im zweiten Fall ist der Wert der Down-and-Out-Option im Knoten $V(i,j)$

wegen der Unterschreitung der Barriere H gleich der Rückvergütung rbt. Die genaue Formulierung sieht man im Algorithmus 7.

Algorithmus 7: V-Baum einer europäischen Down-and-Out-Option

Eingabe : S-Baum, C-Baum, p, q, r, g, H, rbt, Δt, N
Ausgabe : Alle Knoten des V-Baumes

1 // Initialisierung zum Zeitpunkt T_N
for $j = 0$ **to** N **do**
  if $S(N,j) \leq H$ **then**
    $V(N,j) \longleftarrow rbt$
  else
    $V(N,j) \longleftarrow C(N,j)$
  end
end

2 // Diskontierung vom Zeitpunkt T_N zum Zeitpunkt T_0
for $i = N-1$ **to** 0 **do**
  for $j = 0$ **to** i **do**
    if $S(i,j) \leq H$ **then**
      $V(i,j) \longleftarrow rbt$
    else
      $V(i,j) \longleftarrow e^{-(r-g)\Delta t}(p \cdot V(i+1,j) + q \cdot V(i+1,j+1))$
    end
  end
end

Durch ähnliche Überlegungen lässt sich diese Methode auch für die anderen Knock-Out-Optionen nach kleiner Modifizierung anwenden, indem das Knock-Out-Kriterium $S(i,j) \leq H$ dem entsprechenden Barriere-Typ angepasst wird. Im Anhang B verallgemeinern wir diese Methode für alle Knock-Out-Optionen durch den Algorithmus 17. Wir sehen, dass die Bewertung einer Option ohne Barriere in der Tat ein spezieller Fall zur Bewertung derselben Option mit Knock-Out-Barriere ist, wobei $S(i,j)$ für alle $i = 0,\ldots,N$ und $j = 0,\ldots,i$ niemals das Knock-Out-Kriterium erfüllt.

Knock-In-Fall

Nun diskutieren wir die Bewertung einer Knock-In-Option mit der Barriere H und Rückvergütung rbt. Dazu führen wir den $\tilde{V}$-Baum ein. Der $\tilde{V}$-Baum ist der V-Baum der äquivalenten Option der betreffenden Knock-In-Option ohne Barriere. Wir beschreiben dann den Algorithmus für eine europäische Down-and-In-Option (siehe Algorithmus 8). Dabei wird die Rückvergütung rbt nur bei der Initialisierung zur Zeit t_N ausgezahlt. Die Bewertung des Knoten $V(i,j)$ erfolgt rekursiv je nachdem, ob der Kurs des Basiswertes in dem Knoten $S(i,j)$ im S-Baum über- oder unterhalb der Barriere H liegt. Falls $S(i,j)$ oberhalb der Barriere H ist, ergibt sich der Wert der Down-and-In-Option im Knoten $V(i,j)$ durch die diskontierte Formel (2.1). Sonst ist der Wert der Down-and-In-Option

im Knoten $V(i,j)$ gleich dem Wert im Knoten $\tilde{V}(i,j)$ im $\tilde{V}$-Baum. Ein allgemeiner Algorithmus für alle Knock-In-Optionen befindet sich im Anhang B (Algorithmus 18).

Algorithmus 8: V-Baum einer europäischen Down-and-In-Option

Eingabe : S-Baum, C-Baum, p, q, r, g, H, rbt, Δt, N
Ausgabe : Alle Knoten des V-Baumes

```
1 // Initialisierung für den Ṽ-Baum und V-Baum zum Zeitpunkt T_N
for j = 0 to N do
    Ṽ(N,j) ⟵ C(N,j)
    if S(N,j) ≤ H then
        V(N,j) ⟵ Ṽ(N,j)
    else
        V(N,j) ⟵ rbt
    end
end
2 // Diskontierung des Ṽ-Baumes vom Zeitpunkt T_N zum Zeitpunkt T_0
for i = N − 1 to 0 do
    for j = 0 to i do
        Ṽ(i,j) ⟵ e^{−(r−g)Δt}(pṼ(i+1,j) + qṼ(i+1,j+1))
    end
end
3 // Diskontierung des V-Baumes vom Zeitpunkt T_N zum Zeitpunkt T_0
for i = N − 1 to 0 do
    for j = 0 to i do
        if S(i,j) ≤ H then
            V(i,j) ⟵ Ṽ(i,j)
        else
            V(i,j) ⟵ e^{−(r−g)Δt}(p · V(i+1,j) + q · V(i+1,j+1))
        end
    end
end
```

Bislang haben wir die Bewertung von Optionen mit Barrieren im Binomialmodell vorgestellt. Für eine europäische Barriere-Option gilt darüber hinaus die Behauptung, dass der Wert eines Portfolios, das aus einer Knock-In-Option und einer Knock-Out-Option mit den gleichen Barrieren besteht, gleich dem Wert der entsprechenden europäischen Option ohne Barriere ist. Wir nennen diese Tatsache die In-Out-Parität einer europäischen Barriere-Option.

Die Methoden, die wir bisher vorgestellt haben, lassen sich nicht nur im CRR-Modell durchführen, sondern auch in vielen anderen Binomialmodellen. Ab dem nächsten Unterabschnitt diskutieren wir die Binomialmethoden zur Bewertung einiger weiterer Typen von Optionen mit speziellen Binomialbäumen von Basiswerten, welche als Varianten des S-Baumes betrachtet werden und nur im CRR-Modell funktionieren. Die Knoten von solchen speziellen Binomialbäumen sind nicht mehr skalare Werte, sondern reelle Wertlisten. Außerdem wollen wir im verbleidenden Teil des Kapitels alle Algorithmen zur Berechnung des V-Baumes

bzw. der Varianten des V-Baumes in zwei Versionen aufstellen: Die Knock-Out-Version und die Knock-In-Version. Die Version ohne Barriere wird automatisch nach der Knock-Out-Version klassifiziert.

2.2.5. Bewertung von Optionen mit Lookback-Basiswerten

In diesem Unterabschnitt zeigen wir, wie die Binomialbäume so erweitert werden, dass sie Lookback-Optionen bewerten können, die vom maximalen oder minimalen Basiswert abhängen. Die entsprechenden Binomialmethoden können sowohl den europäischen Typ, als auch den amerikanischen oder bermudischen Typ von Lookback-Optionen mit oder ohne Barriere handhaben und sind rechnerisch hinreichend effizient.

Bei der Bewertung einer Lookback-Option durch die Binomialmethode ist zu beachten, dass die Auszahlungen nicht nur vom Basiswert selbst abhängen, sondern auch von einer Pfadfunktion F bzgl. des Kurses des Basiswertes (bei der Lookback-Option: $F = \max()$ oder $F = \min()$). Infolgedessen müssen verschiedene Werte der Pfadfunktion F in einem Knoten des Binomialbaumes auftreten. Unter Berücksichtigung dieses Umstandes werden wir neben dem normalen S-Baum noch einen speziellen S^F-Baum mittels der Pfadfunktion F entwickeln, nämlich den S^{max}-Baum oder S^{min}-Baum für Lookback-Option. Anders als der S-Baum steht eine Liste (oder ein Vektor) von Werten in jedem Knoten des entsprechenden S^F-Baumes. Korrespondierend muss die Berechnung des V-Baum (genau V^F-Baum) dem S^F-Baum gemäß modifiziert werden, dabei ergibt sich der C-Baum (genauer C^F-Baum) aus Algorithmus 9.

Algorithmus 9: Vom S^F-Baum zum C^F-Baum einer Option

Eingabe : S^F-Baum, N, f, T_A
Ausgabe : Alle Knoten des C^F-Baumes

```
1 for i = 0 to N do
    if t_i ∈ T_A then
      for j = 0 to i do
        for k = 1 to ♯|S^F(i,j)| do
          C^F(i,j)[k] ⟵ f(S^F(i,j)[k])
        end
      end
    else
      for j = 0 to i do
        C^F(i,j) ⟵ ⊥
      end
    end
  end
```

Wir sehen, dass ein Knoten vom C^F-Baum die gleiche Länge hat wie der entsprechenden Knoten vom zugehörigen S^F-Baum. Die Notation $\sharp\left|S^F(i,j)\right|$ steht für die Länge der Liste $S^F(i,j)$ und $C^F(i,j)[k]$ bzw. $S^F(i,j)[k]$ bezeichnet das $k$te-

Element der Werteliste $C^F(i,j)$ bzw. $S^F(i,j)$.

Das allgemeine Bewertungsverfahren besteht darin, an jedem Knoten des S^F-Baumes alle Werte für die zugrunde liegende Pfadfunktion F aufzustellen, den C^F-Baum dem Algorithmus 9 gemäß zu entwicklen und bei einer rückwärtigen rekursiven Bewertung durch den V^F-Baum den Wert der betreffenden Option für jeden Wert der Pfadfunktion F zu berechnen.

Maximaler Basiswert

Wir beginnen mit den Lookback-Optionen auf den maximalen Basiswert. Die erste Aufgabe besteht darin, den Binomialbaum des maximalen Basiswertes, nämlich den S^{max}-Baum, aufzubauen. Wir betrachten den Indexbaum in der Abbildung 2.6. Dieser ist der Binomialbaum des maximalen Basiswertes hinsichtlich der Indizes der Wertebenen. Genauer: Wir ordnen in jedem Knoten des Binomialbaumes die Indizes der Wertebene, die alle möglichen maximalen Basiswerte bis zu diesem Knoten abbilden, aufsteigend an. So entspricht z. B. der Knoten mit dem Indexvektor $(a,b,c,\ldots)^\top$ der Werteliste $(S(0)u^a, S(0)u^b, S(0)u^c, \ldots)^\top$.

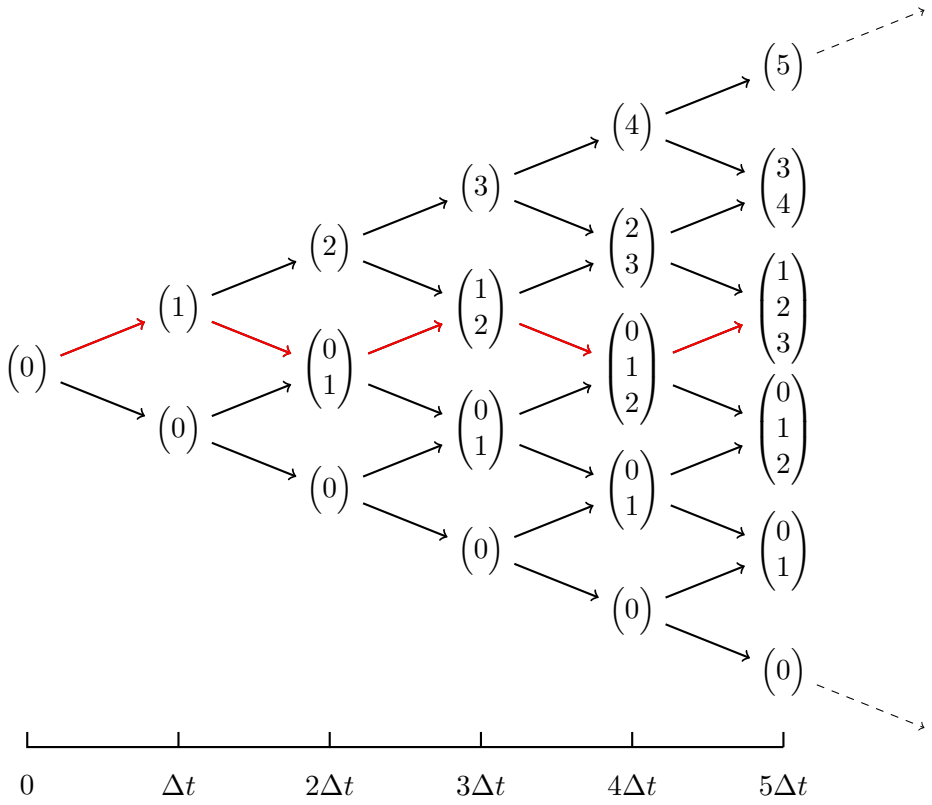


Abbildung 2.6.: Binomialbaum für den maximalen Basiswert hinsichtlich der Indizes der Wertebenen

Die Konstruktionsregel des Indexbaumes können wir anhand der Abbildung 2.6 abstrahieren. Zuerst initialisieren wir den oberen Rand des Indexbaumes, wobei jeder Knoten des oberen Randes nur den Index der zugehörigen Wertebene besitzt. Dann entwickelt sich der ganze Baum entlang der Richtung nach unten rechts nach dem folgenden Gesetz:

- Ein Knoten unterhalb des roten Pfads erbt einfach den Indexvektor vom Knoten oben links.
- Ein Knoten oberhalb oder auf dem roten Pfad verbindet den Index der zugehörigen Wertebene mit dem Indexvektor vom Knoten oben links.

In gleicher Weise können wir den S^{max}-Baum kalkulieren. Das Detail formulieren wir im Algorithmus 10.

Algorithmus 10: S^{max}-Baum für den maximalen Basiswert einer Lookback-Option

Eingabe : S-Baum, N
Ausgabe : Alle Knoten des S^{max}-Baumes

```
// Initialisierung des oberen Randes des S^max-Baum vom Zeitpunkt T_0 zum
   Zeitpunkt T_N
for i = 0 to N do
    S^max(i,0) ⟵ (S(i,0))^⊤
end
// Entwicklung des S^max-Baum vom Zeitpunkt T_0 zum Zeitpunkt T_N
for i = 1 to N do
    for j = 1 to i do
        if i/2 ≥ j then
            // Für die Knoten oberhalb oder auf dem roten Pfad
            S^max(i,j) ⟵ (S(i,j))^⊤ ⋈ S^max(i−1,j−1)
        else
            // Für die Knoten unterhalb des roten Pfads
            S^max(i,j) ⟵ S^max(i−1,j−1)
        end
    end
end
```

Dabei verbindet der Operator $\bowtie$ die Werte von zwei Wertlisten in eine gemeinsame Wertliste.

Jetzt diskutieren wir die Berechnung des V^{max}-Baumes einer Lookback-Option. Dabei spielt die Diskontierungsmethode eine wichtige Rolle, welche ungleich dem normalen V-Baum ist. Denn in jedem Knoten des V^{max}-Baumes liegt eine Wertliste anstatt eines einzelnen Wertes.

Wir beschreiben die Diskontierungsmethode anhand der Abbildung 2.7. Wenn wir den Knoten $V^{max}(i,j)$ berechnen möchten, brauchen wir die Informationen aus den Knoten $V^{max}(i+1,j)$ und $V^{max}(i+1,j+1)$. Die Länge eines jeden der beiden Knoten ist größer gleich als die Länge des Knotens $V^{max}(i,j)$, welche

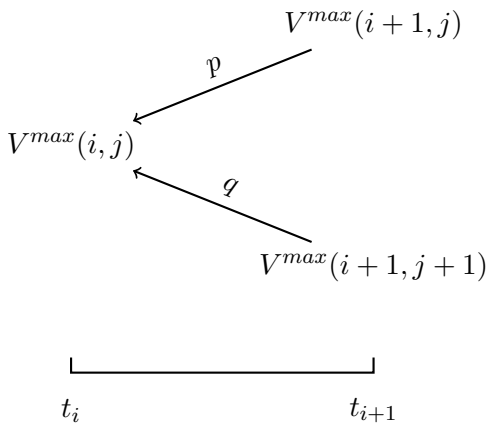


Abbildung 2.7.: Teil eines V^{max}-Baumes

der Länge des Knotens $C^{max}(i,j)$ gleicht. Angenommen, $\sharp|C^{max}(i,j)| = l$, dann diskontieren wir die ersten l Elemente von $V^{max}(i+1,j)$ und die letzten l Elemente von $V^{max}(i+1,j+1)$ durch die Formel:

$$V(i,j) \longleftarrow e^{-(r-g)\Delta t}\left(q \cdot (l \blacktriangleleft V(i+1,j)) + q \cdot (V(i+1,j+1) \blacktriangleright l)\right),$$

wobei der Operator $l \blacktriangleleft V(i+1,j)$ die ersten l Elemente der Liste $V(i+1,j)$ nimmt und der Operator $V(i+1,j+1) \blacktriangleright l$ die letzten l Elemente der Liste $V(i+1,j+1)$ nimmt. Auf diesem Wege erhalten wir den Knoten $V^{max}(i,j)$.

Im Anhang B verallgemeinern wir den vorgenannten Bewertungsansatz zur Berechnung des V^{max}-Baumes für eine Lookback-Option auf den maximalen Basiswert im Fall der Knock-Out-Barriere (siehe Algorithmus 19) oder im Fall der Knock-In-Barriere (siehe Algorithmus 20).

Minimaler Basiswert

Nun kommen wir zur Bewertung der Lookback-Optionen auf den minimalen Basiswert. Gleichermaßen beginnen wir mit dem Indexbaum in Abbildung 2.8. Beim Aufbau des Indexbaumes initialisieren wir zunächst im Gegensatz zum Fall maximales Basiswertes den unteren Rand des Indexbaumes, wobei jeder Knoten des unteren Randes nur den Index der zugehörigen Wertebene hat. Danach entwickeln wir den ganzen Baum entlang der Richtung nach oben rechts nach dem Gesetz:

- Ein Knoten oberhalb des roten Pfads erbt einfach den Indexvektor vom Knoten unten links.
- Ein Knoten unterhalb oder auf dem roten Pfad verbindet den Index der zugehörigen Wertebene mit dem Indexvektor vom Knoten unten links.

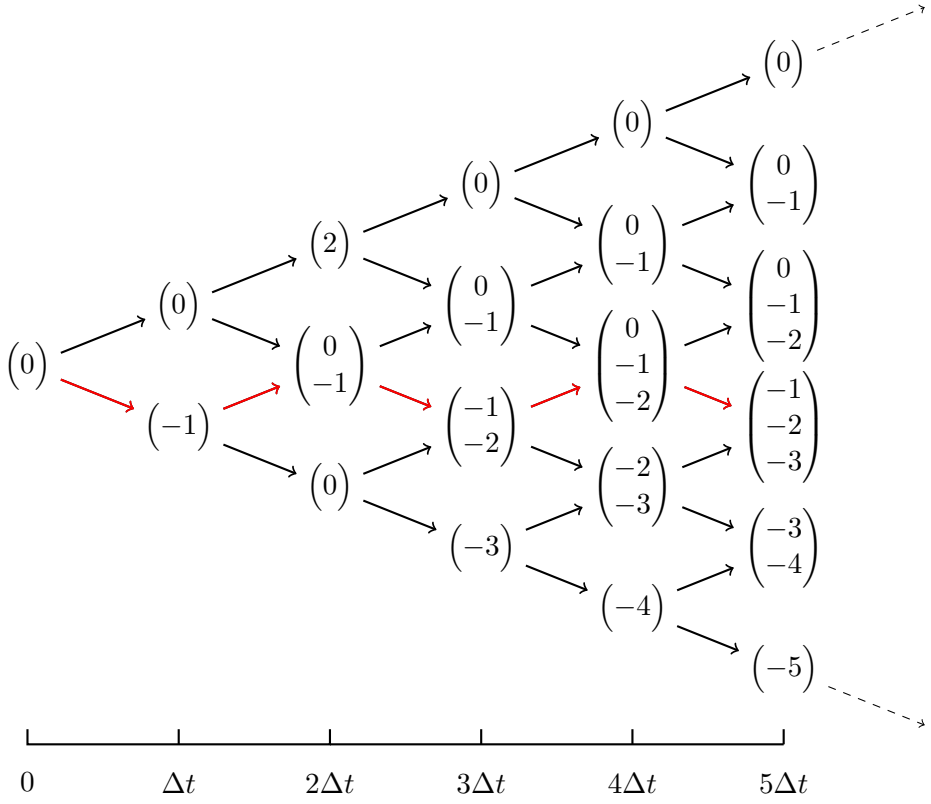


Abbildung 2.8.: Binomialbaum für den minimalen Basiswert hinsichtlich der Indizes der Wertebenen

Gleicherweise erstellen wir den S^{max}-Baum. Den Durchlauf fassen wir im Algorithmus 11 zusammen.

Algorithmus 11: S^{min}-Baum für den minimalen Basiswert einer Lookback-Option

Eingabe : S-Baum, N
Ausgabe : Alle Knoten des S^{min}-Baumes

1 // Initialisierung des unteren Randes des S^{min}-Baum vom Zeitpunkt T_0 zum Zeitpunkt T_N
for $i = 0$ **to** N **do**
  $S^{min}(i,i) \longleftarrow (S(i,i))^{\top}$
end

2 // Entwicklung des S^{min}-Baum vom Zeitpunkt T_0 zum Zeitpunkt T_N
for $i = 1$ **to** N **do**
  for $j = 0$ **to** $i - 1$ **do**
    if $\frac{i}{2} \leq j$ **then**
      // Für die Knoten unterhalb oder auf dem roten Pfad
      $S^{min}(i,j) \longleftarrow (S(i,j))^{\top} \bowtie S^{min}(i-1,j)$
    else
      // Für die Knoten oberhalb des roten Pfads
      $S^{min}(i,j) \longleftarrow S^{min}(i-1,j)$
    end
  end
end

Anschließend gehen wir auf die Diskontierungsmethode anhand der Abbildung 2.9 ein. Um den Knoten $V^{min}(i,j)$ zu berechnen, brauchen wir ebenfalls

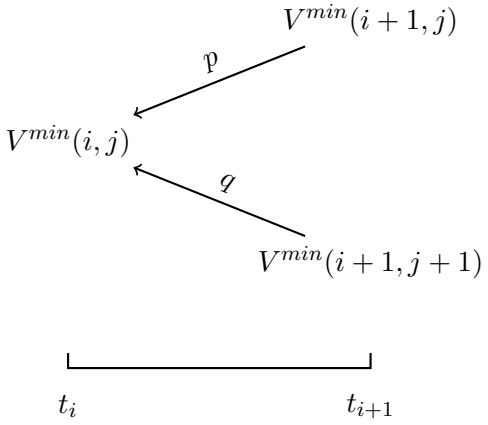


Abbildung 2.9.: Teil eines V^{max}-Baumes

die Abschnitte der Knoten $V^{max}(i+1,j)$ und $V^{max}(i+1,j+1)$. Falls $\sharp|C^{max}(i,j)| = l$ ist, dann bekommen wir den Knoten $V^{max}(i,j)$ durch die Diskontierung der ersten l Elemente von $V^{max}(i+1,j)$ und der letzten l Elemente von $V^{max}(i+1,j+1)$ hinsichtlich der folgenden Formel:

$$V(i,j) \longleftarrow e^{-(r-g)\Delta t}\left(p \cdot (V(i+1,j) \blacktriangleright l) + q \cdot (l \blacktriangleleft V(i+1,j+1))\right).$$

Zuletzt kristallisieren wir das allgemeine Bewertungsverfahren für die Lookback-Option mit dem maximalen Basiswert heraus. Nach wie vor unterscheiden wir das Bewertungsverfahren in zwei Fälle. Für den Knock-Out-Fall bzw. Knock-In-Fall formulieren wir Algorithmus 21 bzw. Algorithmus 22 im Anhang B.

2.2.6. Bewertung von Optionen mit asiatischen Basiswerten

Der eben beschriebene Bewertungsansatz für pfadabhängige Optionen ist numerisch gut zu handhaben, falls die Anzahl der verschiedenen Werte der Pfadfunktion F in jedem Knoten bei steigender Anzahl an Zeitschritten nicht zu schnell anwächst. Die Binomialmethode zur Bewertung der Lookback-Option im letzten Unterabschnitt stellt dabei kein Problem dar, da die Anzahl der verschiedenen Werte für den maximalen oder minimalen Basiswert an einem Knoten des S^{max}-Baumes oder S^{min}-Baumes mit i Zeitschritten nie größer als i ist, also linear anwächst.

Wie auch die die Lookback-Option ist die asiatische Option eine pfadabhängige Option. Deshalb können wir denselben Bewertungsansatz auf die asiatischen Optionen anwenden. Dabei berechnet die Pfadfunktion $F = ave()$ den Mittelwert aller Teilpfade im S-Baum vom Kurs des Basiswertes. Auf diese Weise können

wir den S^{ave}-Baum für eine asiatische Option erhalten. Allerdings wächst die Anzahl der Werte des Knotens im S^{ave}-Baum nicht mehr linear in den Zeitschritten wie im S^{max}-Baum bzw. S^{min}-Baum an, sondern viel schneller, was rechnerisch nicht handhabbar ist.

Hull [14] hat den Bewertungsansatz so erweitert, dass dieser Konstellationen bewältigt, bei denen an jedem Knoten des S^F-Baumes eine sehr große Anzahl verschiedener Werte der Pfadfunktion F auftritt. Die Grundidee sieht folgendermaßen aus: An jedem Knoten des S^F-Baumes werden die Berechnungen für eine kleine Anzahl repräsentativer Werte der Pfadfunktion F durchgeführt. Basierend auf den S^F-Baum entwickelt sich der C^F-Baum vorwärts und der Wert der zugrunde liegenden Option berechnet sich durch eine rückwärtige rekursive Bewertung auf dem V^F-Baum für jeden repräsentativen Wert der Pfadfunktion F. Wird ein Knoten des V^F-Baumes für andere Werte der Pfadfunktion F benötigt, ermittelt man ihn aus den bekannten Werten durch Interpolation.

Wir erläutern nun ausführlich den Konstruktionsprozess des S^{ave}-Baumes für die Bewertung einer asiatische Option nach John Hulls Ansatz. Im ersten Schritt

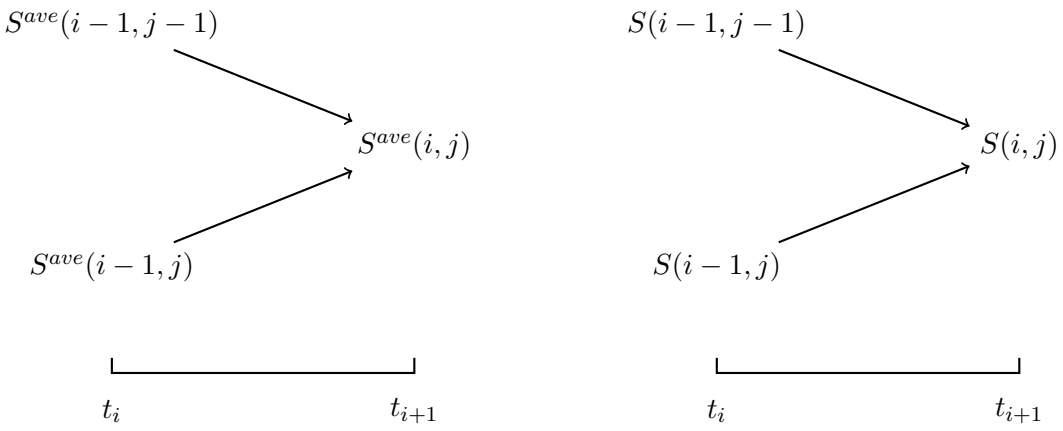


Abbildung 2.10.: Teilbaum des S^{ave}-Baumes und entsprechender Teilbaum des S-Baumes

arbeitet man sich vorwärts durch den S^{ave}-Baum und stellt dabei die maximalen und minimalen Mittelwerte des Basiswertes in jedem Knoten des S^{ave}-Baumes auf. Unter Berücksichtigung der Tatsache, dass der Mittelwert des Basiswertes zum Zeitschritt $i+1$ nur vom Mittelwert des Basiswertes zum Zeitschritt i und dem Kurs des Basiswertes zum Zeitschritt $i+1$ abhängt, können die maximalen und minimalen Mittelwerte des Basiswertes eines Knotens des S^{ave}-Baumes zum Zeitschritt $i+1$ direkt aus den Werten der beiden vorhergehenden Knoten des S^{ave}-Baumes zum Zeitschritt i und dem entsprechenden Knoten des S-Baumes

zum Zeitschritt $i+1$ berechnet werden. Man erhält also für $i = 1, \ldots, N$

$$\begin{aligned} \max\left(S^{ave}(i,j)\right) &= \frac{i \cdot \max\left(S^{ave}(i-1,j-1)\right) + S(i,j)}{i+1}, \\ \min\left(S^{ave}(i,j)\right) &= \frac{i \cdot \min\left(S^{ave}(i-1,j)\right) + S(i,j)}{i+1} \end{aligned}$$

im Fall $j = 1, \ldots, i-1$ (vgl. Abbildung 2.10) bzw.

$$\begin{aligned} S^{ave}(i,0) &= \frac{i \cdot S^{ave}(i-1,0) + S(i,0)}{i+1}, \\ S^{ave}(i,i) &= \frac{i \cdot S^{ave}(i-1,i-1) + S(i,0)}{i+1} \end{aligned}$$

im Fall $j = 0$ oder $j = i$. Wir sehen, dass es nur einen Wert in den Knoten $S^{ave}(i,0)$ und $S^{ave}(i,i)$ für $i = 0, \ldots, N$ gibt.

Der zweite Schritt besteht in der Wahl von repräsentativen Werten für jeden Knoten des S^{ave}-Baumes. Eine einfache Vorgehensweise besteht darin, als repräsentative Werte den maximalen und den minimalen Mittelwert sowie eine Anzahl von anderen Mittelwerten, die gleichmäßig zwischen dem Maximum und Minimum verteilt sind, auszuwählen. Bei der Wiederholung des Durchlaufs durch den S^{ave}-Baum erhält man für jeden Knoten die repräsentative Mittelwerte des Basiswertes.

Der Ansatz von John Hull ist einerseits effizienter als der normale Bewertungsansatz durch die Einschränkung der Anzahl der repräsentativen Mittelwerte jedes Knotens des S^{ave}-Baumes. Andererseits ist er ungenauer als der normale Ansatz wegen der Verringerung der Informationen in jedem Knoten des S^{ave}-Baumes. In unserem Projekt kombinieren wir beide Ansätze, um die Genauigkeit zu erhöhen, ohne die Effizienz von Hulls Ansatzes zu beeinträchtigen. Dabei ersellen wir den S^{ave}-Baum zunächst gemäß dem normalen Bewertungsansatz vorwärts. Falls die Länge der Werteliste eines Knotens größer ist als eine Hürde HD, wird Hulls Ansatz darauf angewendet. Also wählen wir an jedem solchen Knoten HD gleichmäßig verteilte Werte als repräsentative Werte für den Mittelwert aus. Das dem kombinierten Ansatz entsprechende Verfahren zum Aufbau des S^{ave}-Baumes einer asiatischen Option wird im Algorithmus 12 ausführlich

präsentiert:

Algorithmus 12: S^{ave}-Baum des durchschnittlichen Basiswertes einer asiatischen Option

Eingabe : S-Baum, N, HD
Ausgabe : Alle Knoten des S^{ave}-Baumes

1 // Initialisierung zum Zeitpunkt T_0
$S^{ave}(0,0) \longleftarrow (S(0,0))^{\top}$

2 // Entwicklung der beiden Ränder des S^{ave}-Baumes vom Zeitpunkt T_1 zum Zeitpunkt T_N
for $i = 1$ **to** N **do**
  $S^{ave}(i,0) \longleftarrow \frac{1}{i+1} \cdot (i \cdot S^{ave}(i-1,0) \oplus S(i,0))$
  $S^{ave}(i,i) \longleftarrow \frac{1}{i+1} \cdot (i \cdot S^{ave}(i-1,i-1) \oplus S(i,i))$
end

3 // Entwicklung der inneren Knoten des S^{ave}-Baumes vom Zeitpunkt T_1 zum Zeitpunkt T_N
for $i = 1$ **to** N **do**
  for $j = 1$ **to** $i-1$ **do**
    $l \longleftarrow \sharp\,|S^{ave}(i-1,j-1)| + \sharp\,|S^{ave}(i-1,j)|$
    if $l < HD$ **then**
      $L_a \longleftarrow \frac{1}{i+1} \cdot (i \cdot S^{ave}(i-1,j-1) \oplus S(i,j))$
      $L_b \longleftarrow \frac{1}{i+1} \cdot (i \cdot S^{ave}(i-1,j) \oplus S(i,j))$
      $S^{ave}(i,j) \longleftarrow L_a \bowtie L_b$
    else
      $a \longleftarrow \frac{1}{i+1} \cdot (i \cdot \min(S^{ave}(i-1,j)) + S(i,j))$
      $b \longleftarrow \frac{1}{i+1} \cdot (i \cdot \max(S^{ave}(i-1,j-1)) + S(i,j))$
      $S^{ave}(i,j) \longleftarrow \left(a \blacktriangleleft\!\overset{HD}{\cdots}\!\blacktriangleright b\right)^{\top}$
    end
  end
end

Die Notation $\oplus$ steht für eine Addition, bei der ein einzelner Wert komponentenweise auf eine Werteliste hinzuaddiert wird. Die Notation $a \blacktriangleleft\!\overset{HD}{\cdots}\!\blacktriangleright b$ bedeutet, dass HD repräsentative Werte gleichmäßig auf das Intervall $[a,b]$ verteilt werden.

Basierend auf dem S^{ave}-Baum kann sich den C^{ave}-Baum entwickeln, damit wir den V^{ave}-Baum berechnen können. Das Diskontierungsverfahren des V^{ave}-Baumes vom Zeitpunkt t_{i+1} zum Zeitpunkt t_i wird in vier Unterfälle unterteilt,

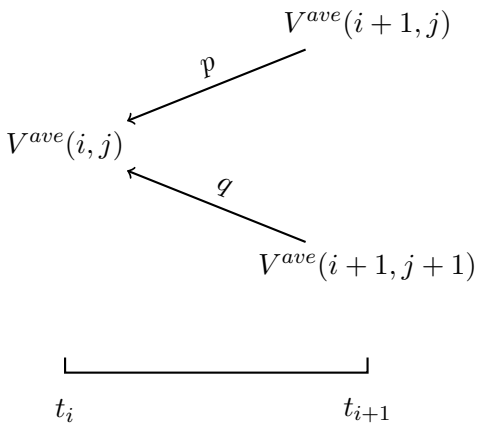


Abbildung 2.11.: Teil eines V^{ave}-Baumes

was wir anhand der Abbildung 2.11 im Algorithmus 13 zusammenfassen.

Algorithmus 13: Diskontierung des V^{ave}-Baumes vom Zeitpunkt t_{i+1} zum Zeitpunkt t_i

Eingabe : S^{ave}-Baum, C^{ave}-Baum, p, q, r, g, rbt, Δt, HD

Ausgabe : $\mathbb{DISCA}\left(C^{ave}(i,j)\middle|\begin{matrix}V^{ave}(i+1,j)\\V^{ave}(i+1,j+1)\end{matrix}\right)$

$l \longleftarrow \sharp\,|C^{ave}(i,j)|$, $l_u \longleftarrow \sharp\,|V^{ave}(i+1,j)|$, $l_d \longleftarrow \sharp\,|V^{ave}(i+1,j+1)|$

switch l_u, l_d **do**

case $(l_u < HD) \wedge (l_d < HD)$

$$\mathbb{DISCA}\left(C^{ave}(i,j)\middle|\begin{matrix}V^{ave}(i+1,j)\\V^{ave}(i+1,j+1)\end{matrix}\right) \longleftarrow$$
$$e^{-(r-g)\Delta t}\left(p\cdot(V^{ave}(i+1,j)\blacktriangleright l)+q\cdot(l\blacktriangleleft V^{ave}(i+1,j+1))\right)$$

endsw

case $(l_u < HD) \wedge (l_d = HD)$

$$\mathbb{DISCA}\left(C^{ave}(i,j)\middle|\begin{matrix}V^{ave}(i+1,j)\\V^{ave}(i+1,j+1)\end{matrix}\right) \longleftarrow$$
$$e^{-(r-g)\Delta t}\left(p\cdot(V^{ave}(i+1,j)\blacktriangleright l)+q\cdot\mathbb{IP}\left(V^{ave}(i+1,j+1)\middle|\begin{matrix}S^{ave}(i+1,j+1)\\S^{ave}(i,j)\end{matrix}\right)\right)$$

endsw

case $(l_u = HD) \wedge (l_d < HD)$

$$\mathbb{DISCA}\left(C^{ave}(i,j)\middle|\begin{matrix}V^{ave}(i+1,j)\\V^{ave}(i+1,j+1)\end{matrix}\right) \longleftarrow$$
$$e^{-(r-g)\Delta t}\left(p\cdot\mathbb{IP}\left(V^{ave}(i+1,j)\middle|\begin{matrix}S^{ave}(i+1,j)\\S^{ave}(i,j)\end{matrix}\right)+q\cdot(l\blacktriangleleft V^{ave}(i+1,j+1))\right)$$

endsw

case $(l_u = HD) \wedge (l_d = HD)$

$$\mathbb{DISCA}\left(C^{ave}(i,j)\middle|\begin{matrix}V^{ave}(i+1,j)\\V^{ave}(i+1,j+1)\end{matrix}\right) \longleftarrow$$
$$e^{-(r-g)\Delta t}\left(p\cdot\mathbb{IP}\left(V^{ave}(i+1,j)\middle|\begin{matrix}S^{ave}(i+1,j)\\S^{ave}(i,j)\end{matrix}\right)+q\cdot\mathbb{IP}\left(V^{ave}(i+1,j+1)\middle|\begin{matrix}S^{ave}(i+1,j+1)\\S^{ave}(i,j)\end{matrix}\right)\right)$$

endsw

endsw

Dabei steht $\mathbb{IP}\left(V^{ave}(i+1,j)\middle|\begin{matrix}S^{ave}(i+1,j)\\ S^{ave}(i,j)\end{matrix}\right)$ für die lineare Interpolation von $V^{ave}(i+1,j)$ hinsichtlich der Verhältnisse zwischen den Werten von $S^{ave}(i,j)$ und den Werten von $S^{ave}(i+1,j)$, In Algorithmus 14 wird diese Interpolation zusammen mit der Interpolation $\mathbb{IP}\left(V^{ave}(i+1,j+1)\middle|\begin{matrix}S^{ave}(i+1,j+1)\\ S^{ave}(i,j)\end{matrix}\right)$ durchgeführt.

Algorithmus 14: Interpolation

Eingabe : S^{ave}-Baum, u, d

Ausgabe : $\mathbb{IP}\left(V^{ave}(i+1,j)\middle|\begin{matrix}S^{ave}(i+1,j)\\ S^{ave}(i,j)\end{matrix}\right)$, $\mathbb{IP}\left(V^{ave}(i+1,j+1)\middle|\begin{matrix}S^{ave}(i+1,j+1)\\ S^{ave}(i,j)\end{matrix}\right)$

$l \longleftarrow \sharp|S^{ave}(i,j)|$, $l_u \longleftarrow \sharp|S^{ave}(i+1,j)|$, $l_d \longleftarrow \sharp|S^{ave}(i+1,j+1)|$

for $k = 1$ **to** l **do**

  switch $S_u = u \cdot S^{ave}(i,j)[k]$ **do**

    case $\exists k' \in \{1,\dots,l_u\}: \; S_u = S^{ave}(i+1,j)[k']$

      $\mathbb{IP}\left(V^{ave}(i+1,j)\middle|\begin{matrix}S^{ave}(i+1,j)\\ S^{ave}(i,j)\end{matrix}\right)[k] \longleftarrow V^{ave}(i+1,j)[k']$

    endsw

    case $\exists k' \in \{1,\dots,l_u-1\}: \; S^{ave}(i+1,j)[k'] < S_u < S^{ave}(i+1,j)[k'+1]$

      $\mathbb{IP}\left(V^{ave}(i+1,j)\middle|\begin{matrix}S^{ave}(i+1,j)\\ S^{ave}(i,j)\end{matrix}\right)[k] \longleftarrow$

      $\frac{(S_u - S^{ave}(i+1,j)[k'])\cdot V^{ave}(i+1,j)[k'+1] + (S^{ave}(i+1,j)[k'+1] - S_u)\cdot V^{ave}(i+1,j)[k']}{S^{ave}(i+1,j)[k'+1] - S^{ave}(i+1,j)[k']}$

    endsw

  endsw

  switch $S_d = S^{ave}(i,j)[k]$ **do**

    case $\exists k' \in \{1,\dots,l_d\}: \; S_d = S^{ave}(i+1,j+1)[k']$

      $\mathbb{IP}\left(V^{ave}(i+1,j+1)\middle|\begin{matrix}S^{ave}(i+1,j+1)\\ S^{ave}(i,j)\end{matrix}\right)[k] \longleftarrow V^{ave}(i+1,j+1)[k']$

    endsw

    case $\exists k' \in \{1,\dots,l_d-1\}: \; S^{ave}(i+1,j+1)[k'] < S_d < d \cdot S^{ave}(i+1,j+1)[k'+1]$

      $\mathbb{IP}\left(V^{ave}(i+1,j+1)\middle|\begin{matrix}S^{ave}(i+1,j+1)\\ S^{ave}(i,j)\end{matrix}\right)[k] \longleftarrow$

      $\frac{(S_d - S^{ave}(i+1,j+1)[k'])\cdot V^{ave}(i+1,j+1)[k'+1] + (S^{ave}(i+1,j+1)[k'+1] - S_d)\cdot V^{ave}(i+1,j+1)[k']}{S^{ave}(i+1,j+1)[k'+1] - S^{ave}(i+1,j+1)[k']}$

    endsw

  endsw

end

Bei der Berechnung des V^{ave}-Baum einer asiatischen Option berücksichtigen wir ebenso zwei Fälle: Der Knock-Out-Fall und der Knock-In-Fall. Für den Knock-Out- bzw. Knock-In-Fall formulieren wir Algorithmus 23 bzw. Algorithmus 24 im Anhang B.

2.3. Multidimensionales Binomialmodell

Nachdem wir im vorangegangenen Abschnitt das Binomialmodell für verschiedene Single-Asset-Optionen vorgestellt und diskutiert haben, wollen wir uns in diesem Abschnitt der Bewertung von Multi-Asset-Optionen widmen, worin die mathematische Grundlage für unser multidimensionales Projekt liegt. Wir werden zeigen, wie ein fairer Preis für Multi-Asset-Optionen mit Hilfe des multidimensionalen Binomialmodells bestimmt werden kann. Wir beginnen mit einer kurzen Vorstellung des klassischen multidimensionalen Binomialmodells.

Bei den multidimensionalen Binomialmodellen nehmen wir ebenfalls an, dass der Finanzmarkt arbitragefrei und vollständig ist und mit dem risikoneutralen Maß (äquivalenten Martingalmaß) P sowie dem kontinuierlichen, jährlichen risikolosen Zinssatz r ausgestattet ist.

Wir betrachten nun eine Multi-Asset-Option basierend auf den M-dimensionalen Basiswert

$$S = (S_1, \ldots, S_M)^\top$$

mit Anfangskursen

$$S(0) = (S_1(0), \ldots, S_M(0))^\top.$$

Dabei nehmen wir an, dass der Kurs jedes zugrunde liegenden Basiswertes S_i einer geometrischen Brownschen Bewegung folgt. D. h. die Basiswertkurse entsprechen einem M-dimensionalen Black-Scholes-Modell, deren Dynamik unter dem risikoneutralen Maß P durch

$$dS_i(t) = S_i(t)rdt + S_i(t)\sigma_i dW_i(t) \qquad t \in [0,T] \tag{2.2}$$

für $i = 1, \ldots, M$ gegeben sind, wobei σ_i die Volatilitäten von $S_i(t)$ und W_i die Wiener-Prozesse sind. Die Wiener-Prozesse W_i und W_j für $i \neq j$ haben die Korrelation ρ_{ij}. Das impliziert

$$Cov\left(\frac{dS_i(t)}{S_i(t)}, \frac{dS_j(t)}{S_j(t)}\right) = \sigma_i\sigma_j\rho_{ij}dt$$

für $i, j = 1, \ldots, M$. Die Korrelationen werden so angenommen, dass die entsprechende Varianz-Kovarianz-Matrix

$$\Sigma = \begin{pmatrix} \sigma_1^2 & \cdots & \rho_{1M}\sigma_1\sigma_M \\ \vdots & \ddots & \vdots \\ \rho_{1M}\sigma_1\sigma_M & \cdots & \sigma_M^2 \end{pmatrix}$$

positiv definit ist.

Ein entsprechender M-dimensionaler Binomialbaum von den obengenannten Basiswerten S_i für $i = 1, \ldots, M$ kann wie folgt aufgebaut werden. Zuerst zerlegen wir wie im eindimensionalen Fall das Zeitintervall $[0,T]$ in N äquidistante Zeitintervalle $[t_k, t_{k+1}]$ der Länge $\Delta t = T/N$ für $k = 0, \ldots, N-1$. Anschließend werden

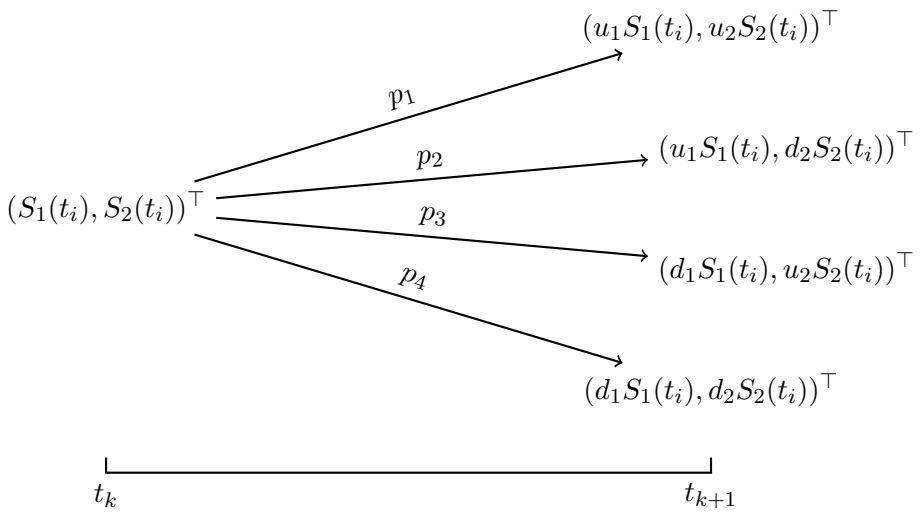


Abbildung 2.12.: Kursbewegungen des Basiswertes über einer Zeitperiode Δt im zweidimensionalen Binomialmodell

die Parameter u_i und d_i für jeden Basiswert S_i im JR-Modell (siehe Unterabschnitt 2.3.1) und die Wahrscheinlichkeiten p_l mit $l = 1, \ldots, 2^M$ für alle Zweige eines Knotens berechnet, wobei die gesamte Korrelationsstruktur berücksichtigt werden muss. Wie im eindimensionalen Fall werden alle zukünftigen Basiswertkurse, die in den Simulationszeitpunkten $\{t_0 = 0, t_1, \ldots, t_{N-1}, t_N = T\}$ auftreten können, in der Vorwärtsphase komponentenweise initialisiert. Auch in der Rückwärtsphase werden die Optionswerte an allen Knoten analog bestimmt, mit dem einzigen Unterschied zum eindimensionalen Modell, dass zur Bestimmung des Optionswertes an einem Knoten zum Zeitpunkt t_k nun 2^M Optionswerte zum Zeitpunkt t_{k+1} nötig sind. Wenn wir beispielsweise einen zweidimensionalen Binomialbaum auf den Basiswerten S_1 und S_2 entwickeln, werden die Zweige an jedem Knoten zum Zeitpunkt t_k in Abbildung 2.12 dargestellt. Anstelle von zwei Zweigen an jedem Knoten im eindimensionalen Modell, haben wir jetzt vier Zweige auf der Grundlage der vier verschiedenen möglichen Kursbewegungen.

Für unser Projekt wählen wir aber nicht das klassische multidimensionale Binomialmodell, sondern ein entkoppeltes multidimensionales Binomialmodell, welches im Jahr 2009 von Korn und Müller [20] entwickelt wurde und ein modernes und innovatives Modell basierend auf dem JR-Modell ist. Durch die Zerlegung der Varianz-Kovarianz-Matrix transformieren Korn und Müller [20] einen M-dimensionalen kontinuierlichen Basiswertkurs in einen neuen Prozess mit M unkorrelierten eindimensionalen Komponenten vor der Konstruktion eines diskreten Binomialmodells. Anhand dieser Transformation besteht der entsprechende multidimensionale Binomialbaum aus M unabhängigen entkoppelten eindimensinalen Binomialbäumen, auf denen wir die Forschungsergebnis-

se vom eindimensionalen Binomialmodell wiederverwenden können. Außerdem weist die entkoppelte multidimensionale Binomialmothode ein deutlich regulärere-res Konvergenzverhalten als die klassische multidimensionale Binomialmothode auf.

Vor der Diskussion des entkoppelten multidimensionalen Binomialmodells wollen wir das JR-Modell im eindimensionalen Fall kurz vorstellen.

2.3.1. Das JR-Modell

Das JR-Modell ist ein alternativer Ansatz des Binomialmodells zum CRR-Modell, das 1983 von Jarrow und Rudd [16] entwickelt wurde. Sie schlugen

$$p = q = \frac{1}{2} \tag{2.3}$$

vor. D. h. eine Aufwärtsbewegung vom Kurs des Basiswertes tritt um den Faktor u mit der gleichen Wahrscheinlichkeit wie eine Abwärtsbewegung um den Faktor d ein. Aus dem Prinzip der risikoneutralen Bewertung ohne Berücksichtigung der Dividendenzahlung des Basiswertes folgen dann die genauen Darstellungen von u und d:

$$\begin{aligned} u &= e^{\left(r-\frac{1}{2}\sigma^2\right)\Delta t+\sigma\sqrt{\Delta t}}, \\ d &= e^{\left(r-\frac{1}{2}\sigma^2\right)\Delta t-\sigma\sqrt{\Delta t}}. \end{aligned}$$

2.3.2. Das entkoppelte multidimensionale Binomialmodell

In diesem Unterabschnitt werden wir den für unser multidimensionales Projekt relevanten Inhalt von Korn und Müller [20] unter Verwendung ähnlicher Notationen zusammenfassen.

Die allgemeine Entkopplungsregel

Wir stellen zuerst die allgemeine Entkopplungsregel vor. Unser Ziel ist die Umwandlung der M korrelierten geometrischen Brownschen Bewegungen $S_i(t)$ für $i = 1, \ldots, M$ in M unabhängige Brownsche Bewegungen. Dafür betrachten wir den folgenden Prozess

$$\ln(S(t)) = (\ln(S_1(t)), \ldots, \ln(S_M(t)))^\top,$$

dessen Dynamik unter dem risikoneutralen Maß P durch

$$d(\ln(S_i(t))) = \left(r - \frac{1}{2}\sigma_i^2\right) dt + \sigma_i dW_i(t) \qquad X_i(0) = \ln(S_i(0))$$

für $i = 1, \ldots, M$ gegeben sind. Dann zerlegen wir die zugehörige Varianz-Kovarianz-Matrix mittels der Formel

$$\Sigma = \begin{pmatrix} \sigma_1^2 & \cdots & \rho_{1M}\sigma_1\sigma_M \\ \vdots & \ddots & \vdots \\ \rho_{1M}\sigma_1\sigma_M & \cdots & \sigma_M^2 \end{pmatrix} = GDG^\top \tag{2.4}$$

mit

$$\sigma_i\sigma_j\rho_{ij} = \sum_{k=1}^{M} g_{ik}d_{kk}g_{jk}$$

für $i, j = 1, \ldots, M$, wobei $G, D \in \mathbb{R}^{M\times M}$ und D eine Diagonalmatrix ist. Weil Σ positiv definit angenommen wurde, ist es ohne weiteres invertierbar. Somit sind G und D auch invertierbar, und die Diagonalelemente von D müssen ungleich null sein. Ferner transformieren wir den M-dimensionalen Prozess S in einen neuen Prozess Y mit M unabhängigen Komponenten:

$$S(t) \longmapsto Y(t) = G^{-1}(\ln(S(t))). \tag{2.5}$$

Unter Verwendung der neuen Notation

$$\begin{pmatrix} g_{11}^{(-1)} & \cdots & g_{1M}^{(-1)} \\ \vdots & \ddots & \vdots \\ g_{M1}^{(-1)} & \cdots & g_{MM}^{(-1)} \end{pmatrix} = G^{-1}$$

für die Terme von G^{-1} können wir den Prozess Y komponentenweise darstellen als

$$Y_i(t) = \sum_{j=1}^{M} g_{ij}^{(-1)} \ln(S_j(t))$$

für $i = 1, \ldots, M$. Die Dynamik von Y ist daher durch

$$dY(t) = \alpha dt + \begin{pmatrix} \sqrt{d_{11}} & & \\ & \ddots & \\ & & \sqrt{d_{MM}} \end{pmatrix} d\bar{W}(t) \tag{2.6}$$

mit der Initialisierung

$$Y(0) = G^{-1}(\ln(S(0)))$$

gegeben, wobei $\bar{W}(t) = \left(\bar{W}_1(t), \ldots, \bar{W}_M(t)\right)^\top$ und der Driftvektor

$$\alpha = G^{-1}\left(\underline{r} - \frac{1}{2}\underline{\sigma}^2\right)$$

mit $\underline{r} = (r, \ldots, r)^\top \in \mathbb{R}^M$ und $\underline{\sigma}^2 = (\sigma_1^2, \ldots, \sigma_M^2)^\top \in \mathbb{R}^M$. Die komponentenweisen Darstellungen der partiellen Differentialgleichung (2.6) lauten

$$dY_i(t) = \alpha_i dt + \sqrt{d_{ii}}d\bar{W}_i(t) \tag{2.7}$$

mit den Initialisierungen

$$Y_i(0) = \sum_{j=1}^{M} g_{ij}^{(-1)} \ln(S_j(0))$$

für $i = 1, \ldots, M$, wobei

$$\alpha_i = \sum_{j=1}^{M} g_{ij}^{(-1)} \left(r - \frac{1}{2}\sigma_j^2 \right).$$

Korn und Müller [20] haben gezeigt, dass die Wiener-Prozesse $\bar{W}_i(t)$ und $\bar{W}_j(t)$ für $i \neq j$ voneinander unabhängig sind. D. h. für $i \neq j$ sind die Komponenten Y_i und Y_j des Prozesses Y unkorreliert.

Wir sehen, dass die Zerlegung (2.4) im entkoppelten multidimensionalen Binomialmodell eine entscheidende Rolle spielt. Deswegen müssen wir diese im Folgenden genauer analysieren. Hierfür haben Korn und Müller [20] zwei Zerlegungsmöglichkeiten geboten. Diese sind die Eigenwertzerlegung (Spektralzerlegung) und die Cholesky-Zerlegung.

Eigenwertzerlegung

Mithilfe der Eigenwertzerlegung kann die positiv definite symmetrische Matrix Σ durch die Form

$$\Sigma = GDG^{-1}$$

zerlegt werden, wobei G eine orthogonale Matrix und D eine Diagonalmatrix ist. Jedes Diagonalelement von D ist ein Eigenwert von Σ. Da Σ symmetrisch und positiv definit ist, sind ihre Eigenwerte reell und positiv. Die Spalten- bzw. Zeilenvektoren von G bilden eine Orthonormalbasis im $\mathbb{R}^M$ und werden als normierte Eigenvektoren betrachtet. Durch die Orthogonalität von G haben wir $G^{-1} = G^\top$. Gemäß Formel (2.7) kann die Dynamik des transformierten Prozesses Y bezogen auf die Eigenwertzerlegung durch die partiellen Differentialgleichungen

$$dY_i(t) = \sum_{j=1}^{M} g_{ij}^{(-1)} \left(r - \frac{1}{2}\sigma_j^2 \right) dt + \sqrt{\lambda_i} d\bar{W}_i(t) \tag{2.8}$$

mit den Initialisierungen

$$Y_i(0) = \sum_{j=1}^{M} g_{ij}^{(-1)} \ln(S_j(0))$$

für $i = 1, \ldots, M$ beschrieben werden, wobei $\lambda_1, \ldots, \lambda_M$ die Eigenwerte der Varianz-Kovarianz-Matrix Σ und die neuen Diffusionskoeffizienten von (2.8) die Wurzeln der Eigenwerte von Σ sind.

Cholesky-Zerlegung

Die Cholesky-Zerlegung bezeichnet eine alternative Möglichkeit der Zerlegung einer positiv definiten symmetrischen Matrix. Dadurch kann die Varianz-Kovarianz-Matrix Σ eindeutig in der Form

$$\Sigma = GG^\top$$

zerlegt werden. Dabei ist $G \in \mathbb{R}^{M\times M}$ eine untere Dreiecksmatrix mit positiven Diagonalelementen und $G^\top$ die entsprechende obere Dreiecksmatrix. Die Matrix D ist hier die Einheitsmatrix, d. h. $d_{ii} = 1$ für alle $i = 1, \ldots, M$. Gemäß der allgemeinen Formel (2.7) kann die Dynamik des transformierten Prozess Y bezogen auf die Cholesky-Zerlegung durch die partiellen Differentialgleichungen

$$dY_i(t) = \sum_{j=1}^{i} g_{ij}^{(-1)} \left(r - \frac{1}{2}\sigma_j^2\right) dt + d\bar{W}_i(t) \tag{2.9}$$

mit den Initialisierungen

$$Y_i(0) = \sum_{j=1}^{i} g_{ij}^{(-1)} \ln(S_j(0))$$

für $i = 1, \ldots, M$ beschrieben werden, wobei der Index j der Summe im Drifftterm von (2.9) und im Startwert $Y_i(0)$ wegen der Dreiecksform der Matrix G nur bis zum Index i der Komponente läuft.

Für unser eigenes Projekt wählen wir die Cholesky-Zerlegung, denn laut Korn und Müller [20] hat die Cholesky-Zerlegung die folgenden Vorteile für das entkoppelte multidimensionale Binomialmodell im Vergleich zur Eigenwertzerlegung. Erstens, die Cholesky-Zerlegung von Σ ist eindeutig und stabil. Zweitens, die Cholesky-Zerlegung ist relativ einfach und effizient zu implementieren. Drittens, der Binomialbaum kann bei der Verwendung einer Cholesky-Zerlegung leicht um weitere Basiswerte erweitert werden.

2.3.3. Implementierung des entkoppelten multidimensionalen Binomialbaumes

In diesem Unterabschnitt diskutieren wir die Binomialmethode zur Bewertung von Multi-Asset-Optionen basierend auf der Entkopplungsregel. Zuerst diskretisieren wir den M-dimensionalen stochastischen Prozess Y (vgl. Formel (2.6)) mittels des Binomialbaumes. Weil der Prozess Y aus den unabhängigen eindimensionalen Y_i (vgl. Formel (2.7)) mit $i = 1, \ldots, M$ besteht, können wir den Prozess Y einfach komponentenweise implementieren. Dabei hat jede Komponente Y_i in jeder Zeitperiode $[t_k, t_{k+1}]$ für $k = 0, \ldots, N$ ebenfalls zwei verschiedene Bewegungsrichtungen u_i^Y und d_i^Y. In Übereinstimmung mit dem JR-Modell sehen sie

wie folgt aus (für Details siehe Korn und Müller [20]):

$$\begin{aligned} u_i^Y &= \alpha_i \Delta t + \sqrt{d_{ii}}\sqrt{\Delta t}, \\ d_i^Y &= \alpha_i \Delta t - \sqrt{d_{ii}}\sqrt{\Delta t}. \end{aligned}$$

Wir setzen $\Omega_i = \left\{u_i^Y, d_i^Y\right\}$ und die Wahrscheinlichkeitmaße P_i auf Ω_i für $i = 1, \ldots, M$. Dann gilt gemäß dem JR-Modell:

$$P_i\left(u_i^Y\right) = P_i\left(d_i^Y\right) = \frac{1}{2}.$$

Die Kursbewegung von Y_i über $[t_k, t_{k+1}]$ ist in Abbildung 2.13 dargestellt. Der

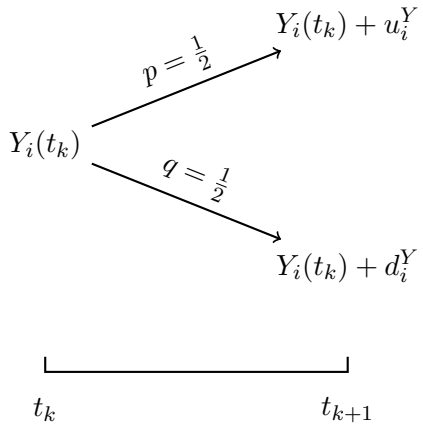


Abbildung 2.13.: Kursbewegungen des transformierten i-ten Basiswertes über einer Zeitperiode Δt nach dem entkoppelten multidimensionalen Binomialmodell

daraus folgende Y_i-Baum ist deshalb ein rekombinierbarer additiver Binomialbaum. Der vollständige Y_i-Baum sieht wie in Abbildung 2.14 aus. Der Y_1-Baum, Y_2-Baum, ... und Y_M-Baum bilden zusammen den Y-Baum. Da die Kursbewegung jedes Y_i-Baumes über einer Zeitperiode Δt zwei Möglichkeiten u_i^Y und d_i^Y hat, hat eine Kursbewegung des gesamten Y-Baum über jeder Zeitperiode 2^M Möglichkeiten. Die M-dimensionale Veränderung des Kurses über einer Zeitperiode Δt bezeichnen wir mit ω. Dann gilt

$$Y(t_{k+1}) = Y(t_k) + \omega$$

oder komponentenweise

$$\begin{pmatrix} Y_1(t_{k+1}) \\ \vdots \\ Y_M(t_{k+1}) \end{pmatrix} = \begin{pmatrix} Y_1(t_k) \\ \vdots \\ Y_M(t_k) \end{pmatrix} + \begin{pmatrix} \omega_1 \\ \vdots \\ \omega_M \end{pmatrix}$$

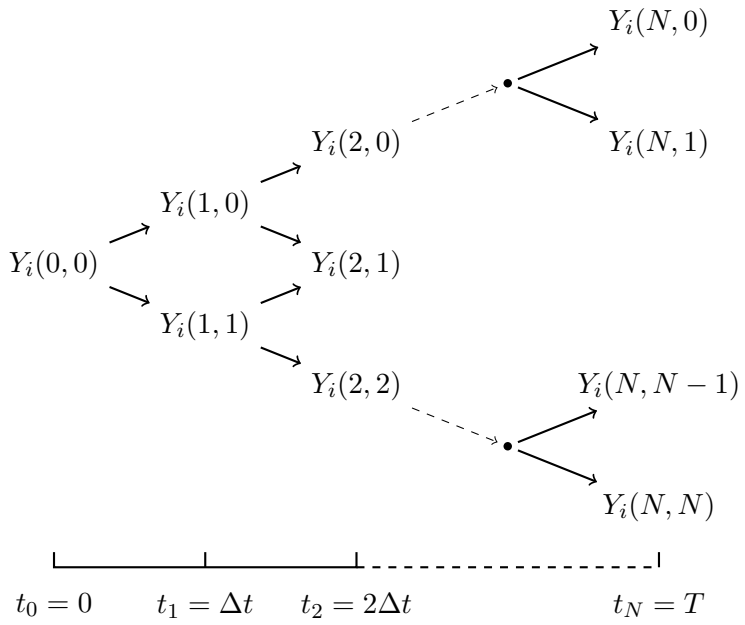


Abbildung 2.14.: Binomialbaum vom i-ten Basiswert (Y_i-Baum)

für $k = 0, \ldots, N-1$ mit

$$\omega = (\omega_1, \ldots, \omega_M)^\top \in \underbrace{\left\{u_1^Y, d_1^Y\right\}}_{=\Omega_1} \times \cdots \times \underbrace{\left\{u_M^Y, d_M^Y\right\}}_{=\Omega_M}.$$

Wir definieren weiter die Menge

$$\Omega = \Omega_1 \times \cdots \times \Omega_M$$

und das Wahrscheinlichkeitsmaß

$$P = P_1 \otimes \cdots \otimes P_M$$

auf Ω, wobei $\times$ das kartesische Produkt von zwei Mengen und $\otimes$ das Produktmaß bezeichnet. Die Menge Ω hat offenbar 2^M Elemente, nämlich 2^M verschiedene Kursbewegungen des Prozesses Y über jeder Zeitperiode. So können wir die Menge Ω auch elementarweise darstellen

$$\Omega = \left\{\omega[l] \in \mathbb{R}^M \,\middle|\, l = 1, \ldots, 2^M\right\}.$$

Dem Prinzip des JR-Modells gemäß folgt

$$P(\omega) = \prod_{i=1}^{M} P_i(\omega_i) = \prod_{i=1}^{M} \frac{1}{2} = 2^{-M} \tag{2.10}$$

für alle $\omega \in \Omega$ oder $P(\omega[l]) = 2^{-M}$ für alle $l = 1, \ldots, 2^M$. D. h. alle 2^M möglichen Kursbewegungen von $Y(t_k)$ zu $Y(t_{k+1})$ treten mit gleicher Wahrscheinlichkeit auf. Das impliziert, dass alle Pfade des entkoppelten M-dimensionalen Binomialbaumes vom Prozess Y auch gleich wahrscheinlich sind.

Jetzt konstruieren wir den Y-Baum nach unseren Bedürfnissen. Dafür setzen wir

$$Y^{(k)} = Y_1^{(k)} \times \cdots \times Y_M^{(k)}$$

für $k = 0, \ldots, N$ und

$$Y_i^{(k)} = \{Y_i(k,0), \ldots, Y_i(k,k)\}$$

für $i = 0, \ldots, M$, wobei $\times$ das kartesische Produkt zweier Mengen ist. Die Knotenmenge $Y^{(k)}$ versammelt offenbar alle Knoten des Y-Baumes zur Zeit t_k. Da $Y_i^{(k)}$ jeweils $k+1$ Knoten für $i = 1, \ldots, M$ hat, hat $Y^{(k)}$ insgesamt $(k+1)^M$ Knoten. Deshalb können wir die Knotenmenge $Y^{(k)}$ des Y-Baumes auch elementenweise beschreiben:

$$Y^{(k)} = \left\{ Y^{(k)}[l] \in \mathbb{R}^M \,\middle|\, l = 0, \ldots, (k+1)^M - 1 \right\}.$$

Dabei fängt der Index des Knotens nach wie vor von null an. Abbildung 2.15 zeigt eine grobe Darstellung des Y-Baumes in der obigen Notation. Darüber hin-

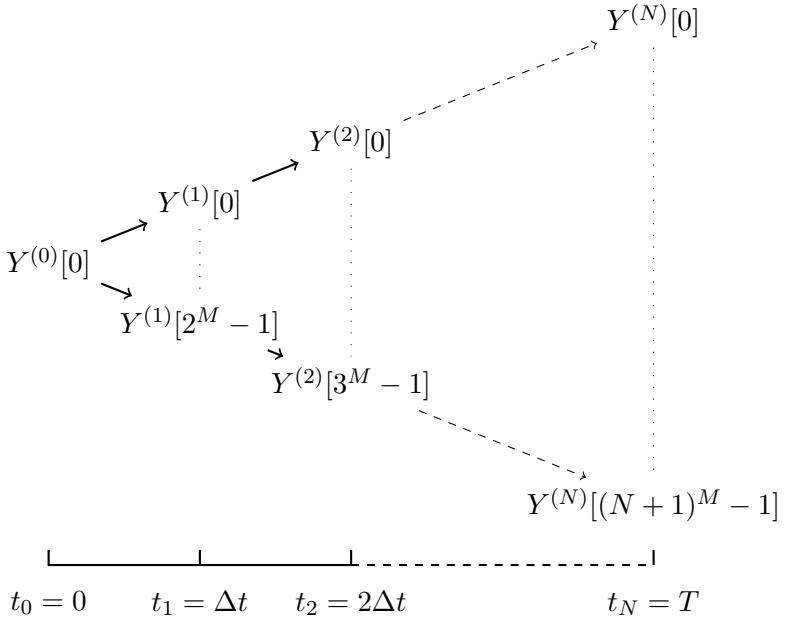


Abbildung 2.15.: Y-Baum im entkoppelten multidimensionalen Binomialmodell

aus gibt es $(2 \cdot (k+1))^M$ Kanten zwischen den Knotenmengen $Y^{(k)}$ und $Y^{(k+1)}$. Die entsprechende Methode zur Konstruktion des Y-Baumes formulieren wir im Algorithmus 15.

Algorithmus 15: Y-Baum im entkoppelten multidimensionalen Binomialmodell

Eingabe : M, $Y(0)$, u^Y, d^Y, N
Ausgabe : Sämtliche Knotenmengen des M-dimensionalen Y-Baumes

```
// Entwicklung der Y_i-Bäume für i = 1,...,M
for i = 1 to M do
    Y_i(0,0) = Y_i(0)
    for k = 1 to N do
        Y_i(k,0) ⟵ Y_i(k-1,0) + u_i^Y
        for j = 1 to k do
            Y_i(k,j) ⟵ Y_i(k-1,j-1) + d_i^Y
        end
    end
end
// Aufbau des Y-Baumes durch die Zusammenbindung der M Y_i-Bäume
for k = 0 to N do
    for i = 1 to M do
        Y_i^(k) ⟵ ∅
        for j = 0 to k do
            Y_i^(k) ⟵ Y_i^(k) ∪ Y_i(k,j)
        end
    end
    Y^(k) ⟵ Y_1^(k) × ··· × Y_M^(k)
end
```

Nach dem Aufbau des Y-Baumes wollen wir die folgende Funktion einführen:

$$\begin{aligned} h \quad : \quad & \mathbb{R}^M \longrightarrow \mathbb{R}^M, \\ & x \longmapsto \left(e^{G_{1\cdot}x}, \ldots, e^{G_{M\cdot}x}\right)^\top, \end{aligned}$$

wobei $G_{i\cdot}$ für $i = 1, \ldots, M$ die i-te Zeile der Matrix G bezeichnet. Die Funktion h ist die Umkehrfunktion der Transformation (2.5) vom Prozess S zum Prozess Y, d. h. sie transformiert den Prozess Y in den Prozess S zurück. Im Fall der Cholesky-Zerlegung sieht h wegen der Dreiecksform der Matrix G wie folgt aus:

$$\begin{aligned} h \quad : \quad & \mathbb{R}^M \longrightarrow \mathbb{R}^M, \\ & x \longmapsto \left(e^{g_{11}x_1}, \ldots, e^{\sum_{j=1}^{i} g_{ij}x_j}, \ldots, e^{\sum_{j=1}^{M} g_{Mj}x_j}\right)^\top. \end{aligned}$$

Wir wandeln jetzt mittels der Funktion h den Y-Baum in den S-Baum um, der die Diskretisierung des M-dimensionalen Basiswertes S einer Multi-Asset-Option (vgl. die stochastischen Prozesse (2.2)) in Baumform darstellt. Dazu definieren wir die Knotenmenge

$$S^{(k)} = h\left(Y^{(k)}\right) = \left\{h(y) \,\middle|\, y \in Y^{(k)}\right\}$$

für $k = 0, \ldots, N$. Weil die Knotenmenge $Y^{(k)}$ jeweils $(k+1)^M$ Elemente besitzt, hat $S^{(k)}$ ebenso $(k+1)^M$ Elemente. Somit können wir die Knotenmenge $S^{(k)}$ auch elementenweise darstellen:

$$S^{(k)} = \left\{ S^{(k)}[l] = h\left(Y^{(k)}[l]\right) \middle| \; l = 0, \ldots, (k+1)^M - 1 \right\}.$$

Aus den Knotenmengen $S^{(0)}, \ldots, S^{(N)}$ wird der gesamte S-Baum zusammengestellt, welcher eine ähnliche Struktur hat wie der Y-Baum. Basierend auf dem S-Baum können wir die zugrunde liegende Multi-Asset-Option bewerten, also den V-Baum (siehe Abbildung 2.16) berechnen. Die Form des V-Baumes ist fast

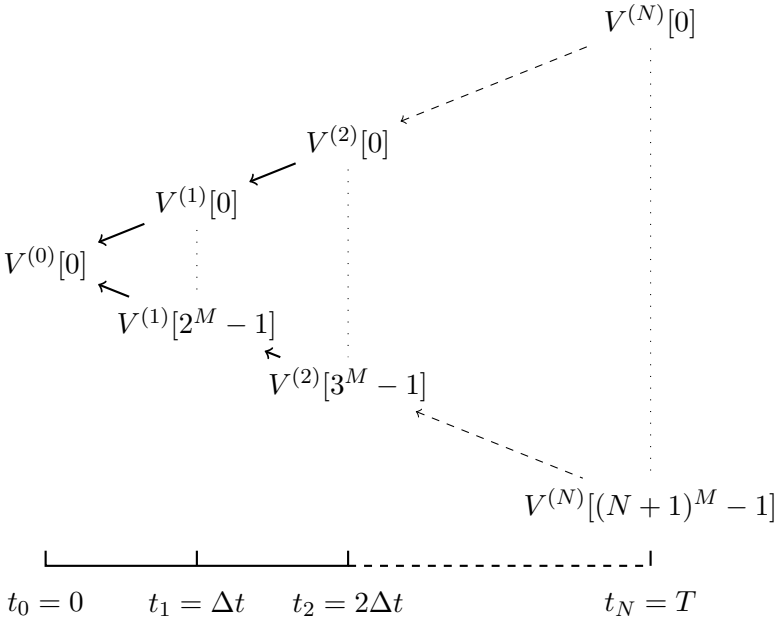


Abbildung 2.16.: V-Baum im entkoppelten multidimensionalen Binomialmodell

gleich der des Y-Baumes bzw. S-Baumes bis auf die Richtung der Baumentwicklung, weil der V-Baum nach wie vor von t_N bis t_0 rückwärts berechnet werden muss, während sich der Y-Baum und S-Baum von t_0 bis t_N vorwärts entwickeln.

Der Schwerpunkt der Berechnung des V-Baumes besteht in seiner Diskontierungsregel. Die Abbildung 2.17 zeigt die Diskontierung von 2^M Knoten im Zeitpunkt t_{k+1} zu einem Knoten im Zeitpunkt t_k des V-Baumes. Dabei sind die Wahrscheinlichkeiten aller rückwärtigen Pfade gleich 2^{-M}, was vollkommen regelrecht dem Prinzip des JR-Modells gemäß ist (vgl. Formel (2.10)). Wir formulieren dann die Diskontierungsfunktion disc() von der Knotenmenge $V^{(k+1)}$ zur Knotenmenge $V^{(k)}$ in den Algorithmus 16.

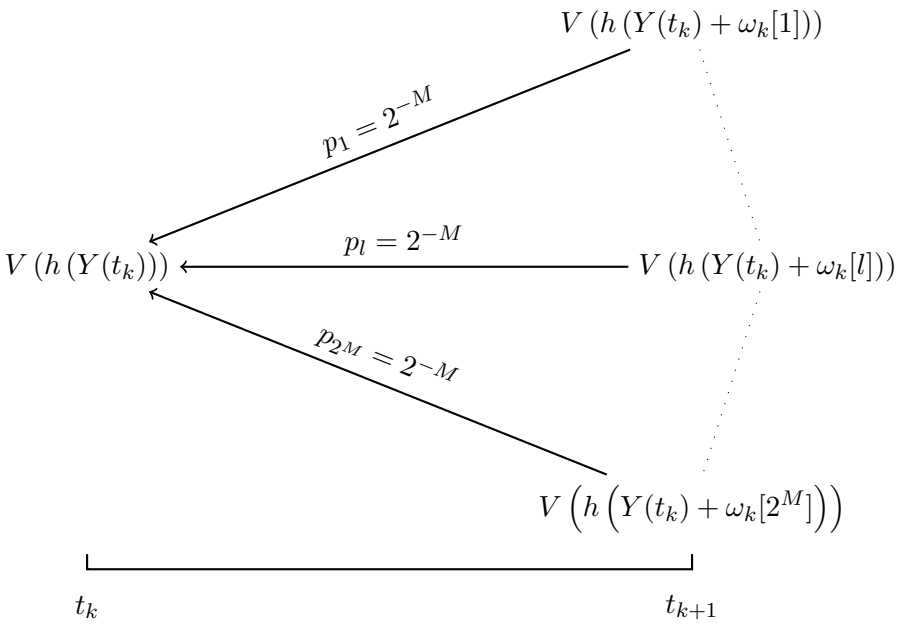


Abbildung 2.17.: Berechnung des Wertes eines Knotens in der Rückwärtsphase eines entkoppelten multidimensionalen Binomialbaumes

Algorithmus 16: disc-Funktion zum Diskontieren des Wertes einer Multi-Asset-Option von t_{k+1} zu t_k für $k = 0, \ldots, N-1$

Eingabe : $V^{(k+1)}$, Y-Baum, h, M, V_t
Ausgabe : $V^{(k)} = \mathsf{disc}(V^{(k+1)})$

1 **for** $l = 0$ **to** $(k+1)^M - 1$ **do**
  $v \longleftarrow 0$
  for $j = 1$ **to** 2^M **do**
    $v \longleftarrow v + V_{t_{k+1}}\left(h\left(Y^{(k)}[l] + \omega[j]\right)\right)$
  end
  $V^{(k)}[l] \longleftarrow \frac{e^{-r\Delta t} \cdot v}{2^M}$
end

Bislang haben wir schon das Prinzip der entkoppelten multidimensionalen Binomialmethode zur Bewertung von Multi-Asset-Optionen eingehend vorgestellt. Die verallgemeinerten Bewertungsverfahren für Multi-Asset-Optionen kann man im Anhang B nachschlagen. Diese werden in drei Fälle unterschieden: Multi-Asset-Option ohne Barriere (siehe Algorithmus 25), Multi-Asset-Option mit Knock-Out-Barrieren (siehe Algorithmus 26) und Multi-Asset-Option mit mindestens einer Knock-In-Barriere (siehe Algorithmus 27). Dabei bezeichnet die Menge T_A weiterhin die Menge aller Auszahlungstermine. Der Reihenfolge nach ist jeder Fall komplizierter als der andere. Wir werden die letzten zwei Fälle

kurz diskutieren.

Die Bewertung einer multidimensionalen Barriere-Option ist schwieriger als der eindimensionale Barriere-Option, da bei ihnen mehr als ein Basiswert mit Barrieren zusammenarbeiten könnte. Wir berücksichtigen zuerst Multi-Asset-Optionen, die nur Knock-Out-Barrieren aufweisen. In diesem Fall ist die betreffende Option relativ einfach zu bewerten, unter der Regel, dass die Option sofort entwertet wird, wenn ihre Basiswerte eines der Knock-Out-Kriterien erreichen. Wenn wir eine Multi-Asset-Option mit mindestens einer Knock-In-Barriere bewerten möchten, unterscheiden wir die folgenden zwei Unterfälle. Im ersten Unterfall betrachten wir Multi-Asset-Optionen, die lediglich Knock-In-Barrieren haben. Hierbei wird eine Option erst ausgelöst, wenn eines der Knock-In-Kriterien erfüllt wird. Der zweite Unterfall behandelt Multi-Asset-Optionen mit sowohl Knock-In- als auch Knock-Out-Barrieren, was komplizierter ist. Dabei nehmen wir an, dass die Knock-Out-Barriere vorrangiger ist als die Knock-In-Barriere. D. h. falls die Basiswerte der Multi-Asset-Option eines der Knock-Out-Kriterien erfüllt, wird die Option verfallen, selbst wenn die Knock-In-Kriterien in kommenden Zeitpunkten erfüllt werden. Die Option tritt erst in Kraft, wenn eines der Knock-In-Kriterien eintritt, sofern die Knock-Out-Kriterien bis zu diesem Zeitpunkt niemals erfüllt wurden.

3. Grundidee von Composable-Derivative-Contracts

In diesem Kapitel wollen wir eingehend die Grundidee unseres Projektes diskutieren, die auf dem Konzept zu “Composing-Contracts” basiert, welches von Peyton-Jones, Eber und Seward in den Artikeln [30] und [31] am Anfang dieses Jahrhundertes eingeführt wurde. Wir präsentieren zuerst ein physikalisches Beispiel, um einen ersten Eindruck von der Grundidee zu erhalten.

3.1. Einführung anhand eines physikalischen Beispiels

Wir betrachten in diesem Abschnitt ein physikalisches Beispiel, welches eine ähnliche Idee wie unser eigenes Projekt hat. Die Fragestellung des Beispiels ist den Gesamtwiderstand von einer beliebigen gemischten elektronischen Schaltung mit Widerständen zu berechnen. Wir wollen die Aufgabe durch die im Anhang A eingeführte funktionale Programmiersprache `Haskell` (d. h. mittels der funktionalen Denkweise) lösen. Das physikalische Beispiel ist zwar viel einfacher als unser Projekt, aber sinnvoll zu diskutieren. Denn wir können die Erfahrungen daaus auswerten und diese auf unser eigenes Projekt methodisch anwenden.

Zum Anfang werden die physikalischen Grundlagen des Beispiels kurz erwähnt. Eine gemischte elektronische Schaltung beeinhaltet Reihen- und Parallelschaltungen. In der Reihenschaltung sind zwei Teilschaltungen hintereinander verschaltet, durch die der gleiche Strom fließt. Bei der Parallelschaltung dagegen gibt es einen Knotenpunkt mit zwei Abzweigen, die jeweils eine Teilschaltung enthalten, an denen sich der Strom aufteilt. Ein Widerstand kann als die kleinste Teilschaltung angesehen werden.

