Entwicklung einer graphischen Benutzeroberfläche zur Auswertung von Röntgendiffraktogrammen ungeordneter Systeme

Von der Fakultät Chemie der Universität Stuttgart
zur Erlangung der Würde eines Doktors der
Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Dipl.-Chem. Ulrich Eberhardinger

aus Stuttgart

Hauptberichter:	Prof. Dr. H. Bertagnolli
Mitberichter:	Prof. Dr. K. Müller
Tag der mündlichen Prüfung:	22. April 2002

Institut für Physikalische Chemie
der Universität Stuttgart
2002

Danksagungen und Widmung

Die vorliegende Arbeit entstand in der Zeit vom März 1997 bis April 2002 unter der Leitung von Herrn Dekan Prof. Dr. H. Bertagnolli am Institut für Physikalische Chemie an der Universität Stuttgart.

Meinem Doktorvater Prof. Bertagnolli danke ich für die freundliche Betreuung und großzügige Unterstützung meiner Arbeit. Insbesondere danke ich ihm, dass er den interdisziplinären Ansatz dieser Arbeit interessiert mitgetragen hat. Schon während meines Studiums begeisterte mich Prof. Bertagnolli durch seine lehrreichen Vorlesungen und sein Engagement für die Physikalische Chemie. Er ist mir ein Vorbild als Wissenschaftler und als Mensch.

Mein besonderer Dank gilt meinem Kollegen Gerhard Heusel, der mir die Ergebnisse und Daten seiner Untersuchungen an der Bismutchloridschmelze zur Verfügung gestellt hat. Diese Messungen waren wichtig für die Überprüfung meines Programms und die Darstellung dessen Möglichkeiten im Rahmen der Musterauswertung in der vorliegenden Dissertation.

Ich danke allen meinen Kollegen in der Arbeitsgruppe, Herrn Dr. Leicht und Frau Hoppe für die gute Zusammenarbeit und das hervorragende Arbeitsklima. Besonders erwähnen möchte ich meinen Zimmerkollegen Björn Schmid, mit dem ich Freude und Leid geteilt habe und zahlreiche interessante Gespräche über wissenschaftliche und unwissenschaftliche Themen führen durfte. Seine außerordentliche Hilfsbereitschaft und sein ebenso fachkundiger wie unermüdlicher Einsatz für die Hard- und Software des Institutes verdienen viel Dank und Anerkennung. Nennen möchte ich auch die Herren Martin P. Feth und Michael Seiler, mit denen ich ebenfalls sehr gerne zusammengearbeitet habe und die zudem mein Programm einem Praxistest unterzogen haben. Die gemeinsame Arbeit mit Euch allen hat mir viel Spaß gemacht!

Herrn Bernhard Riedhofer danke ich herzlich für die aufmerksame Durchsicht dieser Arbeit und für seine moralische Unterstützung. Meinen Eltern und Großeltern danke ich, dass sie mir mein Studium und meine Promotion ermöglicht haben und mich dabei nicht nur finanziell, sondern vor allem durch ihre große Liebe unterstützt haben.

Diese Arbeit widme ich meiner Familie.

Inhaltsverzeichnis

Symbolverzeichnis

2θ	Beugungswinkel
$\vec{\kappa}$	Streuvektor
λ	Wellenlänge
μ	linearer Absorptionskoeffizient
ρ	Massendichte
$\bar{\rho}$	mittlere Teilchendichte
χ_T	isotherme Kompressibilität
C	Comptonstreuung
c	Lichtgeschwindigkeit
C_p	Molwärme bei konstantem Druck
E	Energie
f_0	Atomformfaktor
f_e	mittlerer Atomformfaktor
$g(r)$	Atompaarkorrelationsfunktion
$G(r)$	Gesamtkorrelationsfunktion
h	Plancksche Konstante
I	Intensität
I_{atom}	Atomstreuung
I_{red}	reduzierte Intensität
$\vec{k}$	Wellenvektor
K	Koordinationszahl
m	Masse
m_e	Elektronenmasse
M	Molmasse
n	Stoffmenge
N_A	Avogadro-Konstante
N	Teilchenzahl
V	Volumen
p	Druck
r	interatomarer Abstand
R	Gaskonstante
w	Wichtungsfaktor
$\bar{z}$	mittlere effektive Elektronenzahl
Z	Kernladungszahl

Kapitel 1

Einführung

Bei seiner Arbeit mit Kathodenstrahlröhren entdeckte Wilhelm Conrad Röntgen im Jahre 1895, dass bei der Bestrahlung dünner Metallfolien mit durch Hochspannung beschleunigten Elektronen eine bis dahin unbekannte Strahlung auftrat, die in der Lage war, Materie scheinbar ungehindert zu durchdringen [1]. Diese wurde von Röntgen als X-Strahlung bezeichnet und fand bald breite Anwendung in Medizin und Technik. Laue, Friedrich und Knipping stellten 1912 bei der Bestrahlung von Kupfersulfat fest, dass die von der Probe ausgehende Strahlung auf Photoplatten regelmäßige Punktmuster erzeugte, welche sie auf Interferenzerscheinungen zurückführten, die durch Beugung am Kristallgitter verursacht wurden. Laue, Bragg, Debye und Scherrer entwickelten verschiedene Verfahren, um aus den Interferenzmustern die Kristallstrukturen zu bestimmen. Debye konnte 1915 zeigen, dass sich auch die Struktur ungeordneter Systeme wie Flüssigkeiten und Gläser mit Hilfe der Röntgenbeugung bestimmen lässt. Aber erst mit der Entwicklung leistungsfähiger Rechner fand das rechenintensive Verfahren zur Auswertung von Diffraktogrammen ungeordneter Systeme eine größere Verbreitung und gehört heute zu den Standardverfahren in der Physikalischen Chemie und angewandten Wissenschaften. Durch die Verfügbarkeit leistungsfähiger Röntgenquellen wie der Drehanode oder des Synchrotrons, welche Strahlung hoher Energie und Intensität erzeugen, ist nun auch die Untersuchung stark absorbierender Substanzen oder die Messung in Druckapparaturen möglich. Gleichzeitig kommen zunehmend speziellere Verfahren wie die energiedispersive und anomale Röntgenbeugung zum Einsatz, die einerseits eine effizientere Ausnutzung der Strahlung erlauben und andererseits zusätzliche Strukturinformationen liefern. Durch die Messung bei kleinen Beugungswinkeln kann zudem die mittelreichweitige Ordnung von Systemen wie z. B. den Polymeren bestimmt werden. Damit liefert die Röntgenbeugung einen direkten Zugang zur Bestimmung atomarer Abstände in kondensierter Materie.

Das in dieser Arbeit vorgestellte Programm dient der Auswertung von Röntgendiffraktogrammen ungeordneter Systeme. Es wurde in der 1995 eingeführten Sprache Java entwickelt, die aufgrund ihrer Unterstützung objektorientierter Techniken sehr gut für die Programmierung komplexer Benutzeroberflächen geeignet ist. Java-Programme bieten zudem den Vorteil, dass sie in compilierter Form unter zahlreichen Betriebssystemen wie Windows, Solaris, Linux und MacOS unverändert ausgeführt werden können, die im universitären Bereich eingesetzt werden.

Kapitel 2

Grundlagen

2.1 Röntgenbeugung an ungeordneten Systemen

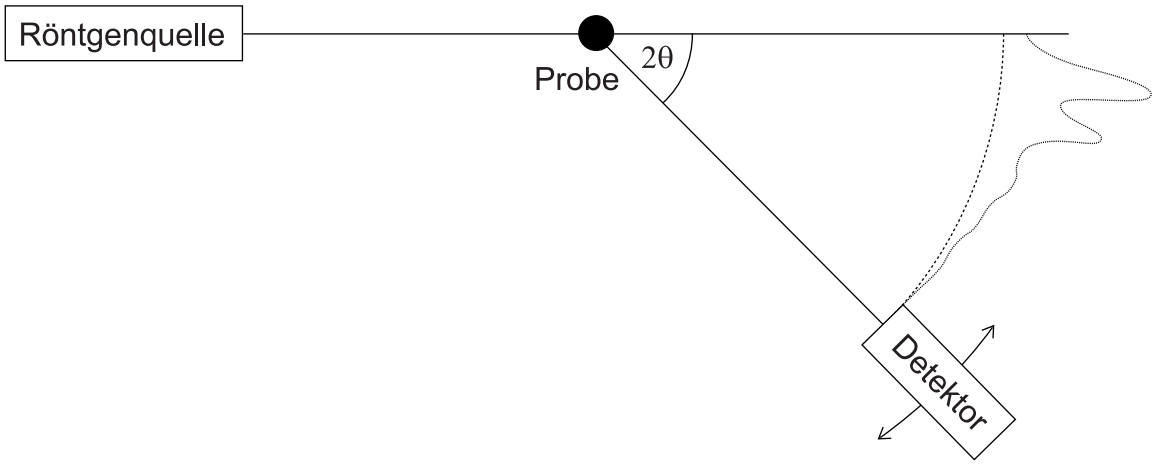


Abbildung 2.1: Aufbau eines Röntgenbeugungsexperimentes

Der prinzipielle Aufbau eines winkeldispersiven Röntgenbeugungsexperimentes zur Strukturuntersuchung ungeordneter Systeme ist in Abbildung 2.1 dargestellt: Die monochromatische Strahlung einer Röntgenquelle trifft auf eine Probe und wird dort gebeugt. Ein Detektor misst die Intensität der gebeugten Strahlung in Abhängigkeit des Beugungswinkels 2θ. Die gepunktete Linie zeigt den typischen Verlauf der gemessenen Intensität einer amorphen Probe. Auf der Primärseite — also zwischen der Quelle und der Probe — befindet sich zudem ein Filter oder besser ein Monochromator zur Erzeugung der monochromatischen Strahlung, die bei der winkeldispersiven Methode benötigt wird.

Die Beugung der Strahlung findet an Elektronen statt. Abbildung 2.2 zeigt, wie ein Röntgenphoton der Wellenlänge λ mit dem zum Wellenvektor $\vec{k_0}$ proportionalen Impuls $\vec{p_0} = \hbar \cdot \vec{k_0}$ auf ein Elektron trifft. Bei elastischer bzw. kohärenter Streuung haben die Wellenvektoren sowohl des eingehenden wie auch des ausgehenden Photons den gleichen Betrag $\left|\vec{k}\right| = 2\pi/\lambda$.

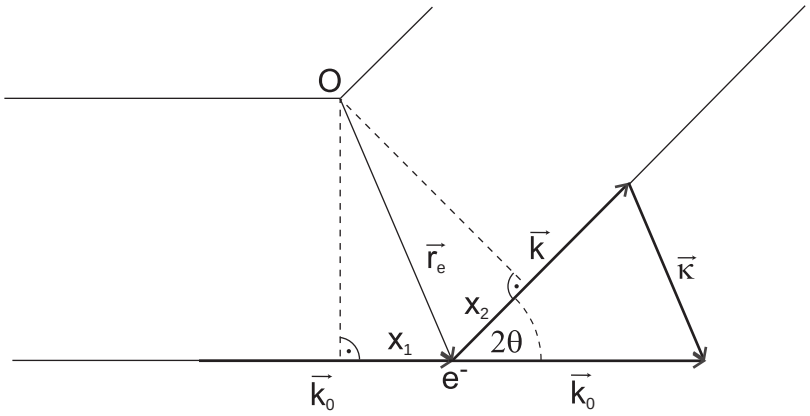


Abbildung 2.2: Beugung eines Röntgenstrahls an einem Elektron

Aufgrund der Richtungsänderung tritt ein Streuvektor $\vec{\kappa} = \vec{k}_0 - \vec{k}$ mit dem Betrag

$$|\vec{\kappa}| = \frac{4\pi \sin\theta}{\lambda} \tag{2.1}$$

auf [2]. Ein Teil der Photonen wird inelastisch gestreut. Der Wellenvektor dieser Comptonstreuung hat einen kleineren Betrag als derjenige der einfallenden Strahlung. Die Comptonstreuung ist meist unerwünscht und kann entweder durch einen sekundärseitigen Monochromator zum Teil experimentell beseitigt werden oder bei der Auswertung[1] berücksichtigt werden.

Neben der Beugung treten bei dem beschriebenen Experiment weitere Effekte auf, die bei der Auswertung zu berücksichtigen sind. Der Röntgenstrahl wird bei der Beugung an der Probe und am Monochromator polarisiert und ändert dadurch seine Intensität. Eine Polarisationskorrektur berechnet diejenige Intensität, die zu erwarten wäre, wenn keine Polarisation aufträte. Außerdem wird ein Teil der Strahlung sowohl durch die Probe als auch die für fluide Proben erforderliche Messzelle absorbiert. Für eine Vielzahl möglicher Geometrien der Messanordnung existieren Absorptionskorrekturen. Weiterhin kann Röntgenfluoreszenz auftreten, wenn die Probe Elemente enthält, deren Absorptionskanten nur wenig unterhalb der Energie der einfallenden Strahlung liegen. Auch die Fluoreszenz kann durch einen sekundärseitigen Monochromator oder einen energieauflösenden Detektor eliminiert werden oder trägt andernfalls als nahezu winkelunabhängiger Untergrund zur Gesamtintensität bei.

Ähnlich wie bei der Beugung von Licht an einem optischen Gitter interferieren die von den Elektronen ausgehenden Kugelwellen und erzeugen so ein Beugungsmuster, das in direktem Zusammenhang mit der Anordnung der Elektronen in der bestrahlten Probe steht. Das Verhältnis der wesentlich kleineren Atomabstände zum Abstand der Gitterstriche entspricht dabei der wesentlich kleineren Wellenlänge der Röntgenstrahlung gegenüber der Wellenlänge des sichtbaren Lichtes. Für ein Elektron, das gegenüber dem Ursprung O um seinen Ortsvektor $\vec{r}_e$ verschoben ist, tritt gegenüber einem Strahl durch den Ursprung ein Gangunterschied $x_1 + x_2$ auf, der in Abhängigkeit von Ort $\vec{r}_e$ und Streuvektor $\vec{\kappa}$ zu

[1] bei der Normierung, siehe Abschnitt 4.2.6

konstruktiver oder destruktiver Interferenz führt. Die Gesamtintensität ergibt sich durch Berücksichtigung der Interferenz aller Kugelwellen, die von den im System vorhandenen Elektronen ausgehen.

Zur Bestimmung der relativen Anordnung der Atome trennt man rechnerisch denjenigen Anteil der Streuung, der von Elektronen innerhalb eines Atomes herrührt, nämlich die Atomstreuung, von dem Anteil ab, welcher von Elektronen verschiedener Atome stammt. Die Atomstreuung korrespondiert demnach mit der quantenmechanisch berechenbaren Elektronendichteverteilung der jeweiligen Atomsorte. Analog können insbesondere in sphärischen Molekülen die Streubeiträge einzelner Atome zu einer Molekülstreuung zusammengefasst werden, um intermolekulare Strukturen zu untersuchen.

Die Beugung an einem optischen Gitters mit äquidistanten Strichen oder entsprechend einem Kristalls mit einer periodischen Anordnung von Atomen ergibt ein Beugungsmuster mit scharfen Maxima bzw. Reflexen. Amorphe Substanzen wie Flüssigkeiten, überkritische Systeme und unterkühlte Schmelzen (Gläser) verfügen im Gegensatz dazu nicht über eine periodische atomare Struktur (Fernordnung), sondern nur über eine definierte Umgebung in wenigen Koordinationssphären eines Referenzatoms (Nahordung). Eine solche Umgebung wird durch eine Atomzentren-Paarverteilungsfunktion beschrieben:

$$p_{ij}\left(\vec{r}\right) = \frac{1}{N_j} \int\limits_{V^*} \rho_i\left(\vec{r} + \vec{r}^*\right) \rho_j\left(\vec{r}^*\right) d\vec{r}^* - \delta\left(\vec{r}\right) \delta_{ij} \tag{2.2}$$

Darin sind $\rho_i\left(\vec{r}^*\right)$ die Anzahldichte der Atomsorte i am Ort $\vec{r}^*$, N die Zahl der Atome im Volumen V^* und $\delta\left(\vec{r}\right)$ die Deltafunktion, die mit dem Kroneckersymbol δ_{ij} das Referenzatom ausschließt. Die Verteilungsfunktion konvergiert für große Werte von r gegen die mittlere Teilchendichte

$$\lim_{r \to \infty} p_{ij} = \bar{\rho}_i. \tag{2.3}$$

Die auf eine stöchiometrische Einheit bezogenen Größen lauten

$$\bar{\rho} = \frac{\rho N_A}{M} \tag{2.4}$$

mit der Massendichte ρ, der Molmasse M und der Avogadro-Konstante N_A. Man verwendet daher häufig die Atompaarkorrelationsfunktion mit $g_{ij}\left(r\right) = p_{ij}\left(r\right) / \bar{\rho}$, welche für große Atomabstände gegen den Wert 1 konvergiert. Beide Verteilungsfunktionen enthalten Informationen über die wahrscheinlichsten Atomabstände in einer amorphen Probe. Das Integral über das Volumen der Kugelschalen der Atompaarkorrelationsfunktion in einem Intervall ist gleich der Anzahl der Atome in dieser Kugelsphäre, also der Koordinationszahl.

Der Zusammenhang zwischen korrigierter gemessener Intensität $I\left(\kappa\right)$, Atomstreuung und den Verteilungsfunktionen wird durch die Gesamtkorrelationsfunktion

$$G\left(r\right) = \frac{\sum\limits_{i,j} n_i n_j \bar{z}_i \bar{z}_j g_{ij}\left(r\right)}{\left(\sum\limits_i n_i Z_i\right)^2} = 1 + \frac{1}{2\pi^2 r \bar{\rho} \left(\sum\limits_i n_i Z_i\right)^2} \int\limits_0^\infty \kappa \cdot \underbrace{\frac{I\left(\kappa\right) - I_{atom}\left(\kappa\right)}{f_e^2\left(\kappa\right)}}_{I_{red}} \sin\left(\kappa r\right) d\kappa \tag{2.5}$$

beschrieben mit folgenden Größen:

I_{atom}	Atomstreuung	I_{red}	reduzierte Intensität
f_0	Atomformfaktor	f_e	mittlerer Atomformfaktor
Z	Elektronenzahl und	$\bar{z}$	mittlere effektive Elektronenzahl
n	Stoffmenge		

Der mittlere Atomformfaktor wird berechnet mit

$$f_e(\kappa) = \frac{\sum_i n_i f_i(\kappa)}{\sum_i n_i Z_i} \tag{2.6}$$

und die (mittlere) effektive Elektronenzahl mit

$$z_i(\kappa) = \frac{f_i(\kappa)}{f_e(\kappa)} \text{ und } \bar{z}_i = \frac{1}{\kappa_{max}} \int_0^{\kappa_{max}} z_i(\kappa)\, d\kappa \approx Z_i \tag{2.7}$$

Die mehrseitige Herleitung von Formel 2.5 wurde in der Literatur [3]-[11] und der vorangegangenen Diplomarbeit [12] vielfach beschrieben und soll hier nicht wiederholt werden. Die Formel zeigt, dass die gemessene Intensität über eine Fouriertransformation mit der Verteilungsfunktion verknüpft ist. Im Gegensatz zu Gleichung 2.2 enthält sie keine vektoriellen, sondern nur noch skalare Größen, da aus einer skalaren Messung $I(2\theta)$ auch nur eine skalare Verteilungsfunktion $g(r)$ berechenbar ist. Winkelabhängige Verteilungsgrößen sind demnach nur indirekt zugänglich. Man erkennt weiterhin, dass nicht einzelne Atompaarkorrelationsfunktionen bestimmt werden können, sondern nur deren gewichtete Summe, die Gesamtkorrelationsfunktion.

Es ist daher sinnvoll, Ergebnisse der Röntgenbeugungsexperimente mit Daten aus Neutronenbeugungsmessungen zu kombinieren, in welche die Einzelbeiträge mit anderen Wichtungsfaktoren eingehen, um die aussagekräftigeren Atompaarkorrelationsfunktionen zu berechnen. Analog ist es möglich, Mischungen von Komponenten mit ähnlicher Struktur einzusetzen und die Mischungsverhältnisse zu variieren. Die anomale Röntgenbeugung nutzt in ähnlicher Weise die Energieabhängigkeit des Atomformfaktors zur Gewinnung zusätzlicher Informationen. Stehen diese Methoden nicht zur Verfügung, so können nur in denjenigen Bereichen zuverlässige Aussagen gemacht werden, in denen man annehmen kann, dass nur eine Atompaarkorrelationsfunktion zur Gesamtkorrelationsfunktion beiträgt oder weitere Beiträge numerisch eliminiert werden können.

2.2 Java

Java ist eine verhältnismäßig neue Programmiersprache, die zunächst unter dem Namen Oak in einer Gruppe um James Gosling und Bill Joy bei der Firma Sun entwickelt wurde. Die endgültige Sprachspezifikation wurde 1996 veröffentlicht [13] und später durch die Spezifikation innerer Klassen [14], die auch kleinere Änderungen zur Syntax anderer Sprachelemente

enthält, ergänzt. Aus einer Publikation von Sun [15] stammt die folgende Beschreibung, die für Java häufig zitiert wird:

> Java ist eine einfache, objektorientierte, interpretierte, robuste, sichere, architekturneutrale, portable, hochleistungsfähige, parallele (Multithread-fähige), verteilte und dynamische Sprache.

Einige wichtige Eigenschaften, die für die vorliegende Arbeit von Bedeutung sind, werden im Folgenden vorgestellt. Zu einer ausführlichen Beschreibung dieser Schlagworte siehe „Java in a Nutshell“ [16]. Eine umfassende Einführung in die Programmiersprache und die Verwendung ihrer Klassenbibliothek bietet das Java-Tutorial von Sun [17] - [19].