Der Widerstand R^r einer Reihenschaltung setzt sich aus den Widerständen R_1^r und R_2^r der beiden seriellen Teilschaltungen zusammen:

$$R^r = R_1^r + R_2^r. \tag{3.1}$$

Der Widerstand R^p einer Parallelschaltung errechnet sich aus den Widerständen R_1^p und R_2^p der beiden parallelen Teilschaltungen nach der Formel:

$$R^p = \frac{R_1^p \cdot R_2^p}{R_1^p + R_2^p}. \tag{3.2}$$

Der Gesamtwiderstand einer gemischten elektronischen Schaltung entspricht einem Ersatzwiderstand, durch den man die gesamte elektronische Schaltung ersetzen könnte und durch den der gleiche Gesamtstrom fließen kann. Um den Gesamtwiderstand zu errechenen, wird die gesamte elektronische Schaltung in einzelne Ersatzwiderstände zerlegt, welche wiederum Teilschaltungen in der Art der Reihen- und Parallelschaltung bilden. Dann kann der Gesamtwiderstand durch Berechnung aus den Ersatzwiderständen von Teilschaltungen nach mehrmaliger Verwendungen der Formeln (3.1) und (3.2) berechnet werden. Dabei ersetzt man schrittweise alle Teilschaltungen durch Ersatzwiderstände bis auf einen echten Widerstand. D. h. wir rechnen den Gesamtwiderstand durch einen rekursiven Berechnungsprozess, was zu den Stärken der funktionalen Programmierung gehört. Ein von uns formulierter `Haskell`-Code für diese Aufgabe sieht wie folgt aus.

Programm 3.1: `Haskell`-Evaluator für Gesamtwiderstand einer gemischten elektronischen Schaltung

```
1  -- Einheit Ohm
2  type Ohm = Double
3
4  -- Primitive Konstruktoren einer gemischten elektronischen
       Schaltung
5  data Schaltung = Leitung
6                 | Widerstand Ohm
7                 | Reihen Schaltung Schaltung
8                 | Parallel Schaltung Schaltung
9
10 -- Bewertung des Widerstands einer gemischten elektronischen
       Schaltung
11 eval :: Schaltung -> Ohm
12 eval Leitung          = 0
13 eval (Widerstand r)   = r
14 eval (Reihen s1 s2)   = (eval s1) + (eval s2)
15 eval (Parallel s1 s2) = ((eval s1) * (eval s2)) / ((eval s1) +
16                          (eval s2))
```

In der zweiten Zeile des Programms 3.1 definieren wir einen Datentyp `Ohm` durch das Typsynonym (siehe Seite 195), wobei `Ohm` die abgeleitete Einheit des elektrischen Widerstands mit dem Einheitenzeichen Ω ist. In Zeile 5 bis Zeile 8 wird ein rekursiver algebraischer Datentyp (vgl. Seite 197) `Schaltung` erstellt, der aus vier Bauteilen (primitiven Konstruktoren) `Leitung`, `Widerstand`, `Reihen` und `Parallel` besteht, durch die man ein reales Objekt von gemischter elektronischer Schaltung konstruieren könnte. Die physikalischen Bedeutungen von diesen vier primitiven Konstruktoren werden in Tabelle 3.1 erklärt. Natürlich können wir anhand der vier primitiven Konstruktoren jede gemischte elektro-

`Leitung`	Elektrische Leitung
`Widerstand Ohm`	Elektrischer Widerstand mit Größe der Einheit Ω
`Reihen Schaltung Schaltung`	Reihenschaltung mit zwei Teilschaltungen
`Parallel Schaltung Schaltung`	Parallelschaltung mit zwei Teilschaltungen

Tabelle 3.1.: Sinne von primitiven Konstruktoren des Datentyps `Schaltung`

nische Schaltung mit Widerständen in den Datentyp `Schaltung` konstruieren. Jedoch kann eine solche Konstruktion im Fall einer komplizierten Schaltung zu langatmig sein. `schaltungAB` in Abbildung 3.1 ist ein Beispiel für eine gemischte elektronische Schaltung, wobei $R_i = i\ \Omega$ für alle $i = 1, \ldots, 8$ wäre. Wir konstruie-

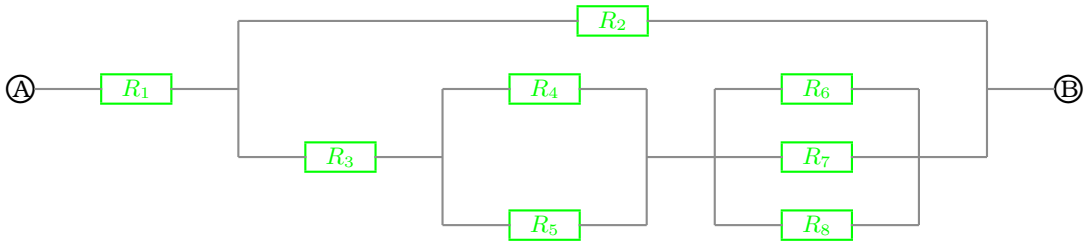


Abbildung 3.1.: Gestaltung von `schaltungAB`

ren nun `schaltungAB` in `Haskell` mithilfe der vier primitiven Konstruktoren.

```
1 schaltungAB = Reihen (Widerstand 1) (Parallel (Widerstand 2)
2                (Reihen (Widerstand 3) (Reihen (Parallel
3                (Widerstand 4) (Widerstand 5)) (Parallel
4                (Widerstand 6) (Parallel (Widerstand 7)
5                (Widerstand 8))))))
```

Um solch eine lang ausschweifende Konstruktion zu verkürzen und zu verdeutlichen können wir eine kleine Klasse bilden, deren Mitglieder `Haskell`-Funktionen mit dem Wertebereich `Schaltung` sind und im Prinzip durch die vier primitiven Konstruktoren kombiniert werden. Diese Mitglieder sind zwar nicht-primitive Konstruktionsfunktion, sind aber wie auch die primitiven Konstruktoren sehr oft in Gebrauch. Basierend auf den primitiven Konstruktoren wird die Klasse nützlicher Konstruktionsfunktionen nach Bedarf erweitert. Deswegen nennen wir diese Klasse die erweiterbare Bibliothek von Konstruktionsfunktionen.

Programm 3.2: Erweiterbare Bibliothek von Konstruktionsfunktionen

```
1 -- Kuerzungen von Widerstaenden mit Groessen
```

```
2 r1 = Widerstand 1 :: Schaltung -- R1 = 1 Ohm
3 r2 = Widerstand 2 :: Schaltung -- R2 = 2 Ohm
4 r3 = Widerstand 3 :: Schaltung -- R3 = 3 Ohm
5 r4 = Widerstand 4 :: Schaltung -- R4 = 4 Ohm
6 r5 = Widerstand 5 :: Schaltung -- R5 = 5 Ohm
7 r6 = Widerstand 6 :: Schaltung -- R6 = 6 Ohm
8 r7 = Widerstand 7 :: Schaltung -- R7 = 7 Ohm
9 r8 = Widerstand 8 :: Schaltung -- R8 = 8 Ohm
10
11 -- Reihenschaltung mit drei Teilschaltungen
12 reihen3 :: Schaltung -> Schaltung -> Schaltung -> Schaltung
13 reihen3 s = Reihen s $ Reihen
14
15 -- Parallelschaltung mit drei Teilschaltungen
16 parallel3 :: Schaltung -> Schaltung -> Schaltung -> Schaltung
17 parallel3 s = Parallel s $ Parallel
```

Mit Hilfe der erweiterbaren Bibliothek von Konstruktionsfunktionen können wir wiederum`schaltungAB` wie folgt in `Haskell` beschreiben, was sichtlich einfacher und kürzer als die erste Darstellung ist:

```
1 schaltungAB = Reihen r1 (Parallel r2 (reihen3 r3
2               (Parallel r4 r5) (parallel3 r6 r7 r8)))
```

Nun wollen wir die `eval`-Funktion kurz erklären, die ab Zeile 11 bis zum Ende des Programms 3.1 definiert wird und durch die der Gesamtwiderstand einer gemischten elektronischen Schaltung berechnet werden kann. Zeile 11 gibt die Signatur der `eval`-Funktion an, welche `Schaltung -> Ohm` lautet. D. h. wir geben eine gemischte elektronische Schaltung in die `eval`-Funktion ein, die durch die primitiven Konstruktoren von `Schaltung` in `Haskell` beschrieben worden ist. Daraus können wir einen `Double`-Wert mit dem Datentyp `Ohm` erhalten, der in der Tat der gewünschte Gesamtwiderstand ist. Zeile 12 läßt den Widerstand einer elektrischen `Leitung` auf null sinken. Zeile 13 zieht die Größe mit der Einheit `Ohm` aus einem einzelnen elektronischen Widerstand heraus. Zeile 14 und 15 verwenden unmittelbar die Bewertungsformeln (3.1) und (3.2) auf Seite 47 mithilfe der Funktionen höherer Ordnung (vgl. Abschnitt A.2.6).

Übergibt man nun beispielsweise `schaltungAB` der `Haskell`-Funktion `eval`, so erhält man direkt den Gesamtwiderstand der `schaltungAB`, nämlich $2.58\ \Omega$.

Jedoch deckt die `eval`-Funktion noch nicht sämtliche Eingabemöglichkeiten ab. Wenn jemand versehentlich `Parallel Leitung Leitung` in den Datentyp `Schaltung` zur Konstruktion einer gemischten elektronischen Schaltung eingibt, wird die Fehlermeldung “Division durch Null” ausgegeben. Um diese mögliche Fehlermeldung zu vermeiden, werden im Folgenden zwei verbesserte Versionen der `eval`-Funktion vorgeschlagen, die unter Einsatz des `Maybe`-Datentyps bzw. `Maybe`-Monads (vgl. Abschnitt A.2.7) das obige Problem erledigen können:

Programm 3.3: `eval`-Funktion (verbesserte Version 1)

```
1 -- Bewertung des Widerstands einer gemischten elektronischen
     Schaltung mittels des Maybe-Monads
2 eval :: Schaltung -> Maybe Ohm
3 eval Leitung          = Nothing
4 eval (Widerstand r)   = Just r
5 eval (Reihen s1 s2)   = case eval s1 of
6                           Nothing -> eval s2
7                           Just x  -> case eval s2 of
8                                        Nothing -> Just x
9                                        Just y  -> Just $ x + y
10 eval (Parallel s1 s2) = eval s1 >>= \x -> eval s2 >>= \y ->
11                         return $ (x * y) / (x + y)
```

Das Programm 3.3 vermeidet das oben erwähnte Problem durch das `Maybe`-Monad. Dabei wird eine gemischte elektronische Schaltung mit der Größe null als `Nothing` bewertet. Im Programm 3.4 schreiben wir die Kette der monadischen Operationen durch die `do`-Syntax um, sodass die Bedeutung des Codes anschaulicher wird:

Programm 3.4: `eval`-Funktion (verbesserte Version 2)

```
1 -- Bewertung des Widerstands einer gemischten elektronischen
     Schaltung mittels des Maybe-Monads und der do-Syntax
2 eval :: Schaltung -> Maybe Ohm
3 eval Leitung          = Nothing
4 eval (Widerstand r)   = Just r
5 eval (Reihen s1 s2)   = case eval s1 of
6                           Nothing -> eval s2
7                           Just x  -> case eval s2 of
8                                        Nothing -> Just x
9                                        Just y  -> Just $ x + y
10 eval (Parallel s1 s2) = do x <- eval s1
11                            y <- eval s2
12                         return $ (x * y) / (x + y)
```

Nun kehren wir zu unserem eigenen Projekt zurück, dessen Grundidee der des oben vorgestellten physikalischen Beispiels ähnlich ist. Während wir in dem physikalischen Beispiel den Gesamtwiderstand einer gemischten elektronischen Schaltung von Widerständen berechnen, würden wir in unserem eigenen Projekt den fairen Preis eines Finanzderivats ermitteln. Allerdings hat unser eigenes Projekt ein vergleichbares Bewertungsverfahren zu dem des physikalischen Beispiels, das wir im Folgenden kurz schrittweise (bzgl. des physikalischen Beispiels) zusammenfassen:

1. Entwerfe eine vernünftige ausgewählte Klasse von primitiven Konstruktoren (Tabelle 3.1) und eine erweiterbare Bibliothek mit ausreichend vielen

Konstruktionsfunktionen (Programm 3.2). Durch diese beiden kann man beliebig viele Zielobjekte (gemischte elektronische Schaltungen) kombinieren.

2. Konstruiere eine Bewertungsfunktion (`eval`-Funktion), die auf den betreffenden mathematischen Grundlagen (Formeln (3.1) und (3.2)) basiert. Ihr Definitionsbereich (`Schaltung`) entspricht dem Datentyp vom Zielobjekt und ihr Wertebereich (`Ohm`) ist der Datentyp vom Zielwert (Gesamtwiderstand).

In dem nächsten Abschnitt werden wir Peyton-Jones' Konzept zu *Composing-Contracts*, welches den Ausgangspunkt unseres eigenen Projekt darstellt, gemäß dem obigen Bewertungsverfahren genau diskutieren. Wegen der höheren Komplexität brauchen wir dafür neben den primitiven Konstruktoren und der Bewertungsfunktion noch eine passende Bewertungsstruktur. D. h. für unser eigenes Projekt müssen die drei relevanten Teilkonzepte sorgfältig entworfen und perfekt koordiniert werden

3.2. Peyton-Jones' Konzept zu Composing-Contracts

Die Finanzindustrie hat einen enormen Wortschatz der Fachsprache für typische Kombinationen von finanziellen Kontrakten (Forwards, Futures, europäische Optionen, eermudische Optionen, amerikanische Optionen, Lookback-Optionen, Knock-In-Optionen, Knock-Out-Optionen, ...). Um jeden dieser finanziellen Kontrakte individuell zu behandeln, müssten wir einen großen Katalog von vorgefertigten Bauteilen erstellen. Falls wir dann aber einen neuen finanziellen Kontrakt betrachten möchten, der nicht bereits in dem vorhandenen Katalog ist, müssten wir diesen finanziellen Kontrakt dem Katalog hinzufügen, wodurch diese immer weiter wachsen würde und kein Ende abzusehen wäre.

Wenn wir nun stattdessen eine feste, relativ kleine Klasse von primitiven Konstruktoren definieren könnten, mit denen jeder finanzielle Kontrakt dargestellt werden könnte, hätten wir einen enormen Vorteil gegenüber dem katalogbasierten Ansatz. Einerseits ist es viel einfacher die vorhandenen und auch viele neue, unvorhergesehene finanzielle Kontrakte zu kombinieren. Andererseits können wir die finanziellen Kontrakte systematisch analysieren und die Berechnungen nicht nur über die vorhandenen, sondern auch über die neuen Kontrakte führen, weil diese lediglich aus einer festen, nicht allzu großen Klasse von primitiven Konstruktoren kombiniert werden.

Der Schwerpunkt dieses Abschnitts besteht darin, die Analyse finanzieller Kontrakte mithilfe der Kenntnisse aus der funktionalen Programmierung zu unterstützen. Wir werden nun Peyton-Jones' Vorgehensweise zur Modellierung finanzieller Kontrakte präsentieren, die er ursprünglich in den Artikeln [30] und [31] über das Konzept zu *Composing-Contracts* vorgestellt hat. Dabei wird ein

Überblick des Konzepts zu *Composing-Contracts* und seiner Vorteile zusammen mit einer einfachen Implementierung in `Haskell` eingeführt. Das originale `Haskell`-Programm und das vollständige Code-Dokument wurden 2007 von Anton van Straaten entwickelt undauf der folgenden Webseite veröffentlicht:

http://contracts.scheming.org/

Die erforderlichen Vorkenntnisse über finanzielle Kontrakte bzw. die funktionale Programmiersprache `Haskell` werden in Kapitel 2 bzw. Anhang A behandelt.

3.2.1. Konstruktion von finanziellen Kontrakten

Der erste Schritt ist ganz klar der Ansatz zur Konstruktion von finanziellen Kontrakten. Wie wir aus dem physikalischen Beispiel im Abschnitt 3.1 erfahren haben, ist das Design der primitiven Konstruktoren am wichtigsten für die Konstruktion, weil wir dadurch die erweiterbare Bibliothek bzw. alle gewünschten finanziellen Kontrakte kombinieren können.

Primitive Konstruktoren

Peyton-Jones hat in seinem Konzept zu *Composing-Contracts* den interessanten Datentyp Observable (`Obs a`) eingeführt und damit die primitiven Konstruktoren entworfen. Unter einem Objekt vom Datentyp Observable (`Obs a`) versteht man im Allgemeinen eine zeitliche veränderliche Datenmenge vom Datentyp `a`, wobei eine Datenmenge in diesem Zusammenhang nicht ein einziger Wert ist, sondern eine Zufallsvariable, d. h. eine Reihe von möglichen Werten. Anton van Straaten hat alle primitiven Konstruktoren (vgl. Programm 3.5) der *Composing-Contracts* exakt durch den `Haskell`-Code präsentiert:

Programm 3.5: Implementierung der primitiven Konstruktoren

```
1  data Contract =
2      Zero
3    | One  Currency
4    | Give Contract
5    | And  Contract Contract
6    | Or   Contract Contract
7    | Cond     (Obs Bool)   Contract Contract
8    | Scale    (Obs Double) Contract
9    | When     (Obs Bool)   Contract
10   | Anytime  (Obs Bool)   Contract
11   | Until    (Obs Bool)   Contract
12   deriving Show
```

Aus Formel (A.1) wissen wir, dass der Datentyp eines Datentypkonstruktors als eine Funktion angesehen wird. Deshalb können wir die obigen primitiven

Konstruktoren auf die folgende Art und Weise funktionalisieren. Der Vorteil der Funktionalisierung (vgl. Programm 3.6) besteht darin, dass wir einen finanziellen Kontrakt rein aus Funktionen kombinieren können, und eine gemischte Darstellung mit Funktionen und Datentypkonstruktoren vermieden wird.

Programm 3.6: Funktionalisierung der primitiven Konstruktoren

```
1  zero :: Contract
2  zero = Zero
3
4  one :: Currency -> Contract
5  one = One
6
7  give :: Contract -> Contract
8  give = Give
9
10 cAnd :: Contract -> Contract -> Contract
11 cAnd = And
12
13 cOr :: Contract -> Contract -> Contract
14 cOr = Or
15
16 cond :: Obs Bool -> Contract -> Contract -> Contract
17 cond = Cond
18
19 scale :: Obs Double -> Contract -> Contract
20 scale = Scale
21
22 cWhen :: Obs Bool -> Contract -> Contract
23 cWhen = When
24
25 anytime :: Obs Bool -> Contract -> Contract
26 anytime = Anytime
27
28 cUntil :: Obs Bool -> Contract -> Contract
29 cUntil = Until
```

Die Bedeutungen dieser primitiven Konstruktionsfunktionen werden anschließend der Reihe nach genau erläutert:

- `zero` (Programm 3.6, Zeile 1) ist ein finanzieller Kontrakt, der keine Rechte, Geld zu erhalten, und keine Pflichten, Geld zu bezahlen, beinhaltet.
- `one k` (Programm 3.6, Zeile 4) ist ein finanzieller Kontrakt, der dem Inhaber des Kontraktes sofort eine Einheit der Währung `k` (z. B. `USD`, `GBP`, `EUR` u. s. w.) auszählt.

- Der finanzielle Kontrakt `give c` (Programm 3.6, Zeile 7) wandelt alle Rechte des Kontraktes `c` in Pflichten und alle Pflichten des Kontraktes `c` in Rechte um. Das bedeutet, dass für einen bilateralern Kontrakt `q` zwischen den beiden Kontrahenten A und B gilt: A erwirbt den Kontrakt `q` impliziert, dass B den Kontrakt `give q` erwirbt.

- Wenn man den finanziellen Kontrakt `cAnd c1 c2` (Programm 3.6, Zeile 10) erwirbt, dann erwirbt man unverzüglich sowohl den Kontrakt `c1`, als auch den Kontrakt `c2`.

- Erwirbt man den finanziellen Kontrakt `cOr c1 c2` (Programm 3.6, Zeile 13), dann erwirbt man unverzüglich entweder den Kontrakt `c1` oder den Kontrakt `c2`, jedoch nicht die beiden.

- Wenn man den finanziellen Kontrakt `cond b c1 c2` (Programm 3.6, Zeile 16) erwirbt, dann erwirbt man den Kontrakt `c1`, falls das Observable `b` **`True`** ist, und den Kontrakt `c1` sonst.

- Wenn man den finanziellen Kontrakt `scale o c` (Programm 3.6, Zeile 19) erwirbt, dann erwirbt man den Kontrakt `c` im gleichen Augenblick mit der Zusatzbedingung, dass die Zahlungen bzgl. aller Rechte und Pflichten des Kontraktes `c` mit dem Wert des Observables `o` zu jeden Erwerbungszeitpunkten multipliziert werden.

- Wenn man den finanziellen Kontrakt `cWhen o c` (Programm 3.6, Zeile 22) erwirbt, so muss man den Kontrakt `c` kaufen, sobald das Observable `o` **`True`** wird. Der Kontrakt ist wertlos in den Zuständen, in denen das Observable `o` nie wieder **`True`** wird.

- Wenn man den finanziellen Kontrakt `anytime o c` (Programm 3.6, Zeile 25) erwirbt, darf man den Kontrakt `c` zu allen Zeitpunkten erwerben, in denen das Observable `o` **`True`** ist. Der Kontrakt ist daher wertlos in den Zuständen, in denen das Observable `o` nie wieder **`True`** wird.

- Einmal erworben, ist der finanzielle Kontrakt `cUntil o c` (Programm 3.6, Zeile 28) genau wie Kontrakt `c`. Dieser Kontrakt muss aufgegeben werden, sobald das Observable `o` **`True`** ist. In den Zuständen, in denen das Observable `o` **`True`** ist, wird der Kontrakt wertlos, weil er sofort aufgegeben werden muss.

Die Kombination durch die primitiven Konstruktionsfunktionen ist sehr vielfältig. Z. B. können wir eine Konstruktionsfunktion folgendermaßen kombinieren:

```
1 andGive :: Contract -> Contract -> Contract
2 andGive c d = c `cAnd` give d
```

Der Inhaber des finanziellen Kontraktes `andGive c d` erwirbt alle Rechte und Pflichten des Kontraktes `c` und die Umkehrung aller Rechte und Pflichten des Kontraktes `d`. Obwohl `andGive` nicht zu den primitiven Konstruktionsfunktionen gehört, können wir `andGive` ggf. direkt zur Konstruktion eines finanziellen Kontraktes verwenden. D. h. wir halten `andGive` für eine nicht-primitive Konstruktionsfunktion.

Jetzt beschreiben wir anwendungsbezogen einen realen finanziellen Kontrakt, die sogenannte Nullkuponanleihe, mit der man 100 € zum Zeitpunkt 10 erhalten kann. Diese Nullkuponanleihe wird durch die Komposition von nicht weniger als drei primitiven Konstruktionsfunktionen dargestellt. Wir beginnen mit der primitiven Konstruktionsfunktion `one` (Programm 3.6, Zeile 4):

```
1 c1 = one EUR
```

Wenn wir `c1` erwerben, dann erhalten wir sofort 1 € zum aktuellen Zeitpunkt, nämlich dem Zeitpunkt null. Aber bei der Nullkuponanleihe möchten wir 100 € bekommen. Dafür wenden wir die primitive Konstruktionsfunktion `scale` (Programm 3.6, Zeile 19) auf den Kontrakt `c1` an:

```
1 c2 = scale (konst 100) c1
```

wobei die Funktion `konst 100` die Konstante 100 auf ein Observable hebt, dessen Wert zu jedem Zeitpunkt 100 € ist. Nun können wir mit dem Kontrakt `c2` 100 € erhalten, jedoch nicht zum Zeitpunkt 10, sondern zum Zeitpunkt null. Um den gewünschten finanziellen Kontrakt richtig zu beschreiben, benutzen wir die primitive Konstruktionsfunktion `cWhen` (Programm 3.6, Zeile 22):

```
1 c3 = when (at (mkDate 10)) c2
```

wobei die Funktion `at (mkDate 10)` ein boolesches Observable ist, das zum Zeitpunkt 10 **`True`** wird und sonst **`False`** ist. Dann läßt sich zusammenfassend eine Konstruktionsfunktion für die Nullkuponanleihe kombinieren:

```
1 zcb :: Date -> Double -> Currency -> Contract
2 zcb t x k = cWhen (at t) (scale (konst x) (one k))
```

Diese Definition der `zcb` ergänzt effektiv unsere erweiterbare Bibliothek von (nicht-primitiven) Konstruktionsfunktionen, ebenso nützlich wie `andGive`.

Erweiterbare Bibliothek von Konstruktionsfunktionen

Nachstehend sind beispielhaft einige sehr häufig benutzte Konstruktionsfunktionen aus der erweiterbaren Bibliothek des Peyton-Jones' Konzeptes aufgeführt.

```
1 zcb :: Date -> Double -> Currency -> Contract
2 zcb t x k = cWhen (at t) (scale (konst x) (one k))
```

`zcb t x k` ist ein finanzieller Kontrakt, dessen Inhaber `x` Einheiten der Währung `k` zum Zeitpunkt `t` bekommen wird.

```
1 andGive :: Contract -> Contract -> Contract
2 andGive c d = c `cAnd` give d
```

Der Inhaber des finanziellen Kontraktes `andGive c d` erwirbt alle Rechte und Pflichten des Kontraktes `c` und die Umkehrung aller Rechte und Pflichten des Kontraktes `d`.

```
1 european :: Date -> Contract -> Contract
2 european t u = cWhen (at t) (u `cOr` zero)
```

`european t u` ist ein finanzieller Kontrakt, der einer europäischen Option entspricht. Zum Zeitpunkt `t` kann der Inhaber entweder den Kontrakt `u` erwerben oder darauf verzichten. Dabei ist `at t` ein boolesches Observable, das zum Zeitpunkt `t` **`True`** wird.

```
1 american :: (Date, Date) -> Contract -> Contract
2 american (t1, t2) u = anytime (between t1 t2) u
```

`american (t1, t2) u` ist ein finanzieller Kontrakt, der einer amerikanischen Option entspricht und dessen Inhaber folglich den Kontrakt `u` irgendwann zwischen den Zeitpunkten `t1` und `t2` erwerben oder endgültig darauf verzichten kann. Dabei ist `between t1 t2` ein boolesches Observable, das zwischen den Zeitpunkten `t1` und `t2` **`True`** wird.

Die Liste der nicht-primitiven Konstruktionsfunktionen kann nach Bedarf immer verlängert werden, wodurch sich die erweiterbare Bibliothek von Konstruktionsfunktionen vergrößert.

3.2.2. Bewertungsfunktion

Aus der Untersuchung des physikalischen Beispiels in Abschnitt 3.1 wissen wir, dass neben den primitiven Konstruktoren auch die Bewertungsfunktion von hoher Relevanz für solche ein Projekt ist. In diesem Unterabschnitt wollen wir die Bewertungsfunktion zur Ermittlung des fairen Preises eines finanziellen Kontraktes vorstellen, die eng mit den primitiven Konstruktoren aus Abschnitt 3.2.1 verbunden ist.

Datentyp der Bewertungsstruktur

Anders als das physikalische Beispiel werden wir zuerst in *Composing-Contracts* neben der Bewertungsfunktion noch die sogenannte Bewertungsstruktur einführen. Die Bewertungsstruktur ist ein infomationstechnologisches Objekt zur Speicherung und Verwaltung von Daten. Es handelt sich um eine Struktur, in der die Daten in einer bestimmten Art und Weise angeordnet und verknüpft werden, um den Zugriff und die Verwaltung auf diese besser zu ermöglichen. Die Bewertungsstruktur sind daher nicht nur durch ihre beinhalteten Daten

charakterisiert wird, sondern vor allem durch die Operationen auf diesen Daten, welche Zugriff und Verwaltung realisieren. Die Bewertungsstruktur von *Composing-Contracts* wird folgenderweise in `Haskell` definiert:

```
1 newtype PR a = PR {unPr :: [RV a]} deriving Show
2 type RV a = [a]
```

Die Bewertungsstruktur `PR a` präsentiert einen stochastischen Prozess vom Datentyp `a`. Laut ihrer Definition ist sie in `Haskell` ein Datentyp von einer zweidimensionalen Liste. Aber aus finanzmathematischer Sicht halten wir die Bewertungsstruktur `PR a` eher für einen Binomialbaum als für eine Liste. Siehe hierzu Abbildung 3.2 bezogen auf das Beispiel

```
1 PR [[0],[1,1],[2,2,2],[3,3,3,3]] :: PR Int
```

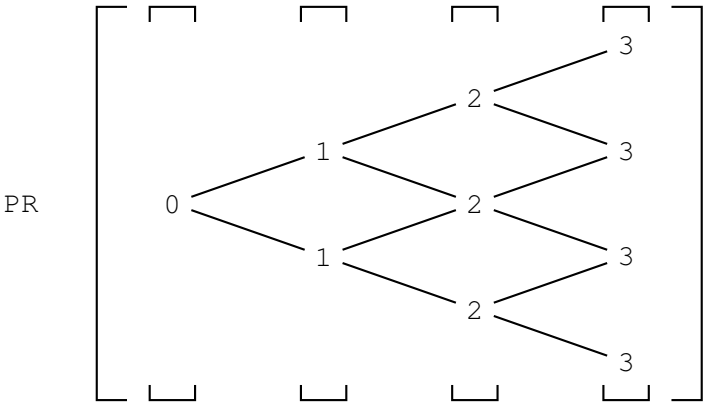


Abbildung 3.2.: Verdeutlichung der baumartigen Bewertungsstruktur von *Composing-Contracts* anhand eines Beispiels

Um die Bewertungsfunktion perfekt vorzubereiten, stellen wir nachfolgend noch die wichtigen arithmetischen und booleschen Operatoren (vgl. Programm 3.7) auf dem Datentyp `PR a` vor, die von den Typklassen `Num` und `Ord` (siehe Abschnitt A.2.3) vererbt worden sind. Solche vererbte Operationen haben die gleichen Bedeutungen wie im Datentyp `a`, doch werden sie auf dem Datentyp `PR a` elementweise der Liste (dem Binomialbaum) entlang durchgeführt, auch im Fall der zweistelligen Operatoren `(+)`, `(-)` und `(*)`, sogar wenn die zwei Argumente Listen (Binomialbäumen) mit verschiedenen Längen sind. Die Implementierung solcher Operatoren sehen wir in Programm 3.7:

Programm 3.7: Fundamentale Operatoren auf `PR a`

```
1 instance Num a => Num (PR a) where
2   fromInteger i = bigK (fromInteger i)
3   (+) = lift2PrAll (+)
4   (-) = lift2PrAll (-)
```

```
5    (*) = lift2PrAll (*)
6    abs = liftPr abs
7    signum = liftPr signum
8
9  instance Ord a => Ord (PR a) where
10   max = lift2Pr max
```

Mithilfe der Funktionen `liftPr` und `lift2PrAll` können die fundamentalen Operatoren die elementweise Operation bewältigen. Die Funktion `lift2PrAll` realisiert zusätzlich die Behandlung von zwei Listen mit verschiedenen Längen. In einem Beispiel haben wir zwei Listen (Binomialbäume) vom Datentyp `PR Int`:

```
1 b1 = PR [[1],[1,1],[1,1,1]] :: PR Int
2 b2 = PR [[2],[2,2],[2,2,2],[2,2,2,2]] :: PR Int
```

Nach der Addition der beiden `(b1 + b2)` erhalten wir das folgende Ergebnis:

```
1 PR [[3],[3,3],[3,3,3],[2,2,2,2]] :: PR Int
```

Implementierung der Bewertungsfunktion

Basierend auf der Bewertungsstruktur `PR a` können wir in diesem Unterabschnitt mit der Implementierung Bewertungsfunktion fortfahren. Dazu schlug Peyton-Jones eine Reihe von primitiven Funktionen vor:

Programm 3.8: Implementierung der Bewertungsfunktion

```
1  evalC :: Currency -> Contract -> PR Double
2  evalC k = eval
3    where
4      eval Zero            = bigK 0
5      eval (One k2)        = exch k k2
6      eval (Give c)        = -(eval c)
7      eval (o `Scale` c)   = (evalO o) * (eval c)
8      eval (c1 `And` c2)   = (eval c1) + (eval c2)
9      eval (c1 `Or` c2)    = max (eval c1) (eval c2)
10     eval (Cond o c1 c2) = condPr (evalO o) (eval c1) (eval c2)
11     eval (When o c)      = disc   k (evalO o, eval c)
12   -- eval (Anytime o c)  = snell  k (evalO o, eval c)
13     eval (Until o c)     = absorb k (evalO o, eval c)
```

Der Definitionsbereich von `eval` ist `Contract`. Beim Pattern-Matching von `eval` genügt es, die Bewertung der primitiven Konstruktoren zu implementieren. Denn die anderen Instanzen von `Contract` werden von den primitiven Konstruktoren kombiniert. Diese Instanzen können in `Haskell` wegen der Funktionen höherer Ordnung durch die Bewertungsfunktion `evalC` automatisch bewertet werden.

Die Bedeutungen dieser primitiven Funktionen werden nun der Reihe nach genau erklärt:

- Gegeben einen reellen Wert `a`, stellt die primitive Funktion `bigK a` (Programm 3.8, Zeile 4) einen konstanten reellwertigen Prozess dar, dessen Zustände alle gleich dem Wert `a` sind.
- Die primitive Funktion `exch k k2` (Programm 3.8, Zeile 5) repräsentiert einen in Währung `k2` ausgedrückten reellwertigen Prozess durch einen in Währung `k` ausgedrückten reellwertigen Prozess, wobei der Wechselkurs zwischen den Währungen `k` und `k2` vorgegeben wird.
- Die primitiven Funktionen `(-)`, `(*)`, `(+)` und `max` (Programm 3.8, Zeile 6-9) haben wir schon auf Seite 58 erklärt, die genau die fundamentalen Operatoren auf dem Datentyp `PR Double` sind.
- Gegeben einen Prozess `o` von booleschem Datentyp, wandelt die primitive Funktion `condPr o c1 c2` (Programm 3.8, Zeile 10) den reellwertigen Prozess `c` in einen anderen reellwertigen Prozess um. In den Zuständen, wo `o` **`True`** ist, sind die Ergebnisse die gleichen wie `c1`. In den übrigen Zuständen sind die Ergebnisse die gleichen wie `c2`.
- Gegeben einen Prozess `o` von booleschem Datentyp, wandelt die primitive Funktion `disc k (o,c)` (Programm 3.8, Zeile 11) den reellwertigen Prozess `c` in einen anderen reellwertigen Prozess innerhalb derselben Währung `k` um. In den Zuständen, in denen `o` **`True`** ist, sind die Ergebnisse gleich `c`. In den anderen Zuständen sind die Ergebnisse gleich den Erwartungswerten, die dem diskontierten stochastischen Prozess bzgl. `c` in der gleichen Währung `k` entsprechen.
- Die primitive Funktion `snell k (o,c)` (Programm 3.8, Zeile 12) dient eigentlich zur Bewertung eines finanziellen Kontraktes vom amerikanischen Typ. Allerdings wurde diese in van Straatens Programm nicht implementiert.
- Gegeben einen Prozess `o` von booleschem Datentyp, wandelt die primitive Funktion `absorb k (o,p)` (Programm 3.8, Zeile 13) den reellwertigen Prozess `c` in einen anderen reellwertigen Prozess innerhalb derselben Währung `k` um. In jedem Zustand ist das Ergebnis gleich `c`, wenn `o` in diesem Zustand nie **`True`** ist. In den Zuständen, in denen `o` einmal **`True`** vorkommt, ist das Ergebnis gleich null.

3.3. Fazit

Bislang haben wir Peyton-Jones' Konzept zu *Composing-Contracts* und van Straatens Implementierung desselben vollständig vorgestellt. Das Konzept stellt

eine Brücke zwischen Financial Engineering (Optionsbewertung) und funktionaler Programmierung (`Haskell`) dar. Es besteht aus zwei wesentlichen Teilen: Die primitiven Konstruktoren und die Bewertungsfunktion. Durch die primitiven Konstruktoren kann man eine riesige Auswahl an finanziellen Kontrakten kombinieren. Ein Abkömmling der primitiven Konstruktoren bzw. der entsprechenden primitiven Konstruktionsfunktionen ist die erweiterbare Bibliothek von Konstruktionsfunktionen. Im Rahmen der primitiven Konstruktoren wird eine rationale Bewertungsfunktion in Kooperation mit einer vernünftigen Bewertungsstruktur erstellt.

Laut Peyton-Jones müssen die zwei Teile des Konzepts perfekt kooperieren. D. h. man soll die primitiven Konstruktoren und die Bewertungsfunktion nicht separat voneinander entwerfen, sondern miteinander koordiniert während der Entwicklungsphase.

Peyton-Jones' Konzept ist intelligent und kreativ und motivierte die Durchführung unseres eigenen Projekts. Leider kann die Implementierung gemäß van Straatens Konzept nur eine kleine Auswahl von finanziellen Kontrakten bewerten, etwa Forward-Kontrakte und Futures, obwohl sie in der Lage ist, eine große Anzahl von finanziellen Kontrakten zu beschreiben. Der Makel liegt im Design und der Implementierung des Konzeptes. So werden finanzmathematische Prinzipien (etwa die Bewertungstheorie) nur oberflächlich behandelt. Dies haben wir im Rahmen unseres Projektes verbessert.

Das Prinzip der Implementierung unseres Projekts ist jetzt klar definiert. Wir übernehmen Peyton-Jones' Grundidee und entwerfen in `Haskell` zunächst die primitiven Konstruktoren, die erweiterbare Bibliothek von Konstruktionsfunktionen, die Bewertungsstruktur und die Bewertungsfunktion, welche ein funktionales Framework bilden. Dann bringen wir die anspruchsvollen finanzmathematischen Kenntnisse (vgl. Kapitel 2) in dieses Framework ein. Der Schwerpunkt unseres Projekts besteht deshalb darin, die Entwicklung des funktionalen Frameworks und das System von Algorithmen zur Optionsbewertung im Kapitel 2 perfekt zu kombinieren.

Den Kern unseres Projekts präsentieren wir in den folgenden Kapiteln 4 (eindimensionales Projekt für Single-Asset-Optionen) und 5 (multidimensionales Projekt für Multi-Asset-Optionen).

4. Projekt für Single-Asset-Optionen

In diesem Kapitel widmen wir uns einem eindimensionalen Projekt für Single-Asset-Optionen. Dafür entwerfen wir in den folgenden Abschnitten der Reihenfolge nach die primitiven Konstruktoren, die erweiterbare Bibliothek von Konstruktionsfunktionen, die Bewertungsstruktur und die Bewertungsfunktion

4.1. Konstruktion von Single-Asset-Optionen

Das erste Ziel besteht darin, möglichst viele Optionen durch eine möglichst kleine Anzahl von Konstruktoren zu beschreiben. Wir werden im Folgenden auf die primitiven Konstruktoren unseres Konzepts eingehen.

4.1.1. Fundamentale Datentypen

Bevor wir unsere primitiven Konstruktoren einführen, wollen wir im Folgenden einige fundamentale Datentypen definieren.

```
1 data Currency = USD | GBP | EUR | CNY | HKD | JPY | AUD | CHF
2                 deriving (Eq, Show)
```

Der Datentyp `Currency` besteht aus Typkonstruktoren, die den ISO-Währungscodes entsprechen. So steht etwa `EUR` für den Euro und `USD` für den US-Dollar. Dieser Datentyp kann nach Bedarf erweitert werden.

```
1 type StartPrice = Double
```

`StartPrice` ist der Datentyp des Anfangskurses des Basiswertes einer Option.

```
1 type Rate = Double
```

`Rate` ist der Datentyp des risikolosen Zinssatzes.

```
1 type Dividend = Double
```

`Dividend` ist der Datentyp der stetigen Dividendenrendite des Basiswertes einer Option.

```
1 type Volatility = Double
```

`Volatility` ist der Datentyp der Volatilität einer Option.

```
1 type LifeTime = Double
```

`LifeTime` ist der Datentyp des Zeitintervalls von der Anfangszeit bis zur Fälligkeit einer Option.

```
type Date = Int
```

`Date` ist der Datentyp ganzzahliger Zeitschritte bzw. Zeitpunkte.

```
type Datelist = [Date]
```

Der Datentyp `Datelist` enspricht deshalb einer Liste von Zeitschritten bzw. Zeitpunkten, dessen Objekte die Indizes der Auszahlungstermine der Menge T_A enthalten.

```
type Input = (StartPrice, Rate, Dividend, Volatility, LifeTime,
              Date)
```

Der Datentyp `Input` versammelt die wichtigen Informationen einer Option. Sie sind: Anfangskurs des Basiswerts, risikoloser Zinssatz, stetige Dividendenrendite des Basiswerts, Volatilität der Option, Fälligkeit der Option und Anzahl der gesamten Zeitschritten.

```
data ExType = EBA | LBB | LBS | ASI Double
              deriving (Eq, Show)
```

Unter dem Datentyp `ExType` versteht man die Art des Basiswertes einer Option. Der Typkonstruktor `EBA` zeigt, dass diese Option auf dem normalen Kurs ihres Basiswertes basiert. `LBB` bezieht sich auf das Maximum des Kurses des Basiswertes und `LBS` bezieht sich auf das Minimum des Kurses des Basiswertes. Optionen mit dem Typkonstruktor `ASI` basieren auf dem Mittelwert des Kurses ihres Basiswertes.

```
type Rabatt  = Double

data Barrier = NoBar | Out Rabatt | In Rabatt
               deriving (Eq, Show)
```

Der Datentyp `Barrier` besteht aus drei Typkonstruktoren. Eine Option mit `NoBar` hat keine Barriere. Eine Option mit `Out rbt :: Out Rabatt` ist eine Knock-Out-Option mit der Rückvergütung `rbt` und eine Option mit `In rbt :: In Rabatt` ist eine Knock-In-Option mit der Rückvergütung `rbt`.

4.1.2. Primitive Konstruktoren

Nun gehen wir auf unser Konzept der primitiven Konstruktoren für Single-Asset-Optionen ein, welches allgemeiner und somit komplizierter als das Konzept von Peyton-Jones (siehe Programm 3.5 in Abschnitt 3.2.1) ist.

Programm 4.1: Primitive Konstruktoren für Single-Asset-Optionen

```
data Contract =
```

```
2       Zero
3     | One Currency
4     | Column Date Double
5     | Add Contract Contract
6     | Sub Contract Contract
7     | Scale Double Contract
8     | Multi Contract Contract
9     | Max Contract Contract
10    | Min Contract Contract
11    | Exp Contract
12    | Power Double Contract
13    | Ln Contract
14    | Log Double Contract
15    | Truncate Date Contract
16    | At Datelist Contract
17    | When Contract Contract
18    | If Contract Contract Contract
19    | Get Date Date ExType Barrier Contract Contract
20    | StartP Contract
21    | Underlying StartPrice
22    | MaxUnderlying StartPrice
23    | MinUnderlying StartPrice
24    | AveUnderlying StartPrice
25    | Area Date Date Contract
26    | IfB Contract Contract Contract
27    | LargerAS Contract Contract
28    | SmallerAS Contract Contract
29    | Equal Contract Contract
30    | And Contract Contract
31    | Or Contract Contract
32    | Not Contract
33    deriving (Eq, Show)
```

In unserem Datentyp `Contract` gibt es insgesamt 31 Typkonstruktoren. Diese sind 31 primitive Konstruktoren, mit denen man eine sehr große Unterklasse und auch viele neue Arten von Single-Asset-Optionen konstruieren kann. Im Gegensatz zu Peyton-Jones' Konzept ist kein Datentyp von der Art Observable in unserem Konzept enthalten. Die Funktion und Wirkung eines solchen Datentyps werden durch unseren Datentyp `Contract` abgedeckt. D. h. gerade ein Datentyp von der Art Observable ist in unserem Konzept überflüssig.

4.1.3. Bibliothek der primitiven Konstruktionsfunktionen

Nun werden wir die obigen primitiven Konstruktoren wie in Abschnitt 3.2.1 funktionalisieren, um eine Single-Asset-Option rein aus Funktionen kombinieren zu können und eine gemischte Kombination aus Funktionen und Typkonstruktoren zu vermeiden. Die Bedeutungen der primitiven Konstruktoren für Single-Asset-Optionen werden im Folgenden ausfürlich erklärt. Dabei führen wir neben dem normalen finanziellen Kontrakt noch den Hilfskontrakt ein, welcher eng mit einem normalen finanziellen Kontrakt zusammenarbeitet und bei der Entscheidung der Ausübung und Auszahlung des finanziellen Kontraktes eine große Rolle spielt. Ein Hilfskontrakt ist boolesch bewertbar und wird als Kriterium eines normalen finanziellen Kontraktes angesehen.

Im Folgenden widmen wir uns den Funktionalisierungen und Erklärungen der primitiven Konstruktoren:

```
1 zero :: Contract
2 zero = Zero
```

`zero` ist entweder ein trivialer finanzieller Kontrakt, der keine Rechte verbrieft, Geld zu bekommen, und keine Pflichten beinhaltet, Geld zu bezahlen, oder ein Hilfskontrakt, dessen Wert an jedem Zeitpunkt `True` ist.

```
1 one :: Currency -> Contract
2 one = One
```

`one k` ist ein finanzieller Kontrakt, der zu jedem Zeitschritt eine Einheit der Währung `k` (z. B. `USD`, `GBP`, `EUR` u. s. w.) auszahlt.

```
1 column :: Date -> Double -> Contract
2 column = Column
```

`column t o` ist ein finanzieller Kontrakt, der am Zeitschritt `t` den konstanten `Double`-Wert `o` auszahlt.

```
1 add :: Contract -> Contract -> Contract
2 add = Add
```

Der Inhaber des finanziellen Kontraktes `add c1 c2` erhält die Summe von zwei finanziellen Kontrakten `c1` und `c2`.

```
1 sub :: Contract -> Contract -> Contract
2 sub = Sub
```

Der Inhaber des finanziellen Kontraktes `sub c1 c2` erwirbt die Differenz zwischen zwei finanziellen Kontrakten `c1` und `c2`.

```
1 scale :: Double -> Contract -> Contract
2 scale = Scale
```

Der Wert des finanziellen Kontraktes `scale o c` ist gegeben durch das Produkt aus der reellen Zahl `o` und dem Wert des finanziellen Kontraktes `c`.

```
multi :: Contract -> Contract -> Contract
multi = Multi
```

Der Wert des finanziellen Kontraktes `multi c1 c2` entspricht der Multipikation der Werten der beiden finanziellen Kontrakten `c1` und `c2`.

```
cMax :: Contract -> Contract -> Contract
cMax = Max
```

Der finanzielle Kontrakt `cMax c1 c2` besteht aus dem Maximum von zwei finanziellen Kontrakten `c1` und `c2`.

```
cMin :: Contract -> Contract -> Contract
cMin = Min
```

Der finanzielle Kontrakt `cMin c1 c2` besteht aus dem das Minimum von zwei finanziellen Kontrakten `c1` und `c2`.

```
cTruncate :: Date -> Contract -> Contract
cTruncate = Truncate
```

`Truncate t c` ist ein abgeschnittener Teilkontrakt des finanziellen Kontraktes `c`, der nach dem Zeitschritt `t` wertlos ist.

```
at :: Datelist -> Contract -> Contract
at = At
```

`at o c` beschreibt einen finanziellen Kontrakt mit den gleichen Auszahlungen wie `c`, allerdings wird der nur zu den Zeitpunkten der Datenliste `o` ausgezahlt.

```
cWhen :: Contract -> Contract -> Contract
cWhen = When
```

Der finanzielle Kontrakt `cWhen c0 c` bedeutet, dass der finanzielle Kontrakt `c` an einem Zeitschritt ausgezahlt wird, wenn der Hilfskontrakt `c0` an demselben Zeitschritt **`True`** ist.

```
cIf :: Contract -> Contract -> Contract -> Contract
cIf = If
```

Der finanzielle Kontrakt `cIf c0 c1 c2` gleicht dem finanziellen Kontrakt `c1`, falls der Hilfskontrakt `c0` zum Anfangszeitpunkt **`True`** ist. Sonst gleicht `cIf c0 c1 c2` dem finanziellen Kontrakt `c2`.

```
get :: Date -> Date -> ExType -> Barrier -> Contract ->
       Contract -> Contract
get = Get
```

`get t1 t2 e b c0 c` beschreibt einen vollständigen finanziellen Kontrakt, der in unserem Konzept der Normalform einer Single-Asset-Option entspricht. Dabei stehen `c` für das Subjekt, `t1` und `t2` für den Anfangs- und Endzeitpunkt und `e` für die Art des Basiswertes des finanziellen Kontraktes. Ferner bezeichnet `b` den Barrieretyp des finanziellen Kontraktes und `c0` ist der Hilfskontrakt.

```
startP :: Contract -> Contract
startP = StartP
```

`StartP c` ist ein finanzieller Kontrakt, dessen Inhaber in allen Zeitschritten den Anfangswert des finanziellen Kontraktes `c` erwerben kann.

```
underlying :: StartPrice -> Contract
underlying = Underlying
```

`underlying s` ist ein finanzieller Kontrakt, dessen Auszahlung dem Kurs seines Basiswerts entspricht.

```
maxUnderlying :: StartPrice -> Contract
maxUnderlying = MaxUnderlying
```

`maxUnderlying s` ist ein finanzieller Kontrakt, dessen Auszahlung dem Maximum des Kurses seines Basiswerts entspricht.

```
minUnderlying :: StartPrice -> Contract
minUnderlying = MinUnderlying
```

`minUnderlying s` ist ein finanzieller Kontrakt, dessen Auszahlung dem Minimum des Kurses seines Basiswertes entspricht.

```
aveUnderlying :: StartPrice -> Contract
aveUnderlying = AveUnderlying
```

`aveUnderlying s` ist ein finanzieller Kontrakt, dessen Auszahlung dem Mittelwert des Kurses seines Basiswertes entspricht.

```
area :: Date -> Date -> Contract -> Contract
area = Area
```

Der Hilfskontrakt `Area t1 t2 c` ist gleich dem Hilfskontrakt `c` zwischen den Zeitschritten `t1` und `t2`. Außerhalb des Zeitintervalls von `t1` bis `t2` ist `Area t1 t2 c` überall **`True`**.

```
ifB :: Contract -> Contract -> Contract -> Contract
ifB = IfB
```

Der Hilfskontrakt `IfB c0 c1 c2` gleicht dem HilfsKontrakt `c1`, Falls der Hilfskontrakt `c0` zum Anfangszeitpunkt **`True`** ist. Sonst gleicht `If c0 c1 c2` dem Hilfskontrakt `c2`.

```
largerAS :: Contract -> Contract -> Contract
largerAS = LargerAS
```

In jedem Zeitpunkt nimmt der Hilfskontrakt `largerAS c1 c2` den Wert **`True`** an, falls in diesem Zeitpunkt der Wert des finanziellen Kontraktes `c1` größer als der Wert des finanziellen Kontraktes `c2` ist. Andernfalls hat `largerAS c1 c2` in diesem Zeitpunkt den Wert **`False`**.

```
smallerAS :: Contract -> Contract -> Contract
smallerAS = SmallerAS
```

In jedem Zeitpunkt entspricht der Hilfskontrakt `smallerAS c1 c2` dem Wert **`True`**, falls in diesem Zeitpunkt der Wert des finanziellen Kontraktes `c1` kleiner als der Wert des finanziellen Kontraktes `c2` ist. Sonst hat `smallerAS c1 c2` in diesem Zeitpunkt den Wert **`False`**.

```
equal :: Contract -> Contract -> Contract
equal = Equal
```

In jedem Zeitpunkt nimmt der Hilfskontrakt `equal c1 c2` den Wert **`True`** an, falls in diesem Zeitpunkt der Wert des finanziellen Kontraktes `c1` dem Wert des finanziellen Kontraktes `c2` gleicht. Andernfalls hat `equal c1 c2` in diesem Zeitpunkt den Wert **`False`**.

```
cAnd :: Contract -> Contract -> Contract
cAnd = And
```

In jedem Zeitpunkt ist der boolesche Wert des Hilfskontraktes `cAnd c1 c2` durch die Konjunktion der booleschen Werte der Hilfskontrakte `c1` und `c1` in diesem Zeitpunkt gegeben.

```
cOr :: Contract -> Contract -> Contract
cOr = Or
```

In jedem Zeitpunkt ist der boolesche Wert des Hilfskontraktes `cOr c1 c2` durch die Disjunktion der booleschen Werte der Hilfskontrakte `c1` und `c1` in diesem Zeitpunkt gegeben.

```
cNot :: Contract -> Contract
cNot = Not
```

In jedem Zeitpunkt ist der boolesche Wert des Hilfskontraktes `cNot c` durch die Negation des booleschen Wertes des Hilfskontraktes `c` in diesem Zeitpunkt gegeben.

4.1.4. Erweiterbare Bibliothek der Konstruktionsfunktionen

Bislang haben wir alle 31 primitiven Konstruktionsfunktionen vorgestellt. Durch die Kombinationen primitiver Konstruktionsfunktionen kann die Bibliothek der Konstruktionsfunktionen je nach Bedarf erweitert werden. Diese Erweiterung bereichert nicht nur die Bibliothek, sondern vereinfacht auch die Konstruktion finanzieller Kontrakte. Im Folgenden führen wir einige Beispiele nicht-primitiver Konstruktionsfunktionen auf.

```
give :: Contract -> Contract
give c = sub zero c
```

Die Konstruktionsfunktion `give` besteht aus den primitiven Konstruktionsfunktionen `sub` und `zero`. Der Wert des Kontraktes `give c` ist in jedem Zeitpunkt durch den negativen Wert des Kontraktes `c` gegeben.

```
correct :: Contract
correct = zero
```

Die Konstruktionsfunktion `correct` ist durch den als Hilfskontrakt aufgefassten Kontrakt `zero` gegeben und nimmt somit in jedem Zeitpunkt den booleschen Wert **`True`** an.

```
notCorrect :: Contract
notCorrect = cNot zero
```

Die Konstruktionsfunktion `notCorrect` besteht aus den primitiven Konstruktionsfunktionen `cNot` und `zero`. Sie ist ein Hilfskontrakt, dessen boolescher Wert in jedem Zeitpunkt **`False`** ist.

```
konst :: Currency -> Double -> Contract
konst k o = scale o (one k)
```

Die Konstruktionsfunktion `konst` besteht aus den primitiven Konstruktionsfunktionen `scale` und `one`. Der finanzielle Kontrakt `konst k o` zahlt in jedem Zeitschritt `o` Einheiten der Währung `k` aus.

```
konstE :: Double -> Contract
konstE o = scale o (one EUR)
```

Die Konstruktionsfunktion `konstE` ist eine Variante der Konstruktionsfunktionen `konst`. Der finanzielle Kontrakt `konstE k` zahlt in jedem Zeitschritt `o` Euro aus.

```
largerEQ :: Contract -> Contract -> Contract
largerEQ c1 c2 = cOr (largerAS c1 c2) (equal c1 c2)
```

Die Konstruktionsfunktion `largerEQ` besteht aus den primitiven Konstruktionsfunktionen `cOr`, `largerAS` und `equal`. In jedem Zeitpunkt hat der Hilfskontrakt `largerEQ c1 c2` den Wert **`True`**, falls der Wert des Kontraktes `c1` größer oder gleich dem Wert des Kontraktes `c2` zu diesem Zeitpunkt ist. Sonst ist `largerEQ c1 c2` in diesem Zeitspunkt **`False`**.

```
smallerEQ :: Contract -> Contract -> Contract
smallerEQ c1 c2 = cOr (smallerAS c1 c2) (equal c1 c2)
```

Die Konstruktionsfunktion `smallerEQ` besteht aus den primitiven Konstruktionsfunktionen `cOr`, `smallerAS` und `equal`. In jedem Zeitpunkt hat der Hilfskontrakt `smallerEQ c1 c2` den Wert **`True`**, falls der Wert des Kontraktes `c1` kleiner oder gleich der Wert des Kontraktes `c2` zu diesem Zeitpunkt ist. Sonst ist `smallerEQ c1 c2` in diesem Zeitspunkt **`False`**.

```
1 funcColumn :: Date -> Double -> (Double -> Double) -> Contract
2 funcColumn t d_t f = foldr add (column 0 (f 0.0))
3                        [column t1 (f (t2 * d_t)) |
4                        (t1,t2) <- take t $ zip [1..] [1.0..]]
```

Der finanzielle Kontrakt `funcColumn t d_t f` zahlt im Zeitschritt `n` zwischen `0` und `t` den Wert `f(n*T/t)` aus, wobei `f` eine reelle Funktion und `T` die Laufzeit des finanziellen Kontraktes ist. Diese Konstruktionsfunktion kann beispielsweise zum Bilden einer Floating-Barriere beitragen.

Natürlich können wir auch Funktionen von realen Optionen in die Bibliothek von Konstruktionsfunktionen einfügen. Beispielhaft betrachten wir die Funktionen für die Plain-Vanilla-Optionen, welche mit Hilfe der Funktion `get` in Normalform erzeugt werden:

```
1 euroCall :: StartPrice -> Strike -> Date -> Contract
2 euroCall s k t_n = get 0 t_n EBA NoBar correct $ cTruncate t_n
3                    (at [t_n] (cMax zero (sub (underlying s)
4                    (konstE k))))
```

`euroCall s k t_n` ist eine europäische Call-Option mit dem Ausübungspreis `k` und der Fälligkeit `t_n`, die mit der Anzahl der gesamten Zeitschritten gemessen wird. `s` ist der Anfangskurs ihres Basiswerts.

```
1 euroPut :: StartPrice -> Strike -> Date -> Contract
2 euroPut s k t_n = get 0 t_n EBA NoBar correct $ cTruncate t_n
3                   (at [t_n] (cMax zero (sub (konstE k)
4                   (underlying s))))
```

`euroPut s k t_n` ist eine europäische Put-Option mit dem Anfangskurs ihres Basiswerts `s`, dem Ausübungspreis `k` und der Fälligkeit `t_n`.

```
1 amerCall :: StartPrice -> Strike -> Date -> Contract
2 amerCall s k t_n = get 0 t_n EBA NoBar correct $ cTruncate t_n
3                    (at [0..t_n] (cMax zero (sub (underlying s)
4                    (konstE k))))
```

`amerCall s k t_n` ist eine amerikanische Call-Option mit dem Anfangskurs ihres Basiswerts `s`, dem Ausübungspreis `k` und der Fälligkeit `t_n`.

```
1 amerPut :: StartPrice -> Strike -> Date -> Contract
2 amerPut s k t_n = get 0 t_n EBA NoBar correct $ cTruncate t_n
3                   (at [0..t_n] (cMax zero (sub (konstE k)
4                   (underlying s))))
```

`amerPut s k t_n` ist eine amerikanische Put-Option mit dem Anfangskurs ihres Basiswerts `s`, dem Ausübungspreis `k` und der Fälligkeit `t_n`.

4.2. Bewertungsfunktion für Single-Asset-Optionen

In diesem Abschnitt diskutieren wir die Bewertungsfunktion unseres Konzepts zur Bewertung der Single-Asset-Optionen, welche eng mit den primitiven Konstruktoren aus Abschnitt 4.1 zusammenarbeiten wird. Eine wichtige Voraussetzung der Zusammenarbeit liegt an einer geeigneten Bewertungsstruktur für die Bewertungsfunktion.

4.2.1. Bewertungsstruktur und zugehörige Operatoren

Für die Implementierung des Peyton-Jones' Konzepts hat Anton van Straaten eine passende Bewertungsstruktur `PR a` entworfen, die einer zweidimensionalen Liste in `Haskell` entspricht, nämlich der Liste von Listen der Knoten eines Binomialbaumes im selben Zeitschritt (siehe Abbildung 3.2). Leider erlaubt jeder Baumknoten in van Straatens Design nur einen Wert, was für unser Konzept ungeeignet ist. In unsem Konzept kommen oft eine Reihe von Werten in einem Baumknoten vor, wenn es beispielsweise um Optionen auf Lookback-Basiswerte oder asiatische Basiswerte geht. Zweckmäßig setzen wir in jedem Knoten wiederum eine Liste, die beliebige Anzahl von Werten speichern könnte. Der genaue `Haskell`-Datentyp der Bewertungsstruktur in unserem Konzept wird wie folgt dargestellt:

```
1 newtype BTree knoten = BTree {unBT :: [TCol knoten]}
2                       deriving Show
3
4 type TCol knoten = [[knoten]]
```

Der Datentyp `TCol knoten` beschreibt die Zeitspalte eines Binomialbaum und der Datentyp `BTree knoten` entspricht daher dem ganzen Binomialbaum, welcher wie in Abbildung 4.1 aussieht. Damit die Bewertungsstruktur funktionie-

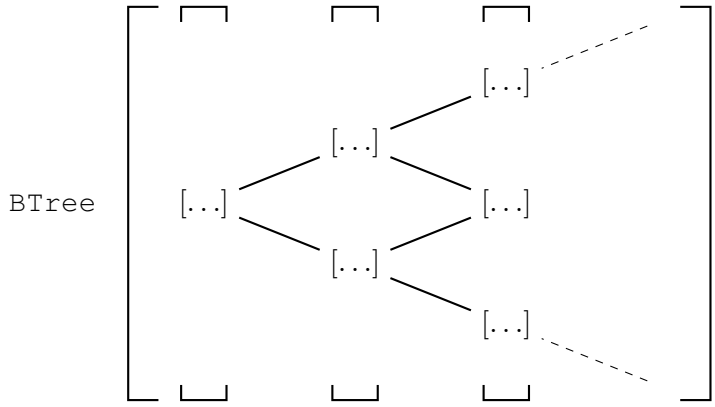


Abbildung 4.1.: Bewertungsstruktur für Single-Asset-Optionen

ren kann, müssen die zugehörigen Operatoren definiert werden. Zunächst lässt sich `BTree knoten` in eine Instanz der Typklassen **`Num`** und **`Ord`** entwickeln (vgl. Programm 4.2), sodass die fundamentalen Operationen von **`Num`** und **`Ord`** an den Datentyp `BTree knoten` vererbt werden können.

Programm 4.2: Fundamentale Operatoren auf `BTree knoten` bzgl. Typklassen `Num` und `Ord` für Single-Asset-Optionen

```
1  instance Num a => Num (BTree a) where
2     fromInteger = undefined
3     (+)         = lift2BTAll (+)
4     (-)         = lift2BTAll (-)
5     (*)         = lift2BTAll (*)
6     abs         = liftBT abs
7     signum      = liftBT signum
8
9  instance Ord a => Ord (BTree a) where
10    max = lift2BTAll max
11    min = lift2BTAll min
```

Die Funktion `liftBT f (BTree a)` nimmt zunächst die dreidimensionale Liste `a` aus dem Typkonstruktor `BTree`. Danach wird die Ein-Argument-Funktion `f` auf jedes nicht-triviale Element innerhalb der Liste a angewendet. Anschließend kapselt die Ausgabeliste wieder in den Typkonstruktor `BTree` ein. So hat etwa

```
1 liftBT abs $ BTree [[[-1]],[[],[]],[[-2],[-3,-4],[-5]]]
```

das Ergebnis

```
1 [[[1]],[[],[]],[[2],[3,4],[5]]]
```

Die Funktion `lift2BTAll f a b` wendet die Zwei-Argument-Funktion `f` auf jeden Knoten des Baumes `a` und den entsprechenden Knoten des Baumes `b` in der gleichen Position an. Wir bemerken, dass jeder Baumknoten in unserer Bewertungsstruktur einer eindimensionalen Liste entspricht. Bei der Ausführung der Funktion `lift2BTAll` gibt es in unserem Konzept zwei verschiedene Fälle:

- Der Knoten `k` des Baumes `a` und der entsprechende Knoten `k'` des Baumes `b` haben die gleiche Länge. In diesem Fall wird die Funktion `zipWith f k k'` (siehe Seite 203) aufgerufen. Beispielsweise haben wir:

  ```
  1 a = BTree [[[1]],[[2,2],[2,2]],[[1,1,1],[1,1,1],[1,1,1]]]
  2 b = BTree [[[2]],[[1,1],[1,1]],[[2,2,2],[2,2,2],[2,2,2]]]
  ```

  Nach der Ausführung des `Haskell`-Ausdruckes

  ```
  1 max $ a b
  ```

  erhalten wir das folgende Ergebnis:

```
BTree [[[2]],[[2,2],[2,2]],[[2,2,2],[2,2,2],[2,2,2]]]
```

- Einer der beiden Knoten hat nur ein Element. O. B. d. A. nehmen wir `k = [x]` an. In diesem Fall wird die Funktion `map (f x) k'` aufgerufen (siehe Seite 201). Z. B. haben wir:

```
a = BTree [[[1]],[[1],[1]],[[1],[1],[1]]]
b = BTree [[[2]],[[2,2],[2,2]],[[2,2,2],[2,2,2],[2,2,2]]]
```

  Nach der Ausführung des `Haskell`-Ausdruckes `a + b` erhalten wir das folgende Ergebnis:

```
BTree [[[3]],[[3,3],[3,3]],[[3,3,3],[3,3,3],[3,3,3]]]
```

Die Operatoren in Programm 4.2 sind lediglich die fundamentalen Operatoren auf `BTree knoten`. Diese genügen jedoch nicht für unser Konzept. Deshalb wollen wir im Folgenden mithilfe der Funktionen `liftBT` und `lift2BTAll` die Menge der zugehörigen Operatoren von `BTree knoten` erweitern, die alle wichtigen zusätzlichen Operatoren enthält. Wir fangen mit den arithmetische Operatoren an (siehe Programm 4.3).

Programm 4.3: Zusätzliche arithmetische Operatoren auf `BTree knoten` für Single-Asset-Optionen

```
expBT, logBT :: BTree Double -> BTree Double
expBT = liftBT exp
logBT = liftBT log

logbase :: Double -> BTree Double -> BTree Double
logbase a = liftBT (logBase a)

(%^) :: BTree Double -> Double -> BTree Double
(%^) c a = liftBT (\x -> x ** a) c
```

Die Funktionen `exp`, `log`, `logBase` und `(**)` sind Standardfunktionen in `Haskell` für reelle Zahlen. So ist `exp` die natürliche Exponentialfunktion und `log` gibt den natürlichen Logarithmus ihres Arguments aus. Ferner berechnet `logBase` den Logarithmus des zweiten Arguments in der Basis des ersten und `(**)` potenziert das erste Argument mit dem zweiten.

Nun definieren wir einige Vergleichsoperatoren auf `BTree knoten` in `Haskell` (siehe Programm 4.4).

Programm 4.4: Zusätzliche Vergleichsoperatoren auf `BTree knoten` für Single-Asset-Optionen

```
(%<), (%<=), (%==), (%>=), (%>) :: Ord a => BTree a -> BTree a
                                        -> BTree Bool
```

```
(%<)  = lift2BT (<)
(%<=) = lift2BT (<=)
(%==) = lift2BT (==)
(%>=) = lift2BT (>=)
(%>)  = lift2BT (>)
```

In `Haskell` sind `(<)`, `(<=)`, `(==)`, `(>=)` und `(>)` die Standardvergleichsoperatoren.

Zum Schluss werden noch ein paar logische Operatoren auf `BTree knoten` in `Haskell` definiert (siehe Programm 4.5).

Programm 4.5: Zusätzliche logische Operatoren auf `BTree knoten` für Single-Asset-Optionen

```
(%&&), (%||) :: BTree Bool -> BTree Bool -> BTree Bool
(%&&) = lift2BTAll (&&)
(%||) = lift2BTAll (||)

notBT :: BTree Bool -> BTree Bool
notBT = liftBT not
```

Die Standardlogikfunktionen in `Haskell` sind `(&&)`, `(||)` und `not`, wobei `(&&)` die Konjunktion, `(||)` die Disjunktion und `not` die Negation darstellt.

4.2.2. Implementierung der Bewertungsfunktion

In diesem Unterabschnitt gehen wir auf die Bewertungsfunktion für Single-Asset-Optionen ein, die eng mit den primitiven Konstruktoren und der Bewertungsstruktur zusammenarbeiten muss. Die Bewertungsfunktion besteht aus zwei Teilen. Der erste ist die Bewertungsfunktion `eval` mit einem normalen finanziellen Kontrakt als Argument, die einen reellen Binomialbaum ausgibt (siehe Programm 4.6).

Programm 4.6: Bewertungsfunktion bzgl. eines reellen Binomialbaumes für Single-Asset-Optionen

```
eval :: Contract -> BTree Double
eval Zero                 = bigK 0.0
eval (One k)              = exch EUR k
eval (Column t o)         = makeCol t o
eval (c1 `Add` c2)        = (eval c1) + (eval c2)
eval (c1 `Sub` c2)        = (eval c1) - (eval c2)
eval (o `Scale` c)        = (bigK o) * (eval c)
eval (c1 `Multi` c2)      = (eval c1) * (eval c2)
eval (c1 `Max` c2)        = max (eval c1) (eval c2)
eval (c1 `Min` c2)        = min (eval c1) (eval c2)
eval (Exp c)              = expBT (eval c)
```

```
12 eval (Power o c)           = (eval c) %^ o
13 eval (Ln c)                = logBT (eval c)
14 eval (Log o c)             = logbase o (eval c)
15 eval (Truncate t c)        = takeBT (t+1) (eval c)
16 eval (At o c)              = possiblePay o (eval c)
17 eval (When c0 c)           = payment (evalB c0) (eval c)
18 eval (If c0 c1 c2)         = if (evalBool c0)
19                              then (eval c1) else (eval c2)
20 eval (Get 0 t e b c0 c)    = discount t e b (evalB c0) (eval c)
21 eval (Get t1 t2 e b c0 c) = backward [[evalC (Get 0 (t2-t1) e b
22                              (substitute t1 ((preisList t1)!!x)
23                              c0) (substitute t1 ((preisList t1)
24                              !!x) c))] | x <- [0..t1]]
25 eval (StartP c)            = bigK $ evalC c
26 eval (Underlying s)        = underlyingTree s
27 eval (MaxUnderlying s)     = underlyingMaxT s
28 eval (MinUnderlying s)     = underlyingMinT s
29 eval (AveUnderlying s)     = underlyingAveT s
```

Dabei konstruieren die primitiven Funktionen in Zeilen 26-29 die verschiedenen Typen von S-Bäumen einer Single-Asset-Option. Die primitiven Funktionen in Zeile 20 und Zeile 21 berechnen den V-Baum und die anderen primitiven Funktionen tragen zu dem C-Baum bei.

Der zweite Teil ist die Bewertungsfunktion `evalB` mit einem Hilfskontrakt als Argument, die einen booleschen Binomialbaum zurückgibt (siehe Programm 4.7).

Programm 4.7: Bewertungsfunktion bzgl. eines booleschen Binomialbaumes für Single-Asset-Optionen

```
1 evalB :: Contract -> BTree Bool
2 evalB Zero                  = bigK True
3 evalB (Area t1 t2 c)        = makeArea t1 t2 (evalB c)
4 evalB (IfB c0 c1 c2)        = if (evalBool c0)
5                               then (evalB c1) else (evalB c2)
6 evalB (c1 `LargerAS` c2)    = (eval c1) %> (eval c2)
7 evalB (c1 `SmallerAS` c2)   = (eval c1) %< (eval c2)
8 evalB (c1 `Equal` c2)       = (eval c1) %== (eval c2)
9 evalB (c1 `And` c2)         = (evalB c1) %&& (evalB c2)
10 evalB (c1 `Or` c2)         = (evalB c1) %|| (evalB c2)
11 evalB (Not c)              = notBT (evalB c)
12 evalB (Truncate t c)       = takeBT (t+1) (evalB c)
```

Wenn wir den V-Baum einer Instanz `c` vom Datentyp `Contract` bewerten möchten, führen wir einfach den Befehl `eval c` aus. Dabei werden einige Teile von `c` automatisch durch die Funktion `evalB` bewertet. Somit fassen wir die Bewer-

tungsfunktion `evalB` als eine Begleitfunktion der Bewertungsfunktion `eval` auf. Beim Pattern-Matching von `eval` und `evalB` genügt es wie in der Diskussion des Peyton-Jones' Konzeptes (siehe Unterabschnitt 3.2.2), die primitiven Funktionen aller primitiven Konstruktoren von `Contract` zu implementieren. Die Bedeutungen aller primitiven Funktionen in `eval` und `evalB` werden im Folgenden der Reihe nach genau erklärt:

- **Programm 4.6, Zeile 2 und Programm 4.7, Zeile 2:** Gegeben sei ein reeller oder boolescher Wert `a`. Die primitive Funktion `bigK a` stellt einen reellen oder booleschen Binomialbaum dar, dessen Knoten alle gleich dem Wert `a` sind. Deshalb ist `eval Zero` gleich

```
BTree [[[0.0]],[[0.0],[0.0]],[[0.0],[0.0],[0.0]]..]
```

  Und das Ergebnis von `evalB Zero` lautet:

```
BTree [[[True]],[[True],[True]],[[True],[True],[True]]..]
```

- **Programm 4.6, Zeile 3:** Die primitive Funktion `exch k k2` repräsentiert einen in Währung `k2` ausgedrückten reellen Binomialbaum durch einen in Währung `k` ausgedrückten reellen Binomialbaum unter Verwendung eines vordefinierten Wechselkurses zwischen den Währungen `k` und `k2`. Z. B. liefert `exch EUR USD` das Ergebnis:

```
BTree [[[0.68]],[[0.68],[0.68]],[[0.68],[0.68],[0.68]]..]
```

- **Programm 4.6, Zeile 4:** Die primitive Funktion `makeCol t o` erstellt einen Binomialbaum, dessen Knoten im Zeitschritt `t` alle gleich `[o]` sind, während die anderen Knoten alle leere Listen sind. So hat etwa `makeCol 1 2` das Ergebnis:

```
BTree [[[]],[[2],[2]],[[],[],[]]..]
```

- **Programm 4.6, Zeile 5-14:** Die primitiven Funktionen `(+)`, `(-)`, `(*)`, `max`, `min`, `expBT`, `(%^)`, `logBT` und `logbase`, die die zugehörigen Operatoren auf dem Datentyp `BTree Double` sind, wurden bereits in den Programmen 4.2 und 4.3 genau erklärt.

- **Programm 4.6, Zeile 15 und Programm 4.7, Zeile 12:** Die primitive Funktion `takeBT t b` nimmt nur die ersten `t` Spalten des Binomialbaumes `b`. Z. B. ist `takeBT 3 (bigK 1)` gleich

```
BTree [[[1]],[[1],[1]],[[1],[1],[1]]]
```

  und `takeBT 2 (bigK False)` ist gleich:

```
BTree [[[False]],[[False],[False]]]
```

- **Programm 4.6, Zeile 16:** Durch die primitive Funktion `possiblePay o b` behält sich der Binomialbaum `b` alle Knoten in den Zeitspalten vor, die der Datenliste `o` entsprechen. Das Ergebnis von `possiblePay [0,2] (bigK 1)` ist:

```
BTree [[[1]],[[],[]],[[1],[1],[1]],[[],[],[],[]]..]
```

- **Programm 4.6, Zeile 17:** Durch die primitive Funktion `payment b1 b2` wandelt sich der reelle Binomialbaum `b2` gemäß der folgenden Regel um. Alle Knoten von `b2`, die leere Listen sind, bleiben unverändert. Ein nichtleerer Knoten von `b2` bleibt ebenfalls derselbe, falls der entsprechende Knoten von `b1` gleich `[True]` ist. Andernfalls wird sich dieser Knoten in eine Liste von Nullen mit derselben Länge umwandelt. Z. B. haben wir:

```
b1 = BTree [[[False]],[[True],[False]],[[True],[False],
           [False]]]
b2 = BTree [[[]],[[1],[1]],[[2],[2,2],[2]]]
```

  Die Ausgabe von `payment b1 b2` ist:

```
BTree [[[]],[[1],[0]],[[2],[0,0],[0]]]
```

- **Programm 4.6, Zeile 18 und Programm 4.7, Zeile 4:** Die Funktion `evalBool c` gibt den booleschen Wert aus dem Anfangsknoten des booleschen Binomialbaumes `evalB c` aus.

- **Programm 4.6, Zeile 20:** Die primitive Funktion `discount` ist die Kernfunktion unseres Konzepts, die durch den Schlüsselkontrakt `Get` mit dem Anfangszeitpunkt null ausgelöst wird:

```
discount :: Date -> ExType -> Barrier -> BTree Bool ->
            BTree Double -> BTree Double
discount t e b (BTree c0) (BTree c) =
 case (e,b) of
   (EBA, NoBar)      -> BTree $ discountEBA t c
   (LBB, NoBar)      -> BTree $ discountLBB t c
   (LBS, NoBar)      -> BTree $ discountLBS t c
   (ASI s, NoBar)    -> BTree $ discountASI t s c

   (EBA, Out rbt)    -> BTree $ discountEBAOut t rbt c0 c
   (LBB, Out rbt)    -> BTree $ discountLBBOut t rbt c0 c
   (LBS, Out rbt)    -> BTree $ discountLBSOut t rbt c0 c
   (ASI s, Out rbt) -> BTree $ discountASIOut t rbt s c0 c

   (EBA, In rbt)     -> BTree $ discountEBAIn (t,t) rbt
                                (discountEBA t c) c0 c
```

```
17   (LBB, In rbt)     -> BTree $ discountLBBIn (t,t) rbt
18                                  (discountLBB t c) c0 c
19   (LBS, In rbt)     -> BTree $ discountLBSIn (t,t) rbt
20                                  (discountLBS t c) c0 c
21   (ASI s, In rbt)   -> BTree $ discountASIIn (t,t) rbt s
22                                  (discountASI t s c) c0 c
```

wobei `rbt` die Rückvergütung einer Barriere-Option bezeichnet und die primitive Funktion `discount` rückwärts vom Zeitschritt `t` bis zum Zeitschritt null durchgeführt wird. Die primitive Funktion `discount` unterteilt sich in zwölf spezielle Fälle, die den Kombinationen von vier Typen des Basiswertes aus dem Datentyp `ExType` und drei Arten der Barriere aus dem Datentyp `Barrier` entsprechen. Die vier Typen des Basiswertes sind der normale Basiswert (`EBA`), der maximale Basiswert (`LBB`), der minimale Basiswert (`LBS`) und der durchschnittliche Basiswert (`ASI s`). Die drei Arten der Barriere sind erstens "ohne Barriere" (`NoBar`), zweitens "mit Knock-Out-Barriere" (`Out rbt`), drittens "mit Knock-In-Barriere" (`In rbt`).

Die ersten vier Fälle ohne Barriere werden als Spezialfa"lle der Variante mit Knock-Out-Barriere aufgefasst:

```
1  discountEBA :: Date -> [TCol Double] -> [TCol Double]
2  discountEBA t ps = discountEBAOut t 0 (konstSlices True) ps
3
4  discountLBB :: Date -> [TCol Double] -> [TCol Double]
5  discountLBB t ps = discountLBBOut t 0 (konstSlices True) ps
6
7  discountLBS :: Date -> [TCol Double] -> [TCol Double]
8  discountLBS t ps = discountLBSOut t 0 (konstSlices True) ps
9
10 discountASI :: Date -> Double  -> [TCol Double] ->
11                [TCol Double]
12 discountASI t s ps = discountASIOut t 0 s (konstSlices True)
13                     ps
```

Dabei ergibt sich aus `konstSlices True` ein Binomialbaum ohne den Typkonstruktor `BTree`, dessen Knoten alle gleich `[True]` sind. Infolgedessen werden wir uns nur auf die anderen acht Fälle konzentrieren, die vier Fälle mit Knock-Out-Barriere und die vier Fälle mit Knock-In-Barriere:

- Die Funktion `discountEBAOut` berechnet gemäß Algorithmus 17 aus Anhang B gemäß den V-Baum einer Single-Asset-Option auf den normalen Basiswert mit der Knock-Out-Barriere, die durch `(EBA, Out rbt)` gekennzeichnet wird.
- Die Funktion `discountLBBOut` berechnet gemäß Algorithmus 19 aus Anhang B gemäß den V-Baum einer Single-Asset-Option auf den ma-

ximalen Basiswert mit der Knock-Out-Barriere, die durch `(LBB, Out rbt)` gekennzeichnet wird.

- Die Funktion `discountLBSOut` berechnet gemäß Algorithmus 21 aus Anhang B gemäß den V-Baum einer Single-Asset-Option auf den minimalen Basiswert mit der Knock-Out-Barriere, die durch `(LBS, Out rbt)` gekennzeichnet wird.
- Die Funktion `discountASIOut` berechnet gemäß Algorithmus 23 aus Anhang B gemäß den V-Baum einer Single-Asset-Option auf den durchschnittlichen Basiswert mit der Knock-Out-Barriere, die durch `(ASI s, Out rbt)` gekennzeichnet wird.
- Die Funktion `discountEBAIn` bewertet gemäß Algorithmus 18 aus Anhang B gemäß den V-Baum einer Single-Asset-Option auf den normalen Basiswert mit der Knock-In-Barriere, die durch `(EBA, In rbt)` gekennzeichnet wird.
- Die Funktion `discountLBBIn` bewertet gemäß Algorithmus 20 aus Anhang B gemäß den V-Baum einer Single-Asset-Option auf den maximalen Basiswert mit der Knock-In-Barriere, die durch `(LBB, In rbt)` gekennzeichnet wird.
- Die Funktion `discountLBSIn` bewertet gemäß Algorithmus 22 aus Anhang B gemäß den V-Baum einer Single-Asset-Option auf den minimalen Basiswert mit der Knock-In-Barriere, die durch `(LBS, In rbt)` gekennzeichnet wird.
- Die Funktion `discountASIIn` bewertet gemäß Algorithmus 24 aus Anhang B gemäß den V-Baum einer Single-Asset-Option auf den durchschnittlichen Basiswert mit der Knock-In-Barriere, die durch `(ASI s, In rbt)` gekennzeichnet wird.

- **Programm 4.6, Zeile 21:** Die Signatur der primitiven Funktion `backward` lautet:

```
1 backward :: TCol Double -> BTree Double
```

wobei `backward l` alle Knoten des Zeitpunkts `l` gemäß Formel (2.1) schrittweise auf den Zeitpunkt null diskontiert. Dabei bilden alle ausgerechneten Knoten zusammen mit dem Typkonstruktor `BTree` wieder einen Binomialbaum. Die Funktion `evalC c` gibt den reellen Wert aus dem Anfangsknoten des reellen Binomialbaumes `eval c` aus. Der Kontrakt `substitute t s1 c` verschiebt den Kontrakt `c` vom Zeitpunkt null zum Zeitpunkt `t`, wobei `s1` den Anfangswert des Basiswertes von `substitute t s1 c` bezeichnet.