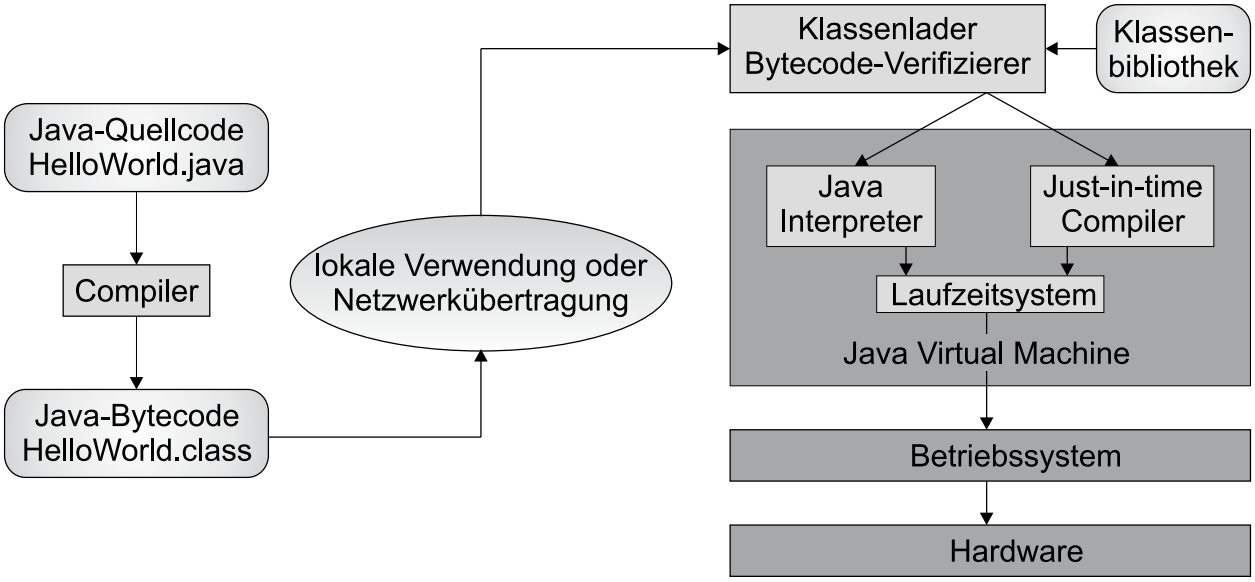


Abbildung 2.3: Übersetzung und Ausführung von Java-Programmen

Java hat sich innerhalb kürzester Zeit zur bedeutendsten Programmiersprache im Internet entwickelt. Die Hauptursache dafür ist ihre Plattformunabhängigkeit, d. h. Java-Programme sind auf jedem beliebigen Betriebssystem wie UNIX, LINUX, Windows, MacOS u. a. ohne Änderung lauffähig. Die Anwendung von Java ist jedoch keineswegs auf das Internet beschränkt, sondern eignet sich grundsätzlich für die meisten Anwendungsgebiete, in denen auf einer Vielzahl unterschiedlicher Rechnersysteme gearbeitet wird. Die Plattformunabhängigkeit beruht darauf, dass Java sowohl eine compilierte als auch eine interpretierte Sprache ist. Der Weg vom Quellcode zur Programmausführung ist in Abbildung 2.3 dargestellt. Der Quellcode, der sich konventionell in Dateien mit der Endung `.java` befindet, wird von einem Compiler in Java-Bytecode übersetzt, der in Klassendateien mit der konventionellen Dateiendung `.class` geschrieben wird. Es handelt sich dabei weiterhin um plattformunabhängigen Code. Der Bytecode wird lokal verwendet oder über ein Netzwerk übertragen. Der Klassenlader lädt die Klassen und überprüft sie auf ihre Korrektheit. Zusätzlich werden die benötigten Klassen der Java-Klassenbibliothek geladen. Ein als Java Virtual Machine

(JVM) bezeichnetes System interpretiert den Bytecode. Zusätzlich zur Interpretation kann ein Just-in-time-Compiler (JIT) zur Laufzeit bestriebssystem-spezifischen Maschinencode erzeugen, um bei mehrfach auszuführendem Programmtext Geschwindigkeitsvorteile zu erzielen. Das Laufzeitsystem kommuniziert mit dem Betriebssystem, über welches letztlich ein Zugriff auf die Hardware erfolgt. Bei geringer Auslastung führt es eine automatische Speicherbereinigung (Garbage Collection) durch, welche den Speicher nicht mehr benötigter Objekte (s. u.) automatisch wieder freigibt und erledigt damit eine Aufgabe, die in anderen Programmiersprachen häufig zu gefürchteten Speicherlecks führt. Die JVM liegt meist als Software vor, kann aber auch als Hardware fest in Geräte integriert werden. Insbesondere enthalten javafähige Internetbrowser eine JVM. Die JVM bildet zusammen mit der Klassenbibliothek (Application Programming Interface, API) die in Abbildung 2.4[2] dargestellte Java-Plattform.

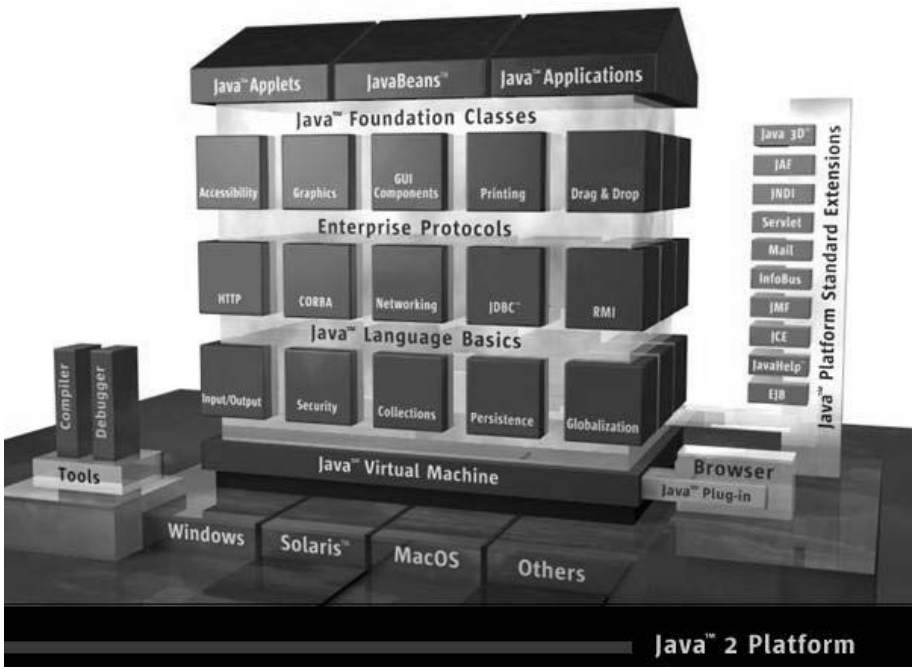


Abbildung 2.4: Die Java 2 Plattform

Das API besteht aus einer mit fast 2000 öffentlichen Klassen umfangreichen Klassenbibliothek, die fester Bestandteil von Java ist. Da mit reinem Java kein unmittelbarer Zugriff auf das verwendete Betriebssystem möglich ist, stellt sie maschinenspezifische (native) Methoden zur Verfügung, mit denen dies erreicht werden kann und die überwiegend in C geschrieben sind. Dazu gehören Methoden zum Zugriff auf das Dateisystem und Netzwerke, plattformunabhängiges Drucken und die Ansteuerung der Bildschirmausgabe. Daneben enthält sie aber auch zahlreiche Lösungen für häufig anfallende Programmieraufgaben, z. B. Datenstrukturen wie Bäume, Mengen, Listen, Vektoren und Hash-Funktionen (Hashtables),

[2]Diese Abbildung wurde einer Dokumentation auf den Internetseiten der Firma Sun unter `http://java.sun.com/` entnommen

numerische Methoden, Elemente graphischer Benutzeroberflächen, Klassen zur Ausnahmebehandlung, Threads (Unterprozesse), Datenbankzugriffe, Archivierungswerkzeuge, RMI (Remote Method Invocation), u. v. a.

Java ähnelt syntaktisch stark der Sprache C und ist ebenso wie C++ objektorientiert, wobei sich jedoch die objektorientierten Elemente der Sprache deutlich von denjenigen in C++ unterscheiden. In Java wurde die Zahl der Sprachelemente auf einen grundlegenden und zugleich effizienten Basissatz beschränkt. Insbesondere wurden die schwerwiegenden Sicherheitsprobleme bei der Speicherverwaltung von C/C++ vermieden, indem z. B. Zeiger durch Referenzen ersetzt wurden und Felder (Arrays) und Zeichenketten (Strings) durch eigene Sprachelemente repräsentiert werden, die einen Zugriff nur auf den dimensionierten Bereich erlauben.

Java besitzt ein Format zur Komprimierung und Archivierung (Java Archive, JAR), mit dem die für ein Programm erforderlichen Klassendateien, Daten, Bilder u. a. zu einer `.jar`-Datei zusammengefasst werden können, welche von der JVM direkt interpretiert werden kann.

Mit `javadoc` steht ein Dokumentationswerkzeug zur Verfügung, welches spezielle Kommentare vor Klassen, Feldern, Methoden und Konstruktoren auswertet und für komplette Programmpakete automatisch eine Klassendokumentation erstellt, in welcher auch strukturelle Zusammenhänge zwischen Paketen, Klassen, Methoden usw. dargestellt werden. Diese Dokumentation wird in der ebenfalls plattformunabhängigen Skriptsprache HTML (Hypertext Markup Language) erstellt.

Während sich die Sprachdefinition seit ihrer Einführung (mit Ausnahme der in der Version 1.1. neu hinzugekommenen inneren Klassen) nicht verändert hat, wurden seither die internen Implementierungen der JVM verbessert. Insbesondere mit der Entwicklung neuer JIT-Kompiler wurde die Geschwindigkeit der Ausführung von Java-Programmen drastisch gegenüber früheren Versionen erhöht. Auch die Klassenbibliothek wurde stark erweitert und verbessert. Dies betrifft besonders die in der Version 1.1 vollständig veränderte Ereignisbehandlung und die Gestaltung graphischer Benutzeroberflächen (Graphical User Interface, GUI). Hier löst mit der Version 1.2 ein als Swing bezeichnetes Paket die alten AWT-Komponenten (Abstract Windowing Toolkit) ab. Sun bezeichnet mit dieser Anfang 1999 eingeführten und in Java 2 Platform umbenannten Version die Entwicklung der Kernklassen als abgeschlossen. Die hier vorgestellten eigenen Klassen befinden sich vollständig auf dem Stand der Java 2 Platform in der Version 1.3.

2.3 Objektorientierung

2.3.1 Motivation

Da die bisher im Arbeitskreis verwendeten Programme überwiegend in Fortran 77 geschrieben sind und somit auf prozeduralen Techniken beruhen, sollen hier einige zentrale Begriffe und Konzepte der Objektorientierung an Beispielen dargestellt werden. Vertiefungen finden

sich beispielsweise in grundlegenden Werken von Grady Booch [20] und Timothy Budd [21]. Die Erläuterungen gelten von den gekennzeichneten Ausnahmen abgesehen für die meisten objektorientierten Programmiersprachen. Zur Beschreibung objektorientierter Programme hat sich in den letzten Jahren die Unified Modeling Language UML [22] durchgesetzt. UML ist eine Metasprache, die jenseits einzelner Programmiersprachen eine Modellierung der Struktur und des Verhaltens objektorientierter Systeme erlaubt. Diese Arbeit macht von UML Gebrauch und führt wichtige Sprachelemente mit den Beispielen zur Objektientierung ein.

Die objektorientierte Programmiertechnik entstand als Ansatz zur Lösung des Problems, dass sich parallel zur Entwicklung um Größenordnungen schnellerer Rechner zugleich Umfang und Komplexität der darauf laufenden Programme enorm erhöhte. Diese Entwicklung war unter anderem mit dem Übergang von menügesteuerten Programmen zu graphischen Benutzeroberflächen verbunden, die ein intuitives Arbeiten ermöglichen. Solche Programme bringen einen erheblichen internen Verwaltungsaufwand mit sich, der vor dem Anwender jedoch verborgen bleiben soll. Standen am Anfang der Entwicklung meist zahlreiche spezialisierte Programme für einzelne Arbeitsschritte, so umfassen graphische Benutzeroberflächen normalerweise sehr viele zusammengehörige Teile eines Prozesses, die aufeinander abgestimmt werden müssen und dennoch flexibel bleiben sollen. Gewünscht ist auch die gleichzeitige Bearbeitung mehrerer Dokumente, Datensätze oder Projekte. Während bei traditionellen Programmen eine Gliederung in Eingabe, Bearbeitung und Ausgabe die Regel war, wird mit modernen Programmen interaktiv gearbeitet. Dies bedeutet, dass dem Benutzer zu jedem Zeitpunkt der Programmausführung eine aktuelle Darstellung der internen Daten zur Verfügung steht, an denen er schrittweise Veränderungen vornimmt. Damit erhöhte sich zugleich die Laufzeit der auszuführenden Programme, und es wurde unerlässlich, dass die verwendeten Programmiersprachen geeignete Techniken zur Fehler- und Ausnahmebehandlung unterstützen, die einen Programmabsturz durch Rechenfehler, eine Fehlbedienung durch den Benutzer oder unerwartete Zustände wie Netzwerk- oder Ein/Ausgabe-Fehler verhindern. Hierbei ist zu berücksichtigen, dass solche Fehler häufig nicht dort behoben werden können, wo sie auftreten, sondern über zahlreiche Programmteile weitergeleitet werden müssen, ohne dass diese starr miteinander verbunden sind. Darin liegt eine der größten Herausforderungen komplexer Anwendungen, die einerseits aus unabhängigen und wiederverwendbaren Modulen bestehen sollen, andererseits jedoch optimal zusammenarbeiten sollen.

Es zeigte sich, dass die Verwendung herkömmlicher prozeduraler Techniken in umfangreichen Projekten zu unzuverlässig arbeitenden, schwer wart- und erweiterbaren Programmen führte, da die eingeschränkten Strukturierungsmöglichkeiten dieser Techniken nicht ausreichen, um die mit der Komplexität der Probleme überproportional ansteigende Zahl von Verknüpfungen innerhalb eines Programms bewältigen zu können. Für die Lösung anspruchsvoller numerischer Probleme sind objektorientierte Techniken hingegen nicht unbedingt erforderlich, da diese häufig eine lineare Struktur aufweisen. Eine ausführliche Diskussion zu dieser unter dem Namen Softwarekrise bekannten Problematik findet sich in Ian Sommervilles Buch „Software Engineering“ [23]. Objektorientierte Programmierung hat sich in den letzten Jahren insbesondere auch im Bereich der Oberflächenprogrammierung als eine Standardtechnik für die Entwicklung umfangreicher und komplexer Projekte eta-

bliert. Die Programmiersprache C++ verdankt diesem Umstand ihre große Verbreitung und wirtschaftliche Bedeutung.

Prozedurale Programme werden dadurch strukturiert, dass speziellere Aufgaben an Unterroutinen delegiert werden. Die Entscheidung über die Aufgabenverteilung wird sowohl in der Programmstruktur als auch zur Laufzeit von der aufrufenden Routine getroffen. Dazu muss die aufrufende Routine über Daten verfügen, welche als Entscheidungskriterium dienen und als Parameter für Fallunterscheidungen verwendet werden. Weil Daten, Kontrollstrukturen und auszuführende Berechnungen nicht klar getrennt sind, sondern über mehrere Programmteile verteilt sind, bleiben verschiedene Programmteile relativ eng aneinander gekoppelt. Die Struktur prozeduraler Programme folgt aus der Fragestellung, welche Berechnungen in welcher *zeitlichen* Abfolge durchgeführt werden müssen und bringt mit sich, dass Entscheidungsstrukturen stets in Richtung aufrufender Prozeduren verlagert werden. Dieses Konzept widerspricht einer *logischen* Gliederung, die ein Programm entsprechend unterschiedlicher Zuständigkeiten aufteilt. Dieses logische Konzept wird vom objektorientierten Ansatz verfolgt.

Nachfolgend wird am Beispiel der Simulation einer Dynamik von miteinander in Wechselwirkung stehenden Massepunkten (Himmelskörper, Moleküle, Ionen usw.) erläutert, welche Strukturierungsmöglichkeiten objektorientierte Programmtechniken bieten.

2.3.2 Klassen und Objekte

Der objektorientierte Ansatz besteht darin, dass man versucht, reale Systeme oder Prozesse durch Klassen nachzubilden. Klassen bilden eigenständige Einheiten, die unabhängig voneinander entwickelt, verändert und verwendet werden können. Eine Klasse definiert den Zustand eines Systems in Form von Attributen (auch: Instanzvariablen) und seine möglichen Veränderungen in Form von Operationen (auch: Methoden).

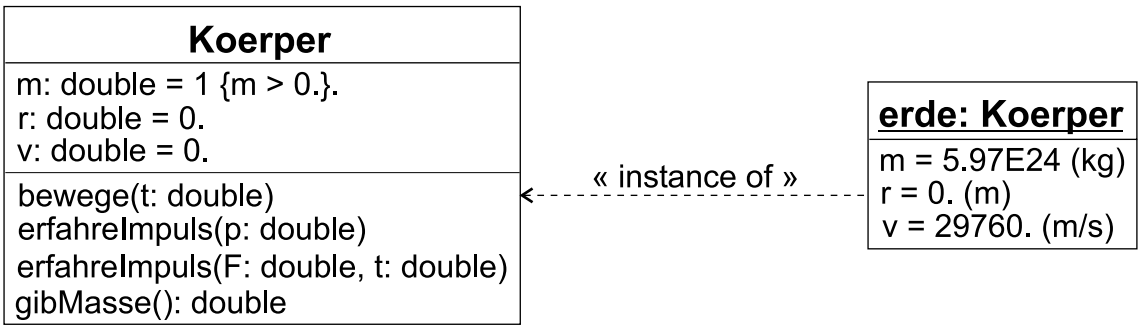


Abbildung 2.5: Die Klasse `Koerper` und eine Instanz `erde` dieser Klasse

Im genannten Beispiel einer Dynamik lässt sich leicht die logische Einheit *Körper* identifizieren und als Klasse `Koerper` formulieren. Die linke Seite in Abbildung 2.5 zeigt eine Darstellung dieser Klasse in der UML-Notation. Ein Körper wird für eine Massenpunktdynamik durch Angabe der Attribute Masse `m`, Ort `r` und Geschwindigkeit `v` vollständig

beschrieben. Bei diesen Attributen handele es sich zunächst[3] um numerische Variablen doppelter Genauigkeit vom Typ `double`. Mögliche Operationen, die dieser Klasse zugeordnet werden können, sind eine Änderung der Position aufgrund der Geschwindigkeit nach einer Zeit `t` durch die Methode `bewege(t)`, eine Impulsänderung durch Übertragung eines Impulses p in der Methode `erfahreImpuls(double p)` oder einen Kraftstoß $p = \int F dt$ in der Methode `erfahreImpuls(double F, double t)` und Zugriffsmethoden auf Attribute wie z. B. die Methode `double gibMasse()`, welche den Wert des Attributs `Masse` zurückgibt. Methoden können ebenso wie Funktionen Argumente (Parameter) und einen Rückgabewert haben. Der Sonderfall eines leeren Rückgabewertes wird durch das Schlüsselwort `void` gekennzeichnet. In der UML-Notation wird der Typ nach der Variablen bzw. der Methode durch einen Doppelpunkt abgetrennt angegeben, während die Java-Syntax den Typ vor der Bezeichnung der Variable bzw. Methode angibt. Die beiden Schreibweisen

UML: `erfahreImpuls(p: double): void` und
Java: `void erfahreImpuls(double p)`

sind also gleichbedeutend und bezeichnen eine Methode mit einem Parameter vom Typ `double` und einem Rückgabewert vom Typ `void`. Während in Java die Angabe des Rückgabetyps zwingend ist, wird ein leerer Rückgabetyp bei UML üblicherweise weggelassen. Methoden können überladen werden, d. h. mehrere Methoden wie z. B. `erfahreImpuls` haben die gleiche Bezeichnung, unterscheiden sich jedoch in der Anzahl und/oder dem Typ der Parameter. Der Methodenname und die Abfolge der Parametertypen werden zusammengenommen als Signatur bezeichnet, welche eine Methode eindeutig und vollständig beschreibt. Die Namen der Parameter spielen in dieser Hinsicht keine Rolle.

In einem Java-Programm formuliert man eine Klasse folgendermaßen:

```
public class Koerper
{
   private double m, r, v;

   public Koerper(double m, double r, double v)
   {
      if (m <= 0.)
      {
         throw new IllegalArgumentException("Die Masse muss größer als null sein!");
      }
      this.m = m;
      this.r = r;
      this.v = v;
   }

   public bewege(double t)
   {
      r += v*t;
   }
```

[3] siehe Abschnitt 2.3.4

```
   public double gibMasse()
   {
      return m;
   }

   ...
}
```

Darin ist `this` ist eine Referenz (s. u.) auf die aktuelle Klasse und dient hier nur zur Unterscheidung der Instanzvariable `this.m` vom übergebenen Parameter `m`.

Der zentrale Gegenbegriff zur Klasse ist das Objekt. Während Klassen die abstrakte Beschreibung eines Systems sind, welche als Programmtext kodiert wird, sind Objekte konkrete Repräsentanten möglicher Systemzustände, wie sie zur Laufzeit[4] im Speicher liegen. Man sagt, Objekte seien „Instanzen“ ihrer Klassen. Beispielsweise ist das Objekt `erde` eine Instanz der Klasse `Koerper`. Objektnamen werden im Gegensatz zu Klassennamen konventionell klein geschrieben. Die in einer Klasse definierten Attribute haben für ein Objekt bestimmte Werte. Das Objekt `erde` unterscheidet sich beispielsweise von anderen Objekten wie `sonne` oder `mond` nicht in der abstrakten logischen Struktur, sondern nur in den konkreten Werten wie z. B. der Masse. Die in der Abbildung 2.5 für die Klasse angegebenen Werte sind Vorgabewerte und $m > 0$ ist eine Bedingung (Zusicherung), die zu jedem Zeitpunkt erfüllt sein muss. Diese wird durch Methoden kontrolliert, welche den Zugriff auf das jeweilige Attribut regeln. Im obigen Beispiel wird für diesen Fall mit `throw` eine Fehlerbehandlung eingeleitet, die von aufrufenden Methoden berücksichtigt werden kann.

Um mit einem Objekt arbeiten zu können, muss es zunächst erzeugt werden, indem es als Instanz einer Klasse definiert wird und der für das Objekt erforderliche Speicherplatz angefordert wird. Beides geschieht durch Aufruf eines Konstruktors mit dem Operator `new`. Ein Konstruktor initialisiert ein Objekt mit einem definierten Ausgangszustand. So erzeugt der Aufruf

```
Koerper erde = new Koerper(5.98E24, 0., 29760.);
```

den Körper `erde` durch den Aufruf des zur Klasse `Koerper` gehörigen Konstruktors `Koerper(double m, double r, double v)`. Formal sind Konstruktoren Methoden mit dem Namen der Klasse ohne Rückgabewert — auch nicht `void`. Im Beispiel der Klasse `Koerper` wird ein Fehlerobjekt der Klasse `IllegalArgumentException` erzeugt, wenn versucht wird, ein Objekt mit einer nicht-positiven Masse zu initialisieren. Dieses Fehlerobjekt wird über den hier nicht näher zu beschreibenden Fehlerbehandlungsmechanismus an die aufrufende Methode übergeben und kann dort ausgewertet werden.