- **Programm 4.6, Zeile 25:** Die primitive Funktion `bigK` und die Funktion `evalC` wurden bereits auf Seite 77 und 80 vorgestellt.

- **Programm 4.6, Zeile 26:** Die primitive Funktion `underlyingTree s` entwickelt gemäß Algorithmus 3 den S-Baum einer Single-Asset-Option im CRR-Modell mit dem Anfangskurs `s`, also den Binomialbaum des normalen Basiswerts.
- **Programm 4.6, Zeile 27:** Die primitive Funktion `underlyingMaxT s` entwickelt gemäß Algorithmus 10 den S^{max}-Baum einer Single-Asset-Option im CRR-Modell mit dem Anfangskurs `s`, also den Binomialbaum des maximalen Basiswerts.
- **Programm 4.6, Zeile 28:** Die primitive Funktion `underlyingMinT s` entwickelt gemäß Algorithmus 11 den S^{min}-Baum einer Single-Asset-Option im CRR-Modell mit dem Anfangskurs `s`, also den Binomialbaum des minimalen Basiswerts.
- **Programm 4.6, Zeile 29:** Die primitive Funktion `underlyingAveT s` entwickelt gemäß Algorithmus 12 den S^{ave}-Baum einer Single-Asset-Option im CRR-Modell mit dem Anfangskurs `s`, also den Binomialbaum des durchschnittlichen Basiswerts.
- **Programm 4.7, Zeile 3:** Durch die primitive Funktion `makeArea t1 t2 b` werden alle Knoten des booleschen Binomialbaumes `b` zwischen den Zeitpunkten `t1` und `t2` unverändert gelassen. Die anderen Knoten von `b` sind alle gleich `[True]`.
- **Programm 4.7, Zeile 6-8:** Die primitiven Funktionen `(%>)`, `(%<)` und `(%==)`, die die Vergleichsoperatoren mit dem Ausgabedatentyp `BTree Bool` sind, haben wir bereits in Programm 4.4 genau erläutert.
- **Programm 4.7, Zeile 9-11:** Die primitiven Funktionen `(%&&)`, `(%||)` und `notBT`, die die logischen Operatoren auf dem Datentyp `BTree Bool` sind, wurden in Programm 4.5 genau erklärt.

Wie bereits erwähnt, kann die Bewertungsfunktion `eval` den V-Baum eines finanziellen Kontraktes `c :: Contract` berechnen, wobei der Anfangsknoten des V-Baumes zum Zeitpunkt null dem fairen Preis von `c` entspricht. Somit können wir durch die folgende `Haskell`-Funktion den fairen Preis von `c` unmittelbar bewerten:

```
1 evalCon :: Input -> Contract -> Double
2 evalCon (startPrice, rate, dividend, volatility, lifeTime, t_N)
3         = head . head . head . unBT . eval
```

Dabei beinhaltet der Datentyp `Input` (siehe Seite 64) alle wichtige Informationen einer Single-Asset-Option, etwa den Anfangskurs des Basiswertes (`startPrice`), den risikolosen Zinssatz (`rate`), die stetige Dividendenrendite des Basiswertes (`dividend`), die Volatilität der Option (`volatility`), die Fälligkeit der Option (`lifeTime`) sowie die Anzahl der gesamten Zeitschritte (`t_N`).

4.3. Numerische Beispiele

In diesem Abschnitt wollen wir unseren eindimensionalen Algorithmus für die Single-Asset-Optionen überprüfen und die dabei gesammelten numerischen Erfahrungen darstellen. Hierbei sehen wir von der Betrachtung der Effizienz des Algorithmus ab, da eine schnelle Bewertung mit dem allgemeinen Algorithmus nicht möglich ist. Im Folgenden legen wir den Schwerpunkt auf die Bewertungsfähigkeit des Algorithmus den Umfang seiner Anwendung.

Ein vollständiges numerisches Beispiel besteht hier aus zwei Teilen: Die Konstruktion der betreffenden Single-Asset-Option in `Haskell` und das Bewertungsergebnis. Wir wollen zuerst anhand eines einfachen Beispiels die Vorgehensweise der Konstruktion einer Option und den Bewertungsvorgang schrittweise demonstrieren.

4.3.1. Demonstration

Das ausgewählte Beispiel zur Demonstration ist ein bermudischer Down-and-Out-Call mit der Auszahlungsfunktion zum Ausübungszeitpunkt t:

$$(S(t) - 50)^+ \cdot \mathbf{1}\left\{\min_{t\in[0,T]} S(t) > 44.55\right\}. \tag{4.1}$$

Der Anfangskurs des Basiswertes $S(0)$ ist 50 € und seine Volatilität ist 0.4. Der Basiswert hat keine Dividendenrendite, der risikolose Zinssatz liegt bei 0.1 und die Laufzeit bzw. Fälligkeit der Option beträgt fünf Monate. Die Auszahlungstermine liegen an den Enden des ersten, dritten und fünften Monats. Des weiteren ist die Anzahl der gesamten Zeitschritte der Einfachheit halber auf 5 gesetzt. In der folgenden Tabelle listen wir die Eingabedaten gemäß unseren Notationen auf:

$S(0)$	r	g	σ	T	N	T_A
50 €	0.1	0	0.4	$5/12$	5	$\{t_1, t_3, t_5\}$

In unserem `Haskell`-Programm fassen wir die wichtigen Eingabedaten der Option in einem Objekt des Datentyps `Input` zusammen:

```
1 input = (50, 0.08, 0.03, 0.2, 0.5, 1000) :: Input
```

Nun konstruieren wir Schritt für Schritt die Option mittels der Konstruktionsfunktionen aus unserem eindimensionalen Konzept.

- Konstruktion der Auszahlung

$$(S(t) - 50)^+ : \tag{4.2}$$

  - Die Darstellung des Ausübungspreises 50 € in `Haskell` lautet:

```
scale 50 (one EUR)
```

– Die `Haskell`-Form für den Basiswert $S(t)$ mit Anfangskurs 50 € ist:

```
underlying 50
```

– Die `Haskell`-Form für die Subtraktion innerhalb der beiden Klammern der Formel (4.2) ist:

```
sub (underlying 50) (scale 50 (one EUR))
```

– Somit lautet die `Haskell`-Darstellung der Formel (4.2):

```
cMax zero (sub (underlying 50) (scale 50 (one EUR)))
```

– Falls wir die Menge der Auszahlungstermine T_A berücksichtigen, erhalten wir die `Haskell`-Form:

```
at [1,3,5] (cMax zero (sub (underlying 50) (scale 50
           (one EUR))))
```

– Ferner können wir die `Haskell`-Form der Formel (4.2) erweitern:

```
payment = cTruncate 5 (at [1,3,5] (cMax zero (sub
          (underlying 50) (scale 50 (one EUR)))))
```

Hierdurch kann die Fälligkeit der Option klar aufgezeigt werden.

- Konstruktion des Kriteriums der Knock-Out-Barriere

$$\min_{t\in[0,T]} S(t) > 44.55 : \tag{4.3}$$

– Die Darstellung der Barriere 44.55 € in `Haskell` lautet:

```
scale 44.55 (one EUR)
```

– Die `Haskell`-Form für den Basiswert $S(t)$ mit Anfangskurs 50 € ist:

```
underlying 50
```

– Das Kriterium (4.3) bzw. seine äquivalente Form

$$\forall t \in [0,T] : S(t) > 44.55$$

ist daher in `Haskell` wie folgt dargestellt:

```
largerAS (underlying 50) (scale 44.55 (one EUR))
```

– Durch die Konstruktionsfunktion `cTruncate` beschränkt sich die Beurteilung des Barriere-Kriteriums nur auf die Fälligkeit der Option:

```
barriereOut = cTruncate 5 (largerAS (underlying 50)
                (scale 44.55 (one EUR)))
```

- Einsetzen von Auszahlung und Knock-Out-Barriere in die Normalform der Option:

  Unter Verwendung der Konstruktionsfunktion `get` (siehe Seite 67) erhalten wir die Option `beispielOption` als:

```
beispielOption = get 0 5 EBA (Out 0) barriereOut
                 payment
```

  oder ausführlich:

```
beispielOption = get 0 5 EBA (Out 0) (cTruncate 5
                 (largerAS (underlying 50) (scale 44.55
                 (one EUR)))) (cTruncate 5 (at [1,3,5]
                 (cMax zero (sub (underlying 50) (scale 50
                 (one EUR))))))
```

Abbildung C.1 aus Anhang C wandelt die obige Normalform von `beispielOption` in eine übersichtlich zerlegte Baumdarstellung um. Im Folgenden wollen wir anhand dieser zerlegten Baumdarstellung und einer Reihe von Abbildungen aus Anhang C den Bewertungsvorgang des `Haskell`-Befehls

```
eval beispielOption
```

Schritt für Schritt erklären.

- Abbildung C.2 zeigt den Bewertungsbaum von `eval $ one EUR` bzgl. der Auszahlung mittels der Bewertungsform `exch EUR EUR`.
- In Abbildung C.3 ist der Bewertungsbaum von `eval $ scale 50 (one EUR)` bzgl. der Auszahlung mittels der Bewertungsform `(bigK 50) * (exch EUR EUR)` dargestellt.
- Abbildung C.4 zeigt den Bewertungsbaum von `eval $ underlying 50` bzgl. der Auszahlung mittels der Bewertungsform `underlyingTree 50`, der dem S-Baum der Option entspricht.
- Abbildung C.5 zeigt den Bewertungsbaum von

```
eval $ sub (underlying 50) (scale 50 (one EUR))
```

  bzgl. der Auszahlung mittels der Bewertungsform

```
(underlyingTree 50) - ((bigK 50) * (exch EUR EUR))
```

- Abbildung C.6 zeigt den Bewertungsbaum von `eval zero` bzgl. der Auszahlung mittels der Bewertungsform `bigK 0.0`.

- In Abbildung C.7 ist der Bewertungsbaum von

```
eval $ cMax zero (sub (underlying 50) (scale 50 (one EUR)))
```

  bzgl. der Auszahlung mittels der Bewertungsform

```
max (bigK 0.0) $
    (underlyingTree 50) - ((bigK 50) * (exch EUR EUR))
```

  dargestellt.

- Abbildung C.8 zeigt den Bewertungsbaum von

```
eval $ at [1,3,5] (cMax zero (sub (underlying 50)
                  (scale 50 (one EUR))))
```

  bzgl. der Auszahlung mittels der Bewertungsform

```
possiblePay [1,3,5] $ max (bigK 0.0) ((underlyingTree 50) -
                      ((bigK 50) * (exch EUR EUR)))
```

- In Abbildung C.9 ist der Bewertungsbaum von

```
eval $ cTruncate 5 (at [1,3,5] (cMax zero (sub
       (underlying 50) (scale 50 (one EUR)))))
```

  bzgl. der Auszahlung mittels der Bewertungsform

```
takeBT 6 $ possiblePay [1,3,5] (max (bigK 0.0)
((underlyingTree 50) - ((bigK 50) * (exch EUR EUR))))
```

  dargestellt. Dieser Binomialbaum entspricht dem C-Baum der Option.

- Abbildung C.10 zeigt den Bewertungsbaum von `eval $ one EUR` bzgl. des Barriere-Kriteriums mittels der Bewertungsform `exch EUR EUR`.

- In Abbildung C.11 ist der Bewertungsbaum von `eval $ scale 44.55 (one EUR)` bzgl. des Barriere-Kriteriums mittels der Bewertungsform `(bigK 44.55) * (exch EUR EUR)` dargestellt.

- Abbildung C.12 zeigt nochmals den Bewertungsbaum von `eval $ underlying 50` bzgl. des Barriere-Kriteriums mittels der Bewertungsform `underlyingTree 50`, der dem S-Baum der Option entspricht.

- In Abbildung C.13 ist der boolesche Bewertungsbaum von

```
evalB $ largerAS (underlying 50) (scale 44.55 (one EUR))
```

  bzgl. des Barriere-Kriteriums mittels der Bewertungsform

```
(underlyingTree 50) %> ((bigK 44.55) * (exch EUR EUR))
```

dargestellt.

- Abbildung C.14 zeigt den booleschen Bewertungsbaum von

```
evalB $ cTruncate 5 (largerAS (underlying 50)
                    (scale 44.55 (one EUR)))
```

bzgl. des Barriere-Kriteriums mittels der Bewertungsform

```
takeBT 6 $ (underlyingTree 50) %>
           ((bigK 44.55) * (exch EUR EUR))
```

- Die letzte Abbildung, Abbildung C.15, zeigt den Bewertungsbaum von `eval beispielOption` mittels der Bewertungsform

```
discount 5 EBA (Out 0) (takeBT 6 $ (underlyingTree 50) %>
((bigK 44.55) * (exch EUR EUR))) (takeBT 6 $ possiblePay
[1,3,5] (max (bigK 0.0) ((underlyingTree 50) - ((bigK 50) *
(exch EUR EUR)))))
```

Dieser Binomialbaum entspricht dem V-Baum der Option.

Weil der Wert am Anfangsknoten des V-Baumes genau dem durch das Binomialmodell simulierten Preis der Option entspricht, können wir diesen direkt durch die `Haskell`-Funktion `evalCon input beispielOption` ermitteln.

4.3.2. Bewertungsbeispiele

In diesem Unterabschnitt werden wir eine Reihe von unterschiedlichen Single-Asset-Optionen durch unseren eindimensionalen Algorithmus bewerten. Da diese Optionen zu vielen verschiedenen Klassen gehören und daher vielfältige Bewertungsmöglichkeiten haben, ist es sehr aufwendig, die Referenzwerte für die numerischen Ergebnisse aller ausgewählten Optionen selbständig zu bestimmen. Um dieses Verfahren zu vereinfachen haben wir solche Single-Asset-Optionen ausgewählt, die bereits in einigen wissenschaftlichen Artikeln zur Optionsbewertung bewertet wurden. Dabei verwenden wir die in diesen Artikeln ermittelten Optionswerte als Referenzwerte für die numerischen Ergebnisse unseres eigenen Algorithmus. Diese wissenschaftlichen Artikel sind [1], [14], [34] und [35]. Einige der Referenzwerte sind analytische Lösungen, d. h. sie ergeben sich direkt aus der Black-Scholes-Formel. Die übrigen werden durch passende numerische Methoden berechnet.

Beispiel I: Europäische Call-Option

Die Auszahlungsfunktion der Single-Asset-Option zur Zeit T ist:

$$(S(T) - 105)^+$$

mit den Parameter:

$S(0)$	r	g	σ	T	N	T_A
100 €	0.2	0	0.3	0.5	1000	$\{t_{1000}\}$

Der entsprechende `Haskell`-Code ist:

```
input :: Input
input = (100, 0.2, 0, 0.3, 0.5, 1000)
```

Die Konstruktion der Single-Asset-Option in `Haskell` lautet:

```
contract :: Contract
contract = euroCall 100 105 1000
```

wobei es sich bei `euroCall` um eine Konstruktionsfunktion aus der erweiterbaren Bibliothek handelt (siehe Unterabschnitt 4.1.4).

Das Ergebnis des `Haskell`-Befehls `evalCon input contract` ist 10.97 €, währewnd der entsprechende Referenzwert 10.89 € beträgt. Dieses Beispiel und der Referenzwert stammen aus dem Buch [35] von Zhang.

Beispiel II: Europäische Put-Option

Die Auszahlungsfunktion der Single-Asset-Option zur Zeit T ist:

$$(105 - S(T))^+$$

mit den Parameter:

$S(0)$	r	g	σ	T	N	T_A
100 €	0.2	0	0.3	0.5	1000	$\{t_{1000}\}$

Der entsprechende `Haskell`-Code ist:

```
input :: Input
input = (100, 0.2, 0, 0.3, 0.5, 1000)
```

Die Konstruktion der Single-Asset-Option in `Haskell` lautet:

```
contract :: Contract
contract = euroPut 100 105 1000
```

wobei `euroPut` eine Konstruktionsfunktion aus der erweiterbaren Bibliothek ist (siehe Unterabschnitt 4.1.4).

Das Ergebnis des `Haskell`-Befehls `evalCon input contract` ist 5.979 € und der entsprechende Referenzwert ist 5.898 €. Das Beispiel und der Referenzwert stammen aus dem Buch [35] von Zhang.

Beispiel III: Amerikanische Call-Option

Die Auszahlungsfunktion der Single-Asset-Option zur Zeit $t \in [0, T]$ ist:

$$(S(t) - 105)^+$$

mit den Parameter:

$S(0)$	r	g	σ	T	N	T_A
100 €	0.08	0.12	0.2	1	800	$\{t_0, \ldots, t_{800}\}$

Der entsprechende `Haskell`-Code ist:

```
input :: Input
input = (100, 0.08, 0.12, 0.2, 1, 800)
```

Die Konstruktion der Single-Asset-Option in `Haskell` lautet:

```
contract :: Contract
contract = amerCall 100 100 800
```

wobei es sich bei `amerCall` um eine Konstruktionsfunktion aus der erweiterbaren Bibliothek handelt (siehe Unterabschnitt 4.1.4).

Das Ergebnis des `Haskell`-Befehls `evalCon input contract` ist 6.1211 € und der entsprechende Referenzwert beträgt 6.1750 €. Das Beispiel und der Referenzwert stammen aus dem Buch [35] von Zhang.

Beispiel IV: Amerikanische Put-Option

Die Auszahlungsfunktion der Single-Asset-Option zur Zeit $t \in [0, T]$ ist:

$$(105 - S(t))^+$$

mit den Parameter:

$S(0)$	r	g	σ	T	N	T_A
100 €	0.1	0	0.2	1/3	4	$\{t_0, \ldots, t_4\}$

Der entsprechende `Haskell`-Code ist:

```
input :: Input
input = (100, 0.1, 0, 0.2, 1/3, 4)
```

Die Konstruktion der Single-Asset-Option in `Haskell` lautet:

```
contract :: Contract
contract = amerPut 100 100 4
```

wobei `amerPut` eine Konstruktionsfunktion aus der erweiterbaren Bibliothek ist (siehe Unterabschnitt 4.1.4).

Das Ergebnis des `Haskell`-Befehls `evalCon input contract` ist 3.288 € und der entsprechende Referenzwert liegt bei 3.29 €. Das Beispiel und der Referenzwert stammen aus dem Buch [35] von Zhang.

Beispiel V: Europäische Asiatische Call-Option

Die Auszahlungsfunktion der Single-Asset-Option zur Zeit T ist:

$$\left(\frac{1}{T}\int_0^T S(t)dt - 50\right)^+$$

mit den Parameter:

$S(0)$	r	g	σ	T	N	T_A
50 €	0.1	0	0.4	1	60	$\{t_{60}\}$

Der entsprechende `Haskell`-Code ist:

```
input :: Input
input = (50, 0.1, 0, 0.4, 1, 60)
```

Die Konstruktion der Single-Asset-Option in `Haskell` lautet:

```
contract :: Contract
contract = get 0 60 (ASI 50) NoBar correct $ cTruncate 60
           (at [60] (cMax zero (sub (aveUnderlying 50)
           (konstE 50))))
```

wobei `correct` und `konstE` Konstruktionsfunktionen aus der erweiterbaren Bibliothek sind (siehe Unterabschnitt 4.1.4).

Das Ergebnis des `Haskell`-Befehls `evalCon input contract` ist 5.59 € und der entsprechende Referenzwert ist 5.62 €. Das Beispiel und der Referenzwert stammen aus dem Buch [14] von Hull.

Beispiel VI: Amerikanische Asiatische Call-Option

Die Auszahlungsfunktion der Single-Asset-Option zur Zeit $t \in [0, T]$ ist:

$$\left(\frac{1}{t}\int_0^t S(t')dt' - 50\right)^+$$

mit den Parameter:

$S(0)$	r	g	σ	T	N	T_A
50 €	0.1	0	0.4	1	60	$\{t_0, \ldots, t_{60}\}$

Der entsprechende `Haskell`-Code ist:

```
input :: Input
input = (50, 0.1, 0, 0.4, 1, 60)
```

Die Konstruktion der Single-Asset-Option in `Haskell` lautet:

```
contract :: Contract
contract = get 0 60 (ASI 50) NoBar correct $ cTruncate 60
           (at [0..60] (cMax zero (sub (aveUnderlying 50)
           (konstE 50))))
```

wobei `correct` und `konstE` Konstruktionsfunktionen aus der erweiterbaren Bibliothek sind (siehe Unterabschnitt 4.1.4).

Das Ergebnis des `Haskell`-Befehls `evalCon input contract` ist 6.19 € und der entsprechende Referenzwert ist 6.17 €. Das Beispiel und der Referenzwert stammen aus dem Buch [14] von Hull.

Beispiel VII: Forward-Start-Call-Option

Die Auszahlungsfunktion der Single-Asset-Option zur Zeit $T > T'$ ist:

$$(S(T) - S(T'))^+$$

mit den Parameter:

$S(0)$	r	g	σ	T	T'	N	T_A
50 €	0.1	0.05	0.15	1	0.5	200	$\{t_{200}\}$

Der entsprechende `Haskell`-Code ist:

```
input :: Input
input = (50, 0.1, 0.05, 0.15, 1, 200)
```

Die Konstruktion der Single-Asset-Option in `Haskell` lautet:

```
contract :: Contract
contract = get 100 200 EBA NoBar correct $ cTruncate 200
           (at [200] (cMax zero (sub (underlying 50) (startP
           (underlying 50)))))
```

wobei `correct` eine Konstruktionsfunktion aus der erweiterbaren Bibliothek ist (siehe Unterabschnitt 4.1.4).

Das Ergebnis des `Haskell`-Befehls `evalCon input contract` ist 2.624 € und der entsprechende Referenzwert ist 2.629 €. Das Beispiel und der Referenzwert stammen aus dem Buch [35] von Zhang.

Beispiel VIII: Forward-Start-Put-Option

Die Auszahlungsfunktion der Single-Asset-Option zur Zeit $T > T'$ ist:

$$(S(T') - S(T))^+$$

mit den Parameter:

$S(0)$	r	g	σ	T	T'	N	T_A
50 €	0.1	0.05	0.15	1	0.5	200	$\{t_{200}\}$

Der entsprechende `Haskell`-Code ist:

```
input :: Input
input = (50, 0.1, 0.05, 0.15, 1, 200)
```

Die Konstruktion der Single-Asset-Option in `Haskell` lautet:

```
contract :: Contract
contract = get 100 200 EBA NoBar correct $ cTruncate 200
           (at [200] (cMax zero (sub (startP (underlying 50))
           (underlying 50))))
```

wobei `correct` eine Konstruktionsfunktion aus der erweiterbaren Bibliothek ist (siehe Unterabschnitt 4.1.4).

Das Ergebnis des `Haskell`-Befehls `evalCon input contract` ist 1.449 € und der entsprechende Referenzwert ist 1.454 €. Das Beispiel und der Referenzwert stammen aus dem Buch [35] von Zhang.

Beispiel IX: Europäischer Digital-Call

Die Auszahlungsfunktion der Single-Asset-Option zur Zeit T ist:

$$\mathbf{1}\{S(T) > 0.5\}$$

mit den Parameter:

$S(0)$	r	g	σ	T	N	T_A
0.5 €	0.1	0	0.5	0.5	1000	$\{t_{1000}\}$

Der entsprechende `Haskell`-Code ist:

```
input :: Input
input = (0.5, 0.1, 0, 0.5, 0.5, 1000)
```

Die Konstruktion der Single-Asset-Option in `Haskell` lautet:

```
contract :: Contract
contract = get 0 1000 EBA NoBar correct $ cTruncate 1000
       (at [1000] (cWhen (largerAS (underlying 0.5) (konstE 0.5)
          ) (konstE 1)))
```

wobei `correct` und `konstE` Konstruktionsfunktionen aus der erweiterbaren Bibliothek sind (siehe Unterabschnitt 4.1.4).

Das Ergebnis des `Haskell`-Befehls `evalCon input contract` ist 0.4502150 € und der entsprechende Referenzwert ist 0.4622006 €. Das Beispiel und der Referenzwert stammen aus der Dissertation [34] von Quecke.

Beispiel X: Amerikanischer Digital-Call

Die Auszahlungsfunktion der Single-Asset-Option zur Zeit $t \in [0, T]$ ist:

$$\mathbf{1}\{S(t) > 0.5\}$$

mit den Parameter:

$S(0)$	r	g	σ	T	N	T_A
a €	0.1	0	0.5	0.5	1000	$\{t_0, \ldots, t_{1000}\}$

Der entsprechende `Haskell`-Code ist:

```
input :: Input
input = (a, 0.1, 0, 0.5, 0.5, 1000)
```

Die Konstruktion der Single-Asset-Option in `Haskell` lautet:

```
contract :: Contract
contract = get 0 1000 EBA NoBar correct $ cTruncate 1000
        (at [0..1000] (cWhen (largerAS (underlying a) (konstE
           0.5)) (konstE 1)))
```

wobei `correct` und `konstE` Konstruktionsfunktionen aus der erweiterbaren Bibliothek sind (siehe Unterabschnitt 4.1.4).

Die Ergebnisse des `Haskell`-Befehls `evalCon input contract` sind:

a	`evalCon input contract`	Referenzwert
0.4 €	0.5057639 €	0.4944380 €
0.3 €	0.1341434 €	0.1318085 €
0.2 €	0.0083291 €	0.0079269 €

Das Beispiel und die Referenzwerte stammen aus der Dissertation [34] von Quecke.

Beispiel XI: Europäischer Floating-Lookback-Call

Die Auszahlungsfunktion der Single-Asset-Option zur Zeit T ist:

$$\left(S(T) - \min_{t \in [0,T]} S(t)\right)^+$$

mit den Parameter:

$S(0)$	r	g	σ	T	N	T_A
50 €	0.1	0	0.4	0.25	200	$\{t_{200}\}$

Der entsprechende `Haskell`-Code ist:

```
input :: Input
input = (50, 0.1, 0, 0.4, 0.25, 200)
```

Die Konstruktion der Single-Asset-Option in `Haskell` lautet:

```
contract :: Contract
contract = get 0 200 LBS NoBar correct $ cTruncate 200
           (at [200] (sub (underlying 50) (minUnderlying 50)))
```

wobei `correct` eine Konstruktionsfunktion aus der erweiterbaren Bibliothek ist (siehe Unterabschnitt 4.1.4).

Das Ergebnis des `Haskell`-Befehls `evalCon input contract` ist 7.75 € und der entsprechende Referenzwert ist 8.04 €. Das Beispiel und der Referenzwert stammen aus dem Buch [14] von Hull.

Beispiel XII: Europäischer Floating-Lookback-Put

Die Auszahlungsfunktion der Single-Asset-Option zur Zeit T ist:

$$\left(\max_{t\in[0,T]} S(t) - S(T)\right)^+$$

mit den Parameter:

$S(0)$	r	g	σ	T	N	T_A
50 €	0.1	0	0.4	0.25	200	$\{t_{200}\}$

Der entsprechende `Haskell`-Code ist:

```
input :: Input
input = (50, 0.1, 0, 0.4, 0.25, 200)
```

Die Konstruktion der Single-Asset-Option in `Haskell` lautet:

```
contract :: Contract
contract = get 0 200 LBS NoBar correct $ cTruncate 200
           (at [200] (sub (maxUnderlying 50) (underlying 50)))
```

wobei `correct` eine Konstruktionsfunktion aus der erweiterbaren Bibliothek ist (siehe Unterabschnitt 4.1.4).

Das Ergebnis des `Haskell`-Befehls `evalCon input contract` ist 7.39 € und der entsprechende Referenzwert ist 7.79 €. Das Beispiel und der Referenzwert stammen aus dem Buch [14] von Hull.

Beispiel XIII: Europäischer Down-and-Out-Call mit Rückvergütung

Die Auszahlungsfunktion der Single-Asset-Option zur Zeit T ist:

$$(S(T) - 98)^+ \cdot \mathbf{1}\left\{\min_{t\in[0,T]} S(t) > 95\right\}$$

mit den Parameter:

$S(0)$	r	g	σ	rbt	T	N	T_A
100 €	0.08	0.03	0.2	1	0.5	1000	$\{t_{1000}\}$

Der entsprechende `Haskell`-Code ist:

```
input :: Input
input = (100, 0.08, 0.03, 0.2, 0.5, 1000)
```

Die Konstruktion der Single-Asset-Option in `Haskell` lautet:

```
contract :: Contract
contract = get 0 1000 EBA (Out 1) (cTruncate 1000 (largerAS
           (underlying 100) (konstE 95))) $ cTruncate 1000
           (at [1000] (cMax zero (sub (underlying 100)
           (konstE 98))))
```

wobei `konstE` eine Konstruktionsfunktion aus der erweiterbaren Bibliothek ist (siehe Unterabschnitt 4.1.4).

Das Ergebnis des `Haskell`-Befehls `evalCon input contract` ist 0.669 € und der entsprechende Referenzwert ist 0.682 €. Das Beispiel und der Referenzwert stammen aus dem Buch [35] von Zhang.

Beispiel XIV: Europäischer Down-and-In-Call mit Rückvergütung

Die Auszahlungsfunktion der Single-Asset-Option zur Zeit T ist:

$$(S(T) - 98)^+ \cdot \mathbf{1}\left\{\min_{t\in[0,T]} S(t) \leq 95\right\}$$

mit den Parameter:

$S(0)$	r	g	σ	rbt	T	N	T_A
100 €	0.08	0.03	0.2	1.5	0.5	1000	$\{t_{1000}\}$

Der entsprechende `Haskell`-Code ist:

```
input :: Input
input = (100, 0.08, 0.03, 0.2, 0.5, 1000)
```

Die Konstruktion der Single-Asset-Option in `Haskell` lautet:

```
contract :: Contract
contract = get 0 1000 EBA (In 1.5) (cTruncate 1000 (largerAS
           (underlying 100) (konstE 95))) $ cTruncate 1000
           (at [1000] (cMax zero (sub (underlying 100)
           (konstE 98))))
```

wobei `konstE` eine Konstruktionsfunktion aus der erweiterbaren Bibliothek ist (siehe Unterabschnitt 4.1.4).

Das Ergebnis des `Haskell`-Befehls `evalCon input contract` ist 0.467 € und der entsprechende Referenzwert ist 0.449 €. Das Beispiel und der Referenzwert stammen aus dem Buch [35] von Zhang.

Beispiel XV: Europäischer Down-and-In-Call mit Floating-Barriere

Die Auszahlungsfunktion der Single-Asset-Option zur Zeit T ist:

$$(S(T) - 98)^+ \cdot \mathbf{1}\left\{\exists t \in [0,T] : S(t) \leq 95 \cdot e^{0.04 \cdot t}\right\}$$

mit den Parameter:

$S(0)$	r	g	σ	T	N	T_A
100 €	0.08	0.03	0.2	0.5	1000	$\{t_{1000}\}$

Der entsprechende `Haskell`-Code ist:

```
input :: Input
input = (100, 0.08, 0.03, 0.2, 0.5, 1000)
```

Die Konstruktion der Single-Asset-Option in `Haskell` lautet:

```
contract :: Contract
contract = get 0 1000 EBA (In 0) (cTruncate 1000 (largerEQ
           (underlying 100) (funcColumn 1000 (0.5 / 1000)
           (\x -> 95 * exp (0.04 * x))))) $ cTruncate 1000
           (at [1000] (cMax zero (sub (underlying 100)
           (konstE 98))))
```

wobei `funcColumn` und `konstE` Konstruktionsfunktionen aus der erweiterbaren Bibliothek sind (siehe Unterabschnitt 4.1.4).

Das Ergebnis des `Haskell`-Befehls `evalCon input contract` ist 2.878 € und der entsprechende Referenzwert ist 3.108 €. Das Beispiel und der Referenzwert stammen aus dem Buch [35] von Zhang.

Beispiel XVI: Europäischer Down-and-Out-Call mit Early-Ending-Barriere

Die Auszahlungsfunktion der Single-Asset-Option zur Zeit T ist:

$$(S(T) - 98)^+ \cdot \mathbf{1}\left\{\min_{t \in [0, \frac{T}{2}]} S(t) > 95\right\}$$

mit den Parameter:

$S(0)$	r	g	σ	T	N	T_A
100 €	0.08	0.03	0.2	0.5	1000	$\{t_{1000}\}$

Der entsprechende `Haskell`-Code ist:

```
input :: Input
input = (100, 0.08, 0.03, 0.2, 0.5, 1000)
```

Die Konstruktion der Single-Asset-Option in `Haskell` lautet:

```
contract :: Contract
contract = get 0 1000 EBA (Out 0) (cTruncate 1000 (area 0 500
           (largerAS (underlying 100) (konstE 95))))
           $ cTruncate 1000 (at [1000] (cMax zero (sub
           (underlying 100) (konstE 98))))
```

wobei `konstE` eine Konstruktionsfunktion aus der erweiterbaren Bibliothek ist (siehe Unterabschnitt 4.1.4).

Das Ergebnis des `Haskell`-Befehls `evalCon input contract` ist 5.483 € und der entsprechende Referenzwert ist 5.329 €. Das Beispiel und der Referenzwert stammen aus dem Buch [35] von Zhang.

Beispiel XVII: Europäischer Down-and-In-Call mit Early-Ending-Barriere

Die Auszahlungsfunktion der Single-Asset-Option zur Zeit T ist:

$$(S(T) - 98)^+ \cdot \mathbf{1}\left\{\min_{t\in[0,\frac{T}{2}]} S(t) \leq 95\right\}$$

mit den Parameter:

$S(0)$	r	g	σ	T	N	T_A
100 €	0.08	0.03	0.2	0.5	1000	$\{t_{1000}\}$

Der entsprechende `Haskell`-Code ist:

```
input :: Input
input = (100, 0.08, 0.03, 0.2, 0.5, 1000)
```

Die Konstruktion der Single-Asset-Option in `Haskell` lautet:

```
contract :: Contract
contract = get 0 1000 EBA (In 0) (cTruncate 1000 (area 0 500
           (largerAS (underlying 100) (konstE 95))))
           $ cTruncate 1000 (at [1000] (cMax zero (sub
           (underlying 100) (konstE 98))))
```

wobei `konstE` eine Konstruktionsfunktion aus der erweiterbaren Bibliothek ist (siehe Unterabschnitt 4.1.4).

Das Ergebnis des `Haskell`-Befehls `evalCon input contract` ist 2.400 € und der entsprechende Referenzwert ist 2.547 €. Das Beispiel und der Referenzwert stammen aus dem Buch [35] von Zhang.

Beispiel XVIII: Europäischer Down-and-Out-Call mit Forward-Start-Barriere

Die Auszahlungsfunktion der Single-Asset-Option zur Zeit T ist:

$$(S(T) - 102)^+ \cdot \mathbf{1}\left\{\min_{t\in[\frac{T}{2},T]} S(t) > 98\right\}$$

mit den Parameter:

$S(0)$	r	g	σ	T	N	T_A
100 €	0.1	0.05	0.2	0.5	500	$\{t_{500}\}$

Der entsprechende `Haskell`-Code ist:

```
input :: Input
input = (100, 0.1, 0.05, 0.2, 0.5, 500)
```

Die Konstruktion der Single-Asset-Option in `Haskell` lautet:

```
contract :: Contract
contract = get 0 500 EBA (Out 0) (cTruncate 500 (area 250 500
           (largerAS (underlying 100) (konstE 98))))
           $ cTruncate 500 (at [500] (cMax zero (sub
           (underlying 100) (konstE 102))))
```

wobei `konstE` eine Konstruktionsfunktion aus der erweiterbaren Bibliothek ist (siehe Unterabschnitt 4.1.4).

Das Ergebnis des `Haskell`-Befehls `evalCon input contract` ist 4.889 € und der entsprechende Referenzwert ist 5.068 €. Das Beispiel und der Referenzwert stammen aus dem Buch [35] von Zhang.

Beispiel XIX: Amerikanischer Down-and-In-Put

Die Auszahlungsfunktion der Single-Asset-Option zur Zeit $t \in [0, T]$ ist:

$$(100 - S(t))^+ \cdot \mathbf{1}\left\{\min_{t' \in [0,t]} S(t') \leq H\right\}$$

mit den Parameter:

$S(0)$	r	g	σ	T	N	T_A
a	0.06	0	0.2	0.5	500	$\{t_0, \ldots, t_{500}\}$

Der entsprechende `Haskell`-Code ist:

```
input :: Input
input = (75, 0.06, 0, 0.2, 0.5, 500)
```

Die Konstruktion der Single-Asset-Option in `Haskell` lautet:

```
contract :: Contract
contract = get 0 500 EBA (In 0) (largerAS (underlying a)
           (konstE H)) $ cTruncate 500 (at [0..500] (sub
           (konstE 100) (underlying a)))
```

wobei `konstE` eine Konstruktionsfunktion aus der erweiterbaren Bibliothek ist (siehe Unterabschnitt 4.1.4).

Die Ergebnisse des `Haskell`-Befehls `evalCon input contract` sind:

a	H	`evaluation contract`	Referenzwert
80 €	70 €	8.4346 €	8.8767 €
90 €	70 €	1.6776 €	1.7136 €
90 €	80 €	6.9489 €	7.0649 €
100 €	80 €	1.6989 €	1.7847 €
100 €	90 €	4.0223 €	4.1244 €
110 €	90 €	1.2346 €	1.2557 €

Das Beispiel und die Referenzwerte stammen aus dem Buch [1] von Aitsahlia, Imhof und Lai.

5. Projekt für Multi-Asset-Optionen

In diesem Kapitel stellen wir unser multidimensionales Projekt für Multi-Asset-Optionen vor. Die Vorgehensweise ist ähnlich der des letzten Kapitels gemäß Peyton-Jones' Konzept. Zuerst konstruieren wir die Multi-Asset-Optionen. Danach entwerfen wir die Bewertungsfunktion. Wir trennen das multidimensionale Projekt deshalb vom eindimensionalen, weil den Projekten verschiedenartige Konstruktionsideen und Bewertungsmethoden zugrunde liegen, die wir gesondert betrachten müssen.

5.1. Konstruktion von Multi-Asset-Optionen

Im Gegensatz zum eindimensionalen Projekt berücksichtigen wir im multidimensionalen Projekt keine Optionen auf maximale, minimale oder durchschnittliche Basiswerte, sondern nur Optionen mit normalen Basiswerten. Dies liegt daran, dass wegen des höhen Zeitaufwandes die Entwicklung des S-Baumes (genauer: S^{max}-Baumes, S^{min}-Baumes und S^{ave}-Baumes) einer Multi-Asset-Option mit maximalen, minimalen oder durchschnittlichen Basiswerten rechnerisch undurchführbar ist.

Angesichts dieses Aspektes können wir Multi-Asset-Optionen relativ einfach konstruieren. Dabei entwerfen wir, anders als im eindimensionalen Projekt, lediglich die primitiven Konstruktoren für die Auszahlungen bzw. die Auszahlungsfunktion einer Multi-Asset-Option. Danach setzen wir die aus den primitiven Konstruktoren kombinierte Auszahlungsfunktion und die anderen wichtigen Informationen über eine Multi-Asset-Option (z. B. die Barrieren, die Auszahlungstermine usw.) zusammen, damit eine Multi-Asset-Option lückenlos dargestellt werden kann.

5.1.1. Fundamentale Datentypen

Bevor wir die Konstruktion einer Multi-Asset-Option vorstellen, führen wir einige fundamentale Datentypen ein.

```
1 type Date = Int
```

`Date` steht nicht für ein echtes Datum, sondern den Datentyp ganzzahliger Zeitschritte bzw. Zeitpunkte.

```
1 type Datelist = [Date]
```

Der Datentyp `Datelist` enspricht einer Liste von Zeitschritten bzw. Zeitpunkten, dessen Objekte die Indizes der Auszahlungstermine der Menge T_A auflisten.

```
type Index = Int
```

Die Objekte vom Datentpy `Index` bezeichnen die Indizes der Basiswerte einer Multi-Asset-Option.

```
type LifeTime = Double
```

`LifeTime` ist der Datentyp des Zeitintervalls vom Anfangszeitpunkt bis zur Fälligkeit einer Multi-Asset-Option.

```
type Rate = Double
```

`Rate` ist der Datentyp des risikolosen Zinssatzes.

```
type Correlation = Double
```

Ein Objekt vom Datentyp `Correlation` bezeichnet die Korrelation zwischen zwei Basiswerten einer Multi-Asset-Option.

```
type Volatility = Double
```

`Volatility` ist der Datentyp der Volatilitäten der Basiswerte einer Multi-Asset-Option.

```
type StartPreis = Double
```

`StartPrice` ist der Datentyp der Anfangskurse der Basiswerte einer Multi-Asset-Option.

5.1.2. Primitive Konstruktoren von Auszahlungen

Wie bereits erwähnt, konzentrieren wir uns bei der Beschreibung einer Multi-Asset-Option auf die Konstruktion ihrer Auszahlungen bzw. ihrer Auszahlungsfunktion. Im Folgenden listen wir die primitiven Konstruktoren der Auszahlungen mittels der `Haskell`-Grammatik auf.

Programm 5.1: Primitive Konstruktoren der Auszahlungen einer Multi-Asset-Option

```
data Payment =
  -- Block 1
    Zero
  | S Index
  | Konst Double
  | Add Payment Payment
  | Sub Payment Payment
  | Multi Payment Payment
  | Max Payment Payment
  | Min Payment Payment
```

```
11    | Exp Payment
12    | Power Payment Payment
13    -- Block 2
14    | LargerAS Payment Payment
15    | SmallerAS Payment Payment
16    | Equal Payment Payment
17    | And Payment Payment
18    | Or Payment Payment
19    | Not Payment
20    -- Block 3
21    | If Payment Payment Payment
22    deriving Show
```

Es gibt vorläufig 17 Typkonstruktoren im Datentyp `Payment`, um die Auszahlungen einer Multi-Asset-Option zu beschreiben. Die Menge der Typkonstruktoren ist je nach Bedarf erweiterbar. Der Datentyp `Payment` besteht aus drei Blöcken. Die primitiven Konstruktoren im Block 1 können die Auszahlungen einer Multi-Asset-Option kombinieren. Die primitiven Konstruktoren im Block 2 sind boolesch bewertbar und werden als Kriterien für die Ausübung einer Multi-Asset-Option angesehen. Im Block 3 steht nur der primitive Konstruktor `If`.

5.1.3. Bibliothek der Konstruktionsfunktionen von Auszahlungen

In diesem Unterabschnitt funktionalisieren wir wie in Abschnitt 3.2.1 die primitiven Konstruktoren der Auszahlungen einer Multi-Asset-Option, um eine gemischte Kombination aus Funktionen und Typkonstruktoren zu vermeiden. Dabei werden die Bedeutungen der einzelnen primitiven Konstruktoren der Auszahlungen genau erklärt.

```
1 zero :: Payment
2 zero = Zero
```

Die Funktion `zero` ergibt eine leere Auszahlung.

```
1 cS :: Index -> Payment
2 cS = S
```

Durch `cS i` wird der Wert des i-ten Basiswertes einer Multi-Asset-Option ausgezahlt.

```
1 konst :: Double -> Payment
2 konst = Konst
```

Die Auszahlung `konst o` zahlt lediglich `o` Währungseinheiten aus.

```
1 add :: Payment -> Payment -> Payment
2 add = Add
```

Die Funktion `add a1 a2` ergibt die Summe der Auszahlungen `a1` und `a2`.

```
1 sub :: Payment -> Payment -> Payment
2 sub = Sub
```

Die Auszahlung `sub a1 a2` ergibt sich aus der Differenz der zwei Auszahlungen `a1` und `a2`.

```
1 multi :: Payment -> Payment -> Payment
2 multi = Multi
```

Die Auszahlung `multi a1 a2` entspricht der Multipikation der zwei Auszahlungen `a1` und `a2`.

```
1 cMax :: Payment -> Payment -> Payment
2 cMax = Max
```

`cMax a1 a2` zahlt das Maximum der Auszahlungen `a1` und `a2` aus.

```
1 cMin :: Payment -> Payment -> Payment
2 cMin = Min
```

`cMin a1 a2` zahlt das Minimum der Auszahlungen `a1` und `a2` aus.

```
1 cExp :: Payment -> Payment
2 cExp = Exp
```

Die Auszahlung `cExp a` entspricht dem natürlichen Exponential der Auszahlung `a`.

```
1 power :: Payment -> Payment -> Payment
2 power = Power
```

Die Funktion `power o a` ergibt das Exponential der Auszahlung `a` zur Basis `o`.

```
1 largerAS :: Payment -> Payment -> Payment
2 largerAS = LargerAS
```

Die Funktion `largerAS a1 a2` ist boolesch bewertbar. Sie vergleicht, ob die Auszahlung `a1` größer ist als die Auszahlung `a2`.

```
1 smallerAS :: Payment -> Payment -> Payment
2 smallerAS = SmallerAS
```

Die Funktion `smallerAS a1 a2` ist boolesch bewertbar. Sie vergleicht, ob die Auszahlung `a1` kleiner ist als die Auszahlung `a2`.

```
1 equal :: Payment -> Payment -> Payment
2 equal = Equal
```

Die Funktion `equal a1 a2` ist boolesch bewertbar. Sie überprüft, ob die Auszahlung `a1` der Auszahlung `a2` entspricht.

```
1 cAnd :: Payment -> Payment -> Payment
2 cAnd = And
```

Die boolesche Auszahlung `cAnd a1 a2` ist die Konjunktion der zwei booleschen Auszahlungen `a1` und `a2`.

```
cOr :: Payment -> Payment -> Payment
cOr = Or
```

Die boolesche Auszahlung `cOr a1 a2` ist die Disjunktion der zwei booleschen Auszahlungen `a1` und `a2`.

```
cNot :: Payment -> Payment
cNot = Not
```

Bei der booleschеb Auszahlung `cNot a` handelt es sich um die Negation der booleschen Auszahlung `a`.

```
cIf :: Payment -> Payment -> Payment -> Payment
cIf = If
```

Die Auszahlung `cIf a0 a1 a2` zahlt `a1` aus, falls die boolesche Auszahlung `a0` **`True`** ist, sonst zahlt sie `a2` aus.

Jetzt erweitern wir die Bibliothek der Konstruktionsfunktionen der Auszahlungen, die durch die obigen primitiven Konstruktionsfunktionen kombiniert werden. Die Bibliothek kann nach Bedarf beliebig erweitert werden.

```
correct :: Payment
correct = zero
```

Die Konstruktionsfunktion `correct` gleicht der die Konstruktionsfunktion `zero`. Die entsprechende Auszahlung wird immer boolesch als **`True`** bewertet.

```
notCorrect :: Payment
notCorrect = cNot correct
```

Die Konstruktionsfunktion `notCorrect` besteht aus der primitiven Konstruktionsfunktion `cNot` und der obigen Funktion `correct`. Die entsprechende Auszahlung wird immer boolesch als **`False`** bewertet.

```
largerEQ :: Payment -> Payment -> Payment
largerEQ q1 q2 = cOr (largerAS q1 q2) (equal q1 q2)
```

Die Konstruktionsfunktion `largerEQ` besteht aus den primitiven Konstruktionsfunktionen `cOr`, `largerAS` und `equal`. Die Auszahlung `largerEQ a1 a2` ist boolesch bewertbar und überprüft, ob die Auszahlung `a1` größer oder gleich der Auszahlung `a2` ist.

```
smallerEQ :: Payment -> Payment -> Payment
smallerEQ q1 q2 = cOr (smallerAS q1 q2) (equal q1 q2)
```

Die Konstruktionsfunktion `smallerEQ` besteht aus den primitiven Konstruktionsfunktionen `cOr`, `smallerAS` und `equal`. Die Auszahlung `smallerEQ a1 a2` ist boolesch bewertbar und überprüft, ob die Auszahlung `a1` kleiner oder gleich der Auszahlung `a2` ist.

5.1.4. Datenstrukturen zur Konstruktion der Multi-Asset-Optionen

Basierend auf dem Datentyp `Payment` können wir die Normalform einer Multi-Asset-Option definieren. Zunächst stellen wir die Datentypen über ihre Barrieren und Eingabeinformationen vor.

```
data BarrierType = NoBarrier | OnlyOut | ExistIn
                 deriving (Eq, Show)
```

Der Datentyp `BarrierType` steht für den Barrieretyp einer Multi-Asset-Option, wobei `NoBarrier` auf eine Multi-Asset-Option ohne Barriere, `OnlyOut` auf eine Multi-Asset-Option nur mit Knock-Out-Barrieren und `ExistIn` auf eine Multi-Asset-Option mit mindestens einer Knock-In-Barriere zutrifft.

```
data Barrier = InWenn Datelist Payment
             | OutWenn Datelist Payment
               deriving Show
```

Der Datentyp `Barrier` hat zwei Typkonstruktoren `InWenn` und `OutWenn`, die sich wie folgt funktionalisieren.

```
inWenn :: Datelist -> Payment -> Barrier
inWenn = InWenn
```

`inWenn dl pay` ist eine Knock-In-Barriere mit dem Knock-In-Kriterium `pay` und den Entscheidungsterminen `dl`.

```
outWenn :: Datelist -> Payment -> Barrier
outWenn = OutWenn
```

`outWenn dl pay` ist eine Knock-Out-Barriere mit dem Knock-Out-Kriterium `pay` und den Entscheidungsterminen `dl`.

```
type BarrierList = (BarrierType, [Barrier])
```

Der Datentyp `BarrierList` fügt den Typ der Barrieren `BarrierType` einer Multi-Asset-Option und die Liste der Informationen von Barrieren `[Barrier]` zusammen.

```
type Matrix a = [[a]]
```

`Matrix a` definiert die Matrizen, deren Elemente zum Datentyp `a` gehören.

```
type Input = ([(StartPreis, Volatility)], Matrix Correlation,
              Rate, LifeTime, Date)
```

Der Datentyp `Input` vereint alle wichtigen Informationen einer Multi-Asset-Option. Diese sind: Anfangskurse und Volatilitäten der Basiswerte, Korrelationsmatrix, risikoloser Zinssatz, Fälligkeit der Option und Anzahl der gesamten Zeitschritte der Option.

```
data Contract = Get Input Datelist BarrierList Payment
                deriving Show
```

Ein Objekt vom Datentyp `Contract` entspricht der Normalform einer Multi-Asset-Option in unserem Konzept, die wir im Folgenden funktionalisieren und erklären.

```
1 get :: Input -> Datelist -> BarrierList -> Payment -> Contract
2 get = Get
```

Die Konstruktionsfunktion "`get input dateL (barrierT, barrierL) payM`" beschreibt die vollständige Form einer Multi-Asset-Option, die wirin unserem Konzept als die Normalform bezeichnen. Dabei umfasst `input` die Grundinformationen einer Multi-Asset-Option und `dateL` bezeichnet die Liste aller Auszahlungszeitpunkte in der Menge T_A. Das Tupel `(barrierT, barrierL)` enthält die Informationen über die Barrieren einer Multi-Asset-Option und `payM` ist die Auszahlungsfunktion der Multi-Asset-Option.

5.2. Bewertungsfunktion für Multi-Asset-Optionen

In diesem Abschnitt diskutieren wir die Bewertungsfunktion unseres multidimensionalen Konzepts zur Bewertung der Multi-Asset-Optionen. Wir beginnen mit der Bewertungsstruktur.

5.2.1. Bewertungsstruktur

Durch die folgenden Typ-Synonyme haben wir in `Haskell` die Bewertungsstruktur entworfen:

```
1 newtype BTree knoten = BTree {unBT :: [Tree knoten]}
2                        deriving Show
3
4 type Tree knoten = [[knoten]]
```

Der Datentyp `Tree knoten` ist eine zweidimensionale Liste bzgl. des Datentyps `knoten`, welcher in unserem Konzept einen Binomialbaum beschreibt, der in jedem Knoten nur einen Wert enthält. Der Datentyp `BTree knoten` ist deshalb eine Liste von Binomialbäumen, welcher wie in Abbildung 5.1 aussieht. Die

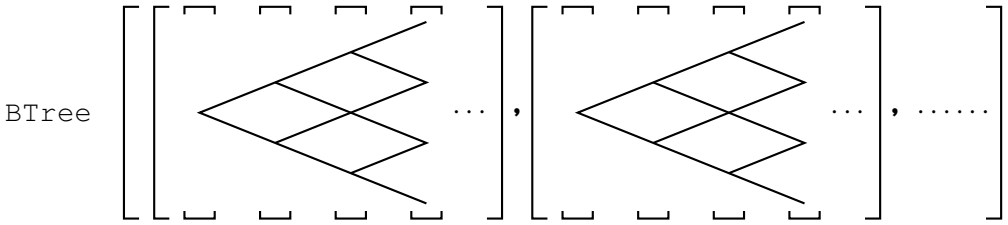


Abbildung 5.1.: Bewertungsstruktur für Multi-Asset-Optionen

Bewertungsstruktur unseres multidimensionalen Konzepts trägt lediglich zur

Konstruktion des Binomialbaumes für die Basiswerte einer Multi-Asset-Option (die Y_i-Bäume für $i = 1, \ldots, M$) bei, während die Bewertungsstruktur unseres eindimensionalen Konzepts den ganzen Bewertungsablauf (von S-Baum bis V-Baum) einer Single-Asset-Option begleitet. Die Bestimmung des Y-Baumes und S-Baumes sowie die Berechnung des V-Baumes einer Multi-Asset-Option werden wir in unserem `Haskell`-Programm nur aufgrund des Datentyps der reellen Liste `[Double]` durchf'hren.

5.2.2. Implementierung der Bewertungsfunktionen

In diesem Unterabschnitt gehen wir auf die Bewertungsfunktionen für Multi-Asset-Optionen ein. Hinsichtlich der verschiedenen Bewertungsobjekte werden wir die Bewertungsfunktionen für `Payment` und `Contract` separat diskutieren.

Bewertungsfunktion für `Payment`

Es gibt zwei Bewertungsfunktionen für den Datentyp `Payment`, die eng mit den primitiven Konstruktoren der Auszahlungen einer Multi-Asset-Option zusammenarbeiten. Die erste ist die Bewertungsfunktion `calculate` mit zwei Argumenten. Das erste Argument ist eine Liste der Kurse der Basiswerte der Multi-Asset-Option. Das zweite Argument ist eine reelle Auszahlungsfunktion vom Datentyp `Payment`. Das Resultat der Bewertungsfunktion `calculate` ist ein reeller Wert (siehe Programm 5.2).

Programm 5.2: Bewertungsfunktion für die reelle Auszahlung einer Multi-Asset-Optionen

```
1  calculate :: [Double] -> Payment -> Double
2  calculate l = calcul
3    where calcul Zero           = 0
4          calcul (S i)          = l !! (i-1)
5          calcul (Konst d)      = d
6          calcul (Add p1 p2)    = (calcul p1) + (calcul p2)
7          calcul (Sub p1 p2)    = (calcul p1) - (calcul p2)
8          calcul (Multi p1 p2)  = (calcul p1) * (calcul p2)
9          calcul (Div p1 p2)    = (calcul p1) / (calcul p2)
10         calcul (Max p1 p2)    = max (calcul p1) (calcul p2)
11         calcul (Min p1 p2)    = min (calcul p1) (calcul p2)
12         calcul (Exp p)        = exp (calcul p)
13         calcul (Power d p)    = (calcul p) ** d
14         calcul (Ln p)         = log (calcul p)
15         calcul (Log d p)      = logBase d (calcul p)
16         calcul (If p1 p2 p3)  = if (estimate l p1)
17                                 then (calcul p2) else (calcul p3)
```

Die zweite Bewertungsfunktion ist die Funktion `estimate` mit ebenfalls zwei Argumenten. Erneut ist das erste Argument eine Liste der Kurse der Basiswerte der Multi-Asset-Option. Das zweite Argument ist eine boolesche Auszahlungsfunktion vom Datentyp `Payment`, die einen booleschen Wert zurückgibt (siehe Programm 5.3).

Programm 5.3: Bewertungsfunktion für die boolesche Auszahlung einer Multi-Asset-Optionen

```
estimate :: [Double] -> Payment -> Bool
estimate l = estima
  where
    estima Zero               = True
    estima (LargerAS p1 p2)   =
                              (calculate l p1) > (calculate l p2)
    estima (SmallerAS p1 p2) =
                              (calculate l p1) < (calculate l p2)
    estima (Equal p1 p2)      =
                              (calculate l p1) == (calculate l p2)
    estima (And p1 p2)        = (estima p1) && (estima p2)
    estima (Or p1 p2)         = (estima p1) || (estima p2)
    estima (Not p)            = not (estima p)
```

Die primitiven Funktionen in den Bewertungsfunktionen `calculate` und `estimate` sind allesamt Standardfunktionen von `Haskell`.

Bewertungsfunktion für `Contract`

Wir stellen jetzt die Bewertungsfunktion für `Contract`, die `evaluation`-Funktion, vor. Diese ist die zentrale Bewertungsfunktion unseres Konzepts, mit der wir die Normalform einer Multi-Asset-Option bewerten können, die durch das Schlüsselwort `Get` gekennzeichnet wird. Die Bewertungsfunktion `evaluation` gestalt sich wie folgt:

```
evaluation :: Contract -> Double
evaluation (Get input@(sVol, corr, rate, rTime, tStep) dateL
           (barrierT, barrierL) payM)
  | barrierT == NoBarrier = head (discountNo 0)
  | barrierT == OnlyOut   = head (discountOut 0)
  | barrierT == ExistIn   = head (discountIn 0)
```

Die Verfahrensweise von `evaluation` folgt buchstäblich der entkoppelten multidimensionalen Binomialmethode aus Unterabschnitt 2.3.2. Die Funktion `evaluation a` erstellt zunächst die Y_i-Bäume für $i = 1, \ldots, M$ einer Multi-Asset-Option `a`. Danach entwickelt sie den Y-Baum durch das kartesische Produkt aller Y_i-Bäume und wandelt diesen in den S-Baum um. Zuletzt berechnet sie

rückwärts entlang der Zeitschritte den Anfangsknoten des V-Baumes von `a` durch die primitiven Funktionen `discountNo`, `discountOut` und `discountIn`.

- Die primitive Funktion

```
discountNo :: Date -> [Double]
```

  berechnet gemäß Algorithmus 25 den V-Baum einer Multi-Asset-Option ohne Barriere, die durch den Barrieretyp `NoBarrier` gekennzeichnet wird.

- Die primitive Funktion

```
discountOut :: Date -> [Double]
```

  berechnet gemäß Algorithmus 26 den V-Baum einer Multi-Asset-Option mit lediglich Knock-Out-Barrieren, die durch den Barrieretyp `OnlyOut` gekennzeichnet wird.

- Die primitive Funktion

```
discountIn :: Date -> [Double]
```

  ermittelt gemäß Algorithmus 27 den V-Baum einer Multi-Asset-Option mit mindestens einer Knock-In-Barriere, die durch den Barrieretyp `ExistIn` gekennzeichnet wird.

Diese drei Algorithmen können in Anhang B nachgelesen werden.

Mit Blick auf die Konstruktionsidee einer Multi-Asset-Option (siehe Abschnitt 5.1) halten wir fest, dass die Bewertungsfunktion den S-Baum einer Multi-Asset-Option nicht berechnen muss. Die Bestimmung der Knoten des Y-Baumes und S-Baumes kann direkt in den Bewertungsprozess nach dem Aufbau der Y_i-Bäume bei der Berechnung des V-Baumes integriert werden. Dies ist deshalb möglich, weil alle Knoten des Y-Baumes und S-Baumes im multidimensionalen Projekt eindeutig bestimmt werden, ohne Mischung von maximalen, minimalen oder durchschnittlichen Basiswerten.

5.3. Numerische Beispiele

In diesem Abschnitt werden wir unseren multidimensionalen Algorithmus für Multi-Asset-Optionen überprüfen und einige Optionen als numerische Beispiele bewerten. Dabei konzentrieren wir uns wie auch im eindimensionalen Fall auf die Bewertungsfähigkeit des Algorithmus und den Umfang seiner Anwendung.

5.3.1. Demonstration

Wir beginnen mit einer kurzen Demonstration. Das ausgewählte Beispiel dieser Demonstration ist ein amerikanischer Down-and-Out-Call mit zwei Basiswerten, dessen Auszahlungsfunktion zur Zeit t wie folgt gegeben ist:

$$\left(\sqrt{S_1(t) \cdot S_2(t)} - 20\right)^+ \cdot \mathbf{1}\left\{\min_{t\in[0,T]} S_2(t) > 45\right\}. \tag{5.1}$$

Der Anfangskurs des ersten Basiswertes S_1 ist 20 € und seine Volatilität liegt bei 0.2. Der Anfangskurs des zweiten Basiswertes S_2 beträgt 30 € bei einer Volatilität von 0.3. Die Korrelation zwischen S_1 und S_2 ist 0.5. Der risikolose Zinssatz liegt bei 0.1 und die Laufzeit der Zwei-Asset-Option ist ein Jahr. Darüber hinaus werden für die Simulation insgesamt 2 Zeitschritte durchgeführt. Die folgende Tabelle listen die Eingabedaten auf:

i	$S_i(0)$	σ_i	ρ_{1i}	ρ_{2i}	r	T	N	T_A
1	20 €	0.2	1	0.5	0.1	1	2	$\{t_0, t_1, t_2\}$
2	30 €	0.3	0.5	1				

Der entsprechende `Haskell`-Input lautet:

```
input = ([(20, 0.2), (30, 0.3)], [[1, 0.5], [0.5, 1]],
         0.1, 1, 2) :: Input
```

Schritt für Schritt konstruieren wir nun diese Zwei-Asset-Option mittels der Konstruktionsfunktionen aus unserem multidimensionalen Konzept.

- Konstruktion der Auszahlung

$$\left(\sqrt{S_1(t) \cdot S_2(t)} - 20\right)^+ : \tag{5.2}$$

  - Die `Haskell`-Darstellungen der beiden Basiswerte $S_1(t)$ und $S_2(t)$ sind jeweils `cS 1 :: Payment` und `cS 2 :: Payment`.
  - Die Multiplikation der Kurse der beiden Basiswerte lautet:

    ```
    multi (cS 1) (cS 2) :: Payment
    ```

  - Die Quadratwurzel dieser Multiplikation ist dann:

    ```
    power (konst 0.5) (multi (cS 1) (cS 2)) :: Payment
    ```

  - Die Darstellung des Ausübungspreises 20 € in `Haskell` lautet `konst 20 :: Payment`.
  - Die `Haskell`-Form für die Subtraktion innerhalb der beiden Klammern in Formel (5.2) lautet:

```
sub (power (konst 0.5) (multi (cS 1) (cS 2))) (konst 20)
:: Payment
```

– Somit ergibt sich die `Haskell`-Darstellung der Auszahlung (5.2) als:

```
payment = cMax (konst 0) (sub (power (konst 0.5) (multi
          (cS 1) (cS 2))) (konst 20)) :: Payment
```

Abbildung D.1 in Anhang D wandelt die obige `Haskell`-Form für die Auszahlung in eine übersichtlich zerlegte Baumdarstellung um.

- Konstruktion der Bedingung der Knock-Out-Barriere

$$\min_{t\in[0,T]} S_2(t) > 45 : \tag{5.3}$$

– Die Darstellung der Barriere 45 € in `Haskell` ist `konst 45 :: Payment`.

– Die `Haskell`-Form für den Basiswert $S_2(t)$ lautet `cS 2 :: Payment`.

– Das Kriterium (5.3) bzw. seine äquivalente Form

$$\forall t \in [0,T] : S(t) > 45$$

ist daher in `Haskell` wie folgt gegeben:

```
barriereOut = largerAS (cS 2) (konst 45) :: Payment
```

Die Abbildung D.2 in Anhang D wandelt die obige `Haskell`-Form für das Barriere-Kriterium in eine übersichtlich zerlegte Baumdarstellung um.

- Einsetzen von Auszahlung und Knock-Out-Barriere in die Normalform der Zwei-Asset-Option:

  Unter Verwendung der Konstruktionsfunktion `get` erhalten wir die Option `beispielOption` als:

```
beispielOption = get input4 [0..2] (OnlyOut, [OutWenn [0..2]
                 barriereOut]) payment :: Contract
```

  Oder ausführlich:

```
beispielOption = get input4 [0..2] (OnlyOut, [OutWenn [0..2]
                 (largerAS (cS 2) (konst 45))]) (cMax
                 (konst 0) (sub (power (konst 0.5) (multi
                 (cS 1) (cS 2))) (konst 20))) :: Contract
```

Die Abbildungen D.3, D.4, D.5 und D.6 in Anhang D erklären Schritt für Schritt den Bewertungsvorgang von

```
1 evaluation beispielOption
```

- Abbildung D.3 berechnet den Y_1-Baum und Y_2-Baum der Zwei-Asset-Option. Dabei markieren wir die Knoten jedes Zeitschritts mit einer individuellen Farbe. Die Farbe zur Zeit t_0 ist rot, die Farbe zur Zeit t_1 ist blau und die Farbe zur Zeit t_2 ist grün. In den anderen drei Abbildungen verwenden wir dieselben Farben, um die Zeitschritte zu kennzeichnen.
- Das obere kleinere Quadrat der Abbildung D.4 entwickelt den Y-Baum vom Zeitpunkt t_0 zum Zeitpunkt t_1 durch den Algorithmus 15 basierend auf dem Y_1-Baum und Y_2-Baum. Das untere größere Quadrat der Abbildung D.4 entwickelt den Y-Baum vom Zeitpunkt t_1 zum Zeitpunkt t_2 durch Algorithmus 15.
- Das obere kleinere Quadrat der Abbildung D.5 wandelt den Y-Baum vom Zeitpunkt t_0 zum Zeitpunkt t_1 in den S-Baum der Zwei-Asset-Option um. Das untere größere Quadrat der Abbildung D.5 wandelt den Y-Baum vom Zeitpunkt t_1 zum Zeitpunkt t_2 in den S-Baum um.
- Das obere größere Quadrat der Abbildung D.6 berechnet rückwärts den V-Baum der Zwei-Asset-Option vom Zeitpunkt t_2 zum Zeitpunkt t_1 durch Algorithmus 26. Das untere kleinere Quadrat der Abbildung D.6 berechnet rückwärts den V-Baum vom Zeitpunkt t_2 zum Zeitpunkt t_1 durch Algorithmus 26.

5.3.2. Bewertungsbeispiele

In diesem Unterabschnitt bewerten wir eine Reihe von unterschiedlichen Multi-Asset-Optionen durch unseren multidimensionalen Algorithmus. Dabei übernehmen wir wie beim eindimensionalen Projekt die numerischen Beispiele aus einigen wissenschaftlichen Artikeln und betrachten ihre numerische Ergebnisse als die Referenzwerte der numerischen Ergebnisse unseres eigenen Algorithmus. Die ausgewählten wissenschaftlichen Artikel sind [2], [19], [21], [23], [34].

Beispiel I: Europäische Compound-Exchange-Option

Die Auszahlungsfunktion der Multi-Asset-Option zur Zeit T ist:

$$\Big((S_1(T) - K_1)^+ - (S_2(T) - K_2)^+\Big)^+ .$$

mit den Parameter:

i	$S_i(0)$	σ_i	ρ_{1i}	ρ_{2i}	r	T	N	T_A
1	100 €	0.1	1	0.8	0.04	1	20	$\{t_{20}\}$
2	80 €	0.1	0.8	1				

Die Konstruktion der Multi-Asset-Option in `Haskell` lautet:

```
input :: Input
input = ([(100, 0.1), (80, 0.1)], [[1, 0.8], [0.8, 1]],
         0.04, 1, 20)

payment :: Payment
payment = cMax (konst 0) (sub (cMax (konst 0) (sub (cS 1)
           (konst 60))) (cMax (konst 0) (sub (cS 2) (konst 60))))

contract :: Contract
contract = get input [20] (NoBarrier, []) payment
```

Das Ergebnis des `Haskell`-Befehls `evaluation contract` ist 19.9994 € und der entsprechende Referenzwert ist 20.0676 €. Das Beispiel und der Referenzwert stammen aus dem Artikel [2] von Alexander und Venkatramanan.

Beispiel II: Amerikanischer Rainbow-Minimum-Put

Die Auszahlungsfunktion der Multi-Asset-Option zur Zeit $t \in [0, T]$ ist:

$$(5 - \min(S_1(t), S_2(t)))^+ .$$

mit den Parameter:

i	$S_i(0)$	σ_i	ρ_{1i}	ρ_{2i}	r	T	N	T_A
1	5 €	0.2	1	0.3	0.1	1	100	$\{t_1, \ldots, t_{100}\}$
2	5 €	0.3	0.3	1				

Die Konstruktion der Multi-Asset-Option in `Haskell` lautet:

```
input :: Input
input = ([(5, 0.2), (5, 0.3)], [[1, 0.3], [0.3, 1]],
         0.1, 1, 100)

payment :: Payment
payment = cMax (konst 0) $ sub (konst 5) (cMin (cS 1) (cS 2))

contract :: Contract
contract = get input [0..100] (NoBarrier, []) payment
```

Das Ergebnis des `Haskell`-Befehls `evaluation contract` ist 0.521850 € und der entsprechende Referenzwert ist 0.521123 €. Das Beispiel und der Referenzwert stammen aus der Dissertation [34] von Quecke.

Beispiel III: Europäischer Zwei-Asset-Digital-Put

Die Auszahlungsfunktion der Multi-Asset-Option zur Zeit T ist:

$$\mathbf{1}\left\{\max(S_1(T), S_2(T)) < 5\right\}.$$

mit den Parameter:

i	$S_i(0)$	σ_i	ρ_{1i}	ρ_{2i}	r	T	N	T_A
1	5 €	0.2	1	0.3	0.1	1	100	$\{t_{100}\}$
2	5 €	0.3	0.3	1				

Die Konstruktion der Multi-Asset-Option in `Haskell` lautet:

```
input :: Input
input = ([(5, 0.2), (5, 0.3)], [[1, 0.3], [0.3, 1]],
        0.1, 1, 100)

payment :: Payment
payment = cIf (smallerAS (cMax (cS 1) (cS 2)) (konst 5))
          (konst 1) (konst 0)

contract :: Contract
contract = get input [100] (NoBarrier, []) payment
```

Das Ergebnis des `Haskell`-Befehls `evaluation contract` ist 0.1857 € und der entsprechende Referenzwert ist 0.1737 €. Das Beispiel und der Referenzwert stammen aus der Dissertation [34] von Quecke.

Beispiel IV: Europäische Drei-Asset-Spread-Option

Die Auszahlungsfunktion der Multi-Asset-Option zur Zeit T ist:

$$(S_1(T) - S_2(T) - S_3(T) - K)^+.$$

mit den Parameter:

i	$S_i(0)$	σ_i	ρ_{1i}	ρ_{2i}	ρ_{3i}	r	T	N	T_A
1	150 €	a	1	0.2	0.8	0.05	0.25	10	$\{t_{10}\}$
2	60 €	a	0.2	1	0.4				
3	50 €	a	0.8	0.4	1				

Die Konstruktion der Multi-Asset-Option in `Haskell` lautet:

```
input :: Input
input = ([(150, a), (60, a), (50, a)], [[1, 0.2, 0.8],
        [0.2, 1, 0.4], [0.8, 0.4, 1]], 0.05, 0.25, 10)

```

```
payment :: Payment
payment = cMax (konst 0) $ sub (sub (sub (cS 1) (cS 2))
          (cS 3)) (konst K)

contract :: Contract
contract = get input [10] (NoBarrier, []) payment
```

Die Ergebnisse des `Haskell`-Befehls `evaluation contract` sind:

a	0.3		0.6	
K	`evaluation contract`	Referenzwert	`evaluation contract`	Referenzwert
30 €	13.5924 €	13.5762 €	20.2306 €	20.2066 €
35 €	10.3801 €	10.3573 €	17.5150 €	17.4770 €
40 €	7.6851 €	7.6610 €	15.0668 €	15.0280 €
45 €	5.5179 €	5.4914 €	12.8850 €	12.8516 €
50 €	3.8291 €	3.8150 €	10.9624 €	10.9347 €

Das Beispiel und die Referenzwerte stammen aus dem Artikel [23] von Li, Deng und Zhou.

Beispiel V: Europäischer Vier-Asset-Basket-Call

Die Auszahlungsfunktion der Multi-Asset-Option zur Zeit T ist:

$$\left(\frac{1}{4}\sum_{i=1}^{4} S_i(T)\right)^+.$$

mit den Parameter:

i	$S_i(0)$	σ_i	ρ_{1i}	ρ_{2i}	ρ_{3i}	ρ_{4i}	r	T	N	T_A
1	100 €	0.2	1	0.5	0.5	0.5	0.1	1	20	$\{t_{20}\}$
2	100 €	0.2	0.5	1	0.5	0.5				
3	100 €	0.2	0.5	0.5	1	0.5				
4	100 €	0.2	0.5	0.5	0.5	1				

Die Konstruktion der Multi-Asset-Option in `Haskell` lautet:

```
input :: Input
input = ([(100, 0.2), (100, 0.2), (100, 0.2), (100, 0.2)],
        [[1, 0.5, 0.5, 0.5], [0.5, 1, 0.5, 0.5], [0.5, 0.5,
        1, 0.5], [0.5, 0.5, 0.5, 1]], 0.1, 1, 20)

payment :: Payment
payment = cMax (konst 0) $ sub (multi (konst 0.25) (add (add
          (cS 1) (cS 2)) (add (cS 3) (cS 4)))) (konst K)

contract :: Contract
contract = get input [20] (NoBarrier, []) payment
```

Die Ergebnisse des `Haskell`-Befehls `evaluation contract` sind:

K	`evaluation contract`	Referenzwert
0 €	99.99913382 €	99.99999999 €
50 €	54.75726293 €	54.75813057 €
80 €	27.70829203 €	27.71474151 €
100 €	11.93572969 €	11.92139639 €

Das Beispiel und die Referenzwerte stammen aus der Diplomarbeit [21] von Kürten.

Beispiel VI: Europäischer Basket-Call mit Knock-Out-Barrieren

Die Auszahlungsfunktion der Multi-Asset-Option zur Zeit T ist:

$$(S_1(t) + S_2(t) - 5)^+ \mathbf{1}\left\{\min_{t\in[0,T]}(S_1(t) + S_2(t)) > 5\right\} \cdot \mathbf{1}\left\{\max_{t\in[0,T]}(S_1(t) + S_2(t)) < 10\right\}.$$

mit den Parameter:

i	$S_i(0)$	σ_i	ρ_{1i}	ρ_{2i}	r	T	N	T_A
1	a	0.2	1	0.3	0.1	1	100	$\{t_{100}\}$
2	b	0.3	0.3	1				

Die Konstruktion der Multi-Asset-Option in `Haskell` lautet:

```
input :: Input
input = ([(a, 0.2), (b, 0.3)], [[1, 0.3], [0.3, 1]],
        0.1, 1, 100)

barriereOut1 = smallerEQ (add (cS 1) (cS 2)) (konst 5)

barriereOut2 = largerEQ (add (cS 1) (cS 2)) (konst 10)

payment :: Payment
payment = cMax (konst 0) $ sub (add (cS 1) (cS 2)) (konst 5)

contract :: Contract
contract = get input [100] (OnlyOut, [outWenn [0..100]
           barriereOut1, outWenn [0..100] barriereOut2])
           payment
```

Die Ergebnisse des `Haskell`-Befehls `evaluation contract` sind:

a	b	`evaluation contract`	Referenzwert
3 €	3 €	1.30099 €	1.27747 €
4 €	2 €	1.35419 €	1.33825 €
4 €	4 €	1.59894 €	1.56239 €
6 €	2 €	1.72065 €	1.70626 €

Das Beispiel und die Referenzwerte stammen aus der Dissertation [34] von Quecke.

Beispiel VII: Cash-Or-Nothing-Option mit Knock-In-Barriere und Knock-Out-Barriere

Die Auszahlungsfunktion der Multi-Asset-Option zur Zeit T ist:

$$100 \cdot \mathbf{1}\left\{\max_{t\in[0,T]} S_1(t) \geq 25\right\} \cdot \mathbf{1}\left\{\min_{t\in[0,T]} S_2(t) > 15\right\}.$$

mit den Parameter:

i	$S_i(0)$	σ_i	ρ_{1i}	ρ_{2i}	r	T	N	T_A
1	20 €	0.2	1	0.5	0.1	1	100	$\{t_{100}\}$
2	30 €	0.3	0.5	1				

Die Konstruktion der Multi-Asset-Option in `Haskell` lautet:

```
1  input :: Input
2  input = ([(20, 0.2), (30, 0.3)], [[1, 0.5], [0.5, 1]],
3           0.1, 1, 100)
4
5  barriereIn :: Payment
6  barriereIn = largerEQ (S 1) (konst 25)
7
8  barriereOut :: Payment
9  barriereOut = smallerEQ (S 2) (konst 15)
10
11 payment :: Payment
12 payment = konst 100
13
14 contract :: Contract
15 contract = get input [100] (ExistIn, [InWenn [0..100]
16             barriereIn, OutWenn [0..100] barriereOut]) payment
```

Das Ergebnis des `Haskell`-Befehls `evaluation contract` ist 33.708 € und der entsprechende Referenzwert ist 33.71 €. Das Beispiel und der Referenzwert stammen aus dem Artikel [19] von Korn und Müller.

6. Zusammenfassung und Ausblick

Finanzderivate zu beschreiben und Optionen zu bewerten ist scheinbar kein sehr viel versprechendes Gebiet für die funktionale Programmierung und formale Semantik. Uns ist es jedoch gelungen, Erkenntnisse aus der funktionalen Programmierung unmittelbar auf dieses komplexe Thema anwenden zu können, um eine große Klasse von Finanzderivaten zu beschreiben und zu bewerten.

In jedem unserer beiden Projekte, dem Projekt für Single-Asset-Optionen und dem Projekt für Multi-Asset-Optionen, erstellten wir eine kleine Klasse von primitiven Konstruktoren und die darauf basierende, erweiterbare Bibliothek von Konstruktionsfunktionen, um eine große Klasse von Optionen und anderen Finanzderivaten kombinieren und beschreiben zu können. Ferner entwickelt wir einen kompositorischen Bewertungsmechanismus, welcher aus einer Bewertungsstruktur und einer darauf basierenden Bewertungsfunktion besteht, durch die solche Finanzderivate bewertet werden können.

Wir entwarfen die Implementierungen unseres Konstruktionskonzepts und unseres Bewertungsmechanismus in der funktionalen Programmiersprache `Haskell`, da die funktionale Denkweis bei unserem Projekte eine wichtige Rolle spielt.

Ein entscheidender Vorteil unseres gesamten Konzepts ist es, komplexe Optionen und andere Finanzderivate mit mehreren Eigenschaften flexibel bewerten zu können. Die Flexibilität des Bewertungskonzepts liegt darin, dass es keinen direkten Zusammenhang zwischen der Bewertungsfunktion und den zu bewertenden Optionen gibt. Die zu bewertenden Optionen hängen nur von den primitiven Konstruktoren ab. Wenn man eine Option mithilfe der primitiven Konstruktoren konstruiert und dem Bewertungsmechanismus übergeben hat, dann erfolgt die Bewertung der Option Konstruktor für Konstruktor automatisch durch die Bewertungsfunktion. Den fairen Preis der Option erhält man erst nach der letzten diskontierten Bewertung des primitiven Konstruktors `Get`. Das bedeutet, dass man sich wegen der Allgemeinheit des zugrunde liegenden Algorithmus bei der Bewertung einer Option lediglich mit ihrer Konstruktion befassen muss.

Natürlich ist unser Algorithmus nicht makellos. Die Erreichung der Allgemeinheit des vorgenannten Algorithmus geht damit einher, dass andere wichtige Eigenschaften des Algorithmus, wie etwa Genauigkeit und Effizienz, beeinträchtigt werden, wobei wir mehr Wert auf die Genauigkeit als auf die Effizienz legten. Im Hinblick auf die numerischen Beispiele in den Abschnitten 4.3 und 5.3 ist die Genauigkeit unseres Algorithmus vollkommen akzeptabel. Allerdings können wir die Geschwindigkeit unseres Algorithmus nicht garantieren. In den meisten Fällen bewertet unser allgemeiner Algorithmus eine Option langsamer als ein

normaler individueller Algorithmus. Eine andere Ursache der Ineffizienz liegt in der `Haskell`-Sprache selbst. Da ein `Haskell`-Programm nicht so übersichtlich ist wie eine gängige Programmiersprache (z. B. `C++` oder `JAVA`), ist ein `Haskell`-Code sehr schwer zu analysieren und zu optimieren. Neben der Ineffizienz hat unser Algorithmus noch den Nachteil eines großen Bedarfs an Speicherplatz, insbesondere im multidimensionalen Projekt.

Die zukünftige Arbeit besteht darin, unsere `Haskell`-Codes durch moderne Softwaretechniken zu optimieren und zu beschleunigen, neben dem Binomialmodell die Monte-Carlo-Simulation in das Bewertungskonzept mitaufzunehmen und einen einheitlichen und generelleren Entwurf für die Konstruktion von Finanzderivaten zu entwickeln. Diese Forschungsaufgaben wurden von unserem Kooperationspartner, der AG Softwaretechnik an der TU Kaiserslautern übernommen. Den Fortschritt ihrer Forschung kann man dem Artikel [9] von Gaillourdet entnehmen.

Teil II.

Monte-Carlo-Simulation des Deltas und (Cross-)Gammas von Bermuda-Swaptions im LIBOR-Marktmodell

1. Einführung in Teil II

Zinsderivate wurden im Laufe der Jahre immer komplexer und exotischer. Es handelt sich hierbei um Finanzderivate, deren Basiswert durch einen Zinssatz oder eine zinsbezogene Größe gegeben ist. Der Reiz und die Herausforderung der Zinsderivate liegt in der Modellierung einer gesamten Zinskurve und der Bewertung und Risikoschätzung von exotischen Zinsderivaten, welche von der Dynamik der gesamten Zinskurve abhängen. Historisch wurde bei der Modellierung der Zinskurve der Weg über die Modellierung der Short-Rates gewählt. Im Gagensatz dazu verwenden wir das LIBOR-Marktmodell von Brace, Gatarek und Musiela [4] und Miltersen, Sandmann und Sondermann [26]. Dies ist ein hoch-dimensionales, hoch-parametrisches und hoch-flexibles Modell, welches die Zinskurve in Form von Forward-Rates diskretisiert und als ganzes modelliert. Trotz seiner Komplexität ist das LIBOR-Marktmodell in seiner Grundform ein einfaches Modell. Es handelt sich hierbei nämlich um die gleichzeitige Betrachtung mehrerer modifizierter Black-Scholes-Modelle unter einem gemeinsamen Maß. Der Vorteil des LIBOR-Marktmodells liegt darin, dass die stochastischen Prozesse anhand der am Markt beobachtbaren Größen modelliert und kalibriert werden können.

Eines der gängigsten exotischen Zinsderivate ist die Bermuda-Swaption, welche zu den Callable-LIBOR-Exotics gehört, die eine Klasse von exotischen Zinsderivaten mit einer einzigen Währung sind. Diese Klasse von Zinsderivaten ist in der Praxis gang und gäbe. Bei einem Callable-LIBOR-Exotic handelt es sich um eine Option vom Bermuda-Stil, der ausübbare Verträge auf festgelegte und variable Zinssätze zugrunde liegen. Bermuda-Swaptions werden sowohl zur Absicherung gegen Zinsänderungsrisiken als auch als Spekulationsinvestment genutzt. Es handelt sich um eine Option, deren Inhaber das Recht hat, zu einem der festgelegten aufeinanderfolgenden Zeitpunkte in einen Zinsswap einzutreten. Ein Zinsswap ist ein Zinsderivat, bei dem zwei Vertragsparteien vereinbaren, zu bestimmten zukünftigen Zeitpunkten Zinszahlungen auf einen festgelegten Nennwert auszutauschen. Die Zinszahlungen werden meist so festgesetzt, dass eine Partei einen bei Vertragsabschluss fixierten Festzinssatz zahlt, die andere Partei hingegen einen variablen Zinssatz. Der variable Zinssatz orientiert sich an den üblichen Referenzzinssätzen im Interbankengeschäft, z. B. die London-Interbank-Offered-Rate (kurz LIBOR).

Diese Arbeit bezieht sich auf ein zentrales Problem in der Finanzmathematik, nämlich die Bestimmung der Risikoparameter von Finanzderivaten. Risikoparameter sind die Sensitivitätskennzahlen des Preises eines Finanzderivats, welche mit den griechischen Buchstaben abgekürzt und daher auch Greeks genannt

werden. Es gibt eine Reihe von Risikoparametern wie etwa Delta, Vega, Gamma, Theta, Rho u. s. w.. Jeder dieser Greeks misst eine andere Dimension des Risikos eines Finanzderivats. Sie werden in einem Handelsumfeld zur Bestimmung der Hedge-Paramter verwendet. Das Ziel eines Händlers besteht hierbei darin, die Risikoparameter so zu steuern, dass alle Risiken akzeptabel bleiben. Mathematisch sind Risikoparameter die partiellen Ableitungen des Preises eines Finanzderivates nach Modellparametern, z. B. dem Anfangskurs des Basiswertes oder der Volatilität. Sie beschreiben, wie das Finanzderivat unter dem angenommenen Modell auf infinitesimale Parameteränderungen reagiert. Wir konzentrieren uns in dieser Arbeit auf das Delta, welches die Sensitivität des Preises gegenüber dem Anfangskurs des Basiswertes zeigt, und das Gamma, welches die Sensitivität des Preises gegenüber dem Delta misst.

Bei einfachen Finanzderivaten, deren Preise durch die Formeln von Black und Scholes [3] explizit gegeben sind, können wir die Risikoparameter durch die partiellen Ableitungen ihrer Black-Scholes-Formeln analytisch berechnen. Im Gegensatz dazu ist die Einschätzung der Risikoparameter von komplexeren Finanzderivaten stets eine herausfordernde Aufgabe, die meistens nur mithilfe der Monte-Carlo-Simulation zu lösen ist, was bereits in vielen wissenschaftlichen Arbeiten behandelt worden ist. Dazu gibt es laut Glasserman [11] und R. Korn, E. Korn und Kroisandt [18] im allgmeinen drei auf der Monte-Carlo-Simulation basierende Methoden: die Finite-Differenzen-Methode, die Pathwise-Methode und die Likelihood-Ratio-Methode. Die Finite-Differenzen-Methode ist ein grobes Schätzverfahren und relativ einfach zu implementieren. Die Pathwise-Methode entspricht hingegen einer komplizierten und präzisen Simulationsmethode. Sie stellt Anforderungen an Stetigkeit und Differenzierbarkeit von Auszahlungsfunktionen. Für die Berechnung der Risikoparameter außer Gamma gibt es mittlerweile zwei Arten der Pathwise-Methode. Diese sind die von Glasserman und Zhao [12] entwickelte Forward-Methode und die von Giles und Glasserman [10] entwickelte Adjoint-Methode. Die Adjoint-Methode ist modern und unter bestimmten Umständen noch effizienter als die Forward-Methode. Die Likelihood-Ratio-Methode bietet einen alternativen Ansatz zur Schätzung der Risikoparameter. Sie erfordert hingegen keine Stetigkeit der Auszahlungsfunktion und ergänzt damit die Pathwise-Methode.

Das Hauptthema unserer Arbeit ist es, den Delta-Vektor und die Gamma-Matrix einer Bermuda-Swaption zu berechnen. Die Voraussetzung zur Berechnung der Risikoparameter von Bermuda-Swaptions ist ihre Bewertung bzw. die Bestimmung ihrer optimalen Ausübungszeit, was sich im LIBOR-Marktmodell ursprünglich recht problematisch gestaltete, weil das LIBOR-Marktmodell vom Prinzip her auf der Monte-Carlo-Simulation aufbaut, bei der es schwierig ist, über die vorzeitige Ausübung entlang jedes Pfads zu entscheiden. In der vorliegenden Arbeit können wir jedoch auf den Kleinste-Quadrate-Monte-Carlo-Ansatz zurückgreifen. Dieses Verfahren wurde von Longstaff und Schwarz [25] entwickelt und orientiert sich an den bisher bekannten Methoden, die bereits

bei Gitterverfahren verwendet werden. Die Berechnung beginnt am letztmöglichen Ausübungszeitpunkt der Bermuda-Swaption und wird rückwärts rekursiv in der Zeit ausgeführt. Zu jedem möglichen Ausübungszeitpunkt ist eine Optimierungsbedingung auszuführen und zwar analog wie bei den Gitterverfahren: Vergleiche den Wert beim Ausüben der Bermuda-Swaption mit dem risikoneutralen Wert. Dieser risikoneutrale Wert ist ein bedingter Erwartungswert und muss entsprechend in der Simulation berechnet werden, wobei die Methode der kleinsten Quadrate in Anspruch genommen wird.

Mithilfe des Kleinste-Quadrate-Monte-Carlos hat Piterbarg [32] die Forward-Methode zur exakten Berechnung der Delta-Vektoren und Gamma-Matrizen von Bermuda-Swaptions erfolgreich entwickelt. Eine viel effizientere Umsetzung dazu wurde kürzlich von Leclerc, Liang und Schneider [22] generalisiert. Diese Formulierung benutzt die Adjoint-Methode in Hinblick auf linear algebraische Operationen zur Reduzierung der Ordnung der Zeitkomplexität. Basierend auf den Pionierarbeiten entwickeln wir in diesem Artikel drei weitere, neue Methoden zur Berechnung des Delta-Vektors einer Bermuda-Swaption, welche bisher noch nicht veröffentlicht wurden. Die erste Methode ist eine neue Version der Adjoint-Methode, die ebenso exakt und effizient ist wie die Adjoint-Methode von Leclerc, Liang und Schneider [22]. Sie ist jedoch relativ einfach zu verstehen und zu implementieren. Die zweite Methode ist die Pathwise-Methode im Rahmen einer vereinfachenden Simulation des LIBOR-Marktmodells mit Bias. Diese Methode ist schnell und sehr leicht zu implementieren. Ihre Ergebnisse sind auch exakt genug, obgleich sie bei der Simulation Bias erzeugt. Die dritte Methode ist die Likelihood-Ratio-Methode im Rahmen der oben-genannten inexakten Simulation des LIBOR-Marktmodells, deren Ergebnisse im Gegensatz zu den anderen Methoden nicht genügend präzise sind, weil sie neben Bias noch eine große Varianz erzeugt. Allerdings ist sie effizient und kann zur Berechnung der Delta-Vektoren von amerikanischen oder bermudischen Zinsderivaten mit nicht-stetigen Auszahlungen beitragen.

Darüber hinaus entwikeln wir in dieser Arbeit noch zwei innovative Methoden, um die (Cross-)Gammas bzw. die Gamma-Matrix von Bermuda-Swaptions genau zu berechnen, was völlig neu im Forschungsgebiet der Computational-Finance ist. Eine der Methoden ist die Kombination der Finite-Differenzen-Methode und dem Pathwise-Delta. Dieses Verfahren ist relativ effizient, aber empfindlich und erzeugt Bias. Die andere Methode ist die Pathwise-Methode, welche einem robusten Simulationsverfahren ohne Bias entspricht. Diese zweite Methode stellt den größten Durchbruch dieser Arbeit dar. Auch hier besteht das Pathwise-(Cross-)Gamma aus dem Forward-(Cross-)Gamma und dem Adjoint-(Cross-)Gamma. Das Adjoint-(Cross-)Gamma ist viel komplizierter als das Forward-(Cross-)Gamma, aber dennoch viel effizienter.

Die Arbeit gliedert sich wie folgt. In Kapitel 2 diskutieren wir die Finite-Differenzen-Methode, Pathwise-Methode und Likelihood-Ratio-Methode für allgemeine Finanzderivate. In Kapitel 3 führen wir zunächst die Simulation des

LIBOR-Marktmodells ein. Dabei betrachten wir neben der exakten Simulation noch eine entsprechende inexakte Simulation, welche durch die Forward-Drift, eine kleine Veränderung der Driftterme vom exakten Fall, realisiert wird. Danach erörtern wir Pathwise-Delta (Forward-Delta und Adjoint-Delta), Pathwise-Delta unter Forward-Drift, Likelihood-Ratio-Delta und Pathwise-(Cross-)Gamma (Forward-(Cross-)Gamma und Adjoint-(Cross-)Gamma) für Zinsderivate. In Kapitel 4 definieren wir die Bermuda-Swaption, bzw. ihre Notationen, formell und stellen das Kleinste-Quadrate-Monte-Carlo vor. Anschließend diskutieren wir unser Hauptthema, die Berechnung der Delta-Vektoren und der Gamma-Matrizen von Bermuda-Swaptions. Die entwickelten Algorithmen überprüfen und vergleichen wir anhand einiger numerischer Beispiele in Kapitel 5.

2. Berechnung des Deltas und (Cross-)Gammas von Finanzderivaten

Unter dem Delta, welches mit dem griechischen Buchstaben $\mathbf{\Delta}$ bezeichnet wird, versteht man die erste partielle Ableitung des Preises eines Finanzderivats nach dem Anfangskurs des Basiswertes. Unter dem Gamma, welches mit dem griechischen Buchstaben $\mathbf{\Gamma}$ bezeichnet wird, versteht man die zweite partielle Ableitung des Preises eines Finanzderivats nach dem Anfangskurs des Basiswertes. Gamma misst die Sensitivität des Deltas bzgl. des Anfangskurses.

Um die Risikoparameter eines Finanzderivats im Rahmen der Monte-Carlo-Simulation zu bestimmen, gibt es laut Glasserman [11] und R. Korn, E. Korn und Kroisandt [18] im Allgemeinen drei wichtige Methoden. Diese sind die Finite-Differenzen-Methode, die Pathwise-Methode und die Likelihood-Ratio-Methode. Die Finite-Differenzen-Methode ist ein grobes und einfaches Schätzverfahren. Die Pathwise-Methode ist meistens exakt und effizient unter der Einschränkung, dass die Auszahlungsfunktion des betrachteten Finanzderivats genügend glatt sein muss. Die Likelihood-Ratio-Methode ist ebenso effizient wie die Pathwise-Methode und kann noch Finanzderivate mit nicht-stetigen Auszahlungsfunktionen gut behandeln. Jedoch kann bei der Likelihood-Ratio-Methode eine größere Varianz entstehen. In diesem Kapitel legen wir das Hauptgewicht auf die Pathwise-Methode und die Likelihood-Ratio-Methode, die den exakten Simulationsmethoden ohne Bias entsprechen. Zunächst beginnen wir jedoch mit der Finite-Differenzen-Methode.

2.1. Finite-Differenzen-Methode

Anhand der Finite-Differenzen-Methode können wir eine grobe Schätzung des Deltas erhalten. Unter der Voraussetzung, dass der Wert eines Finanzderivats analytisch berechnet und numerisch simuliert werden kann, ist diese Methode einfach zu realisieren. Hier wollen wir ihr Prinzip kurz und knapp erläutern.

Es sei $V(\theta)$ der Wert eines Finanzderivats, wobei θ der Anfangskurs des Basiswertes ist. Dann entspricht die Formel

$$\mathbf{\Delta}\left(V(\theta)\right) = \frac{\partial V(\theta)}{\partial \theta}$$

dem Delta des Finanzderivats. Wir können diese partielle Abteilung durch den

sogenannten Vorwärtsdifferenzenquotient

$$\frac{\partial V(\theta)}{\partial \theta} \overset{Vor.}{\leftarrow} \frac{V(\theta+\epsilon)-V(\theta)}{\epsilon},$$

den Rückwärtsdifferenzenquotient

$$\frac{\partial V(\theta)}{\partial \theta} \overset{Rück.}{\leftarrow} \frac{V(\theta)-V(\theta-\epsilon)}{\epsilon}$$

und den zentralen Differenzenquotient

$$\frac{\partial V(\theta)}{\partial \theta} \overset{Zent.}{\leftarrow} \frac{V(\theta+\epsilon)-V(\theta-\epsilon)}{2\epsilon} l$$

für eine hinreichend kleine Maschenweite $\epsilon > 0$ approximieren, wobei $V(\theta), V(\theta+\epsilon)$ und $V(\theta-\epsilon)$ mit der gleichen Folge von Zufallszahlen simuliert werden müssen, wenn dies Finanzderivat durch die Monte-Carlo-Simulation bewertet wird. Da alle drei Quotienten im Grenzwert $\epsilon \to 0$ gegen das Delta $\mathbf{\Delta}(V(\theta))$ konvergieren, können wir für hinreichend kleines ϵ eine gute Approximation erhalten. Unter anderem erzeugt der zentrale Differenzenquotient normalerweise Bias mit kleinerer Ordnung als die anderen beiden, er ist der exakter.

Die Finite-Differenzen-Methode zur Berechnung der (Cross-)Gammas von Finanzderivaten funktioniert wegen des großen Bias nomalerweise nicht exakt genug, insbesondere von komplexen Finanzderivaten. Allerdings gelingt es uns im Abschnitt 4.3, die (Cross-)Gammas einer Bermuda-Swaption durch eine modifizierte Finite-Differenzen-Methode hinreichend genau zu berechnen.

2.2. Pathwise-Methode

In diesem Abschnitt stellen wir die Pathwise-Methode vor. Vorab erörtern wir knapp ihre Voraussetzungen. Es sei $g(X(T))$ die diskontierte Auszahlungsfunktion eines Finanzderivats zur heutigen Zeit mit dem Basiswert $X(t) = (X_1(t), \ldots, X_M(t))^\top$ für $t \in [0,T]$, Der faire Preis des Finanzderivats entspricht dem Erwartungswert $\mathbb{E}(g(X(T)))$. Zur Bestimmung des Deltas ist dann die partielle Ableitung

$$\frac{\partial \mathbb{E}(g(X(T)))}{\partial X(0)}$$

zu schätzen, wobei $X(0)$ der Anfangskurs des Basiswertes ist. Die Pathwise-Methode schätzt diese in zwei Schritten. Zuerst berechnet man

$$\frac{\partial g(X(T))}{\partial X(0)}$$

entlang jedes simulierten Pfades, dann berechnet man den Mittelwert über alle Pfade. Daraufhin ist der Schätzwert des Deltas mit der Anwendung der Pathwise-Methode genau dann erwartungstreu, wenn

$$\frac{\partial \mathbb{E}(g(X(T)))}{\partial X(0)} = \mathbb{E}\left(\frac{\partial g(X(T))}{\partial X(0)}\right) \tag{2.1}$$

gilt. Die hinreichende Bedingung für (2.1) ist laut Glasserman [11] wenigstens die Lipschitz-Stetigkeit der diskontierten Auszahlungsfunktion g. Genauer:

1. Für jedes $\theta \in \Theta$ existiert $\partial X_i(\theta)/\partial\theta$ fast sicher für alle $i = 1, \ldots, M$.
2. g ist differenzierbar an den Punkten $X(\theta) \in \mathbb{R}^M$ fast sicher für alle $\theta \in \Theta$.
3. Es gibt eine konstante Variable l_g, so dass für alle $x, y \in \mathbb{R}^M$

   $$|g(x) - g(y)| \leq l_g \|x - y\|$$

   gilt, d. h. g ist Lipschitz-stetig.
4. Es existieren Zufallsvariablen ϑ_i für $i = 1, \ldots, M$, so dass für alle $\theta_1, \theta_2 \in \Theta$

   $$|X_i(\theta_2) - X_i(\theta_1)| \leq \vartheta_i |\theta_2 - \theta_1|$$

   und $\mathbb{E}(\vartheta_i) < \infty$ gelten mit $i = 1, \ldots, M$.

Deshalb eignet sich die Pathwise-Methode für die Berechnung des Deltas und der anderen Risikoparameter mit partieller Ableitung erster Ordnung bzgl. der Finanzderivate mit Lipschitz-stetiger Auszahlung. Im Vergleich zum Delta erscheint die Berechnung des Gammas komplizierter. Dazu müssen wir die entsprechende partielle Ableitung zweiter Ordnung

$$\frac{\partial^2 \mathbb{E}\left(g(X(T))\right)}{\partial X(0)^2}$$

bestimmen. Die Voraussetzung eines erwartungstreuen Schätzwerts vom Gamma unter der Anwendung der Pathwise-Methode ist dann

$$\frac{\partial^2 \mathbb{E}\left(g(X(T))\right)}{\partial X(0)^2} = \mathbb{E}\left(\frac{\partial^2 g(X(T))}{\partial X(0)^2}\right),$$

d. h. die Auszahlungsfunktion g muss bzgl. $X(0)$ zweimal stetig differenzierbar sein. Diese Bedingung erfüllt die Auszahlungsfunktion eines Finanzderivats normalerweise nicht. Allerdings werden wir im Kapitel 4 sehen, wie die Pathwise-Methode für Gamma einer Bermuda-Swaption in einem speziellen Algorithmus trotz der komplexen Auszahlungen gut funktioniert.

Mittlerweile gibt es zwei bekannte Arten der Pathwise-Methode. Die klassische Pathwise-Methode ist die sogenannte Forward-Methode, die in der Arbeit von Glasserman und Zhao [12] ausführlich vorgestellt wurde. Es handelt sich um eine vorwärtige Rekursion. Eine alternative Pathwise-Methode ist die Adjoint-Methode, deren Einzelheiten im Artikel von Giles und Glasserman [10] veröffentlicht wurden. Sie beschäftigt sich mit einer rückwärtigen Rekursion. Die Adjoint-Methode ist jünger und unter einigen Bedingungen, die wir später erläutern werden, noch effizienter als die Forward-Methode.

2.2.1. Forward-Delta

Unsere Diskussion beginnt mit der Forward-Methode zur Berechnung des Deltas. Zunächst untersuchen wir das allgemeine Berechnungsverfahren für das Delta, das nicht nur für Zinsderivate, sondern auch für andere Finanzderivate, z. B. Aktienderivate, gültig ist.

Sei $g(X(T))$ eine Lipschitz-stetige diskontierte Auszahlungsfunktion eines Finanzderivats mit der Fälligkeit T und dem Basiswert $X(t) \in \mathbb{R}^M$ für $0 \leq t \leq T$. Dabei erfüllt der Prozess $X(t)$ die stochastische Differentialgleichung

$$dX(t) = a(X(t))dt + b^\top(X(t))dW(t) \qquad t \in [0,T], \tag{2.2}$$

wobei $a(X(t)) \in \mathbb{R}^M$, $b(X(t)) \in \mathbb{R}^{M\times d}$ und W ein d-dimensionaler Wiener-Prozess ist. Durch die Euler-Diskretisierung auf das Zeitgitter $0 = t_0 < t_1 < \cdots < t_N = T$ kann die stochastische Differentialgleichung (2.2) wie folgt approximiert werden:

$$X(n+1) = X(n) + a(X(n))h_n + b^\top(X(n))Z(n+1)\sqrt{h_n} \tag{2.3}$$

für $n = 0,\ldots,N-1$, wobei $h_n = t_{n+1} - t_n$ und $Z(1),\ldots,Z(N) \sim \mathcal{N}(0, I_d)$ unabhängig sind. In der Simulationsformel (2.3) vereinbaren wir die Notation $X(n) = X(t_n)$. Wir können $X(n+1)$ für eine Funktion der Variablen $X(n)$ halten und setzen dann

$$\Delta(n) = \frac{\partial X(n)}{\partial X(0)} \qquad \Delta_{ij}(n) = \frac{\partial X_i(n)}{\partial X_j(0)}$$

für $n = 0,\ldots,N-1$ und $i,j = 1,\ldots,M$. Wir leiten dann die Relation zwischen $\Delta(n+1)$ und $\Delta(n)$ ab:

$$\Delta(n+1) = \frac{\partial X(n+1)}{\partial X(0)} = \frac{\partial X(n+1)}{\partial X(n)} \cdot \frac{\partial X(n)}{\partial X(0)} = \frac{\partial X(n+1)}{\partial X(n)} \cdot \Delta(n)$$

für $n = 0,\ldots,N-1$. Genauer ist

$$\begin{aligned}\Delta_{ij}(n+1) &= \Delta_{ij}(n) + \sum_{k=1}^{M} \frac{\partial a_i(X(n))}{\partial X_k(n)} \Delta_{kj}(n) h_n \\ &\quad + \sum_{l=1}^{d}\sum_{k=1}^{M} \frac{\partial b_{il}(X(n))}{\partial X_k(n)} \Delta_{kj}(n) Z_l(n+1)\sqrt{h_n}\end{aligned}$$

für $n = 0,\ldots,N-1$ und $i,j = 1,\ldots,M$. Ferner setzen wir für $n = 0,\ldots,N-1$

$$D(n) = \frac{\partial X(n+1)}{\partial X(n)},$$

die $(M \times M)$-Matrizen mit den Termen:

$$\begin{aligned}D_{ij}(n) &= \frac{\partial X_i(n+1)}{\partial X_j(n)} \\ &= \mathbf{1}\{i=j\} + \frac{\partial a_i(X(n))}{\partial X_j(n)} h_n + \sum_{l=1}^{d} \frac{\partial b_{il}(X(n))}{\partial X_j(n)} Z_l(n+1)\sqrt{h_n}\end{aligned}$$

für $i, j = 1, \ldots, M$, wobei $\mathbf{1}$ die Indikatorfunktion bezeichnet mit

$$\mathbf{1}\{A\} = \begin{cases} 1 & : \quad A = True \\ 0 & : \quad A = False. \end{cases}$$

Dann können wir $\Delta(N)$ durch die folgende Rekursion numerisch berechnen:

$$\Delta(n+1) = D(n)\Delta(n) \qquad \Delta(0) = I_M \tag{2.4}$$

für $n = 0, \ldots, N-1$. Weil wir bei jedem Rekursionsschritt zwei Matrizen multiplizieren müssen, entsprechen die Zeitkosten pro Rekursionsschritt $O\left(M^3\right)$.

Wir betrachten jetzt ein Finanzderivat mit der diskontierten Auszahlungsfunktion $g(X(T))$. Die Simulationsformel vom Delta des Finanzderivats ist

$$\begin{aligned} \boldsymbol{\Delta}_j \left(g(X(N)\right) &= \frac{\partial g(X(N))}{\partial X_j(0)} \\ &= \sum_{i=1}^{M} \frac{\partial g(X(N))}{\partial X_i(N)} \cdot \frac{\partial X_i(N)}{\partial X_j(0)} \\ &= \sum_{i=1}^{M} \frac{\partial g(X(N))}{\partial X_i(N)} \cdot \Delta_{ij}(N) \end{aligned}$$

für $j = 1, \ldots, M$. Und die Simulationsformel des ganzen Delta-Vektors lautet

$$\boldsymbol{\Delta}\left(g(X(N)\right) = \frac{\partial g(X(N))}{\partial X(0)} = \frac{\partial g(X(N))}{\partial X(N)} \cdot \Delta(N). \tag{2.5}$$

Auf der Gleichung (2.5) und der Rekursion (2.4) basiert die Grundidee der Forward-Methode zur Berechnung des Deltas des Finanzderivats. Wir bemerken, dass wir wegen (2.5) den Delta-Vektor von beliebig vielen Finanzderivaten mit dem gleichen Basiswert X bewerten können, wenn $\Delta(N)$ schon einmal bestimmt ist.

2.2.2. Adjoint-Delta

Für die Adjoint-Methode werden weiterhin dieselben Voraussetzungen und Notationen wie bei der Forward-Methode verwendet. Wir beginnen mit der Adjoint-Methode zur Berechnung des Deltas eines allgemeinen Finanzderivats. Wie wir schon erwähnt haben, kann die Rekursionsrichtung umgekehrt werden, sodass eine effizientere Berechnung des Deltas möglich ist. Wir strukturieren zunächst

die Simulationsformel (2.5) bzgl. der Berechnung des Deltas um:

$$\begin{aligned}
\boldsymbol{\Delta}\left(g(X(N))\right) &= \frac{\partial g(X(N))}{\partial X(0)} \\
&= \frac{\partial g(X(N))}{\partial X(N)} \cdot \Delta(N) \\
&= \underbrace{\frac{\partial g(X(N))}{\partial X(N)} \cdot D(N-1)D(N-2)\cdots D(0)}_{=V^\top(0)} \cdot \Delta(0) \\
&= V^\top(0) \cdot \Delta(0) \\
&= V^\top(0), \qquad (2.6)
\end{aligned}$$

wobei $V(0)$ offenbar durch die folgende Rekursion auf das Zeitgitter $\{t_0, \ldots, t_N\}$ rückwärts berechnet werden kann:

$$V(n) = D^\top(n)V(n+1) \qquad V(N) = \left(\frac{\partial g(X(N)}{\partial X(N)}\right)^\top, \qquad (2.7)$$

dabei $n = N-1, \ldots, 0$. (2.6) und (2.7) bilden zusammen die Grundidee der Adjoint-Methode zur Berechnung des Deltas.

Wir bemerken, dass bei der Adjoint-Methode eine Vektor-Rekursion, nämlich $V(\cdot) \in \mathbb{R}^M$, mit dem Zeitaufwand $O(M^2)$ per Rekursionsschritt durchzuführen ist, während bei der Forward-Methode eine Matrix-Rekursion mit dem Zeitaufwand $O(M^3)$ per Rekursionsschritt durchzuführen ist. Die Adjoint-Methode ist deshalb vorteilhaft, wenn wir uns mit der Berechnung des Deltas einer einzelnen Auszahlungsfunktion $g(X(T))$ in Bezug auf mehrdimensionale Veränderungen von $X(0)$ beschäftigen, z. B. wenn wir die Sensitivitäten bzgl. $X_i(0)$ für alle $i = 1, \ldots, M$ benötigen. Außerdem muss der Drift-Term des Basiswertes $X(t)$ hinreichend kompliziert sein, sodass die Matrix $D(n)$ zumindest keine Einheitsmatrix ist, z. B. beim LIBOR-Marktmodell, sonst ist die Anwendung der Adjoint-Methode genau so effizient wie die Forward-Methode, also sinnlos, denn die Adjoint-Methode ist komplizierter zu implementieren.

Zusammenfassend ergibt sich, dass, falls die Matrix $D(n)$ hinreichend dicht besetzt ist, sich die beiden oben besprochenen Pathwise-Methdoden hinsichtlich ihrer Effizienz wie folgt unterscheiden:

- Die Forward-Methode ist effizienter für die Berechnung der Risikoparameter, wenn das betrachtete Portfolio aus vielen Instrumenten besteht, die nur von wenigen Parametern beeinflusst werden.

- Die Adjoint-Methode ist effizienter für die Berechnung der Risikoparameter, wenn das betrachtete Portfolio aus wenigen Instrumenten besteht, die von vielen Parametern beeinflusst werden.

Es ist noch zu berücksichtigen, dass die Adjoint-Methode deutlich mehr Speicherplatz als die Forward-Methode verlangt. Dies liegt daran, dass die Adjoint-Methode in der Tat eine Rückwärtsrekursion ist und wir bei der Adjoint-Methode zuerst den simulierten Basiswert $X(t)$ von t_0 bis t_N vorwärts entwickeln und alle einschlägigen Daten abspeichern müssen, um anschließend die Matrizen $D(N-1),\ldots,D(0)$ rückwärts zu berechnen und die rückwärtige Rekursion erfolgreich ausführen zu können. Deshalb wird die Adjoint-Methode auch die Backward-Methode genannt.

2.2.3. Forward-(Cross-)Gamma

In diesem und dem nächsten Unterabschnitt untersuchen wir die Pathwise-Methode zur Berechnung des Gammas von allgemeinen Finanzderivaten, was Giles und Glasserman [10] auch kurz erwähnt haben. Dazu benutzen wir dieselben Notationen wie bei der Untersuchung des Deltas. Die Pathwise-Methode für Gamma ist komplizierter und zeitlich aufwändiger als die Pathwise-Methode für Delta. Zunächst müssen wir annehmen, dass die diskontierte Auszahlungsfunktion $g(X(N))$ nach $X(0)$ zweimal stetig differenzierbar ist. Die Sensitivität zweiter Ordnung, nämlich das Gamma, lautet

$$\begin{aligned} & \mathbf{\Gamma}\left(g(X(N))\right) \\ = \; & \frac{\partial^2 g(X(N))}{\partial X(0)^2} \\ = \; & \sum_{i=1}^{M}\left(\frac{\partial g(X(N))}{\partial X_i(N)}\right)\Gamma^i(N) + \Delta^\top(N)\left(\frac{\partial^2 g(X(N))}{\partial X(N)^2}\right)\Delta(N), \end{aligned} \tag{2.8}$$

wobei

$$\Gamma^i(n) = \frac{\partial^2 X_i(n)}{\partial X(0)^2}$$

eine Matrix mit den Einträgen

$$\Gamma^i_{jk}(n) = \frac{\partial^2 X_i(n)}{\partial X_j(0)\partial X_k(0)}$$

für $n = 0,\ldots,N$ und $j,k = 1,\ldots,M$ ist. Das Gamma ist mathematisch gesehen eine $(M \times M)$-Matrix mit den Einträgen $\mathbf{\Gamma}_{jk}\left(g(X(N))\right.$ für $j,k = 1,\ldots,M$. Dabei nennen wir jeden Term $\mathbf{\Gamma}_{jk}\left(g(X(N))\right.$ für $j \neq k$ auch das Cross-Gamma.

Die Berechnung von $\Delta(N)$ ist bereits in der Diskussion des Deltas geklärt worden. So bleibt in (2.8) nur noch $\Gamma^i(N)$ zu berechnen. Durch zweimaliges Differenzieren der beiden Seiten der Euler-Diskretisierung (2.3) des Basiswertes oder einfache Ableitung der beiden Seiten der Rekursion (2.4) erhalten wir die Rekursionsformel zur Berechnung von $\Gamma^i(N)$ für $i = 1,\ldots,M$. Sie ist

$$\Gamma^i(n+1) = \sum_{l=1}^{M} D_{il}(n)\Gamma^l(n) + \Delta^\top(n)E^i(n)\Delta(n) \tag{2.9}$$

für $n = 0, \ldots, N-1$ mit der Initialisierung $\Gamma^i(0) = \mathbf{0} \in \mathbb{R}^{M \times M}$, wobei

$$E^i(n) = \frac{\partial^2 X_i(n+1)}{\partial X(n)^2}$$

eine Matrix mit den Einträgen

$$E^i_{jk}(n) = \frac{\partial^2 X_i(n+1)}{\partial X_j(n) \partial X_k(n)}$$

für $n = 0, \ldots, N$ und $j, k = 1, \ldots, M$ ist. Die Simulationsformel (2.8) und die Rekursionsformel (2.9) bilden zusammen das Prinzip der Pathwise-Methode zur Berechnung der (Cross-)Gammas. Wegen der vorwärtigen Simulationsrichtung bzgl. des Zeitgitters $\{t_0, \ldots, t_N\}$ in Rekursionsformel (2.9) entspricht dieses Verfahren einer Forward-Methode.

2.2.4. Adjoint-(Cross-)Gamma

Jetzt diskutieren wir die Adjoint-Methode zur Berechnung der (Cross-)Gammas von allgemeinen Finanzderivaten. Dafür schreiben wir die andere Dimension der Gleichung (2.9)

$$\Gamma_{jk}(n+1) = D(n)\Gamma_{jk}(n) + \begin{pmatrix} \Delta_{\cdot j}^\top(n) E^1(n) \Delta_{\cdot k}(n) \\ \vdots \\ \Delta_{\cdot j}^\top(n) E^M(n) \Delta_{\cdot k}(n) \end{pmatrix} \tag{2.10}$$

für $n = 0, \ldots, N-1$ und $j, k = 1, \ldots, M$, wobei $\Delta_{\cdot j}(n)$ j-te Spalte und $\Delta_{\cdot k}(n)$ k-te Spalte der Matrix $\Delta(n)$ ist und

$$\Gamma_{jk}(n) = \frac{\partial^2 X(n)}{\partial X_j(0) \partial X_k(0)}.$$

Für die Adjoint-Methode setzen wir weiter die beiden Vektoren

$$G(n) = \Gamma_{jk}(n)$$

und

$$C(n) = \begin{pmatrix} \Delta_{\cdot j}^\top(n) E^1(n) \Delta_{\cdot k}(n) \\ \vdots \\ \Delta_{\cdot j}^\top(n) E^M(n) \Delta_{\cdot k}(n) \end{pmatrix}$$

für $n = 0, \ldots, N$ und $j, k = 1, \ldots, M$. Dann wird die Gleichung (2.10) wie folgt umgeschrieben:

$$G(n+1) = D(n)G(n) + C(n)$$

für $n = 0, \ldots, N-1$. Davon gehen wir aus, die Idee des Adjoint-(Cross-)Gammas abzuleiten:

$$\begin{aligned}
& \mathbf{\Gamma}_{jk}\left(g(X(N))\right) \\
= \; & \frac{\partial^2 g(X(N))}{\partial X_j(0)\partial X_k(0)} \\
= \; & \frac{\partial g(X(N))}{\partial X(N)}\Gamma_{jk}(N) + \underbrace{\Delta_{.j}^{\top}(N)\left(\frac{\partial^2 g(X(N))}{\partial X(N)^2}\right)\Delta_{.k}(N)}_{=B_{jk}(g(X(N)))} \\
= \; & V^{\top}(N)G(N) + B_{jk}\left(g(X(N))\right) \\
= \; & V^{\top}(N)(C(N-1) + D(N-1)G(N-1)) + B_{jk}\left(g(X(N))\right) \\
= \; & V^{\top}(N)C(N-1) + V^{\top}(N-1)(C(N-2) + D(N-2)G(N-2)) \\
& + B_{jk}\left(g(X(N))\right) \\
\vdots \; & \\
= \; & V^{\top}(N)C(N-1) + V^{\top}(N-1)C(N-2) + \cdots + V^{\top}(1)C(0) \\
& + B_{jk}\left(g(X(N))\right) \\
= \; & \sum_{n=0}^{N-1} V^{\top}(n+1)C(n) + B_{jk}\left(g(X(N))\right), \qquad (2.11)
\end{aligned}$$

wobei $V(n)$ für $n = 1, \ldots, N$ mittels der Formel (2.7) rekursiv berechnet werden. Die Formel (2.11) entspricht dem Prinzip der Adjoint-Methode zur Berechnung des (Cross-)Gammas eines Finanzderivats. Der einzige Unterschied zwischen dem Forward-(Cross-)Gamma und dem Adjoint-(Cross-)Gamma liegt darin, den Term

$$\frac{\partial g(X(N))}{\partial X(N)}\Gamma_{jk}(N)$$

für jedes Paar (j, k) zu berechnen. Im Forward-(Cross-)Gamma wird der Vektor $\Gamma_{jk}(N)$ durch die N-schrittige Rekursion (2.10) bzw.

$$G(n+1) = D(n)G(n) + C(n)$$

vorwärts berechnet. Und im Adjoint-(Cross-)Gamma ist

$$\frac{\partial g(X(N))}{\partial X(N)}\Gamma_{jk}(N) = \sum_{n=0}^{N-1} V^{\top}(n+1)C(n).$$

D. h. wir müssen $C(n)$ für $n = 0, \ldots, N-1$ in beiden Methoden berechnen. Daneben brauchen wir zusätzlich $O(M^2)$ im Forward-(Cross-)Gamma und $O(M)$ im Adjoint-(Cross-)Gamma für jedes $n = 0, \ldots, N-1$. Denn $D(n)G(n)$ dem Produkt zwischen einer Matrix und einem Vektor entspricht und $V^{\top}(n+1)C(n)$ das Skalarprodukt von zweien Vektoren ist. Dank dieses Tricks spart das Adjoint-(Cross-)Gamma mehr Zeitkosten als das Forward-(Cross-)Gamma. Allerdings verlangt

die Adjoint-Methode bei der Berechnung des (Cross-)Gammas deutlich mehr Speicherplatz als die Forward-Methode wie bei der Berechnung des Deltas.

2.3. Likelihood-Ratio-Methode

Die Likelihood-Ratio-Methode bietet einen alternativen Lösungsansatz ohne Bias zur Schätzung der Risikoparameter eines Finanzderivats. Sie erfordert keine zusätzliche Informationen über die Auszahlungsfunktion, z. B. die Glattheit, und ergänzt damit die Pathwise-Methode. Bei der Likelihood-Ratio-Methode ist statt der Differenziation der Auszahlungsfunktion die Differenziation der Wahrscheinlichkeitsdichte des Basiswertes zu berechnen. Vorab erklären wir kurz das Prinzip.

Es sei $g(X(\theta))$ die diskontierte Auszahlungsfunktion eines Finanzderivats mit dem Basiswert $X(\theta) = (X_1(\theta), \ldots, X_M(\theta))^\top$. Für Delta bezeichnet θ hier den Anfangskurs des Basiswertes. Der Erwartungswert des Finanzderivats entspricht der folgenden Integration

$$\mathbb{E}\left(g(X(\theta))\right) = \int_{\mathbb{R}^M} g(x) f_X(x) dx, \tag{2.12}$$

wobei $f_X(x)$ die Wahrscheinlichkeitsdichte vom Basiswert $X(\theta)$ ist. Wir leiten dann die beiden Seiten der Gleichung (2.12) nach θ ab, um das Delta des Finanzderivats zu berechnen:

$$\begin{aligned}
\frac{\partial \mathbb{E}\left(g(X(\theta))\right)}{\partial \theta} &= \int_{\mathbb{R}^M} g(x) \underbrace{\frac{\partial f_X(x)}{\partial \theta}}_{=f'_X(x)} dx \\
&= \int_{\mathbb{R}^M} \left(g(x) \frac{f'_X(x)}{f_X(x)} \right) f_X(x) dx \\
&= \mathbb{E}\left(g(X(\theta)) \frac{f'_X(X(\theta))}{f_X(X(\theta))} \right).
\end{aligned} \tag{2.13}$$

Aus Formel (2.13) folgt das Prinzip zur Bewerung des Deltas gemäß der Likelihood-Ratio-Methode, welches aus zwei Schritten besteht. Zuerst schätzt man die Kernform

$$g(X(\theta)) \cdot \frac{f'_X(X(\theta))}{f_X(X(\theta))} \tag{2.14}$$

entlang jedes simulierten Pfades, dann berechnet man den Mittelwert über alle Pfade und bekommt das gewünschte Ergebnis. Das ist ähnlich wie das Berechnungsprinzip der Pathwise-Methode. Deshalb fassen wir die Likelihood-Ratio-Methode als eine Variante der Pathwise-Methode auf.

Wegen Formel (2.13) erfordert die Likelihood-Ratio-Methode überhaupt keine Glätte in der diskontierten Auszahlungsfunktion. Das ist ein Vorteil im Vergleich zur Pathwise-Methode. Allerdings verlangt sie, dass die Wahrscheinlichkeitsdichte des Basiswertes $X(\theta)$ bekannt sein muss und darüber hinaus, dass

ihre Ableitung nach dem Parameter θ auch analytisch berechenbar ist. Das ist ein ganz seltener Fall und wird daher als der große Nachteil der Likelihood-Ratio-Methode angesehen. Noch zu betrachten ist, obwohl die Likelihood-Ratio-Methode eine Simulationsmethode ohne Bias ist und für die Finanzderivate mit nicht-stetigen Auszahlungen gut funktioniert, hat sie ein Problem der Genauigkeit bei Finanzderivaten mit glatten Auszahlungen: Die Varianz der Likelihood-Ratio-Methode ist normalerweise größer als die Varianz der Finite-Differenzen-Methode und Pathwise-Methode. Außerdem ist sie wegen zu größe Varianz nicht geeignet für die Berechnung des (Cross-)Gammas von Finanzderivaten.

Nun diskutieren wir die Likelihood-Ratio-Methode zur Berechnung des Deltas anhand eines Beispiels eines Finanzderivats mit einem multivariaten normalverteilten Basiswert, welches Glasserman und Zhao [12] auch schon diskutiert haben. Wir nehmen an, dass $X(\theta) \sim \mathcal{N}(\bar{\mu}(\theta), \Sigma_\theta)$ mit $\bar{\mu}(\theta) \in \mathbb{R}^M$ und $\Sigma_\theta \in \mathbb{R}^{M\times M}$. Die Wahrscheinlichkeitsdichte von $X(\theta)$ ist aus der Statistik bekannt:

$$f_X(x) = \frac{1}{(2\pi)^{\frac{M}{2}} \left(\det \Sigma_\theta\right)^{\frac{1}{2}}} \cdot \exp\left(-\frac{1}{2}(x-\bar{\mu}(\theta))\Sigma_\theta^\top (x-\bar{\mu}(\theta))\right).$$

Der Faktor $f_X'(x)/f_X(x)$ in der Kernform (2.14) sieht in diesem Fall wie folgt aus:

$$\frac{f_X'(x)}{f_X(x)} = \frac{\partial \ln\left(f_X(x)\right)}{\partial\theta} = (x-\bar{\mu}(\theta))^\top \Sigma_\theta^{-1} \cdot \frac{\partial \bar{\mu}(\theta)}{\partial\theta}.$$

Dann gilt

$$\begin{aligned}\frac{\partial \mathbb{E}\left(g(X(\theta))\right)}{\partial\theta} &= \mathbb{E}\left(g(X(\theta))\frac{f_X'(X(\theta))}{f_X(X(\theta))}\right)\\ &= \mathbb{E}\left(g(X(\theta))(X(\theta)-\bar{\mu}(\theta))^\top \Sigma_\theta^{-1} \cdot \frac{\partial\bar{\mu}(\theta)}{\partial\theta}\right). \qquad (2.15)\end{aligned}$$

Der multivariate normalverteilte Basiswert $X(\theta)$ kann durch die folgende Formel simuliert werden:

$$X(\theta) = \bar{\mu}(\theta) + B_\theta \cdot Z_\theta,$$

wobei $Z_\theta \sim \mathcal{N}(0, I_d)$ und $B_\theta \cdot B_\theta^\top = \Sigma_\theta$ für eine Matrix $B_\theta \in \mathbb{R}^{M\times d}$. Falls die Matrix B_θ quadratisch ist, folgt eine vereinfachte Berechnungsformel des Deltas in dem multivariaten normalverteilten Fall:

$$\begin{aligned}\frac{\partial \mathbb{E}\left(g(X(\theta))\right)}{\partial\theta} &= \mathbb{E}\left(g(X(\theta))(X(\theta)-\bar{\mu}(\theta))^\top \Sigma_\theta^{-1} \cdot \frac{\partial\bar{\mu}(\theta)}{\partial\theta}\right)\\ &= \mathbb{E}\left(g(X(\theta))(B_\theta\cdot Z_\theta)^\top \left(B_\theta \cdot B_\theta^\top\right)^{-1} \cdot \frac{\partial\bar{\mu}(\theta)}{\partial\theta}\right)\\ &= \mathbb{E}\left(g(X(\theta))Z_\theta^\top B_\theta^{-1} \cdot \frac{\partial\bar{\mu}(\theta)}{\partial\theta}\right). \qquad (2.16)\end{aligned}$$

Wir werden in Abschnitt 3.4 die Likelihood-Ratio-Methode in Bezug auf den multivariaten normalverteilten Basiswert weiter diskutieren und zeigen, wie diese auf das LIBOR-Marktmodell mit einigen notwendigen Änderungen angewendet werden kann.

3. Berechnung des Deltas und (Cross-)Gammas von Zinsderivaten

In Kapitel 2 haben wir bereits die Pathwise-Methode und die Likelihood-Ratio-Methode zur Bewertung des Deltas und (Cross-)Gammas von allgemeinen Finanzderivaten ausführlich vorgestellt. Von jetzt an wollen wir die beiden Methoden auf die Zinsderivate gemäß dem LIBOR-Marktmodell erweitern. Dazu befassen wir uns in diesem Kapitel nur mit der theoretischen Untersuchung. Die praktische Anwendung auf die Bermuda-Swaptions werden wir im nächsten Kapitel präsentieren. Unsere Diskussion beginnt mit dem LIBOR-Marktmodell.

3.1. LIBOR-Marktmodell

Das LIBOR-Marktmodell ist ein bekanntes und wichtiges Zinsstrukturmodell zur Behandlung von Zinsderivaten, insbesondere von komplexen Zinsderivaten. Im Gegensatz zu anderen Zinsstrukturmodellen verwendet es anstelle von fiktiven Forward-Zinssätzen die tatsächlich am Markt beobachtbaren Zinssätze. Dieses Modell wurde erstmals in den Arbeiten von Brace, Gatarek und Musiela [4] und Miltersen, Sandmann und Sondermann [26] angewandt, um die Forward-LIBORs zu simulieren. Das LIBOR-Marktmodell führt zu Bewertungsgleichungen ähnlich der im Finanzbereich sehr bekannten Black-Scholes-Formel von Black und Scholes [3] und hat sich daher heutzutage in der Praxis durchgesetzt. Die Formulierung dieses Abschnittes orientiert sich hauptsächlich an Kapitel 3 von Glasserman [11]. Außerdem kann man die Monte-Carlo-Simulation des LIBOR-Marktmodells in Kapitel 31 von Hull [14] oder Kapitel 5 von R. Korn, E. Korn und Kroisandt [18] nachschlagen.

Bevor wir das LIBOR-Marktmodell formell vorstellen, wollen wir das bekannte und einfachste Zinsderivat, den Bond, einführen. Ein Bond mit Fälligkeit T ist ein Wertpapier, das zum Zeitpunkt T eine Währungseinheit, z. B. 1€, auszahlt. Den Preis eines Bonds zum Zeitpunkt $t \leq T$ bezeichnen wir mit $B_T(t)$. Offensichtlich sind $B_T(T) = 1$ und $B_T(t) \in (0, 1]$, wenn man negative Verzinsung ausschließt. Der Bond ist manchmal für die Behandlung anderer Zinsderivate so relevant wie der Zinssatz. Denn Bonds eignen sich bereits hervorragend zur Beschreibung anderer Zinsderivate. Oft wird er auch als Diskontfaktor betrachtet.

Unter der Notation $L_T(t)$ versteht man den Forward-LIBOR in der durch die

Bonds dargestellten Form

$$L_T(t) = \frac{1}{\delta}\left(\frac{B_T(t) - B_{T+\delta}(t)}{B_{T+\delta}(t)}\right) \qquad 0 \le t \le T, \tag{3.1}$$

der ein Forward-Zinssatz zur Zeit t ist und den zukünftigen Zeitraum $[T, T+\delta]$ diskret verzinst. Bei der weiteren Diskussion beschränken wir uns der Einheitlichkeit halber auf die festen Zeitpunkte $T_0 < T_1 < \cdots < T_M$ mit $T_0 = 0$ und $\delta_i = T_{i+1} - T_i$. Die Zeitstruktur $\{T_0, T_1, \ldots, T_M\}$ wird auch die Tenorstruktur genannt. Wir können die Gleichungen (3.1) gemäß der Tenorstruktur umschreiben. Dabei schreiben wir $B_i(\cdot)$ statt $B_{T_i}(\cdot)$ und statt $L_{T_i}(\cdot)$ schreiben wir $L_i(\cdot)$:

$$L_i(t) = \frac{1}{\delta_i}\left(\frac{B_i(t) - B_{i+1}(t)}{B_{i+1}(t)}\right) \qquad 0 \le t \le T_i,$$

für $i = 0, \ldots, M-1$. Wir definieren weiter eine Funktion $\eta : [0, T_M) \longrightarrow \{1, \ldots, M\}$ durch $T_{\eta(t)-1} \le t < T_{\eta(t)}$. D. h. $\eta(t)$ ist der Index des nächsten Tenors nach t. Mit dieser Notation haben wir

$$B_i(t) = B_{\eta(t)}(t) \cdot \prod_{j=\eta(t)}^{i-1} \frac{1}{1 + \delta_j L_j(t)} \qquad 0 \le t < T_i$$

und

$$B_i(T_l) = \prod_{j=l}^{i-1} \frac{1}{1 + \delta_j L_j(T_l)} \qquad l = 0, \ldots, i-1 \tag{3.2}$$

für $i = 1, \ldots, M$.

Wir betrachten jetzt einen stochastischen Prozess B^* mit

$$B^*(t) = B_{\eta(t)}(t) \cdot \prod_{j=0}^{\eta(t)-1} (1 + \delta_j L_j(T_j)) \qquad 0 \le t \le T_i, \tag{3.3}$$

für $i = 1, \ldots, M-1$. Wir bemerken, dass $B^*(T_0) = 1$ ist und in der Gleichung (3.3) $L_j(T_j) = L_j(T_i)$ gelten. Wenn wir B^* als ein Numeraire behandeln, nennen wir das zugehörige äquivalente Martingalmaß das Spot-Maß.

Jamshidian [15] entwickelt einen Ansatz um das LIBOR-Marktmodell in Bezug auf das Spot-Maß zu untersuchen. Besteht das LIBOR-Marktmodell aus M Bonds, deren Fälligkeiten jeweils $T_1, \ldots, T_M$ sind, und seien die Volatilitäten $\sigma_1, \ldots, \sigma_{M-1} : [0, T) \longrightarrow \mathbb{R}^d$ deterministisch, dann entwickeln sich die Forward-LIBORs $L_i(t)$ unter dem Spot-Maß gemäß den stochastischen Differentialgleichungen

$$\frac{dL_i(t)}{L_i(t)} = \mu_i(t)dt + \sigma_i^\top(t)dW^*(t) \qquad t \in [0, T_i] \tag{3.4}$$

für $i = 1, \ldots, M-1$, wobei die Driftterme mittels Volatilitäten dargestellt werden können:

$$\mu_i(t) = \sum_{j=\eta(t)}^{i} \frac{\delta_j L_j(t) \sigma_i^\top(t) \sigma_j(t)}{1 + \delta_j L_j(t)},$$

und W^* ein d-dimensionaler Wiener-Prozess unter dem Spot-Maß ist. Wegen der starken Pfadabhängigkeit der Driftterme im LIBOR-Marktmodell und der hohen Dimensionalität des Modells ist die Monte-Carlo-Simulation fast das einzige erfolgversprechende Verfahren, das zur Simulation des LIBOR-Marktmodell und seinen numerischen Behandlungen von Zinsderivaten geeignet ist.

Wir simulieren nun das LIBOR-Marktmodell unter dem Spot-Maß auf dem Zeitgitter $t_0 < t_1 < \cdots < t_N$ mit $t_0 = T_0$ und $t_N = T_{M-1}$, wobei $\{t_0, \ldots, t_N\}$ die Teiltenormenge $\{T_0, \ldots, T_{M-1}\}$ abdeckt, d. h. $\{T_0, \ldots, T_{M-1}\} \subseteq \{t_0, \ldots, t_N\}$, dabei sind $T_i = t_{N_i}$ für $i = 0, \ldots, M-1$ und daher gelten $N_0 = 0$ und $N_{M-1} = N$. Weiterhin implementieren wir auf $\{t_0, \ldots, t_N\}$ die Euler-Diskretisierung des Itô-Prozesses von $\ln(L_i(t))$:

$$L_i(n+1) = L_i(n) \exp\left(\left(\mu_i(n) - \frac{1}{2}\|\sigma_i(n)\|^2\right) h_n + \sqrt{h_n}\, \sigma_i^\top(n) Z(n+1)\right) \tag{3.5}$$

für $n = 0, \ldots, N-1$ und $i = \eta(t_n), \ldots, M-1$ mit den Drifttermen

$$\mu_i(n) = \sigma_i^\top(n) \cdot \sum_{j=\eta(t_n)}^{i} \frac{\delta_j L_j(n) \sigma_j(n)}{1 + \delta_j L_j(n)}, \tag{3.6}$$

wobei $h_n = t_{n+1} - t_n$ für $n = 0, \ldots, N-1$ und $Z(1), \ldots, Z(N) \sim \mathcal{N}(0, I_d)$ unabhängig sind. In der Simulationsformel (3.5) vereinbaren wir die Notation $L_i(n) = L_i(t_n)$ und bemerken, dass $L_i(n) = L_i(N_i)$ für alle $n \geq N_i$ gelten.

Man erkennt an der simulierten Darstellung (3.6) der Driftterme, dass jeder einzelne Driftterm $\mu_i(n)$ in sehr komplizierter Art und Weise von den vorhergehenden Forward-LIBORs beeinflusst wird. Glasserman und Zhao [12] empfehlen eine praktische Approximation vom LIBOR-Marktmodell bzgl. des Spot-Maßes, die sogenannte Forward-Drift-Approximation. Dabei verwenden sie eine alternative und vereinfachte Simulation der Driftterme:

$$\mu_i^0(n) = \sigma_i^\top(n) \cdot \sum_{j=\eta(t_n)}^{i} \frac{\delta_j L_j(0) \sigma_j(n)}{1 + \delta_j L_j(0)}, \tag{3.7}$$

d. h. jeder simulierter Driftterm $\mu_i^0(n)$ hängt nur von den Anfangs-LIBORs $L(0)$ ab. Dadurch können wir die Forward-LIBORs direkt an dem gewünschten Termin simulieren, ohne die vorherigen Forward-LIBORs zu berechnen. Im Gegensatz zu der rekursiven Simulationsformel (3.5) sieht die Forward-Drift-Approximation der Forward-LIBORs wie folgt aus:

$$L_i(n) = L_i(0) \exp\left(\sum_{l=0}^{n-1} \left(\left(\mu_i^0(l) - \frac{1}{2}\|\sigma_i(l)\|^2\right) h_l + \sqrt{h_l}\, \sigma_i^\top(l) Z(l+1)\right)\right) \tag{3.8}$$

für $n = 0, \ldots, N_i$ und $i = 0, \ldots, M-1$. Die Formel (3.8) entspricht offenbar keiner rekursiven Simulation, sondern einer direkten Simulation.

Unter anderem wollen wir noch die Volatilitätenstruktur im Rahmen der Simulation des LIBOR-Marktmodells kurz erwähnen. Da die Forward-LIBORs in den praktischen Anwendungen normalerweise direkt auf das Zeitgitter $\{t_0, t_1, \ldots, t_{M-1}\} \equiv \{T_0, T_1, \ldots, T_{M-1}\}$ implementiert werden, ist es natürlich und üblich, die Volatilitätenfunktionen $\sigma_1(t), \ldots, \sigma_{M-1}(t)$ konstant zwischen den nacheinanderliegenden Tenordaten zu wählen. So nehmen wir an, dass jedes $\sigma_i(t)$ mit $i = 1, \ldots, M-1$ rechtsstetig ist und damit bezeichnet $\sigma_i(n)$ seinen Wert über dem Intervall $[T_n, T_{n+1})$. Falls das LIBOR-Marktmodell ein Ein-Faktor-Modell ist, d. h. es durch einen skalaren Wiener-Prozess angetrieben wird und jede $\sigma_i(t)$ eindimensional ist, ist die Volatilitätenstruktur durch eine untere Dreiecksmatrix der folgenden Form angegeben:

$$\begin{pmatrix} \binom{L_0(0)}{\bullet} & & & & \\ \binom{L_1(0)}{\bullet} & \binom{L_1(1)}{\sigma_1(0)} & & & \\ \binom{L_2(0)}{\bullet} & \binom{L_2(1)}{\sigma_2(0)} & \binom{L_2(2)}{\sigma_2(1)} & & \\ \vdots & \vdots & \vdots & \ddots & \\ \binom{L_{M-1}(0)}{\bullet} & \binom{L_{M-1}(1)}{\sigma_{M-1}(0)} & \binom{L_{M-1}(2)}{\sigma_{M-1}(1)} & \cdots & \binom{L_{M-1}(M-1)}{\sigma_{M-1}(M-2)} \end{pmatrix}, \tag{3.9}$$

wobei $\binom{L_i(n)}{\sigma_i(n-1)}$ bedeutet, dass $\sigma_i(n-1)$ erst bei der Simulation des Zinssatzes $L_i(n)$ auftaucht und dazu beiträgt. Die oben-rechte Hälfte der Matrix (3.9) ist leer, weil jeder $L_i(n)$ gleich $L_i(i)$ ist für $n > i$, also trivial.

Eine solche Volatilitätenstruktur ist stationär, wenn jedes $\sigma_i(t)$ nur von der Differenz $T_i - t$ abhängt. Wir können also jedes $\sigma_i(t)$ durch $\sigma(T_i - t)$ umschreiben. Dabei gilt $\sigma_i(n) = \sigma(i-n)$ nach unserer Notation. Für eine stationäre, eindimensionale und stückweise konstante Volatilitätenstruktur nimmt die Matrix (3.9) einfach die folgende Form mit den skalaren Werten $\sigma(1), \ldots, \sigma(M-1)$ an:

$$\begin{pmatrix} \binom{L_0(0)}{\bullet} & & & & \\ \binom{L_1(0)}{\bullet} & \binom{L_1(1)}{\sigma(1)} & & & \\ \binom{L_2(0)}{\bullet} & \binom{L_2(1)}{\sigma(2)} & \binom{L_2(2)}{\sigma(1)} & & \\ \vdots & \vdots & \vdots & \ddots & \\ \binom{L_{M-1}(0)}{\bullet} & \binom{L_{M-1}(1)}{\sigma(M-1)} & \binom{L_{M-1}(2)}{\sigma(M-2)} & \cdots & \binom{L_{M-1}(M-1)}{\sigma(1)} \end{pmatrix}.$$

Diese stationäre Volatilitätenstruktur hilft sowohl bei der Simulation als auch bei der Kalibrierung des LIBOR-Marktmodells.

Nun sind wir in der Lage, Pfade der Forward-LIBORs $L_0, \ldots, L_{M-1}$ vom Zeitpunkt T_0 bis T_{M-1} zu erzeugen, so können wir damit ein Zinsderivat numerisch bewerten. Wir betrachten nun ein Zinsderivat mit der Auszahlungsfunktion $g(L(T_{M-1})|\, T_m)$ zur Zeit T_m mit $T_0 < T_m \leq T_{M-1}$. Simulieren wir die Auszahlung $g(L(N_{M-1})\,|T_m)$ im Rahmen des LIBOR-Marktmodells unter dem Spot-Maß, dann erhalten wir den Wert des Zinsderivates zum Zeitpunkt $T_n < T_m$:

$$B^*(T_n) \cdot \mathbb{E}^* \left(\frac{g(L(N_{M-1})\,|T_m)}{B^*(T_m)} \middle| \mathcal{F}^*_{T_n} \right)$$

und nach genauer Berechnung gilt

$$\mathbb{E}^* \left(g(L(N_{M-1}) \,|T_m) \cdot \prod_{j=n}^{m-1} \frac{1}{1+\delta_j L_j(N_j)} \right), \tag{3.10}$$

wobei $\mathbb{E}^*(\cdot)$ der Erwartungswert unter dem Spot-Maß und $\mathcal{F}_t^*$ die Filtration bzgl. des Wiener-Prozesses W^* ist. Wir bemerken noch, dass in der Gleichung (3.10) $L_j(N_j) = L_j(N_m)$ gilt. Den Bruch

$$\frac{B^*(T_n)}{B^*(T_m)} = \prod_{j=n}^{m-1} \frac{1}{1+\delta_j L_j(N_j)}$$

interpretieren wir als Diskontfaktor vom Zeitpunkt T_m bis zum Zeitpunkt T_n im Rahmen der Simulation des LIBOR-Marketmodells.

Das LIBOR-Marktmodell spielt nicht nur bei der Bewertung eines komplexen Zinsderivates eine wichtige Rolle, sondern auch bei der Bestimmung seiner Risikoparameter, z. B. des Deltas und (Cross-)Gammas, was wir in den folgenden Abschnitten Schritt für Schritt erklären wollen. Vorher befassen wir uns kurz mit der Forward-Drift-Approximation des LIBOR-Marktmodells. Die Simulationsformel (3.5) mit der Driftform (3.6) ist eine exakte Simulation ohne Bias. Im Gegensatz dazu ist die Forward-Drift-Approximation (3.8) eine inexakte Simulation mit Bias. Jedoch hat die Vereinfachung der Driftterme einige Vorteile für die Bestimmung der Risikoparameter eines Zinsderivates. Die von der Forward-Drift-Approximation abgeleitete Pathwise-Methode ist erstens schnell und zweitens einfacher zu implementieren als die auf der exakten Simulationsformel basierende Pathwise-Methode. Und ihre numerischen Ergebnisse bzgl. der Berechnung des Deltas sind trotz des Bias vollkommen akzeptabel. Außerdem kann man dank der Forward-Drift-Approximation die Likelihood-Ratio-Methode für das LIBOR-Marktmodell umsetzen, was beim exakt simulierten Fall wegen der komplexen Driftterme nicht möglich ist. Diese Themen werden wir noch in den folgenden Abschnitten genauer diskutieren.

3.2. Pathwise-Delta

Wir beginnen mit der Pathwise-Methode zur Berechnung des Deltas eines Zinsderivates und gehen dabei zunächst auf das Forward-Delta und danach auf das Adjoint-Delta ein.

3.2.1. Forward-Delta

Der Schwerpunkt dieses Unterabschnittes liegt auf der Berechnung des Deltas eines Zinsderivates im Rahmen der Simulation des LIBOR-Marktmodells durch die Forward-Methode. Dazu benutzen wir die bereits bekannte Dynamik (3.4)

des Forward-LIBORs unter dem Spot-Maß und ihre Simulationsformel (3.5) im Abschnitt 3.1.

Wir betrachten jetzt erneut die diskontierte Auszahlungsfunktion eines Zinsderivates. Bezeichnen wir hier diese statt mit $g(X(T))$ für ein allgemeines Finanzderivat mit $g(L(T_{M-1}))$, so ergibt sich die entsprechende Simulationsformel der Forward-Methode für den Delta-Vektor unmittelbar aus der Formel (2.5). Dabei wird $X(N)$ durch $L(N)$ ersetzt:

$$\boldsymbol{\Delta}\left(g(L(N))\right) = \frac{\partial g(L(N))}{\partial L(0)} = \frac{\partial g(L(N))}{\partial L(N)} \cdot \Delta(N) \tag{3.11}$$

bzw.

$$\boldsymbol{\Delta}_j\left(g(L(N))\right) = \frac{\partial g(L(N))}{\partial L_j(0)} = \sum_{i=j}^{M-1} \frac{\partial g(L(N))}{\partial L_i(N)} \cdot \Delta_{ij}(N),$$

wobei

$$\Delta_{ij}(n) = \frac{\partial L_i(n)}{\partial L_j(0)}$$

für $i = 0, \ldots, M-1$ und $j = 0, \ldots, i$. Unsere nächste Aufgabe besteht darin, die Rekursionsformel $\Delta(n+1) \longleftarrow \Delta(n)$ bzw. die Darstellung der Matrix $D(n)$ im Rahmen der Simulation des LIBOR-Marktmodells zu bestimmen, damit $\Delta(N)$ numerisch berechnet werden kann. Durch die Ableitung der beiden Seiten der Gleichung (3.5) bekommen wir die Rekursionsformeln

$$\begin{aligned} \Delta_{ij}(n+1) &= \frac{L_i(n+1)}{L_i(n)}\Delta_{ij}(n) \\ &\quad + L_i(n+1)h_n\sigma_i^\top(n) \cdot \sum_{k=\eta(t_n)}^{i} \frac{\delta_k \sigma_k(n)\Delta_{kj}(n)}{\left(1+\delta_k L_k(n)\right)^2} \qquad i \geq \eta(t_n) \end{aligned} \tag{3.12}$$

$$\Delta_{ij}(n+1) = \Delta_{ij}(n) \qquad i < \eta(t_n) \tag{3.13}$$

für $n = 0, \ldots, N-1$, $i = 0, \ldots, M-1$ und $j = 0, \ldots, i$ mit der Initialmatrix $\Delta(0) = I_M$. Der Zeitaufwand pro Rekursionsschritt der Rekursionsformel (3.12) ist kleiner als der Zeitaufwand $O(M^3)$ im allgemeinen Fall. Die Summierung auf der rechten Seite von Gleichung (3.12) kostet $O(M)$ für jedes $j = 1, \ldots, i$, deshalb beschränken sich die gesamten Zeitkosten pro Rekursionsschritt nur auf $O(M^2)$.

Wenn wir die rekursiven Gleichungen (3.12) und (3.13) zusammen mit der Gleichung

$$\Delta_{ij}(n+1) = \sum_{k=1}^{M-1} D_{ik}(n)\Delta_{kj}(n),$$

vergleichen, dann erhalten wir die Darstellung der Matrix $D(n)$ im Rahmen der

Simulation des LIBOR-Marktmodells:

$$D(n) = \begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & & D_{\eta(t_n),\eta(t_n)}(n) & & \\ & & & \vdots & \ddots & \\ & & & D_{M-1,\eta(t_n)}(n) & \cdots & D_{M-1,M-1}(n) \end{pmatrix}. \tag{3.14}$$

mit den diagonalen Termen

$$D_{ii}(n) = \begin{cases} 1 & i < \eta(t_n) \\ \frac{L_i(n+1)}{L_i(n)} + \frac{L_i(n+1)\|\sigma_i(n)\|^2\delta_i h_n}{(1+\delta_i L_i(n))^2} & i \geq \eta(t_n) \end{cases}$$

und den nicht-diagonalen Termen

$$D_{ij}(n) = \begin{cases} \frac{L_i(n+1)\sigma_i^\top(n)\sigma_j(n)\delta_j h_n}{(1+\delta_j L_j(n))^2} & i > j \geq \eta(t_n) \\ 0 & \text{sonst.} \end{cases}$$

3.2.2. Adjoint-Delta

Wir diskutieren nun die Adjoint-Methode zur Bewertung des Deltas eines Zinsderivates. Dabei verwenden wir dieselben Voraussetzungen und Notationen wie die Forward-Methode. Aus den rekursiven Formeln (2.7) im Unterabschnitt 2.2.2 wissen wir bereits, dass die Adjoint-Methode zur Bewertung von $\boldsymbol{\Delta} = V^\top(0)$ im Wesentlichen einer Rekursion mit dem Anfangsvektor $V(N)$ entspricht. Wenn es sich jetzt um die Bewertung des Delta-Vektors eines Zinsderivates im LIBOR-Marktmodell handelt, sieht der Anfangsvektor so aus:

$$V(N) = \left(\frac{\partial g(L(N)}{\partial L(N)}\right)^\top. \tag{3.15}$$

Dann bleibt nur noch eins zu tun, nämlich eine konkrete Rekursionsformel $V(n) \longleftarrow V(n+1)$ im Rahmen der Simulation des LIBOR-Marktmodells zu formulieren. Dazu betrachten wir wiederum die Darstellung (3.14) der Matrix $D(n)$. Ihre transponierte Matrix ist

$$D^\top(n) = \begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & & D^\top_{\eta(t_n),\eta(t_n)}(n) & \cdots & D^\top_{\eta(t_n),M-1}(n) \\ & & & & \ddots & \vdots \\ & & & & & D^\top_{M-1,M-1}(n) \end{pmatrix}$$

mit den diagonalen Einträgen

$$D_{ii}^{\top}(n) = \begin{cases} 1 & i < \eta(t_n) \\ \frac{L_i(n+1)}{L_i(n)} + \frac{L_i(n+1)\|\sigma_i(n)\|^2\delta_i h_n}{(1+\delta_i L_i(n))^2} & i \geq \eta(t_n) \end{cases}$$

und den nicht-diagonalen Einträgen

$$D_{ij}^{\top}(n) = \begin{cases} \frac{L_j(n+1)\sigma_j^{\top}(n)\sigma_i(n)\delta_i h_n}{(1+\delta_i L_i(n))^2} & j > i \geq \eta(t_n) \\ 0 & \text{sonst.} \end{cases}$$

Wegen der Rekursionsformel (2.7) gilt

$$V_i(n) = \sum_{j=1}^{M-1} D_{ij}^{\top}(n) \cdot V_j(n+1). \tag{3.16}$$

Ersetzen wir jeden Eintrag von $D^{\top}(n)$ in der Form (3.16) durch seine konkrete Form, dann folgt die konkrete Rekursionsformel von $V(n) \longleftarrow V(n+1)$ im Rahmen der Simulation des LIBOR-Marktmodells:

$$\begin{aligned} V_i(n) &= \frac{L_i(n+1)}{L_i(n)} V_i(n+1) \\ &\quad + \frac{\sigma_i^{\top}(n)\delta_i h_n}{(1+\delta_i L_i(n))^2} \sum_{j=i}^{M-1} L_j(n+1) V_j(n+1)\sigma_j(n) \quad i \geq \eta(t_n) \end{aligned} \tag{3.17}$$

$$V_i(n) = V_i(n+1) \qquad i < \eta(t_n) \tag{3.18}$$

für $n = N-1, \ldots, 0$, und $i = 0, \ldots, M-1$ mit dem Anfangsvektor (3.15). Die Summierung auf der rechten Seite von (3.17) kostet $O(M)$, deshalb beschränkt sich der gesamte Zeitaufwand pro Rekursionsschritt auch auf $O(M)$. Das ergibt einen deutlichen Rechengewinn im Vergleich zur rekursiven Formel (3.12) der Forward-Methode.

3.3. Pathwise-Delta unter Forward-Drift

Bislang haben wir die Pathwise-Methode zur Berechnung des Deltas eines Zinsderivates im Rahmen der exakten Simulation des LIBOR-Marktmodells vollständig diskutiert. In diesem Abschnitt wollen wir eine alternative Pathwise-Methode im Rahmen der Forward-Drift-Approximation des LIBOR-Marktmodells vorstellen, welche einmal von Glasserman und Zhao [12] erörtert wurde. Die Forward-Methode und Adjoint-Methode sind exakte Simulationsmethoden mit Rekursionen. Die Pathwise-Methode unter Forward-Drift entspricht hingegen einer inexakten Simulationsmethode ohne Rekursionen. Wir fangen mit der Simulationsformel (3.8) der Forward-Drift-Approximation des LIBOR-Marktmodells an.

Daraus bekommen wir die Formen von allen $L_i(N)$ für $i = 0, \ldots, M-1$:

$$L_i(N_i) = L_i(0) \exp\left(\sum_{l=0}^{N_i-1} \left(\left(\mu_i^0(l) - \frac{1}{2} \|\sigma_i(l)\|^2 \right) h_l + \sqrt{h_l}\, \sigma_i^\top(l) Z(l+1) \right) \right), \tag{3.19}$$

wobei $Z(1), \ldots, Z(N) \sim \mathcal{N}(0, I_d)$ und $\mu_i^0(l)$ in Formel (3.7) nachgeschlagen werden kann. Durch die Ableitungen der beiden Seiten der Gleichung (3.19) können wir alle Einträge der Matrix $\Delta(N)$ unmittelbar zur Zeit T_N simulieren:

$$\begin{aligned} & \Delta_{ij}(N_i) \\ = \; & \mathbf{1}\{i=j\} \frac{L_i(N_i)}{L_i(0)} + \mathbf{1}\{i \geq j\} L_i(N_i) \sum_{l=0}^{N_j-1} h_l \frac{\partial \mu_i^0(l)}{\partial L_j(0)} \\ = \; & \mathbf{1}\{i=j\} \frac{L_i(N_i)}{L_i(0)} + \mathbf{1}\{i \geq j\} \frac{L_i(N_i)\delta_j}{(1+\delta_j L_j(0))^2} \sum_{l=0}^{N_j-1} h_l \sigma_i^\top(l) \sigma_j(l) \end{aligned} \tag{3.20}$$

für $i, j = 0, \ldots, M-1$. Damit kann der Delta-Vektor eines Zinsderivates durch die Formel (3.11) gemäß der Forward-Methode direkt berechnet werden. Für die Pathwise-Methode unter Forward-Drift gibt es im Gegensatz zur exakten Pathwise-Methode keine Adjoint-Version, weil diese Methode keiner Rekursion, sondern einer direkten Simulation entspricht und die Simulationsrichtung hierbei keine Rolle spielt. Ein Nachteil des Pathwise-Deltas unter Forward-Drift ist die Existenz von Bias. Allerdings hat das Verfahren noch zwei offensichtliche Vorteile: Es ist schnell und viel einfacher zu implementieren als die exakten Pathwise-Methoden.

3.4. Likelihood-Ratio-Delta

In diesem Abschnitt wenden wir die Likelihood-Ratio-Methode zur Berechnung des Deltas auf das LIBOR-Marktmodell an, was auf die Likelihood-Ratio-Methode bzgl. des multivariaten normalverteilten Basiswertes in Abschnitt 2.3 basiert. Dieser Ansatz wurde schon im Artikel von Glasserman und Zhao [12] ausführlich entwickelt. Sie empfehlen dazu eher die Forward-Drift-Approximation als die exakte Simulation des LIBOR-Marktmodells anzuwenden. Denn die Drifttermen der exakten Simulation des LIBOR-Marktmodells hängen auf komplizierte Art und Weise von den Forward-LIBORs selbst ab. Im Gegensatz dazu sind die Driftterme der Forward-Drift-Approximation deterministisch und hängen nur von den Anfangs-LIBORs ab. Sie bietet daher die Möglichkeit, die Likelihood-Ratio-Methode auf das LIBOR-Marktmodell hinsichtlich der implizierten multivariaten Normalverteilung der Forward-Drift-Approximation umzusetzen. Zunächst modifizieren wir die Forward-Drift-Approximation des LIBOR-Marktmodells durch die Verwendung des natürlichen Logarithmus auf die bei-

den Seiten der Formel (3.19):

$$\ln L_i(N_i) = \ln L_i(0) + \sum_{l=0}^{N_i-1} \left(\mu_i^0(l) - \frac{1}{2} \|\sigma_i(l)\|^2 \right) h_l + \sum_{l=0}^{N_i-1} \sqrt{h_l}\, \sigma_i^\top(l) Z(l+1) \qquad (3.21)$$

für $i = 1, \ldots, M-1$, wobei $Z(1), \ldots, Z(N) \sim \mathcal{N}(0, I_d)$ und $\mu_i^0(l)$ in der Formel (3.7) nachgeschlagen werden kann. Ferner erinnern wir uns an den multivariaten normalverteilten Basiswert $X(\theta)$. Falls $X(\theta) \sim \mathcal{N}(\bar{\mu}(\theta), \Sigma_\theta)$, dann existiert

$$X(\theta) = \bar{\mu}(\theta) + B_\theta \cdot Z_\theta$$

für eine Matrix B_θ mit $B_\theta \cdot B_\theta^\top = \Sigma_\theta$. Wir schreiben jetzt die Simulationsformeln (3.21) für $i = 1, \ldots, M-1$ in die Vektorform um, dabei setzen wir $\theta = L(0)$ für die Bewertung des Delta-Vektors eines Zinsderivates und bekommen

$$X(L(0)) = \bar{\mu}(L(0)) + B_{L(0)} \cdot Z_{L(0)}$$

mit

$$X(L(0)) = \begin{pmatrix} \ln L_1(N_1) \\ \vdots \\ \ln L_{M-1}(N) \end{pmatrix} \in \mathbb{R}^{M-1} \qquad Z_{L(0)} = \begin{pmatrix} Z(1) \\ \vdots \\ Z(N) \end{pmatrix} \in \mathbb{R}^{N \cdot d}$$

$$\bar{\mu}(L(0)) = \begin{pmatrix} \ln L_1(0) + \sum_{l=0}^{N_1-1} \left(\mu_{M-1}^0(l) - \frac{1}{2} \|\sigma_{M-1}(l)\|^2 \right) h_l \\ \vdots \\ \ln L_{M-1}(0) + \sum_{l=0}^{N-1} \left(\mu_{M-1}^0(l) - \frac{1}{2} \|\sigma_{M-1}(l)\|^2 \right) h_l \end{pmatrix} \in \mathbb{R}^{M-1}$$

$$\underbrace{\begin{pmatrix} \sqrt{h_0}\sigma_1^\top(0) & \cdots & \sqrt{h_{N_1-1}}\sigma_1^\top(N_1-1) & & \\ \vdots & & \vdots & \ddots & \\ \sqrt{h_0}\sigma_{M-1}^\top(0) & \cdots & \sqrt{h_{N_1-1}}\sigma_{M-1}^\top(N_1-1) & \cdots & \sqrt{h_{N-1}}\sigma_{M-1}^\top(N-1) \end{pmatrix}}_{= B_{L(0)} \in \mathbb{R}^{(M-1)\times(N\cdot d)}},$$

wobei die Matrix $B_{L(0)}$ den Rang $M-1$ hat, weil $N \cdot d \geq M-1$ und $B_{L(0)}$ eine $(M-1)$-stufige Matrix ist. Es sei weiter die Matrix $\Sigma_{L(0)} = B_{L(0)} \cdot B_{L(0)}^\top \in \mathbb{R}^{(M-1)\times(M-1)}$, die auch den vollen Rang $M-1$ hat, dann erfüllt $X(L(0))$ die multivariate Normalverteilung mit $X(L(0)) \sim \mathcal{N}(\bar{\mu}(L(0)), \Sigma_{L(0)})$. Falls die Auszahlungsfunktion eines Zinsderivates wiederum als $g(L(N))$ bezeichnet wird, ersetzen wir die durch eine alternative Funktion $\tilde{g}$:

$$\tilde{g} \begin{pmatrix} \ln L_0(0) \\ X(L(0)) \end{pmatrix} = g(L(N)).$$

Nun können wir mithilfe der Berechnungsformel (2.15) den Delta-Vektor des Zinsderivates im Rahmen der Forward-Drift-Approximation des LIBOR-Marktmo-

dells unter dem Spot-Maß wie folgt berechnen:

$$\begin{aligned}
&\frac{\partial \mathbb{E}^* \left(g(L(N))\right)}{\partial L(0)} \\
= \quad &\frac{\partial \mathbb{E}^* \left(\tilde{g}\begin{pmatrix}\ln L_0(0) \\ X(L(0))\end{pmatrix}\right)}{\partial L(0)} \\
= \quad &\mathbb{E}^* \left(\tilde{g}\begin{pmatrix}\ln L_0(0) \\ X(L(0))\end{pmatrix} (X(L(0)) - \bar{\mu}(L(0)))^\top \Sigma_{L(0)}^{-1} \cdot \frac{\partial \bar{\mu}(L(0))}{\partial L(0)}\right) \\
= \quad &\mathbb{E}^* \left(g(L(N))(X(L(0)) - \bar{\mu}(L(0)))^\top \Sigma_{L(0)}^{-1} \cdot \frac{\partial \bar{\mu}(L(0))}{\partial L(0)}\right), \qquad (3.22)
\end{aligned}$$

wobei die Einträge der Matrix $\partial\bar{\mu}(L(0))/\partial L(0)$ sind

$$\begin{aligned}
\frac{\partial \bar{\mu}_i(L(0))}{\partial L_j(0)} &= \frac{\mathbf{1}\{i=j\}}{L_i(0)} + \mathbf{1}\{i \geq j\} \sum_{l=0}^{N_j-1} h_l \frac{\partial \mu_i^0(l)}{\partial L_j(0)} \\
&= \frac{\mathbf{1}\{i=j\}}{L_i(0)} + \frac{\mathbf{1}\{i \geq j\}\delta_j}{(1+\delta_j L_j(0))^2} \sum_{l=0}^{N_j-1} h_l \sigma_i^\top(l)\sigma_j(l)
\end{aligned}$$

für $i, j = 1, \ldots, M-1$. Falls die Matrix $B_{L(0)}$ zusätzlich quadratisch ist, d. h. $M - 1 = N \cdot d$, dann können wir die folgende vereinfachte Berechnungsformel anhand der Formel (2.16) ableiten:

$$\begin{aligned}
\frac{\partial \mathbb{E}^* \left(g(L(N))\right)}{\partial L(0)} &= \mathbb{E}^* \left(g(L(N))(X(L(0)) - \bar{\mu}(L(0)))^\top \Sigma_{L(0)}^{-1} \cdot \frac{\partial \bar{\mu}(L(0))}{\partial L(0)}\right) \\
&= \mathbb{E}^* \left(g(L(N)) Z_{L(0)}^\top \cdot \underbrace{B_{L(0)}^{-1} \frac{\partial \bar{\mu}(L(0))}{\partial L(0)}}_{=B^{-1}\cdot\bar{\mu}'}\right). \qquad (3.23)
\end{aligned}$$

Wir betrachten die Berechnungsformel (3.23). Der Faktor $B^{-1} \cdot \bar{\mu}'$ ist offenbar unabhängig von den Zufallszahlen, bleibt also konstant in allen Simulationspfaden. Deshalb mus man diesen Faktor bei der Simulation nur einmal berechnen und dann auf jeden Pfad anwenden, um Rechnungsaufwand zu sparen.