Die Schlüsselwörter `public` und `private`[5] regeln die Zugriffsmöglichkeiten auf Instanzelemente wie Konstruktoren, Variablen und Methoden. Es bedeutet `public`, dass auf das entsprechende Element frei zugegriffen werden kann und `private`, dass auf dieses Element

[4] und ggf. darüber hinaus in Datenbanken oder in anderer Form

[5] In Java gibt es zusätzlich das Schlüsselwort `protected` und den Standardzugriffsmodifikator, der automatisch verwendet wird, wenn kein Schlüsselwort angegeben wird.

nur durch andere Elemente derselben Klasse zugegriffen werden darf. Dieses System ist vergleichbar mit den Zugriffsrechten auf Dateien. Ziel ist es, den Zugriff auf die Instanzelemente zu beschränken und zu kontrollieren. Man sagt, die öffentlichen (`public`) Elemente einer Klasse bilden die nach außen sichtbare Schnittstelle, während private Elemente gekapselt bzw. versteckt werden. Häufig deklariert man alle Variablen als `private`, während eine Vielzahl von Methoden `public` ist. Im obigen Beispiel soll die Masse eines Körpers nach dessen Initialisierung nicht mehr verändert werden können. Die Werte für Ort und Geschwindigkeit dürfen ebenfalls nicht direkt geändert werden, sondern nur durch die Übertragung eines Impulses.

Abbildung 2.6: Konzept der Kapselung

Der wichtige Begriff der Kapselung lässt sich anschaulich durch Abbildung 2.6[6] erklären. Für den Besitzer einer Uhr ist es zunächst nur wichtig, dass er auf ihr die Zeit ablesen und sie gegebenenfalls verstellen kann, ohne dass er sich um deren komplizierte Konstruktion kümmern muss. Mehr noch: Die empfindliche Mechanik muss vor unzweckmäßigen Eingriffen geschützt werden. Würde man das technische System *Uhr* als eine Klasse programmieren, so würden im Wesentlichen folgende Zugriffsmethoden ausreichen, um mit der Uhr arbeiten zu können: `String gibUhrzeit()` und `void verstelleUhr(double zeitdifferenz)`. Kapselung bedeutet nun folgendes:

1. Hinter einer Oberfläche bzw. wenigen öffentlichen Methoden kann sich ein komplexes Innenleben verbergen, das vom Uhrmacher oder Programmierer konstruiert wird. Diese interne Struktur spielt für den Benutzer der Uhr bzw. der Klasse später jedoch keine Rolle mehr, solange die impliziten oder expliziten Randbedingungen (Uhr geht richtig, Klasse zeigt erwartetes Verhalten) eingehalten werden. Man bezeichnet dies als *Verbergen von Informationen (Information hiding).*

[6]Diese Abbildung wurde einer Anzeige der Firma Lange & Söhne entnommen

2. Ein Zugriff auf die Instanzvariablen ist meist nur durch Methoden möglich. Diese Methoden sollten die Konsistenz der Variablen sichern. Beispielsweise ist es nicht möglich, die Position des großen Uhrzeigers zu verändern, ohne die sich dadurch ergebende Positionsänderung des kleinen Zeigers zu verhindern.

3. Die Änderung von Variablen durch Methoden kann protokolliert werden, gezielt rückgängig gemacht werden[7] oder Folgeprozesse auslösen. So führt z. B. ein Uhrzeitwechsel in definierter Weise zu einem Datumswechsel.

In Java besteht die Möglichkeit, zusammengehörige Klassen zu Paketen zusammenzufassen, die in einem Dateisystem ihre Entsprechung in Ordnern finden. Diese Entsprechung gilt auch dahingehend, dass sich Klassendateien in Dateisystem-Ordnern mit Paketnamen befinden und diese Ordner bzw. Pakete ineinander verschachtelt werden dürfen. Für dieses Pakete existieren analoge Zugriffsmodifikatoren, die den Zugriff auf einzelne Klassen regeln, deren Elemente wie oben beschrieben kontrolliert werden. Pakete dienen weiterhin der Definition von Namensräumen, d. h. in unterschiedlichen Paketen dürfen gleichnamige Klassen existieren, die durch Angabe einer qualifizierenden Bezeichnung, welche die Paketnamen und den Klassennamen enthält, adressiert werden können. Wird kein Paketname angegeben, so arbeitet man entweder mit dem namenlosen Standardpaket des Programmierers oder mit dem Java-Standardpaket `java.lang`.

Der Zugriff auf Instanzelemente wie Variablen und Methoden erfolgt innerhalb einer Klasse durch Angabe der Elementbezeichnung wie z. B. `r += v*t` in der Methode `bewege(...)`, welche den Wert von `r` um `v*t` inkrementiert. Der Zugriff auf ein Element außerhalb dessen Klasse erfordert die Angabe des entsprechenden Objekts z. B. `erde.bewege(1.)`.

Von den Instanzelementen zu unterscheiden sind die Klassenelemente, welche nicht einem einzelnen Objekt zugeordnet sind, sondern der Klasse an sich. Ein Beispiel dafür ist die Konstante `PI` in der Klasse `java.lang.Math`, die im Speicher sinnvollerweise nur einmal existiert. Klassenelemente werden in Java mit dem Schlüsselwort `static` gekennzeichnet. Sie werden nicht mit dem Namen eines Objekts, sondern mit demjenigen der Klasse verwendet, z. B. `Math.PI`. In vielen Fällen handelt es sich bei statischen Variablen zugleich um Konstanten, die durch das Schlüsselwort `final` gekennzeichnet sind. Statische Methoden und Konstruktoren sind Klassenelemente, die nur auf statische Variablen zugreifen dürfen. Statische Methoden sind Analoga zu Funktionen in prozeduralen Programmiersprachen. Eine Zuordnung zu Klassen dient dann nur einer Gruppierung in Sachverhalte, z. B. befinden sich in der Klasse `Math` statische numerische Methoden wie `static double sin(double x)`.

Mit den bisherigen Ausführungen lässt sich das „Hello world“ - Programm in Java verstehen, das in seiner jeweiligen Form als einfachstes Beispiel zur Vorstellung einer Programmiersprache dient:

```
public class HelloWorld
{
   public static void main(String[] args)
   {
      System.out.println("Hello world!");
   }
```

[7] siehe unten, undo/redo-Funktionalität

```
}
```

`HelloWorld` ist eine öffentlich zugängige Klasse im Standardpaket des Programmierers. Darin enthalten ist die öffentliche statische Methode `main(...)`. Dies ist eine Methode ohne Rückgabewert, die bei jedem Programmstart vom Betriebssystem z. B. über die Eingabeaufforderung aufgerufen wird. Ihre Argumente sind ein durch eckige Klammern `[]` gekennzeichnetes Feld von Zeichenketten `String`, wobei `String` eine im Java-Paket `java.lang` definierte Klasse ist. In diesem Paket befindet sich zugleich die Klasse `System`, welche eine statische Variable (Zugriff erfolgt über Klassenbezeichnung `System`) `out` definiert, die ihrerseits eine Instanz der Klasse `PrintStream` im Paket `java.io` darstellt. In letzterer Klasse ist die Instanzmethode (Zugriff erfolgt über Objektname `out`) `println(String str)` definiert, welche die Zeichenkette `"Hello world!"` auf der im Objekt `out` festgelegten Standardausgabe ausgibt. Die Variable `out` und die Methode `println(...)` sind dabei wieder öffentlich zugänglich.

Java kennt das sprachliche Element innerer Klassen. Das sind Klassen, die innerhalb anderer Klassen deklariert werden. Sie sind ebenso wie Instanzvariablen, Konstruktoren und Methoden Instanzelemente einer Klasse und haben einen vollständigen Zugriff auf diese Elemente. Solche Klassen können auch ohne Angabe eines Namens erzeugt werden und werden dann als anonyme innere Klassen bezeichnet.

2.3.3 Referenzierung von Objekten

Klassen und statische Felder (Arrays), also Felder mit unveränderlicher Feldgröße, zählen in Java zu den Referenzdatentypen. Im Gegensatz dazu bezeichnet man die Typen `int`, `double`, `boolean` etc. als primitive Datentypen. Übergibt man primitive Datentypen an eine Methode, so wird eine Kopie derselben auf den Stapelspeicher (Stack) gelegt, auf dem die Methode operiert. Eine Änderung der entsprechenden Variable in der aufgerufenen Methode hat keine Änderung derselben Variable in der aufrufenden Methode zur Folge. Man bezeichnet dies als „Übergabe durch Wert“ (call by value). Für zusammengesetzte Datentypen/Referenzdatentypen[8] wäre diese Vorgehensweise unökonomisch in Bezug auf Rechen- und Speicherbedarf, da diese Typen im Gegensatz zu den primitiven Datentypen häufig viel Speicherplatz benötigen. Referenzdaten werden als Referenz übergeben (call by reference). Dies bedeutet, dass die aufgerufene Methode implizit auf den selben Speicherbereich (Kellerspeicher, Heap) zugreift wie die aufrufende Methode und damit eine Änderung in beiden Zugriffsbereichen verursacht. Der mögliche Zugriff wird dabei allein durch die zu diesem Objekt gehörige Klasse bestimmt. Eine alternative Vorgehensweise ist die Übergabe von Zeigern (call by pointer), wie sie in C/C++ praktiziert wird. Zeiger enthalten explizite Angaben über den Speicherort und erlauben einen direkten und freien Zugriff auf den Speicher. Durch Zeigerarithmetik ist ein schneller und leistungsfähiger Speicherzugriff möglich, letztlich werden aber die objektorientierten Konzepte, die einen kontrollierten Zugriff einführen, außer Kraft gesetzt. 80% der in C/C++ auftretenden Fehler sind auf das Zeigerkonzept zurückzuführen. Dies ist umso problematischer, als die Fehler in vollkommen unabhängigen Programmteilen auftreten können und unter Umständen lange latent bleiben können.

[8] Prinzipiell bestehen die genannten Möglichkeiten zur Parameterübergabe unabhängig vom Konzept der Objektorientierung, sie sind jedoch für dessen Verständnis fundamental.

Dieses Konzept hat Auswirkungen auf Zuweisungen und Vergleiche. Mit dem Audruck `o = p` wird erreicht, dass o auf den gleichen Speicherbereich wie p zugreift. Wird p verändert, so ändert sich auch o. Befindet sich diese Zuweisung allerdings in einer aufgerufenen Methode, z. B.

```
void reinitialisiere(Koerper erde)
{
   erde = new Koerper(1., 0., 0.);      // Vorsicht!!!
}
```

so zeigt nur die Referenz `erde` in der aufgerufenen Methode auf einen neuen Inhalt, nicht aber in der aufrufenden Methode. Hingegen würde der Ausdruck `erde.bewege(1.)` die Position der `erde` auch in der aufrufenden Methode verändern. Um tatsächlich Kopien zu erstellen, steht in Java[9] die Methode `clone()` zur Verfügung. Diese Methode legt jedoch nicht fest, ob es sich um eine flache oder eine tiefe Kopie handelt. Flache Kopien erzeugen eine neue Referenz auf dasselbe Objekt, während tiefe Kopien den Inhalt[10] von Objekten rekursiv kopieren. Auf der anderen Seite prüft der Vergleichsoperator `==` für Objekte, ob deren Referenzen auf den gleichen Speicherbereich zeigen, während die Methode `equals(Object)` die Inhalte der Objekte vergleicht.

Zu jedem primitiven Datentyp existiert in Java ein komplexer Datentyp z. B. die Klasse `Integer` für `int`, die Klasse `Double` für `double` und die Klasse `Boolean` für `boolean`. Es handelt sich um unveränderliche Datentypen (Immutables), die mit der Initialisierung durch `new` einen konstanten Wert erhalten, der nicht mehr verändert werden kann, weil keine entsprechenden Methoden definiert sind. Dies ist nicht immer wünschenswert, siehe dazu Abschnitt 10.2.

2.3.4 Beziehungen zwischen Klassen

Wurden bisher im Wesentlichen einzelne Klassen und deren Objekte betrachtet, so sollen nun die Beziehungen zwischen verschiedenen Klassen und Objekten beschrieben werden. Man unterscheidet drei fundamentale Beziehungstypen:

1. Assoziationen bzw. Verwendungsbeziehungen
2. Aggregationen bzw. Teil-Ganzes-Beziehungen
3. Spezialisierungsbeziehungen

Assoziationen

Von einer Assoziation spricht man, wenn Objekte einer Klasse Objekte derselben oder einer anderen Klasse verwenden. Die zugrundeliegende Modellvorstellung liegt im Austausch von Botschaften zwischen Objekten.
In Abbildung 2.7 sieht man die Assoziation der Klassen `Simulation` und `Koerper`, die darin besteht, dass die Klasse `Simulation` die Klasse `Koeper` kontrolliert. Auf Objektebene

[9] in der weiter unten einzuführenden Klasse `Object`
[10] Objekte können weitere Objekte enthalten, siehe folgender Abschnitt

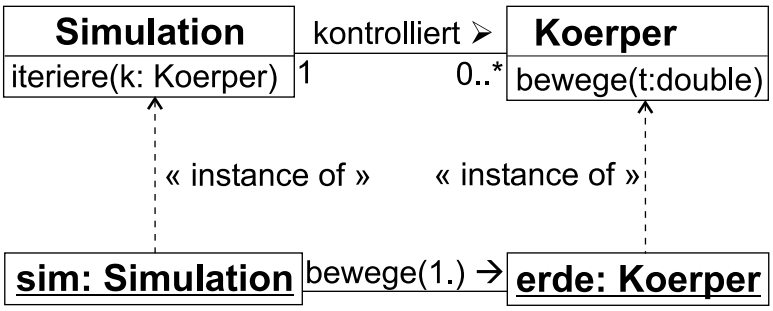


Abbildung 2.7: Beispiel einer Assoziation und einer Botschaft

betrachtet funktioniert diese Kontrolle, indem z. B. ein Objekt `sim` vom Typ `Simulation` in seiner Methode `iteriere(Koerper k)` die Methode `bewege(double t)` des bestimmten `Koerpers` `k = erde` aufruft: `erde.bewege(1.)`. Die Multiplizitäten `1` bzw. `0..*` geben an, dass genau ein Objekt der Klasse `Simulation` eine beliebige Anzahl an Objekten der Klasse `Koerper` steuern kann. Die Menge möglicher Assoziationen wird zum einen dadurch bestimmt, von welchen anderen Objekten ein Objekt Kenntnis hat und zum anderen, welche Methoden für ein Objekt aufgerufen werden dürfen. So kann in einem Programm zwar ein Zugriff eines Objektes auf ein anderes durch Zugriffsmodifikatoren prinzipiell erlaubt sein, kann aber nicht stattfinden, weil ein Objekt keine Kenntnis von dem anderen hat. Ein wesentlicher Teil objektorientierten Entwurfs besteht in der Konstruktion solcher Klassen- und Objektbeziehungen.

Aggregationen und Kompositionen

Eine Aggregation liegt dann vor, wenn eine Klasse Teil einer anderen Klasse ist. Im obigen Beispiel wäre es wünschenswert, den Ort und die Geschwindigkeit eines Körpers nicht als skalare Größe vom Typ `double`, sondern als vektorielle Größe anzugeben. Dies lässt sich dadurch erreichen, dass man mehrere skalare Größen zu einer neuen Klasse `Vektor` zusammenfasst:

```
public class Vektor
{
   private double x, y, z;

   public Vektor(double x, double y, double z)
   {
      this.x = x;
      this.y = y;
      this.z = z;
   }

   public Vektor addiere(Vektor v)
   {
      Vektor w = new Vektor(x, y, z);
      w.x += v.x;
      w.y += v.y;
```

```
      w.z += v.z;
      return w;
   }

   public Vektor multipliziere(double s)
   {
      Vektor w = new Vektor(x, y, z);
      w.x *= s;
      w.y *= s;
      w.z *= s;
      return w;
   }

   ...
}
```

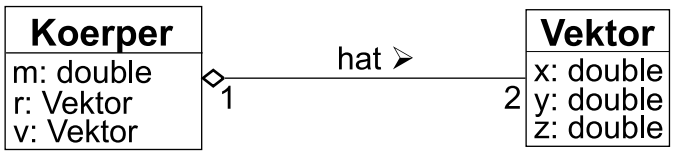


Abbildung 2.8: Beispiel einer Aggregation

In Abbildung 2.8 ist dargestellt, dass jedem Objekt der Klasse `Koerper` zwei Objekte der Klasse `Vektor` zugeordnet sind, welche den Ort und die Geschwindigkeit des Körpers kapseln. Dadurch müssen auch der Konstruktor und die Methode `bewege(double t)` der Klasse `Koerper` angepasst werden:

```
public void bewege(double t)
{
   r = r.addiere(v.multipliziere(t));
}
```

Ein Spezialfall einer Aggregation ist die Komposition. In diesem Fall sind die Komponenten in ihrer Existenz an das Vorhandensein der Kompositionsklasse gebunden. Im Beispiel von Abbildung 2.9 ist die Definition einer Klasse `Etage` nur im Zusammenhang mit der Existenz einer Klasse `Haus` sinnvoll. Die UML-Notation hebt dies gegenüber einer Aggregation durch eine gefüllte Raute hervor.

Abbildung 2.9: Beispiel einer Komposition

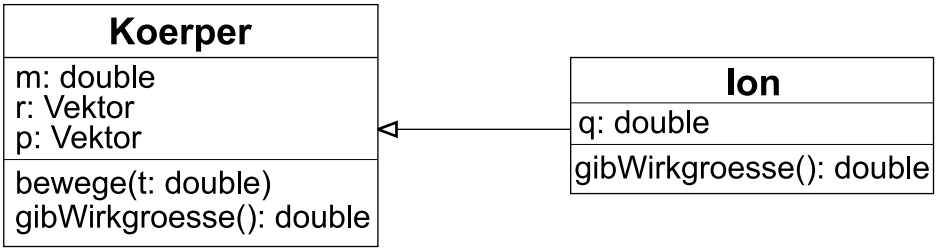


Abbildung 2.10: Beispiel einer Spezialisierung

Spezialisierungen

Spezialisierungsbeziehungen beschreiben den Zusammenhang von Klassen, die sich im Grad ihrer Abstraktion unterscheiden.
Beispielsweise lässt sich ein Ion als Spezialfall eines Körpers betrachten. Das Ion verfügt zusätzlich über eine Ladung `q` und unterscheidet sich in seiner Wechselwirkung mit anderen Ionen oder Körpern. In einem Java-Programm formuliert man das folgendermaßen:

```
public class Ion extends Koerper
{
   private double q;

   public Ion(Vektor r, Vektor p, double q)
   {
      super(r, p);
      this.q = q;
   }

   public double gibWirkgroesse()
   {
      return q;
   }
}
```

Das Schlüsselwort `extends` bedeutet, dass sich die Klasse `Ion` von der Klasse `Koerper` ableitet. Man bezeichnet dann `Koerper` als Superklasse von `Ion` und umgekehrt `Ion` als Subklasse oder abgeleitete Klasse von `Koerper`. Dieser als Vererbung bezeichneter Mechanismus bedeutet, dass die Klasse `Ion` Attribute und Operationen der Klasse `Koerper` übernimmt. Jede Instanz der Klasse `Ion` ist damit zugleich eine Instanz der Klasse `Koerper` und verhält sich entsprechend. Der Konstruktor in `Ion` erlaubt die Festlegung der Variablen `r` und `v` und setzt sie mit Hilfe des Schlüsselwortes `super` durch Aufruf eines zu implementierenden Superklassenkonstruktors `Koerper(Vektor r, Vektor v)` in der Klasse `Koerper`. Der Zugriff auf Superklassenelemente ist durch die bereits erwähnten Zugriffsmodifikatoren geregelt. Ein Aufruf wie `i.bewege(...)` für ein Ion `i` verwendet automatisch die entsprechende Methode der Klasse `Koerper`. Abgeleitete Klassen übernehmen jedoch nicht alle Eigenschaften ihrer Superklasse, sondern weisen neue Verhaltensweisen auf oder unterscheiden sich in einem prinzipiell ähnlichen Verhalten. So kann ein Ion im Gegensatz zu einem allgemein definierten Körper ein chemisches Verhalten zeigen, das nicht durch Methoden in der Klasse `Koerper`

beschrieben werden kann. Während schwere Körper durch die Gravitationskraft wechselwirken, wird die Wechselwirkung von Ionen durch deren Coulomb-Potential bestimmt. Ein Simulationsprogramm, das diesem Umstand Rechnung tragen soll, muss die jeweils zu verwendende Wirkgröße kennen, also die Masse bzw. die Ladung. Diese Größe ist über die Methode `double gibWirkgroesse()`, die für beide Klassen `Koerper` und `Ion` implementiert sei, zugänglich. Die Klasse `Ion` definiert deshalb ihre eigene Methode `gibWirkgroesse()`, die bei einem entsprechenden Aufruf `i.gibWirkgroesse()` für ein Ion `i` anstelle der gleichnamigen Methode der Klasse `Koerper` verwendet wird. Man nennt dies das Überschreiben von Methoden.

Während Java[11] für jede Klasse nur eine Superklasse erlaubt, kann jede Klasse beliebig viele Subklassen haben. Die rekursive Anwendung des Vererbungsmechanismus führt zu Klassenhierarchien mit Baumstruktur. An der Wurzel dieses Baumes steht in Java die Klasse `Object`, d. h. alle Klassen in Java leiten sich implizit von dieser Klasse ab und verwenden deren Variablen und Methoden.