Die Vorteile des Likelihood-Ratio-Deltas bestehen darin, Zinsderivate mit nicht-stetigen Auszahlungen behandeln zu können und den Delta-Vektor relativ schnell berechnen zu können. Die Nachteile sind das Auftreten großer Varianz und, dass man Delta bzgl. $L_0(0)$ nicht messen kann.

3.5. Pathwise-(Cross-)Gamma

Wir widmen uns in diesem Abschnitt dem (Cross-)Gamma bzw. der Gamma-Matrix eines Zinsderivates im Rahmen der Simulation des LIBOR-Marktmodells.

Um die Gamma-Matrix genau zu berechnen, verwenden wir wiederum die Pathwise-Methode.

3.5.1. Forward-(Cross-)Gamma

Nach wie vor diskutieren wir zuerst die Forward-Methode. Dazu müssen wir annehmen, dass die diskontierte Auszahlungsfunktion $g(L(N))$ nach dem Anfangs-LIBOR $L(0)$ zweimal stetig differenzierbar ist. So ist die Gamma-Matrix eine $(M \times M)$-Matrix mit den Einträgen

$$\begin{aligned}\mathbf{\Gamma}_{jk}\left(g(L(N)\right) &= \frac{\partial^2 g(L(N))}{\partial L_j(0)\partial L_k(0)} \\ &= \sum_{i=0}^{M-1}\left(\frac{\partial g(L(N))}{\partial L_i(N)}\right)\underbrace{\left(\frac{\partial^2 L_i(N)}{\partial L_j(0)\partial L_k(0)}\right)}_{=\Gamma^i_{jk}(N)} \\ &\quad + \sum_{i=0}^{M-1}\sum_{l=0}^{M-1}\left(\frac{\partial^2 g(L(N))}{\partial L_i(N)\partial L_l(N)}\right)\Delta_{ij}(N)\Delta_{lk}(N),\end{aligned}$$

wobei $j,k = 0,\ldots,M-1$ und

$$\Gamma^i_{jk}(n+1) = \underbrace{\sum_{l=0}^{M-1} D_{il}(n)\Gamma^l_{jk}(n)}_{=\Psi^i_{jk}(n)} + \underbrace{\sum_{l=0}^{M-1}\sum_{m=0}^{M-1}\frac{\partial^2 L_i(n+1)}{\partial L_l(n)\partial L_m(n)}\Delta_{lj}(n)\Delta_{mk}(n)}_{=\Omega^i_{jk}(n)} \tag{3.24}$$

für $n = 0,\ldots,N-1$ mit der Initialisierung $\Gamma^i_{jk}(0) = 0$ für alle $i,j,k = 0,\ldots,M-1$. Wir gehen jetzt auf die konkreten Simulationsformeln in (3.24) ein. Wenn es um das LIBOR-Marktmodell unter dem Spot-Maß geht, können wir direkt die beiden Seiten von den Rekursionsformeln (3.12) und (3.13) ableiten. Dann erhalten wir die folgenden Hauptrekursionen für (Cross-)Gamma:

$$\begin{aligned}\Gamma^i_{jk}(n+1) &= \Psi^i_{jk}(n) + \Omega^i_{jk}(n) \qquad && i \geq \eta(t_n) \\ \Gamma^i_{jk}(n+1) &= \Gamma^i_{jk}(n) && i < \eta(t_n),\end{aligned} \tag{3.25}$$

wobei

$$\begin{aligned}\Psi^i_{jk}(n) &= \frac{L_i(n+1)}{L_i(n)}\Gamma^i_{jk}(n) \\ &\quad +h_n\sigma_i(n)L_i(n+1)\sum_{l=\max(j,k,\eta(t_n))}^{i}\frac{\delta_l\sigma_l(n)\Gamma^l_{jk}(n)}{(1+\delta_l L_l(n))^2} \qquad (3.26)\\ \Omega^i_{jk}(n) &= \frac{\Delta_{ij}(n)}{L_i(n)}\left(\Delta_{ik}(n+1)-\frac{L_i(n+1)}{L_i(n)}\Delta_{ik}(n)\right) \\ &\quad +h_n\sigma_i(n)\left(\Delta_{ik}(n+1)\sum_{l=\max(j,\eta(t_n))}^{i}\frac{\delta_l\sigma_l(n)\Delta_{lj}(n)}{(1+\delta_l L_l(n))^2}\right. \\ &\quad \left.-2L_i(n+1)\sum_{l=\max(j,k,\eta(t_n))}^{i}\frac{\delta_l^2\sigma_l(n)\Delta_{lj}(n)\Delta_{lk}(n)}{(1+\delta_l L_l(n))^3}\right). \qquad (3.27)\end{aligned}$$

Wir bemerken noch, dass $\Gamma^i_{jk}(\cdot)=0$ sein muss, falls $i<\max(j,k)$ ist.

3.5.2. Adjoint-(Cross-)Gamma

Wir befassen uns nun mit der Adjoint-Methode, mit der die (Cross-)Gammas bzw. die Gamma-Matrix eines Zinsderivates im Rahmen der Simulation des LIBOR-Marktmodells genau und schneller berechnet werden. Dafür brauchen wir die Berechnungsformel (2.11) aus Unterabschnitt 2.2.4, wobei $X(N)$ durch $L(N)$ ersetzt wird:

$$\mathbf{\Gamma}_{jk}\left(g(L(N))\right)=\sum_{n=0}^{N-1}V^\top(n+1)C(n)+B_{jk}\left(g(L(N))\right) \qquad (3.28)$$

für $j,k=0,\ldots,M-1$. Die Vektoren $V(n)$ für $n=1,\ldots,N$ in der Formel (3.28) können wir durch die rückwärtigen Rekursionsformeln (3.17) und (3.18) zusammen mit dem Anfangsvektor

$$V^\top(N)=\frac{\partial g(L(N)}{\partial L(N)}$$

berechnen. Für den Vektor $C(n)$ gilt dann

$$C_i(n)=\Omega^i_{jk}(n)$$

für $n=0,\ldots,N-1$ und $i=0,\ldots,M-1$. Die genauen Darstellungen kann man der Formel (3.27) entnehmen.

4. Berechnung des Deltas und (Cross-)Gammas von Bermuda-Swaptions

Bislang haben wir die auf Monte-Carlo-Simulation basierenden Methoden zur Bewertung des Delta-Vektors und der Gamma-Matrix eines Zinsderivates im Rahmen der Simulation des LIBOR-Marktmodells ausführlich diskutiert. Im aktuellen Kapitel spezialisiert sich unsere Untersuchung auf ihre Anwendung auf die Bermuda-Swaptions, welche ein weit verbreitetes Zinsderivat und das bekannteste Callable-LIBOR-Exotic ist. Am Anfang dieses Kapitels führen wir die Definition und Notationen einer Bermuda-Swaption ein. Ferner stellen wir ein bekanntes Verfahren zur Bewertung einer Bermuda-Swaption vor, dem die Algorithmen zur Berechnung der Deltas und (Cross-)Gammas zugrunde liegen.

4.1. Bermuda-Swaption

Eine Bermuda-Swaption ist eine modifizierte amerikanische Option, deren Basiswert ein Zinsswap ist. Sie gibt dem Inhaber das Recht, den Basiswert an jeden der verabredeten Zeitpunkte eines Zeitplans, der Tenorstruktur, auszuüben, sofern der Inhaber nicht zu einem früheren Zeitpunkt ausgeübt hat.

Wir definieren nun eine Bermuda-Swaption formell. Eine $(H \times M)$-Bermuda-Swaption auf der Tenorstruktur $0 = T_0 < T_1 < \cdots < T_M = T$ ist ein Zinsderivat, dessen Inhaber an den Zahlungsterminen $\{T_n \,|\, n = H, \ldots, M-1\}$ den zugrunde liegenden Basiswert ausüben kann, welcher den Fixed-To-Floating Zinsswaps von Ausübungszeit bis T_M entspricht. Wenn eine $(H \times M)$-Bermuda-Swaption zur Zeit T_r mit $H \leq r \leq M-1$ ausgeübt wird, d. h. wenn T_r der optimalen Ausübungszeit entspricht, dann zahlt der Basiswert, ein $(r \times M)$-Zinsswap, eine Menge von Kupons $\{X_i \,|\, i = r, \ldots, M-1\}$ aus. Im Rahmen der Simulation des LIBOR-Marktmodells unter dem Spot-Maß auf das Zeitgitter $T_0 = t_0 < t_1 < \cdots < t_N = T_{M-1}$ sind

$$X_i = \phi \mathscr{N} \delta_i \left(L_i(N_i) - R \right)$$

für $i = r, \ldots, M-1$, wobei R der feste Zinssatz und $\mathscr{N}$ der Nennwert ist. Zudem ist es ein Payer-Zinsswap im Fall $\phi = 1$ oder ein Receiver-Zinsswap im Fall $\phi = -1$, welches jeweils einer Payer-Bermuda-Swaption oder einer Receiver-Bermuda-Swaption entspricht. Wir erinnern uns noch daran, dass N_i vom $t_{N_i} = T_i$ stammt für $i = 0, \ldots, M-1$. Darüber hinaus gilt es zu beachten, dass jeder Kupon X_i

zum Zeitpunkt T_i schon festgestellt ist, aber dennoch erst zum Zeitpunkt T_{i+1} ausgezahlt wird. Daher muss der Kupon X_i vom Zeitpunkt T_{i+1} zur heutigen Zeit T_0 diskontiert werden, mit dem Diskontfaktor unter dem Spot-Maß:

$$PV_{i+1} = \frac{B^*(T_0)}{B^*(T_{i+1})} = \frac{1}{B^*(T_{i+1})} = \prod_{j=0}^{i} \frac{1}{1+\delta_j L_j(N_j)}.$$

Der Wert dieser $(H \times M)$-Bermuda-Swaption entspricht daher dem Wert des $(r \times M)$-Zinsswaps zur Zeit T_0, der durch die Notation $V_{r\times M}^{Swap}(T_0)$ bezeichnet und durch die folgende Formel genau beschrieben ist:

$$V_{r\times M}^{Swap}(T_0) = \sum_{i=r}^{M-1} PV_{i+1}X_i.$$

Allerdings kommt die optimale Ausübungszeit T_r wie auch der optimale Index r in der Tat immer als eine Zufallsvariable vor. Deswegen beschreiben wir im LIBOR-Marktmodell den Wert der Bermuda-Swaption mit dem Operator des Erwartungswertes unter dem Spot-Maß:

$$V_{H\times M}^{BS}(T_0) = \mathbb{E}^*\left(V_{r\times M}^{Swap}(T_0)\right) = \mathbb{E}^*\left(\sum_{i=r}^{M-1} PV_{i+1}X_i\right), \tag{4.1}$$

wobei die Notation $V_{H\times M}^{BS}(T_0)$ den Wert der $(H \times M)$-Bermuda-Swaption zum Zeitpunkt T_0 bezeichnet. Formel (4.1) bietet nicht nur die theoretische Grundlage der Monte-Carlo-Simulation zur Bewertung einer $(H \times M)$-Bermuda-Swaption, sondern, wie wir später sehen, wird auch zur Berechnung des Deltas einer $(H \times M)$-Bermuda-Swaption in Anspruch genommen.

4.2. Kleinste-Quadrate-Monte-Carlo

Die Monte-Carlo-Simulation ist grundsätzlich nicht zur Bewertung von Optionen amerikanischen oder bermudischen Typs geeignet. Die Methode von Longstaff und Schwarz [25] bietet aber eine Möglichkeit, sie hierfür anzupassen. Dies beinhaltet die Verwendung einer Kleinste-Quadrate-Approximation, welche den Wert des Fortbestehens der Option, d. h. der Nicht-Ausübung, zu den Werten der relevanten Variablen in Beziehung setzt. Wir wenden in diesem Abschnitt dieses Verfahren auf die Bewertung einer Bermuda-Swaption an.

Bei der Bewertung einer $(H \times M)$-Bermuda-Swaption gemäß Formel (4.1), gilt es im Wesentlichen, den optimalen Ausübungszeitpunkt T_r bzw. den Index r in Formel (4.1) zu bestimmen. Die optimale Ausübungsstrategie besteht darin, dass wir zu jedem möglichen Ausübungszeitpunkt den inneren Wert der Bermuda-Swaption bei Ausübung des Rechts mit der Nicht-Ausübung vergleichen und dann die bessere Alternative, Ausüben oder nicht, wählen. Die so erhaltene Strategie liefert den maximalen Wert für die Bermuda-Swaption im Zeitpunkt null. Zur Berechnung des Ausübungskriteriums ist es also nötig, einen

bedingten Erwartungswert zu berechnen. Die Berechnung des bedingten Erwartungswertes durch eine Monte-Carlo-Simulation ist nicht trivial, was im übrigen Teil dieses Unterabschnittes verdeutlicht wird.

In den praktischen Anwendungen steht aber bei der Bestimmung des optimalen Ausübungszeitpunkts T_r die Bewertung der $(H \times M)$-Bermuda-Swaption im Vordergrund. Wir fomulieren die folgenden Rekursionsschritte im LIBOR-Marktmodell unter dem Spot-Maß:

1. Die Rekursion startet im vorletzten Zeitpunkt T_{M-1}:

$$V^{BS}_{(M-1)\times M}(T_{M-1}) = \mathbb{E}^*\left(\max\left(0, V^{Swap}_{(M-1)\times M}(T_{M-1})\right)\middle|\mathcal{F}^*_{T_{M-1}}\right). \tag{4.2}$$

2. Die Rekursion schreitet gemäß der Tenorstruktur rückwärts mit dem Index $i = M-2, \ldots, H$ fort, um $V^{BS}_{H\times M}(T_H)$ zu erreichen:

$$V^{BS}_{i\times M}(T_i) = B^*(T_i)\cdot\mathbb{E}^*\left(\frac{\max\left(V^{BS}_{(i+1)\times M}(T_{i+1}), V^{Swap}_{(i+1)\times M}(T_{i+1})\right)}{B^*(T_{i+1})}\middle|\mathcal{F}^*_{T_i}\right). \tag{4.3}$$

3. $V^{BS}_{H\times M}(T_H)$ wird von T_H auf T_0 diskontiert, um $V^{BS}_{H\times M}(T_0)$ zu erhalten:

$$V^{BS}_{H\times M}(T_0) = B^*(T_0)\cdot\mathbb{E}^*\left(\frac{V^{BS}_{H\times M}(T_H)}{B^*(T_H)}\middle|\mathcal{F}^*_{T_0}\right) = \mathbb{E}^*\left(\frac{V^{BS}_{H\times M}(T_H)}{B^*(T_H)}\right). \tag{4.4}$$

Nebenbei wird der Index r der optimalen Ausübungszeit T_r durch die folgende Formel bestimmt:

$$r = \min\left(\left\{i\in\{H,\ldots,M-1\}\;\middle|\; V^{BS}_{i\times M}(T_i) < V^{Swap}_{i\times M}(T_i)\right\}\cup\{M\}\right). \tag{4.5}$$

Ist $r = M$, dann wird diese Bermuda-Swaption niemals ausgeübt und wertlos sein.

Die Gleichungen (4.2), (4.3), (4.4) und (4.5) liefern das theoretische Gerüst zur Bewertung einer $(H \times M)$-Bermuda-Swaption und zur Bestimmung ihrer optimalen Ausübungszeit. Dies sind die fundamentalen Schritte, die bei jedem Pfad der Anwendung der Monte-Carlo-Simulation auszuführen sind. Unsere nächste Aufgabe besteht darin, eine relativ komplexe und effiziente Methode aufgrund der obigen Rekursionsschritte vorzustellen, um bedingte Erwartungswerte und Ausübungskriterien jedes Pfads richtig einzuschätzen.

Diese Methode ist bekannt unter dem Namen LSM-Algorithmus, wobei LSM kurz für den englischen Ausdruck "*least-squares Monte Carlo*" steht. Sie wird in Longstaff und Schwarz [25] entwickelt und gilt nicht nur für die Bermuda-Swaption, sondern auch andere amerikanische und bermudische Typen von Optionen.

Longstaff und Schwarz [25] geben also einen Weg an, den Wert einer Bermuda-Swaption beim Halten dieser zu bestimmen, wenn die Monte-Carlo-Simulation

benutzt wird. Ihr Ansatz beihaltet die Verwendung der Methode der kleinsten Quadrate in der Regressionsanalyse jeder möglichen Ausübungszeit. Hiermit wird eine Funktion geschätzt, welche den Zusammenhang zwischen dem Wert beim Halten der Bermuda-Swaption und den Werten der relevanten Variablen zu jedem Ausübungszeitpunkt bestmöglich beschreibt. Diese Funktion ist das Skalarprodukt zwischen einer κ-dimensionalen Basisfuntion f und einem Gewichtsvektor $\alpha \in \mathbb{R}^{\kappa}$. Hierbei ist die Auswahl einer (Familie von) Basisfunktion(en) eine nicht-triviale Aufgabe, die je nach der Komplexität der Form der Auszahlungsfunktion, großen Einfluss auf die Genauigkeit der Methode hat. Für eine $(H \times M)$-Bermuda-Swaption wählen wir im unserer Arbeit die folgende Basisfunktion:

$$f(L(T_i)) = \begin{pmatrix} 1 \\ S_i^M(T_i) \\ \left(S_i^M(T_i)\right)^2 \end{pmatrix} \in \mathbb{R}^3$$

für $i = M-2, \ldots, H$, wobei $S_i^M(T_i)$ Forward-Swap-Satz genannt wird und die folgende Form hat:

$$S_i^M(T_i) = \frac{1 - B_M(T_i)}{\sum_{l=i}^{M-1} \delta_l B_{l+1}(T_i)}.$$

$B.(\cdot)$ bezeichnet nach wie vor den Wert eines Bonds (vgl. die Formel (3.2)). In Anhang E formulieren wir den vollständigen LSM-Algorithmus, Algorithmus 29, nach unseren eigenen Anforderungen und Notationen.

Wir beschäftigen uns in dieser Arbeit eigentlich nicht mit der Bewertung einer Bermuda-Swaption, sondern damit, ihren Delta-Vektor und Gamma-Matrix zu berechnen. Die Einführung des LSM-Algorithmus rührt daher, dass man dank diesem neben den Werten auch die optimalen Ausübungszeiten aller Pfaden bestimmen kann, welche die wichtigste Informationen zur Berechnung der Greeks der Bermuda-Swaption sind. Es überrascht deshalb nicht, dass der LSM-Algorithmus immer der erste Schritt in den Algorithmen zur Bestimmung der Greeks einer Bermuda-Swaption ist, die wir in den folgenden Abschnitten entwickeln werden.

4.3. Modifizierte Finite-Differenzen-Methode

Wir behandeln nun das Hauptthema dieser Arbeit, den Delta-Vektor und die Gamma-Matrix einer Bermuda-Swaption numerisch zu berechnen. Dafür werden wir hier verschiedene numerische Methoden basierend auf der Monte-Carlo-Simulation entwickeln. Uns fällt zunächst die Finite-Differenzen-Methode auf. Anhand der Finite-Differenzen-Methode können wir eine grobe Schätzung der Greeks einer Bermuda-Swaption erhalten. Diese Methode ist zumindest oberflächlich leichter zu verstehen und umzusetzen. Sie führt aber oft zu relativ großen Bias und Varianzen. Anschließend wollen wir das Prinzip der Finite-Differenzen-Methode kurz und knapp erläutern, was wir schon in Abschnitt 2.1

für allgemeine Finanzderivate erwähnt haben. Nach wie vor bezeichnet $V^{BS}_{H\times M}(T_0)$ den Wert einer $(H \times M)$-Bermuda-Swaption zur Teit T_0. Dann entspricht die Formel

$$\mathbf{\Delta}_i \left(V^{BS}_{H\times M}(T_0) \right) = \frac{\partial V^{BS}_{H\times M}(T_0)}{\partial L_i(0)}$$

ihrem i-ten Delta für $i = 0, \ldots, M-1$. Wir können diese partielle Abteilung durch den Vorwärtsdifferenzenquotient

$$\mathbf{\Delta}_i \left(V^{BS}_{H\times M}(T_0) \right) \overset{Vor.}{\longleftarrow} \frac{V^{BS}_{H\times M}(T_0, L_i(0)+\epsilon) - V^{BS}_{H\times M}(T_0, L_i(0))}{\epsilon}, \tag{4.6}$$

den Rückwärtsdifferenzenquotient

$$\mathbf{\Delta}_i \left(V^{BS}_{H\times M}(T_0) \right) \overset{Rück.}{\longleftarrow} \frac{V^{BS}_{H\times M}(T_0, L_i(0)) - V^{BS}_{H\times M}(T_0, L_i(0)-\epsilon)}{\epsilon} \tag{4.7}$$

oder den zentralen Differenzenquotient

$$\mathbf{\Delta}_i \left(V^{BS}_{H\times M}(T_0) \right) \overset{Zent.}{\longleftarrow} \frac{V^{BS}_{H\times M}(T_0, L_i(0)+\epsilon) - V^{BS}_{H\times M}(T_0, L_i(0)-\epsilon)}{2\epsilon} \tag{4.8}$$

für eine hinreichend kleine Maschenweite $\epsilon > 0$ approximieren, wobei $V^{BS}_{H\times M}(T_0, L_i(0))$, $V^{BS}_{H\times M}(T_0, L_i(0)+\epsilon)$ und $V^{BS}_{H\times M}(T_0, L_i(0)-\epsilon)$ durch den LSM-Algorithmus mit der gleichen Folge von Zufallszahlen simuliert werden müssen. Da alle drei Quotienten im Grenzwert $\epsilon \to 0$ gegen das i-te Delta $\mathbf{\Delta}_i$ konvergieren, können wir für hinreichend kleines ϵ eine gute Approximation erhalten. Unter anderem erzeugt der zentrale Differenzenquotient normalerweise Bias mit kleinerer Ordnung als die anderen beiden, ist somit exakter.

Theoretisch können die (Cross-)Gammas ebenfalls durch die Formel der Finite-Differenzen-Methode dargestellt werden. Hierfür werden die rechten Seiten von (4.6), (4.7), (4.8) anstatt des Deltas $\mathbf{\Delta}\left(V^{BS}_{H\times M}(\cdot)\right)$ mit dem Gamma $\mathbf{\Gamma}\left(V^{BS}_{H\times M}(\cdot)\right)$ und die Zähler der linken Seiten von (4.6), (4.7), (4.8) anstatt des Wertes $V^{BS}_{H\times M}(\cdot)$ mit Deltas $\mathbf{\Delta}\left(V^{BS}_{H\times M}(\cdot)\right)$ ersetzt. Dabei muss man die Positionen der Indices i und j genau beachten:

$$\mathbf{\Gamma}_{ij} \left(V^{BS}_{H\times M}(T_0) \right) \overset{Vor.}{\longleftarrow} \frac{\mathbf{\Delta}_i \left(V^{BS}_{H\times M}(T_0, L_j(0)+\epsilon) \right) - \mathbf{\Delta}_i \left(V^{BS}_{H\times M}(T_0, L_j(0)) \right)}{\epsilon} \tag{4.9}$$

$$\mathbf{\Gamma}_{ij} \left(V^{BS}_{H\times M}(T_0) \right) \overset{Rück.}{\longleftarrow} \frac{\mathbf{\Delta}_i \left(V^{BS}_{H\times M}(T_0, L_j(0)) \right) - \mathbf{\Delta}_i \left(V^{BS}_{H\times M}(T_0, L_j(0)-\epsilon) \right)}{\epsilon} \tag{4.10}$$

$$\mathbf{\Gamma}_{ij} \left(V^{BS}_{H\times M}(T_0) \right) \overset{Zent.}{\longleftarrow} \frac{\mathbf{\Delta}_i \left(V^{BS}_{H\times M}(T_0, L_j(0)+\epsilon) \right) - \mathbf{\Delta}_i \left(V^{BS}_{H\times M}(T_0, L_j(0)-\epsilon) \right)}{2\epsilon}, \tag{4.11}$$

wobei

$$\mathbf{\Gamma}_{ij}\left(V^{BS}_{H\times M}(T_0)\right) = \frac{\partial^2 V^{BS}_{H\times M}(T_0)}{\partial L_i(0)\partial L_j(0)}$$

für $i,j = 0,\ldots,M-1$ und die Delta-Terme $\mathbf{\Delta}_i\left(V^{BS}_{H\times M}(\cdot)\right)$ an den rechten Seiten wiederum durch die Formel (4.6), (4.7), (4.8) gemäß der Finite-Differenzen-Methode numerisch berechnet werden können. Wir interpretieren ein solches Verfahren als Finite-Differenzen-Methode höherer Ordnung. Allerdings ist dieses Verfahren infolge der hohen Ungenauigkeit der Ergebnisse, insbesondere bei Berechnung von Cross-Gammas, bei praktischen Anwendungen sehr schlecht. Dies liegt vornehmlich daran, dass die Finite-Differenzen-Methode selbst ein großes Bias erzeugt, ganz zu geschweigen von der Finite-Differenzen-Methode höher Ordnung.

Um ein vernüftiges Berechnungsverfahren der Gamma-Matrix einer Bermuda-Swaption aufgrund der Finite-Differenzen-Methode zu entwickeln, wollen wir die Delta-Terme $\mathbf{\Delta}_i\left(V^{BS}_{H\times M}(\cdot)\right)$ für $i = 0,\ldots,M-1$ auf den rechten Seiten von (4.9), (4.10), (4.11) so exakt wie möglich simulieren. Dazu können wir statt der Finite-Differenzen-Methode die Pathwise-Methode verwenden, um diese Delta-Terme numerisch zu berechnen. Wie wir schon in Kapitel 3 diskutiert haben, ist die Pathwise-Methode in der Tat eine exakte Simulationsmethode ohne Bias. Das gesamte Berechnungsverfahren entspricht deshalb einer modifizierten Finite-Differenzen-Methode, nämlich der Kombination von Finite-Differenzen-Methode und Pathwise-Delta.

Immer noch erzeugt der zentrale Differenzenquotient von Pathwise-Delta Bias mit kleinerer Ordnung, ist also exakter als die Vorwärts- und Rückwärtsdifferenzenquotienten. Dennoch hat der zentrale Differenzenquotient einen höheren Rechenaufwand als die anderen beiden. Denn beim Vorwärts- oder Rückwärtsdifferenzenquotient braucht man für jedes $i = 0,\ldots,M-1$ nur $M+1$ Ausführungen von Pathwise-Deltas, die sind $\mathbf{\Delta}_i\left(V^{BS}_{H\times M}(T_0)\right)$ und für $j = 0,\ldots,M-1$ noch $\mathbf{\Delta}_i\left(V^{BS}_{H\times M}(T_0, L_j(0)\pm\epsilon)\right)$. Im Gegensatz dazu braucht der zentralen Differenzenquotient für jedes $i = 0,\ldots,M-1$ aber $2M$ Ausführungen von Pathwise-Deltas, die sind $\mathbf{\Delta}_i\left(V^{BS}_{H\times M}(T_0, L_j(0)+\epsilon)\right)$ und $\mathbf{\Delta}_i\left(V^{BS}_{H\times M}(T_0, L_j(0)-\epsilon)\right)$ für alle $j = 0,\ldots,M-1$.

Bislang haben wir das Prinzip der modifizierten Finite-Differenzen-Methode schon klar erklärt. Im nächsten Abschnitt steht die Pathwise-Methode wieder im Mittelpunkt unserer Diskussion, nämlich den Pathwise-Delta-Vektor der Bermuda-Swaption zu berechnen, welches eine große Rolle bei der modifizierten Finite-Differenzen-Methode spielt.

4.4. Pathwise-Delta-Vektor

Obwohl wir schon in den Unterabschnitten 3.2.1, 3.2.2 und dem Abschnitt 3.3 die Pathwise-Methode zur Berechnung des Deltas eines allgemeinen Zinsderivates ausführlich vorgestellt haben, können wir sie nicht direkt auf die Bermuda-Swaption anwenden. Denn die Umsetzung des Pathwise-Deltas in die Bermuda-Swaption ist wegen ihrer spezifischen Eigenschaft nicht trivial. Eine genaue Diskussion darüber ist daher notwendig.

Zunächst wollen wir die theoretische Basis legen. Wir gehen von der alternativen Rekursionsformel von (4.3) zur Bewertung eines $(H \times M)$-Bermuda-Swaption aus:

$$\frac{V^{BS}_{i\times M}(T_i)}{B^*(T_i)} = \mathbb{E}^*\left(\frac{\max\left(V^{BS}_{(i+1)\times M}(T_{i+1}), V^{Swap}_{(i+1)\times M}(T_{i+1})\right)}{B^*(T_{i+1})}\middle| \mathcal{F}^*_{T_i}\right). \tag{4.12}$$

Wenn wir die beiden Seiten von (4.12) nach dem Anfangs-LIBOR-Vektor $L(0)$ ableiten, bekommen wir die folgende Rekursionsformel unter Berücksichtigung der Linearität des Erwartungswertoperators $\mathbb{E}^*$:

$$\begin{aligned}
&\boldsymbol{\Delta}\left(\frac{V^{BS}_{i\times M}(T_i)}{B^*(T_i)}\right)\\
=\ &\mathbb{E}^*\left(\boldsymbol{\Delta}\left(\frac{\max\left(V^{BS}_{(i+1)\times M}(T_{i+1}), V^{Swap}_{(i+1)\times M}(T_{i+1})\right)}{B^*(T_{i+1})}\right)\middle| \mathcal{F}^*_{T_i}\right)\\
=\ &\mathbb{E}^*\left(\mathbf{1}\left\{V^{BS}_{(i+1)\times M}(T_{i+1}) > V^{Swap}_{(i+1)\times M}(T_{i+1})\right\}\boldsymbol{\Delta}\left(\frac{V^{BS}_{(i+1)\times M}(T_{i+1})}{B^*(T_{i+1})}\right)\right.\\
&\left.+\mathbf{1}\left\{V^{BS}_{(i+1)\times M}(T_{i+1}) \le V^{Swap}_{(i+1)\times M}(T_{i+1})\right\}\boldsymbol{\Delta}\left(\frac{V^{Swap}_{(i+1)\times M}(T_{i+1})}{B^*(T_{i+1})}\right)\middle| \mathcal{F}^*_{T_i}\right).
\end{aligned} \tag{4.13}$$

Wenn wir die linke und rechte Seite von (4.4) nach dem Anfangs-LIBOR-Vektor $L(0)$ ableiten, erhalten wir die folgende Formel wegen der Linearität des Erwartungswertoperators $\mathbb{E}^*$:

$$\boldsymbol{\Delta}\left(V^{BS}_{H\times M}(T_0)\right) = \mathbb{E}^*\left(\boldsymbol{\Delta}\left(\frac{V^{BS}_{H\times M}(T_H)}{B^*(T_H)}\right)\right). \tag{4.14}$$

Mithilfe der Rekursionsformel (4.13) können wir Formel (4.14) wie folgt ausführ-

lich schreiben:

$$
\begin{aligned}
& \boldsymbol{\Delta}\left(V^{BS}_{H\times M}(T_0)\right) \\
= \; & \mathbb{E}^*\left(\mathbf{1}\left\{V^{BS}_{(H+1)\times M}(T_{H+1}) > V^{Swap}_{(H+1)\times M}(T_{H+1})\right\}\boldsymbol{\Delta}\left(\frac{V^{BS}_{(H+1)\times M}(T_{H+1})}{B^*(T_{H+1})}\right)\right. \\
& \left. +\mathbf{1}\left\{V^{BS}_{(H+1)\times M}(T_{H+1}) \leq V^{Swap}_{(H+1)\times M}(T_{H+1})\right\}\boldsymbol{\Delta}\left(\frac{V^{Swap}_{(H+1)\times M}(T_{H+1})}{B^*(T_{H+1})}\right)\right) \\
\vdots & \\
= \; & \sum_{j=H}^{M-1}\mathbb{E}^*\left(\prod_{i=H}^{j-1}\mathbf{1}\left\{V^{BS}_{i\times M}(T_i) > V^{Swap}_{i\times M}(T_i)\right\}\mathbf{1}\left\{V^{BS}_{j\times M}(T_j) \leq V^{Swap}_{j\times M}(T_j)\right\}\boldsymbol{\Delta}\left(\frac{V^{Swap}_{j\times M}(T_j)}{B^*(T_j)}\right)\right)
\end{aligned}
$$

Ist r der Index der optimalen Ausübungszeit T_r, dann ist

$$
\mathbf{1}\{j=r\} = \prod_{i=H}^{j-1}\mathbf{1}\left\{V^{BS}_{i\times M}(T_i) > V^{Swap}_{i\times M}(T_i)\right\}\mathbf{1}\left\{V^{BS}_{j\times M}(T_j) \leq V^{Swap}_{j\times M}(T_j)\right\}.
$$

Deshalb gilt:

$$
\begin{aligned}
& \boldsymbol{\Delta}\left(V^{BS}_{H\times M}(T_0)\right) \\
= \; & \sum_{j=H}^{M-1}\mathbb{E}^*\left(\mathbf{1}\{j=r\}\boldsymbol{\Delta}\left(\frac{V^{Swap}_{j\times M}(T_j)}{B^*(T_j)}\right)\right) \\
= \; & \sum_{j=H}^{M-1}\mathbb{E}^*\left(\mathbf{1}\{j=r\}\boldsymbol{\Delta}\left(\mathbb{E}^*\left(\sum_{i=j}^{M-1}PV_{i+1}X_i\middle|\mathcal{F}^*_{T_j}\right)\right)\right) \\
= \; & \sum_{j=H}^{M-1}\mathbb{E}^*\left(\mathbf{1}\{j=r\}\mathbb{E}^*\left(\sum_{i=j}^{M-1}\boldsymbol{\Delta}\left(PV_{i+1}X_i\right)\middle|\mathcal{F}^*_{T_j}\right)\right). \qquad (4.15)
\end{aligned}
$$

Die Vertauschung des Erwartungswertoperators und des Delta-Operators in Formel (4.15) folgt aus der Linearität des Erwartungswertoperators. Das Ereignis $\{j=r\}$ ist $\mathcal{F}^*_{T_j}$-messbar, weil die optimale Ausübungszeit T_r eine stochastische Stoppzeit ist. So können wir die Indikatorfunktion $\mathbf{1}\{j=r\}$ innerhalb des

Erwartungswertoperators verschieben und erhalten

$$\begin{aligned}
& \boldsymbol{\Delta}\left(V^{BS}_{H\times M}(T_0)\right) \\
= & \sum_{j=H}^{M-1} \mathbb{E}^*\left(\mathbb{E}^*\left(\mathbf{1}\{j=r\}\sum_{i=j}^{M-1}\boldsymbol{\Delta}\left(PV_{i+1}X_i\right)\middle|\mathcal{F}^*_{T_j}\right)\right) \\
= & \sum_{j=H}^{M-1} \mathbb{E}^*\left(\mathbf{1}\{j=r\}\sum_{i=j}^{M-1}\boldsymbol{\Delta}\left(PV_{i+1}X_i\right)\right) \\
= & \mathbb{E}^*\left(\sum_{j=H}^{M-1}\left(\mathbf{1}\{j=r\}\sum_{i=j}^{M-1}\boldsymbol{\Delta}\left(PV_{i+1}X_i\right)\right)\right) \\
= & \mathbb{E}^*\left(\sum_{i=r}^{M-1}\boldsymbol{\Delta}\left(PV_{i+1}X_i\right)\right).
\end{aligned}$$

Deshalb berechnen wir den Delta-Vektor einer Bermuda-Swaption durch die Pathwise-Methode in zwei Schritten. Zuerst berechnen wir

$$\sum_{i=r}^{M-1}\boldsymbol{\Delta}\left(PV_{i+1}X_i\right)$$

entlang jedes simulierten Pfades, wobei der optimale Index r jedes Pfades durch den LSM-Algorithmus geschätzt wird. Dann berechnen wir den Mittelwert über alle Pfade. Die Pathwise-Methode funktioniert hier gut, weil sowohl PV_{i+1} als auch X_i für $i = r, \ldots, M-1$ stetig differenzierbar bzgl. des Anfangs-LIBOR-Vektors $L(0)$ sind.

Aus der obigen Diskussion wissen wir, dass die Berechnung des Delta-Vektors dieser $(H \times M)$-Bermuda-Swaption unter Anwendung der Pathwise-Methode basiert auf der Gleichung

$$\boldsymbol{\Delta}\left(V^{BS}_{H\times M}(T_0)\right) = \mathbb{E}^*\left(\sum_{i=r}^{M-1}\boldsymbol{\Delta}\left(PV_{i+1}X_i\right)\right) \tag{4.16}$$

bzw. der Gleichung

$$\boldsymbol{\Delta}_j\left(V^{BS}_{H\times M}(T_0)\right) = \mathbb{E}^*\left(\sum_{i=r}^{M-1}\boldsymbol{\Delta}_j\left(PV_{i+1}X_i\right)\right) \tag{4.17}$$

für $j = 0, \ldots, M-1$. Unter Berufung auf die theoretischen Grundlagen (4.16) und (4.17) können wir die Pathwise-Methode zur Bewertung des Delta-Vektors einer Bermuda-Swaption in den folgenden Unterabschnitten genau entwickeln. Übrigens fassen wir die Gleichungen (4.16) und (4.17) auch als die theoretischen Grundlagen der Likelihood-Ratio-Methode auf, welche eine Variante der Pathwise-Methode ist. Nach wie vor beginnen wir mit dem Forward-Delta-Vektor.

4.4.1. Forward-Delta-Vektor

Basierend auf den Gleichungen (4.16) und (4.17) hat Piterbarg [32] schon die Forward-Methode zur Berechnung der Delta-Vektoren von Bermuda-Swaptions im Rahmen der Simulation des LIBOR-Marktmodells erfolgreich formuliert. Sein Prinzip fassen wir im folgenden Entwurf zusammen:

1. Wir führen den LSM-Algorithmus aus. Dabei bestimmen wir die optimalen Ausübungszeiten für alle Pfade.

2. Entlang eines jeden Pfades berechnen wir durch die Forward-Methode aus Unterabschnitt 3.2.1 die Delta-Vektoren aller Auszahlungen von der optimalen Ausübungszeit bis zur Fälligkeit der Bermuda-Swaption und summieren diese Delta-Vektoren.

3. Über alle Pfade summieren wir die Ergebnisse und berechnen ihren Mittelwert.

Für eine ausführliche Formulierung gemäß unserer Notationen kann man Algorithmus 30 in Anhang E nachschlagen. Allerdings ist unsere Vorstellung der Forward-Methode von Piterbarg [32] damit nicht abgeschlossen. Es fehlt noch die $\boldsymbol{\Delta}_j\left(PV_{i+1}X_i\right)$ für alle $i = r, \ldots, M-1$ und $j = 0, \ldots, M-1$ bzgl. der Forward-Methode genau zu berechnen. Durch die Kettenregel entwickelt sich das j-te Delta wie folgt:

$$\begin{aligned} & \boldsymbol{\Delta}_j\left(PV_{i+1}X_i\right) \\ = & \sum_{l=0}^{M-1}\left(\frac{\partial\left(PV_{i+1}X_i\right)}{\partial L_l(N_l)}\right)\Delta_{lj}(N_l) \\ = & \mathbf{1}\{j \le i\}\cdot PV_{i+1}\cdot\left(\phi\mathscr{N}\delta_i\Delta_{ij}(N_i) - X_i\cdot\sum_{l=j}^{i}\frac{\delta_l\Delta_{lj}(N_l)}{1+\delta_l L_l(N_l)}\right). \end{aligned} \tag{4.18}$$

Dabei ergeben sich die Delta-Faktoren $\Delta_{ij}(N)$ für alle $i, j = 0, \ldots, M-1$ aus den Rekursionen (3.12) und (3.13).

Bislang haben wir die Forward-Methode zur Berechnung des Delta-Vektors einer Bermuda-Swaption im Rahmen der exakten Simulation des LIBOR-Marktmodells lückenlos diskutiert. Nun werfen wir unseren Blick der Reihe nach auf die Adjoint-Methode.

4.4.2. Adjoint-Delta-Vektor

Wie wir schon in Unterabschnitt 3.2.2 diskutiert haben, ist die Adjoint-Methode zur Berechnung des Deltas eines Zinsderivates effizienter als die Forward-Methode im Rahmen der exakten Simulation des LIBOR-Marktmodells. Deshalb wollen wir hier für die Bermuda-Swaption auch die Adjoint-Methode benutzen,

um ihren Delta-Vektor schneller zu berechnen. Aus den theoretischen Grundlagen (4.16) und (4.17) folgt der erste Entwurf für die Bestimmung des Adjoint-Deltas einer Bermuda-Swaption:

1. Wir führen den LSM-Algorithmus aus und bestimmen dabei die optimalen Ausübungszeiten für alle Pfade.

2. Entlang eines jeden Pfades bestimmen wir durch die Adjoint-Methode aus Unterabschnitt 3.2.2 die Delta-Vektoren aller Auszahlungen von der optimalen Ausübungszeit bis zur Fälligkeit der Bermuda-Swaption und summieren diese Delta-Vektoren.

3. Über alle Pfade summieren wir die Ergebnisse und berechnen ihren Mittelwert.

Der einzige Unterschied zum Forward-Delta einer Bermuda-Swaption besteht darin, dass wir oben versuchen, die Adjoint-Methode direkt an der Stelle anzuwenden, wo früher die Forward-Methode angewendt wurde. D. h. für alle $i = r, \ldots, M-1$ und $j = 0, \ldots, M-1$ werden $\boldsymbol{\Delta}_j \left(PV_{i+1}X_i\right)$ statt durch die Forward-Methode aus Unterabschnitt 3.2.1 nun durch die Adjoint-Methode aus Unterabschnitt 3.2.2 berechnet. Da es sich entlang eines jeden Pfades um alle Auszahlungen von der optimalen Ausübungszeit bis zur Fälligkeit der betreffenden Bermuda-Swaption handelt, wollen wir hier eine neue Notation einführen, um einen Notationskonflikt mit den mehrmaligen Vorkommen des Adjoint-Vektors $V(\cdot)$ aus Unterabschnitt 3.2.2 zu vermeiden. Der neue Vektor sieht wie folgt aus:

$$V^\top(N_j|T_i) = \frac{\partial\left(PV_{i+1}X_i\right)}{\partial L(N_j)}$$

für $i = r, \ldots, M-1$ und $j = 0, \ldots, M-1$. Dann folgt

$$V^\top(0|T_i) = V^\top(N_0|T_i) = \frac{\partial\left(PV_{i+1}X_i\right)}{\partial L(0)} = \boldsymbol{\Delta}\left(PV_{i+1}X_i\right).$$

Gemäß Gleichung (2.6) im Unterabschnitt 3.2.2 kann der gesamte Delta-Vektor $\boldsymbol{\Delta}\left(PV_{i+1}X_i\right)$ so mittels der Adjoint-Methode bestimmt werden:

$$\boldsymbol{\Delta}\left(PV_{i+1}X_i\right) = V^\top(0|T_i) = V^\top(N_i|T_i) \cdot D(N_i - 1) \cdots D(0). \tag{4.19}$$

Daraus folgt der Delta-Vektor der Bermuda-Swaption:

$$\boldsymbol{\Delta}^\top\left(V^{BS}_{H\times M}(T_0)\right) = \mathbb{E}^*\left(\sum_{i=r}^{M-1} \boldsymbol{\Delta}^\top\left(PV_{i+1}X_i\right)\right) = \mathbb{E}^*\left(\sum_{i=r}^{M-1} V(0|T_i)\right)$$

Wir bemerken, dass die obige rückwärtige Rekursion für jeden Zeitindex $i = r, \ldots, M-1$ einmal durchgeführt werden muss. D. h. wenn wir den Delta-Vektor

eines Pfads bewerten möchten, müssen wir $M-r$ Rekursionen gemäß (4.19) ausführen. Also entsteht dann das Problem, dass die Adjoint-Methode zwar schneller als die Forward-Methode für jede einzelne Auszahlung ist, aber für einen gesamten Pfad mit mehreren Auzahlungen hat sie leider keinen Vorteil im Vergleich zur Forward-Methode. Denn bei der Anwendung der Forward-Methode brauchen wir nur einmal die vorwärtige Rekursion gemäß der Formeln (3.12) und (3.13) durchzuführen, um die Matrix $\Delta(N)$ zu simulieren. Damit können wir die Delta-Vektoren aller Auszahlungen eines Pfads durch Formel (4.18) direkt berechnen. Dieser Prozess ist offenbar schneller als die mehrfachen rückwärtigen Rekursionen unter Anwendung der Adjoint-Methode.

Falls die Adjoint-Methode immer noch bei der Bewertung des Deltas einer Bermuda-Swaption zu einer vernünftigen Anwendung gelangen soll, müssen solche mehrfache rückwärtige Rekursionen vermieden werden. Dafür haben Leclerc, Liang und Schneider [22] eine gute Lösung gefunden. Dank der Linearität der rückwärtigen Rekursion (4.19) haben sie einen linear algebraischen Superpositionsvektor eingeführt. Damit fängt die rückwärtige Rekursion ab Zeit T_{M-1} an. Der Superpositionsvektor liest bei jedem Auszahlungszeitpunkt die entsprechende Auszahlung ein. Nach der Addition dieser Auszahlung wird der Superpositionsvektor weiterhin rückwärts rekursiv simuliert bis zum letzten Auszahlungszeitpunkt. Dann wird die obige Vorgehensweise wiederholt bis zu T_r. Von da an wird die normale rückwärtige Rekursion bis zu T_0 ausgeführt. Daraufhin definieren wir diesen wichtigen Superpositionsvektor $\mathbf{V}(\cdot)$ in mathemathischer Form

$$\mathbf{V}(n) = \begin{cases} V(N_{M-1}|T_{M-1}) & n = N_{M-1} = N \\ D^\top(N_i)\cdot\mathbf{V}(N_i+1) + V(N_i|T_i) & n \in \{N_i | \, i = r, \ldots, M-2\} \\ D^\top(n)\cdots D^\top(N_r-1)\cdot\mathbf{V}(N_r) & n < N_r \\ D^\top(n)\cdots D^\top(N_{\eta(t_n)}-1)\cdot\mathbf{V}(N_{\eta(t_n)}) & sonst \end{cases} \tag{4.20}$$

für $n = N, \ldots, 0$. Mithilfe des Superpositionsvektors reicht die einmalige rückwärtige Rekrsion zur Berechnung des Delta-Vektors einer Bermuda-Swaption unter Anwendung der Adjoint-Methode. Das Prinzip und die theoretische Grundlage wird durch die folgenden Formeln erklärt:

$$\begin{aligned} & \sum_{i=r}^{M-1} V(0|T_i) \\ = & \sum_{i=r}^{M-1} D^\top(0)D^\top(1)\cdots D^\top(N_i-1)\cdot V(N_i|T_i) \end{aligned}$$

$$
\begin{aligned}
&= D^\top(0)D^\top(1)\cdots D^\top(N_r-1)\cdot \\
&\quad \Big(V(N_r|T_r) + D^\top(N_r)\cdots D^\top(N_{r+1}-1)\cdot \\
&\quad \Big(V(N_{r+1}|T_{r+1}) + \cdots + D^\top(N_{m-1})\cdots D^\top(N_m-1)\cdot \\
&\quad \Big(V(N_m|T_m) + \cdots + D^\top(N_{M-3})\cdots D^\top(N_{M-2}-1)\cdot \\
&\quad \Big(V(N_{M-2}|T_{M-2}) + D^\top(N_{M-2})\cdots D^\top(N-1)\cdot \\
&\quad V(N|T_{M-1}))\cdots)\cdots)) \\
&= D^\top(0)D^\top(1)\cdots D^\top(N-1)\cdot \mathbf{V}(N). \qquad (4.21)
\end{aligned}
$$

Aufgrund der Formel (4.21) können wir einen effizienten Prozess für das Adjoint-Delta einer Bermuda-Swaption formulieren:

1. Führe den LSM-Algorithmus aus und bestimme dabei die optimalen Ausübungszeiten aller Pfade.

2. Berechne den Delta-Vektor eines jeden Pfades durch die Kombination der Adjoint-Methode aus Unterabschnitt 3.2.2 und dem Superpositionsvektor (4.20).

3. Berechne den Mittelwert der Ergebnissen über alle Pfade.

Die formelle Version des obigen Prozesses nach unseren Notationen, Algorithmus 31, steht in Anhang E. Um den Algorithmus zu vervollständigen müssen noch die Vektoren $V(N_i|T_i)_j$ für alle möglichen i und j berechnet werden. Die Berechnungsformeln sind

$$
\begin{aligned}
V^\top(N_i|T_i)_j &= \frac{\partial\left(PV_{i+1}X_i\right)}{\partial L_j(N_i)} \\
&= \mathbf{1}\{j\le i\}PV_{i+1}\phi\mathscr{N}\delta_i\left(\mathbf{1}\{j=i\} - \frac{\delta_j(L_i(N_i)-R)}{1+\delta_j L_j(N_j)}\right) \qquad (4.22)
\end{aligned}
$$

für $i = r,\ldots,M-1$ und $j = 0,\ldots,M-1$.

4.4.3. Adjoint-Delta-Vektor (Neue Version)

Der Adjoint-Ansatz von Leclerc, Liang und Schneider [22] für Bermuda-Swaption ist zwar sehr erfolgreich, aber er ist relativ schwer zu verstehen und zu implementieren. Wir entwickeln in diesem Unterabschnitt eine alternative Adjoint-Methode zur Berechnung des Delta-Vektors einer Bermuda-Swaption, die Liang [24] bereits kurz erwähnt hat und ebenso exakt und effizient wie die Adjoint-Methode von Leclerc, Liang und Schneider [22] ist. Dies ist allergings leichter

zu verstehen und einfacher zu implementieren. Zuerst behaupten wir, dass für $i = r, \ldots, M-1$ gelten:

$$\begin{aligned} V(0|T_i) &= D^\top(0)\cdots D^\top(N_i-1)\cdot V(N_i|T_i) \\ &= D^\top(0)\cdots D^\top(N_i-1)D^\top(N_i)\cdots D^\top(N-1)\cdot V(N_i|T_i). \end{aligned} \tag{4.23}$$

Denn gemäß den Simulationsformeln (3.17) und (3.18) ändert die Operation $D^\top(N_i)\cdots D^\top(N-1)\cdot V(N_i|T_i)$ lediglich die $(i+1)$-te,...,$(M-1)$-te Komponente des Vektors $V(N_i|T_i)$ und diese Komponenten sind wegen der Rechnungsformel (4.22) gleich null. Deswegen ändert sich der Vektor $V(N_i|T_i)$ nach der Multiplikation mit $D^\top(N_i)\cdots D^\top(N-1)$ nicht. Aus der Gleichung (4.23) folgt unsere neue Idee gemäß der Adjoint-Methode:

$$\begin{aligned} & \sum_{i=r}^{M-1} V(0|T_i) \\ = & \sum_{i=r}^{M-1} D^\top(0)D^\top(1)\cdots D^\top(N_i-1)\cdot V(N_i|T_i) \\ = & \sum_{i=r}^{M-1} D^\top(0)D^\top(1)\cdots D^\top(N_i-1)\cdot D^\top(N_i)\cdots D^\top(N-1)\cdot V(N_i|T_i) \\ = & D^\top(0)D^\top(1)\cdots D^\top(N-1)\cdot \sum_{i=r}^{M-1} V(N_i|T_i) \\ = & D^\top(0)D^\top(1)\cdots D^\top(N-1)\cdot \sum_{i=r}^{M-1} \left(\frac{\partial\left(PV_{i+1}X_i\right)}{\partial L(N_i)}\right)^\top \\ = & D^\top(0)D^\top(1)\cdots D^\top(N-1)\cdot \left(\frac{\partial\left(\sum_{i=r}^{M-1} PV_{i+1}X_i\right)}{\partial L(N)}\right)^\top. \end{aligned} \tag{4.24}$$

Im Vergleich zur Formel (4.21) ist die Formel (4.24) offenbar besser zu verstehen und zu implementieren, weil diese Formel eine einfachere Simulationsstruktur als die Formel (4.21) präsentiert. Dabei verschieben wir alle Auszahlungen nach der optimalen Ausübungszeit bis zur Fälligkeit der Bermuda-Swaption und führen wir einmal die rückwärtige Rekursion gemäß der Adjoint-Methode von der Fälligkeit bis zur heutigen Zeit durch.

Formel (4.24) zeigt auch die Überlegenheit der Adjoint-Methode im Vergelich zur Forward-Methode. Wir erinnern uns an unsere Behauptung in Unterabschnitt 2.2.2 bzgl. der Adjoint-Methode: "*Die Adjoint-Methode ist effizienter für die Berechnung der Risikoparameter, wenn das einschlägige Portfolio aus wenigen Instrumenten besteht, die von vielen Parametern beeinflusst werden.*" Obwohl es sich bei der Risikoschätzung einer Bermuda-Swaption im Prinzip um ein Portfolio mit $M-r$ Instrumenten handelt, die $M-r$ Auszahlungen nach der optimalen Ausübungszeit entsprechen, können wir gemäß Formel (4.24) die

Summe aller Auszahlungen nach der optimalen Ausübungszeit als ein einzelnes Instrument behandeln, das von M Parametern, nämlich die Anfangs-LIBORs $L_0(0), \ldots, L_{M-1}(0)$, beeinflusst wird. Darauf kann die Adjoint-Methode besser und effizienter angewandt werden als die Forward-Methode. Der entsprechende Prozess lautet wie folgt:

1. Der LSM-Algorithmus wird ausgeführt, um die optimalen Ausübungszeiten von allen Pfaden zu bestimmen.

2. Entlang eines jeden Pfades werden alle Auszahlungen der optimalen Ausübungszeit bis zur Fälligkeit der Bermuda-Swaption addiert.

3. Entlang eines jeden Pfades wird der Delta-Vektor durch die Adjoint-Methode aus Unterabschnitt 3.2.2 mit der obigen Summe von Auszahlungen berechnet.

4. Der Mittelwert der Ergebnisse über alle Pfade wird berechnet.

Den entsprechenden Algorithmus formulieren wir als Algorithmus 32 in Anhang E. Für die Vollständigkeit des Algorithmus müssen wir noch den Anfangsvektor, also die Summe der Auszahlungen in der Formel (4.24) genau berechnen:

$$\begin{aligned} & \frac{\partial\left(\sum_{i=r}^{M-1} PV_{i+1}X_i\right)}{\partial L_j(N)} \\ = \; & -\frac{\delta_j \cdot \sum_{i=\max(r,j)}^{M-1} PV_{i+1}X_i}{1+\delta_j L_j(j)} + \mathbf{1}\{j \geq r\} \cdot \phi \mathscr{N} \delta_j \cdot PV_{j+1}, \end{aligned}$$

für $j = 0, \ldots, M-1$.

Die Adjoint-Methode von Leclerc, Liang und Schneider [22] ist zwar relativ schwer zu verstehen und zu implementieren, aber sie ist immer noch unersetzbar. Denn die Bewertungsstruktur davon ist nicht nur für die Berechnung des Delta-Vektors geeignet, sondern auch für die Berechnung der Gamma-Matrix und der anderen Risikoparameter. Im Gegensatz dazu gilt aber die Bewertungsstruktur der neuen Adjoint-Methode in diesem Unterabschnitt nur für der Berechnung des Delta-Vektors.

4.5. Pathwise-Delta-Vektor unter Forward-Drift

Der Forward-Algorithmus und die beiden Adjoint-Algorithmen sind exakte Simulationsmethoden, da sie auf dem exakten simulierten LIBOR-Marktmodell basieren. Wir führen in diesem Unterabschnitt die Pathwise-Methode unter Forward-Drift zur Berechnung des Delta-Vektors einer Bermuda-Swaption ein, welche auf der Forward-Drift-Approximation des LIBOR-Marktmodells basiert und daher einer Simulationsmethode mit Bias entspricht.

Der Pathwise-Algorithmus unter Forward-Drift hat selbstverständlich dieselbe theoretische Grundlage wie der Forward-Algorithmus und der Adjoint-Algorithmus, nämlich die Gleichungen (4.16) und (4.17), und auch das ähnliche Bewertungsprinzip wie der Forward-Algorithmus:

1. Führe den LSM-Algorithmus aus und bestimme die optimalen Ausübungszeiten aller Pfade.

2. Berechne entlang eines jeden Pfades die Delta-Vektoren aller Auszahlungen von der optimalen Ausübungszeit bis zur Fälligkeit der Bermuda-Swaption durch die Pathwise-Methode unter Forward-Drift aus Abschnit 3.3 und addiere diese Delta-Vektoren.

3. Berechne den Mittelwert der Ergebnisse aller Pfade.

Für eine ausführliche Formulierung des Algorithmus gemäß unseren Notationen kann man Algorithmus 33 in Anhang E nachschlagen.

Wir berechnen die $\boldsymbol{\Delta}_j\left(PV_{i+1}X_i\right)$ für alle $i = r, \ldots, M-1$ und $j = 0, \ldots, M-1$ ebenfalls durch Formel (4.18) gemäß der Pathwise-Methode unter Forward-Drift. Dabei ergibt sich die Matrix der Delta-Faktoren $\Delta(N)$ aus der direkten Simulationsformel (3.20) in Abschnitt 3.3 anstatt aus den Rekursionen (3.12) und (3.13) in Unterabschnitt 3.2.1.

4.6. Likelihood-Ratio-Delta-Vektor

Bislang haben wir alle (exakten und nicht-exakten) Pathwise-Methoden zur Berechnung des Delta-Vektors einer Bermuda-Swaption diskutiert. In diesem Unterabschnitt stellen wir die entsprechende Likelihood-Ratio-Methode vor, welche ebenfalls auf der Forward-Drift-Approximation des LIBOR-Marktmodells basiert. Wir fassen die Gleichungen (4.16) und (4.17) auch als die theoretischen Rüstungen der Likelihood-Ratio-Methode auf, weil diese eine Variante der Pathwise-Methode ist. Darauf entwickeln wir die $\boldsymbol{\Delta}\left(PV_{i+1}X_i\right)$ für $i = r, \ldots, M-1$ fort mithilfe der Formel (3.22) in Bezug auf die Likelihood-Ratio-Methode:

$$\boldsymbol{\Delta}\left(PV_{i+1}X_i\right) = \left(PV_{i+1}X_i\right)\left\lfloor\left(X(L(0)) - \bar{\mu}(L(0))\right\rfloor_i^\top \left[\Sigma_{L(0)}^{-1} \cdot \frac{\partial\bar{\mu}(L(0))}{\partial L(0)}\right]_i$$

und

$$\boldsymbol{\Delta}_j\left(PV_{i+1}X_i\right) = \left(PV_{i+1}X_i\right)\left\lfloor\left(X(L(0)) - \bar{\mu}(L(0))\right\rfloor_i^\top \left[\Sigma_{L(0)}^{-1} \cdot \frac{\partial\bar{\mu}(L(0))}{\partial L_j(0)}\right]_i$$

für $j = 1, \ldots, M-1$, wobei für eine Matrix oder einen Vektor A gelte:

$$\lfloor A\rfloor_a = A \cdot \begin{pmatrix} I_a & 0 \\ 0 & 0 \end{pmatrix}.$$

Falls die Matrix $B_{L(0)}$ quadratisch ist, können wir die folgenden vereinfachten Formeln gemäß Formel (3.23) erhalten:

$$\boldsymbol{\Delta}\left(PV_{i+1}X_i\right) = \left(PV_{i+1}X_i\right)\left\lfloor Z_{L(0)}\right\rfloor_{N_i\cdot d}^{\top}\left\lfloor (B_{L(0)}^{-1}\frac{\partial\bar{\mu}(L(0))}{\partial L(0)}\right\rfloor_{N_i\cdot d}$$

und

$$\boldsymbol{\Delta}_j\left(PV_{i+1}X_i\right) = \left(PV_{i+1}X_i\right)\left\lfloor Z_{L(0)}\right\rfloor_{N_i\cdot d}^{\top}\left\lfloor (B_{L(0)}^{-1}\frac{\partial\bar{\mu}(L(0))}{\partial L_j(0)}\right\rfloor_{N_i\cdot d}$$

für $j = 1,\ldots,M-1$. Gemäß den oben genannten theoretischen Grundlagen ist das Prinzip in Bezug auf den Likelihood-Ratio-Algorithmus klar:

1. Wir führen den LSM-Algorithmus aus und bestimmen dabei die optimalen Ausübungszeiten für alle Pfade.
2. Entlang eines jeden Pfades berechnen wir durch die Likelihood-Ratio-Methode aus Abschnitt 3.4 die Delta-Vektoren aller Auszahlungen von der optimalen Ausübungszeit bis zur Fälligkeit der Bermuda-Swaption und summieren diese Delta-Vektoren.
3. Über alle Pfade summieren wir die Ergebnisse und berechnen ihren Mittelwert.

Eine genaue Formulierung kann man in Algorithmus 34 in Anhang E nachschlagen. Dabei bemerken wir, dass der Likelihood-Ratio-Algorithmus nicht zur Bewertung des Deltas bzgl. $L_0(0)$ beitragen kann.

4.7. Pathwise-Gamma-Matrix

In Abschnitt 4.3 haben wir bereits eine effiziente Methode zur Bewertung der Gamma-Matrix einer Bermuda-Swaption vollständig vorgestellt. Sie ist die modifizierte Finite-Differenzen-Methode bzw. die Finite-Differenzen-Methode des Pathwise-Deltas, welche auf dem exakten simulierten LIBOR-Marktmodell basiert, nämlich die drei Pathwise-Methoden aus Abschnitt 4.4. Es gibt insgesamt sechs abgleitete Formen der modifizierten Finite-Differenzen-Methode: Der Vorwärtsdifferenzenquotient des Forward-Deltas, der Vorwärtsdifferenzenquotient des Adjoint-Deltas, der Rückwärtsdifferenzenquotient des Forward-Deltas, der Rückwärtsdifferenzenquotient des Adjoint-Deltas, der zentrale Differenzenquotient des Forward-Deltas und der zentrale Differenzenquotient des Adjoint-Deltas.

Allerdings besitzt dieses Verfahren einen Bias, d. h. sein Ergebnis kann nicht ganz exakt sein. In diesem Abschnitt wollen wir hingegen eine exakte numerische Simulationsmethode basierend auf der Monte-Carlo-Simulation entwickeln, worin der wesentliche Beitrag unserer Arbeit besteht. Dazu beschäftigen wir uns mit der reinen Pathwise-Methode (Forward-Methode und Adjoint-Methode), ohne Mischung mit anderen numerischen Methoden.

Zunächst leiten wir die theoretische Basis her. Wir gehen von der alternativen Rekursionsformel von (4.3) zur Bewertung eines $(H \times M)$-Bermuda-Swaption aus:

$$\frac{V^{BS}_{i\times M}(T_i)}{B^*(T_i)} = \mathbb{E}^*\left(\frac{\max\left(V^{BS}_{(i+1)\times M}(T_{i+1}), V^{Swap}_{(i+1)\times M}(T_{i+1})\right)}{B^*(T_{i+1})}\middle|\mathcal{F}^*_{T_i}\right) \tag{4.25}$$

Wenn wir die beiden Seiten von (4.25) nach dem Anfangs-LIBOR-Vektor $L(0)$ zweimal ableiten, erhalten wir die folgende Rekursionsformel unter Berücksichtigung der Linearität des Erwartungswertoperators $\mathbb{E}^*$:

$$\begin{aligned}
&\mathbf{\Gamma}\left(\frac{V^{BS}_{i\times M}(T_i)}{B^*(T_i)}\right)\\
=\;& \mathbb{E}^*\left(\mathbf{\Gamma}\left(\frac{\max\left(V^{BS}_{(i+1)\times M}(T_{i+1}), V^{Swap}_{(i+1)\times M}(T_{i+1})\right)}{B^*(T_{i+1})}\right)\middle|\mathcal{F}^*_{T_i}\right)\\
=\;& \mathbb{E}^*\left(\mathbf{1}\left\{V^{BS}_{(i+1)\times M}(T_{i+1}) > V^{Swap}_{(i+1)\times M}(T_{i+1})\right\}\mathbf{\Gamma}\left(\frac{V^{BS}_{(i+1)\times M}(T_{i+1})}{B^*(T_{i+1})}\right)\right.\\
&\left.+\mathbf{1}\left\{V^{BS}_{(i+1)\times M}(T_{i+1}) \le V^{Swap}_{(i+1)\times M}(T_{i+1})\right\}\mathbf{\Gamma}\left(\frac{V^{Swap}_{(i+1)\times M}(T_{i+1})}{B^*(T_{i+1})}\right)\middle|\mathcal{F}^*_{T_i}\right).
\end{aligned} \tag{4.26}$$

Wenn wir die linke und rechte Seite von (4.4) nach dem Anfangs-LIBOR-Vektor $L(0)$ zweimal ableiten, erhalten wir die folgende Formel wegen der Linearität des Erwartungswertoperators $\mathbb{E}^*$:

$$\mathbf{\Gamma}\left(V^{BS}_{H\times M}(T_0)\right) = \mathbb{E}^*\left(\mathbf{\Gamma}\left(\frac{V^{BS}_{H\times M}(T_H)}{B^*(T_H)}\right)\right). \tag{4.27}$$

Mithilfe der Rekursionsformel (4.26) können wir Formel (4.27) wie folgt ausführlich schreiben:

$$\begin{aligned}
&\mathbf{\Gamma}\left(V^{BS}_{H\times M}(T_0)\right)\\
=\;& \mathbb{E}^*\left(\mathbf{1}\left\{V^{BS}_{(H+1)\times M}(T_{H+1}) > V^{Swap}_{(H+1)\times M}(T_{H+1})\right\}\mathbf{\Gamma}\left(\frac{V^{BS}_{(H+1)\times M}(T_{H+1})}{B^*(T_{H+1})}\right)\right.\\
&\left.+\mathbf{1}\left\{V^{BS}_{(H+1)\times M}(T_{H+1}) \le V^{Swap}_{(H+1)\times M}(T_{H+1})\right\}\mathbf{\Gamma}\left(\frac{V^{Swap}_{(H+1)\times M}(T_{H+1})}{B^*(T_{H+1})}\right)\right)\\
&\vdots\\
=\;& \sum_{j=H}^{M-1}\mathbb{E}^*\left(\prod_{i=H}^{j-1}\mathbf{1}\left\{V^{BS}_{i\times M}(T_i) > V^{Swap}_{i\times M}(T_i)\right\}\mathbf{1}\left\{V^{BS}_{j\times M}(T_j) \le V^{Swap}_{j\times M}(T_j)\right\}\mathbf{\Gamma}\left(\frac{V^{Swap}_{j\times M}(T_j)}{B^*(T_j)}\right)\right)
\end{aligned}$$

Ist r der Index der optimalen Ausübungszeit T_r, dann ist

$$\mathbf{1}\{j=r\} = \prod_{i=H}^{j-1}\mathbf{1}\left\{V^{BS}_{i\times M}(T_i) > V^{Swap}_{i\times M}(T_i)\right\}\mathbf{1}\left\{V^{BS}_{j\times M}(T_j) \le V^{Swap}_{j\times M}(T_j)\right\}.$$

Deshalb gilt

$$\begin{aligned}
& \mathbf{\Gamma}\left(V_{H\times M}^{BS}(T_0)\right) \\
= & \sum_{j=H}^{M-1} \mathbb{E}^*\left(\mathbf{1}\{j=r\}\,\mathbf{\Gamma}\left(\frac{V_{j\times M}^{Swap}(T_j)}{B^*(T_j)}\right)\right) \\
= & \sum_{j=H}^{M-1} \mathbb{E}^*\left(\mathbf{1}\{j=r\}\,\mathbf{\Gamma}\left(\mathbb{E}^*\left(\sum_{i=j}^{M-1} PV_{i+1}X_i \middle| \mathcal{F}_{T_j}^*\right)\right)\right) \\
= & \sum_{j=H}^{M-1} \mathbb{E}^*\left(\mathbf{1}\{j=r\}\,\mathbb{E}^*\left(\sum_{i=j}^{M-1} \mathbf{\Gamma}\left(PV_{i+1}X_i\right) \middle| \mathcal{F}_{T_j}^*\right)\right)
\end{aligned} \tag{4.28}$$

Die Vertauschung des Erwartungswertoperators und des Gamma-Operators in Formel (4.28) folgt aus der Linearität des Erwartungswertoperators. Das Ereignis $\{j=r\}$ ist $\mathcal{F}_{T_j}^*$-messbar, weil die optimale Ausübungszeit T_r eine stochastische Stoppzeit ist. So können wir die Indikatorfunktion $\mathbf{1}\{j=r\}$ innerhalb des Erwartungswertoperators verschieben und erhalten

$$\begin{aligned}
& \mathbf{\Gamma}\left(V_{H\times M}^{BS}(T_0)\right) \\
= & \sum_{j=H}^{M-1} \mathbb{E}^*\left(\mathbb{E}^*\left(\mathbf{1}\{j=r\}\sum_{i=j}^{M-1} \mathbf{\Gamma}\left(PV_{i+1}X_i\right) \middle| \mathcal{F}_{T_j}^*\right)\right) \\
= & \sum_{j=H}^{M-1} \mathbb{E}^*\left(\mathbf{1}\{j=r\}\sum_{i=j}^{M-1} \mathbf{\Gamma}\left(PV_{i+1}X_i\right)\right) \\
= & \mathbb{E}^*\left(\sum_{j=H}^{M-1}\left(\mathbf{1}\{j=r\}\sum_{i=j}^{M-1} \mathbf{\Gamma}\left(PV_{i+1}X_i\right)\right)\right) \\
= & \mathbb{E}^*\left(\sum_{i=r}^{M-1} \mathbf{\Gamma}\left(PV_{i+1}X_i\right)\right).
\end{aligned}$$

Deshalb berechnen wir die Gamma-Matrix einer Bermuda-Swaption durch die Pathwise-Methode in zwei Schritten. Zuerst berechnen wir

$$\sum_{i=r}^{M-1} \mathbf{\Gamma}\left(PV_{i+1}X_i\right)$$

entlang jedes simulierten Pfades, wobei der Index r der optimalen Ausübungszeit jedes Pfades durch den LSM-Algorithmus geschätzt wird. Dann berechnen wir den Mittelwert über alle Pfade. Die Pathwise-Methode funktioniert hier reibungslos, da sowohl PV_{i+1} als auch X_i für $i=r,\ldots,M-1$ bzgl. des Anfangs-LIBOR-Vektors $L(0)$ zweimal stetig differenzierbar ist.

Aus der obigen Diskussion wissen wir, dass die Berechnung der Gamma-Matrix dieser $(H \times M)$-Bermuda-Swaption unter Anwendung der Pathwise-Methode basiert auf der Gleichung

$$\mathbf{\Gamma}\left(V^{BS}_{H\times M}(T_0)\right) = \mathbb{E}^*\left(\sum_{i=r}^{M-1} \mathbf{\Gamma}\left(PV_{i+1}X_i\right)\right) \tag{4.29}$$

bzw. der Gleichung

$$\mathbf{\Gamma}_{jk}\left(V^{BS}_{H\times M}(T_0)\right) = \mathbb{E}^*\left(\sum_{i=r}^{M-1} \mathbf{\Gamma}_{jk}\left(PV_{i+1}X_i\right)\right) \tag{4.30}$$

für $j = 0, \ldots, M-1$. Anhand der theoretischen Grundlagen (4.29) und (4.30) können wir die Pathwise-Methode zur Bewertung der Gamma-Matrix einer Bermuda-Swaption in den folgenden Unterabschnitten genau entwickeln. Wir beginnen wie immer mit der Forward-Methode.

4.7.1. Forward-Gamma-Matrix

Die Untersuchung der Forward-Methode zur Berechnung der (Cross-)Gammas bzw. der Gamma-Matrix einer Bermuda-Swaption ist ähnlich zur Forward-Methode zur Bewertung ihres Delta-Vektors, aber komplizierter. Im Hinblick auf den Algorithmus für den Forward-Delta-Vektor einer Bermuda-Swaption und hinsichtlich der Gleichungen (4.29) oder (4.30) ist das Prinzip der Forward-Gamma-Matrix von Bermuda-Swaption klar:

1. Der LSM-Algorithmus wird ausgeführt, um die optimalen Ausübungszeiten aller Pfade zu bestimmen.

2. Entlang eines jeden Pfades werden anhand der Forward-Methode, die bereits in Abschnitt 3.5 vorgestellt wurde, die Gamma-Matrizen aller Auszahlungen von der optimalen Ausübungszeit bis zur Fälligkeit der Bermuda-Swaption simuliert und addiert.

3. Der Mittelwert der Ergebnisse über alle Pfade wird berechnet.

Eine vollständige Formulierung des Algorithmus des Pathwise-(Cross-)Gammas einer Bermuda-Swaption kann in Anhang E unter Algorithmus 35 nachgeschlagen werden.

Um die Berechnung der Forward-Gamma-Matrix einer Bermuda-Swaption lückenlos zu erklären, müssen noch die $\mathbf{\Gamma}_{jk}\left(PV_{i+1}X_i\right)$ für alle $i = r, \ldots, M-1$ und $j, k = 0, \ldots, M-1$ dargestellt werden. Diese Berechnung ist von der Berechnung der Formel (4.18) abgeleitet. Wir müssen allerdings viele Spezialfälle

berücksichtigen und separat berechnen. Glücklicherweise lassen sich alle Fälle in einer allgemeinen Form beschreiben:

$$
\begin{aligned}
& \boldsymbol{\Gamma}_{jk}\left(PV_{i+1}X_i\right) \\
= & \sum_{i=1}^{M}\left(\frac{\partial\left(PV_{i+1}X_i\right)}{\partial L_i(N)}\right)\Gamma_{jk}^{i}(N) \\
& +\sum_{l=1}^{M}\left(\sum_{i=1}^{M}\left(\frac{\partial^2\left(PV_{i+1}X_i\right)}{\partial L_i(N)\partial L_l(N)}\right)\Delta_{ij}(N)\right)\Delta_{lk}(N) \\
= & \mathbf{1}\left\{j\le i\right\}\cdot\mathbf{1}\left\{k\le i\right\}\cdot PV_{i+1}\cdot \\
& \left(\phi\mathcal{N}\delta_i\left(\Gamma_{jk}^{i}(N_i)-\Delta_{ij}(N_i)\cdot\sum_{l=k}^{i}\frac{\delta_l\Delta_{lk}(N_l)}{1+\delta_l L_l(N_l)}-\Delta_{ik}(N_i)\cdot\right.\right. \\
& \left.\sum_{l=j}^{i}\frac{\delta_l\Delta_{lj}(N_l)}{1+\delta_l L_l(N_l)}\right)+X_i\cdot\left(\sum_{l=j}^{i}\frac{\delta_l\Delta_{lj}(N_l)}{1+\delta_l L_l(N_l)}\cdot\sum_{l=k}^{i}\frac{\delta_l\Delta_{lk}(N_l)}{1+\delta_l L_l(N_l)}\right. \\
& \left.\left.+\sum_{l=\max(j,k)}^{i}\left(\frac{\delta_l^2\Delta_{lj}(N_l)\Delta_{lk}(N_l)}{\left(1+\delta_l L_l(N_l)\right)^2}-\frac{\delta_l\Gamma_{jk}^{l}(N_l)}{1+\delta_l L_l(N_l)}\right)\right)\right).
\end{aligned}
\tag{4.31}
$$

Dabei ergeben sich die Delta-Faktoren $\Delta_{ij}(N)$ für alle $i,j=0,\ldots,M-1$ aus den Rekursionen (3.12) und (3.13). Die Gamma-Faktoren $\Gamma_{jk}^{i}(N)$ für alle $i,j,k=0,\ldots,M-1$ werden durch die Rekursionen (3.25), (3.26) und (3.27) berechnet. Für die Diagonale der Gamma-Matrix, d. h. die (Nicht-Cross-)Gammas, sieht die Form wie folgt aus:

$$
\begin{aligned}
& \boldsymbol{\Gamma}_{jj}\left(PV_{i+1}X_i\right) \\
= & \mathbf{1}\left\{j\le i\right\}\cdot PV_{i+1}\cdot\left(\phi\mathcal{N}\delta_i\left(\Gamma_{jj}^{i}(N_i)-2\Delta_{ij}(N_i)\cdot\sum_{l=j}^{i}\frac{\delta_l\Delta_{lk}(N_l)}{1+\delta_l L_l(N_l)}\right)\right. \\
& +X_i\cdot\left(\left(\sum_{l=j}^{i}\frac{\delta_l\Delta_{lj}(N_l)}{1+\delta_l L_l(N_l)}\right)^2\right. \\
& \left.\left.+\sum_{l=j}^{i}\left(\left(\frac{\delta_l\Delta_{lj}(N_l)}{1+\delta_l L_l(N_l)}\right)^2-\frac{\delta_l\Gamma_{jj}^{l}(N_l)}{1+\delta_l L_l(N_l)}\right)\right)\right)
\end{aligned}
\tag{4.32}
$$

für $i=r,\ldots,M-1$ und $j=0,\ldots,M-1$. Die Form (4.32) ist offenbar ein Spezialfall von Form (4.31) im Fall $j=k$. Nach dem Vergleich der beiden Formen (4.31) und (4.32) wissen wir, dass die Berechnung eines Cross-Gammas einer Bermuda-Swaption einen höheren Aufwand als im Fall eines (Nicht-Cross-)Gammas benötigt.

4.7.2. Adjoint-Gamma-Matrix

In diesem Unterabschnitt diskutieren wir eine effizientere Alternative der Forward-Gamma-Matrix, nämlich die Adjoint-Methode zur Berechnung der Gamma-Matrix einer Bermuda-Swaption. Im Folgenden leiten wir die Berechnungsformel der Adjoint-Gamma-Matrix Schritt für Schritt her. Aus Formel (3.28) erhalten wir

$$\mathbf{\Gamma}_{jk}\left(PV_{i+1}X_i\right)) = \sum_{n=0}^{N_i-1} V^\top(n+1|T_i)C(n) + B_{jk}\left(PV_{i+1}X_i\right) \tag{4.33}$$

für $i = r, \ldots, M-1$ und $j, k = 0, \ldots, M-1$. Und anhand der Formel (4.20) können wir den Superpositionsvektor $\mathbf{V}(n)$ wie folgt neu darstellen:

$$\begin{aligned}\mathbf{V}(n) &= \sum_{i=\max(r,\eta(t_{n-1}))}^{M-1} \left(\prod_{l=n}^{N_i-1} D^\top(l)\cdot V(N_i|T_i)\right)\\ &= \sum_{i=\max(r,\eta(t_{n-1}))}^{M-1} V(n|T_i)\end{aligned} \tag{4.34}$$

für $n = N, \ldots, 0$. Aus den beiden obigen Formeln (4.33) und (4.34) können wir die wesentliche Berechnungsformel der Adjoint-Gamma-Matrix herleiten:

$$\begin{aligned}&\sum_{i=r}^{M-1} \mathbf{\Gamma}_{jk}\left(PV_{i+1}X_i\right)\\ =\ &\sum_{i=r}^{M-1}\left(\sum_{n=0}^{N_i-1} V^\top(n+1|T_i)C(n) + B_{jk}\left(PV_{i+1}X_i\right)\right)\\ =\ &\sum_{i=r}^{M-1}\sum_{n=0}^{N_i-1} V^\top(n+1|T_i)C(n) + \sum_{i=r}^{M-1} B_{jk}\left(PV_{i+1}X_i\right)\\ =\ &\sum_{i=r}^{M-1}\Big(V^\top(1|T_i)C(0) + V^\top(2|T_i)C(1) + \cdots + V^\top(N_i|T_i)C(N_i-1)\Big)\\ &+\sum_{i=r}^{M-1} B_{jk}\left(PV_{i+1}X_i\right)\\ =\ &\sum_{i=1}^{r}\sum_{n=N_{i-1}}^{N_i-1}\left(\sum_{l=r}^{M-1} V^\top(n+1|T_l)\right)\cdot C(n)\\ &+\sum_{i=r+1}^{M-1}\sum_{n=N_{i-1}}^{N_i-1}\left(\sum_{l=i}^{M-1} V^\top(n+1|T_l)\right)\cdot C(n) + \sum_{i=r}^{M-1} B_{jk}\left(PV_{i+1}X_i\right)\end{aligned}$$

$$
\begin{aligned}
&= \sum_{i=1}^{r} \sum_{n=N_{i-1}}^{N_i-1} \mathbf{V}(n+1)C(n) + \sum_{i=r+1}^{M-1} \sum_{n=N_{i-1}}^{N_i-1} \mathbf{V}(n+1)C(n) \\
&\quad + \sum_{i=r}^{M-1} B_{jk}\left(PV_{i+1}X_i\right) \\
&= \sum_{n=0}^{N-1} \mathbf{V}(n+1)C(n) + \sum_{i=r}^{M-1} B_{jk}\left(PV_{i+1}X_i\right) \qquad (4.35)
\end{aligned}
$$

für $j,k = 0,\ldots,M-1$. Daraus folgt das Prinzip der Adjoint-Gamma-Matrix, was wir als Höhepunkt dieser Arbeit ansehen:

1. Der LSM-Algorithmus wird ausgeführt, um die optimalen Ausübungszeiten aller Pfade zu bestimmen.
2. Entlang eines jeden Pfad wird die Gamma-Matrix mittels der Berechnungsformel (4.35) und des Superpositionsvektors simuliert.
3. Der Mittelwert der Ergebnisse über alle Pfade wird berechnet.

Den entsprechenden Algorithmus formulieren wir als Algorithmus 28 in Anhang E. Danach muss nur noch $B_{jk}\left(PV_{i+1}X_i\right)$ berechnet werden. Die Ergebnisse sind

$$
\begin{aligned}
&B_{jk}\left(PV_{i+1}X_i\right) \\
= \;& \mathbf{1}\left\{j \le i\right\} \cdot \mathbf{1}\left\{k \le i\right\} \cdot PV_{i+1} \cdot \\
& \left(-\phi \mathcal{N} \delta_i \left(\Delta_{ij}(N_i) \cdot \sum_{l=k}^{i} \frac{\delta_l \Delta_{lk}(N_l)}{1+\delta_l L_l(N_l)} + \Delta_{ik}(N_i) \cdot \right.\right. \\
& \left. \sum_{l=j}^{i} \frac{\delta_l \Delta_{lj}(N_l)}{1+\delta_l L_l(N_l)} \right) + X_i \cdot \left(\sum_{l=j}^{i} \frac{\delta_l \Delta_{lj}(N_l)}{1+\delta_l L_l(N_l)} \cdot \sum_{l=k}^{i} \frac{\delta_l \Delta_{lk}(N_l)}{1+\delta_l L_l(N_l)} \right. \\
& \left.\left. + \sum_{l=\max(j,k)}^{i} \frac{\delta_l^2 \Delta_{lj}(N_l)\Delta_{lk}(N_l)}{\left(1+\delta_l L_l(N_l)\right)^2} \right) \right)
\end{aligned}
$$

für alle $i = r,\ldots,M-1$ und $j,k = 0,\ldots,M-1$. Und im Fall $j = k$ gelten

$$
\begin{aligned}
&B_{jj}\left(PV_{i+1}X_i\right) \\
= \;& \mathbf{1}\left\{j \le i\right\} \cdot PV_{i+1} \cdot \left(-2\phi \mathcal{N} \delta_i \Delta_{ij}(N_i) \cdot \sum_{l=j}^{i} \frac{\delta_l \Delta_{lk}(N_l)}{1+\delta_l L_l(N_l)} \right. \\
& \left. + X_i \cdot \left(\left(\sum_{l=j}^{i} \frac{\delta_l \Delta_{lj}(N_l)}{1+\delta_l L_l(N_l)} \right)^2 + \sum_{l=j}^{i} \left(\frac{\delta_l \Delta_{lj}(N_l)}{1+\delta_l L_l(N_l)} \right)^2 \right) \right)
\end{aligned}
$$

für alle $i = r,\ldots,M-1$ und $j = 0,\ldots,M-1$.

5. Numerische Resultate

Wir haben bereits die theoretischen Untersuchungen zur Berechnung des Delta-Vektors und der Gamma-Matrix von Bermuda-Swaptions im Rahmen der Monte-Carlo-Simulation des LIBOR-Marktmodells ausführlich diskutiert. In diesem Kapitel wollen wir einige numerische Beispiele einführen, damit die von uns entwickelten Methoden überprüft und verglichen werden können.

5.1. Numerische Beispiele für den Delta-Vektor

In diesem Abschnitt überprüfen und vergleichen wir sechs Algorithmen zur Berechnung des Delta-Vektors einer Bermuda-Swaption anhand von zwei numerischen Beispielen. Diese sechs Algorithmen sind die Finite-Differenzen-Methode (kurz: FDM), die Forward-Methode (kurz: FWP), die Adjoint-Methode (kurz: ADP), die neue Version der Adjoint-Methode (kurz: ADN), die Pathwise-Methode unter Forward-Drift (kurz: PFD) und die Likelihood-Ratio-Methode (kurz: LRM).

Das erste Beispiel bezieht sich auf die Überprüfung der Ergebnisse aller sechs Algorithmen und beschäftigt sich mit einer (2×20)-Receiver-Bermuda-Swaption und einer entsprechenden (2×20)-Payer-Bermuda-Swaption, die die folgenden, gleichen Voraussetzungen und Parameter haben:

- Die Tenorstruktur ist $\{T_0, \ldots, T_{20}\}$.
- $\delta_i = T_{i+1} - T_i = 0.25$ (in Jahren) für alle $i = 0, \ldots, 19$.
- Die Forward-LIBORs werden direkt auf dem Zeitgitter $\{t_0 \ldots, t_{19}\} \equiv \{T_0, \ldots, T_{19}\}$ implementiert.
- $h_n = t_{n+1} - t_n = 0.25$ (in Jahren) für alle $n = 0, \ldots, 18$.
- Der Nennwert $\mathscr{N}$ beträgt 10000€.
- Der feste Basissatz R ist gleich 4.5%.
- Die Anfangs-LIBOR-Kurve ist flach mit $L_0(0) = \cdots = L_{19}(0) = 5\%$.
- Die Volatilitätenstruktur ist ein-dimensional und stationär, wobei $\sigma(1) = \cdots = \sigma(19) = 20\%$.

Wir führen zuerst den LSM-Algorithmus mit 65536 Pfaden aus, um die beiden Bermuda-Swaptions zu bewerten. Dabei reduzieren wir die Varianz durch die

	Preis(€)	Standardfehler
(2×20)-Receiver-Bermuda-Swaption	115.942890333	0.247838664
(2×20)-Payer-Bermuda-Swaption	290.562677057	0.394864815

Tabelle 5.1.: Preise der beiden (2×20)-Bermuda-Swaptions

antithetische Variable. Die Ergebnisse befinden sich in Tabelle 5.1. Dann berechnen wir den Delta-Vektor der beiden Bermuda-Swaptions. Dazu benutzen wir für jede Bermuda-Swaption die sechs von uns vorgestellten Algorithmen mit jeweils 65536 Pfaden und der gleichen Folge von Zufallszahlen. Bei der Finite-Differenzen-Methode nehmen wir den zentralen Differenzenquotient mit der Maschenweite $\epsilon = 0,0000001$. Und für das varianzreduzierende Verfahren wählen wir ebenfalls die antithetische Variable. Die entsprechenden numerischen Ergebnisse illustrieren wir in den Abbildungen F.1 und F.2 in Anhang F. Im Folgenden wollen wir die numerischen Resultate genau analysieren und gegeneinander vergleichen.

Abbildung F.1 zeigt die simulierten Delta-Vektoren von der (2×20)-Receiver-Bermuda-Swaption durch alle sechs Algorithmen und Abbildung F.2 zeigt die simulierten Delta-Vektoren von der (2×20)-Payer-Bermuda-Swaption durch alle sechs Algorithmen. In der Finanzmathematik werden kleine Veränderungen in Zinssätzen oft in Basispunkten gemessen. Ein Basispunkt ist 0.01% per annum und wird durch $1\mathbf{bp}$ gekennzeichnet. Deshalb interpretieren wir $\mathbf{bp}^{-1}$ als die Einheit der Ergebnisse in den beiden Abbildungen. Ein Wert x an der i-ten Stelle des Delta-Vektor bedeutet eine Veränderung des Preises $V^{BS}_{2\times 20}(T_0)$ um die Größe x, wenn sich der Anfangs-LIBOR $L_i(0)$ in derselben Richtung um $1\mathbf{bp}$ verändert. In unseren Experimenten sind die Ergebnisse durch die Forward-Methode und die beiden Adjoint-Methoden gleich. Dies ist keinen Zufall. Denn bei der Simulation des Deltas sind die Forward-Methode und die Adjoint-Methode grundsätzlich die beiden Versionen von der exakten Pathwise-Methode. Sie haben die gleiche theoretische Grundlage und unterscheiden sich voneinander lediglich in der Rekursionsrichtung bei der Simulation. Die Ergebnisse der Finite-Differenzen-Methode und der Pathwise-Methode unter Forward-Drift sind auch vollkommen akzeptierbar, obwohl sie inexakt mit Bias sind. Im Gegensatz zu den anderen Algorithmen ist die Entwicklung des Delta-Vektors der Likelihood-Ratio-Methode nicht glatt, sondern weist eine Sägezahnform auf, weil außer dem Bias noch eine große Varianz bei der Likelihood-Ratio-Methode vorkommen kann.

In Bezug auf die Genauigkeit der Ergebnissen ist nun das Ranking der sechs Algorithmen klar. Auf dem ersten Platz sind: die Forward-Methode, die Adjoint-Methode und die Adjoint-Methode (Neue Version). Auf dem zweiten Platz sind: die Finite-Differenzen-Methode und Pathwise-Methode unter Forward-Drift. Den letzten Platz nimmt die Likelihood-Ratio-Methode an. Aber wie sieht das Ranking aus, wenn es um die Effizienz geht? Dazu betrachtet wir das zweite Beispiel. Es beschäftigt sich mit einer Reihe von $(2 \times M)$-Receiver-Bermuda-Swaptions mit

$M = 4, 8, 12, 16, 20, 24$, die die gleichen Voraussetzungen und Parameter wie das erste Beispiel haben. Wir berechnen die Delta-Vektoren der sechs Bermuda-Swaptions durch die sechs von uns vorgestellten Algorithmen mit jeweils 65536 Pfaden und der gleichen Folge von Zufallszahlen. Für das varianzreduzierende Verfahren benutzen wir wiederum die antithetische Variable. Die Zeitverhältnisse der Algorithmen präsentieren wir in den Abbildungen F.3 und F.4 in Anhang F.

Abbildung F.3 zeigt und vergleicht die relativen Zeitkosten[1] der sechs Algorithmen zur Berechnung der Delta-Vektoren von $(2 \times M)$-Receiver-Bermuda-Swaptions mit $M = 4, 8, 12, 16, 20, 24$. Die Zeitkostenlinie der Finite-Differenzen-Methode ist fast linear mit einer größten Steigung. Daher ist sie der langsamste unter allen sechs Algorithmen. Die Zeitkostenlinie der Forward-Methode ist fast linear mit viel kleinerer Steigung als die Finite-Differenzen-Methode. Also ist sie auch effizienter als die Finite-Differenzen-Methode. Die anderen vier Algorithmen sind offenbar am schnellsten, weil ihre Zeitkostenlinien fast konstant entlang der Entwicklung von $M = 4, 8, 12, 16, 20, 24$ verlaufen. Um diese vier Zeitkostenlinien genauer zu beobachten, betrachten wir außerdem Abbildung F.4, welche eine Teilabbildung der Abbbildung F.3 ist. In Abbildung F.4 erkennen wir, dass die Zeitkostenlinien der vier Algorithmen auch lineare Tendenz haben und, dass die Pathwise-Methode unter Forward-Drift bzw. die Likelihood-Ratio-Methode relativ größere Steigungen als die beiden Adjoint-Methoden haben. Außerdem ist die Adjoint-Methode (Neue Version) bei diesem numerischen Beispiel immer am schnellsten bei allen $M = 4, 8, 12, 16, 20, 24$. Allerdings gehören die relativen Zeitkosten der vier Algorithmen zur gleichen Ordnung der Zeitkomplexität und sie sind die effizientsten unter den bisher bekannten Algorithmen zur Berechnung des Delta-Vektors von Bermuda-Swaptions.

Unter anderem sind nicht alle sechs Algorithmen stabil genug. Unter Anwendung der Finite-Differenzen-Methode können manchmal Überläufe bei einigen Einträgen des Delta-Vektors vorkommen. Also ist die Finite-Differenzen-Methode empfindlich und instabil im Vergleich zu den anderen fünf Algorithmen, die robust und stabil sind.

Zusammenfassend erstellen wir Tabelle 5.2, um einen Überblick über die Vergleiche zu erhalten. Dabei ist das Ranking der Schwierigkeitsgrade der Implementierung relativ subjektiv nach unserer eigenen Sicht und Erfahrung. Laut Tabelle 5.2 ist die Adjoint-Methode (Neue Version) offensichtlich die beste Methode zur Berechung des Delta-Vektors von Bermuda-Swaptions. Daneben ist die Pathwise-Methode unter Forward-Drift sehr empfehlenswert, weil dieser sehr einfach zu implementieren ist und auch sehr schnell und ausreichend exakt ist, obwohl sie bei der Simulation einen Bias erzeugt. Die schlechtste Methode gemäß den Vergleichen ist offenbar die Likelihood-Ratio-Methode wegen ih-

[1] $$\text{Relative Zeitkosten} = \frac{\text{Zeitkosten zur Berechnung des Delta-Vektors der Bermuda-Swaption}}{\text{Zeitkosten zur Bewertung der Bermuda-Swaption}}.$$

Genauigkeit		
hoch ⟵		⟶ niedrig
FWP = ADP = ADN	FDM ≈ PFD	LRM
Geschwindigkeit		
schnell ⟵		⟶ langsam
ADP ≈ ADN ≈ PFD ≈ LRM	FWP	FDM
Stabilität		
stabil ⟵		⟶ instabil
FWP ≈ ADP ≈ ADN ≈ PFD ≈LRM		FDM

Schwierigkeitsgrad				
leicht ⟵				⟶ schwer
FDM	PFD	FWP	ADN	ADP ≈ LRM

Tabelle 5.2.: Tabellarischer Vergleich der sechs Methoden für die Berechnung des Delta-Vektors

rer großen Varianz. Jedoch kann man nicht darauf verzichten, weil im Fall von nicht-stetigen Auszahlungen nur die Likelihood-Ratio-Methode funktionieren kann. Sie muss allerdings von einem passenden varianzreduzierendem Verfahren begleitet werden.

5.2. Numerische Beispiele für die Gamma-Matrix

In diesem Abschnitt überprüfen und vergleichen wir die von uns entwickelten Methoden zur Berechnung der Gamma-Matrizen von Bermuda-Swaptions. Wir berücksichtigen zwei verschiedene Aspekte: Die Genauigkeit und die Effizienz. In Unterabschnitt 5.2.1 konfrontieren wir die exakte Pathwise-Methode mit der inexakten modifizierten Finite-Differenzen-Methode hinsichtlich der Genauigkeit. Wir wählen dabei die Forward-Methode als Vertreter der exakten Pathwise-Methode. Denn einerseits haben die beiden exakten Pathwise-Methoden (Forward-Methode und Adjoint-Methode) bei der Simulation mit der gleichen Folge von Zufallszahlen immer die gleichen numerischen Ergebnisse, und andererseits benötigt die Adjoint-Methode mehr Speicherplatz als die Forward-Methode, so dass oft der Überlauf bei der Berechnung der Gamma-Matrix einer $(H \times M)$-Bermuda-Swaption mit großem M auftritt. In Unterabschnitt 5.2.2 stellen wir die beiden exakten Pathwise-Methode (Forward-Methode und Adjoint-Methode) hinsichtlich ihrer Effizienz gegenüber.

5.2.1. Pathwise-Methode gegen Finite-Differenzen-Methode

In diesem Unterabschnitt wollen wir ein numerisches Beispiel einführen, um die Pathwise-Methode und die Finite-Differenzen-Methode zu überprüfen und zu

vergleichen. Das numerische Beispiel beschäftigt sich mit einer (4×10)-Receiver-Bermuda-Swaption und einer entsprechenden (4×10)-Payer-Bermuda-Swaption, die die gleichen Voraussetzungen und Parameter wie folgt haben:

- Die Tenorstruktur ist $\{T_0, \ldots, T_{10}\}$.
- $\delta_i = T_{i+1} - T_i = 0.5$ (in Jahren) für alle $i = 0, \ldots, 9$.
- Die Pathwise-LIBORs werden direkt auf dem Zeitgitter $\{t_0 \ldots, t_9\} \equiv \{T_0, \ldots, T_9\}$ implementiert.
- $h_n = t_{n+1} - t_n = 0.5$ (in Jahren) für alle $n = 0, \ldots, 8$.
- Der Nennwert $\mathscr{N}$ beträgt 10000€.
- Der feste Basissatz R ist gleich 5%.
- Die Anfangs-LIBOR-Kurve ist flach mit $L_0(0) = \cdots = L_9(0) = 5\%$.
- Die Volatilitätenstruktur ist ein-dimensional und stationär, wobei $\sigma(1) = \cdots = \sigma(9) = 20\%$.

Wir führen zuerst den LSM-Algorithmus mit 65536 Pfaden aus, um die beiden Bermuda-Swaptions zu bewerten. Dabei reduzieren wir die Varianz durch Verwendung der antithetische Variablen. Die Ergebnisse liegen in Tabelle 5.3. Dann

	Preis(€)	Standardfehler
(4×10)-Receiver-Bermuda-Swaption	`162.141741292`	`0.279309509`
(4×10)-Payer-Bermuda-Swaption	`162.562317890`	`0.474287094`

Tabelle 5.3.: Preise der beiden (4×10)-Bermuda-Swaptions

berechnen wir die Gamma-Matrizen der beiden Bermuda-Swaptions. Dazu benutzen wir für jede Bermuda-Swaption fünf ausgewählte Verfahren mit 65536 Pfaden und der gleichen Folge von Zufallszahlen. Die fünf Verfahren sind der Vorwärtsdifferenzenquotient des Forward-Deltas, der Vorwärtsdifferenzenquotient des Adjoint-Deltas, der zentrale Differenzenquotient des Forward-Deltas, der zentrale Differenzenquotient des Adjoint-Deltas und die exakte Pathwise-Methode. Bei den vier auf der Finite-Differenzen-Methode basierenden Verfahren verwenden wir die Maschenweite $\epsilon = 0,0000001$. Für das varianzreduzierende Verfahren wählen wir ebenfalls die antithetische Variablen. Die entsprechenden numerischen Resultate kann man in Anhang F und G nachschlagen. Im Rest des Abschnittes wollen wir die numerischen Resultate genau analysieren und gegeneinander vergleichen.

Tabelle G.3 in Anhang G zeigt die exakt simulierte Gamma-Matrix der (4×10)-Receiver-Bermuda-Swaption durch die Pathwise-Methode, Algorithmus 35. An

jeder Stelle der Gamma-Matrix steht neben dem (Cross)-Gamma auch der entsprechende Standardfehler. Alle Terme in der Diagonale der Gamma-Matrix, nämlich alle Gammas und ihre Standardfehler haben wir durch Kästen hervorgehoben. Die nicht-eingerahmten Einträge der Gamma-Matrix entsprechen den Cross-Gammas. Die Experimentaldaten werden in Basispunkten niedergeschrieben. D. h. wir interpretieren $\mathbf{bp}^{-1}$ als Einheit der Ergebnisse in Tabelle G.3. Ein Wert x an der Stelle (i, j) der Gamma-Matrix bedeutet eine Veränderung des Deltas $\mathbf{\Delta}_i$ um die Größe x, wenn sich der Anfangs-LIBOR $L_j(0)$ um $1\mathbf{bp}$ verändert, was das Gamma $\mathbf{\Gamma}_{ij}$ von der Größe $x\mathbf{bp}^{-1}$ impliziert. Abbildung F.5 in Anhang F illustriert die Gestaltung der Gamma-Matrix der (4×10)-Receiver-Bermuda-Swaption und Abbildung F.6 in Anhang F zeigt die Entwicklung der Nicht-Cross-Gammas der (4×10)-Receiver-Bermuda-Swaption von Zeitpunkt T_0 bis zum Zeitpunkt T_9. Tabelle G.1 in Anhang G zeigt das Symmetrieverhalten der exakten simulierten Gamma-Matrix in Tabelle G.3. Diese Matrix ergibt sich aus der absoluten Differenz aller Einträge der unteren Hälfte und der entsprechenden Einträge der oberen Hälfte der Gamma-Matrix. Die absolute Differenz zweier Zahlen x_1 und x_2 ist $|x_1 - x_2|$. Laut Tabelle G.1 ist die exakte simulierte Gamma-Matrix der (4×10)-Receiver-Bermuda-Swaption perfekt symmetrisch.

Nun analysieren wir die numerisch simulierten Gamma-Matrizen, die sich aus den anderen vier modifizierten Finite-Differenzen-Methoden ergeben. In unseren Experimenten sind die Ergebnisse durch den Vorwärtsdifferenzenquotienten oder zentralen Differenzenquotienten des Forward-Deltas gleich wie die Ergebnisse des Vorwärtsdifferenzenquotienten oder zentralen Differenzenquotienten des Adjoint-Deltas. Das ist kein Zufall. Denn bei der Simulation des Deltas sind die Forward-Methode und die Adjoint-Methode die beiden Arten der Pathwise-Methode. Diese haben die gleiche theoretische Grundlage und unterscheiden sich voneinander lediglich in der Rekursionsrichtung bei der Simulation. Deswegen können wir durch die vier Methoden nur zwei verschiedene Gamma-Matrizen erhalten, eine in Bezug auf den zentralen Differenzenquotienten des Pathwise-Deltas und die andere in Bezug auf den Vorwärtsdifferenzenquotient des Pathwise-Deltas. Weil die modifizierten Finite-Differenzen-Methoden den Simulationen mit Bias entsprechen, konzentrieren wir uns hier deshalb nicht auf die dadurch simulierten Gamma-Matrizen selbst, sondern auf die Matrizen der absoluten Differenz zwischen der dadurch simulierten Gamma-Matrizen und der durch die Pathwise-Methode exakt simulierten Gamma-Matrix. Tabelle G.5 in Anhang G präsentiert die Matrix der absoluten Differenz zwischen Pathwise-Gamma-Matrix und zentraler Differenzenquotient-Gamma-Matrix des Pathwise-Deltas bzgl. der Receiver-Bermuda-Swaption. Tabelle G.6 in Anhang G präsentiert die Matrix der absoluten Differenz zwischen Pathwise-Gamma-Matrix und Vorwärtsdifferenzenquotient-Gamma-Matrix des Pathwise-Deltas bzgl. der Receiver-Bermuda-Swaption. Die beiden Tabellen zeigen, dass die beiden simulierten Gamma-Matrizen die exakte simulierte Pathwise-Gamma-Matrix sehr gut approximieren und der zentrale Differenzenquotient des Pathwise-Deltas ei-

ne bessere Approximation mit kleinerem Bias als der Vorwärtsdifferenzenquotient liefert, wie wir schon in Abschnitt 4.3 erwähnten. Nun betrachten wir das Symmetrieverhalten der beiden nicht-exakt simulierten Gamma-Matrizen. Dafür schlagen wir die Tabellen G.7 und G.8 in Anhang G nach. Die beiden Tabellen zeigen, dass die beiden nicht-exakt simulierten Gamma-Matrizen zwar annähernd symmetrisch, aber noch nicht perfekt symmetrisch sind und die aufgrund des zentralen Differenzenquotienten simulierten Gamma-Matrizen besseres Symmetrieverhalten als die auf der Basis des Vorwärtsdifferenzenquotienten simulierten Gamma-Matrizen aufweisen.

In Bezug auf die Genauigkeit und das Symmetrieverhalten der Ergebnisse ist nun das Ranking der fünf Verfahren klar: Erstens, die Pathwise-Methode, zweitens, der zentrale Differenzenquotient des Forward- oder Adjoint-Deltas und drittens, der Vorwärtsdifferenzenquotient des Forward- oder Adjoint-Deltas. Aber wenn es um die Effizienz geht, sieht das Ranking ganz anders aus. Dazu betrachten wir Abbildung F.9 in Anhang F. Diese Abbildung zeigt und vergleicht die relativen Zeitkosten[2] der fünf ausgewählten Methoden zur Berechnung der Gamma-Matrix der (4×10)-Receiver-Bermuda-Swaption. Der Vorwärtsdifferenzenquotient des Adjoint-Deltas ist offenbar am schnellsten und liegt ganz vorne in diesem Ranking. Hingegen steht die exakte Pathwise-Methode (hier: Forward-Methode) bei der Effizienz ganz hinten. Außerdem läuft die Finite-Differenzen-Methode des Adjoint-Deltas schneller als die des Forward-Deltas, sowohl beim zentralen Differenzenquotienten, als auch beim Vorwärtsdifferenzenquotienten. Diese Tatsache stimmt mit der entsprechenden Behauptung unserer theoretischen Untersuchung überein.

Wenn wir die fünf Methoden darauf anwenden, die Gamma-Matrix der (4×10)-Payer-Bermuda-Swaption zu berechnen, erhalten wir dieselben Schlussfolgerungen wie bei der (4×10)-Receiver-Bermuda-Swaption. Dazu erstellen wir die numerischen Resultate ähnlich wie oben und stellen die Abbildungen F.7, F.8, F.10 in Anhang F und die Tabellen G.4, G.2, G.9, G.10, G.11, G.12 in Anhang G zur Verfügung. In Tabelle G.4 stehen die durch die Pathwise-Methode exakt simulierte Gamma-Matrix und ihre Standardfehler-Matrix der (4×10)-Payer-Bermuda-Swaption. Die Abbildungen F.7 und F.8 zeigen jeweils die Gamma-Matrix und die Nicht-Cross-Gammas. Tabelle G.9 zeigt die Matrix der absoluten Differenz zwischen der Pathwise-Gamma-Matrix und der zentralen Differenzenquotient-Gamma-Matrix des Pathwise-Deltas. Die Tabelle G.10 zeigt die Matrix der absoluten Differenz zwischen der Pathwise-Gamma-Matrix und der Vorwärtsdifferenzenquotient-Gamma-Matrix des Pathwise-Deltas. Die Tabellen G.2, G.11 und G.12 präsentieren das Symmetrieverhalten der Gamma-Matrix jeweils durch die Pathwise-Methode, den zentralen Differenzenquotient und den Vorwärtsdifferenzenquotient des Pathwise-Deltas. Abbildung F.10 ver-

[2] $$\text{Relative Zeitkosten} = \frac{\text{Zeitkosten zur Berechnung der Gamma-Matrix der Bermuda-Swaption}}{\text{Zeitkosten zur Bewertung der Bermuda-Swaption}}.$$

gleicht die relativen Zeitkosten aller fünf Methoden zur Berechnung der Gamma-Matrix bzgl. der (4×10)-Payer-Bermuda-Swaption.

Unter anderem zeigen die Tabellen G.9 und G.10 zusätzlich einen Nachteil der Finite-Differenzen-Methode des Pathwise-Deltas. Unter Anwendung der Finite-Differenzen-Methode treten manchmal bei einigen Enträgen der Gamma-Matrix Überläufe auf, insbesondere bei den Enträgen $\mathbf{\Gamma}_{ij}$ mit großem i oder j. D. h., die Finite-Differenzen-Methoden sind empfindlich im Vergleich zu der robusten Pathwise-Methode. In den Tabellen G.9 und G.10 haben wir solche Überlaufstellen durch die Romanschrift hervorgehoben. Solche Überläufe beeinflussen natürlich das Symmetrieverhalten der Gamma-Matrizen negativ, was in den Tabellen G.11 und G.12 klar wird. Die offenbar nicht-symmetrischen Enträge werden ebenfalls durch die Romanschrift hervorgehoben. Wir bemerken noch, dass die Ergebnisse des zentralen Differenzenquotienten immer noch besser sind als die des Vorwärtsdifferenzenquotienten, selbst wenn Überläufe vorkommen.

5.2.2. Adjoint-Methode gegen Forward-Methode

In diesem Unterabschnitt wollen wir die Forward-Methode und die Adjoint-Methode zur Berechnung der Gamma-Matrix einer Bermuda-Swaption hinsichtlich der Effizienz vergleichen. Dazu betrachten wir ein numerisches Beispiel. Dieses befasst sich mit sechs $(1 \times M)$-Receiver-Bermuda-Swaptions für $M = 4, 5, 6, 7, 8, 9$, die die gleichen Voraussetzungen und Parameter wie folgt haben:

- Die Tenorstruktur ist $\{T_0, \ldots, T_M\}$.
- $\delta_i = T_{i+1} - T_i = 0.25$ (in Jahren) für alle $i = 0, \ldots, M-1$.
- Die Forward-LIBORs werden direkt auf dem Zeitgitter $\{t_0 \ldots, t_{M-1}\} \equiv \{T_0, \ldots, T_{M-1}\}$ implementiert.
- $h_n = t_{n+1} - t_n = 0.25$ (in Jahren) für alle $n = 0, \ldots, M-2$.
- Der Nennwert $\mathcal{N}$ beträgt 10000€.
- Der feste Basissatz R ist gleich 4.5%.
- Die Anfangs-LIBOR-Kurve ist flach mit $L_0(0) = \cdots = L_{M-1}(0) = 5\%$.
- Die Volatilitätenstruktur ist ein-dimensional und stationär, wobei $\sigma(1) = \cdots = \sigma(M-1) = 15\%$.

Wir führen zuerst den LSM-Algorithmus mit 65536 Pfaden aus, um die sechs Bermuda-Swaptions zu bewerten. Dabei reduzieren wir die Varianz durch Verwendung der antithetische Variablen. Die Ergebnisse liegen in Tabelle 5.4. Danach berechnen wir die Gamma-Matrizen der sechs Bermuda-Swaptions durch die Forward-Methode und die Adjoint-Methode mit jeweils 65536 Pfaden

$(1 \times M)$-Receiver-Bermuda-Swaption	Preis(€)	Standardfehler
M=4	37.912360562	0.014558858
M=5	50.760684551	0.024573335
M=6	63.750562978	0.036882204
M=7	76.784540342	0.051337651
M=8	90.043731061	0.067280689
M=9	103.259557105	0.085145851

Tabelle 5.4.: Preise der sechs Receiver-Bermuda-Swaptions

und der gleichen Folge von Zufallszahlen. Für das varianzreduzierende Verfahren benutzen wir nach wie vor die antithetische Variable. Die Gamma-Matrizen der (1×6)-Receiver-Bermuda-Swaption und der (1×8)-Receiver-Bermuda-Swaption zeigen wir als Beispiele in den Abbildungen F.11 und F.12 in Anhang F. Abbildung F.13 in Anhang F vergleicht die relativen Zeitkosten der Forward-Gamma-Matrix und der Adjoint-Gamma-Matrix der sechs $(1 \times M)$-Receiver-Bermuda-Swaptions für $M = 4, 5, 6, 7, 8, 9$. Dabei fügen wir noch eine Benchmarklinien hinzu, nämlich die Zeitkosten des zentralen Differenzenquotienten des Adjoint-Delta-Vektors.

Gemäß Abbildung F.13 ist die Adjoint-Methode offenbar effizienter als die Forward-Methode. Jedoch liefert die Adjoint-Methode keinen sehr großen Zeitgewinn. Im Vergleich zur Forward-Methode benötigt die Adjoint-Methode deutlich mehr Speicherplatz. Deshalb können wir die Zeitverhältnisse der beiden Pathwise-Methode bzgl. der Gamma-Matrix nur bis $M = 9$ überprüfen.

6. Zusammenfassung

In dieser Arbeit haben wir die numerische Berechnung des Delta-Vektors und der Gamma-Matrix einer Bermuda-Swaption ausführlich untersucht und diskutiert. Diese sind analytisch nicht lösbar und spielen bei Absicherungsstrategien in Form des Delta-Hedgings und Gamma-Hedgings die entscheidende Rolle. Das zugrunde liegende Zinsstrukturmodell ist das LIBOR-Marktmodell, das in den letzten Jahren eine auffällige Entwicklung in der Finanzmathematik gemacht hat, um Zinsderivate genau bewerten und ihre Risikoparameter präzise berechnen zu können. Bei der Simulation und Anwendung des LIBOR-Marktmodells fällt die Monte-Carlo-Simulation ins Gewicht.

Für die Berechung des Delta-Vektors einer Bermuda-Swaption stellten wir insgesamt sechs numerische Methoden vor, welche fast alle vorhandenen Arten der Monte-Carlo-Simulation zur Berechnung des Delta-Vektors einer Bermuda-Swaption im Rahmen des LIBOR-Marktmodells enthalten. Diese sind die Finite-Differenzen-Methode, die Forward-Methode, die Adjoint-Methode, die neue Version der Adjoint-Methode, die Pathwise-Methode unter Forward-Drift und die Likelihood-Ratio-Methode. Die letzten drei Methoden wurde neu entwickelt und mit dieser Arbeit erstmals veröffentlicht. Jede dieser Methoden hat ihre eigenen Vorteile.

Die Finite-Differenzen-Methode ist einfach zu verstehen und zu implementieren, aber sie ist am langsamsten und hat einen Bias. Die Forward-Methode ist am exaktesten und schneller als die Finite-Differenzen-Methode. Die Adjoint-Methode ist ebenso exakt wie die Forward-Methode und noch schneller als diese. Die neue Version der Adjoint-Methode besitzt alle Vorteile der traditionellen Adjoint-Methode und ist einfacher zu verstehen und zu implementieren als diese. Die Pathwise-Methode unter Forward-Drift ist noch einfacher zu implementieren und ebenso schnell, jedoch hat sie einen kleinen Bias aus dem nicht-exakt simulierten Basiswert, nämlich die Forward-Drift-Approximation des LIBOR-Marktmodells. Die Likelihood-Ratio-Methode ist auch schnell, aber nicht exakt genug wegen des Bias und ihrer großen Varianz. Trotzdem bietet sie die Möglichkeit, die Delta-Vektoren von bermudischen oder amerikanischen Zinsderivaten mit nicht-stetigen Auszahlungen zu berechnen.

Zusammenfassend sind die neue Version der Adjoint-Methode und die Pathwise-Methode unter Forward-Drift sehr empfehlenswert für die Berechnung der Delta-Vektoren von Bermuda-Swaptions.

Für die Berechung der Gamma-Matrix einer Bermuda-Swaption entwickelten wir in dieser Arbeit zwei innovative Methoden, die unseres Wissens bisher nicht publiziert wurden. Eine Methode ist die Pathwise-Methode (die Forward-Methode

und die Adjoint-Methode). Sie entspricht einer exakten und robusten Simulation ohne Bias, die auf pfadweiser Differentialrechnung basiert. Die dadurch simulierte Gamma-Matrix ist hinreichend exakt und perfekt symmetrisch. Leider ist die Pathwise-Methode relativ langsam, insbesondere die Forward-Methode. Die Adjoint-Methode ist zwar ein wenig schneller als die Forward-Methode, aber sie benötigt deutlich mehr Speicherplatz. Trotzdem halten wir die beiden Pathwise-Methoden für die bisher besten Methoden aufgrund ihrer perfekten Simulationsergebnisse.

Die andere Methode ist die modifizierte Finite-Differenzen-Methode. Sie ist die Kombination aus traditioneller Finite-Differenzen-Methode und pfadweiser Differentialrechnung der Delta-Vektoren von Bermuda-Swaptions. Diese Methode ist empfindlich und hat Bias. Allerdings kann man dadurch relativ schnell eine ziemlich präzise und symmetrische Gamma-Matrix einer Bermuda-Swaption berechnen. Diese Methode besteht aus zwei Komponenten: Finite-Differenz und Pathwise-Delta. Die Finite-Differenz hat drei Formen: Vorwärtsdifferenzenquotient, Rückwärtsdifferenzenquotient und zentraler Differenzenquotient. Beim Pathwise-Delta gibt es zwei Möglichkeiten: Forward-Delta und Adjoint-Delta. Deshalb hat die Finite-Differenzen-Methode des Pathwise-Deltas insgesamt sechs konkrete Verfahren. Die Art der Finite-Differenz beeinflusst die Genauigkeit der Simulationsergebnisse. Die durch den zentralen Differenzenquotient berechnete Gamma-Matrix hat einen kleineren Bias als die durch den Vorwärts- und Rückwärtsdifferenzenquotienten, ist also exakter. Was die Effizienz angeht, spielt die Art des Pathwise-Deltas eine entscheidende Rolle. Das Forward-Delta hat einen deutlich höheren Zeitaufwand als das Adjoint-Delta. Deswegen ist der zentrale Differenzenquotient des Adjoint-Deltas die beste Methode im Rahmen der Finite-Differenzen-Methode des Pathwise-Deltas sowohl hinsichtlich der Genauigkeit wie auch der Geschwindigkeit.

Zusammenfassend empfehlen wir die Pathwise-Methode (die Forward-Methode und die Adjoint-Methode) und den zentralen Differenzenquotient des Adjoint-Deltas für die Berechnung der Gamma-Matrizen von Bermuda-Swaptions.

Wie wir schon am Anfang der Arbeit erwähnt haben, gehören die Bermuda-Swaptions zur Klasse der Callable-LIBOR-Exotics und sind mit Abstand die gängigste und wichtigste Art in dieser Klasse. Die anderen Callable-LIBOR-Exotics, z. B. Callable-Capped-Floaters und Callable-Inverse-Floaters, unterscheiden sich von der Bermuda-Swaptions im Prinzip nur in den Kupons X_i. Daraufhin können wir auch den Delta-Vektor und die Gamma-Matrix für die anderen Callable-LIBOR-Exotics direkt durch die von uns entwikelten Methoden numerisch berechnen. Falls die Kupons X_i eines Callable-LIBOR-Exotics nicht Lipschitz-stetig sind, können wir die Likelihood-Ratio-Methode darauf anwenden. Allerdings gilt die Likelihood-Ratio-Methode momentan nur für die Berechnung des Delta-Vektors.

Teil III.

Anhang

A. Funktionale Programmierung und `Haskell` (Teil I)

Die funktionale Programmierung spielt eine entscheidende Rolle in unserem ersten Projekt. Es handelt sich hierbei um einen Programmierstil, bei dem Programme ausschließlich aus Funktionen bestehen. Die funktionalen Programmiersprachen wie `Haskell`, `Erlang`, `Camel` und `F#` sind aber weniger populär als die klassischen imperativen Programmiersprachen wie `Fortran`, `Pascal`, `C` und `Basic` oder die neuerdings in Mode gekommenen objektorientierten Programmiersprachen wie z. B. `C++`, `Java` oder `C#`. Für unsere Projekte haben wir die funktionale Programmiersprache `Haskell` ausgewählt. Die `Haskell`-Ausdrücke werden folglich sehr oft in unserer Arbeit vorkommen. Deshalb ist ein Einführungskapitel in die funktionale Programmierung bzw. `Haskell` notwendig, bevor wir unsere Projekte diskutieren können. Die Ausarbeitung dieses Kapitels orientiert sich an den Arbeiten von Braun [5], Buschmann [6], Daume III [8], Newbern [27], O'Sullivan, Stewart und Goerze [28], Peyton-Jones [29] und Puhlmann [33].

A.1. Funktionale Programmierung

In diesem Abschnitt stellen wir die Entwicklung und die Anwendungen der funktionalen Programmierung kurz vor. Zuerst erläutern wir einige Begriffe.

A.1.1. Was ist funktionale Programmierung

Programmierungen kann man nach dem sogenannten Programmierstil klassifizieren. Eine grobe Unterteilung ist die zwischen imperativer und deklarativer Programmierung.

Funktionale Programmierung ist ein Programmierstil der deklarativen Programmierung, bei der die Programme rein aus Funktionen gebildet sind. Sie stammt ursprünglich aus der akademischen Forschung. In den 30er Jahren arbeiteten verschiedene Wissenschaftler am λ-Kalkül. Dies ist eine formale Sprache zur Untersuchung der Berechenbarkeit von bestimmten Funktionen und ihrer Auswertungen. Der λ-Kalkül bildet einen Teil der formalen Grundlage der theoretischen Informatik und insbesondere der funktionalen Programmierung. Während in der imperativen Programmierung das Konzept der Variablen als Speicherbereich für Werte im Vordergrund steht, ist in der funktionalen Programmierung der Begriff der Funktion von einer zentralen Bedeutung.

Die zentrale Anweisung in der imperativen Programmierung ist die Zuweisung. Ein Problem wird durch die schrittweise Veränderung des Zustandsraumes, nämlich der Werte der Variablen, gelöst. D. h. es wird beim Programmieren gesagt, "wie" ein Problem bzw. eine Aufgabe zu lösen ist. Dabei spielt die Reihenfolge der Anweisungen (Statements) eine entscheidende Rolle.

In der funktionalen Programmierung werden Berechnungen hingegen vornehmlich durch die Auswertung von Ausdrücken (Menge von Funktionen) durchgeführt. Bei funktionaler Programmierung wird vom "wie" weitgehend abstrahiert und stattdessen das "was" durchgeführt, nämlich eine Beschreibung des gewünschten Ergebnisses. Das genaue Vorgehen bei der Berechnung des Ergebnisses ist normalerweise nicht explizit in dieser Beschreibung enthalten.

Funktionale Programmierung trägt wesentlich zum Verständnis dessen bei, was Programmierung eigentlich ist, wobei einige ganz neue Aspekte hinzugekommen sind, die man in der traditionellen Programmierungen an keiner Stelle findet.

A.1.2. Die Entwicklung der funktionale Programmierung

Der Ursprung der funktionalen Programmierung liegt im Bereich Mathematik. Church hat 1932 den λ-Kalkül entwickelt, dessen Nutzen in der Programmierung 1936 von Kleene und Rosser bewiesen wurde, d. h. es wurde nachgewiesen, dass man den λ-Kalkül zur Umsetzung von Funktionen in der Informatik nutzen kann. Die heutigen, rein funktional geschriebenen Programme können komplett in den λ-Kalkül umgewandelt werden.

1937 hat Turing bewiesen, dass der λ-Kalkül turing-äquivalent ist. D. h., der λ-Kalkül entspricht als formales Modell zum Beweisen der Programmiersprachen der Turingmaschine.

1960 wurde `LISP` als die erste funktionale Programmiersprache von McCarthy entwickelt. Ein großer Vorteil von `LISP` liegt in ihrer Erweiterbarkeit. Somit wurden einige erweiterte Programmiersprachen entwickelt, z. B. `Common LISP` and `Scheme`.

Ende der 70er entwickelte Backus das `FP`-System, ein Konzept funktionaler Programmiersprachen. Er hatte die Idee von Funktionen höherer Ordnung und setzte diese in `FP`-System um.

Gleichzeitig entwickelte man an der Universität Edinburgh Meta Language (`ML`). Man suchte nach einer Programmiersprache, mit der man Theoreme beweisen konnte. Auch aus `ML` entstanden einige Dialekte, z. B. `SML` und `CAML`.

Eine weitere Programmiersprache auf dem Weg bis zur Entwicklung von `Haskell` ist `Miranda`. `Miranda` wurde zwischen 1985 und 1986 entwickelt. In `Miranda` benutzte man zum ersten Mal die Lazy-Evaluation.

Die heute in der Industrie oft genutzte funktionale Programmiersprache ist `Erlang`, das Ende der 80er von Ericsson für Anwendungen in der Telekommunikation entwickelt wurde. Mit `Erlang` ist parallele funktionale Programmierung

möglich.

Haskell entstand erst 1990 (Version 1.0). In Haskell wurden die Ideen der älteren funktionalen Programmiersprachen vereinigt, nämlich Funktionen höherer Ordnung, Typableitung und Lazy-Evaluation. Eine Neuheit von Haskell sind die Typklassen, mit denen man Methoden auf ähnlich Art und Weise wie in der objektorientierten Programmierung überladen kann.

Bis 1997 entstanden vier weitere Versionen (1.1 bis 1.4) von Haskell. Auf dem Haskell Workshop 1997 in Amsterdam wurde beschlossen, dass eine stabile einheitliche Variante nötig sei. Hierdurch entstand die heutige sehr bekannte rein funktionale Programmiersprache - Haskell 98.

A.1.3. Die Anwendungen der funktionale Programmierung

Funktionale Programmierung ist vor allem in akademischen Kreisen von Interesse. So halten viele Leute Haskell für eine PHD-Sprache. Sie wird insbesondere im Bereich künstliche Intelligenz, Compilerbau und Computeralgebra eingesetzt. Jedoch werden die funktionalen Programmiersprachen heutezutage immer mehr an Bedeutung gewinnen, nicht nur auf der konzeptuellen Ebene akademischer Studien, sondern auch auf der praktischen Ebene industrieller Anwendungen. Mit funktionalen Programmiersprachen kann die Produktivität der Programmierer gesteigert und somit können die Softwarekosten gesenkt werden. Einige prominente funktionale Programmiersprachen sind schon in kaufmännischen und industriellen Anwendungen eingesetzt worden. Z. B. setzte die ABN AMRO Bank Haskell für die Simulation im Bereich Kreditrisikoberechnung ein. Ferner verwendete die Quantitative Analytics Gruppe von Barclays Capital Haskell um Exotic Equity Derivatives anzugeben und die Directional Credit Trading Gruppe der Deutschen Bank benutzte Haskell als primäre Programmiersprache für die Implementierung ihrer Software-Infrastruktur. Für weitere reale Anwendungen funktionaler Programmierung kann man in der Webseite "Haskell in Industry":

http://www.haskell.org/haskellwiki/Haskell_in_industry

und der jährlichen Konferenz "Commercial Users of Functional Programming (CUFP)":

http://cufp.galois.com/

nachschlagen.

A.2. Programmierung mit Haskell

In diesem Abschnitt befassen wir uns mit der rein funktionalen Programmiersprache Haskell. Rein funktionale Programmierung fasst Programme lediglich

als mathematische Funktionen auf. Ein Ausdruck hat dort während der Programmausführung immer den gleichen Wert, der nicht vom Zeitpunkt der Auswertung abhängt. Es gibt keine Zuweisungen, keine Zustände, keine Seiteneffekte und keine Zustandsvariablen, die während einer Berechnung geändert werden. Die meisten anderen funktionalen Programmiersprachen (z. B. Standard ML, Caml, F#) erlauben hingegen die Wirkungen von Benutzerinteraktionen, Eingabe/Ausgabe-Operationen u. s. w. und gehören daher nicht zu den rein funktionalen Programmiersprachen. Um eine rein funktionale Programmiersprache auch mit solchen Wirkungen auszustatten, ohne dabei die reine Funktionalität aufzugeben, wurde das aus der Kategorientheorie stammende Konzept Monad entwickelt, insbesondere von Eugenio Moggi und Philip Wadler. Das Konzept Monad drückt Wirkbehaftung durch parametrische Datentypen aus und zwingt somit das Datentypsystem dazu, zwischen Ausdrücken mit und ohne Nebenwirkungen zu unterscheiden.

Die behandelten Themen dieses Abschnitts umfassen grundlegende Konzepte, algebraische Datentypen, musterbasierte Funktionsdefinitionen, Listenkomprehensionen, parametrische Typpolymorphie, Funktionen höherer Ordnung, verzögerte Auswertung, unendliche Datenstrukturen, Typklassen und Monad, welche in dem Haskell-Code unserer Projekte eingesetzt werden. Die in Haskell vermittelten Kenntnisse fördern auch die Denkart und Vorgehensweise in anderen funktionalen Programmiersprachen.

A.2.1. Was ist Haskell

Haskell ist eine moderne rein funktionale Programmiersprache, benannt nach dem US-amerikanischen Mathematiker Haskell Brooks Curry, dessen Arbeiten zur mathematischen Logik eine Grundlage funktionaler Programmiersprachen bilden. Haskell basiert auf dem λ-Kalkül, weshalb der griechische Buchstabe λ als Logo verwendet wird. Die offizielle Webseite zur Programmiersprache Haskell ist:

http://www.haskell.org

Dort finden sich auch die wichtigsten Haskell-Compiler und Haskell-Interpreter, Glasgow Haskell Compiler (GHC):

http://haskell.org/ghc/index.html

und Hugs 98:

http://www.haskell.org/hugs/

zum Herunterladen.

Um einen ersten Eindruck vom Haskell-Programm bzw. vom funktionalen Programmierungsstil zu gewinnen, vergleichen wir die Implementierung des bekannten Quicksortalgorithmus durch Haskell mit der durch C, die nach dem Prinzip “Teile und herrsche” arbeitet. Wir betrachten zunächst den C-Code:

Programm A.1: C-Version von Quicksort

```
1 void qsort(int a[ ], int lo, int hi) {
2   int h, l, p, t;
3   if (lo < hi) {
4     l = lo;  h = hi;  p = a[hi];
5     do {
6       while ((l < h) && (a[l] <= p))
7         l = l+1;
8       while ((h > l) && (a[h] >= p))
9         h = h-1;
10      if (l < h) {
11        t = a[l];  a[l] = a[h];  a[h] = t;
12      }
13    } while (l < h);
14    a[hi] = a[l];  a[l] = p;
15    qsort(a, lo, l-1);  qsort(a, l+1, hi);
16  }
17 }
```

C ist vielen Programmierern bekannt, weswegen wir diesen C-Code nicht weiter erklären. Nun geben wir den ensprechenden Haskell-Code an:

Programm A.2: Haskell-Version von Quicksort

```
1 qsort :: Ord a => [a] -> [a]
2 qsort []     = []
3 qsort (x:xs) = (qsort [y | y <- xs, y < x]) ++ [x] ++
4                (qsort [y | y <- xs, y >= x])
```

Die erste Zeile typisiert den Datentyp (die Signatur) der Funktion qsort. Die zweite Zeile gibt an, dass die Funktion qsort auf eine leere Liste angewendet wieder eine leere Liste ergeben soll. Die dritte Zeile sortiert rekursiv die nichtleeren Listen. Das erste Element x wird als mittleres Element der Ergebnisliste verwendet. Davor werden alle kleineren und dahinter alle größeren Elemente eingeordnet. Listenbeschreibungen (hier: List-Comprehension) werden dazu verwendet, aus der Restliste xs alle kleineren bzw. größeren Elemente als x auszuwählen.

Durch den obigen Vergleich ist es klar, dass Haskell kürzer und übersichtlicher ist als C, das Prinzip vom Quicksortalgorithmus zu beschreiben. Das ist ein Vorteil von Haskell und auch den anderen funktionalen Programmiersprachen. Im weiteren Verlauf werden wir die Grammatik von Haskell explizit vorstellen.

A.2.2. Datenstrukturen

Die Möglichkeit, aus den Standartdatentypen (in Haskell: Int, Double, Bool, Char, String u. s. w.) neue Datentypen konstruieren zu können, trägt zur Aus-

drucksstärke einer Programmiersprache bei. Haskell bietet neben dem Bilden von Funktionsdatentypen (z. B. Int -> Double) noch zwei wichtige Standardkonstruktionen: Listen und Tupel, und die von Programmierern definierten Datentypen.

Liste

Ein beliebtes Konzept in Programmiersprachen sind die sogenannten Arrays. Die sind Ansammlungen von Daten, die zu dem gleichen Datentyp gehören, aber aus verschiedenen Elemente bestehen und alle auf einmal übergeben werden, um sie zu verarbeiten. In Haskell gibt es dazu ein ähnliches Konstrukt. Ist a ein Haskell-Datentyp, so ist [a] der Listentyp einer Liste von Elementen des Datentyps a. Die Objekte des Listentypes [a] sind folgendermaßen aufgebaut:

- Die leere Liste [ ] ist eine Liste vom Datentyp [a].
- Ist x ein Element vom Datentyp a und xe eine Liste vom Datentyp [a], so ist (x:xe) ebenso eine Liste vom Datentyp [a].

Im Folgenden geben wir einige Beispiele von Listentypen:

```
1 [Int]                -- Int-Liste
2 [Float]              -- Float-Liste
3 [Char]               -- Char-Liste mit Synonym: [Char] = String
4 [[Bool]]             -- Liste von Bool-Listen
5 [Double -> Double] -- Liste von (Double -> Double)-Funktionen
```

Listen kann man aufschreiben, indem man ihre Elemente innerhalb eckiger Klammern durch Kommata getrennt angibt, z. B. [1,2,3,4,5,6] :: [Int]. Listen sind normalerweise keine Mengen im mathematischen Sinne, es kommt also auf die Reihenfolge und Anzahl der Vorkommnisse von Elementen an, wobei die Anzahl unendlich sein kann. Die leere Liste [ ] stellt hinsichtlich ihres Datentyps einen besonderen Fall dar. Sie kann jeden Listentyp annehmen. Welchen Datentyp sie im Einzelfall hat, hängt vom Zusammenhang ihrer Anwendung ab.

Eine Liste hat noch eine andere Darstellungsmöglichkeit, die mit mathematischem Sinn verbunden ist. In der Mathematik gibt es eine Darstellung für Mengen, deren Elemente bestimmte Eigenschaften besitzen. Beispielsweise wäre eine Untermenge von $\{1, \ldots, 100\}$, in der nur gerade Zahlen stehen:

$$\{x \mid x \in \{1, \ldots, 100\} \wedge \{\text{x ist gerade}\}\}\,.$$

Ein ähnliches Konstrukt gibt es auch für das Bilden von Listen in Haskell. Die sogenannte List-Comprehension ist eine beliebte Art und Weise, um mit wenig Aufwand eine Liste zu beschreiben. Der Haskell-Code von der obigen Menge ist:

```
1 [x | x <- [1..100], even x]
```

Die List-Comprehension sieht also ähnlich der korrespondierenden mathematischen Darstellung aus.

Tupel

Während Listen eine beliebige Anzahl (auch unendlich ist erlaubt) von Elementen desselben Datentyps enthalten können, dient Tupel der Zusammenfassung einer festen Anzahl von Elementen, die jedoch unterschiedliche Datentypen haben dürfen. Tupel ist daher der Oberbegriff für die Aggregation verschiedener Größe.

In `Haskell` wird Tupel im Paar von runden Klammern geschrieben, die Elemente werden durch die Kommata getrennt. Ebenso wird der Datentyp eines Tupels in runden Klammern angegeben, in dem der Datentyp jedes Elements für die entsprechende Position innerhalb des Tupels angegeben sind. Nachfolgend sind einige Beispiele von Tupel zusammen mit ihren Datentypangaben gegeben:

```
1 (1, 2.0) :: (Int, Double)
2 (1.0, 2.0) :: (Double, Double)
3 (1, "Haskell") :: (Int, String)
4 (True, 1, [False]) :: (Bool, Int, [Bool])
5 (('a', "b"), 1.0) :: ((Char, String), Double)}
```

Polymorphe Datentypen

Nicht immer ist ein Datentyp in `Haskell` so genau wie oben angegeben. Betrachten wir beispielsweise die Identitätsfunktion. Um diese Funktion für alle mögliche Datentypen zu verallgemeinern, brauchen wir also einen polymorphen (vielfältigen) Datentyp. D. h. wir dürfen in der Signatur keinen Standardatentyp wie etwa `Int`, `String` oder `Float` angeben. Stattdessen schreiben wir dort einen Platzhalter (meistens mit einem Buchstabe) hin.

```
1 id :: a -> a
2 id x = x
```

Der Rückgabewert der Funktion `id` ist also gleich dem Eingabedatentyp des ersten Parameters. Wollten wir beispielsweise noch eine Signatur für eine Funktion `f` konstruieren, deren zweiter und dritter Parameter andere Datentypen als der Erste haben, so werden dort andere Platzhalter verwendet.

```
1 f :: a -> b -> c
```

Typ-Synonyme

Wenn man für einen schon vorhandenen Datentyp ein neues Synonym einführen möchte, um z. B. die Verwendung klarer zu machen, bedient man sich des Befehls `type` in der folgenden Art:

```
type neuerTyp = bekannterTyp
```

Dabei kann der bekannte Datentyp entweder ein Haskell-Standarddatentyp oder einer der komplexeren Datentyp, wie z. B. ein Tupel oder eine Liste sein:

```
1 type Abstand = Float
2 type Punkt = (Float,Float)
```

Anhand des Datentyps Punkt kann beispielsweise die Funktion zur Bestimmung des Abstands zweier Punkte anschaulich definiert werden:

```
1 abstand:: Punkt -> Punkt -> Abstand
2 abstand (a,b) (c,d) = sqrt((a-c)^2 + (b-d)^2)
```

Oftmals stellt man verschiedene Sachverhalte mittels der gleichen Repräsentation dar. Z. B. sind Seitenlänge und Radius verschiedene Objekte, aber beide können als Abstand dargestellt werden. Mit der Typdefinition type sind Seitenlänge und Radius beliebig austauschbar, was unerwünscht ist. Einen eigenen Datentyp mit Konstruktor zu definieren erscheint hier praktisch, ist aber mit zusätzlichem Aufwand beim Pattern-Matching (siehe A.2.4) verbunden. Für solche Fälle gibt es in Haskell das Schlüsselwort newtype, z. B.:

```
1 newtype Punkt = Punkt (Float ,Float)
```

Nun hat der Datentyp Punkt einen einzigen Konstruktor Punkt. Trotz derselben Namen handelt es sich hier um verschiedene Objekte, nämlich Datentyp und Konstruktor. Man beachte, dass bei type und newtype grundsätzlich nur ein Konstruktor erlaubt ist. Um mehrere Konstruktoren in einem Datentyp zu erstellen, brauchen wir die algebraischen Datentypen.

Algebraische Datentypen

Wir erweitern jetzt das Datentypsystem von Haskell, in dem es neben den Standarddatentypen (Int, Double, Char, Bool u. s. w.), Datentypkonstruktoren (Liste und Tupel) und Typ-Synonyme (type und newtype) auch algebraische Datentypen gibt.

Analog zum Datentypkonstruktor Liste, der die beiden Konstruktoren (:) und [ ] einführt, um Werte des Datentyps [a] zu konstruieren, kann der Programmierer neue Konstruktoren definieren, um Werte eines neuen algebraischen Datentyps zu erzeugen. Und wie es bei Listen und Tupeln möglich ist, können Werte dieser neuen algebraischen Datentypen dann mittels Pattern-Matching (siehe A.2.4) wieder analysiert (dekonstruiert) werden. Das allgemeine Schema eines neuen algebraischen Datentyps wird in Haskell mittels der data-Deklaration spezifizert, die aus den folgenden Bausteinen besteht:

- Der Name des Datentypkonstruktors (Kennung beginnend mit den Großbuchstaben) T und sein Datentypparameter α_j.

- Die Namen der Datentypkonstruktoren (Kennung beginnend mit den Großbuchstaben) K_i und der Datentypen β_{ik}, die diese als Parameter erwarten.
- Die Syntax einer `data`-Deklaration ($n \geq 0$, $m \geq 1$, $n_i \geq 0$, die β_{ik} sind entweder gleich einem von $\alpha_1, \alpha_2, \ldots, \alpha_n$ oder einem anderen Typbezeichner):

$$\begin{aligned} \textbf{data}\ T\ \alpha_1\ \alpha_2\ \ldots\ \alpha_n\ = &\ K_1\ \beta_{11}\ \ldots\ \beta_{1n_1} \\ \mid &\ K_2\ \beta_{21}\ \ldots\ \beta_{2n_2} \\ \vdots & \\ \mid &\ K_m\ \beta_{m1}\ \ldots\ \beta_{mn_m} \end{aligned}$$

  Dieses `data`-Statement deklariert einen Typkonstruktor T und m Konstruktoren und besitzt die umgekehrte Funktionsdarstellung:

$$K_i :: \beta_{i1} \to \cdots \to \beta_{in_i} \to T\ \alpha_1\ \alpha_2\ \ldots\ \alpha_n \tag{A.1}$$

Nachfolgend stehen zwei Arten von algebraischen Datentypen mit zwei Code-Beispielen:

- Lineare Datentypen:

```
1 data Jahreszeiten = Fruehling | Sommer | Herbst | Winter
2                   deriving (Eq, Show)
```

- Rekursive Datentypen:

```
1 data BinTree a = Empty | Node (BinTree a) a (BinTree a)
2                deriving (Eq, Show)
```

wobei `deriving (Show, Eq)` die sogenannte Ableitung von Typklassen (siehe A.2.3) bezeichnet, d. h. die neuen Datentypen `Jahreszeiten` und `BinTree` werden damit automatisch Instanzen der Typklasse `Show` aller druckbaren Datentypen und Instanzen der Typklasse `Eq` aller Datentypen, deren Objekte durch das Gleichheitszeichen `(==)` vergleicht werden können.

A.2.3. Typklassen

Eine Besonderheit von `Haskell` sind die Typklassen. Aus `Java` kennt man die Überladung von Methoden bzw. Funktionen aus Klassen. `Haskell` versucht diese Idee auch für sich zu nutzen, in dem es ähnliche Möglichkeiten für Datentypen bietet. Dazu gibt es in `Haskell` eine Klassen-Hierarchie von den Standarddatentypen, namlich die Typklassen.

Bei den Typklassen kann es sein, dass unsere Anforderungen einen Datentyp verlangen, welcher nicht so allgemein gefasst ist, wie ein polymorpher Datentyp,

aber auch nicht so speziell ausgeprägt ist, wie ein konkreter Datentyp. Beispielsweise möchten wir eine Funktion (+) zum Addieren zweier Werte nicht für alle möglichen Datentypen und auch nicht für einen konkreten Datentyp definieren, sondern nur für alle möglichen Datentypen, die die Zahlen darstellen. Es wird also eine Möglichkeit gesucht, die einen Datentyp liefert, der sich zwischen den generellen polymorphen Datentypen und den speziellen konkreten Datentypen bewegt.

```
(+) :: a -> a -> a              -- zu generell
(+) :: Int -> Int -> Int        -- zu speziell
(+) :: Num a => a -> a -> a     -- perfekt
```

Mit Hilfe der Typklasse **Num** können wir festlegen, dass der Datentyp, für den wir unsere Funktion definieren, bestimmte Eigenschaften vorweist. Das => Symbol gibt an, dass der Datentyp a Mitglied einer bestimmten Klasse sein muss, damit er mit dieser Funktion arbeiten kann. In dem obigen Beispiel ist die Typklasse **Num** eine Oberklasse für alle Zahlen, in der die Instanzen von a addiert werden dürfen. Die in Haskell eingebauten Standarddatentypen Double und Int sind beide Mitglieder der Typklasse **Num**.

Neben **Num** bietet Haskell noch viele weitere vordefinierte Typklassen, wie z. B. **Eq**, deren Instanz Äquivalenzgesetze durchführen kann, und ihre obere Typklasse **Ord** für die Vergleichsoperationen. Wir sehen unten die Definition der Typklasse Eq:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool

  x == y  =  not (x /= y)     -- minimale vollständige Definition
  x /= y  =  not (x == y)
```

Diese Typklasse trägt den Namen **Eq** und soll alle diejenigen Datentypen zusammenfassen, für die die Operatoren (==) und (/=) definiert sind. Dabei entspricht (==) der minimalen vollständigen Definition. D. h., dass jeder Datentyp von **Eq** zumindest die Operation (==) definieren muss, um eine valide Instanz dieser Typklasse zu werden. Für einen einfachen Datentyp wie Int könnte dies so aussehen:

```
instance Eq Int where
  x == y = intEq x y
```

wobei die Funktion intEq die grundlegende Funktion zum Vergleichen der Int-Werte ist. Auch der folgende rekursive Datentyp Tree:

```
data Tree a = Leaf a | Branch Tree a Tree a
```

lässt sich als eine Instanz der Typklasse **Eq** auffassen durch die folgende Definition von (==):

```
instance Eq a => Eq (Tree a) where
  Leaf a == Leaf b             = a == b
  Branch l1 r1 == Branch l2 r2 = l1 == l2 && r1 == r2
  _ == _                       = False
```

Der Operator `(==)` wurde hier mehrfach implementiert. Je nach dem, welches Muster passt, also ob `Leaf` mit `Leaf` oder `Branch` mit `Branch` verglichen werden soll, wird der hierzu entsprechende Ausdruck ausgewertet. `\_ == \_` ist ein Muster, welches auf alle Gleichheitsoperationen passt (siehe Wild-Cards auf der Seite 200). Da auch der Datentyp ausgewertet wird, mit dem die Baumstruktur aufgebaut ist und der hier ja polymorph definiert ist, muss also angegeben werden, dass seine mögliche spätere Ausprägung eine Instanz der Typklasse **`Eq`** besitzen muss. Dies wird mit dem `Haskell`-Ausdruck **`Eq`** `a => Eq (Tree a)` sichergestellt.

A.2.4. Pattern-Matching

Ein weiteres, wichtiges Konzept in `Haskell` ist das sogenannte Pattern-Matching. Eine Funktion kann mehrere Definitionen haben. Es wird per Pattern-Matching entschieden, welche Definition anhand der aufgerufenen Parameter angewendet wird. Z. B.:

```
sum :: Num a => [a] -> a
sum [] = 0
sum (x:xs) = x + sum xs
```

Wenn `sum` mit der leeren Liste aufgerufen wird, wird die erste Definition angewendet. Wenn die Liste nicht leer ist, die zweite. In diesem Fall wird das erste Element der Liste an die Variable `x` gebunden und der Rest der Liste an die Variable `xs`.

Man muss beachten, dass die Reihenfolge einer Funktion mit mehren Definitionen sehr wichtig ist. Diese werden immer von oben nach unten bewertet und wenn eine Definition durch eine Eingabe benutzt und bewertet wird, wird der Bewertungsprozess von der Funktion sofort fertiggestellt. Anhand dieser Regel können `Haskell`-Programmierer eine Funktion flexibler beschreiben.

`Haskell` bietet noch viele weitere Möglichkeiten von Pattern-Matching, die wir im Folgenden kurz anhand von Beispielen erklären werden.

As-Pattern

Als Beispiel ist hier eine Funktion, die das erste Element einer Liste dupliziert:

```
f (x:xs) = x:(x:xs)
```

Diese Funktion können wir alternativ auch wie folgt aufschreiben:

```
f s@(x:xs) = x:s
```

Dabei wird x an das erste Element der Liste gebunden, xs an die Restliste und s an die komplette Liste.

Wild-Cards

Wild-Cards sind nützlich, wenn wir gegen einen Ausdruck matchen wollen, dessen Komponenten uns nicht interessieren. Z. B.:

```
1 head (x:_) = x
2 tail (_:xs) = xs
```

Hier matchen die Ausdrücke jeweils eine nicht-leere Liste. Die Wild-Cards matchen dabei eine Liste beliebiger Länge.

If-Pattern

Wie andere Programmiersprachen hat Haskell auch den **if**-Ausdruck. Z. B.:

```
1 fromBoolToInt :: Bool -> Int
2 fromBoolToInt x = if x then 1
3                        else 0
```

With-Pattern

Man kann mit dem With-Pattern die Funktion fromBoolToInt wie folgt umschreiben:

```
1 fromBoolToInt :: Bool -> Int
2 fromBoolToInt x | x          = 1
3                 | otherwise  = 0
```

Case-Pattern

Auch kann man mit dem Case-Pattern die Funktion fromBoolToInt so beschreiben:

```
1 fromBoolToInt :: Bool -> Int
2 fromBoolToInt x = case x of
3                        True  -> 1
4                        False -> 0
```

let und where Klausel

Mit den Schlüsselwörtern **let** und **where** kann man lokale Definitionen vornehmen:

```
1 sumSquares x y = let square n = n * n
2                  in square x + square y
```

```
1 sumSquares x y = square x + square y
2                  where
3                    square n = n * n
```

A.2.5. Lazy-Evaluation

Haskell hat noch eine interessante Eigenschaft, die sogenannte Lazy-Evaluation (Bedarfauswertung). Die Lazy-Evaluation bedeutet, dass Ausdrücke immer erst dann ausgewertet werden, wenn sie benötigt werden.

Die Lazy-Evaluation bietet den Programmierern die Möglichkeit, unendliche Datenstrukturen zu verwenden. Stellen wir uns mal vor, dass wir eine unendliche Liste in einer Funktion verarbeiten wollen. Die meisten Programmiersprachen würden eine solche Funktion erst dann ausführen, sobald alle Parameter abgeschlossen wurden. Haskell wartet jedoch dank der Lazy-Evaluation nur solange bis es arbeiten kann. Ein anderer Vorteil einer solchen Auswertungsstrategie ist die Zeitersparnis, da Funktionsaufrufe ganz vermieden oder zumindest teilweise eingespart werden können. Der Nachteil besteht dann in der erschwerten Fehlerbereinigung (Debugger) in Programmen, die nach Bedarf ausgewertet werden. Denn es ist meistens nicht auf den ersten Blick nachvollziehbar, ob ein Ausdruck zu einem Zeitpunkt bereits ausgewertet wurde.

A.2.6. Funktionen höherer Ordnung

In Haskell spielen die Funktionen eine zentrale Rolle. Sie können selbst als Datenobjekte gelten, d. h. man kann Funktionen auch als Argumente und Ergebnisse anderer Funktionen verwenden. Funktionen, die mit Funktionen als Eingabe arbeiten, nennt man Funktionen höherer Ordnung.

Oft kann man solche Funktionen mithilfe einiger Standardfunktionen von Haskell definieren. Die Funktionen map, filter, foldl, foldr, zip und zipWith sind beispielsweise Funktionen höherer Ordnung zur Listenbearbeitung und gehören zum Sprachumfang von Haskell. Sie machen die Listenbearbeitung leichter und übersichtlicher, indem explizite Rekursion vermieden und in die Funktionen höherer Ordnung ausgelagert wird. Im Folgenden werden wir diese nützlichen Funktionen kurz vorstellen.

map

```
1 map :: (a -> b) -> [a] -> [b]
```

wendet die übergebene Funktion auf jedes einzelne Element der Liste an, und bildet aus den so berechneten Elementen die Ergebnisliste, die zurückgegeben wird. Der Aufruf der `map`-Funktion erfolgt so:

```
map (Operation)Liste
```

Diese Funktion kann man z. B. verwenden, um alle Elemente einer Liste von Zahlen zu verdoppeln:

```
1 doppel:: Num a => [a] -> [a]
2 doppel liste = map (2*) liste
```

Anzumerken ist noch, dass die `Operation` beliebig kompliziert sein kann, solange sie auf die einzelnen Elemente der Liste angewendet werden kann.

filter

```
1 filter :: (a -> Bool) -> [a] -> [a]
```

wendet eine angegebene boolesche Funktion auf jedes Element der übergebenen Liste an, und prüft das Ergebnis. Ist es **`True`**, so wird das Element an die zu erstellende Ergebnisliste angehängt, bei **`False`** nicht. Diese Funktion könnte man beispielsweise verwenden, um aus einer Liste von Zahlen alle negativen Werte zu entfernen:

```
1 ohnenegative:: Num a => [a] -> [a]
2 ohnenegative liste = filter (>=0) liste
```

foldl

```
1 foldl :: (a -> b -> a) -> a -> [b] -> a
```

berechnet aus einer Liste von Elementen ein Ergebniselement, indem die Elemente der Liste nacheinander (beginnend mit einem zusätzlich angegebenen Anfangswert) mittels `Operation` verknüpft werden, die als Parameter übergeben wird. Dabei wird mit den Elementen am Anfang der Liste begonnen, und die Berechnung arbeitet sich nach hinten durch. Der Aufruf der `foldl`-Funktion erfolgt so:

```
foldl (Operation)Anfangswert Liste
```

Der folgende `Haskell`-Code dient als Beispiel:

```
1 foldr (+) 0 [1,2,3,4]
```

Das Resultat ist 10.

foldr

```
1 foldr :: (a -> b -> b) -> b -> [a] -> b
```

arbeitet analog zu `foldl`. Der Unterschied besteht darin, dass die Liste nicht von vorne nach hinten durchgearbeitet wird, sondern umgekehrt mit dem letzten Element beginnend nach vorne durchlaufen wird. Beispielsweise ist 10 auch das Ergebnis von:

```
1 foldr (+) 0 [1,2,3,4]
```

zip

```
1 zip :: [a] -> [b] -> [(a,b)]
```

übernimmt zwei Listen als Parameter, und bildet die Ergebnisliste dadurch, dass aus den beiden Listen jeweils ein Element entnommen wird und diese beiden Elemente dann zu einem Tupel zusammengefügt werden, bis das Ende der Ergebnisliste erreicht wird. Das Ergebnis von

```
1 zip [0,1] [False,True]
```

ist:

```
1 [(0,False), (1,True)]
```

zipWith

```
1 zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

übernimmt zwei Listen als Parameter, sowie eine Funktion, die auf den Datentypen der Listen arbeiten kann, und errechnet eine neue Liste, indem jeweils das nächst aktuelle Element jeder der beiden Listen mit der angegebenen `Operation` verknüpft wird, und in die Ergebnisliste hinzugefügt wird. Der Aufruf der `zipWith`-Funktion erfolgt wie folgt:

```
zipWith (Operation)Liste1 Liste2
```

Mit dieser Funktion (und etwas zusätzlicher Hilfe von `foldl`) ist es sehr einfach, das Standardskalarprodukt beliebig langer Listen zu berechnen:

```
1 skalarprodukt:: Num a => [a] -> [a] -> a
2 skalarprodukt x y = foldl (+) 0 (zipWith (*) x y)
```

λ-Abstraktionen

In `Haskell` können sogar λ-Ausdrücke des λ-Kalküls für Berechnungen vererbt werden, was auch sehr nützlich bei Funktionen höherer Ordnung ist. Die Notation erfolgt angelehnt an die λ-Schreibweisen (statt "λ" schreibt man "\"). Z. B.:

```
1 add :: Int -> Int -> Int
2 add = \x y -> x + y
```

ist gleichbedeutend mit der normalen `Haskell`-Funktion:

```
add:: Int -> Int -> Int
add x y = x + y
```

Die in dem λ-Ausdruck gebundenen Variablen stehen also nach einem "\", ggf. getrennt durch ein Leerzeichen, falls mehr als eine Variable gebunden werden sollen, und der Rumpf der λ-Abstraktion steht nach einem Pfeil der Form "->".

"$", "|>" und "."