2.3.5 Abstrakte Klassen und Interfaces

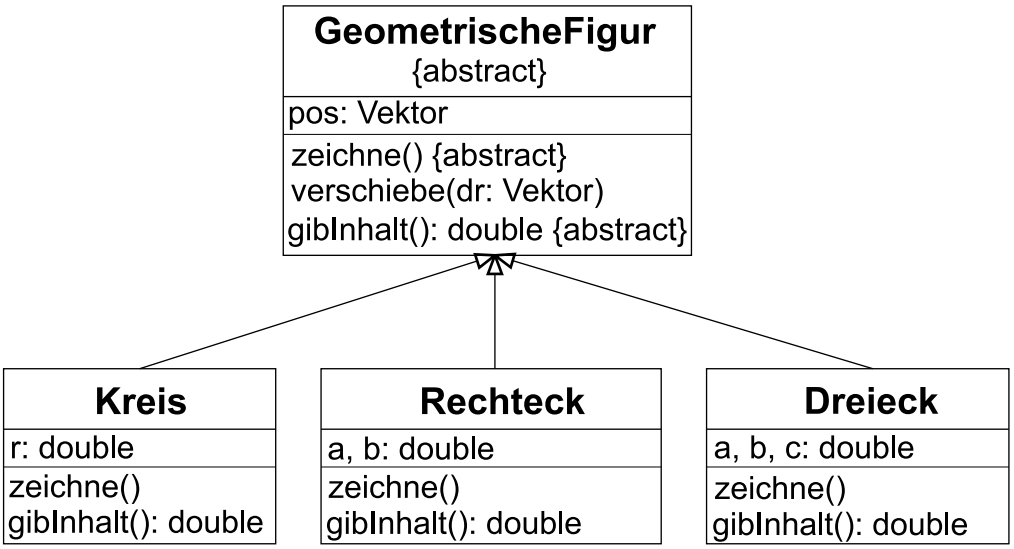


Abbildung 2.11: Abstrakte Klassen und Methoden

Gegeben sei nun das Klassendiagramm in Abbildung 2.11. Von der Klasse `GeometrischeFigur` leiten sich die Klassen `Kreis`, `Rechteck` und `Dreieck` ab. Allen Klassen gemeinsam ist die Angabe eines Ortsvektors `pos`, welcher die Lage eines Referenzpunktes beschreibt, auf dem die Methode `verschiebe(...)` operiert, die in der Klasse `GeometrischeFigur` definiert ist. Es ist weiterhin möglich, für jede geometrische Figur mit `gibInhalt()` einen Flächeninhalt zu berechnen und diese mit `zeichne()` darzustellen. Dies kann jedoch nicht in der Klasse `GeometrischeFigur` geschehen, da diese nicht weiß und nicht zu wissen braucht,

[11] manche Programmiersprachen unterstützen Mehrfachvererbung, d. h. die Existenz mehrerer Superklassen

welche Klassen sich von ihr ableiten, wie sich deren Flächeninhalt berechnet und wie sie gezeichnet werden. Man verwendet daher eine mit dem Java-Schlüsselwort `abstract`[12] gekennzeichnete Methode der Form `public void abstract zeichne()`, welche keinen Methodenkörper enthält, sondern nur die Signatur angibt, die für einen Aufruf benötigt wird. Eine Klasse, die abstrakte Methoden enthält, wird selbst als abstrakt bezeichnet: `public abstract class GeometrischeFigur`. Die so deklarierte Methode muss in einer Subklasse implementiert werden, andernfalls ist die Subklasse selbst wieder als abstrakt zu deklarieren.

Es wurde bereits ausgeführt, dass die Menge aller öffentlich zugänglichen Methoden einer Klasse deren Schnittstelle darstellt, die ihr Verhalten nach außen repräsentiert. In Java existiert ein eigenes Sprachelement für Schnittstellen, das durch das Schlüsselwort `interface` gekennzeichnet wird und ähnlich funktioniert wie abstrakte Klassen. Eine solches Interface[13] enthält ausschließlich implizit abstrakte und öffentliche Methoden sowie Konstanten. Für Interfaces gibt es einen der Vererbung entsprechenden Mechanismus. Man sagt, dass Klassen, welche für die in Interfaces definierte Methoden einen Methodenkörper bereitstellen, dieses Interface implementieren. In Java wird dafür das Schlüsselwort `implements` verwendet.

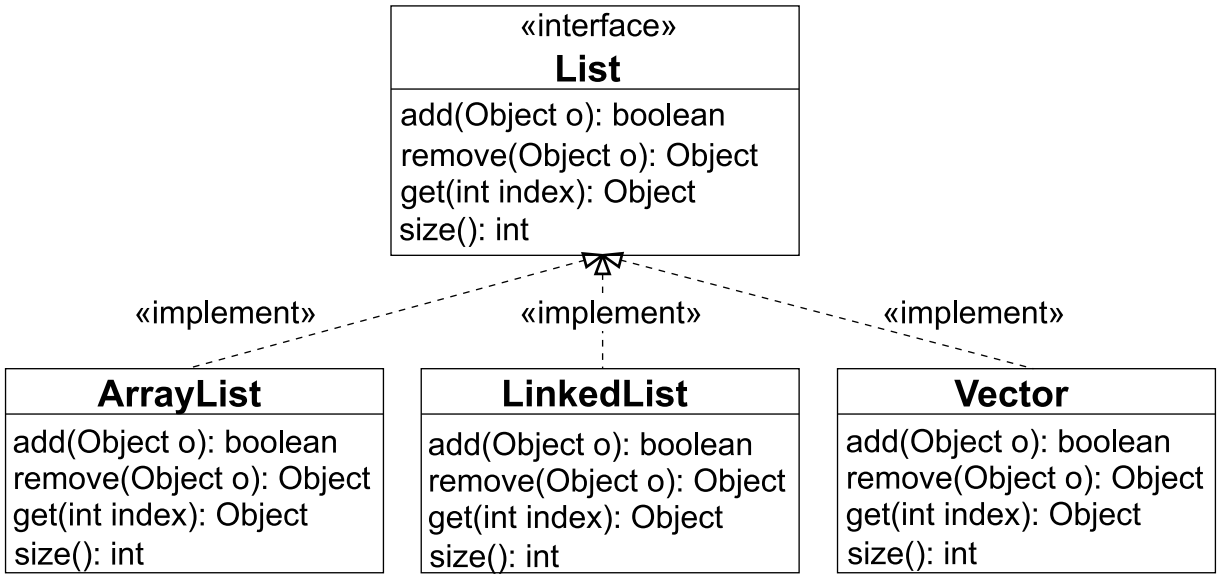


Abbildung 2.12: Implementierung des List-Interfaces

Abbildung 2.12 zeigt das im Paket `java.util` definierte Interface `List`, das im Folgenden noch häufig verwendet wird. Dieses Interface definiert zahlreiche Methoden zur Arbeit mit dynamischen[14] Feldern, von denen hier nur einige aufgeführt sind. Die Klassen `ArrayList`, `LinkedList` und `Vector` sowie die nicht eingezeichnete abstrakte Klasse `AbstractList` im-

[12] gleichbedeutende Schlüsselwörter existieren in anderen Sprachen z. B. `virtual` in C++

[13] Diese Arbeit verwendet für das Java-Sprachelement *Interface* diesen auch in deutschsprachiger Literatur üblichen Begriff in Unterscheidung zum allgemeiner verstandenen Begriff der Schnittstelle

[14] dynamisch bedeutet hier: veränderliche Feldgröße

plementieren das Interface `List`. Der Java-Quelltext für das Interface würde dann so aussehen:

```
package java.util;

public interface List
{
   boolean add(Object o);
   Object remove(Object o);
   Object get(int index);
   int size();
   ...
}
```

und für die Klasse `ArrayList`

```
package java.util;

public class ArrayList implements List
{
   private int size;
   ...

   public int size()
   {
      return size;
   }

...
}
```

Hierin sind die übrigen Methoden des Interfaces aus Platzgründen weggelassen, müssen aber zwingend im Programmtext auftauchen. Jede Klasse kann beliebige viele Interfaces implementieren, ein Interface kann unter Verwendung des Schlüsselwortes `extends` seinerseits beliebig viele Interfaces erweitern.

2.3.6 Typisierung und Polymorphie

Ein Grund für den Einsatz von Vererbungs- und Implementierungsmechanismen liegt in der Wiederverwendung von Software: Man passt existierende Klassen dadurch an eigene speziellere Bedürfnisse an. Ein weiterer Grund liegt in der Strukturierung von Klassen, die sich in Spezialisierungsgrad und Spezialisierungsmerkmal (Diskriminator) unterscheiden. Am Wichtigsten ist jedoch eine als Polymorphie bezeichnete Eigenschaft:

Jedes Objekt hat einen Typ. Mögliche Typen eines Objektes sind seine Klasse, alle Superklassen und alle implementierten Interfaces. Ein Objekt kann unter jedem dieser Typen angesprochen werden. Mit der Erzeugung eines Objektes wird dessen Klasse ein für allemal festgelegt, während sich der Typ im Laufe der Zeit durch Verwendung von Typänderungsoperatoren (Casts) oder bei der Übergabe an Methoden ändern kann. Anstelle des Ausdrucks `Kreis k = new Kreis(...)` kann man auch schreiben:

```
GeometrischeFigur gf1 = new Kreis(...);
GeometrischeFigur gf2 = new Rechteck(...);
GeometrischeFigur gf1 = new Dreieck(...);
```

Damit erzeugt man Objekte vom Typ `GeometrischeFigur` und der jeweiligen Klasse. Diese Objekte kann man nun im Feld `GeometrischeFigur[] gfArr` ablegen und mit der in der abstrakten Klasse `GeometrischeFigur` deklarierten Methode `zeichne()` verwenden:

```
for (int i=0; i<gfArr.length; i++)
{
   gfArr[i].zeichne();
}
```

Diese Prozedur ruft in einer Schleife über den Inhalt des Feldes die Operation `zeichne()` für alle enthaltenen Feldelemente auf. Der Vorteil gegenüber der Verwendung einer Klasse ist offensichtlich: Man abstrahiert auf eine gemeinsame Eigenschaft von Objekten unterschiedlicher Klassen und verwendet diese. Ebenso könnte man diese Objekte einer Methode `fuelle(GeometrischeFigur gf)` als Argument übergeben. In gewisser Hinsicht kann man demnach sagen, dass sich die Begriffe Klasse/Interface und Typ verhalten wie Konstante und Variable.

Als weiteres Beispiel diene das oben eingeführte Interface `List`. Eine Liste ist ein dynamisches Feld, welches eine beliebige Anzahl von Elementen vom Typ `Object` aufnehmen kann, wobei die Objekte innerhalb der Liste eine definierte Reihenfolge haben. Da sich alle Java-Klassen von `Object` ableiten, kann jedes beliebige Objekt in eine Liste eingeordnet werden. Um ein in einer Liste befindliches Objekt zu verwenden, kann es erforderlich sein, eine Typänderung vorzunehmen, die den Aufruf einer spezifischen Methode erlaubt:

```
List l = new ArrayList();
l.add(gf1);
l.add(gf2);
l.add(gf3);
for (Iterator iter = l.iterator(); iter.hasNext(); )
{
   ((GeometrischeFigur)(iter.next())).zeichne();
}
```

In diesem Ausdruck wandelt der Typänderungsoperator `(GeometrischeFigur)` den Rückgabetyp `Object` der Methode `Object Iterator::next()` des Interfaces `Iterator` in den Typ `GeometrischeFigur` um, für welchen die Methode `zeichne()` im Gegensatz zur Klasse `Object` definiert ist. Das Interface `Iterator` erlaubt den sequentiellen Zugriff einschließlich Änderung und Löschung auf die Elemente einer Liste.

Nun wird auch die große Bedeutung von Interfaces als Konstruktionselemente für die objektorientierte Programmierung deutlich: Es ist beispielsweise möglich, in einem Programm nahezu ausschließlich mit dem Typ des Interfaces `List` zu arbeiten. Einzig am Ort der Erzeugung eines Objektes muss die konkrete Klasse angegeben werden. Die konkreten Klassen `ArrayList` und `LinkedList` unterscheiden sich nicht in der durch `List` festgelegten Funktionalität, sondern nur in der Geschwindigkeit von Operationen. Während auf `ArrayList` indizierte Operationen mit hoher Geschwindigkeit angewandt werden können, sind

für `LinkedList` Vorgänger/Nachfolger-Operationen schneller. Allgemein kann so an einer einzigen geeigneten Stelle in einem größeren Kontext das Verhalten optimiert werden.

Sammlungsklassen

Listen sind Vertreter der Sammlungsklassen (Collections), die als Kontainer für gleichartige oder verschiedenartige Objekte dienen. Tatsächlich handelt es sich nicht um Klassen, sondern um Interfaces. Dieser feststehende Begriff wird jedoch auch in vielen anderen Programmiersprachen für entsprechende Datenstrukturen verwendet, die dort meist wie z. B. in der C++ *Standard Template Library* Klassen sind. Weitere Sammlungsklassen, die nachfolgend verwendet werden, sind die Interfaces `Set` und `Map`. Ein `Set` ist eine Menge von unterschiedlichen (in Bezug auf ihre Identität, nicht auf ihren Typ) Objekten, eine `Map` ordnet Schlüsselobjekten eindeutig[15] Wertobjekte zu. Zu jedem Interface gibt es mehrere Klassen, welche dessen Methoden implementieren. Die Implementierungen unterscheiden sich im optionalen Vorhandensein einer Reihenfolge oder im Vorliegen einer Synchronisation bei der Arbeit mit Threads. Die Interfaces `List` und `Set` leiten sich vom Interface `Collection` ab, das unter anderem grundlegende Operationen wie `add(Object o)`, `remove(Object o) boolean contains(Object o)`, `int size()` und `Iterator iterator()` definiert.

2.3.7 Entwurfsmuster

Gamma, Helm, Johnson und Vlissides untersuchten zahlreiche Softwaresysteme und fanden häufig wiederkehrende Muster kooperierender Klassen, die sie Entwurfsmuster nannten [24]. Sie nahmen eine Einteilung in 23 Entwurfsmuster vor, die ihrerseits in Erzeugungs, Struktur- und Verhaltensmuster aufgeteilt sind. Entwurfsmuster sind aus mehreren Klassen oder Interfaces zusammengesetzte Module, die ein bestimmtes Verhalten aufweisen. Betrachtet man Klassen als Zahnräder eines technischen Systems, so kann man sich Entwurfsmuster als Getriebe vorstellen, die über einen vordefinierten Bauplan verfügen und nach einer Anpassung an spezielle Aufgabenstellungen zu komplexeren Einheiten kombiniert werden können.

Wesentlicher Zweck von Entwurfsmustern ist die Umsetzung der im Motivationsabschnitt genannten Entwicklungsziele, Klassen so eng wie nötig, aber so lose wie möglich miteinander zu koppeln, um diese austauschen und wiederverwenden zu können. Bei der Entwicklung der GUI-Bibliothek Swing flossen zahlreiche Erfahrungen aus diesen Entwurfsmustern ein. Nachfolgend werden drei Beispiele erläutert, die für das Verständnis von Benutzeroberflächen von zentraler Bedeutung sind. Auf andere Entwurfsmuster, welche für diese Arbeit eine Rolle spielen, wird an der jeweiligen Stelle hingewiesen.

Dekorierer und Kompositum

Abbildung 2.13 zeigt ein Entwurfsmuster, das die Grundlage des Aufbaus aller Benutzeroberflächen unter Java bildet. Die eigentliche Konstruktion besteht nur aus den beiden Klassen `Component` und `Container` auf der linken Seite. Die rechts abgebildeten Klassen sind Beispiele für Oberflächenelemente wie Schaltflächen (`Button`), Beschriftungen (`Label`),

[15] aber nicht eineindeutig

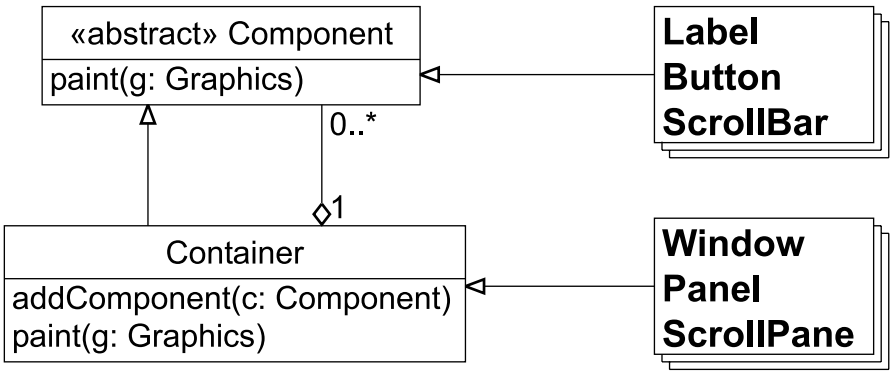


Abbildung 2.13: Entwurfsmuster Kompositum/Dekorierer

Zeichenebenen (`Panel`) und Fenster (`Window`) und sind Subklassen der zuvor genannten Klassen. Die Abbildung zeigt, dass sich die Klasse `Container` von der Klasse `Component` ableitet. Beide Klassen implementieren eine Methode `paint(Graphics g)`, mit denen sie sich selbst in einen graphischen Kontext `g` zeichnen. Gleichzeitig können Objekte der Klasse `Container` eine beliebige Anzahl von Objekten der Klasse `Component` enthalten, die durch die Methode `addComponent(Component c)` dem Kontainer hinzugefügt werden. Da sich die Klasse `Container` von der Klasse `Component` ableitet, können Komponenten rekursiv weitere Komponenten enthalten. Von beiden Klassen existieren weitere Ableitungen, auch in mehreren Ebenen. Neben der gezeigten ersten Ableitungsebene existieren weitere Ebenen, z. B. leiten sich von der Klasse `Window` weiter die Klassen `Frame` und `Dialog` ab. Durch Anwendung beider rekursiven Strukturen entstehen zwei Hierarchien: Eine Vererbungshierarchie mit der Stammklasse `Component` und eine Kontainerhierarchie.
Abbildung 2.14 zeigt die in dieser Arbeit vorzustellende Benutzeroberfläche. Sie arbeitet mit den neuen Klassen des Swing-GUI. Die Klassen `Component` und `Container` sind noch Teil der alten AWT-Benutzeroberfläche, jedoch leiten sich alle Swing-Klassen von der Klasse `JComponent` ab, die ihrerseits eine Subklasse von `Container` ist. Somit gelten die vorangegangenen Ausführungen entsprechend für die neuen Klassen. In der Abbildung erkennt man die Kontainerhierarchie: Die Klasse `JFrame` bildet den äußersten Rahmen der Anwendung. Diese Klasse enthält eine Menüleiste der Klasse `JMenuBar`, eine Schaltflächenleiste der Klasse `JToolBar` und innere Fenster[16] der Klasse `JInternalFrame`. Die Menüleiste enthält mehrere Menüs der Klasse `JMenu` und die einzelnen Menüs weiter Elemente der Klasse `JMenuItem`. Ebenso enthält die Schaltflächenleiste Schaltflächen der Klasse `JButton`, in denen wiederum Symbole der Klasse `Icon` enthalten sind usw.

Die im obigen Klassendiagramm gezeigte Verknüpfung bewirkt zweierlei: Zum einen werden Komponenten zu immer größeren Aggregaten zusammengesetzt. Dies wird durch das Strukturmuster `Kompositum` beschrieben. Ein Kontainer übernimmt darin Funktionen der enthaltenen Komponenten. Wird z. B. das Hauptfenster verschoben, so ändern sich dadurch gleichzeitig die Positionen aller enthaltenen Komponenten. Diese intuitiv erwartete Verhaltensweise ergibt sich aber erst durch Anwendung einer solchen Konstruktion, bei welcher der Kontainer jeweils die Relativpositionen der enthaltenen Komponenten kennt.

[16] hier ist nur dasjenige im Vordergrund sichtbar

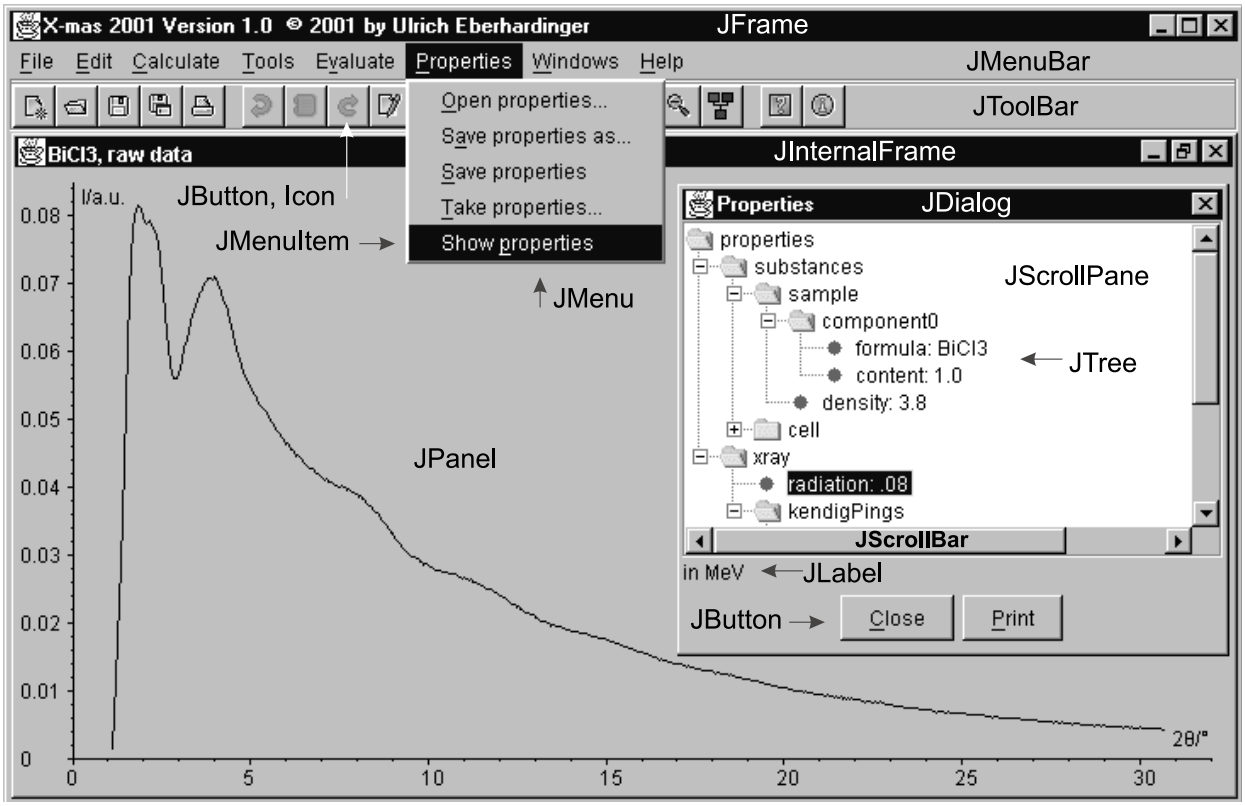


Abbildung 2.14: Kontainer-Hierarchie in Swing

Dieses Verhalten muss jedoch nicht für jede Klasse getrennt implementiert werden, sondern nur einmalig in den Superklassen `Component` und `Container`. Ähnlich funktioniert das Zeichnen einer Oberfläche: Für die äußerste Komponente der Klasse `JFrame` wird die Methode `paint(Graphics g)` aufgerufen und der Kontainer ruft diese Methode weiter für alle in ihm enthaltenen Komponenten auf, die selbst wieder Kontainer sein können. So wird der zu bezeichnende Graphikkontext `Graphics g` rekursiv an alle Komponenten weitergegeben, welche in ihn ihre spezifische Darstellung einzeichnen.

Formal identisch aber in der Zielsetzung verschieden ist das Entwurfsmuster `Dekorierer`, das ebenfalls durch die in Abbildung 2.13 gezeigte Konstruktion repräsentiert wird. Ein Beispiel dafür bietet die in Abbildung 2.14 gezeigte Anordnung. Dort befindet sich in einem Dialog ein Verzeichnisbaum, der in einer Ebene mit Schiebeleisten dargestellt ist. Nicht sichtbar ist die eigentliche Zeichenebene der Klasse `JPanel`, die in ein Objekt der Klasse `JScrollPane` eingebettet ist. Eine denkbare Vorgehensweise bestünde darin, die Klasse `JPanel` abzuleiten zu einer spezielleren Zeichenebene, welche mit Schiebeleisten ausgestattet ist. Flexibler ist jedoch der Ansatz des Entwurfsmusters `Dekorierer`, der darin besteht, eine Komponente durch Verwendung einer Aggregationsbeziehung schrittweise um Funktionalitäten zu erweitern, die sich in den von `Container` abgeleiteten Klassen befinden. Im vorliegenden Fall besteht diese Funktionalität darin, dass eine Zeichenebene nur ausschnittweise dargestellt wird und verschiebbar ist. Auch diese Erweiterung lässt sich rekursiv anwenden.

Obwohl die beiden Strukturmuster hier am Beispiel der Entwicklung von Benutzeroberflächen vorgestellt wurden, handelt es sich bei ihnen um allgemein anwendbare Konstruktionen. Ein weiteres wichtiges Beispiel des Java-API für das Entwurfsmuster `Dekorierer` sind die zahlreichen Eingabe/Ausgabe-Klassen im Paket `java.io`. Die beiden Stammklassen für Datenströme im Byte-Format[17] sind `InputStream` und `OutputStream`. Beispielsweise leiten sich von der Klasse `InputStream` weitere Klassen wie `FileInputStream`, `FilterInputStream`, `PipedInputStream` und `ByteArrayInputStream` ab, von denen sich weitere Klassen wie `BufferedInputStream`, `DataInputStream` und `ZipInputStream` ableiten. Analog zur Klasse `Container` enthalten diese Klassen eine Referenz auf eine Instanz vom Typ `InputStream`. Dadurch lassen sich die Ströme selektiv um die in den Subklassen implementierten Funktionalitäten wie Pufferung, Filterung und/oder Kompression erweitern.

Beobachter

Ein `Beobachter (Observer)` ist ein Verhaltensmuster, das in Java zur Ereignisbehandlung (event handling) eingesetzt wird. Ereignisbehandlung bedeutet beispielsweise folgendes: Wählt ein Benutzer den in Abbildung 2.14 gezeigten Menüeintrag *Show properties*, so öffnet sich der abgebildete Dialog zur Anzeige von Auswertungsparametern. Drückt er dann die Schaltfläche *Print*, so werden die angezeigten Parameter ausgedruckt. Die Frage ist nun: Wie erfährt die für den Ausdruck zuständige Methode davon, dass eine mit ihr logisch verbundene Schaltfläche gedrückt wurde, ohne dass spezielle Schaltflächen entwickelt werden müssen, die bestimmte Druckroutinen aufrufen? Ein Lösungsansatz besteht darin, für die Schaltfläche einen Beobachter zu registrieren, der in diesem Falle informiert wird.

Abbildung 2.15 zeigt das Klassendiagramm für die Ereignisbehandlung am Beispiel eines `ActionListener`s für die Klasse `AbstractButton`. `AbstractButton` ist die abstrakte Superklasse der Klassen `JButton` (Schaltflächen) und `JMenuItem` (Menüeinträge). Sie registriert mit der Methode `addActionListener(ActionListener l)` eine beliebige Anzahl von Objekten, deren Klassen das Interface `ActionListener` implementieren. Dieses Interface definiert nur die Methode `actionPerformed(ActionEvent evt)`. Registriert ein Objekt der konkreten Klasse `JButton` ein Ereignis, also das Drücken dieser Schaltfläche, so ruft `AbstractButton` für alle registrierten `ActionListener` deren Methode `actionPerformed(...)` auf und übergibt ihnen als Argument ein Objekt der Klasse `ActionEvent`, das die Ereignisquelle kapselt. Dies ist notwendig, da eine große Anzahl unterschiedlicher Elemente `ActionEvent`s schicken kann. Adressat der Ereignisse sei zum Beispiel ein von `JDialog` abgeleiteter Dialog der Klasse `MyDialog`, der eines oder mehrere Objekte der Klasse `JButton` enthält. Bei Bedarf kann das Objekt der Klasse `MyDialog` auf Informationen der Ereignisquelle zugreifen z. B. mit `checkButtonState()` auf die Methode `isSelected()`, was allerdings nur bei komplexeren Eingabeelementen sinnvoll ist.

Mit dieser Vorgehensweise wird erreicht, dass letztlich die Klassen `JButton` und `MyDialog` entkoppelt werden, d. h. ein `JButton` weiß nicht, dass es eine Klasse `MyDialog` überhaupt gibt und die Klasse `AbstractButton` leitet die Ereignisbehandlung ohne Kenntnis einer

[17] Daneben gibt es Klassenhierarchien für Datenströme im Character-Format

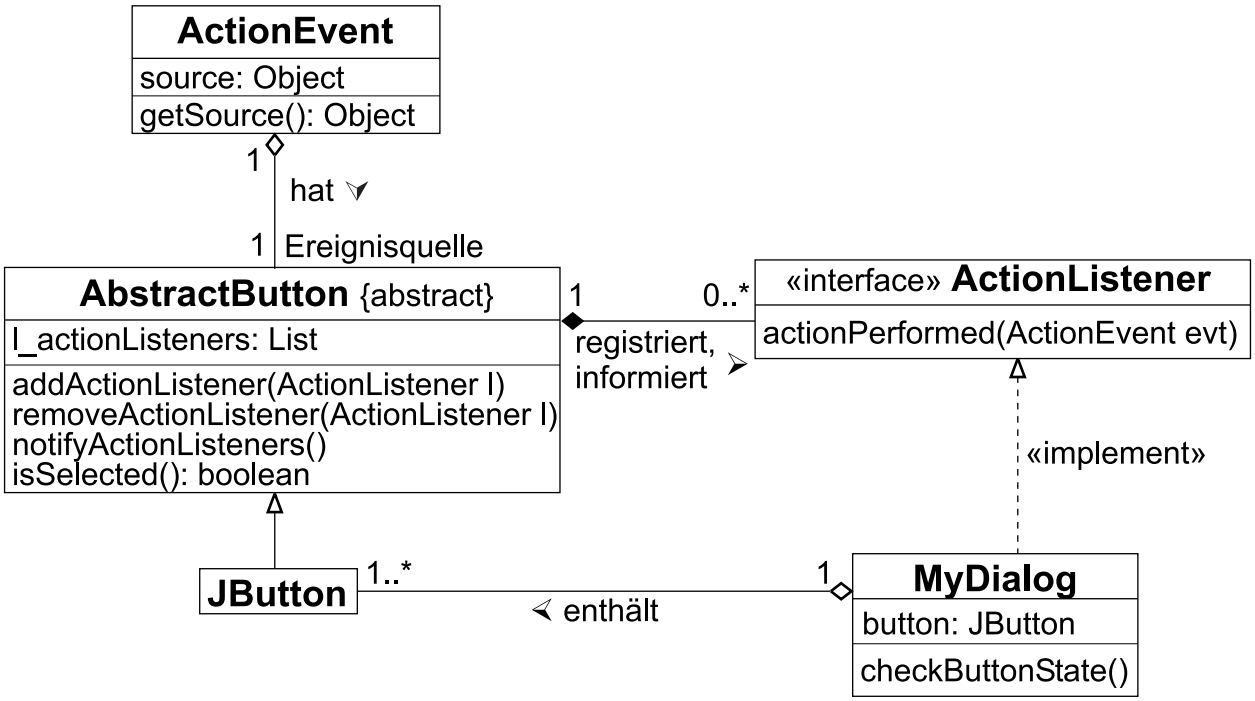


Abbildung 2.15: Ereignisbehandlung mit dem Entwurfsmuster `Beobachter`

spezifischen Klasse ein. Dies ist für die Entwicklung von flexiblen Klassenbibliotheken sehr wichtig.

MVC-Architektur

MVC steht für Model-View-Controller bzw. Modell-Darstellungs-Kontrolleinheit und ist ein seit langem bekanntes und grundlegendes Entwurfsmuster zur Programmierung von Benutzeroberflächen [25]. Seine Funktion entspricht in gewisser Weise der bei der prozeduralen Programmierung üblichen Gliederung in Eingabe, Verarbeitung und Ausgabe, wobei die Kopplung zwischen diesen Teilen möglichst schwach sein sollte.

Abbildung 2.16 zeigt die Beziehungen zwischen den enthaltenen Strukturelementen, bei denen es sich zunächst nicht um Klassen handelt, sondern um logische Einheiten, die auch aus mehreren Klassen bestehen können. Die MVC-Architektur dient der interaktiven Darstellung von Daten. Die Daten sind in der Einheit `Model` gekapselt. Diejenigen Klassen, welche für die Darstellung dieser Daten verantwortlich sind, gehören zur Einheit `View`, wobei dieselben Daten durch unterschiedliche `Views` dargestellt werden können, z. B. als Kuchen-, Balken- oder Liniendiagramm. Die Einheit `Controller` bestimmt die tatsächlich verwendete Darstellung und verarbeitet Eingaben des Benutzers.

Man erkennt, dass die Kontrolleinheit Zugriff auf Modelleinheit und Darstellungseinheit hat, während die Modelleinheit keinen direkten Zugriff auf die anderen Einheiten hat. Sie hat jedoch einen indirekten Zugriff auf die Anzeigeeinheit, um diese über Änderungen

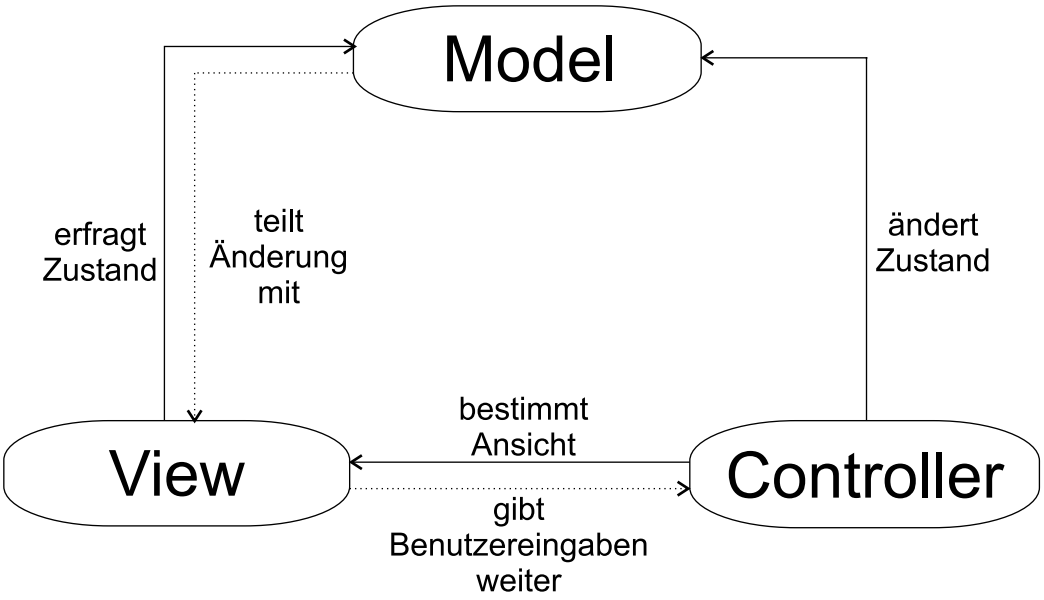


Abbildung 2.16: MVC-Architektur

der in ihr gekapselten Daten zu informieren. Ein solcher Zugriff erfolgt z. B. über ein vorgelagertes Interface mit einem Mechanismus, wie er beim Entwurfsmuster `Beobachter` beschrieben wurde. Häufig werden die Einheiten `View` und `Controller` zu einer einzigen Einheit zusammengefasst.

In der Swing-Architektur wird ein modifiziertes System angewandt. Abbildung 2.17 zeigt eine Anordnung für diejenigen Swing-Komponenten, die ein vergleichbares System verwenden. Zunächst soll nur die oberste Zeile betrachtet werden. Die Stammklasse `Model` existiert nicht und auch die Methode `getModel()` ist in `JComponent` nicht definiert. Jedoch existieren entsprechende Klassen für die zu betrachtenden Komponenten und für diese ist auch die Methode `getModel()` definiert. Beispielsweise ist für eine Tabelle die Definition eines Modells sinnvoll, für eine leere Zeichenebene hingegen nicht. Die Klasse `ComponentUI` kapselt eine austauschbare Darstellungseinheit (`View`), die für ein bestimmtes Betriebssystem spezifisch ist. Dadurch ist es möglich, dass mit Java programmierte Benutzeroberflächen das für das jeweilige Betriebssystem charakteristische Aussehen und Verhalten zeigen, ohne dass das Programm dazu verändert werden muss. Diese Eigenschaft wird als „pluggable Look and Feel“ (L&F) bezeichnet.

Die verwendeten Modelle reichen von sehr einfachen wie denen für Schalter, welche nur die Zustände ein/aus haben, bis zu sehr komplexen Modellen wie z. B. für formatierte Textdokumente mit eingebundenen Graphiken. Als Beispiel für die Verwendung von Modellen sei die Klasse `JTree` vorgestellt, die nachfolgend noch mehrfach verwendet wird. Diese Klasse dient der Darstellung von Baumstrukturen. Ein Baum ist eine elementare hierarchische Datenstruktur und ist aus Knoten und Kanten, welche die Knoten verbinden, aufgebaut. Jeder Knoten hat genau einen Vorfahren und eine beliebige Anzahl an Nachfahren. Ein-

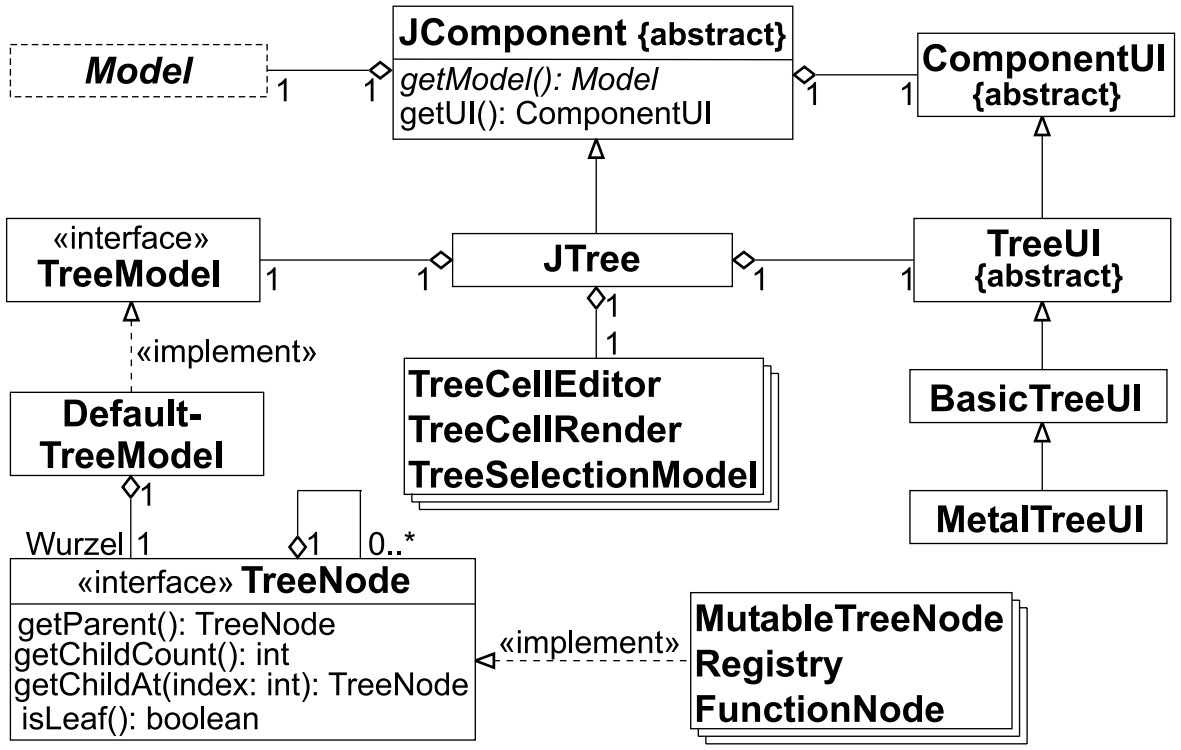


Abbildung 2.17: Swing-Architektur am Beispiel der Klasse `JTree`

zige Ausnahme ist die Wurzel des Baums, die über keinen Vorfahren verfügt. Von diesem Knoten ausgehend kann der gesamte Baum durchlaufen (traversiert) werden. Knoten ohne Nachfahren heißen Blätter.

Abbildung 2.14 zeigt einen Auschnitt des Baumes, der Eigenschaften (Properties) eines Datensatzes kapselt, auf die in Abschnitt 3.2 näher eingegangen wird. Die Klasse `JTree` leitet sich von `JComponent` ab. Ihr Modellinterface heißt `TreeModel` und ihre UI-Klasse `TreeUI`. Von `TreeUI` leitet sich `BasicTreeUI` ab und davon weiter `MetalTreeUI`, die Klasse mit dem Standard-L&F von Java *Metal*. Für Java-Interfaces existieren häufig Vorgabeimplementierungen, sogenannte Adapterklassen, wie z. B. `DefaultTreeModel` für das Interface `TreeModel`. Dieses Interface definiert Methoden, mit denen die Struktur des Baumes beschrieben wird. Es greift über die Wurzel des Baumes auf alle Knoten zu, die ihrerseits durch das Interface `TreeNode` definiert sind. Die Knoten selbst sind Objekte einer Klasse wie `MutableTreeNode`, `Registry` oder `FunctionNode`, welche dieses Interface implementieren. Das Interface legt durch seine Methodendeklarationen implizit fest, dass Instanzvariablen dieser Klassen Referenzen auf den Vorfahren und die Nachkommen haben müssen, welche selbst wieder dieses Interface implementieren. Weitere Instanzvariablen können Daten enthalten, die dem jeweiligen Knoten zugeordnet sind. Durch rekursive Referenzierung von Nachfahrenknoten entsteht ein gesamter Baum.

Baumstrukturen sind nahezu allgegenwärtig. Beispiele sind Dateisysteme, Vererbungshierarchien, Kontainerhierarchien und die im nächsten Kapitel vorzustellenden Klassen

`Registry` und `FunctionNode`. Für die Arbeit mit der Klasse `JTree` stehen weitere Klassen zur Verfügung, an welche Teile der `View-` und `Controller`-Einheiten delegiert werden, z. B. `TreeCellRenderer` zur graphischen Darstellung der einzelnen Knoten und deren gekapselten Daten, `TreeCellEditor` zur Editierung dieser Daten, `TreeSelectionListener` zur Registrierung von Auswahlereignissen. Beispielsweise wird bei der Auswahl einer Größe wie *radiation* aus den `properties` in Abbildung 2.14 in der Statuszeile deren Einheit *in* MeV angezeigt. Das Interface `TreeModelListener` dient der Registrierung einer Änderung der Baumstruktur, falls ein weiterer Unterbaum eingehängt oder ein Knoten entfernt wird und die Klasse `TreePath` beschreibt den Pfad eines Knotens ausgehend von der Wurzel.

Entsprechende Modellinterfaces und Hilfsklassen werden im Folgenden für andere Oberflächenelemente wie Tabellen, Listen, Textseiten usw. verwendet, ohne dass darauf explizit hingewiesen wird.

Kapitel 3

Programmstruktur

Die im Folgenden beschriebenen Klassen sind Teil des Programms X-mas (X-ray measurements analysis system), das der Auswertung von Röntgendiffraktogrammen dient. Es handelt sich vom Typ um eine Anwendung, die mehrere Dokumente (Diagramme) gleichzeitig verwalten kann (Multiple Document Application, MDA).

Die Klassen sind teilweise in UML-Klassendiagrammen dargestellt. Diese Diagramme sind nicht vollständig und beschränken sich auf wichtige Instanzvariablen und Methoden. Auf die Angabe von `get()` und `set()`-Methoden für den Zugriff auf Instanzvariablen wird in der Regel verzichtet. Das Programm besteht aus insgesamt über 300 Klassen und Interfaces mit insgesamt etwa 45 000 Zeilen Quelltext, die in dieser Arbeit aus Platzgründen nicht alle dokumentiert werden können. Die Klassen sind in neun Pakete gegliedert, welche den Verzeichnisnamen entsprechen: `diagram` für die Benutzeroberfläche und Verwaltungsklassen, `xrd` für Klassen zur Bearbeitung von Röntgendaten, `chemistry` für die Bearbeitung chemischer Daten, `math` für numerische Klassen, `math.function` zur Auswertung und Bearbeitung symbolischer Funktionen, `util` für Hilfsklassen, `util.enable` für den Aktivierungsmechanismus und `pointer` für die Referenzierung von Variablen und Funktionen sowie das externe Paket `jamax` (siehe Abschnitt 5.9).

Die in dieser Arbeit verwendete und an C++ angelehnte Schreibweise `Klasse::methode()` bedeutet, dass `methode` eine Instanzmethode in der Klasse `Klasse` ist. Hingegen bedeutet die mit der Java-Syntax identische Schreibweise `Klasse.methode()` den Aufruf einer statischen Methode der entsprechenden `Klasse`. Die Notation `Klasse$innereKlasse` entspricht der internen Darstellung in Java und bedeutet, dass eine innere Klasse in einer umschließenden Klasse definiert wurde.

3.1 Datensätze und Transformationen

Mittelpunkt der vorliegenden Anwendung sind Datensätze, welche Daten von Messkurven enthalten, und die im Verlauf der Auswertung verändert werden. Die Datensätze werden in der Klasse `Data` gekapselt und die darauf ausgeführten Transformationen in der Klasse `Transformation`. Die graphische Darstellung von Datensätzen erfolgt in den im Abschnitt 3.4 vorzustellenden Diagrammen. Abbildung 3.1 zeigt einen Überblick über die in den folgenden Abschnitten vorzustellenden Klassen und Interfaces.