Die drei folgenden Operatoren sind auch bei Funktionen höherer Ordnung hilfreich:

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

```
(|>) :: a -> (a -> b) -> b
x |> f = f x
```

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)
```

Nun sehen wir den Zusammenhang der drei Operatoren:

```
f(g(x)) == f $ g $ x == x |> g |> f == (f.g) x
```

A.2.7. Monad

Wie wir schon erwähnt haben, kann Haskell nach dem Prinzip der rein funktionalen Programmiersprache Seiteneffekte vermeiden. Dennoch kann natürlich keine Programmiersprache, die in der realen Welt eingesetzt werden soll, vollständig auf Seiteneffekte verzichten. Wenn man beispielsweise in Haskell die Berechnungen in Programmen sequentiell anordnen oder Eingabe/Ausgabe-Operationen unternehmen möchte, so trifft man häufig auf Schwierigkeit, während die imperativen Programmiersprachen wie C solche Probleme leicht umgehen. Wie kann man also in der rein funktionalen Programmiersprache Haskell Seiteneffekte ausdrücken? Die bekannteste Technik hierfür ist das sogenannte Monad, das eine Struktur im mathematischen Teilgebiet der Kategorientheorie ist, die gewisse formale Ähnlichkeit mit dem Monoid der Algebra aufweist.

Nach Definition ist ein Monad ein Datentypkonstruktor m mit den Operationen return und (>>=). Dies wird auch bind genannt. Dabei ist m a der Datentyp einer Berechnung, die den inneren Datentyp a liefert. Man verwendet return für die triviale Berechung und bind für eine Sequenz von Berechnungen. In Haskell sieht die Typklasse Monad so aus:

```
class Monad m where
  -- Minimale vollstaendige Definition: (>>=), return
```

```
3   return :: a -> m a
4   (>>=)  :: m a -> (a -> m b) -> m b
5   (>>)   :: m a -> m b -> m b
6   fail   :: String -> m a
7
8   m >> n = m >>= const n
9   fail s = error s
```

Die Operation `(>>)` wird dabei so angewendet wie `bind`, jedoch mit dem Unterschied, dass bei `(>>)` das Vorergebnis der Berechnung keine Rolle spielt und nicht mit diesem übergeben wird. Wie man unschwer erkennen kann, dient die Operation `fail` der Fehleranzeige.

Wir behaupten, dass ein gut funktionierendes Monad die folgenden, sogenannten Monadengesetze erfüllen muss:

```
1 return a >>= k     == k a
2 m >>= return       == m
3 (m >>= k1) >>= k2 == m >>= (\a -> k1 a >>= k2)
```

Für jede Instanz von Monad, die die obigen Monadengesetze erfüllt, kann man mit der **do**-Syntax einen monadischen Code in den imperativ ähnlichen Code umschreiben. Die imperative Syntax **do** bietet eine einfache Abkürzung für die Ketten der monadischen Operationen. Die wesentliche Übersetzung von **do** wird in den folgenden zwei Regeln erfasst:

```
1 do e1 ; e2      = e1 >> e2
2 do p <- e1; e2 = e1 >>= \p -> e2
```

Eine wichtige und oft verwendete Instanz von Monad ist die Klasse `MonadPlus`, die für die Monaden genutzt wird und ein Null-Element sowie einen Plus-Operator aufweisen:

```
1 class (Monad m) => MonadPlus m  where
2   mzero :: m a
3   mplus :: m a -> m a -> m a
```

Die beiden Elemente genügen den folgenden Gesetzen:

```
1 mzero >>= f          == mzero
2 m >>= (\x -> mzero) == mzero
3 mzero `mplus` m      == m
4 m `mplus` mzero      == m
```

Nachdem wir die Grundbegriffe von Monad vollständig vorgestellt haben, betrachten wir ein nützliches Beispiel, das `Maybe`-Monad:

```
1 data Maybe a = Nothing | Just a
2
3 instance Monad Maybe where
```

```
4   return          = Just
5   Nothing  >>= f = Nothing
6   (Just x) >>= f = f x
7   fail            = Nothing
8
9  instance MonadPlus Maybe where
10  mzero             = Nothing
11  Nothing `mplus` x = x
12  x `mplus` _       = x
```

B. Zusammenstellung von Algorithmen (Teil I)

Algorithmus 17: V-Baum einer Option auf den normalen Basiswert mit der Knock-Out-Barriere

Eingabe : S-Baum, C-Baum, p, q, r, g, rbt, Δt, N
Ausgabe : Alle Knoten des V-Baumes

```
1 // Initialisierung zum Zeitpunkt T_N
for j = 0 to N do
    if S(N,j) erfüllt das Knock-Out-Kriterium then
        V(N,j) ⟵ rbt
    else
        V(N,j) ⟵ C(N,j)
    end
end
2 // Diskontierung vom Zeitpunkt T_N zum Zeitpunkt T_0
for i = N − 1 to 0 do
    for j = 0 to i do
        if S(i,j) erfüllt das Knock-Out-Kriterium then
            V(i,j) ⟵ rbt
        else
            V(i,j) ⟵ e^{−(r−g)Δt}(p · V(i+1,j) + q · V(i+1,j+1))
            if C(i,j) ≠ ⊥ then
                V(i,j) ⟵ max(V(i,j), C(i,j))
            end
        end
    end
end
```

Algorithmus 18: V-Baum einer Option auf den normalen Basiswert mit der Knock-In-Barriere

Eingabe : S-Baum, C-Baum, p, q, r, g, rbt, Δt, N
Ausgabe : Alle Knoten des V-Baumes

```
1 // Initialisierung des Ṽ-Baumes und des V-Baumes zum Zeitpunkt T_N
for j = 0 to N do
    Ṽ(N, j) ⟵ C(N, j)
    if S(N, j) erfüllt das Knock-In-Kriterium then
        V(N, j) ⟵ Ṽ(N, j)
    else
        V(N, j) ⟵ rbt
    end
end
2 // Diskontierung des Ṽ-Baumes vom Zeitpunkt T_N zum Zeitpunkt T_0
for i = N − 1 to 0 do
    for j = 0 to i do
        Ṽ(i, j) ⟵ e^{−(r−g)Δt}(p · Ṽ(i + 1, j) + q · Ṽ(i + 1, j + 1))
        if C(i, j) ≠ ⊥ then
            Ṽ(i, j) ⟵ max(Ṽ(i, j), C(i, j))
        end
    end
end
3 // Diskontierung des V-Baumes vom Zeitpunkt T_N zum Zeitpunkt T_0
for i = N − 1 to 0 do
    for j = 0 to i do
        if S(i, j) erfüllt das Knock-In-Kriterium then
            V(i, j) ⟵ Ṽ(i, j)
        else
            V(i, j) ⟵ e^{−(r−g)Δt}(p · V(i + 1, j) + q · V(i + 1, j + 1))
        end
    end
end
```

Algorithmus 19: V^{max}-Baum einer Option auf den maximalen Basiswert mit der Knock-Out-Barriere

Eingabe : S-Baum, C^{max}-Baum, p, q, r, g, rbt, Δt, N
Ausgabe : Alle Knoten des V^{max}-Baumes

1 // Initialisierung zum Zeitpunkt T_N
for $j = 0$ **to** N **do**
  if $S(N,j)$ *erfüllt das Knock-Out-Kriterium* **then**
    for $k = 1$ **to** $\sharp\,|C^{max}(N,j)|$ **do**
      $V^{max}(N,j)[k] \longleftarrow rbt$
    end
  else
    $V^{max}(N,j) \longleftarrow C^{max}(N,j)$
  end
end

2 // Diskontierung vom Zeitpunkt T_N zum Zeitpunkt T_0
for $i = N-1$ **to** 0 **do**
  for $j = 0$ **to** i **do**
    $l \longleftarrow \sharp\,|C^{max}(i,j)|$
    if $S(i,j)$ *erfüllt das Knock-Out-Kriterium* **then**
      for $k = 1$ **to** l **do**
        $V^{max}(i,j)[k] \longleftarrow rbt$
      end
    else
      $V^{max}(i,j) \longleftarrow e^{-(r-g)\Delta t}\,(p \cdot (l \blacktriangleleft V^{max}(i+1,j)) + q \cdot (V^{max}(i+1,j+1) \blacktriangleright l))$
      if $C^{max}(i,j) \neq \perp$ **then**
        for $k = 1$ **to** l **do**
          $V^{max}(i,j)[k] \longleftarrow \max\,(V^{max}(i,j)[k], C^{max}(i,j)[k])$
        end
      end
    end
  end
end

Algorithmus 20: V^{max}-Baum einer Option auf den maximalen Basiswert mit der Knock-In-Barriere

Eingabe : S-Baum, C^{max}-Baum, p, q, r, g, rbt, Δt, N
Ausgabe : Alle Knoten des V^{max}-Baumes

1 // Initialisierung des $\tilde{V}^{max}$-Baumes und des V^{max}-Baumes zum Zeitpunkt T_N
for $j = 0$ **to** N **do**
  $\tilde{V}^{max}(N,j) \longleftarrow C^{max}(N,j)$
  if $S(N,j)$ *erfüllt das Knock-In-Kriterium* **then**
    $V^{max}(N,j) \longleftarrow \tilde{V}^{max}(N,j)$
  else
    for $k = 1$ **to** $\sharp|C^{max}(N,j)|$ **do**
      $V^{max}(N,j)[k] \longleftarrow rbt$
    end
  end
end

2 // Diskontierung des $\tilde{V}^{max}$-Baumes vom Zeitpunkt T_N zum Zeitpunkt T_0
for $i = N-1$ **to** 0 **do**
  for $j = 0$ **to** i **do**
    $l \longleftarrow \sharp|C^{max}(i,j)|$
    $\tilde{V}^{max}(i,j) \longleftarrow e^{-(r-g)\Delta t}\left(p \cdot (l \blacktriangleleft \tilde{V}^{max}(i+1,j)) + q \cdot (\tilde{V}^{max}(i+1,j+1) \blacktriangleright l)\right)$
    if $C^{max}(i,j) \neq \perp$ **then**
      for $k = 1$ **to** l **do**
        $\tilde{V}^{max}(i,j)[k] \longleftarrow \max\left(\tilde{V}^{max}(i,j)[k], C^{max}(i,j)[k]\right)$
      end
    end
  end
end

3 // Diskontierung des V-Baumes vom Zeitpunkt T_N zum Zeitpunkt T_0
for $i = N-1$ **to** 0 **do**
  for $j = 0$ **to** i **do**
    if $S(i,j)$ *erfüllt das Knock-In-Kriterium* **then**
      $V^{max}(i,j) \longleftarrow \tilde{V}^{max}(i,j)$
    else
      $V^{max}(i,j) \longleftarrow e^{-(r-g)\Delta t}\left(p \cdot (l \blacktriangleleft V^{max}(i+1,j)) + q \cdot (V^{max}(i+1,j+1) \blacktriangleright l)\right)$
    end
  end
end

Algorithmus 21: V^{min}-Baum einer Option auf den minimalen Basiswert mit der Knock-Out-Barriere

Eingabe : S-Baum, C^{min}-Baum, p, q, r, g, rbt, Δt, N
Ausgabe : Alle Knoten des V^{min}-Baumes

1 // Initialisierung zum Zeitpunkt T_N
for $j = 0$ **to** N **do**
  if $S(N,j)$ *erfüllt das Knock-Out-Kriterium* **then**
    for $k = 1$ **to** $\sharp\left|C^{min}(N,j)\right|$ **do**
      $V^{min}(N,j)[k] \longleftarrow rbt$
    end
  else
    $V^{min}(N,j) \longleftarrow C^{min}(N,j)$
  end
end

2 // Diskontierung vom Zeitpunkt T_N zum Zeitpunkt T_0
for $i = N-1$ **to** 0 **do**
  for $j = 0$ **to** i **do**
    $l \longleftarrow \sharp\left|C^{min}(i,j)\right|$
    if $S(i,j)$ *erfüllt das Knock-Out-Kriterium* **then**
      for $k = 1$ **to** l **do**
        $V^{min}(i,j)[k] \longleftarrow rbt$
      end
    else
      $V^{min}(i,j) \longleftarrow e^{-(r-g)\Delta t}\left(p \cdot (V^{min}(i+1,j) \blacktriangleright l) + q \cdot (l \blacktriangleleft V^{min}(i+1,j+1))\right)$
      if $C^{min}(i,j) \neq \perp$ **then**
        for $k = 1$ **to** l **do**
          $V^{min}(i,j)[k] \longleftarrow \max\left(V^{min}(i,j)[k], C^{min}(i,j)[k]\right)$
        end
      end
    end
  end
end

Algorithmus 22: V^{min}-Baum einer Option auf den minimalen Basiswert mit der Knock-In-Barriere

Eingabe : S-Baum, C^{min}-Baum, p, q, r, g, rbt, Δt, N
Ausgabe : Alle Knoten des V^{min}-Baumes

1 // Initialisierung des $\tilde{V}^{min}$-Baumes und des V^{min}-Baumes zum Zeitpunkt T_N
for $j = 0$ **to** N **do**
  $\tilde{V}^{min}(N,j) \longleftarrow C^{min}(N,j)$
  if $S(N,j)$ *erfüllt das Knock-In-Kriterium* **then**
    $V^{min}(N,j) \longleftarrow \tilde{V}^{min}(N,j)$
  else
    for $k = 1$ **to** $\sharp\left|C^{min}(N,j)\right|$ **do**
      $V^{min}(N,j)[k] \longleftarrow rbt$
    end
  end
end

2 // Diskontierung des $\tilde{V}^{min}$-Baumes vom Zeitpunkt T_N zum Zeitpunkt T_0
for $i = N-1$ **to** 0 **do**
  for $j = 0$ **to** i **do**
    $l \longleftarrow \sharp\left|C^{min}(i,j)\right|$
    $\tilde{V}^{min}(i,j) \longleftarrow e^{-(r-g)\Delta t}\left(p \cdot (\tilde{V}^{min}(i+1,j) \blacktriangleright l) + q \cdot (l \blacktriangleleft \tilde{V}^{min}(i+1,j+1))\right)$
    if $C^{min}(i,j) \neq \perp$ **then**
      for $k = 1$ **to** l **do**
        $\tilde{V}^{min}(i,j)[k] \longleftarrow \max\left(\tilde{V}^{min}(i,j)[k], C^{min}(i,j)[k]\right)$
      end
    end
  end
end

3 // Diskontierung des V^{min}-Baumes vom Zeitpunkt T_N zum Zeitpunkt T_0
for $i = N-1$ **to** 0 **do**
  for $j = 0$ **to** i **do**
    if $S(i,j)$ *erfüllt das Knock-In-Kriterium* **then**
      $V^{min}(i,j) \longleftarrow \tilde{V}^{min}(i,j)$
    else
      $V^{min}(i,j) \longleftarrow e^{-(r-g)\Delta t}\left(p \cdot (V^{min}(i+1,j) \blacktriangleright l) + q \cdot (l \blacktriangleleft V^{min}(i+1,j+1))\right)$
    end
  end
end

Algorithmus 23: V^{ave}-Baum einer Option auf den durchschnittlichen Basiswert mit der Knock-Out-Barriere

Eingabe : S-Baum, C^{ave}-Baum, p, q, r, g, rbt, Δt, N, HD
Ausgabe : Alle Knoten des V^{ave}-Baumes

1 // Initialisierung zum Zeitpunkt T_N
for $j = 0$ **to** N **do**
  if $S(N,j)$ *erfüllt das Out-Kriterium* **then**
    for $k = 1$ **to** $\sharp\,|C^{ave}(N,j)|$ **do**
      $V^{ave}(N,j)[k] \longleftarrow rbt$
    end
  else
    $V^{ave}(N,j) \longleftarrow C^{ave}(N,j)$
  end
end

2 // Diskontierung vom Zeitpunkt T_N zum Zeitpunkt T_0
for $i = N-1$ **to** 0 **do**
  for $j = 0$ **to** i **do**
    if $S(i,j)$ *erfüllt das Knock-Out-Kriterium* **then**
      for $k = 1$ **to** $\sharp\,|C^{ave}(i,j)|$ **do**
        $V^{ave}(i,j)[k] \longleftarrow rbt$
      end
    else
      $V^{ave}(i,j) \longleftarrow \mathbb{DISCA}\left(C^{ave}(i,j)\,\middle|\,\begin{matrix} V^{ave}(i+1,j) \\ V^{ave}(i+1,j+1) \end{matrix}\right)$
      if $C^{ave}(i,j) \neq \perp$ **then**
        for $k = 1$ **to** $\sharp\,|C^{ave}(i,j)|$ **do**
          $V^{ave}(i,j)[k] \longleftarrow \max\left(V^{ave}(i,j)[k], C^{ave}(i,j)[k]\right)$
        end
      end
    end
  end
end

Algorithmus 24: V^{ave}-Baum einer Option auf den durchschnittlichen Basiswert mit der Knock-In-Barriere

Eingabe : S-Baum, C^{ave}-Baum, p, q, r, g, rbt, Δt, N, HD
Ausgabe : Alle Knoten des V^{ave}-Baumes

1 // Initialisierung des $\tilde{V}^{ave}$-Baumes und des V^{ave}-Baumes zum Zeitpunkt T_N
for $j = 0$ **to** N **do**
  $\tilde{V}^{ave}(N,j) \longleftarrow C^{ave}(N,j)$
  if $S(N,j)$ *erfüllt das Knock-In-Kriterium* **then**
    $V^{ave}(N,j) \longleftarrow \tilde{V}^{ave}(N,j)$
  else
    for $k = 1$ **to** $\sharp|C^{ave}(N,j)|$ **do**
      $V^{ave}(N,j)[k] \longleftarrow rbt$
    end
  end
end

2 // Diskontierung des $\tilde{V}^{ave}$-Baumes vom Zeitpunkt T_N zum Zeitpunkt T_0
for $i = N-1$ **to** 0 **do**
  for $j = 0$ **to** i **do**
    $$\tilde{V}^{ave}(i,j) \longleftarrow \mathbb{DISCA}\left(C^{ave}(i,j)\,\middle|\,\begin{matrix}\tilde{V}^{ave}(i+1,j)\\ \tilde{V}^{ave}(i+1,j+1)\end{matrix}\right)$$
    if $C^{ave}(i,j) \neq \perp$ **then**
      for $k = 1$ **to** $\sharp|C^{ave}(i,j)|$ **do**
        $\tilde{V}^{ave}(i,j)[k] \longleftarrow \max\left(\tilde{V}^{ave}(i,j)[k], C^{ave}(i,j)[k]\right)$
      end
    end
  end
end

3 // Diskontierung des V^{ave}-Baumes vom Zeitpunkt T_N zum Zeitpunkt T_0
for $i = N-1$ **to** 0 **do**
  for $j = 0$ **to** i **do**
    if $S(i,j)$ *erfüllt das Knock-In-Kriterium* **then**
      $V^{ave}(i,j) \longleftarrow \tilde{V}^{ave}(i,j)$
    else
      $$V^{ave}(i,j) \longleftarrow \mathbb{DISCA}\left(C^{ave}(i,j)\,\middle|\,\begin{matrix}V^{ave}(i+1,j)\\ V^{ave}(i+1,j+1)\end{matrix}\right)$$
    end
  end
end

Algorithmus 25: V-Baum einer Multi-Asset-Option ohne Barriere

Eingabe : Y-Baum, M, N, f, h, T_A
Ausgabe : Alle Knoten des V-Baumes

1 // Initialisierung zum Zeitpunkt T_N
for $l = 0$ **to** $(N+1)^M - 1$ **do**
  $S^{(N)}[l] \longleftarrow h\left(Y^{(N)}[l]\right)$
  $V^{(N)}[l] \longleftarrow f\left(S^{(N)}[l]\right)$
end

2 // Diskontierung vom Zeitpunkt T_N zum Zeitpunkt T_0
for $k = N-1$ **to** 0 **do**
  $V^{(k)} \longleftarrow \mathsf{disc}(V^{(k+1)})$
  if $t_k \in T_A$ **then**
    for $l = 0$ **to** $(k+1)^M - 1$ **do**
      $S^{(k)}[l] \longleftarrow h\left(Y^{(k)}[l]\right)$
      $V^{(k)}[l] \longleftarrow \max\left(V^{(k)}[l], f\left(S^{(k)}[l]\right)\right)$
    end
  end
end

Algorithmus 26: V-Baum einer Multi-Asset-Option mit Knock-Out-Barrieren

Eingabe : Y-Baum, M, N, f, h, T_A, B_{out}
Ausgabe : Alle Knoten des V-Baumes