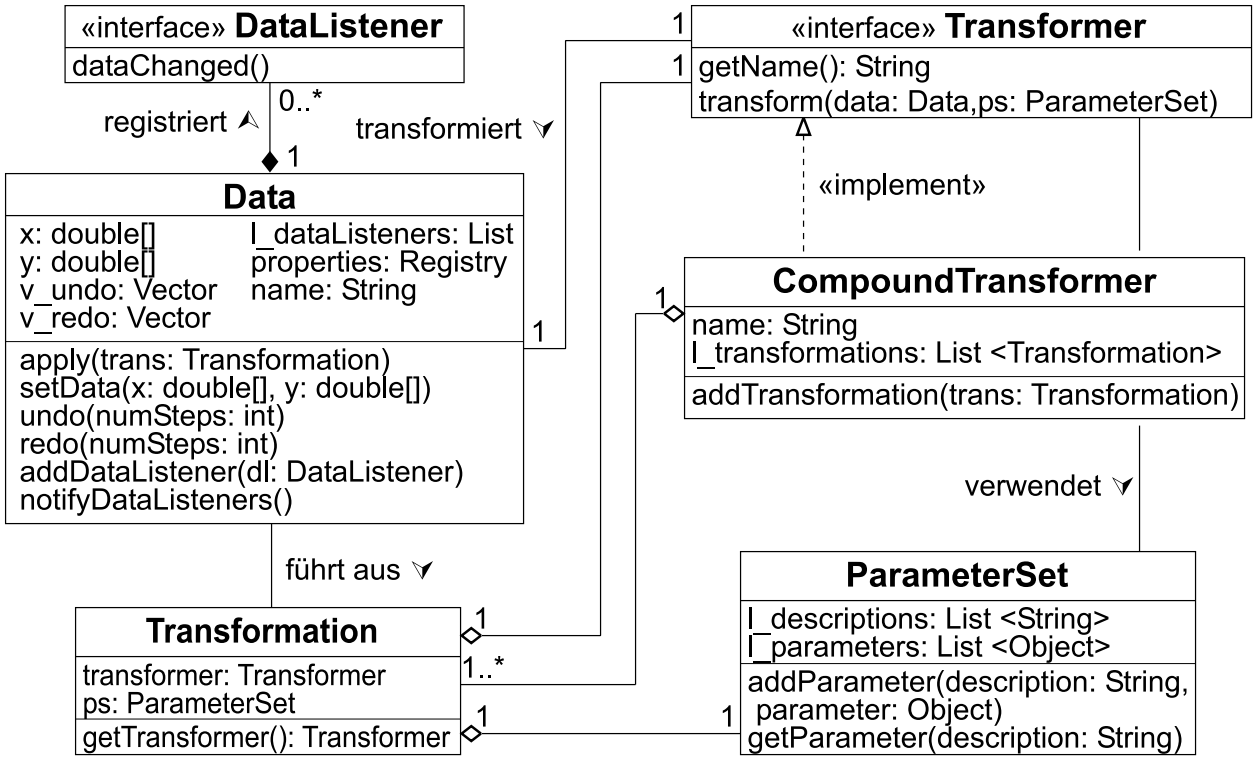


Abbildung 3.1: Klassendiagramm für Transformationen

3.1.1 Die Klasse `Data`

Diese Klasse kapselt die in Diagrammen darzustellenden Daten `x` und `y` und Informationen zu diesen Daten, z. B. Angaben darüber, ob die x-Werte in aufsteigender Reihenfolge sortiert oder äquidistant sind. Diese Angaben werden von zahlreichen numerischen Operationen gefordert und müssen mit den Methoden `isEquidistant()` bzw. `isOrdered()` gegebenenfalls vorher abgefragt werden. Neben den eigentlichen Daten werden in dieser Klasse auch in den `properties` vom Typ `Registry` Informationen abgelegt, die sich aus Parametern ergeben, welche bei Transformationen und anderen Operationen in Dialogen vom Benutzer eingegeben wurden. Diese Eigenschaften werden jedoch nicht für weitere Transformationen sondern nur zur Vorgabe von Werten in Eingabefeldern verwendet.

`Data` stellt `apply(Transformation trans)` und andere Methoden zur Ausführung der Transformationen bereit und führt in Form eines `Vectors` eine Liste über diejenigen Transformationen, die bereits auf ihm ausgeführt wurden. Dadurch können diese Transformationen rückgängig gemacht werden. Da zu vielen Transformationen (z. B. Absorptionskorrekturen, Glättungen usw.) keine oder keine einfach darzustellenden Rücktransformationen existieren, werden stattdessen die Transformationen ausgehend vom zuletzt gespeicherten Ausgangszustand erneut ausgeführt. Diese Vorgehensweise hat zugleich den Vorteil, dass keine Fortpflanzungsfehler durch Hintereinanderausführung zueinander inverser Operationen auftreten. In gleicher Weise existiert eine Liste rückgängig gemachter Transformationen, die wiederholt werden können, solange keine neuen Transformationen ausgeführt wurden. Beide Listen werden bei jeder Speicherung eines Datensatzes zurückgesetzt.

Mit den Methoden `getXData()` und `getYData()` können die Felder, in welchen sich die Abszissen- und Ordinatenwerte befinden, erhalten und verändert werden, wobei jedoch stets die Feldgrößen erhalten bleiben. Für gewünschte Änderungen, die mit einer Feldgrößenveränderung einhergehen, steht darüber hinaus die Methode `setData(double[] x, double[] y)` zur Verfügung, die eine abzufangende Ausnahme vom Typ `IllegalArgumentException` hervorruft, falls die Feldgrößen der übergebenen Arrays sich unterscheiden. In beiden Fällen sollten Veränderungen dieser Werte jedoch nur in `Transformer`n vorgenommen werden.

3.1.2 Das Interface `Transformer`

Dieses Interface ist Teil des Verhaltensmusters `Befehl` (`Command`), welches dazu dient, Operationen anstatt in einer Methode in einer objektorientierten Struktur zu kapseln. Im Gegensatz zu Methoden können die Operationen als Objekte mit dem Typ dieses Interfaces abgelegt und bei Bedarf erneut ausgeführt werden, wodurch die genannten Operationen `undo()` und `redo()` möglich werden.

Das Interface definiert die Methoden `String getName()`, die den Namen eines Transformers zurückgibt und `void transform(Data data, ParameterSet ps)`, welche eine numerische Operation auf den Daten von `Data` mit den in `ParameterSet` abgelegten Parametern ausführt. In Implementierungen von `transform()` wird über die Methoden `double[] Data::getXData()` bzw. `double[] Data::getYData()` auf die Daten des Datensatzes zugegriffen, die nachfolgend wie gewünscht verändert werden.

`Transformer`, welche eine Änderung der in `Data` definierten Eigenschaften `isOrdered()` und `isEquidistant()` erwarten lassen, müssen diese Eigenschaften nach Ausführung der Transformation durch einen Aufruf von `Data::checkOrder()` bzw. `Data::checkEquidistance()` aktualisieren.

3.1.3 Die Klasse `ParameterSet`

`ParameterSet` kapselt die für Transformationen verwendeten Parameter `parameter` und ihre Bezeichnungen `description`. Die Methode `addParameter(String description, Object parameter)` fügt Parameter hinzu. Diese können mit `Object getParameter(String description)` nach Bezeichnung und `Object getParameter(int pos)` nach Position wieder ausgelesen werden. Mit `String getDescription(int pos)` ist eine Abfrage der Bezeichnungen z. B. für die Ausgabe von Parametern in anderen Klassen möglich.

Es ist zu beachten, dass die der Methode `addParameter(...)` übergebenen Parameter konstant sein müssen. Dies können unveränderliche Objekte (Immutables wie `Integer`, `Double` usw.) oder tiefe Kopien sein, die keine Referenzen auf sich verändernde Größen enthalten. Es sei darauf hingewiesen, dass die Methode `Object clone()` des Interfaces `Cloneable` bzw. der Klasse `Object`, welche dieses Interface implementiert, nicht festlegt, ob das Ergebnis der Operation eine tiefe oder ein flache Kopie ist und dass Java-Klassen entsprechend uneinheitlich vorgehen.

3.1.4 Die Klasse `Transformation`

Diese Klasse ist eine Aggregation der zu einer Transformation gehörigen Teile `Transformer` und Parametersatz `ParameterSet`. Objekte dieses Klasse werden als ausführbare Operationen in der Klasse `Data` abgelegt und operieren auf den Datensätzen.

3.1.5 Die Klasse `CompoundTransformer`

In einem `CompoundTransformer` können mehrere `Transformationen` zu einem `Transformer` zusammengefasst werden. Mit der Methode `addTransformation(Transformation trans)` kann eine beliebige Anzahl an Transformationen hinzugefügt werden. Die über das Interface `Transformer` implementierte Methode `transform(...)` ruft dieselbe Methode für die `Transformer` der enthaltenen `Transformation`en in derjenigen Reihenfolge auf, in der sie hinzugefügt wurden.

3.1.6 Das Interface `DataListener`

Die Klasse `Data` enthält eine Liste von Objekten vom Typ `DataListener`. Dieses Interface dient der Realisierung des Verhaltensmusters `Beobachter` und definiert die Methode `dataChanged()`, die aufgerufen wird, wenn ein Datensatz geändert wird. Beispielsweise implementieren die Klasse `Diagram` und damit die davon abgeleiteten Klassen `SingleDiagram` und `MultiDiagram` (siehe Abschnitt 3.4) dieses Interface und bauen ihren Inhalt, die graphische Repräsentation von Datensätzen, nach Aufruf genannter Methode neu auf. Zur Registrierung der `DataListener` bietet die Klasse `Data` die Methode `add(DataListener dl)` und zur Entfernung entsprechend `removeDataListener(DataListener dl)`. Wird ein Datensatz durch eine Transformation (`apply/undo/redo/repeat`) oder eine andere Methode wie `setData()` oder `setName()` verändert, so ruft er intern die Methode `notifyDataListeners()` auf, welche alle registrierten `DataListener` durch Aufruf von `dataChanged()` informiert.

3.1.7 Die Behandlung rechenintensiver Operationen

Einige Berechnungen wie z. B. die Absorptionskorrektur nach Kendig-Pings oder die Fouriertransformation sind so rechenintensiv, dass die Benutzeroberfläche normalerweise minutenlang weder auf eine Eingabe reagiert noch die Oberfläche aktualisiert, falls sie z. B. durch Fenster anderer Programme im Vordergrund gestört wurde. So entsteht für den Benutzer der Eindruck, das Programm sei abgestürzt. In solchen Fällen ist es üblich, Anzeige und Berechnung auf parallel laufende Unterprozesse (Threads) zu verteilen. Bei der Arbeit mit Threads muss darauf geachtet werden, dass die Elemente einer Klasse nicht von mehreren parallel laufenden Prozessen gleichzeitig verändert werden können. Dazu müssen Prozesse synchronisiert werden. In Java steht dafür das Schlüsselwort `synchronized` zur Verfügung, welches veranlasst, dass die JVM den Zugriff auf ein Objekt für dritte Prozesse sperrt, solange ein Prozess auf eine mit diesem Schlüsselwort gekennzeichnete Methode zugreift. Alle Zugriffsmethoden der Klasse `Data` sind synchronisiert. Nahezu alle Methoden des Pakets Swing zur Oberflächengestaltung sind nicht für die Handhabung mit Threads vorbereitet (threadsafe), so dass hier spezielle Maßnahmen ergriffen werden müssen.

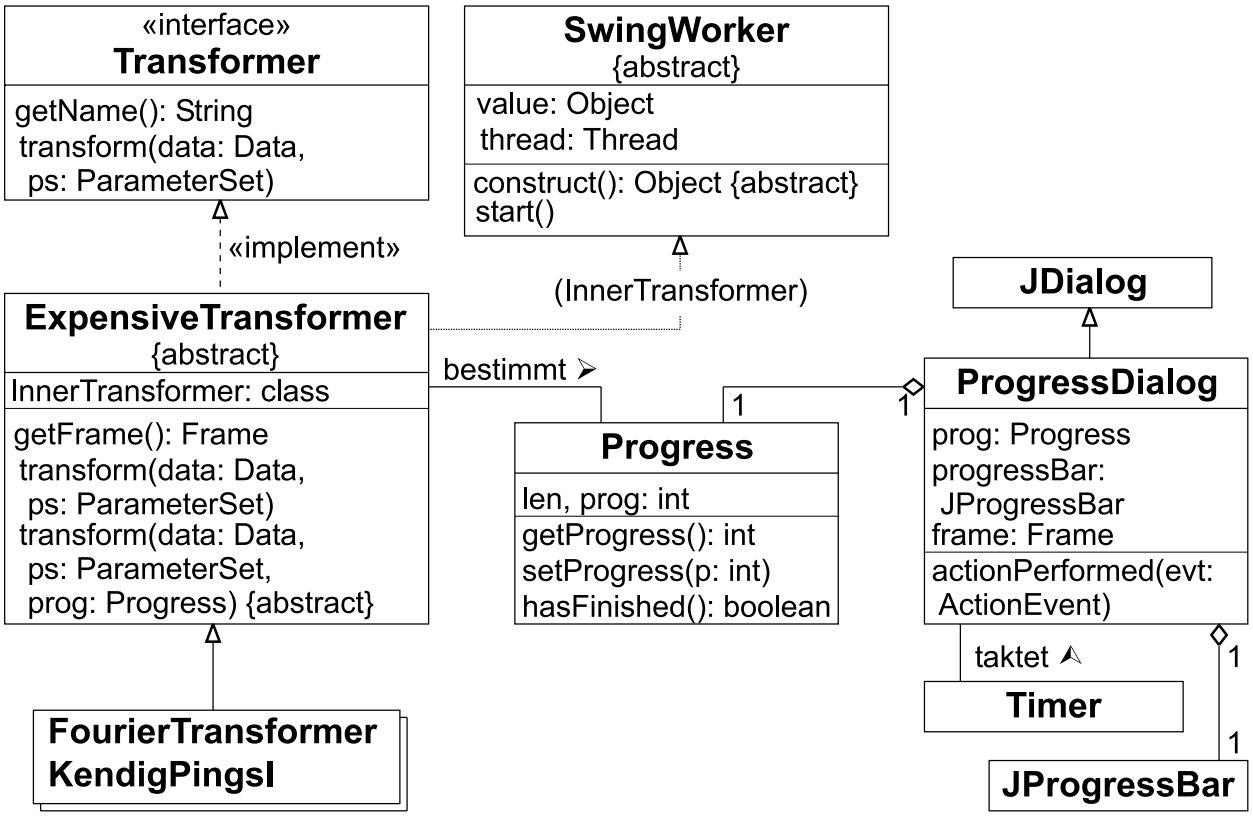


Abbildung 3.2: Klassendiagramm für rechenintensive Transformationen

Normalerweise wird die Benutzeroberfläche im Hauptprozess erzeugt und wird anschließend nur noch durch Ereignisbehandlungsprozesse beeinflusst, die durch Aktivierung von Menüeinträgen und Dialogeingaben ausgelöst werden. Möchte man jedoch den aktuellen Zustand einer laufenden Berechnung anzeigen, so reicht diese Vorgehensweise nicht aus. Für die gleichzeitige Arbeit mit Threads und Swing existiert die Klasse `SwingWorker` der Firma Sun, welche jedoch nicht zum Standardumfang von Java gehört. Diese Klasse wurde daher in der Version 3 in die eigene Klassenbibliothek integriert. Ihre Struktur und Verwendung wurde unter [26] dokumentiert. Sie verwaltet den Zeichenprozess und den Rechenprozess und steuert diese asynchron. Die Handhabung dieser Klasse ist relativ umständlich und wurde für die Ausführung von Transformationen durch einige Klassen verbessert, deren Zusammenwirken in Abbildung 3.2 dargestellt ist.

Als Schnittstelle zur Klasse `SwingWorker` dienen die Methode `abstract construct()`, mit welcher der Konstruktor einer Klasse aufgerufen wird, in welcher die Berechnung ausgeführt wird und die Methode `start()`, mit welcher die Berechnung gestartet wird. Der Konstruktoraufruf findet dabei in einem Thread statt, der in der Klasse `SwingWorker` initiiert wird. Wie oben ausgeführt wurde, bildet das Interface `Transformer` die Schnittstelle zur Ausführung von Transformationen und definiert die Methode `transform(Data data, ParameterSet ps)`. Die abstrakte Klasse `ExpensiveTransformer` implementiert diese Methode und erzeugt in ihr die Instanz einer anonymen inneren Subklasse von `SwingWorker` und ruft für sie die Methode `SwingWorker::start()` auf. Die überschriebene Methode `Ob-`

`ject construct()` dieser inneren Klasse ruft weiter den Konstruktor der nicht dargestellten inneren Klasse `ExpensiveWorker$InnerWorker` auf, die nur einen Konstruktor enthält. In diesem wird die abstrakte Methode `transform(Data data, ParameterSet ps, Progress prog)` aufgerufen, die von Subklassen wie `FourierTransformer` und `KendigPingsI` ebenso wie die Methode `String getName()` implementiert wird. Im Konstruktor wird außerdem eine Instanz der Klasse `Progress` erzeugt und einem ebenfalls erzeugten `ProgressDialog` übergeben. Die Klasse `Progress` gibt an, wie weit eine Berechnung fortgeschritten ist. Die Informationen darüber schreibt die berechnende Methode `transform(...)` in diese Klasse und wird vom `ProgressDialog` wieder ausgelesen. Dieser von `JDialog` abgeleitete Dialog enthält eine `JProgressBar` zur graphischen Anzeige des Fortschreitens (siehe Abbildung 3.3). Eine Instanz der Klasse `javax.swing.Timer` ruft die Methode `actionPerformed()` in definierbaren Zeitintervallen auf, welche zu einer Aktualisierung der Anzeige führt.

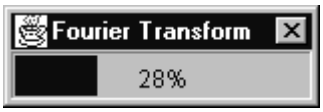


Abbildung 3.3: Dialog zur Anzeige eines Transformationsverlaufs

Diese unübersichtlich erscheinende Konstruktion führt zu einem sehr einfachen Ergebnis: Zeitaufwändige Transformationen implementieren nicht das Interface `Transformer`, sondern leiten sich von der Klasse `ExpensiveTransformer` ab und implementieren deren abstrakte Methode `transform(...)`.

3.2 Die Registratur

Wie schon in der Beschreibung der Klasse `Data` angedeutet, werden neben den zu verändernden Daten auch Informationen zu Auswertungsschritten und insbesondere das untersuchte System benötigt. Für die Vielfalt der auftretenden Informationen musste eine Struktur mit einheitlicher Schnittstelle entwickelt werden: die Registratur. Gleichzeitig sollte es möglich sein, diese Daten zu speichern und wieder zu laden, wozu das XML-Format verwendet wird.

3.2.1 Die Extensible Markup Language XML

Die Übersetzung für XML [27] lautet *erweiterbare Auszeichnungssprache*. Diese plattformunabhängige Sprache hat sich innerhalb kürzester Zeit zum Industriestandard für den Datenaustausch im Internet entwickelt. Sie wurde seit 1996 entwickelt und 1998 vom W3C-Konsortium standardisiert und dient dazu, strukturierte Daten in Textdateien abzulegen. XML ähnelt der Sprache HTML [28] (Hypertext Markup Language), mit der Textseiten im Internet dargestellt werden.

In beiden Sprachen werden Daten zwischen Tags („Schildchen“) eingeschlossen, die Attribute enthalten können. In HTML wird ein Hyperlink (ein netzweiter Querverweis) folgendermaßen formuliert:

```
<A HREF="url">Dies ist ein Hyperlink</A>
```

Ein Hyperlink ist demnach zwischen zwei Tags `<A>` und `</A>`[1] eingeschlossen, wobei das einführende Tag zusätzlich das Attribut `HREF` mit dem Wert `url`[2] enthält. Ein Internetbrowser interpretiert diese Information so, dass der zwischen den Tags stehende Text in der für das Tag festgelegten Formatierung dargestellt wird und ein Klick des Benutzers auf diesen Text zu einem Wechsel auf der im Attribut festgelegten Internetadresse `url` führt. In XML ist für jedes einführende Tag ein gleichnamiges abschließendes Tag vorgeschrieben. Als Spezialfall existieren leere Tags, welche dieses Paar zusammenfassen: `<TAG/>`. Tags können geschachtelt werden, z. B. stellen in HTML `<TR>` Tabellenzeilen dar und `<TD>` Tabellenspalten. Mit

```
<TR>
   <TD>Spalte 1</TD>
   <TD>Spalte 2</TD>
   <TD>Spalte 3</TD>
</TR>
```

lässt sich eine Tabellenzeile darstellen. Das Aneinanderhängen solcher Zeilen ergibt eine ganze Tabelle.

Während in HTML die Bedeutung der Tags und Attribute verbindlich festgelegt ist, ist in XML lediglich diese aus Tags, Attributen und Daten bestehende Struktur vorgeschrieben. Im Gegensatz zu HTML sind bei XML aber keine Ausnahmen von dieser Struktur zugelassen. Es bleibt dem Programmierer überlassen, für eigene Daten eine Grammatik zu definieren, welche eine mögliche Abfolge und Kombination der einzelnen Elemente vorschreibt. XML-Grammatiken werden als DTD (Document Type Definition) bezeichnet. Für eine Vielzahl von Datenformaten existieren bereits Modelle, mit denen Konversionen nach XML durchgeführt werden können.

3.2.2 Die Klasse `Registry`

Die Registratur ist aus zahlreichen Einträgen zusammengesetzt, welche Objekte der Klasse `Registry` sind. Diese Klasse hat im Wesentlichen die in Abbildung 3.4 dargestellte Struktur.

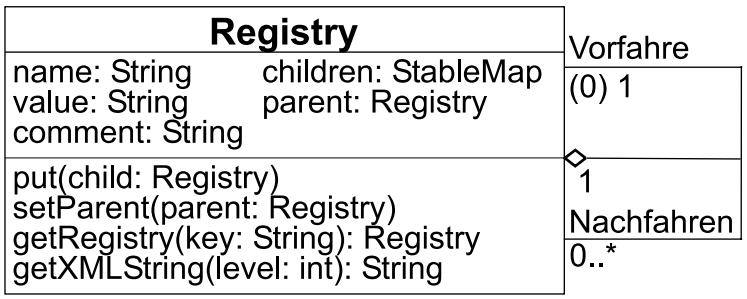


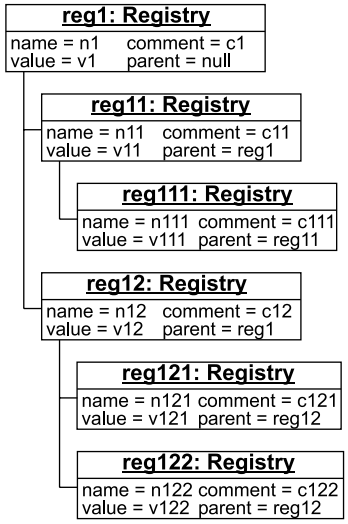
Abbildung 3.4: Die Klasse `Registry`

[1] die Groß- und Kleinschreibung ist nicht festgelegt
[2] die Anführungszeichen sind in XML vorgeschrieben

Jeder Eintrag enthält eine Bezeichnung `name`, einen Wert `value` und einen Kommentar `comment`, wobei nur die Bezeichnung obligatorisch ist. Die Klasse `Registry` implementiert das Interface `javax.swing.tree.TreeNode` und dient damit der Erzeugung einer Baumstruktur (siehe Abschnitt 2.3.7). Die zahlreichen in diesem Interface definierten Methoden sind nicht aufgeführt. Bäume erfordern Referenzen auf die eigene Klasse, zum einen durch die Variable `parent` auf den Vorfahren, zum anderen auf die in der `StableMap children` enthaltenen Nachfahren.

Die Klasse `StableMap` implementiert die Interfaces `Map` und `Cloneable` und enthält zwei Listen gleicher Länge, wobei die Einträge der einen Liste mit denjenigen der anderen Liste und gleichem Index korrespondieren. Diese Klasse vereinigt die Ordinalitätseigenschaft von Listen mit der Zuordnungsfunktionalität von Abbildungen und wird zum Beispiel verwendet, um Zuordungen zu sortieren und auf diese indiziert zuzugreifen.

Im vorliegenden Fall beinhaltet `children` Zuordnungen der Form `name -> (sub)registry`. Damit ist unter Angabe eines Pfades ein direkter Zugriff auf Werte möglich wie auf Dateien in einem Dateisystem. Die Baumstruktur der Registratur erlaubt es, einzelnen Programmmodulen selektiv Teilbäume zu übergeben, wodurch diese Module nur einen Zugriff auf den sie betreffenden Datenbereich haben. Abbildung 3.5 zeigt links ein einfaches Beispiel für eine Registratur.




```
<n1 value="v1" comment="c1">
   <n11 value="v11" comment="c11">
      <n111 value="v111" comment="c111">
      </n111>
   </n11>
   <n12 value="v12" comment="c12">
      <n121 value="v121" comment="c121">
      </n121>
      <n122 value="v122" comment="c122">
      </n122>
   </n12>
</n1>
```

Abbildung 3.5: Beispiel für eine Registratur und ihre XML-Repräsentation

Die Methode `getXMLString(int level)` ruft rekursiv sich selbst in den nachfahrenden Knoten auf und erzeugt so eine Zeichenkette, welche die in der Registratur abgelegten Daten enthält. Der Parameter `level` gibt dabei die Verschachtelungstiefe an. Die Codierung bildet die Bezeichnung der Knoten auf XML-Tags und die Werte und Kommentare auf Attribute ab. Abbildung 3.5 zeigt, wie die Objektstruktur der linken Seite auf die XML-Struktur auf der rechten Seite abgebildet wird, wobei die Referenzen auf Vorgänger und Nachfolger implizit in der Verschachtelungsstruktur enthalten sind. Abbildung 3.6 zeigt eine für die

Programmparameter `properties` verwendete Registratur in einem Dialog der Klasse `RegistryDialog`.

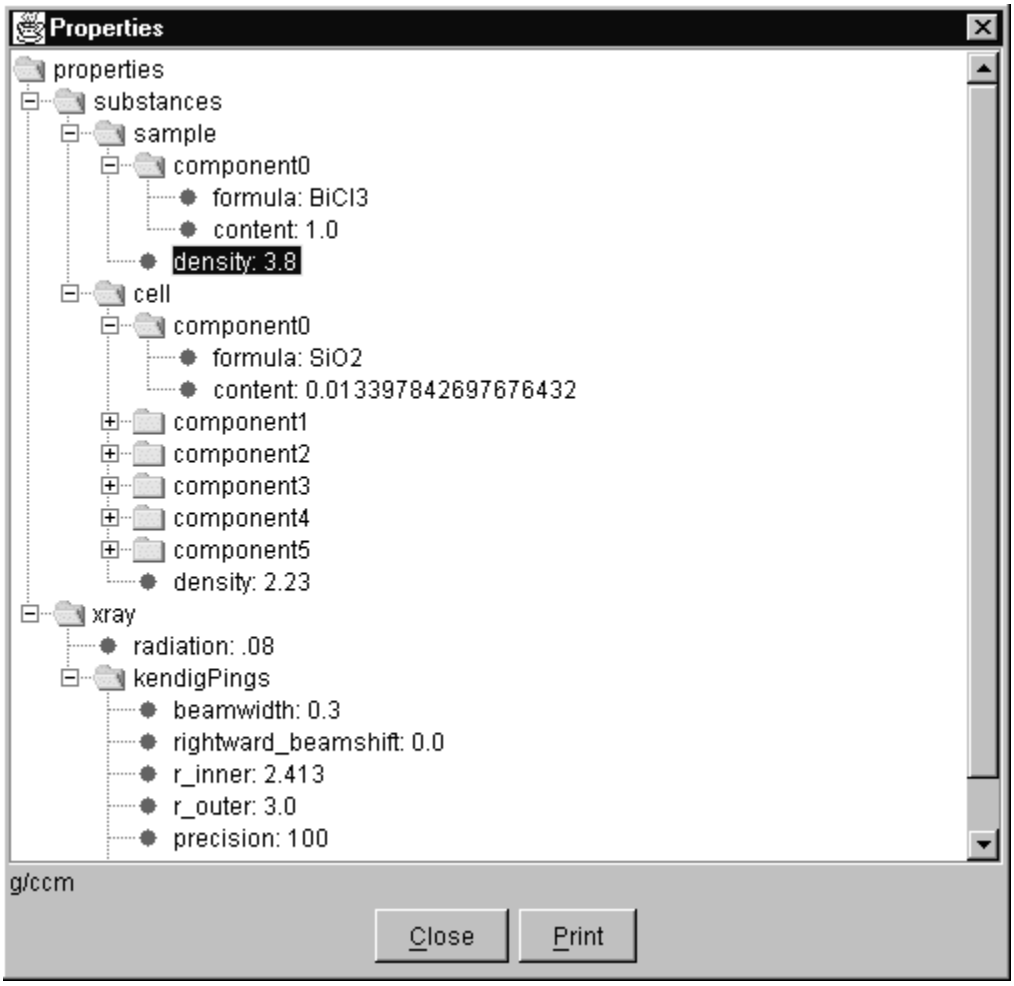


Abbildung 3.6: Anzeige der Registratur mit einem `RegistryDialog`

Anmerkung: Am 25. Mai 2001 wurde auf den Internetseiten der Java-Entwicklergemeinschaft (Java Developer Connection, JDC, `http://developer.java.sun.com/developer/`) die Beta-Version von Java 2 Version 1.4 vorgestellt, die im 4. Quartal 2001 veröffentlicht werden soll. Diese Version enthält eine Klasse `Preferences`, deren Zielsetzung in etwa der Klasse `Registry` entspricht. Zum Zeitpunkt der Entwicklung war diese Klasse dem Autor der vorliegenden Arbeit nicht bekannt und war ihm vor dem genannten Zeitpunkt auch nicht zugänglich.

3.2.3 Die Klasse `ExtendedRegistry`

Es wurde gezeigt, dass alle Daten `value` in der `Registry` als Zeichenketten abgelegt sind. Dies ist prinzipiell auch für Zahlen und andere Formate möglich, wenn geeignete Konversio-

nen angewandt werden. Bei Zahlenwerten ist es unökonomisch, die Umwandlung der Typen `String <-> Double` häufig vorzunehmen und es ist daher wünschenswert, die Werte auch intern in einem geeigneten Zahlenformat darzustellen.

`ExtendedRegistry` ist eine abstrakte Klasse, die sich von der Klasse `Registry` ableitet. Sie enthält Daten in einem beliebigen Format, die nur bei Bedarf als Objekte vom Typ `Registry` in Erscheinung treten. Beispielsweise enthält die im Abschnitt 3.3.5 vorzustellende Klasse `Mixture` die innere Klasse `Mixture$Density`, die sich von `Registry` ableitet und auf die Instanzvariable `Double Mixture::density` zugreifen kann. In anderen Fällen möchte man direkt auf Objekte zugreifen, die von den Nachfolgern als Wert referenziert werden. So verwaltet die Klasse `Mixture` Referenzen auf die Komponenten einer Mischung. Diese Komponenten sind Objekte der Klasse `ChemicalComponent`, die sich von `ExtendedRegistry` ableitet. Sie stellt Methoden zur Verfügung, über welche die Klasse `Mixture` auf ihre Instanzvariablen zugreifen kann, ohne den Umweg über allgemeine Methoden der Klasse `Registry` gehen zu müssen.

Für die im Folgenden beschriebene Serialisierung und Deserialisierung der Registratur wird weiterhin die Methode `String Registry::getXMLString(int level)` überschrieben, die für die Klasse `ExtendedRegistry` als zusätzliches Attribut den Namen der jeweiligen Subklasse in den XML-Code aufnimmt.

3.2.4 Die Klassen `RegistryParser` und `XMLExpression`

Während die Erzeugung von XML-Code relativ einfach ist, erfordert es einigen Aufwand, XML-Dateien wieder einzulesen. Die Unterstützung von XML durch Java befindet sich noch in der Entwicklungsphase. Es existiert jedoch bereits ein Parser mit der Bezeichnung SAX (Simple API/serial access protocol for XML), der XML-Strukturen analysiert. Er ist Teil des Java-Erweiterungspaketes `jaxp` und ist mit einem Tutorial [29] dokumentiert.

Die Klasse `javax.xml.parsers.SAXParser` zerlegt in der Methode `parse(InputStream in, DefaultHandler dh)` die Daten, die ihr über den z. B. aus einer XML-Datei stammenden Eingabestrom `in` zugeleitet werden, in XML-Elemente und ruft für diese eine Instanz von `org.xml.sax.helpers.DefaultHandler` auf. Diese Klasse enthält Leerimplementierungen der Methoden `startElement(String namespaceURI, String localName, String qName, Attributes attrs)` und `endElement(java.lang.String uri, java.lang.String localName, java.lang.String qName)`, die im Interface `org.xml.sax.ContentHandler` definiert sind. Für jedes einleitende oder leere Tag wird die Methode `startElement(...)` aufgerufen und für jedes abschließende Tag die Methode `endElement(...)`. Als Parameter werden diesen Methoden unter anderem die Bezeichnung des Tags `qName` und die Attribute `attrs` übergeben.

Die Klasse `DefaultHandler` nimmt selbst keine Auswertung dieser Parameter vor, sondern die von ihr abgeleitete Klasse `RegistryParser`, welche die genannten Methoden überschreibt. Diese Klasse wurde entwickelt, um die zuvor aufgeführten Parserklassen zu steuern. Sie erzeugt letztlich mit der Methode `static Registry getRegistry(InputStream in)`

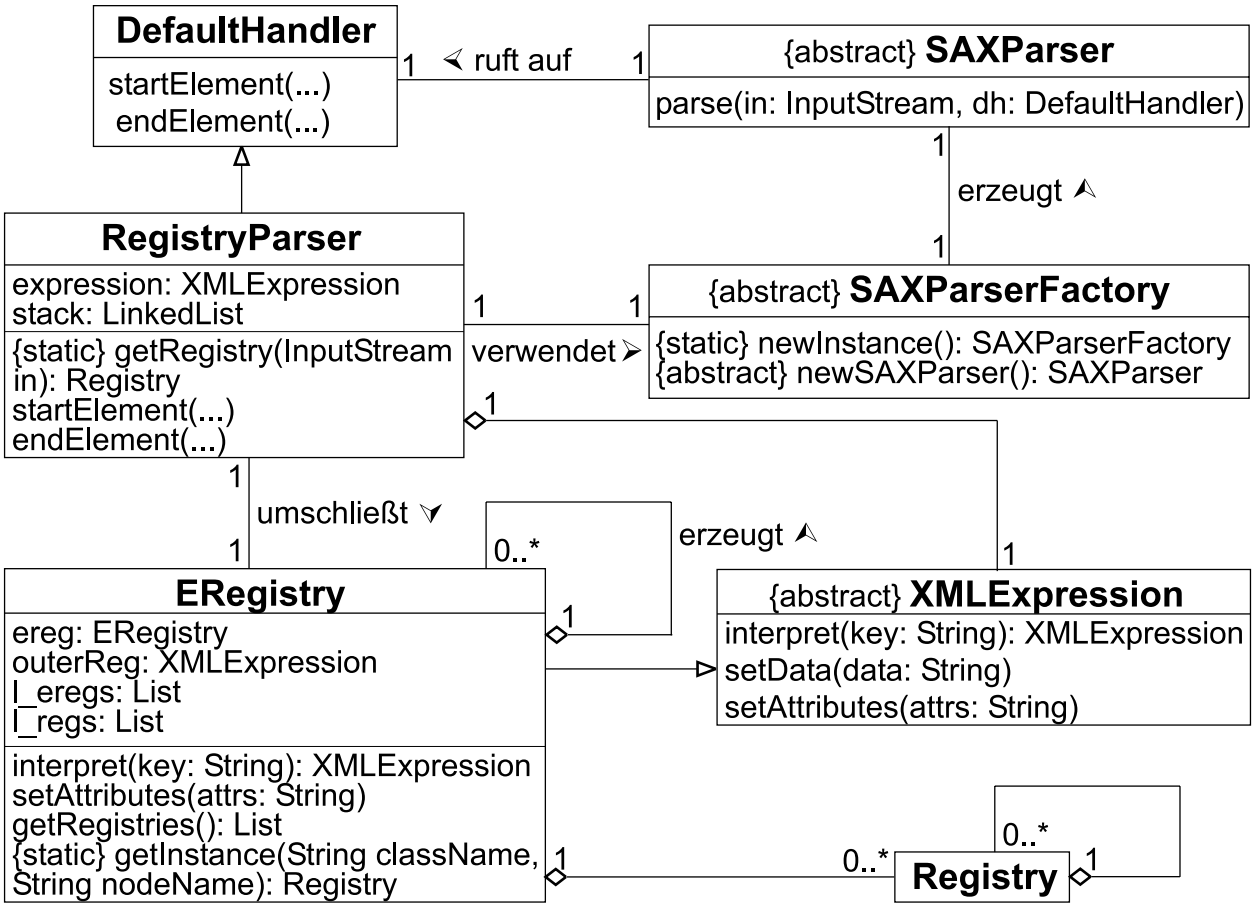


Abbildung 3.7: Klassendiagramm für die Verwendung von `RegistryParser`

aus dem Eingabestrom `in` die `Registry`. Dazu verwendet sie ein mit der statischen Methode `SAXParserFactor SAXParserFactor.getInstance()` erzeugtes Objekt vom Typ eben dieser abstrakten Klasse, mit welchem über die Methode `SAXParser newSAXParser()` ein Objekt der Klasse `SAXParser` erzeugt wird, dem der Eingabestrom und eine Instanz der Klasse `RegistryParser` als `DefaultHandler` übergeben wird. Dadurch schließt sich der Kreis zu den anfänglichen Ausführungen.

Es ist nun Aufgabe der Klasse `RegistryParser`, die Parameter der Methoden `startElement(...)` und `endElement(...)` auszuwerten. Dazu wurde weiter die abstrakte Klasse `XMLExpression` entwickelt, welche die abstrakte Methode `XMLExpression interpret(String key)` definiert und Leerimplementierungen der Methoden `setAttributes(String attrs)` und `setData(String data)` zur Verfügung stellt. Dabei sind `key` die Bezeichnung des Tags, `attrs` die genannten Attribute und `data` die zwischen einleitenden und abschließendem Tag befindlichen Daten, die bei Behandlung der Registratur nicht vorhanden sind bzw. benötigt werden und durch die Leerimplementierung ignoriert werden. Die Klasse `XMLExpression` arbeitet in Anlehnung an das Verhaltensmuster `Interpreter`. Zu ihrer Verwendung werden konkrete Subklassen formuliert, welche den aktuellen Zustand des Interpreters kapseln. Durch Aufruf der Methode `XMLExpression interpret(String key)`

findet in Abhängigkeit des aktuellen Tags `key` der Übergang zu einem neuen Objekt vom Typ `XMLExpression` statt, das als Rückgabewert dieser Methode auftritt. Durch geeignete Implementierungen der Methode `interpret(...)` definieren die Subklassen die für einen Zustand möglichen Übergänge.

Im vorliegenden Fall sind die Zustände und Übergänge trivial[3], da die zu interpretierende Registratur per definitionem nur aus Objekten vom Typ `Registry` aufgebaut ist. Das hieraus folgende Verhalten wird durch die genannten Methoden in der von `XMLExpression` abgeleiteten Klasse `ERegistry` garantiert, die eine innere Klasse von `RegistryParser` ist. `ERegistry` verfügt ebenso wie `Registry` über eine rekursive Kompositionsstruktur. Die Methode `RegistryParser::startElement(...)` führt weiter zu einem Aufruf der Methode `ERegistry ERegistry::interpret(...)` mit demselben Parameter, wobei die jeweilige Instanz der Klasse `ERegistry` eine Liste aller Objekte derselben Klasse enthält, die durch sie erzeugt wurden. So entsteht eine Baumstruktur von `ERegistry`-Objekten, die sich aus der geschachtelten Struktur der XML-Datei ergibt und weiter auf eine Baumstruktur von `Registry`-Objekten abgebildet wird. Diese Umsetzung wird durch Aufruf der Methode `List getRegistries()` eingeleitet, welche rekursiv für alle Knoten des Baumes aufgerufen wird.

Eine Komplikation ergibt sich dadurch, dass zunächst nicht bekannt ist, zu welcher Klasse ein neu zu erzeugendes `Registry`-Objekt gehört. Oben wurde ausgeführt, dass zum Teil Subklassen von `ExtendedRegistry` für spezialisierte Teile der Registratur verwendet werden. Deren Klassen werden durch die Attribute festgelegt. Das Java `Relection`-API erlaubt die Erzeugung von Objekten einer Klasse über den Namen der Klasse. Die Methode `Registry ERegistry.getInstance(String className, String nodeName)` verwendet dieses API zur Erzeugung der gewünschten Objekte vom Typ `ExtendedRegistry`.

3.3 Die Beschreibung chemischer Substanzen und Mischungen

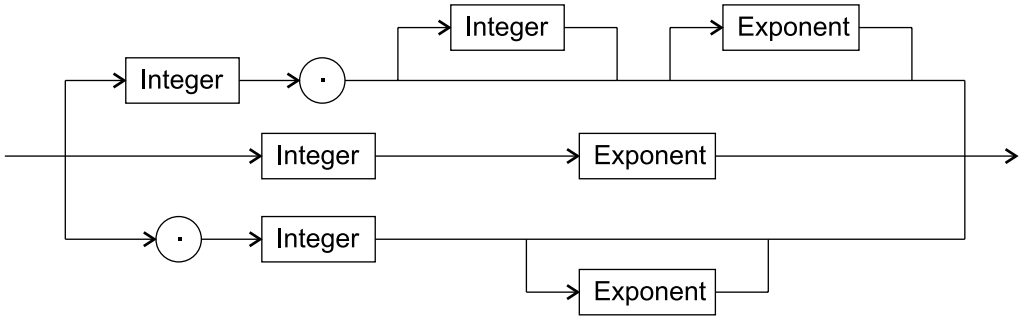
Das Programm benötigt in mehreren Auswertungsschritten Informationen zur chemischen Zusammensetzung des untersuchten Systems. Die Erkennung chemischer Formeln in Zeichenketten erfolgt durch die Klasse `ChemicalFormulaParser`. Die zulässigen Elementsymbole sowie die Atommassen und Ordnungszahlen der Elemente befinden sich in der Klasse `PSE`. Zur Beschreibung von Mischungen wird die Klasse `Mixture` verwendet. Deren Komponenten, also einzelne Reinstoffe, werden durch die Klasse `ChemicalComponent` repräsentiert. Die atomare Zusammensetzung von Reinstoffen und Mischungen ist in der Klasse `Composition` gekapselt. Die Klasse `Chemistry` stellt Methoden zur Berechnung stöchiometrischer Größen zu Verfügung.

3.3.1 Angabe von Grammatiken in der EBNF-Notation

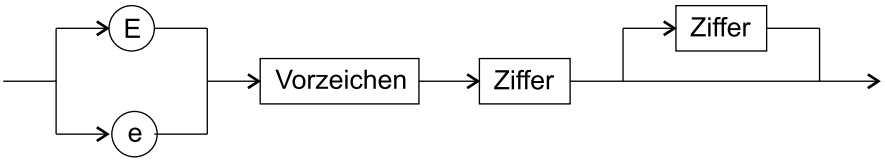
Die meisten Programmiersprachen erkennen in der Eingabe zunächst nur Zeichenketten und Fließkommakonstanten (Dezimalzahlen). Enthalten Zeichenketten Ausdrücke einer

[3]Ein komplexeres Beispiel dazu wird in Abschnitt 3.4.8 behandelt.

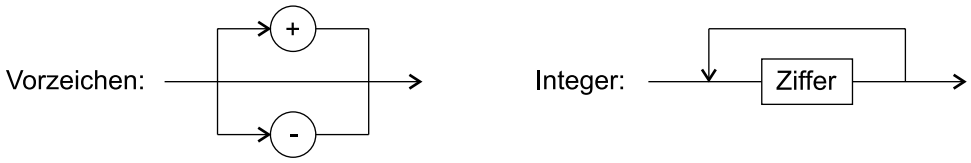
bestimmten Form wie chemische Formeln oder mathematische Terme, so können diese mit Hilfe einer Grammatik syntaktisch analysiert werden. Beispielsweise lässt sich die Grammatik eines Fließkommaliterals[4] in einem Syntaxdiagramm beschreiben:



Ein Fließkommaliteral besteht demnach aus einem ganzzahligen Teil „Integer“ , einem Dezimalpunkt, einem Nachkommateil und einem Exponenten. Das Literal enthält mindestens eine Ziffer entweder im Vor- oder Nachkommateil und einen Dezimalpunkt oder einen Exponenten. Der Exponent besteht aus dem Buchstaben „E“ oder „e“ gefolgt von einem Vorzeichen sowie mindestens einer und höchstens[5] zwei Ziffern:



Ein Vorzeichen besteht aus einem optionalen Plus- oder Minuszeichen. Eine ganze Zahl setzt sich aus einer Ziffer und einer beliebigen[6] Anzahl weiterer Ziffern zusammen:



Nach diesen Diagrammen wären z. B. folgende Fließkommaliterale erlaubt: `3.14159`, `.4`, `2E3` und `1.2E-3`.

Die Diagramme enthalten zwei unterschiedliche syntaktische Elemente: Ausdrücke wie „Integer“ oder „Exponent“ sind in weitere Unterausdrücke zerlegbar und werden als syntaktische Variablen oder Nichtterminalsymbole bezeichnet. Nichtterminalsymbole werden

[4]Diese Darstellung entspricht der Definition in Java nach [13], die auch in anderen Programmiersprachen üblich ist.