```
1 // Initialisierung zum Zeitpunkt T_N
for l = 0 to (N+1)^M − 1 do
    S^(N)[l] ⟵ h(Y^(N)[l])
    if S^(N)[l] erfüllt eines der Knock-Out-Kriterien then
        V^(N)[l] ⟵ 0
    else
        V^(N)[l] ⟵ f(S^(N)[l])
    end
end
2 // Diskontierung vom Zeitpunkt T_N zum Zeitpunkt T_0
for k = N − 1 to 0 do
    V^(k) ⟵ disc(V^(k+1))
    for l = 0 to (k+1)^M − 1 do
        S^(k)[l] ⟵ h(Y^(k)[l])
        if S^(k)[l] erfüllt eines der Knock-Out-Kriterien then
            V^(k)[l] ⟵ 0
        else
            if t_k ∈ T_A then
                V^(k)[l] ⟵ max(V^(k)[l], f(S^(k)[l]))
            end
        end
    end
end
```

Algorithmus 27: V-Baum einer Multi-Asset-Option mit mindestens einer Knock-In-Barrieren

Eingabe : Y-Baum, M, N, f, h, T_A, B_{out}, B_{in}
Ausgabe : Alle Knoten des V-Baumes

1 // Initialisierung zum Zeitpunkt T_N
for $l = 0$ **to** $(N+1)^M - 1$ **do**
  $S^{(N)}[l] \longleftarrow h\left(Y^{(N)}[l]\right)$
  if $S^{(N)}[l]$ *erfüllt eines der Knock-Out-Kriterien* **then**
    $\tilde{V}^{(N)}[l] \longleftarrow 0$, $V^{(N)}[l] \longleftarrow 0$
  else
    $\tilde{V}^{(N)}[l] \longleftarrow f\left(S^{(N)}[l]\right)$
    if $S^{(N)}[l]$ *erfüllt eines der Knock-In-Kriterien* **then**
      $V^{(N)}[l] \longleftarrow \tilde{V}^{(N)}[l]$
    else
      $V^{(N)}[l] \longleftarrow 0$
    end
  end
end

2 // Diskontierung vom Zeitpunkt T_N zum Zeitpunkt T_0
for $k = N-1$ **to** 0 **do**
  $\tilde{V}^{(k)} \longleftarrow \mathsf{disc}(\tilde{V}^{(k+1)})$, $V^{(k)} \longleftarrow \mathsf{disc}(V^{(k+1)})$
  for $l = 0$ **to** $(k+1)^M - 1$ **do**
    $S^{(k)}[l] \longleftarrow h\left(Y^{(k)}[l]\right)$
    if $S^{(k)}[l]$ *erfüllt eines der Knock-Out-Kriterien* **then**
      $\tilde{V}^{(k)}[l] \longleftarrow 0$, $V^{(k)}[l] \longleftarrow 0$
    else
      if $t_k \in T_A$ **then**
        $\tilde{V}^{(k)}[l] \longleftarrow \max\left(\tilde{V}^{(k)}[l], f\left(S^{(k)}[l]\right)\right)$
      end
      if $S^{(k)}[l]$ *erfüllt eines der Knock-In-Kriterien* **then**
        $V^{(k)}[l] \longleftarrow \tilde{V}^{(k)}[l]$
      end
    end
  end
end

C. Bewertungsbeispiel des eindimensionalen ComDeCo (Teil I)

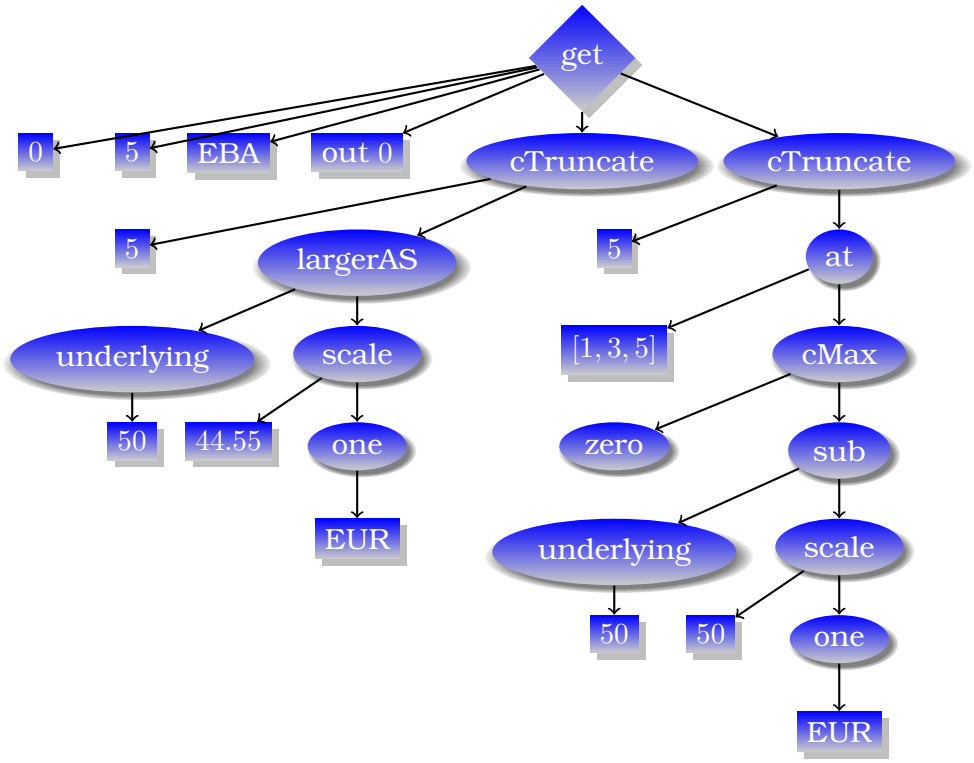


Abbildung C.1.: Zerlegte Baumdarstellung von `beispielOption`

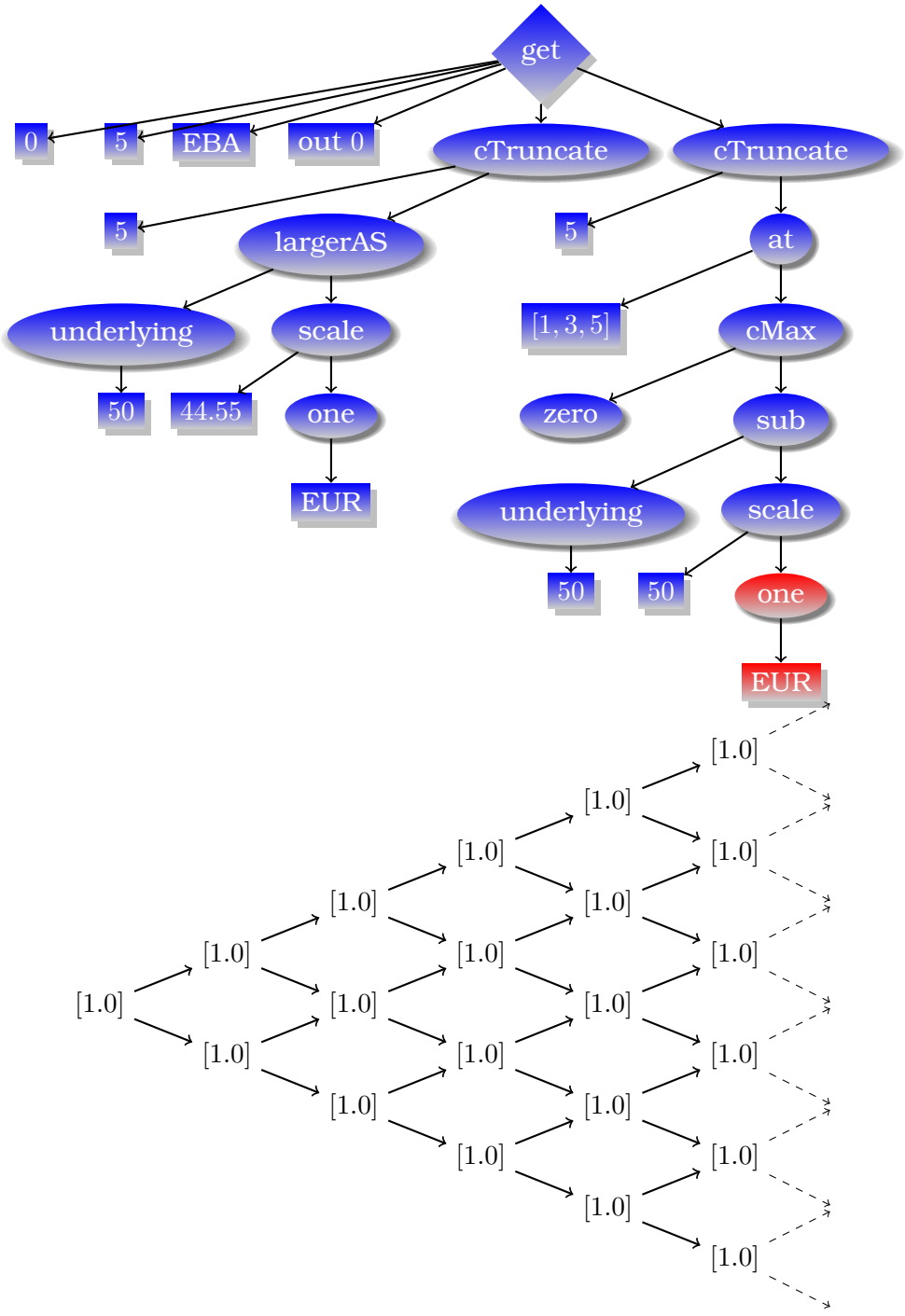


Abbildung C.2.: Bewertungsvorgang von `eval beispielOption` (1)

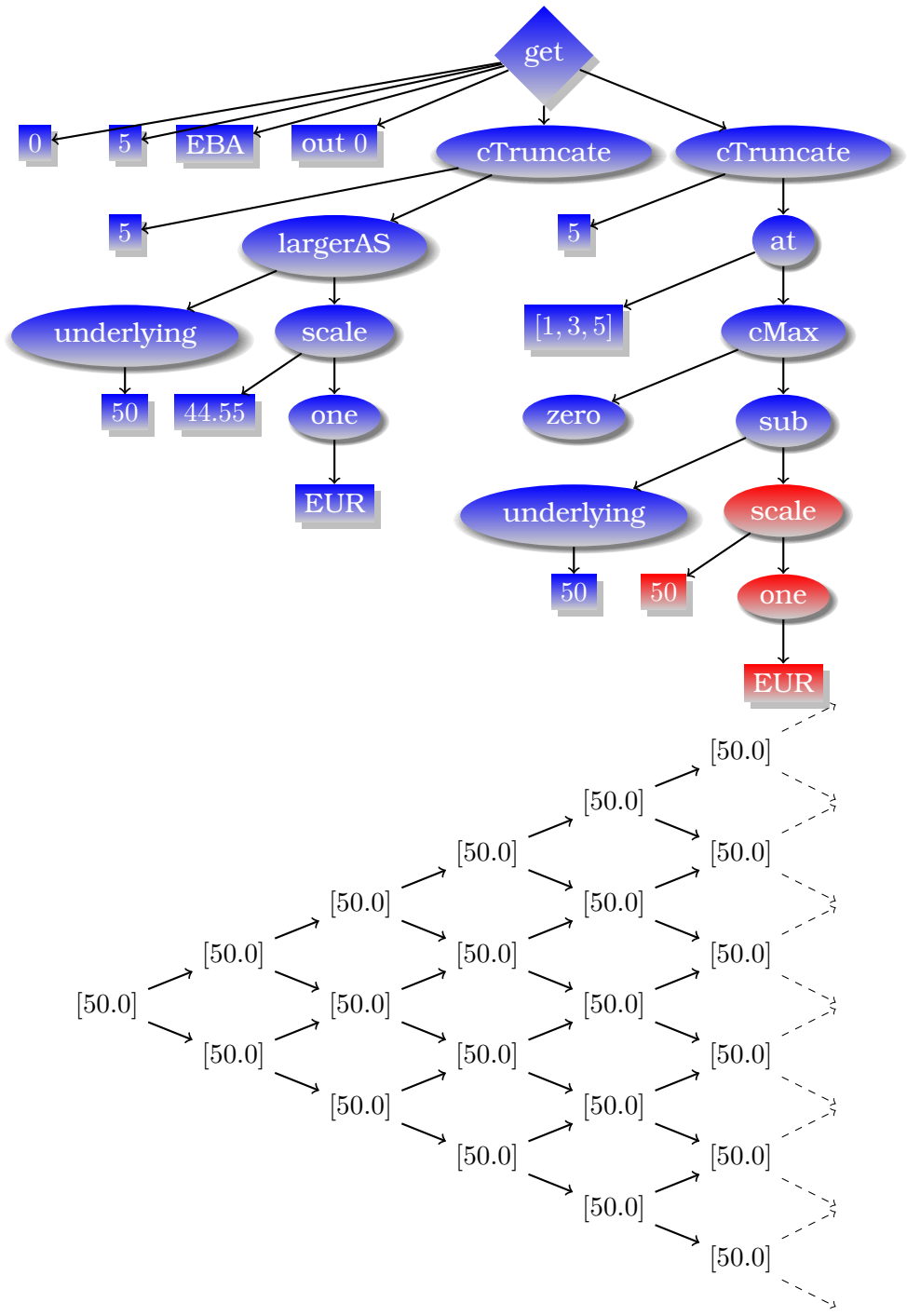


Abbildung C.3.: Bewertungsvorgang von `eval beispielOption` (2)

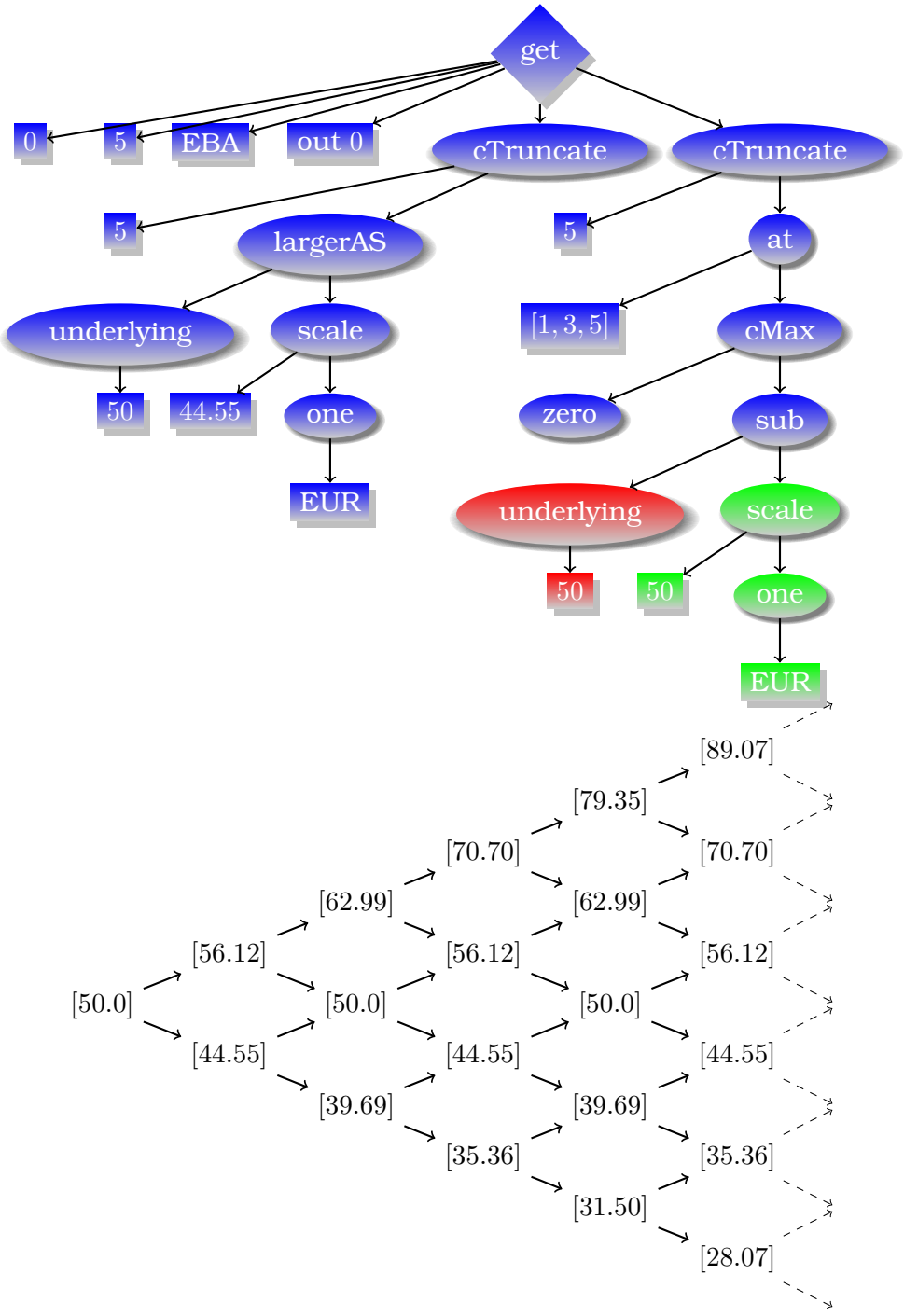


Abbildung C.4.: Bewertungsvorgang von `eval beispielOption` (3)

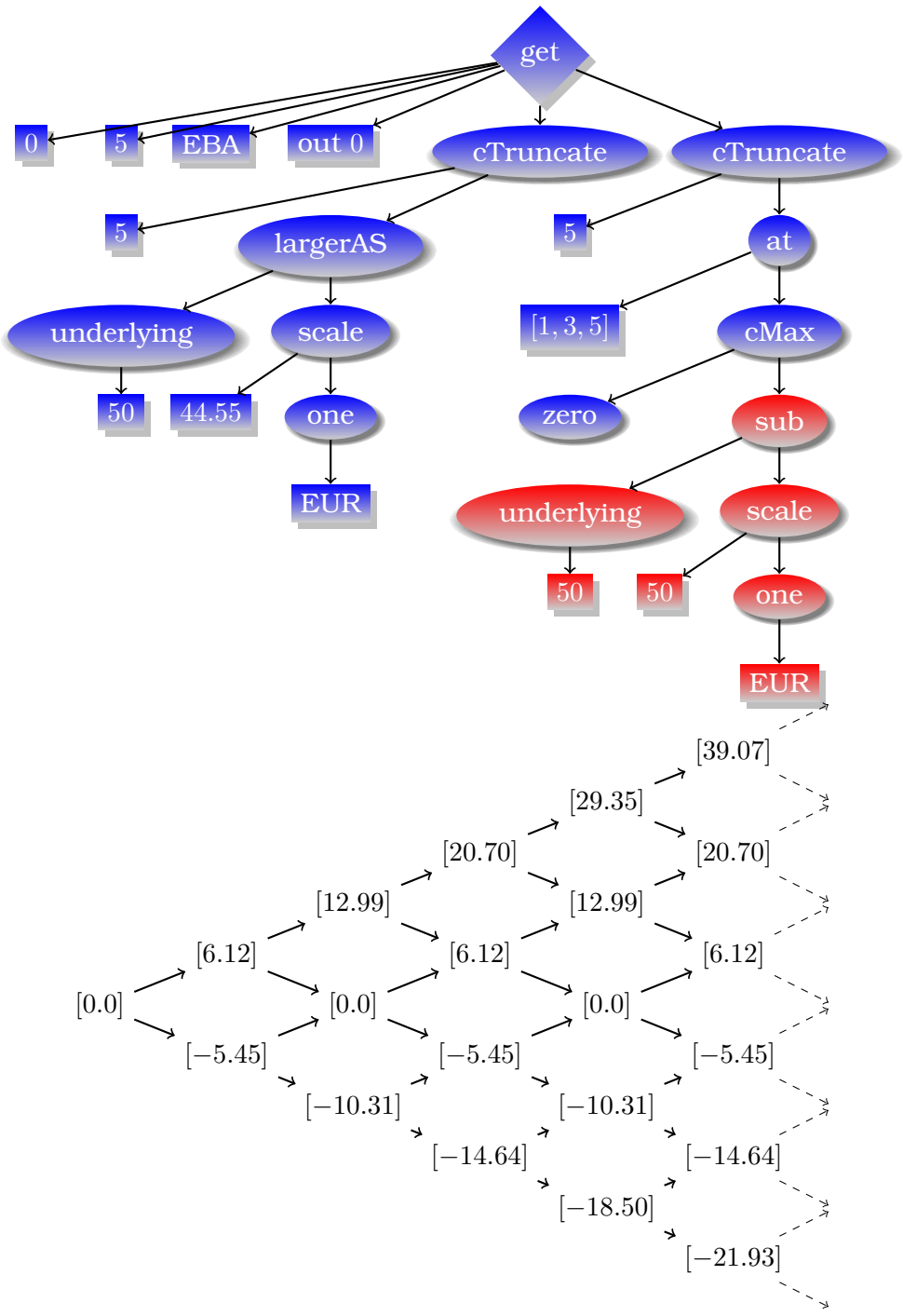


Abbildung C.5.: Bewertungsvorgang von `eval beispielOption` (4)

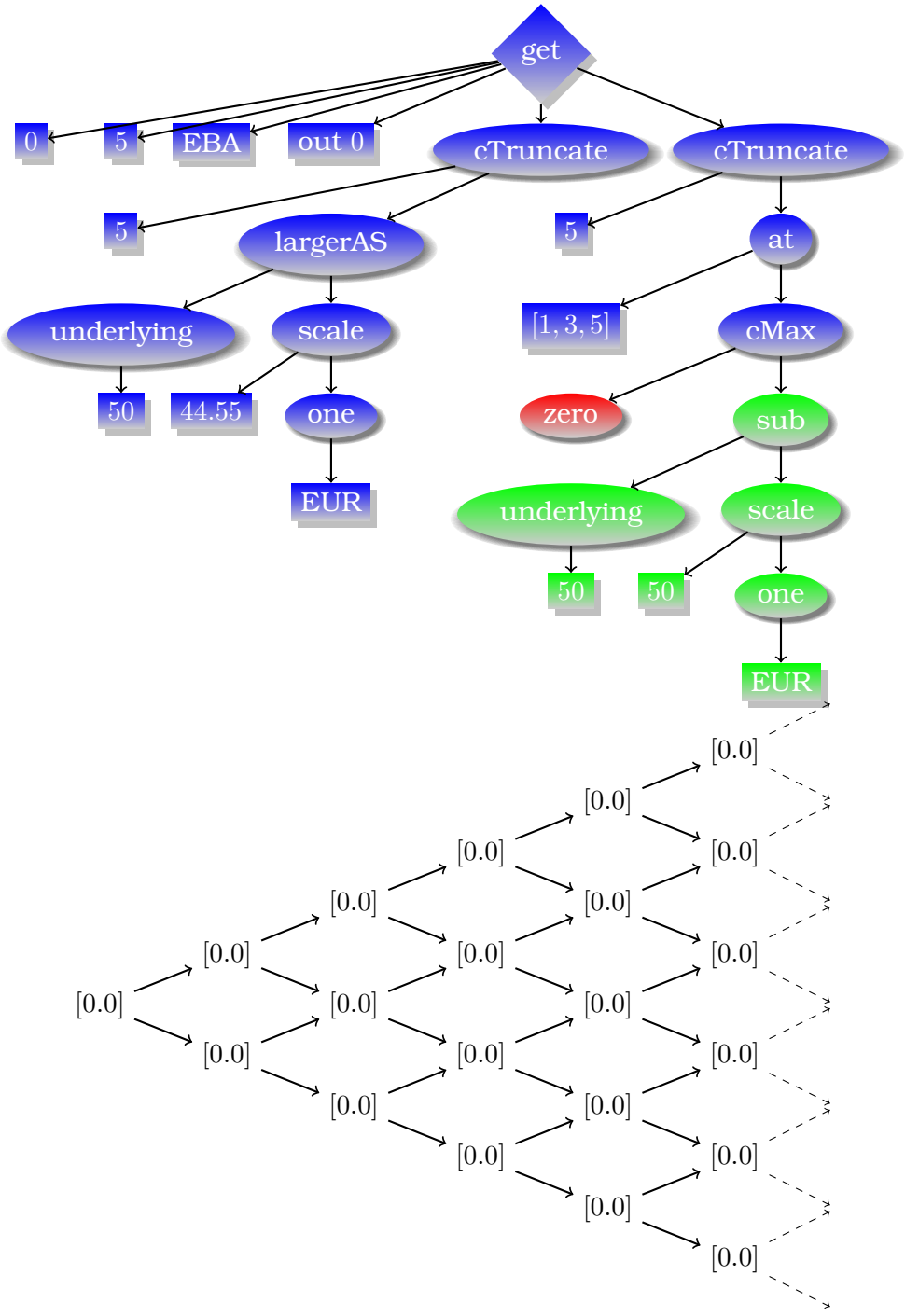


Abbildung C.6.: Bewertungsvorgang von `eval beispielOption` (5)

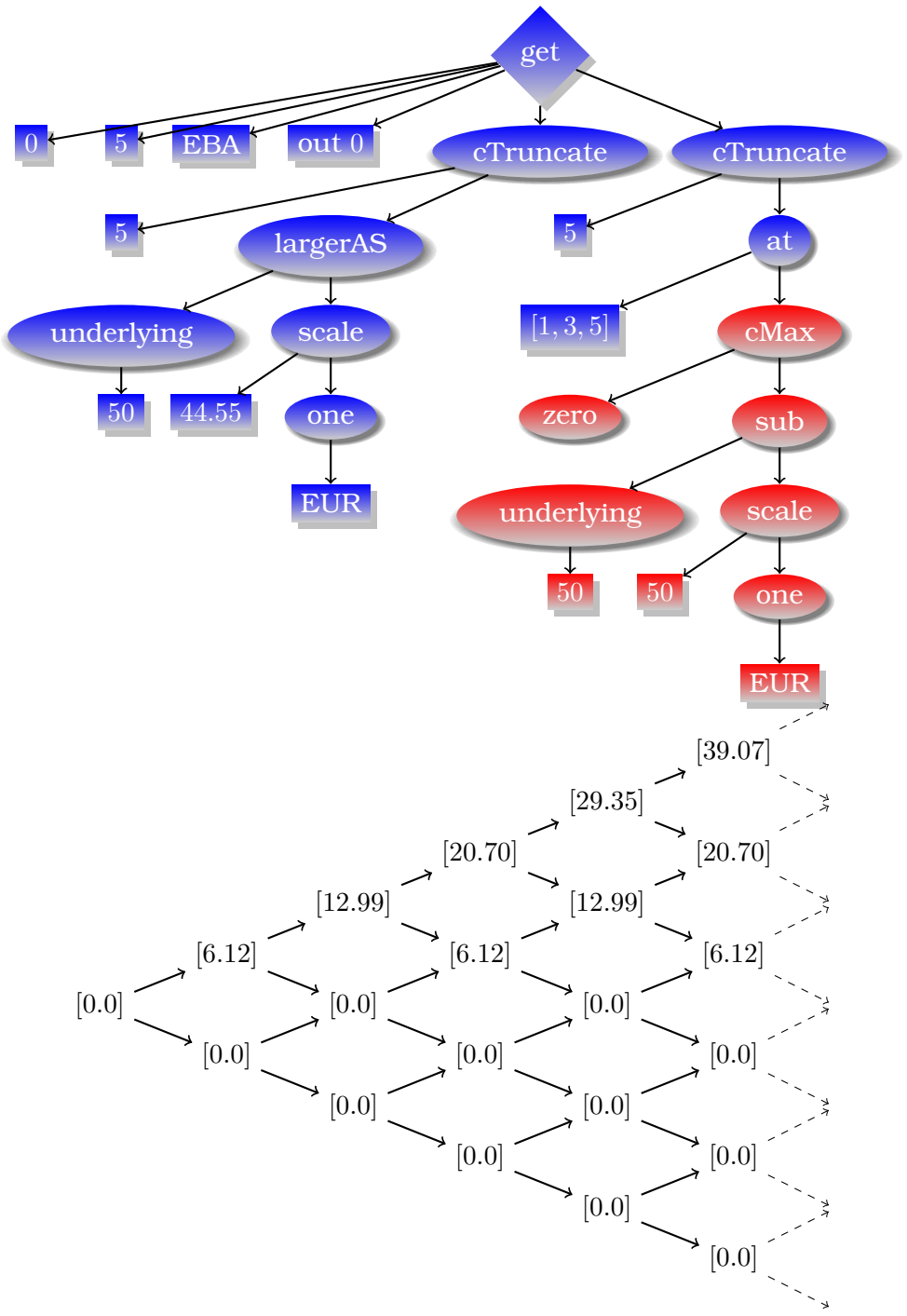


Abbildung C.7.: Bewertungsvorgang von `eval beispielOption` (6)

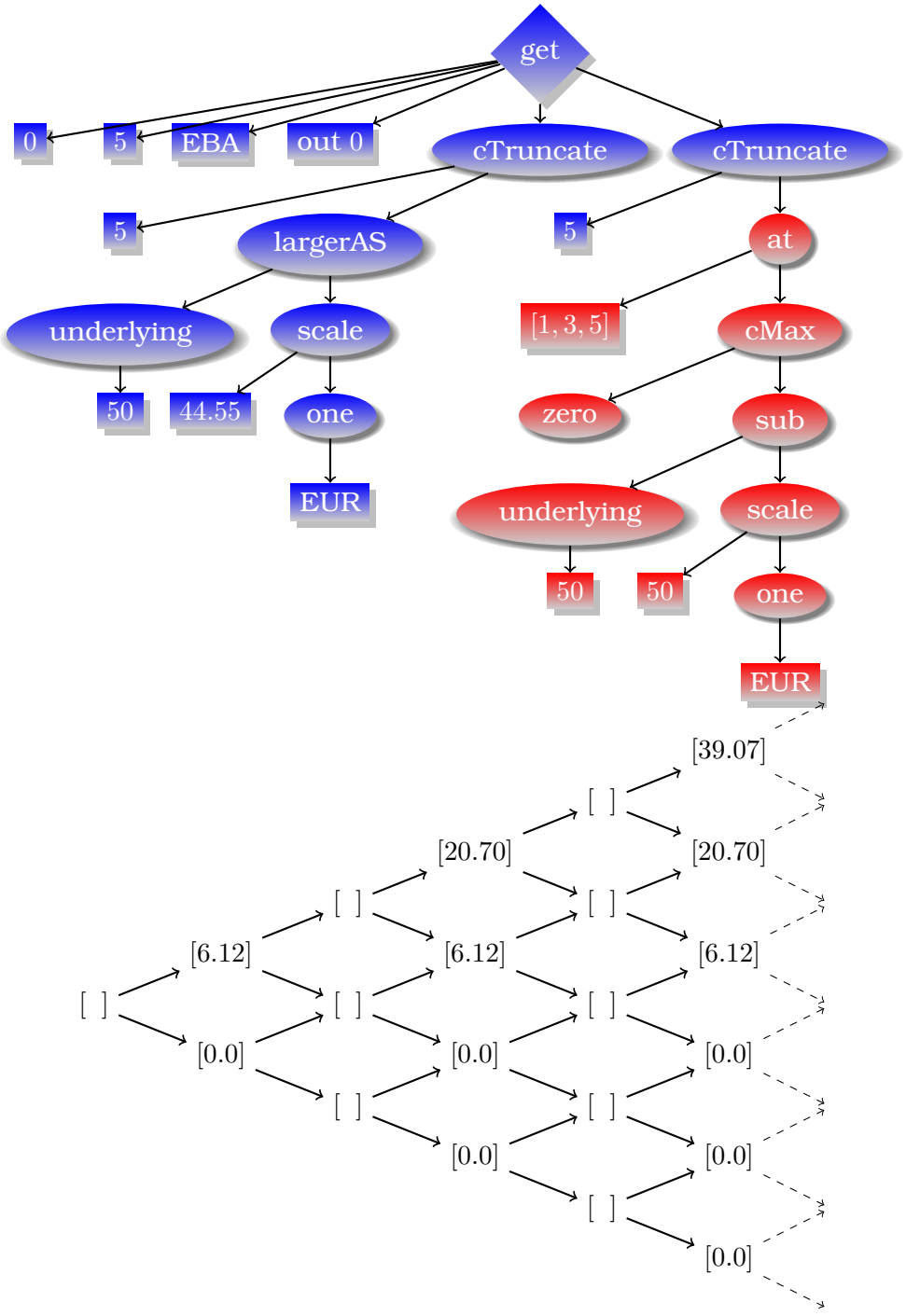


Abbildung C.8.: Bewertungsvorgang von `eval beispielOption` (7)

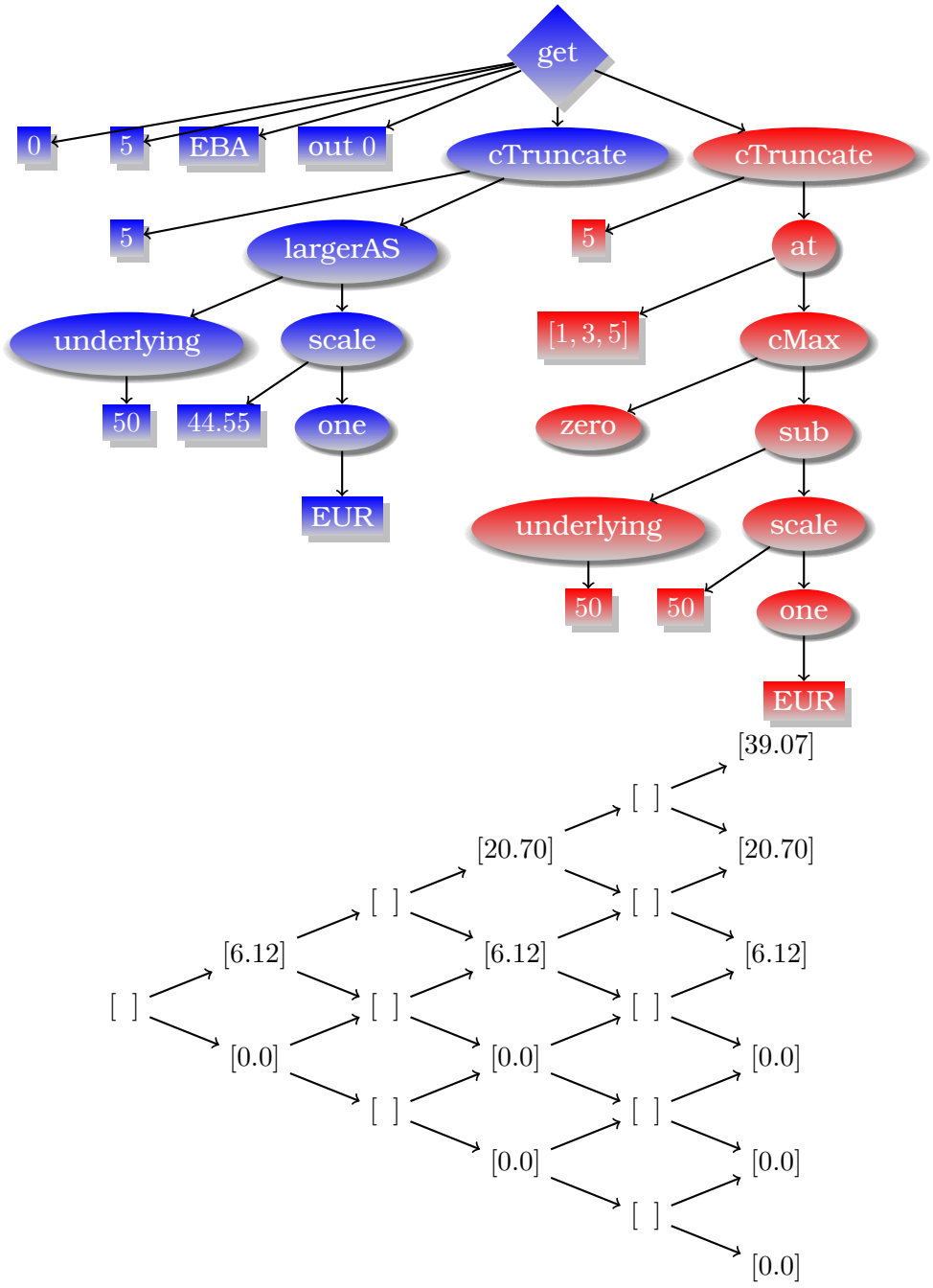


Abbildung C.9.: Bewertungsvorgang von `eval beispielOption` (8)

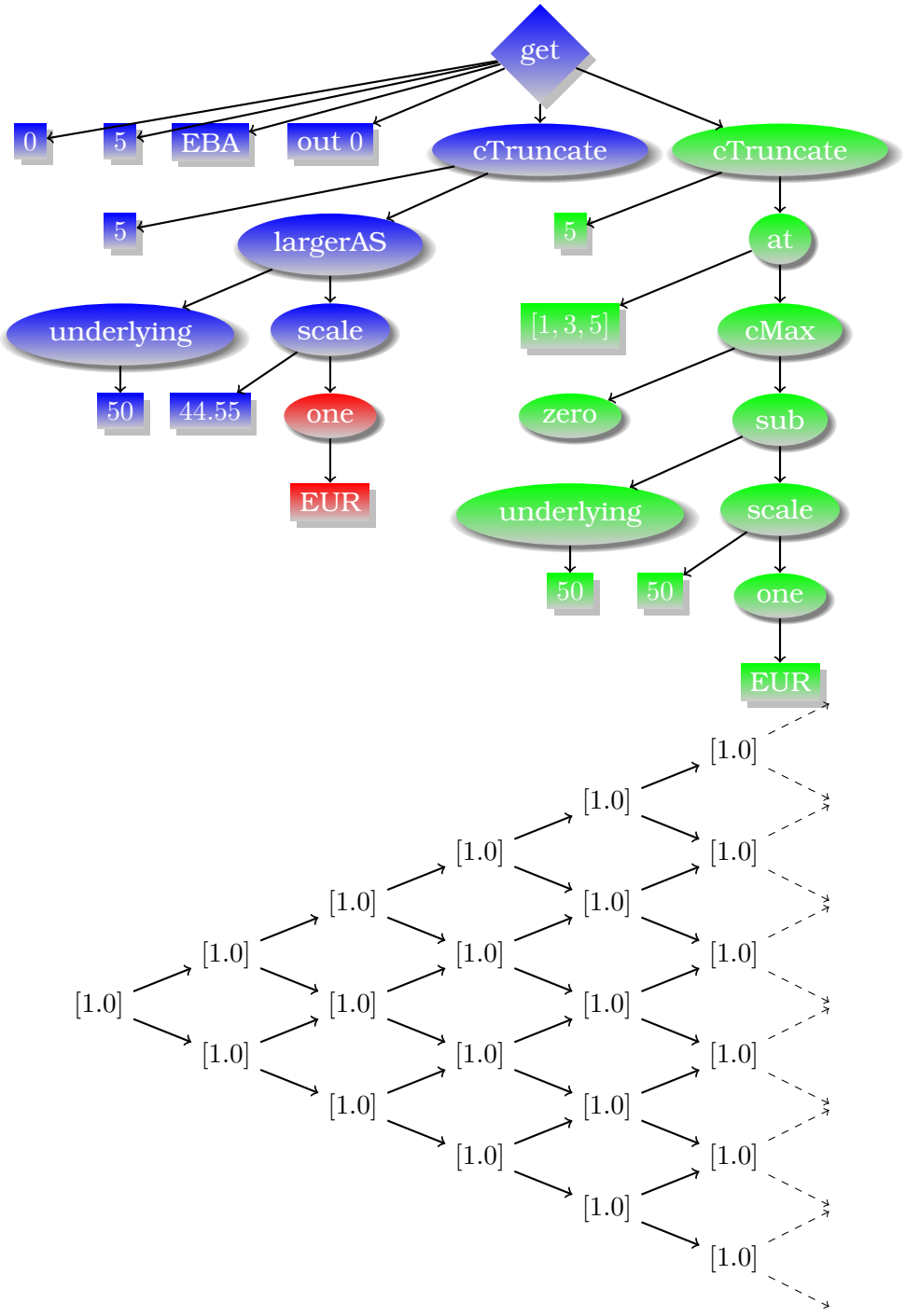


Abbildung C.10.: Bewertungsvorgang von `eval beispielOption` (9)

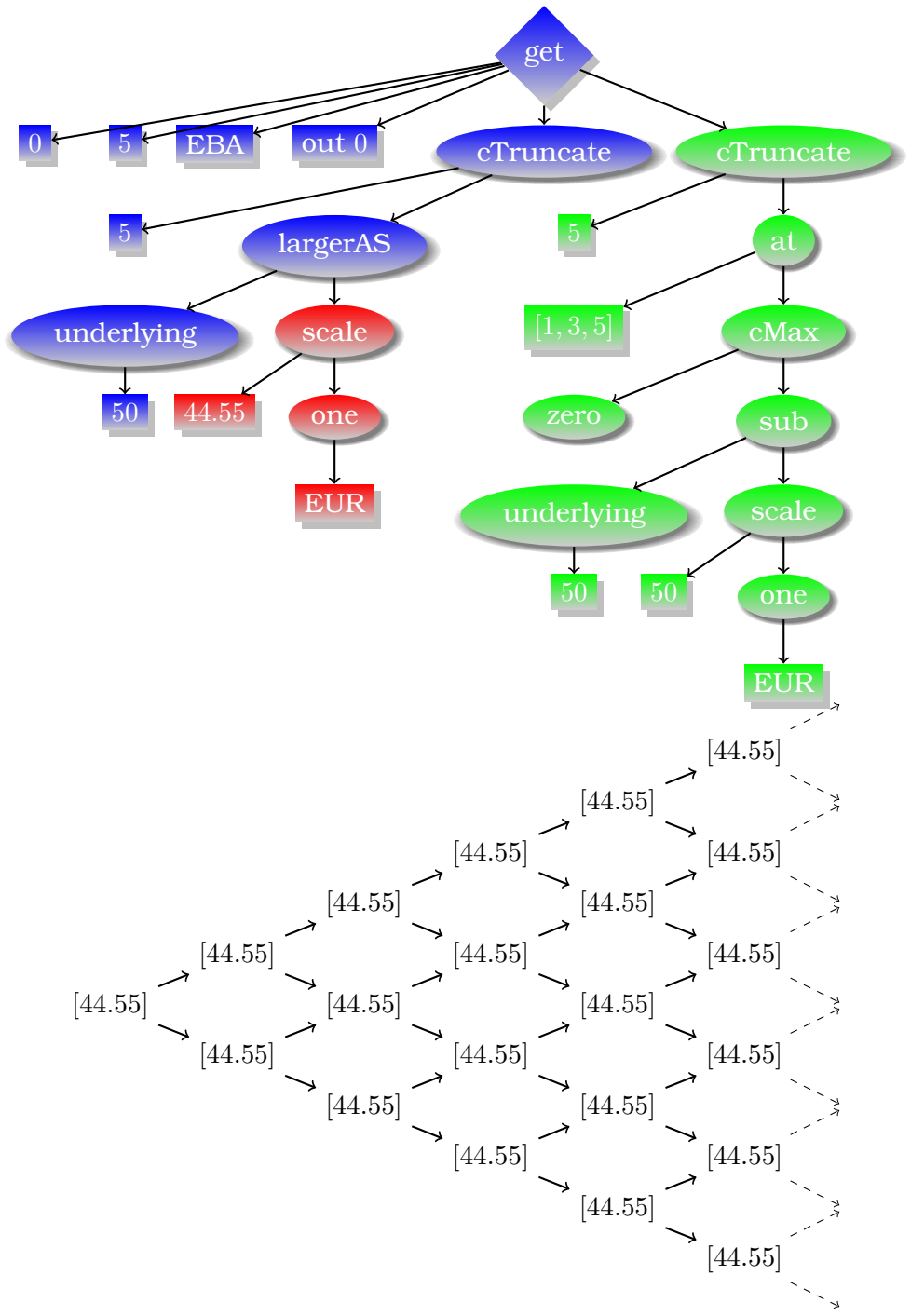


Abbildung C.11.: Bewertungsvorgang von `eval beispielOption` (10)

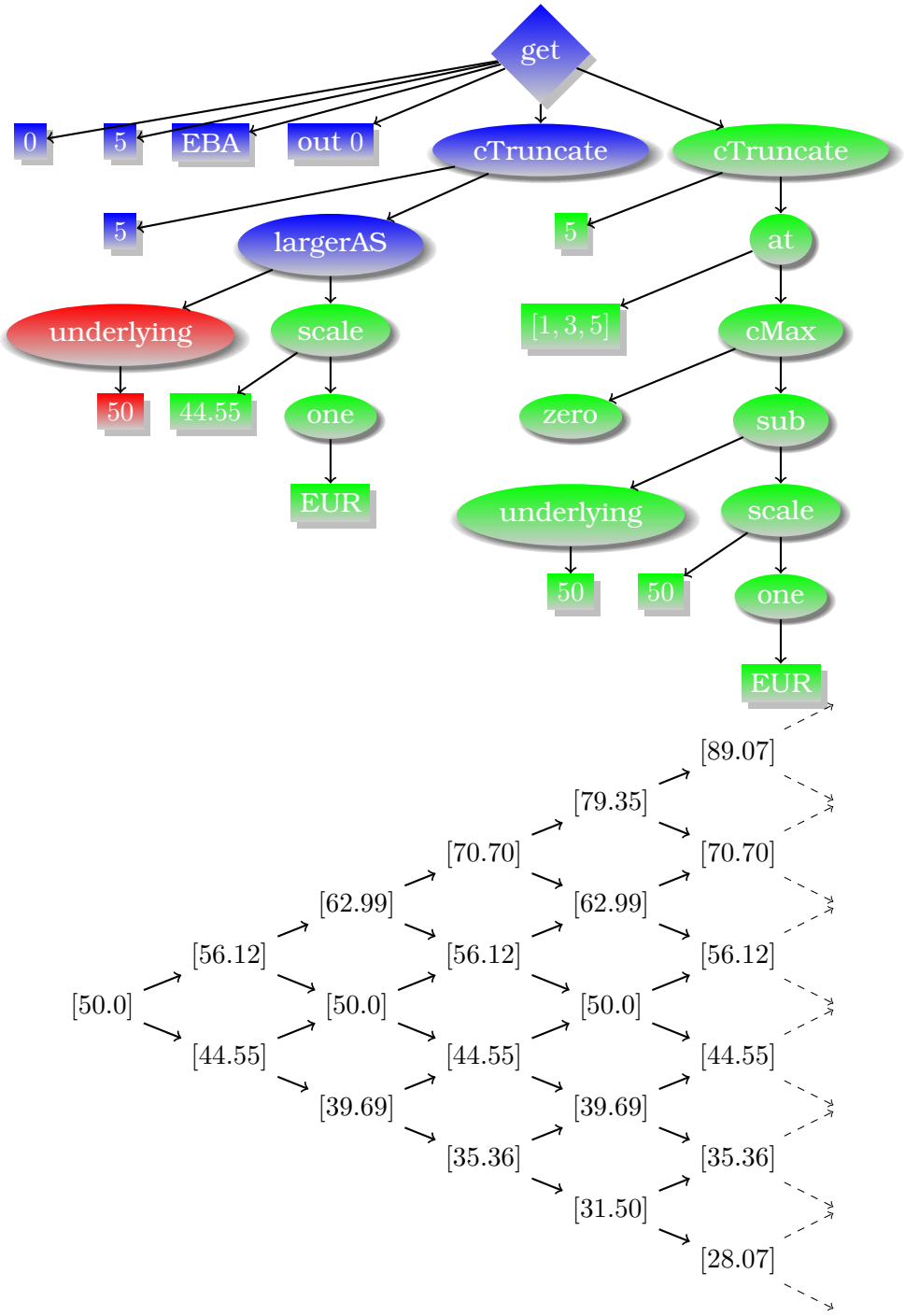


Abbildung C.12.: Bewertungsvorgang von `eval beispielOption` (11)

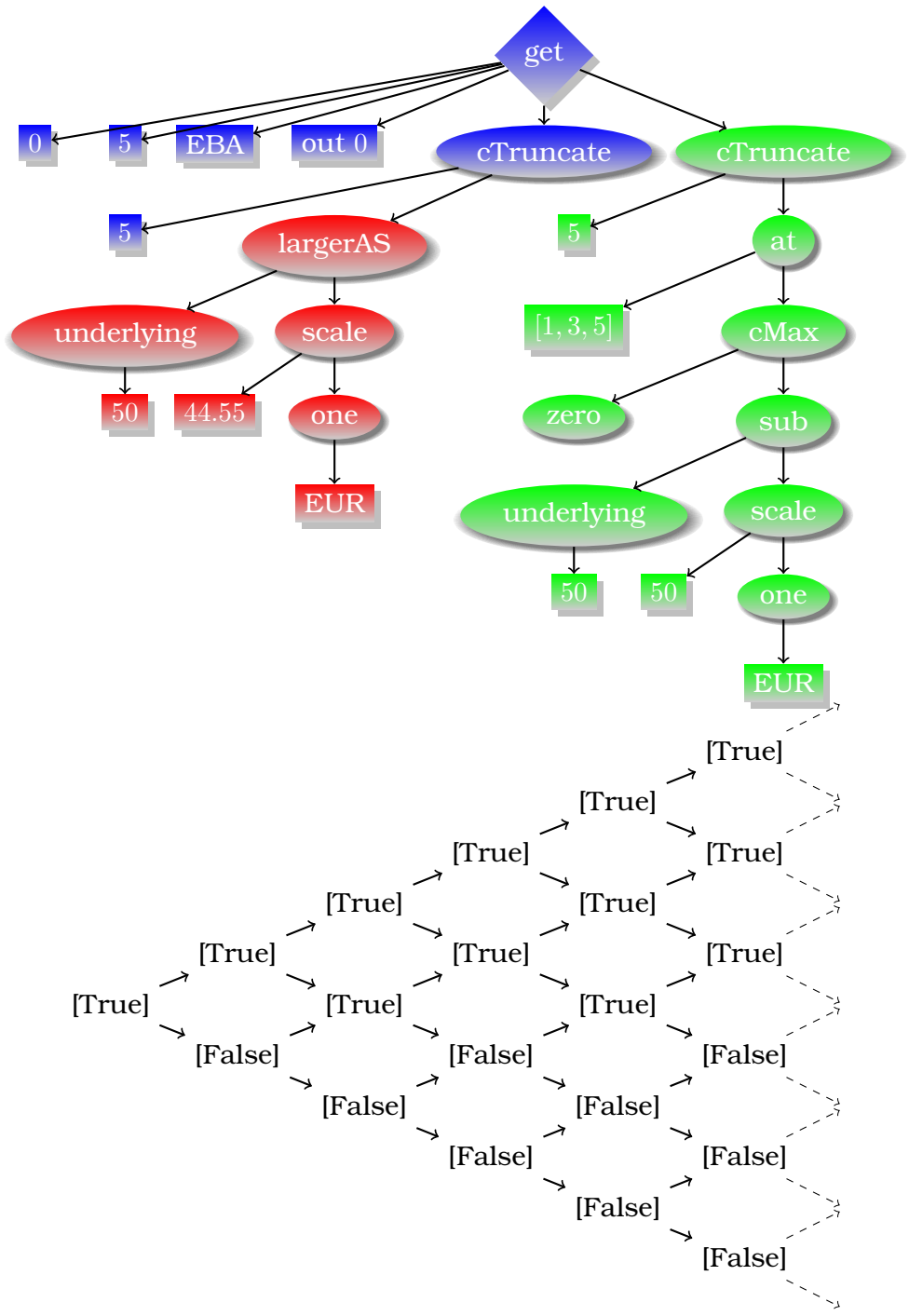


Abbildung C.13.: Bewertungsvorgang von `eval beispielOption` (12)

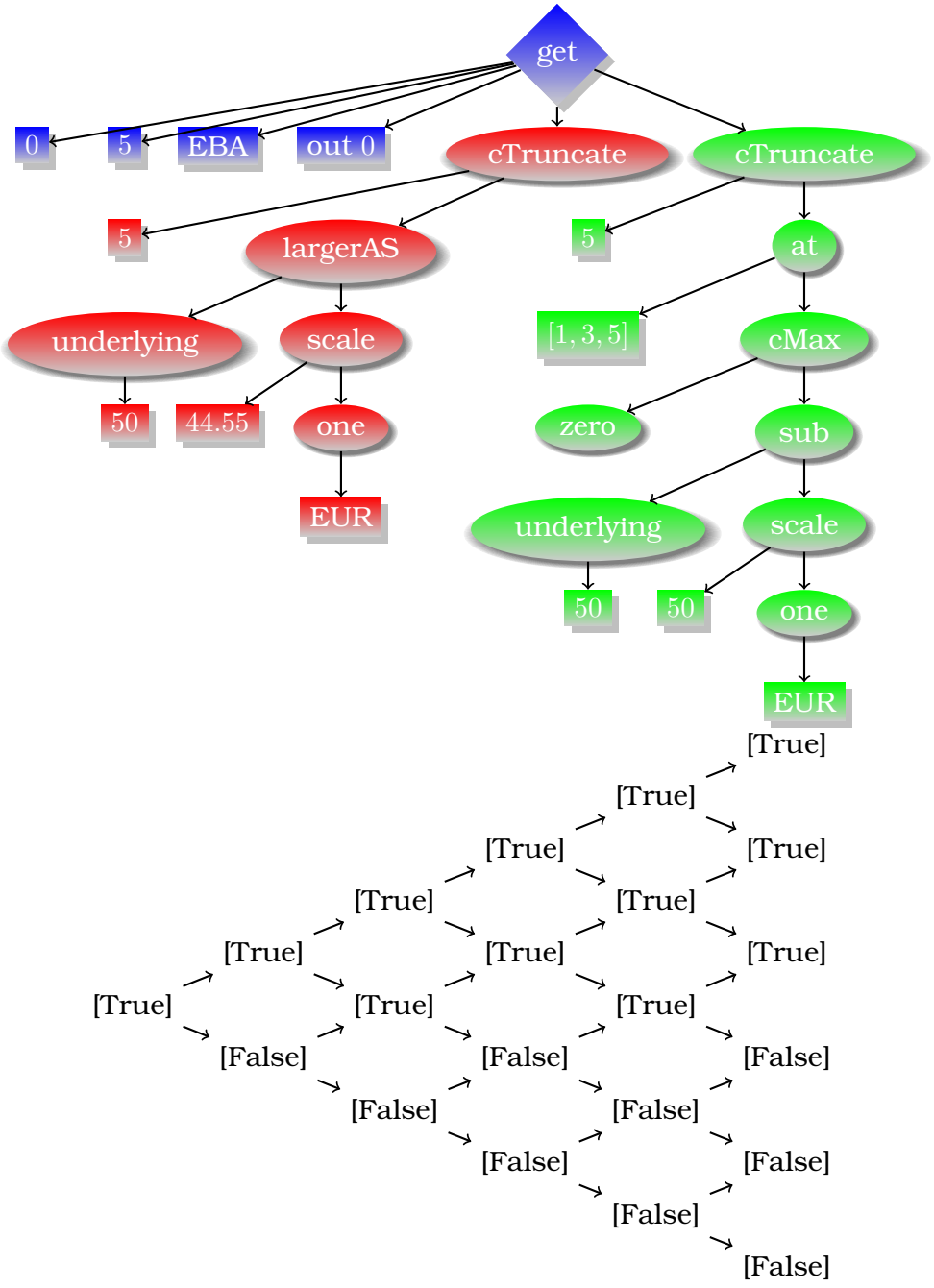


Abbildung C.14.: Bewertungsvorgang von `eval beispielOption` (13)

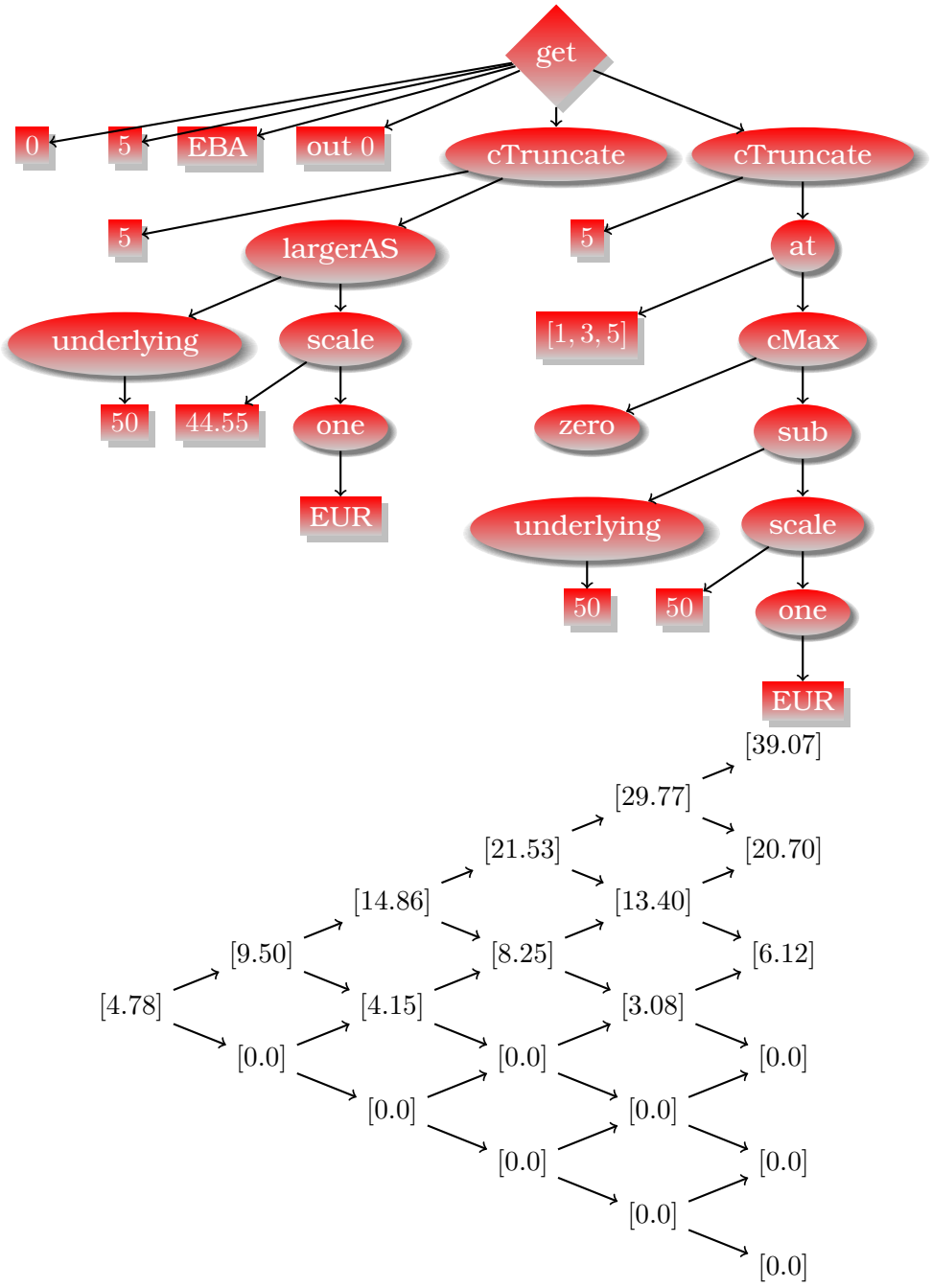


Abbildung C.15.: Bewertungsvorgang von `eval beispielOption` (14)

D. Bewertungsbeispiel des multidimensionalen ComDeCo (Teil I)

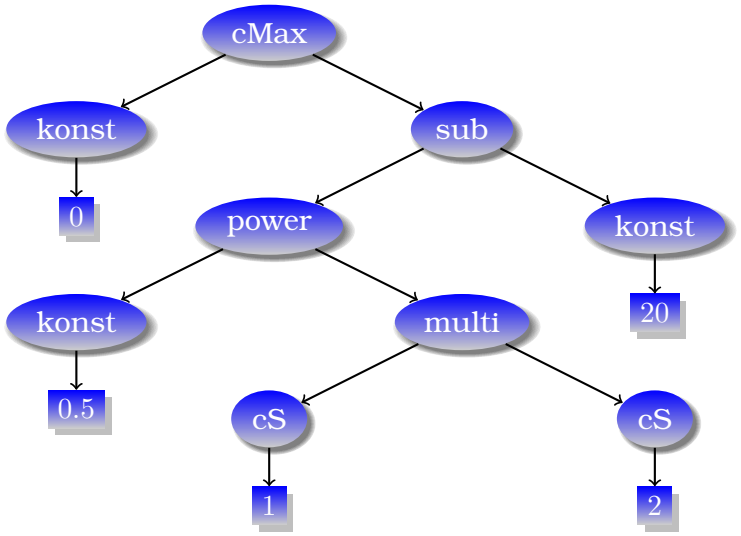


Abbildung D.1.: Zerlegte Baumdarstellung der Auszahlung von `beispielOption`

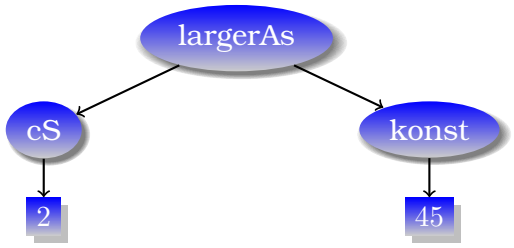


Abbildung D.2.: Zerlegte Baumdarstellung des Barriere-Kriteriums von `beispielOption`

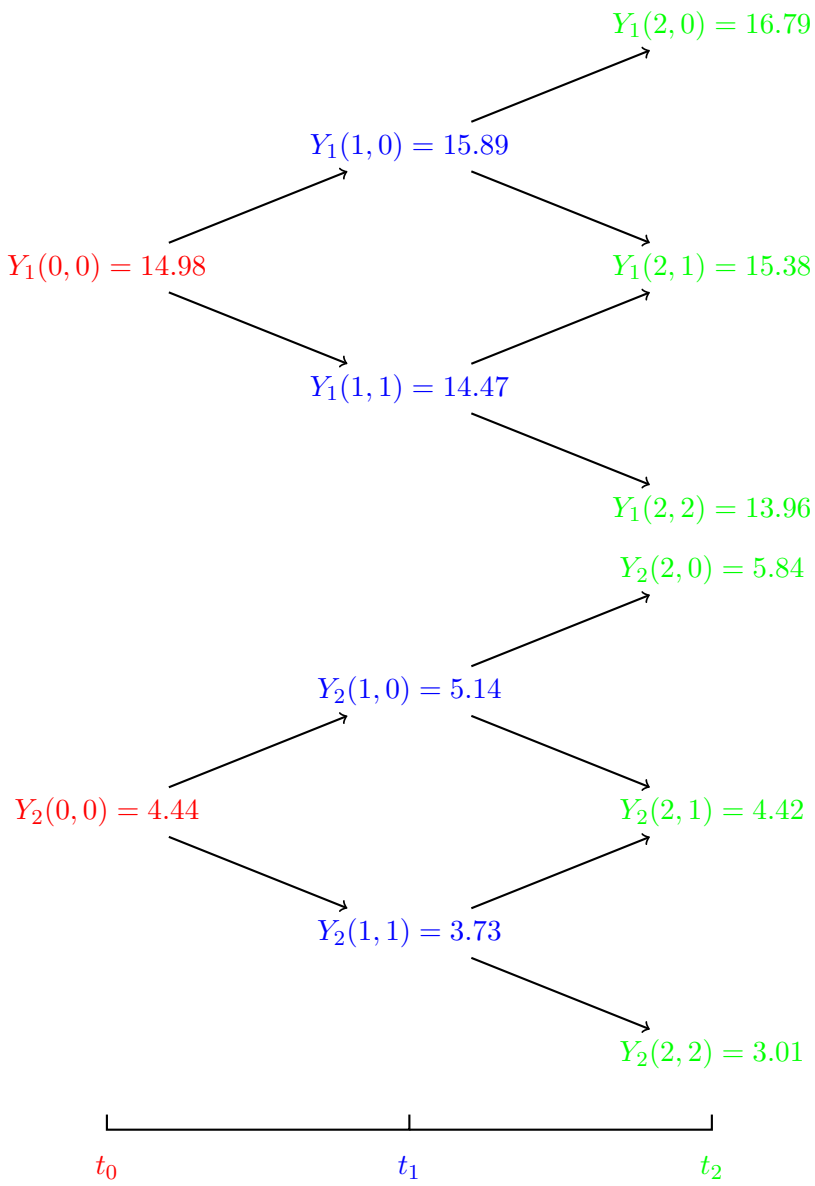


Abbildung D.3.: Y_1-Baum und Y_2-Baum von `evaluation beispielOption`

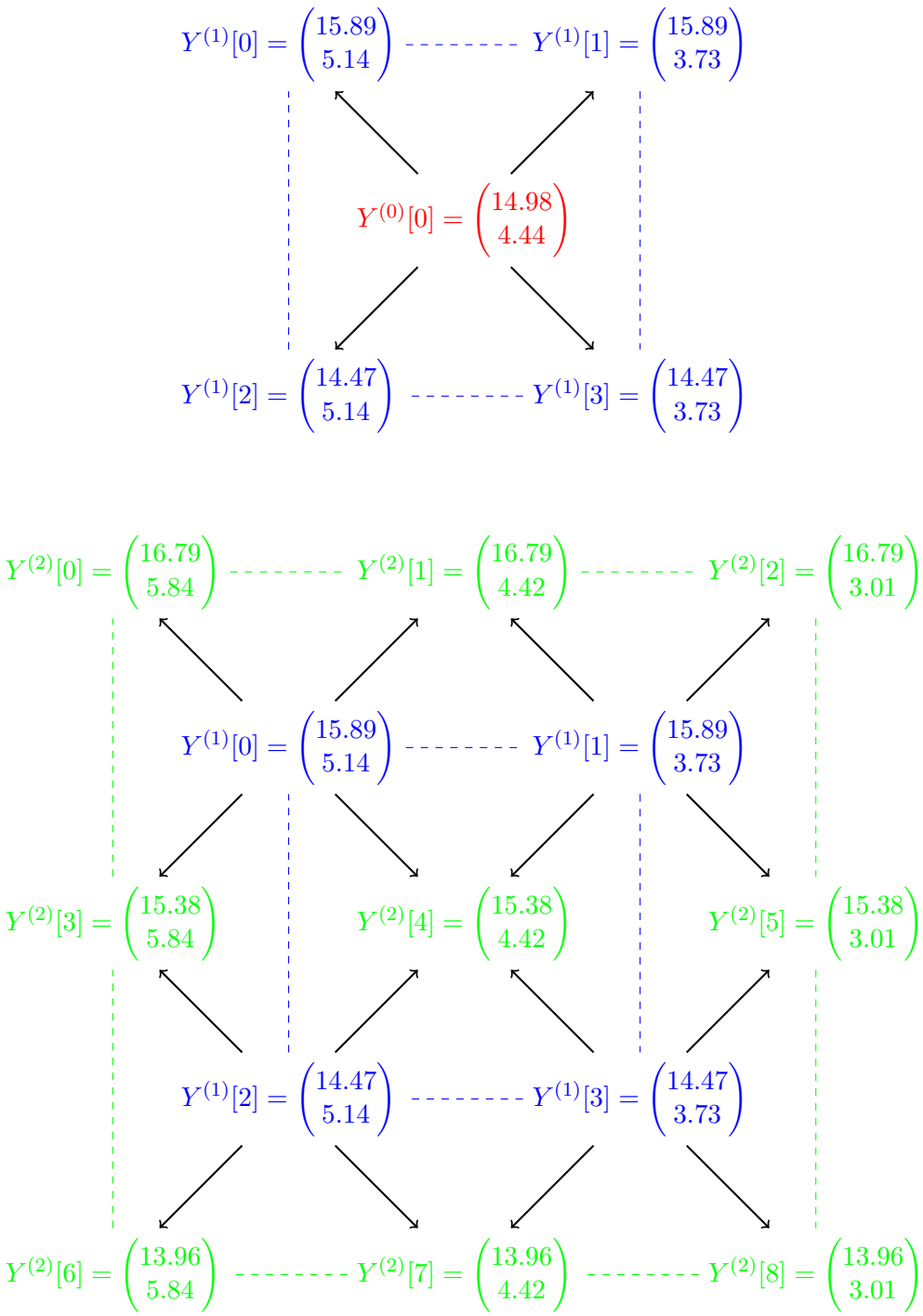


Abbildung D.4.: Y-Baum von `evaluation beispielOption`

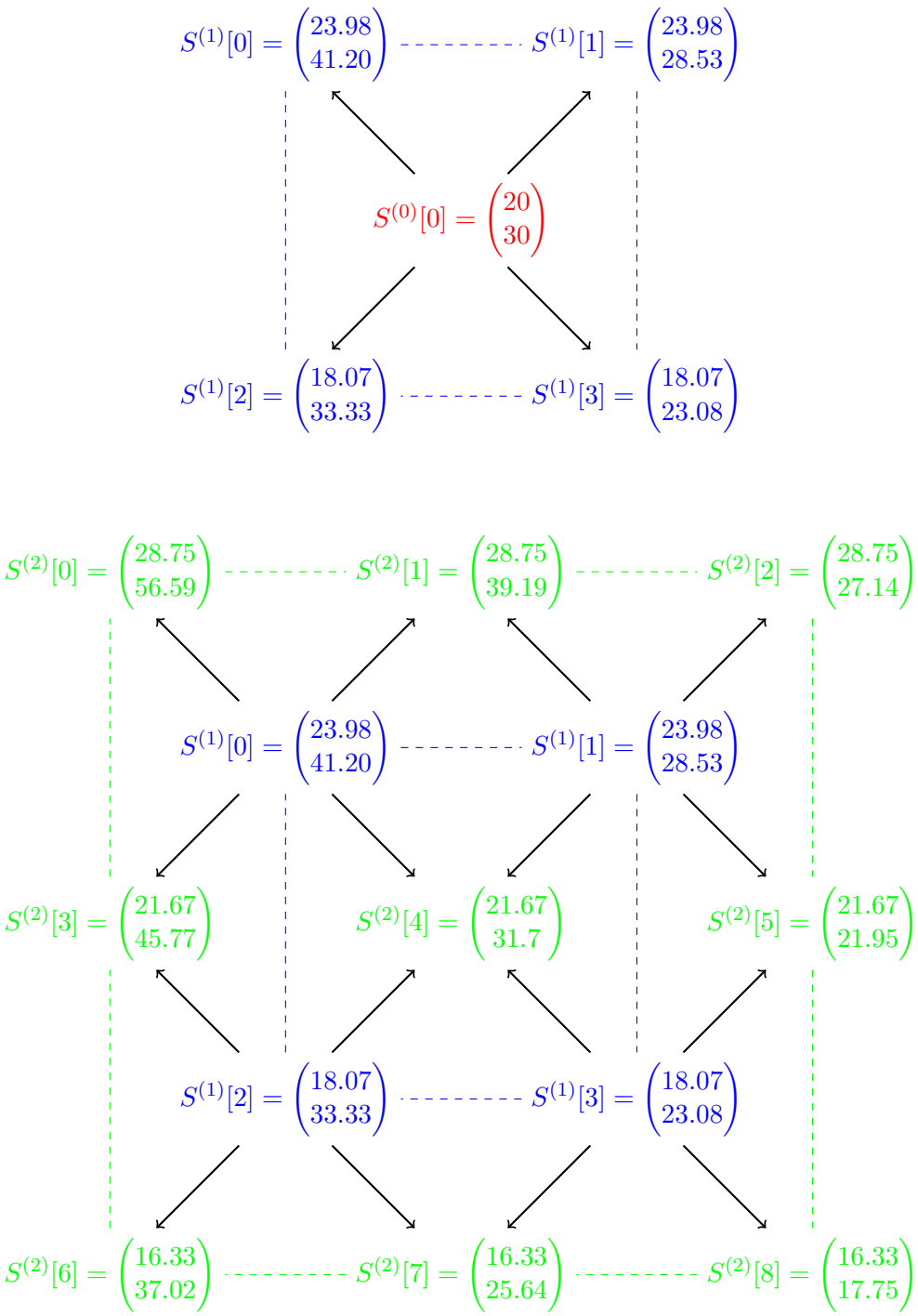


Abbildung D.5.: S-Baum von `evaluation beispielOption`

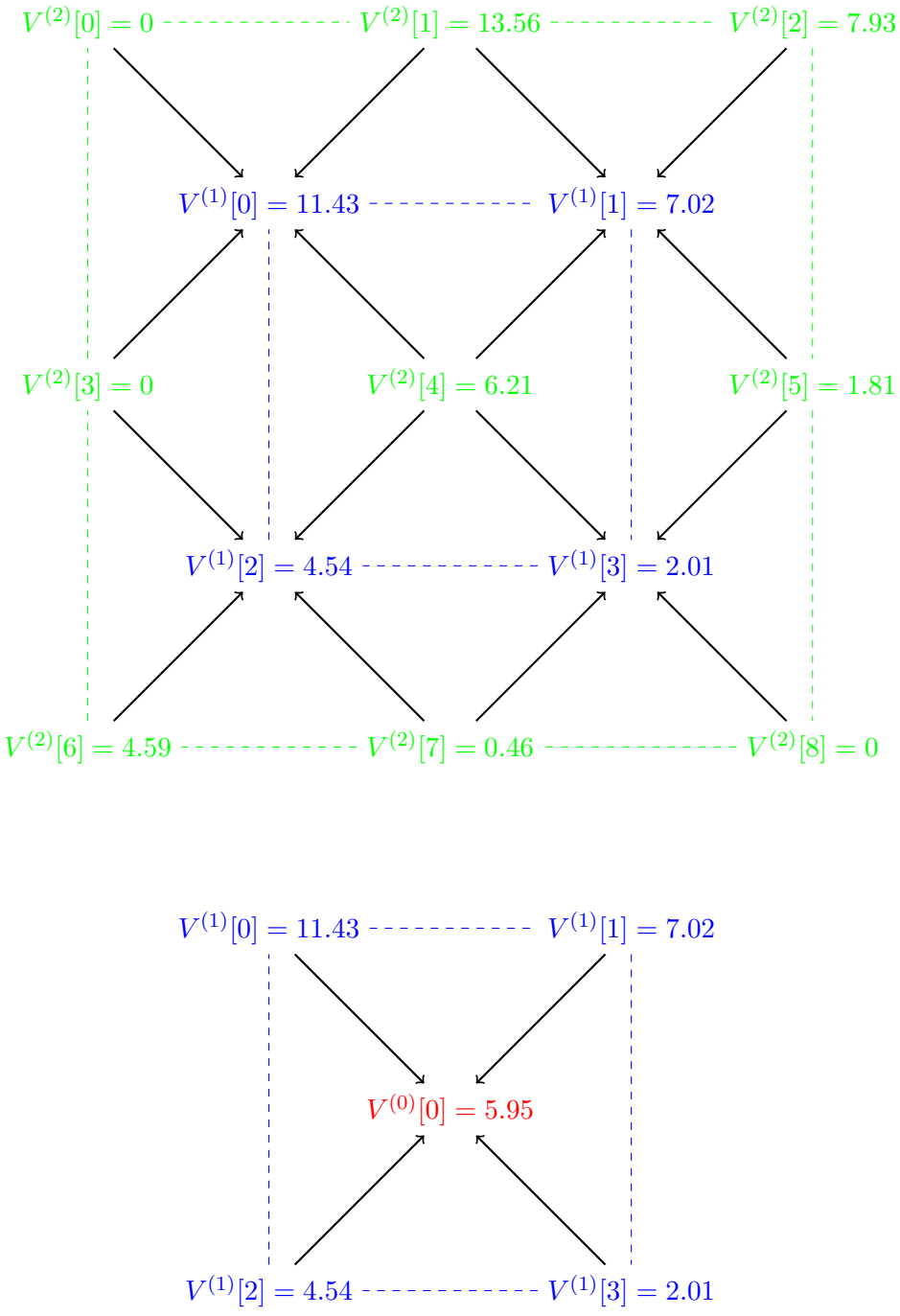


Abbildung D.6.: V-Baum von `evaluation beispielOption`

E. Zusammenstellung von Algorithmen (Teil II)

Algorithmus 28: Adjoint-Gamma-Matrix einer Bermuda-Swaption

Eingabe : Eine $(H \times M)$-Bermuda-Swaption; die Anfangslibors $L(0)$; die Volatilitätenstruktur

Ausgabe : Gamma-Matrix der Bermuda-Swaption $\mathbf{\Gamma}\left(V_{H\times M}^{BS}(T_0)\right)$

1 *Simuliere p Pfade von $L(N_{M-1},\omega)\ \forall\omega\in P=\{1,\ldots,p\}$ unter dem Spot-Maß*

2 *Führe den LSM-Algorithmus aus und bestimme die optimalen Ausübungszeiten $T_{r(\omega)}\ \forall\omega\in P$*

3 `// Berechne die Gamma-Matrix jedes Pfads`

foreach $\omega\in P$ **do**
  for $j=0$ **to** $M-1$ **do**
    for $k=0$ **to** $M-1$ **do**
      Berechne $\sum_{i=r(\omega)}^{M-1}\mathbf{\Gamma}_{jk}\left(PV_{i+1}(\omega)\cdot X_i(\omega)\right)$ *mittels der Berechnungsformel (4.35).*
    end
  end
end

4 `// Letzter Schritt der Monte-Carlo-Simulation`

for $j=0$ **to** $M-1$ **do**
  for $k=0$ **to** $M-1$ **do**
    $\mathbf{\Gamma}_{jk}\left(V_{H\times M}^{BS}(T_0)\right)\longleftarrow\frac{1}{p}\cdot\sum_{\omega\in P}\mathbf{\Gamma}_{jk}\left(V_{H\times M}^{BS}(T_0,\omega)\right)$
  end
end

Algorithmus 29: LSM-Algorithmus

Eingabe : Eine $(H \times M)$-Bermuda-Swaption; p simulierte Pfade von $L(N_{M-1}, \omega)$ $\forall \omega \in P = \{1, \ldots, p\}$; Basisfunktion $f : \mathbb{R}^M \longrightarrow \mathbb{R}^d$

Ausgabe : Wert der Bermuda-Swaption $V^{BS}_{H\times M}(T_0)$; optimale Ausübungszeiten $r(w)$ $\forall \omega \in P$

1 // Initialisierung jedes Pfads zur Zeit T_{M-1}

foreach $\omega \in P$ **do**

  // Initialisierung des inneren Wertes

  $V^{BS}_{(M-1)\times M}(T_{M-1}, \omega) \longleftarrow \max\left(0, V^{Swap}_{(M-1)\times M}(T_{M-1}, \omega)\right)$

  // Initialisierung der optimalen Ausübungszeit

  $r(w) \longleftarrow M - \mathbf{1}\left\{V^{Swap}_{(M-1)\times M}(T_{M-1}, \omega) > 0\right\}$

end

2 // Hauptteil: Rückwärtige Rekursion von T_{M-1} zu T_H jedes Pfads

for $i = M - 2$ **to** H **do**

  // Q ist eine Teilmenge von Pfaden

  $Q \longleftarrow \emptyset$

  foreach $\omega \in P$ **do**

    // Diskont von T_{i+1} zu T_i

    $V^{BS}_{i\times M}(T_i, \omega) \longleftarrow \frac{1}{1+\delta_i L_i(N_i)} \cdot V^{BS}_{(i+1)\times M}(T_{i+1}, \omega)$

    // Q wird für die Regression versammelt

    if $V^{Swap}_{i\times M}(T_i, \omega) > 0$ **then**

      $Q \longleftarrow Q \cup \{\omega\}$

    end

  end

  // Regression durch die Methode der kleinsten Quadrate

  $\alpha_i \longleftarrow \underset{\alpha\in\mathbb{R}^\kappa}{\arg\min} \sum_{\omega\in P} \left(V^{BS}_{i\times M}(T_i, \omega) - \alpha^T \cdot f\left(L(T_i, \omega)\right)\right)^2$

  // Kriterium für die Ausübungen

  foreach $\omega \in Q \subseteq P$ **do**

    if $\alpha_i^T \cdot f\left(L(T_i, \omega)\right) < V^{Swap}_{i\times M}(T_i, \omega)$ **then**

      $V^{BS}_{i\times M}(T_i, \omega) \longleftarrow V^{Swap}_{i\times M}(T_i, \omega)$

      $r(w) \longleftarrow i$

    end

  end

end

3 // Diskontiere von T_H zu T_0 entlang jedes Pfads

foreach $\omega \in P$ **do**

  $V^{BS}_{H\times M}(T_0, \omega) \longleftarrow PV_H \cdot V^{BS}_{H\times M}(T_H)$

end

4 // Letzter Schritt der Monte-Carlo-Simulation

$V^{BS}_{H\times M}(T_0) \longleftarrow \frac{1}{p} \cdot \sum_{\omega\in P} V^{BS}_{H\times M}(T_0, \omega)$

Algorithmus 30: Forward-Delta-Vektor einer Bermuda-Swaption

Eingabe : Eine $(H \times M)$-Bermuda-Swaption; die Anfangslibors $L(0)$; die Volatilitätenstruktur

Ausgabe : Delta-Vektor der Bermuda-Swaption $\boldsymbol{\Delta}\left(V^{BS}_{H\times M}(T_0)\right)$

1 *Simuliere p Pfade von $L(N_{M-1},\omega)\ \forall\omega\in P=\{1,\dots,p\}$ unter dem Spot-Maß*

2 *Führe den LSM-Algorithmus aus und bestimme die optimalen Ausübungszeiten $T_{r(\omega)}\ \forall\omega\in P$*

3 `// Berechne den Delta-Vektor jedes Pfads`

foreach $\omega\in P$ **do**

  for $j=0$ **to** $M-1$ **do**

    for $i=r(\omega)$ **to** $M-1$ **do**

      Berechne $\boldsymbol{\Delta}_j\left(PV_{i+1}(\omega)\cdot X_i(\omega)\right)$ durch die Formel (4.18), wobei sich die $\Delta_{ij}(N)$ $\forall i,j$ aus den Rekursionen (3.12) und (3.13) ergeben

    end

    `// Addiere die Delta-Vektoren aller Auszahlungen dieses Pfads`

    $\boldsymbol{\Delta}_j\left(V^{BS}_{H\times M}(T_0,\omega)\right)\longleftarrow\sum_{i=r(\omega)}^{M-1}\boldsymbol{\Delta}_j\left(PV_{i+1}(\omega)\cdot X_i(\omega)\right)$

  end

end

4 `// Letzter Schritt der Monte-Carlo-Simulation`

for $j=0$ **to** $M-1$ **do**

  $\boldsymbol{\Delta}_j\left(V^{BS}_{H\times M}(T_0)\right)\longleftarrow\frac{1}{p}\cdot\sum_{\omega\in P}\boldsymbol{\Delta}_j\left(V^{BS}_{H\times M}(T_0,\omega)\right)$

end

Algorithmus 31: Adjoint-Delta-Vektor einer Bermuda-Swaption

Eingabe : Eine $(H \times M)$-Bermuda-Swaption; die Anfangslibors $L(0)$; die Volatilitätenstruktur

Ausgabe : Delta-Vektor der Bermuda-Swaption $\boldsymbol{\Delta}\left(V_{H\times M}^{BS}(T_0)\right)$

Simuliere p Pfade von $L(N_{M-1},\omega)\ \forall\omega\in P=\{1,\ldots,p\}$ unter dem Spot-Maß

Führe den LSM-Algorithmus aus und bestimme die optimalen Ausübungszeiten $T_{r(\omega)}\ \forall\omega\in P$

`// Berechne den Delta-Vektor jedes Pfads`

foreach $\omega\in P$ **do**

  `// Initialisierung des Design-Vektors` $\mathbf{V}$ `zur Zeit` T_{M-1}

  $\mathbf{V}^\top(N_{M-1},\omega)\longleftarrow V^\top(N_{M-1},\omega|T_{M-1})$

  `// Rückwärtige Rekursion durch die Kombination von Adjoint-Methode und Design-Vektor` $\mathbf{V}$

  for $i=M-2$ **to** $r(\omega)$ **do**

    for $n=N_{i+1}-1$ **to** N_i **do**

      $\mathbf{V}(n,\omega)\longleftarrow D^\top(n,\omega)\cdot\mathbf{V}(n+1,\omega)$

    end

    $\mathbf{V}(N_i,\omega)\longleftarrow\mathbf{V}(N_i,\omega)+V_i(N_i,\omega)$

  end

  for $n=N_{r(\omega)}-1$ **to** 0 **do**

    $\mathbf{V}(n,\omega)\longleftarrow D^\top(n,\omega)\cdot\mathbf{V}(n+1,\omega)$

  end

  `//` $\mathbf{V}^\top(0,\omega)$ `ist genau der Delta-Vektor dieses Pfads`

  for $j=0$ **to** $M-1$ **do**

    $\boldsymbol{\Delta}_j\left(V_{H\times M}^{BS}(T_0,\omega)\right)\longleftarrow\mathbf{V}_j^\top(0,\omega)$

  end

end

`// Letzter Schritt der Monte-Carlo-Simulation`

for $j=0$ **to** $M-1$ **do**

  $\boldsymbol{\Delta}_j\left(V_{H\times M}^{BS}(T_0)\right)\longleftarrow\frac{1}{p}\cdot\sum_{\omega\in P}\boldsymbol{\Delta}_j\left(V_{H\times M}^{BS}(T_0,\omega)\right)$

end

Algorithmus 32: Adjoint-Delta-Vektor einer Bermuda-Swaption (Neue Version)

Eingabe : Eine $(H \times M)$-Bermuda-Swaption; die Anfangslibors $L(0)$; die Volatilitätenstruktur

Ausgabe : Delta-Vektor der Bermuda-Swaption $\boldsymbol{\Delta}\left(V_{H\times M}^{BS}(T_0)\right)$

1 *Simuliere p Pfade von $L(N_{M-1},\omega)\ \forall\,\omega \in P = \{1,\ldots,p\}$ unter dem Spot-Maß*

2 *Führe den LSM-Algorithmus aus und bestimme die optimalen Ausübungszeiten $T_{r(\omega)}\ \forall\,\omega \in P$*

3 `// Berechne den Delta-Vektor jedes Pfads`

foreach $\omega \in P$ **do**

`// Initialisierung des Anfangsvektors` $V(N)$ `zur Zeit` T_{M-1}

$$V(N_{M-1},\omega) \longleftarrow \frac{\partial\left(\sum_{i=r}^{M-1} PV_{i+1}(\omega)X_i(\omega)\right)}{\partial L(N_{M-1},\omega)}$$

`// Rückwärtige Rekursion durch die Adjoint-Methode`

for $n = N-1$ **to** 0 **do**

$$\mathbf{V}(n,\omega) \longleftarrow D^{\top}(n,\omega)\cdot\mathbf{V}(n+1,\omega)$$

end

`//` $\mathbf{V}^{\top}(0,\omega)$ `ist genau der Delta-Vektor dieses Pfads`

for $j = 0$ **to** $M-1$ **do**

$$\boldsymbol{\Delta}_j\left(V_{H\times M}^{BS}(T_0,\omega)\right) \longleftarrow \mathbf{V}_j^{\top}(0,\omega)$$

end

end

4 `// Letzter Schritt der Monte-Carlo-Simulation`

for $j = 0$ **to** $M-1$ **do**

$$\boldsymbol{\Delta}_j\left(V_{H\times M}^{BS}(T_0)\right) \longleftarrow \tfrac{1}{p}\cdot\sum_{\omega\in P}\boldsymbol{\Delta}_j\left(V_{H\times M}^{BS}(T_0,\omega)\right)$$

end

Algorithmus 33: Pathwise-Delta-Vektor unter Forward-Drift einer Bermuda-Swaption

Eingabe : Eine $(H \times M)$-Bermuda-Swaption; die Anfangslibors $L(0)$; die Volatilitätenstruktur
Ausgabe : Delta-Vektor der Bermuda-Swaption $\boldsymbol{\Delta}\left(V^{BS}_{H\times M}(T_0)\right)$

1 *Simuliere p Pfade von $L(N_{M-1},\omega)\ \forall\,\omega \in P = \{1,\dots,p\}$ durch die Forward-Drift-Approximation*

2 *Führe den LSM-Algorithmus aus und bestimme die optimalen Ausübungszeiten $T_{r(\omega)}\ \forall\,\omega \in P$*

3 `// Berechne den Delta-Vektor jedes Pfads`
foreach $\omega \in P$ **do**
  for $j = 0$ **to** $M-1$ **do**
    for $i = r(\omega)$ **to** $M-1$ **do**
      Berechne $\boldsymbol{\Delta}_j\left(PV_{i+1}(\omega)\cdot X_i(\omega)\right)$ durch die Formel (4.18), wobei sich die $\Delta_{ij}(N)$ $\forall i,j$ aus der Simulationsformel (3.20) ergeben
    end
    `// Addiere die Delta-Vektoren aller Auszahlungen dieses Pfads`
    $\boldsymbol{\Delta}_j\left(V^{BS}_{H\times M}(T_0,\omega)\right) \longleftarrow \sum_{i=r(\omega)}^{M-1} \boldsymbol{\Delta}_j\left(PV_{i+1}(\omega)\cdot X_i(\omega)\right)$
  end
end

4 `// Letzter Schritt der Monte-Carlo-Simulation`
for $j = 0$ **to** $M-1$ **do**
  $\boldsymbol{\Delta}_j\left(V^{BS}_{H\times M}(T_0)\right) \longleftarrow \frac{1}{p}\cdot\sum_{\omega\in P}\boldsymbol{\Delta}_j\left(V^{BS}_{H\times M}(T_0,\omega)\right)$
end

Algorithmus 34: Likelihood-Ratio-Delta-Vektor einer Bermuda-Swaption

Eingabe : Eine $(H \times M)$-Bermuda-Swaption; die Anfangslibors $L(0)$; die Volatilitätenstruktur

Ausgabe : Delta-Vektor der Bermuda-Swaption $\boldsymbol{\Delta}\left(V_{H\times M}^{BS}(T_0)\right)$

1 *Simuliere p Pfade von $L(N_{M-1},\omega)\ \forall\,\omega \in P = \{1,\ldots,p\}$ durch die Forward-Drift-Approximation*

2 *Führe den LSM-Algorithmus aus und bestimme die optimalen Ausübungszeiten $T_{r(\omega)}\ \forall\,\omega \in P$*

3 `// Berechne den Delta-Vektor jedes Pfads`

foreach $\omega \in P$ **do**
  for $j = 1$ **to** $M-1$ **do**
    for $i = r(\omega)$ **to** $M-1$ **do**
      if $B_{L(0)}$ *ist quadratisch* **then**
        $\boldsymbol{\Delta}_j\left(PV_{i+1}(\omega)\cdot X_i(\omega)\right) \longleftarrow$
        $(PV_{i+1}(\omega)X_i(\omega))\left\lfloor Z_{L(0)}(\omega)\right\rfloor_{N_i*d}^{\top}\left\lfloor (B_{L(0)}^{-1}\frac{\partial\bar{\mu}(L(0))}{\partial L_j(0)}\right\rfloor_{N_i*d}$
      else
        $\boldsymbol{\Delta}_j\left(PV_{i+1}(\omega)\cdot X_i(\omega)\right) \longleftarrow$
        $(PV_{i+1}(\omega)X_i(\omega))\left\lfloor (X(L(0),\omega)-\bar{\mu}(L(0))\right\rfloor_i^{\top}\left\lfloor \Sigma_{L(0)}^{-1}\cdot\frac{\partial\bar{\mu}(L(0))}{\partial L_j(0)}\right\rfloor_i$
      end
    end
    `// Addiere die Delta-Vektoren aller Auszahlungen dieses Pfads`
    $\boldsymbol{\Delta}_j\left(V_{H\times M}^{BS}(T_0,\omega)\right) \longleftarrow \sum_{i=r(\omega)}^{M-1}\boldsymbol{\Delta}_j\left(PV_{i+1}(\omega)\cdot X_i(\omega)\right)$
  end
end

4 `// Letzter Schritt der Monte-Carlo-Simulation`

for $j = 1$ **to** $M-1$ **do**
  $\boldsymbol{\Delta}_j\left(V_{H\times M}^{BS}(T_0)\right) \longleftarrow \frac{1}{p}\cdot\sum_{\omega\in P}\boldsymbol{\Delta}_j\left(V_{H\times M}^{BS}(T_0,\omega)\right)$
end

Algorithmus 35: Forward-Gamma-Matrix einer Bermuda-Swaption

Eingabe : Eine $(H \times M)$-Bermuda-Swaption; die Anfangslibors $L(0)$; die Volatilitätenstruktur

Ausgabe : Gamma-Matrix der Bermuda-Swaption $\mathbf{\Gamma}\left(V_{H\times M}^{BS}(T_0)\right)$

1 *Simuliere p Pfade von $L(N_{M-1},\omega)\ \forall\omega\in P=\{1,\ldots,p\}$ unter dem Spot-Maß*

2 *Führe den LSM-Algorithmus aus und bestimme die optimalen Ausübungszeiten $T_{r(\omega)}\ \forall\omega\in P$*

3
```
// Berechne die Gamma-Matrix jedes Pfads
```
foreach $\omega \in P$ **do**
  for $j = 0$ **to** $M-1$ **do**
    for $k = 0$ **to** $M-1$ **do**
      for $i = r(\omega)$ **to** $M-1$ **do**
        Berechne $\mathbf{\Gamma}_{jk}\left(PV_{i+1}(\omega)\cdot X_i(\omega)\right)$ durch die Forward-Methode
      end
      // Addiere die Gamma-Matrizen aller Auszahlungen dieses Pfads
      $\mathbf{\Gamma}_{jk}\left(V_{H\times M}^{BS}(T_0,\omega)\right) \longleftarrow \sum_{i=r(\omega)}^{M-1} \mathbf{\Gamma}_{jk}\left(PV_{i+1}(\omega)\cdot X_i(\omega)\right)$
    end
  end
end

4
```
// Letzter Schritt der Monte-Carlo-Simulation
```
for $j = 0$ **to** $M-1$ **do**
  for $k = 0$ **to** $M-1$ **do**
    $\mathbf{\Gamma}_{jk}\left(V_{H\times M}^{BS}(T_0)\right) \longleftarrow \frac{1}{p}\cdot\sum_{\omega\in P}\mathbf{\Gamma}_{jk}\left(V_{H\times M}^{BS}(T_0,\omega)\right)$
  end
end

F. Abbildungen zu numerischen Resultaten (Teil II)

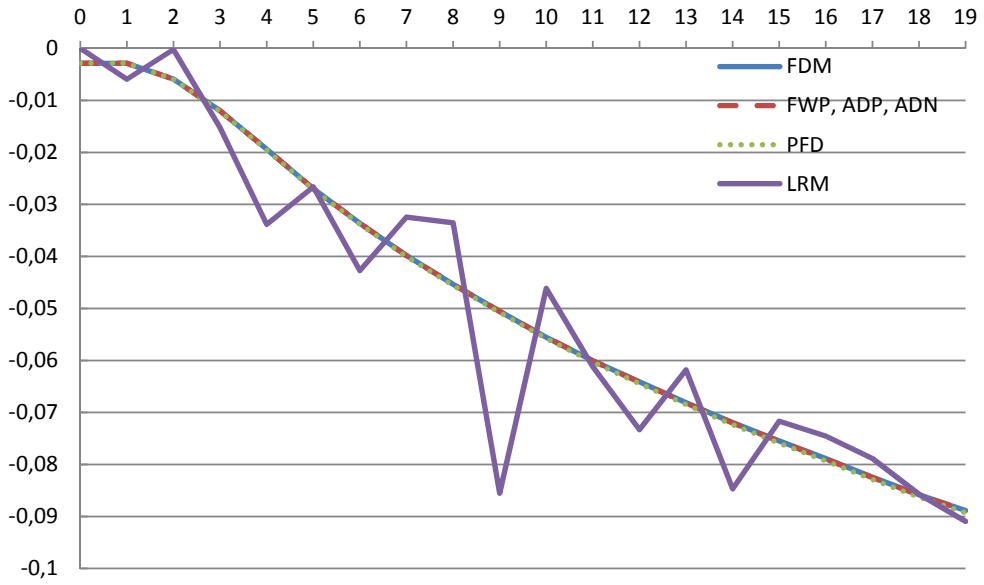


Abbildung F.1.: Delta-Vektoren einer (2×20)-Receiver-Bermuda-Swaption

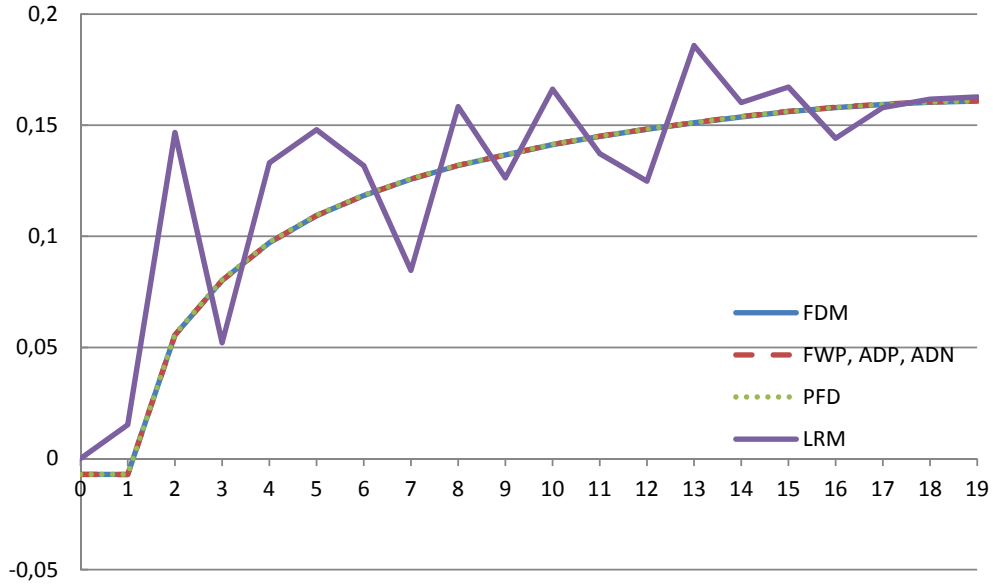


Abbildung F.2.: Delta-Vektoren einer (2×20)-Payer-Bermuda-Swaption

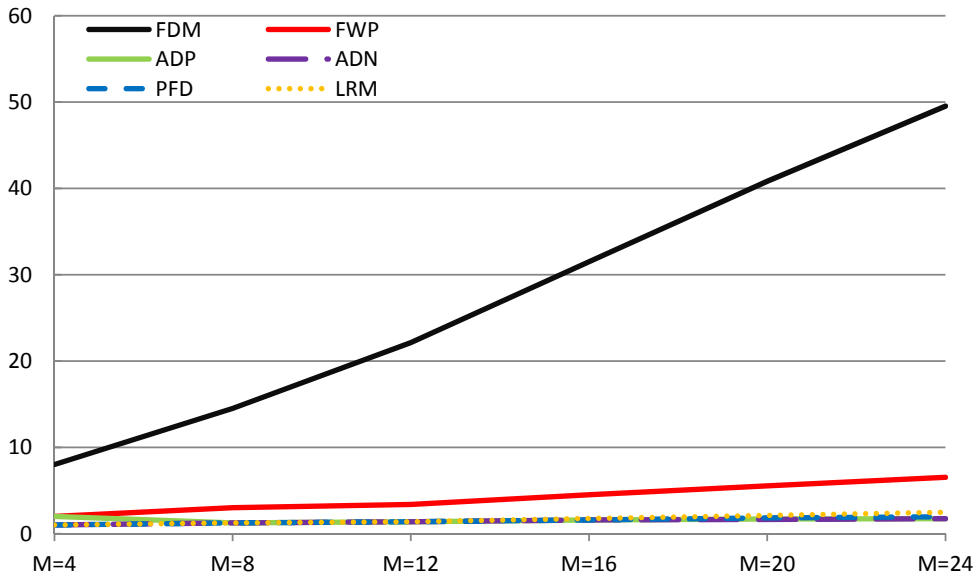


Abbildung F.3.: Relative Zeitkosten von 6 Algorithmen zur Berechnung des Delta-Vektors bzgl. Receiver-Bermuda-Swaption

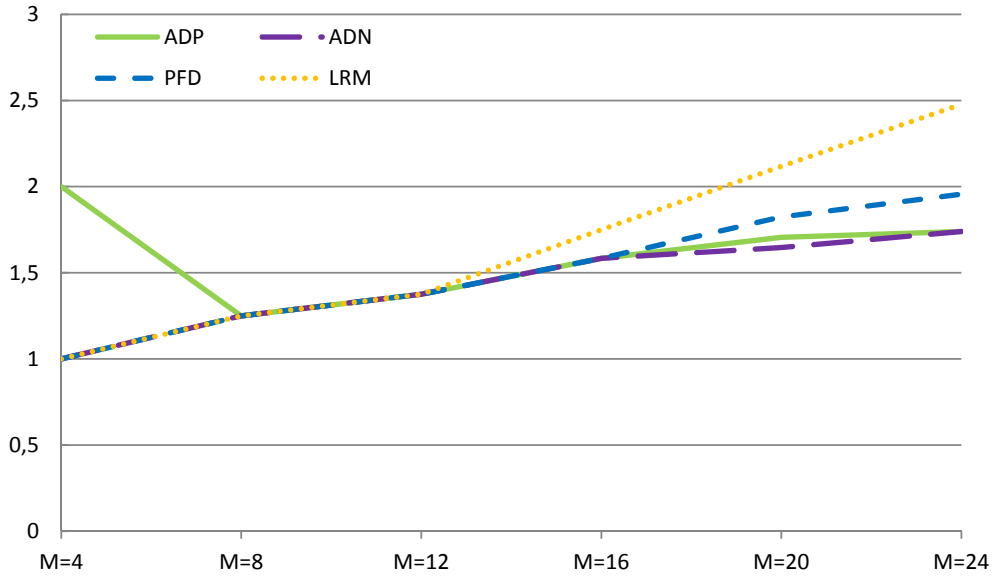


Abbildung F.4.: Relative Zeitkosten von 4 effizienten Algorithmen zur Berechnung des Delta-Vektors bzgl. Receiver-Bermuda-Swaption

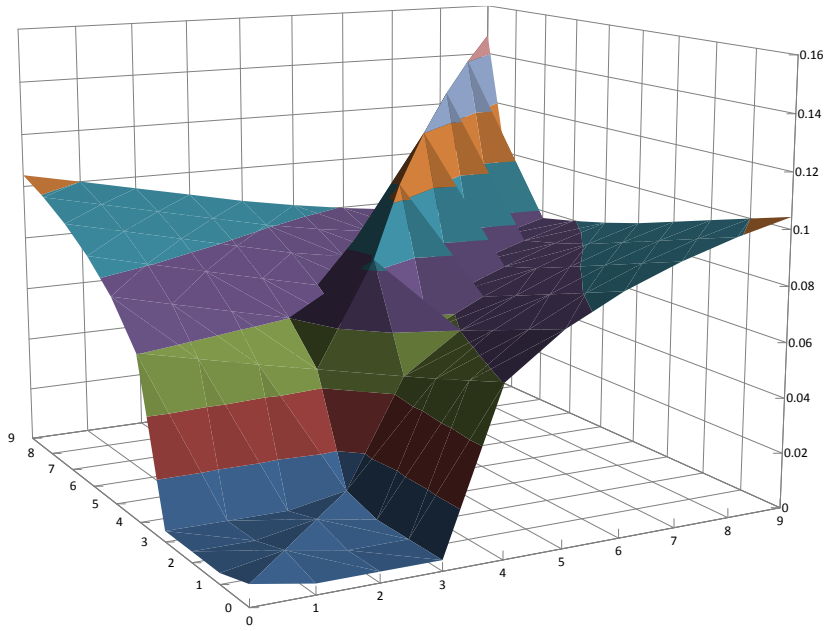


Abbildung F.5.: Exakte simulierte Gamma-Matrix einer (4×10)-Receiver-Bermuda-Swaption

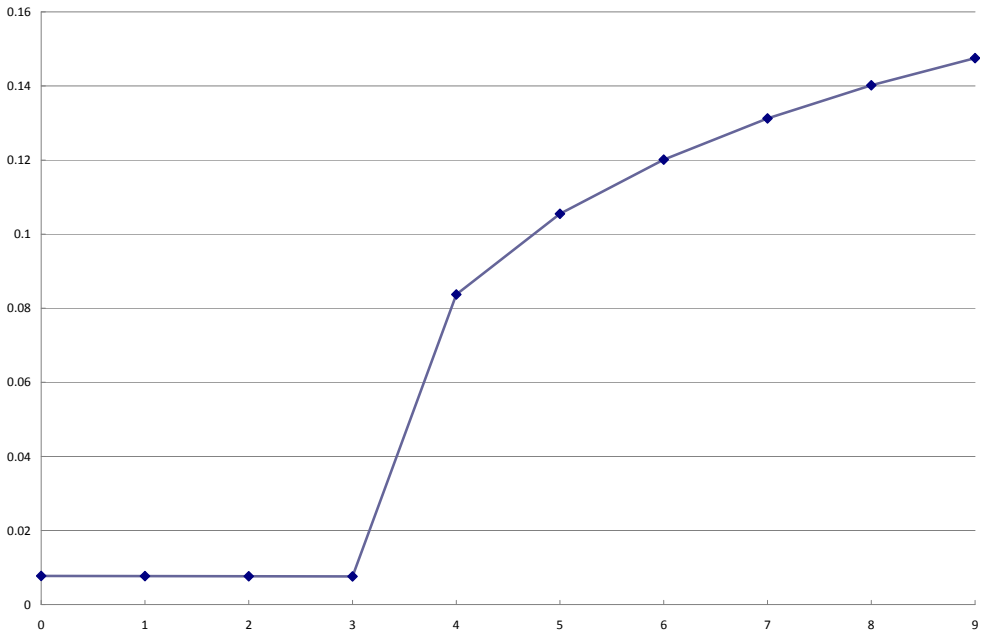


Abbildung F.6.: Exakte simulierte Gammas einer (4×10)-Receiver-Bermuda-Swaption

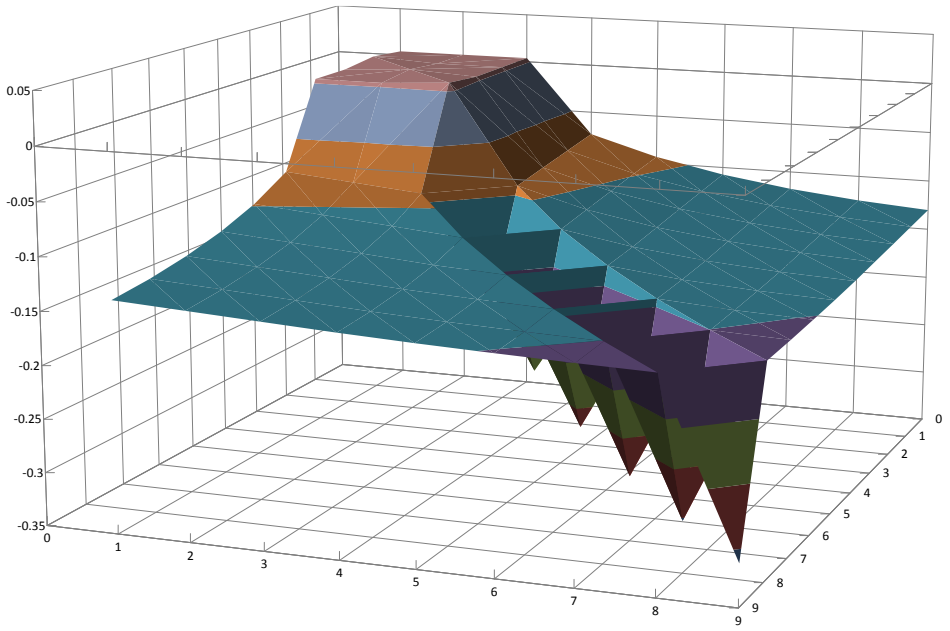


Abbildung F.7.: Exakte simulierte Gamma-Matrix einer (4×10)-Payer-Bermuda-Swaption

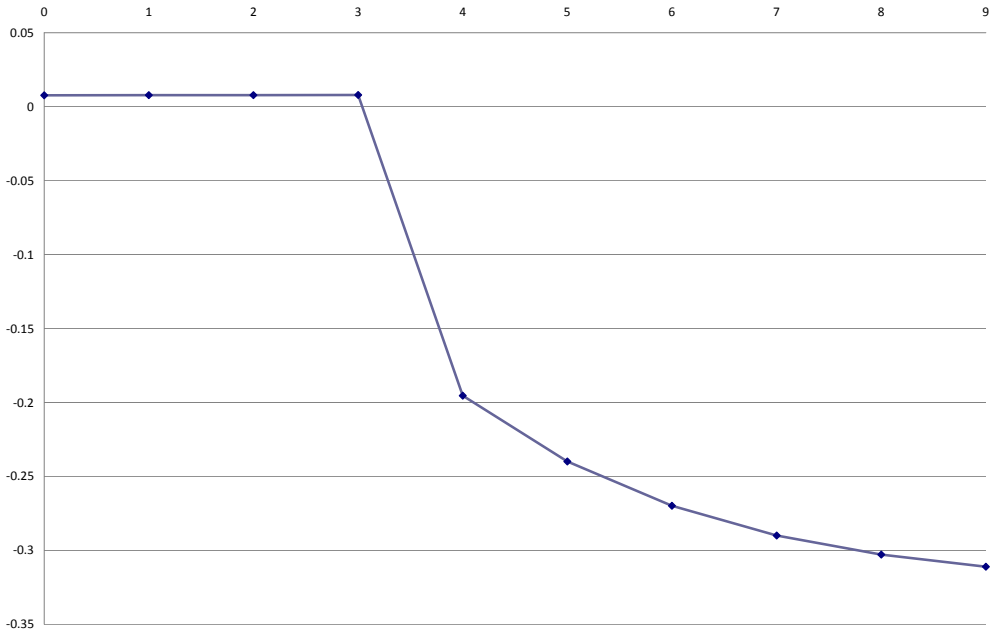


Abbildung F.8.: Exakte simulierte Gammas einer (4×10)-Payer-Bermuda-Swaption

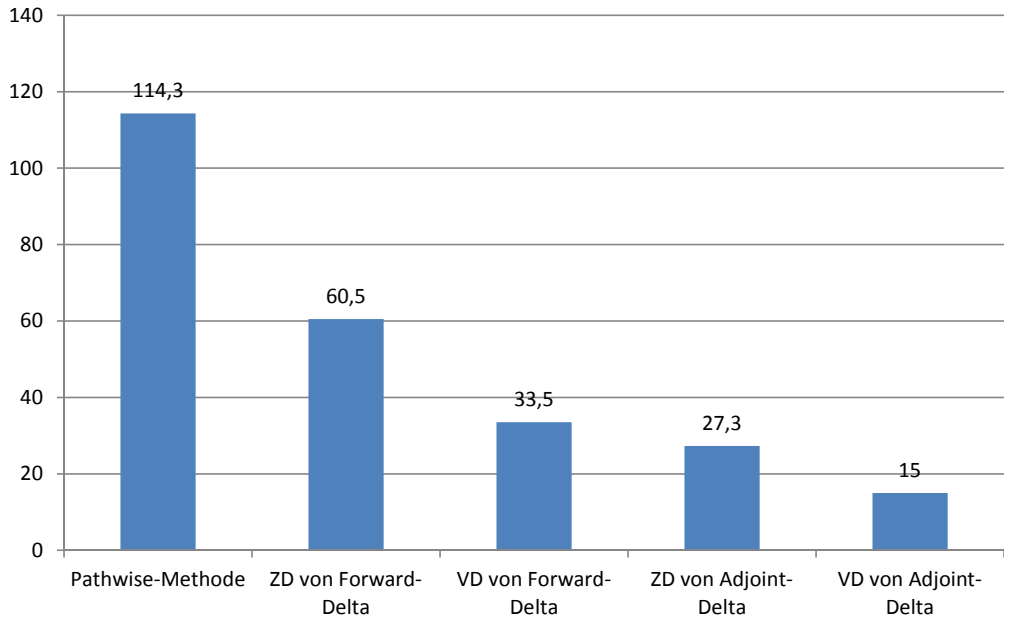


Abbildung F.9.: Relative Zeitkosten zur Berechnung der Gamma-Matrix bzgl. einer (4×10)-Receiver-Bermuda-Swaption

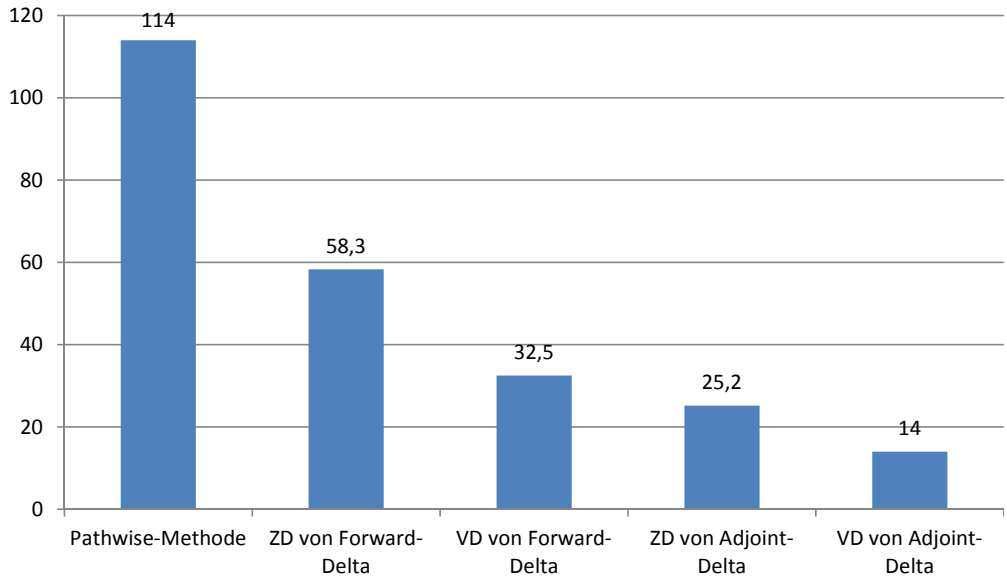


Abbildung F.10.: Relative Zeitkosten zur Berechnung der Gamma-Matrix bzgl. einer (4×10)-Payer-Bermuda-Swaption

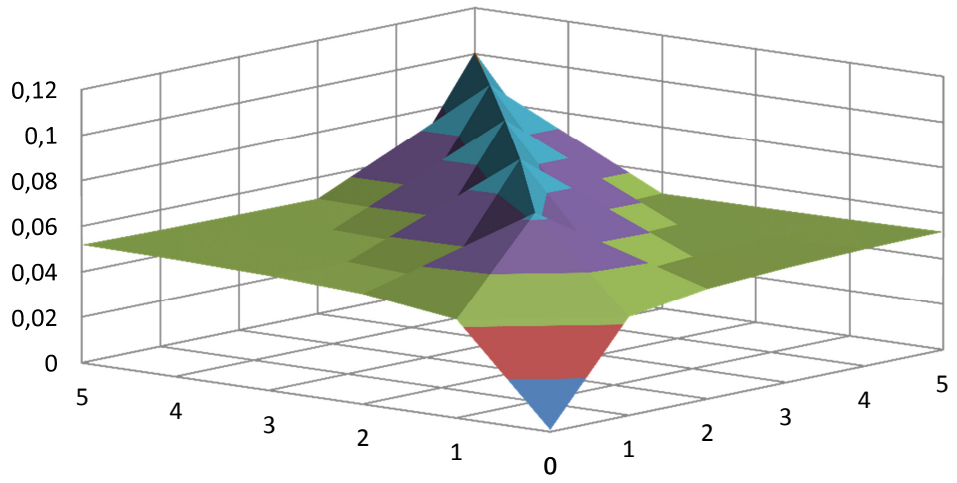


Abbildung F.11.: Gamma-Matrix einer (1×6)-Receiver-Bermuda-Swaption

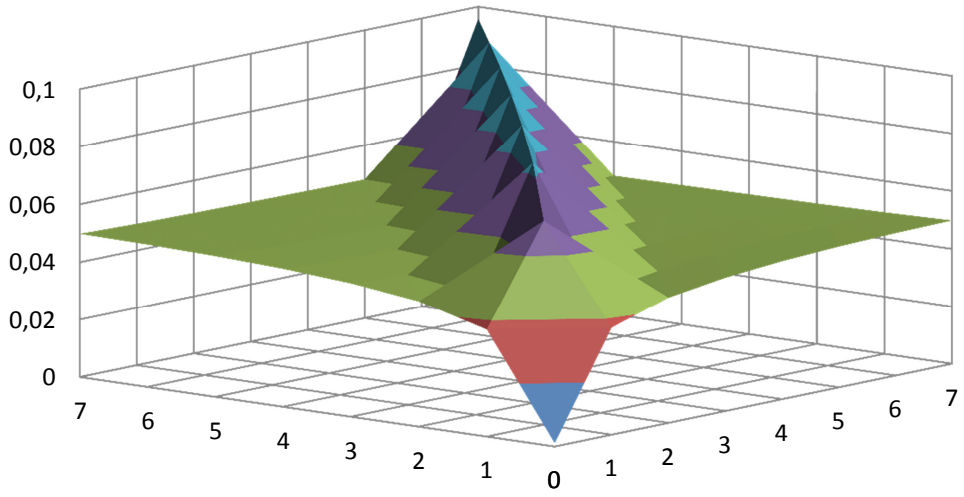


Abbildung F.12.: Gamma-Matrix einer (1×8)-Receiver-Bermuda-Swaption

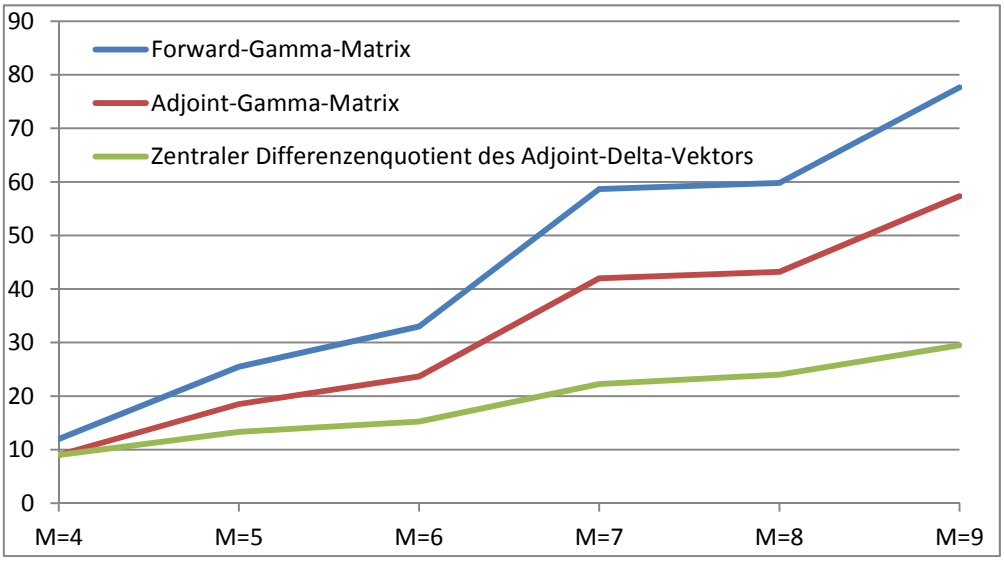


Abbildung F.13.: Vergleich der relativen Zeitkosten der beiden Pathwise-Methoden für Gamma-Matrix

G. Tabellen zu numerischen Resultaten (Teils II)

	0	1	2	3	4	5	6	7	8	9
0	0.0	–	–	–	–	–	–	–	–	–
1	0.0	0.0	–	–	–	–	–	–	–	–
2	0.0	0.0	0.0	–	–	–	–	–	–	–
3	0.0	0.0	0.0	0.0	–	–	–	–	–	–
4	0.0	0.0	0.0	0.0	0.0	–	–	–	–	–
5	0.0	0.0	0.0	0.0	0.0	0.0	–	–	–	–
6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	–	–	–
7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	–	–
8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	–
9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Tabelle G.1.: Symmetrieverhalten der exakten simulierten Gamma-Matrix einer (4×10)-Receiver-Bermuda-Swaption mittels Pathwise-Methode

	0	1	2	3	4	5	6	7	8	9
0	0.0	–	–	–	–	–	–	–	–	–
1	0.0	0.0	–	–	–	–	–	–	–	–
2	0.0	0.0	0.0	–	–	–	–	–	–	–
3	0.0	0.0	0.0	0.0	–	–	–	–	–	–
4	0.0	0.0	0.0	0.0	0.0	–	–	–	–	–
5	0.0	0.0	0.0	0.0	0.0	0.0	–	–	–	–
6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	–	–	–
7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	–	–
8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	–
9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Tabelle G.2.: Symmetrieverhalten der exakten simulierten Gamma-Matrix einer (4×10)-Payer-Bermuda-Swaption mittels Pathwise-Methode

	0	1	2	3	4	5	6	7	8	9
0	0.007705317	0.003852367	0.003849428	0.003848935	0.060389738	0.075396718	0.085569918	0.093338713	0.099494627	0.104418305
S.f.	1.68014E-05	7.27847E-06	6.14283E-06	5.02390E-06	0.000146502	0.000126694	0.000145502	0.000181513	0.000211442	0.000227773
1	0.003852367	0.007692090	0.003817925	0.003803865	0.055468997	0.070210029	0.080433145	0.088416321	0.094809343	0.100011904
S.f.	7.27847E-06	1.36795E-05	5.77607E-06	4.71709E-06	0.000138932	0.000127113	0.000149586	0.000185581	0.000214855	0.000230673
2	0.003849428	0.003817925	0.007646189	0.003757694	0.050667315	0.065070583	0.075316187	0.083446390	0.090060951	0.095547863
S.f.	6.14283E-06	5.77607E-06	1.07512E-05	4.43909E-06	0.000129427	0.000124604	0.000150689	0.000187246	0.000216205	0.000231929
3	0.003848935	0.003803865	0.003757694	0.007582427	0.046100942	0.060078663	0.070281234	0.078527982	0.085347079	0.091063933
S.f.	5.02390E-06	4.71709E-06	4.43909E-06	8.31462E-06	0.000118589	0.000119668	0.000149289	0.000186740	0.000215741	0.000231502
4	0.060389738	0.055468997	0.050667315	0.046100942	0.083717324	0.055242198	0.065354447	0.073660652	0.080655824	0.086588127
S.f.	0.000146502	0.000138932	0.000129427	0.000118589	0.000213404	0.000112764	0.000145866	0.000184398	0.000213679	0.000229676
5	0.075396718	0.070210029	0.065070583	0.060078663	0.055242198	0.105487566	0.061963693	0.069974857	0.076900276	0.082863512
S.f.	0.000126694	0.000127113	0.000124604	0.000119668	0.000112764	0.000231109	0.000146280	0.000183563	0.000212116	0.000227847
6	0.085569918	0.080433145	0.075316187	0.070281234	0.065354447	0.061963693	0.120123225	0.067235795	0.073845502	0.079679549
S.f.	0.000145502	0.000149586	0.000150689	0.000149289	0.000145866	0.000146280	0.000303351	0.000185190	0.000211970	0.000226983
7	0.093338713	0.088416321	0.083446390	0.078527982	0.073660652	0.069974857	0.067235795	0.131256594	0.071467456	0.077017388
S.f.	0.000181513	0.000185581	0.000187246	0.000186740	0.000184398	0.000183563	0.000185190	0.000379104	0.000213491	0.000227248
8	0.099494627	0.094809343	0.090060951	0.085347079	0.080655824	0.076900276	0.073845502	0.071467456	0.140184942	0.074963325
S.f.	0.000211442	0.000214855	0.000216205	0.000215741	0.000213679	0.000212116	0.000211970	0.000213491	0.000434804	0.000229351
9	0.104418305	0.100011904	0.095547863	0.091063933	0.086588127	0.082863512	0.079679549	0.077017388	0.074963325	0.147512423
S.f.	0.000227773	0.000230673	0.000231929	0.000231502	0.000229676	0.000227847	0.000226983	0.000227248	0.000229351	0.000467584

Tabelle G.3.: Exakte simulierte Gamma-Matrix und ihre Standardfehler einer (4×10)-Receiver-Bermuda-Swaption mittels Pathwise-Methode

	0	1	2	3	4	5	6	7	8	9
0	0.007721128	0.003872643	0.003882858	0.003899306	-0.07351034	-0.09329660	-0.10741508	-0.11762514	-0.12502153	-0.13018330
S.f.	2.97565E-05	1.69223E-05	1.89926E-05	2.13159E-05	0.000277054	0.000248697	0.000211056	0.000177494	0.000155371	0.000145681
1	0.003872643	0.007789951	0.003876704	0.003880676	-0.07921521	-0.09884861	-0.11252655	-0.12224513	-0.12917494	-0.13387489
S.f.	1.69223E-05	3.97069E-05	2.18840E-05	2.42933E-05	0.000304846	0.000273258	0.000233892	0.000199068	0.000175310	0.000163374
2	0.003882858	0.003876704	0.007826839	0.003857286	-0.08522157	-0.10460066	-0.11777316	-0.12693793	-0.13334355	-0.13757594
S.f.	1.89926E-05	2.18840E-05	5.04614E-05	2.74227E-05	0.000333177	0.000297941	0.000256294	0.000220115	0.000194911	0.000181393
3	0.003899306	0.003880676	0.003857286	0.007874102	-0.09138348	-0.11041498	-0.12308411	-0.13168949	-0.13754369	-0.14128529
S.f.	2.13159E-05	2.42933E-05	2.74227E-05	6.35714E-05	0.000361635	0.000322489	0.000279185	0.000242047	0.000216160	0.000201829
4	-0.07351034	-0.07921521	-0.08522157	-0.09138348	-0.19547592	-0.11653202	-0.12857468	-0.13655105	-0.14179742	-0.14505646
S.f.	0.000277054	0.000304846	0.000333177	0.000361635	0.000779449	0.000347623	0.000302550	0.000264784	0.000238180	0.000223240
5	-0.09329660	-0.09884861	-0.10460066	-0.11041498	-0.11653202	-0.23996771	-0.13243936	-0.14021649	-0.14508631	-0.14799692
S.f.	0.000248697	0.000273258	0.000297941	0.000322489	0.000347623	0.000743447	0.000327341	0.000289635	0.000262670	0.000247006
6	-0.10741508	-0.11252655	-0.11777316	-0.12308411	-0.12857468	-0.13243936	-0.26989292	-0.14311420	-0.14785258	-0.15053558
S.f.	0.000211056	0.000233892	0.000256294	0.000279185	0.000302550	0.000327341	0.000707474	0.000317121	0.000289849	0.000273877
7	-0.11762514	-0.12224513	-0.12693793	-0.13168949	-0.13655105	-0.14021649	-0.14311420	-0.29001840	-0.15011297	-0.15272899
S.f.	0.000177494	0.000199068	0.000220115	0.000242047	0.000264784	0.000289635	0.000317121	0.000694008	0.000320629	0.000304062
8	-0.12502153	-0.12917494	-0.13334355	-0.13754369	-0.14179742	-0.14508631	-0.14785258	-0.15011297	-0.30286817	-0.15434750
S.f.	0.000155371	0.000175310	0.000194911	0.000216160	0.000238180	0.000262670	0.000289849	0.000320629	0.000705685	0.000336503
9	-0.13018330	-0.13387489	-0.13757594	-0.14128529	-0.14505646	-0.14799692	-0.15053558	-0.15272899	-0.15434750	-0.31111090
S.f.	0.000155371	0.000175310	0.000194911	0.000216160	0.000238180	0.000262670	0.000289849	0.000320629	0.000705685	0.000336503

Tabelle G.4.: Exakte simulierte Gamma-Matrix und ihre Standardfehler einer (4×10)-Payer-Bermuda-Swaption mittels Pathwise-Methode

	0	1	2	3	4	5	6	7	8	9
0	2.19430E-12	2.21924E-12	2.22213E-12	2.19936E-12	3.59081E-11	4.33818E-11	4.97087E-11	5.46993E-11	5.88045E-11	6.00210E-11
1	8.84531E-14	3.47910E-13	4.44865E-14	7.74094E-14	1.54593E-12	3.93921E-12	1.41125E-12	4.36727E-12	3.01686E-12	4.15189E-12
2	9.29101E-14	1.12252E-13	3.67850E-13	1.28148E-13	1.78843E-12	1.99431E-12	1.83936E-12	3.37398E-12	1.34336E-12	2.39088E-12
3	2.24183E-14	2.02533E-14	3.01148E-14	2.80470E-13	9.86738E-13	4.41709E-13	3.44355E-12	2.13767E-12	1.93608E-12	2.23110E-12
4	1.68853E-12	1.57691E-12	1.39724E-12	1.27726E-12	3.56963E-12	1.40904E-12	3.65803E-12	2.85554E-12	4.25966E-12	1.04769E-12
5	2.20175E-12	1.99801E-12	1.88473E-12	1.71865E-12	1.51742E-12	4.30210E-12	1.39053E-12	2.00234E-12	4.15878E-12	3.58936E-12
6	2.39977E-12	2.29489E-12	2.17226E-12	1.99017E-12	2.26916E-12	1.62306E-12	5.15639E-12	2.17423E-12	1.32050E-12	4.18178E-12
7	2.74715E-12	2.59871E-12	2.46066E-12	2.30978E-12	1.77298E-12	1.65390E-12	1.71073E-12	4.39002E-12	1.04318E-12	1.23525E-12
8	2.93036E-12	2.79213E-12	2.64699E-12	2.50436E-12	2.45197E-12	2.30273E-12	1.67966E-12	2.62278E-12	3.74867E-12	4.94896E-13
9	3.01260E-12	2.87849E-12	2.74593E-12	2.61748E-12	2.33095E-12	2.58563E-12	1.93179E-12	2.34444E-12	2.25830E-12	3.51508E-12

Tabelle G.5.: Die Matrix der absoluten Differenz zwischen Pathwise-Gamma-Matrix und zentralen Differenzenquotient-Gamma-Matrix des Forward- oder Adjoint-Deltas bzgl. einer (4×10)-Receiver-Bermuda-Swaption

	0	1	2	3	4	5	6	7	8	9
0	5.53833E-10	1.77063E-10	1.76985E-10	1.76956E-10	2.77566E-09	3.46718E-09	3.93587E-09	4.28994E-09	4.57139E-09	4.80137E-09
1	1.87537E-10	5.60421E-10	1.84270E-10	1.82909E-10	2.46930E-09	3.16964E-09	3.67287E-09	4.06465E-09	4.38864E-09	4.65815E-09
2	1.86354E-10	1.83334E-10	5.51189E-10	1.77356E-10	2.02335E-09	2.68293E-09	3.17815E-09	3.58224E-09	3.92495E-09	4.21533E-09
3	1.84902E-10	1.80665E-10	1.76429E-10	5.37296E-10	1.62140E-09	2.22448E-09	2.69899E-09	3.10571E-09	3.45808E-09	3.76667E-09
4	2.04014E-09	1.84464E-09	1.65447E-09	1.47287E-09	3.81381E-09	1.79823E-09	2.24601E-09	2.64264E-09	2.99892E-09	3.32363E-09
5	2.57063E-09	2.35393E-09	2.14000E-09	1.93197E-09	1.73066E-09	4.75188E-09	1.93946E-09	2.28949E-09	2.62658E-09	2.94311E-09
6	2.92747E-09	2.70457E-09	2.48300E-09	2.26509E-09	2.05185E-09	1.91185E-09	5.32779E-09	2.04046E-09	2.33260E-09	2.62222E-09
7	3.19866E-09	2.97703E-09	2.75430E-09	2.53384E-09	2.31658E-09	2.15816E-09	2.04655E-09	5.70770E-09	2.11273E-09	2.36139E-09
8	3.41617E-09	3.19800E-09	2.97749E-09	2.75843E-09	2.54076E-09	2.37354E-09	2.24509E-09	2.14402E-09	5.98270E-09	2.17196E-09
9	3.59483E-09	3.38164E-09	3.16690E-09	2.95081E-09	2.73621E-09	2.56364E-09	2.42313E-09	2.30756E-09	2.22316E-09	6.18664E-09

Tabelle G.6.: Die Matrix der absoluten Differenz zwischen Pathwise-Gamma-Matrix und Vorwärtsdifferenzenquotient-Gamma-Matrix des Forward- oder Adjoint-Deltas bzgl. einer (4×10)-Receiver-Bermuda-Swaption

	0	1	2	3	4	5	6	7	8	9
0	0.0	–	–	–	–	–	–	–	–	–
1	2.13078E-12	0.0	–	–	–	–	–	–	–	–
2	2.12922E-12	6.77657E-14	0.0	–	–	–	–	–	–	–
3	2.17695E-12	9.76628E-14	9.80331E-14	0.0	–	–	–	–	–	–
4	3.42196E-11	3.09752E-14	3.91194E-13	2.90525E-13	0.0	–	–	–	–	–
5	4.11800E-11	1.94120E-12	1.09579E-13	1.27694E-12	1.08379E-13	0.0	–	–	–	–
6	4.73089E-11	8.83640E-13	3.32900E-13	1.45338E-12	1.38888E-12	2.32536E-13	0.0	–	–	–
7	5.19522E-11	1.76856E-12	9.13325E-13	1.72112E-13	1.08255E-12	3.48443E-13	4.63504E-13	0.0	–	–
8	5.58741E-11	2.24737E-13	3.99035E-12	5.68282E-13	1.80769E-12	1.85606E-12	3.59157E-13	1.57960E-12	0.0	–
9	5.70084E-11	1.27340E-12	3.55049E-13	3.86371E-13	1.28327E-12	1.00374E-12	2.24999E-12	1.10920E-12	2.75320E-12	0.0

Tabelle G.7.: Symmetrieverhalten der Gamma-Matrix einer (4×10)-Receiver-Bermuda-Swaption mittels zentrales Differenzenquotienten des Forward- oder Adjoint-Deltas

	0	1	2	3	4	5	6	7	8	9
0	0.0	–	–	–	–	–	–	–	–	–
1	1.04739E-11	0.0	–	–	–	–	–	–	–	–
2	9.36990E-12	9.35701E-13	0.0	–	–	–	–	–	–	–
3	7.94609E-12	2.24379E-12	9.26673E-13	0.0	–	–	–	–	–	–
4	7.35521E-10	6.24661E-10	3.68884E-10	1.48527E-10	0.0	–	–	–	–	–
5	8.96554E-10	8.15707E-10	5.42936E-10	2.92513E-10	6.75688E-11	0.0	–	–	–	–
6	1.00840E-09	9.68299E-10	6.95154E-10	4.33899E-10	1.94151E-10	2.76089E-11	0.0	–	–	–
7	1.09128E-09	1.08762E-09	8.27940E-10	5.71867E-10	3.26057E-10	1.31328E-10	6.09129E-12	0.0	–	–
8	1.15522E-09	1.19064E-09	9.47461E-10	6.99652E-10	4.58161E-10	2.53035E-10	8.75097E-11	3.12923E-11	0.0	–
9	1.20654E-09	1.27651E-09	1.04843E-09	8.15860E-10	5.87414E-10	3.79471E-10	1.99095E-10	5.38347E-11	5.12077E-11	0.0

Tabelle G.8.: Symmetrieverhalten der Gamma-Matrix einer (4×10)-Receiver-Bermuda-Swaption mittels Vorwärtsdifferenzenquotienten des Forward- oder Adjoint-Deltas

	0	1	2	3	4	5	6	7	8	9
0	2.32695E-12	2.34269E-12	2.29016E-12	2.33876E-12	4.43239E-11	5.57992E-11	6.47673E-11	6.97011E-11	7.45473E-11	7.67703E-11
1	3.01074E-13	2.53017E-13	3.81845E-13	4.93913E-13	1.83815E-12	5.75454E-12	3.98770E-12	7.26673E-12	5.68309E-12	3.34313E-12
2	2.92166E-14	5.41780E-14	3.02154E-14	1.58298E-13	2.05112E-12	4.00417E-12	3.21411E-12	2.35875E-12	4.42479E-12	8.17318E-13
3	6.38144E-14	6.44589E-14	1.34857E-13	5.59587E-14	4.53874E-12	1.35969E-12	1.58178E-12	6.04952E-12	3.15711E-12	2.58923E-12
4	2.26690E-12	2.39946E-12	2.61902E-12	2.71849E-12	6.70303E-12	4.32963E-12	4.30486E-12	3.42723E-12	2.66759E-12	2.89807E-12
5	2.68237E-12	2.92022E-12	3.06613E-12	3.19107E-12	2.67665E-12	6.97656E-12	2.55421E-12	3.95542E-12	6.55490E-12	5.18913E-12
6	3.22063E-12	3.35856E-12	3.48044E-12	3.67163E-12	3.83482E-12	3.89169E-12	6.46766E-12	1.98960E-12	3.03849E-12	7.28065E-12
7	3.51166E-12	3.68403E-12	3.81262E-12	3.88539E-12	3.89186E-12	4.95617E-12	4.74454E-12	6.54826E-12	4.95676E-12	3.85544E-12
8	3.64653E-12	3.74192E-12	3.87290E-12	3.97282E-12	3.68305E-12	4.10563E-12	4.03924E-12	4.54511E-12	9.22057E-12	1.95932E-12
9	6.21659E-09	0.007444735	0.013922089	0.019855179	0.024733918	0.030407298	0.037346489	0.043606531	0.049439071	15.48323841

Tabelle G.9.: Die Matrix der absoluten Differenz zwischen Pathwise-Gamma-Matrix und zentralen Differenzenquotient-Gamma-Matrix des Forward- oder Adjoint-Deltas bzgl. einer (4×10)-Payer-Bermuda-Swaption

	0	1	2	3	4	5	6	7	8	9
0	5.54700E-10	1.77742E-10	1.78239E-10	1.78871E-10	3.37727E-09	4.28693E-09	4.93615E-09	5.40267E-09	5.74257E-09	5.98330E-09
1	1.89434E-10	5.75666E-10	1.89167E-10	1.88425E-10	4.15463E-09	5.09778E-09	5.74328E-09	6.18158E-09	6.50086E-09	6.70588E-09
2	1.90786E-10	1.89677E-10	5.82122E-10	1.87206E-10	4.78053E-09	5.68509E-09	6.26904E-09	6.65218E-09	6.91189E-09	7.06866E-09
3	1.91947E-10	1.89628E-10	1.86938E-10	5.88398E-10	5.43214E-09	6.28110E-09	6.80205E-09	7.11273E-09	7.32086E-09	7.42573E-09
4	4.76532E-09	5.07111E-09	5.39207E-09	5.72288E-09	1.83618E-08	6.90372E-09	7.34018E-09	7.58782E-09	7.72694E-09	7.77935E-09
5	5.85002E-09	6.12496E-09	6.40899E-09	6.69638E-09	7.00315E-09	2.17547E-08	7.70477E-09	7.92008E-09	8.01381E-09	8.03383E-09
6	6.57934E-09	6.81471E-09	7.05258E-09	7.29826E-09	7.55530E-09	7.73867E-09	2.38318E-08	8.17872E-09	8.25651E-09	8.24648E-09
7	7.06997E-09	7.26709E-09	7.46342E-09	7.66859E-09	7.88266E-09	8.04272E-09	8.17402E-09	2.50701E-08	8.44819E-09	8.43032E-09
8	7.38324E-09	7.54854E-09	7.71002E-09	7.88071E-09	8.05471E-09	8.18528E-09	8.30067E-09	8.39849E-09	2.56722E-08	8.55637E-09
9	4.84159E-09	0.014889478	0.027844186	0.039710366	0.049467844	0.060814605	0.074692986	0.087213071	0.098878151	30.96647685

Tabelle G.10.: Die Matrix der absoluten Differenz zwischen Pathwise-Gamma-Matrix und Vorwärtsdifferenzenquotient-Gamma-Matrix des Forward- oder Adjoint-Deltas bzgl. einer (4×10)-Payer-Bermuda-Swaption

Tabelle G.11.: Symmetrieverhalten der Gamma-Matrix einer (4×10)-Payer-Bermuda-Swaption mittels zentrales Differenzenquotienten des Forward- oder Adjoint-Deltas

	0	1	2	3	4	5	6	7	8	9
0	0.0	–	–	–	–	–	–	–	–	–
1	2.04161E-12	0.0	–	–	–	–	–	–	–	–
2	2.26095E-12	3.27667E-13	0.0	–	–	–	–	–	–	–
3	2.27495E-12	4.29454E-13	2.34409E-14	0.0	–	–	–	–	–	–
4	4.20570E-11	5.61301E-13	5.67893E-13	1.82025E-12	0.0	–	–	–	–	–
5	5.31168E-11	2.83432E-12	9.38041E-13	1.83138E-12	1.65298E-12	0.0	–	–	–	–
6	6.15467E-11	6.29136E-13	2.66329E-13	2.08986E-12	4.70041E-13	1.33749E-12	0.0	–	–	–
7	6.61894E-11	3.58270E-12	1.45386E-12	2.16413E-12	4.64628E-13	1.00076E-12	2.75494E-12	0.0	–	–
8	7.09008E-11	1.94117E-12	5.51892E-13	8.15709E-13	1.01547E-12	2.44926E-12	1.00076E-12	4.11643E-13	0.0	–
9	6.13982E-09	0.007444735	0.013922089	0.019855179	0.024733918	0.030407298	0.037346489	0.043606531	0.049439071	0.0

	0	1	2	3	4	5	6	7	8	9
0	0.0	–	–	–	–	–	–	–	–	–
1	1.16913E-11	0.0	–	–	–	–	–	–	–	–
2	1.25472E-11	5.09859E-13	0.0	–	–	–	–	–	–	–
3	1.30761E-11	1.20314E-12	2.68715E-13	0.0	–	–	–	–	–	–
4	1.38805E-09	9.16473E-10	6.11538E-10	2.90743E-10	0.0	–	–	–	–	–
5	1.56309E-09	1.02718E-09	7.23894E-10	4.15277E-10	9.94336E-11	0.0	–	–	–	–
6	1.64319E-09	1.07144E-09	7.83538E-10	4.96208E-10	2.15119E-10	3.38944E-11	0.0	–	–	–
7	1.66731E-09	1.08552E-09	8.11242E-10	5.55859E-10	2.94845E-10	1.22638E-10	4.69366E-12	0.0	–	–
8	1.64067E-09	1.04768E-09	7.98128E-10	5.59843E-10	3.27777E-10	1.71469E-10	4.41591E-11	4.96939E-11	0.0	–
9	1.08249E-08	0.014889471	0.027844179	0.039710359	0.049467837	0.060814597	0.074692978	0.087213062	0.098878143	0.0

Tabelle G.12.: Symmetrieverhalten der Gamma-Matrix einer (4×10)-Payer-Bermuda-Swaption mittels Vorwärtsdifferenzenquotienten des Forward- oder Adjoint-Deltas

Literaturverzeichnis

[1] **Aitsahlia, F.**, **Imhof, L.** und **Lai, T. L.** (2004). *Pricing and Hedging of American Knock-In Options*. The Journal of Derivatives, Spring 2004, Vol. 11, No. 3, 44-50.

[2] **Alexander, C.** und **Venkatramanan, A.** (2009). *Analytic Approximations for Multi-Asset Option Pricing*. ICMA Centre Discussion Papers in Finance DP2009-05.

[3] **Black, F.** und **Scholes, M.** (1973). *The Pricing of Options and Corporate Liabilties*. Journal of Political Economy 81, 637-654.

[4] **Brace, A.**, **Gatarek, D.** und **Musiela, M.** (1997). *The Market Model of Interest Rate Dynamics*. Mathematical Finance 7, 127-155.

[5] **Braun, D.** (2009). *Eine Einführung in Haskell*.

[6] **Buschmann, C.** (2002). *Abstrakte Datentypen und das Typsystem von Haskell*.

[7] **Cox, J. C.**, **Ross, S. A.** and **Rubinstein, M.** (1979). *Option Pricing: a Simplified Approach*. Journal of Financial Economics 7, 229-263.

[8] **Daume III, H.** (2006). *Yet Another Haskell Tutorial*.

[9] **Gaillourdet, Jean-Marie** (2011). *A Software Language Approach to Derivative Contracts in Finance*. Nachwuchsring des Landesforschungszentrum Center for Mathematical and Computational Modelling $(CM)^2$, 40-44.

[10] **Giles, M.** und **Glasserman, P.** (2006). *Smoking Adjoints: Fast Monte Carlo Greeks*. Risk Januar 2006, 88-92.

[11] **Glasserman, P.** (2004). *Monte Carlo Methods in Financial Engineering*. Springer Science.

[12] **Glasserman, P.** und **Zhao, X.** (1999). *Fast Greeks by Simulation in Forward Libor Models*. Journal of Computational Finance 3, 5-39.

[13] **Hoek, J. v. d.** und **Elliott, R. J.** (2006). *Binomial Models in Finance*. Springer Finance.

[14] **Hull, J. C.** (2008). *Options, Futures, and Other Derivatives*. 7te Auflage, Pearson Education.

[15] **Jamshidian, F.** (1997). *Libor und Swap Market Modells and Measures*. Finance and Stochastics 1, 293-330.

[16] **Jarrow, R.** und **Rudd, A.** (1983). *Option Pricing*. Irwin Professional Pub.

[17] **Korn, R.** und **Korn, E.** (2009). *Optionsbewertung und Portfolio-Optimierung - moderne Methoden der Finanzmathematik*. 2te Auflage, Vieweg.

[18] **Korn, R.**, **Korn, E.** und **Kroisandt, G.** (2010). *Monte Carlo Methods and Models in Finance and Insurance*. ChapmanundHall/CRC Financial Mathematics Series.

[19] **Korn, R.** und **Müller, S.** (2009). *Getting Multi-Dimensional Trees into a New Shape*. WILMOTT Journal, Volume 1(3), 145–153.

[20] **Korn, R.** und **Müller, S.** (2009). *The decoupling approach to binomial pricing of multi-asset options*. Journal of Computational Finance, Volume 12(3), 1-30.

[21] **Kürten, C.** (2008). *Dünngitter-Binomialbäume zur Bewertung von Multiasset-Optionen*. Diplomarbeit an der Rheinischen Friedrich-Wilhelms-Universität Bonn.

[22] **Leclerc, M.**, **Liang, Q.** und **Schneider, I.** (2009). *Fast Monte Carlo Bermudan Greeks*. Risk Juli 2009, 84-88.

[23] **Li, M. Q.**, **Deng, S. J.** und **Zhou, J. Y.** (2008). *Multi-asset Spread Option Pricing and Hedging*. MPRA Paper No. 8259, posted 14.

[24] **Liang, Q.** (2008). *Eine systematische Untersuchung der Risikoparameter mittels der adjungierten Methode im Rahmen der Simulation des LIBORMarktmodells*. Diplomarbeit an der Universität Augsburg.

[25] **Longstaff, F. A.** und **Schwartz, E. S.** (2001). *Valuing American Options by Simulation: A Simple Least-Squares Approach*. Review of Financial Studies 14, 113-147.

[26] **Miltersen, K.**, **Sandmann, K.** und **Sondermann, D.** (1997). *Closed Form Solutions for Term Structure Derivatives with Lognormal Interest Rate*. Journal of Finance 52, 409-430.

[27] **Newbern, J.**. *All About Monads - A comprehensive guide to the theory and practice of monadic programming in Haskell*.

[28] **O'Sullivan, B.**, **Stewart, D. B.** und **Goerzen, J.** (2008). *Real World Haskell*. O'Reilly.

[29] **Peyton-Jones, S.** (1999). *Haskell 98 Language and Libraries - The Revised Report*.

[30] **Peyton-Jones, S.** und **Eber, J.-M.** (2003) *How to write a financial contract*. Palgrave Macmillan.

[31] **Peyton-Jones, S.**, **Eber, J.-M.** und **Seward, J.** (2000) *Composing contracts: an adventure in financial engineering*; ICFP 2000.

[32] **Piterbarg, V. V.** (2003). *Computing Deltas of Callable Libor Exotics in Forward Libor Models*. Journal of Computational Finance 7(3), 107-144.

[33] **Puhlmann, H.** (2004). *Funktionale Programmierung mit Haskell*.

[34] **Quecke, S.** (2007). *Efficient Numerical Methods for Pricing American Options under Levy Models*. Dissertation an der Universität zu Köln.

[35] **Zhang, P. G.** (1998). *Exotic options - a guide to second generation options*. 2te Auflage, World Scientific.

Index

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die Dissertation mit dem Thema

“Innovative Techniken und Algorithmen im Bereich Computational-Finance und Risikomanagement”

ohne fremde Hilfe angefertigt habe und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt habe.

Wissenschaftlicher Werdegang

Bildungsweg in Deutschland

04.2003 – 05.2008	Studium der Mathematik an der Universität Augsburg Schwerpunkt Optimierung und Operations Research Nebenfach Informatik
10.2005	Vordiplom in Mathematik an der Universität Augsburg Note: sehr gut
05.2008	Diplom in Mathematik an der Universität Augsburg Note: sehr gut
10.2007 – 08.2008	Diplomand/Praktikant bei der DekaBank, Deutsche Girozentrale, Frankfurt am Main
08.2008 – 06.2012	Doktorand bei Prof. Dr. Ralf Korn an der TU Kaiserslautern Schwerpunkt Finanzmathematik
seit 08.2008	Mitglied des Center for Mathematical and Computational Modelling $(CM)^2$

Wissenschaftliche Veröffentlichungen

05.2008	*Eine systematische Untersuchung der Risikoparameter mittels der adjungierten Methode im Rahmen der Simulation des LIBOR-Marktmodells*, Diplomarbeit (Dipl.-Math.), Universität Augsburg
07.2009	*Fast Monte Carlo Bermudan Greeks*, Risk, zusammen mit Prof. Dr. Matthias Leclerc und Dr. Ingo Schneider
08.2009	*Fast Monte Carlo Bermudan Greeks*, Asia Risk, zusammen mit Prof. Dr. Matthias Leclerc und Dr. Ingo Schneider
2012	*Monte Carlo Simulation of Deltas for Bermudan Swaptions in the LIBOR Market Model*, zusammen mit Prof. Dr. Ralf Korn, erscheint in WILMOTT Journal
2012	*Robust and accurate Monte Carlo simulation of (Cross-) Gammas for Bermudan swaptions in the LIBOR market model*, zusammen mit Prof. Dr. Ralf Korn, erscheint in Journal of Computational Finance

2012 *Adjoint LIBOR (Cross) Gammas for Bermudan swaption*, zusammen mit Prof. Dr. Ralf Korn, wird erreicht in Risk

Scientific Career

Education in Germany

04.2003 – 05.2008	Study of mathematics at University of Augsburg specialization in optimization und operations research minor: computer science
10.2005	Vordiplom in mathematics at University of Augsburg top grade
05.2008	Diplom in mathematics at University of Augsburg top grade
10.2007 – 08.2008	Diplomand/intern at DekaBank, Deutsche Girozentrale, Frankfurt am Main
08.2008 – 06.2012	PhD student of Prof. Dr. Ralf Korn at the Kaiserslautern University of Technology specialization in mathematical finance
since 08.2008	member of the Center for Mathematical and Computational Modelling (CM)^2

Scientific Publications

05.2008	*Eine systematische Untersuchung der Risikoparameter mittels der adjungierten Methode im Rahmen der Simulation des LIBOR-Marktmodells*, Diplomarbeit (Dipl.-Math.), University of Augsburg
07.2009	*Fast Monte Carlo Bermudan Greeks*, Risk, with Prof. Dr. Matthias Leclerc and Dr. Ingo Schneider
08.2009	*Fast Monte Carlo Bermudan Greeks*, Asia Risk, with Prof. Dr. Matthias Leclerc and Dr. Ingo Schneider
2012	*Monte Carlo Simulation of Deltas for Bermudan Swaptions in the LIBOR Market Model*, with Prof. Dr. Ralf Korn, appears in WILMOTT Journal
2012	*Robust and accurate Monte Carlo simulation of (Cross-) Gammas for Bermudan swaptions in the LIBOR market model*, with Prof. Dr. Ralf Korn, appears in Journal of Computational Finance

2012 *Adjoint LIBOR (Cross) Gammas for Bermudan swaption*, with Prof. Dr. Ralf Korn, to be submitted to Risk