[5]in diesem Programm

[6]In der Praxis ist diese Anzahl durch das Speicherformat begrenzt.

als Rechtecke dargestellt. Die als Kreise dargestellten Terminalsymbole treten als solche in einem Ausdruck auf und werden nicht weiter zerlegt.

Eine kompaktere Notation zur Formulierung einer Grammatik bietet die Erweiterte Backus-Naur-Form (EBNF). Ein als Produktion bezeichneter Ausdruck darf folgende syntaktische Elemente enthalten:

=	weist einem Nichtterminalausdruck eine Syntax zu
|	Zeichen für „oder“ (hat niedrigste Priorität)
(...|...)	genau eine Alternative aus der Klammer muss auftreten
[...]	der Inhalt der Klammer darf auftreten oder auch nicht
{...}	der Inhalt der Klammer darf beliebig oft auftreten oder auch gar nicht
.	Ende der Produktion

Terminalsymbole sind in Anführungszeichen eingeschlossen. Aufeinanderfolgende Ausdrücke werden aneinander gereiht. Jede EBNF-Notation ist ebenso als Syntaxdiagramm darstellbar. Hingegen gibt es Syntaxdiagramme, die nicht unmittelbar nach EBNF notiert werden können. Solche Syntaxdiagramme verletzen jedoch zugleich Prinzipien für den sinnvollen Entwurf von Grammatiken. Obige Syntaxdiagramme lauten dann in der EBNF-Notation:

```
Fließkommaliteral = Integer "." [Integer] [Exponent] | Integer Exponent
                    | "." Integer [Exponent].
Vorzeichen        = [ "+" | "-" ].
Integer           = Ziffer {Ziffer}.
Exponent          = "E" Vorzeichen Ziffer [Ziffer].
Ziffer            = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
```

3.3.2 Die Klasse `StringParser`

Diese Klasse wurde zur Untersuchung von Zeichenketten entsprechend einer Grammatik entwickelt. Diese Klasse enthält eine zu untersuchende Zeichenkette und einen Zeiger, an welcher Position in der Zeichenkette sich der Parser gerade befindet. `StringParser` stellt verschiedene Methoden zur Verfügung, das nächste signifikante Zeichen abzufragen, bestimmte Zeichenfolgen einzulesen und zu überprüfen, ob es sich bei einem nachfolgenden Zeichen um einen bestimmten Typ von Zeichen wie z. B. Ziffer, Klammer, Klein- oder Großbuchstabe u. a. handelt.

3.3.3 Die Klasse `ChemicalFormulaParser`

Diese Klasse interpretiert Zeichenketten entsprechend folgender Grammatik:

```
Ausdruck          = Formel {Trennzeichen Formel}.
Formel            = [Fließkommaliteral] Subformel.
Subformel         = (Symbol | ÖffnendeKlammer Subformel SchließendeKlammer)
                    [Fließkommaliteral] {Subformel}.
Symbol            = "Ac" | "Ag" | ... | "Zn" | "Zr" | "Me" | "Et" | "Bu" | "Ph".
```

```
Trennzeichen       = "*" | ",".
ÖffnendeKlammer    = "(" | "[" | "{".
SchließendeKlammer = ")" | "]" | "}".
```

Das oben definierte vorzeichenlose Fließkommaliteral übernimmt in `Formel` die Rolle des Koeffizienten und in `Subformel` diejenige des Indices. `Symbol` sind die in der nachfolgend beschriebenen Klasse `PSE` enthaltenen Symbole chemischer Elemente und zusätzlich die Kürzel `Me` für Methyl, `Et` für Ethyl[7], `Bu` für Butyl und `Ph` für Phenyl.

Die Klasse `ChemicalFormulaParser` bestimmt aus einer Zeichenkette, die eine Formel darstellt, eine Zuordnung der Klasse `Map`, welche die in der Formel enthaltenen Elementsymbole ihrer Anzahl in der Formel zuordnet. Der Klasse wird hierzu im Konstruktor eine Zeichenkette mit dem zu interpretierenden Ausdruck übergeben. Mit Hilfe der Klasse `StringParser` wird diese Zeichenkette in Untereinheiten zerlegt, welche durch Methoden wie `Map parseFormel(StringParser sp)` ausgewertet werden. Die erzeugte `Map` enthält eine Zuordnung der Indices zu den Elementsymbolen. Diese werden in der Klasse `NumberValueMap` ggf. mit weiteren Indices oder Koeffizienten multipliziert und aufsummiert. Mit der Methode `Map getComposition()` kann schließlich die der übergebenen Formel entsprechende Zusammensetzung erhalten werden.

Die Klasse `NumberValueMap` enthält einige Methoden zur Arbeit mit `Maps`, welche einem Schlüsselwert eine Größe vom Typ `Double` oder `Integer` zuordnen. Die Methode erlauben die Erzeugung, Multiplikation und Vereinigung von Einträgen. Lautet beispielsweise ein Ausdruck für Aceton `Me2CO`, so werden zunächst die Zuordnungen `C = 1, H = 3` für `Me` und `C = 1, O = 1` erzeugt. Dann werden die Indices der ersten Zuordnung verdoppelt und mit der zweiten Zuordnung vereinigt.

3.3.4 Die Klasse `PSE`

`PSE` erzeugt und enthält Tabellen mit den chemischen Elementen des Periodensystems. Als Werte sind die Atommassen und Kernladungen der Elemente Nr. 1-103 enthalten, daneben auch Deuterium und Tritium. Für Radioelemente ist als Atommasse die Nuklidmasse eines wichtigen Isotops gespeichert. Als weitere Daten sind Strahlungsenergien in MeV von Übergängen `transition` wichtiger Anodentypen entsprechend der International Tables of Crystallography [30] enthalten. Für die Abfrage dieser Daten stehen statische Methoden zur Verfügung, z. B. `String getSymbol(int atomicNumber)`, `int getAtomicNumber(String symbol)`, `double getAtomicMass(String symbol)` und `getRadiation(String transition)`.

3.3.5 Die Klassen `ChemicalComponent` und `Mixture`

Diese Klassen dienen der Datenverarbeitung von chemischen Substanzen und Mischungen. `Mixture` enthält Angaben zu den enthaltenen Komponenten `ChemicalComponent` und der Dichte `Density`, `ChemicalComponent` setzt sich aus der chemischen Formel `ComponentFormula` und deren Anteil `ComponentContent` zusammen. Die Klassen `Mixture` und `Chemi-`

[7] `Pr` ist nicht Propyl sondern Praseodym und Ac nichtr Acetyl sondern Actinium!

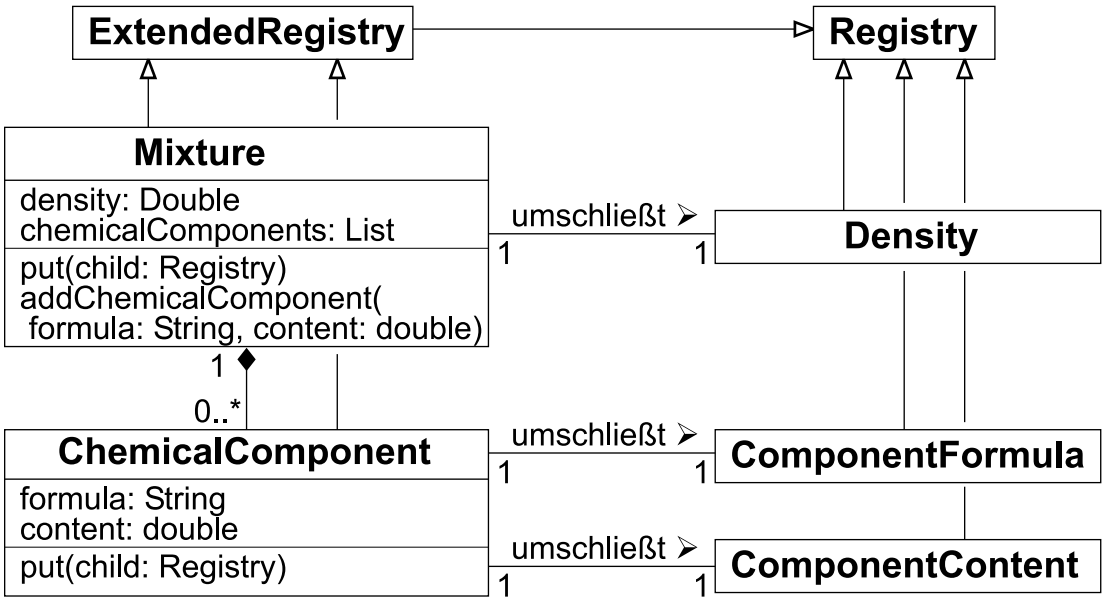


Abbildung 3.8: Die Klassen `ChemicalComponent`, `Mixture` und ihre inneren Klassen

`calComponent` leiten sich von der Klasse `ExtendedRegistry` ab, die ihrerseits von `Registry` abgeleitet ist. Ihre Instanzvariablen sind nicht als Zeichenketten repräsentiert, sondern haben Typen wie `double` oder `Double`[8]. Ein weiterer Grund für die Verwendung von `ExtendedRegistry` liegt darin, dass `Mixture` in einer Liste `chemicalComponents` Referenzen auf ihre einzelnen Komponenten verwaltet. Andernfalls müsste für jeden Zugriff auf eine Mischung die zugehörige `Registry` erneut analysiert werden. Die Nachfahren der Klassen `Mixture` bzw. `ChemicalComponent` werden nicht als Instanzen von `Registry` verwaltet, sondern als Instanzen deren Subklassen `Density`, `ComponentFormula` und `ComponentCount`, welche als innere Klassen enthalten sind. Zur Erkennung der Nachfahren musste die Methode `put(Registry child)` der Klasse `Registry` jeweils überschrieben werden.

3.3.6 Die Klasse `Composition`

Die Klasse `Composition` dient der Beschreibung der Zusammensetzung von Einzelsubstanzen und Mischungen. Sie enthält eine `StableMap`, welche den enthaltenen Elementsymbolen vom Typ `String` die jeweiligen Stoffmengen vom Typ `Double` zuordnet. Diese Klasse ist — ebenso wie `String`, `Double` etc. — unveränderlich, d. h. die Daten einer Instanz können nach der Initialisierung nicht mehr verändert werden, da es hierfür keine öffentlichen Zugriffsmethoden gibt. Die Konstruktoren dieser Klasse erlauben als Argumente eine Mischung vom Typ `Mixture` oder eine Zeichenkette, welche der durch `ChemicalFormulaParser` festgelegten Syntax gehorchen muss. Die Konstruktoraufrufe berechnen die molaren Anteile der atomaren Komponenten. Es bestehen verschiedene Zugriffsmethoden mit Lesezugriff auf enthaltene Datenelemente.

[8] `double` ist ein primitiver Datentyp und `Double` die entsprechende Kapselklasse

3.3.7 Die Klasse `Chemistry`

Mit den statischen Methoden dieser Klasse lassen sich fundamentale Größen wie Molmasse, Molenbrüche, Massenbrüche und Protonenzahl berechnen. Alle Berechnungen operieren auf Objekten der Klasse `Composition`.

3.3.8 Die Klasse `SubstancePanel`

Diese Klasse enthält eine Zeichenebene mit einem Textfeld und einer Auswahlliste, in welches eine neue chemische Formel eingegeben werden kann oder eine bereits definierte Substanz oder Mischung ausgewählt werden kann. Über die Methode `getComposition()` ist die für nachfolgende Berechnungen wichtige Zusammensetzung zugänglich. Diese Klasse wird von zahlreichen der nachfolgend beschriebenen Dialogen verwendet, ohne dass explizit darauf hingewiesen wird.

3.4 Hauptklassen der graphischen Benutzeroberfläche

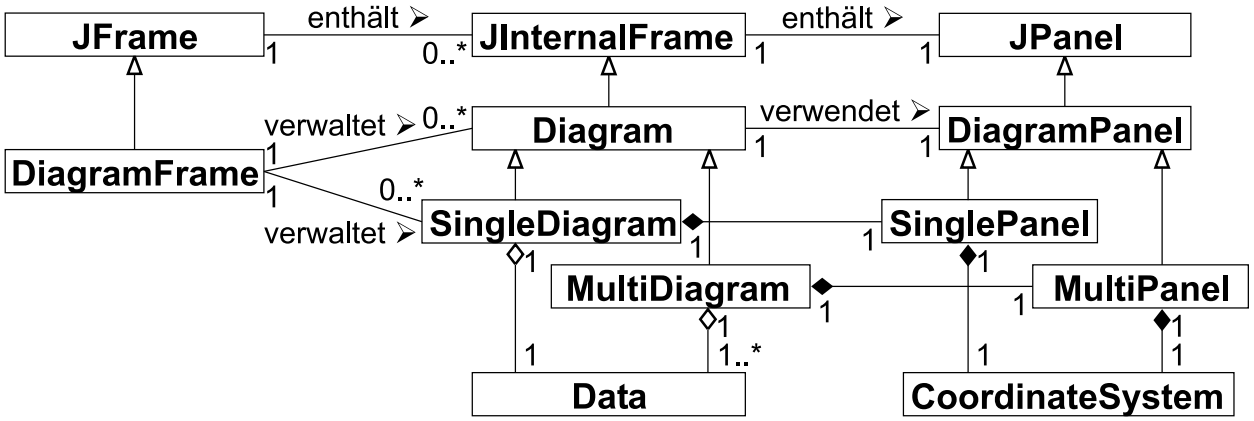


Abbildung 3.9: Übersicht über die Klassen zur Darstellung und Verwaltung von Diagrammen

Die in Abbildung 3.9 gezeigten Klassen spielen die entscheidende Rolle bei der Darstellung und Verwaltung der Diagramme. Man erkennt drei vertikal nebeneinander angeordnete Ebenen:

1. Die von der Klasse `JFrame` abgeleitete Klasse `DiagramFrame` stellt sowohl den äußersten Kontainer der Anwendung dar, in dem sich alle inneren Fenster vom Typ `JInternalFrame` und Dialoge befinden, als auch den funktionellen Rahmen, der für die Verwaltung der Diagramme und Menüs zuständig ist.

2. Von der Klasse `JInternalFrame` leiten sich die inneren Fenster ab, in denen die Diagramme dargestellt sind und die für die Kommunikation mit dem Anwender verantwortlich sind. In dieser Ebene ist auch die Zuordnung von Diagrammen und Datensätzen angesiedelt.

3. Die dritte Ebene leitet sich von der Klasse `JPanel` ab, die eine Zeichenebene ist, welche die graphische Darstellung der Diagramme enthält. Neben den Kurvendaten ist auch das durch `CoordinateSystem` repräsentierte Achsenkreuz enthalten.

Unter Weglassung von Details ist in Abbildung 3.10 auch die in Abschnitt 2.3.7 vorgestellte MVC-Architektur wieder zu erkennen. Ein wesentlicher Teil der Kontrolle findet in den weiter unten beschriebenen Menü- und Dialogklassen statt.

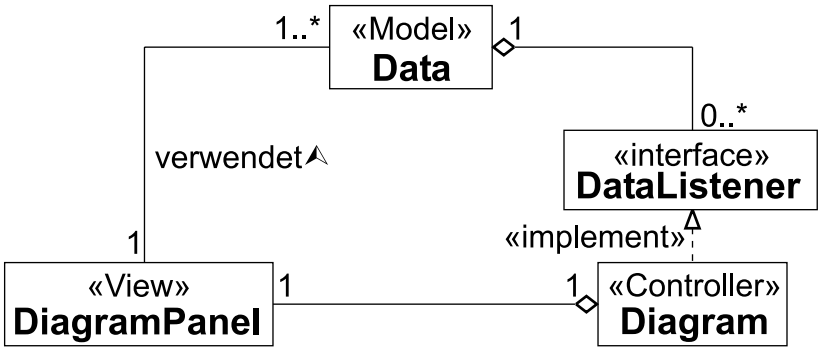


Abbildung 3.10: MVC-Architektur für Diagramme und Datensätze

3.4.1 Die Klasse `DiagramFrame`

Die Klasse `DiagramFrame` dient dem Programm als Hauptfenster, in welchem sich alle weiteren Komponenten wie Diagramme, Menüs und Dialogfenster befinden. Gleichzeitig übernimmt diese Klasse alle zentralen Verwaltungsaufgaben, die beim Hinzufügen, Ändern und Entfernen von Diagrammen und Menüs auftreten, z. B. Kapselung des jeweils aktuellen Diagramms oder Verzeichnisses, Erzeugung von Diagrammnamen usw. Diese Funktionen werden im Zusammenhang mit den jeweiligen Klassen erklärt.

3.4.2 Die Klasse `Diagram`

Die abstrakte Klasse `Diagram` ist von der Klasse `JInternalFrame` abgeleitet. Diagramme sind demnach innere Fenster, in welchen Schaubilder von Daten dargestellt werden. Die Klasse übernimmt die Verwaltung der Ereignisse, die von der Maus ausgehen. Dazu gehören die Anzeige von Koordinaten und die Reskalierung der Diagramme, die von den konkreten Unterklassen ausgeführt wird, welche die Diagramme repräsentieren. Eine abstrakte Druckmethode wird deklariert. Verschiedene Methoden definieren die Markierung von Daten und Datenbereichen, die eigentliche graphische Markierung erfolgt in der Klasse `DiagramPanel`.

3.4.3 Die Klassen `SingleDiagram` und `MultiDiagram`

Ein von `Diagram` abgeleitetes `SingleDiagram` enthält eine einzelne Kurve, während in einem `MultiDiagram` mehrere Kurven gleichzeitig dargestellt werden können. Mit der Methode `MultiDiagram::addSingleDiagram(SingleDiagram sd)` können einem `MultiDiagram`

SingleDiagramme hinzugefügt werden. Beide Klassen implementieren die in Diagram definierte abstrakte Methode setRange(double min, double max), die aus den vorhandenen Daten diejenigen auswählt, die tatsächlich im Diagramm dargestellt werden sollen. Diese Methode wird aufgerufen, wenn der Benutzer mit der Maus einen zu zeichnenden Bereich markiert. Beide Klassen dienen wieder als Kontainer für SinglePanel und MultiPanel.

Die Klassen unterscheiden sich in der Implementierung der in Diagram deklarierten Methoden für Anzeige von Koordinaten, Reskalierung und Ausdruck. Insbesondere unterscheiden sich die beiden Diagrammtypen in ihrer möglichen Behandlung durch den Benutzer. Während MultiDiagram ausschließlich der Darstellung dient, können nur auf dem im SingleDiagram enthaltenen Datensatz Data Transformationen ausgeführt werden. Um die in einem MultiDiagram dargestellten Kurven unterscheiden zu können, kann für diese entweder im Konstruktor oder durch die Methode setDrawModes(List l_drawModes) jeweils ein Zeichenmodus festgelegt werden, welcher durch die Klasse DrawMode definiert ist. Das MultiDiagram enthält ein Menü, in welchem eine Legende dargestellt wird, welches die Bezeichnungen der enthaltenen Kurven und Icons mit deren Zeichenmodus enthält.

3.4.4 Die Klassen DiagramPanel, SinglePanel und MultiPanel

Die von JPanel abgeleitete Klasse DiagramPanel stellt die abstrakte Superklasse der beiden anderen Klassen dar. Während das Aussehen der meisten Benutzeroberflächenelemente durch deren View-Einheit festgelegt ist, enthält die Klasse JPanel nur eine leere Zeichenebene. Ihre Instanzen werden wie alle Komponenten durch Aufruf der Methode paint(Graphics g) gezeichnet. Überschreibt man diese Methode, so können auf den übergebenen graphischen Kontext g Zeichenbefehle, also Methoden der Klasse Graphics, angewandt werden. Mit Einführung der Java Platform 2 hat g den Typ Graphics2D, einer abstrakten Subklasse von Graphics, welche über einen erheblich größeren Methodenumfang verfügt. Gleichzeitig sind die verwendeten Koordinaten nicht mehr vom Typ int, wie er für die Ausgabe auf den Bildschirm mit ganzzahligen Einheiten in Bildschirmpunkten ausreicht, sondern vom Typ float, der zugleich für die Arbeit mit hochauflösenden Druckern geeignet ist.

Die Klasse DiagramPanel arbeitet mit verschiedenen Bereichen, welche in den Klassen DoubleRange und DoubleRange2D gekapselt sind. Diese Klasse definieren ein- bzw. zweidimensionale Intervalle mit Variablen vom Typ double für die Intervallgrenzen sowie 2 Subintervallunterteilungen in jeder Dimension. Zu unterscheiden sind einerseits der Wertebereich dataRange, der sich durch die Methode setRange(...) der Klasse Diagram festgelegt wurde und das Abszissen- und Ordinatenintervall des aktuellen Datenbereiches enthält, und andererseits der etwas größere Darstellungsbereich displayRange, welcher die Intervalle enthält, die dem tatsächlichen Zeichenbereich entsprechen, welcher durch die Bereiche der Achsen definiert ist. Der aktuelle Datenbereich kann vom Benutzer über einen Dialog oder durch Ziehen eines Rahmens mit der Maus geändert werden. Ein dritter Bereich entspricht ebenfalls dem Darstellungsbereich, allerdings nicht in Benutzerkoordinaten, sondern in Koordinaten des graphischen Kontextes Graphics. Der graphische Kontext wird durch eine Zeichenebene vom Typ DiagramPanel, eine Seite für die Druckerausgabe oder die Ausgabe im EPS-Format vorgegeben. Zur Umrechnung der Koordinaten zwischen denjenigen des Darstellungsbereiches displayRange und denjenigen des graphischen Kontextes

dienen die Instanzvariablen `ltx` und `lty` von `DiagramPanel`, welche Lineartransformationen der Klasse `LinearTransformation` enthalten. Diese Klasse definiert einige Methoden zur linearen Koordinatentransformation. Mit Hilfe der Klasse `CoordinateSystem` werden der Darstellungsbereich und die Lineartransformationen berechnet. Die Achsenbereiche werden so bestimmt, dass die Achsenteilstriche bei ganzzahligen Vielfachen von 1, 2 oder 5 in der jeweiligen Größenordnung liegen. Anzahl und Abstand der Achsenteilstriche ergeben sich aus dem für die Achsenbeschriftungen benötigten Platz. Die Berechnung ist im Detail aufwändig und wurde in den genannten Klassen dokumentiert, soll hier aber nicht beschrieben werden.

Die Klasse `DiagramPanel` implementiert einige Methoden zur Markierung von Daten und Datenbereichen. Die Methode `print(Graphics g, PageFormat pf, int pi)` des Interfaces `Printable` wird implementiert, um Diagramme ausdrucken zu können. Sie greift zusammen mit den Methode `paint(...)` und `Diagram::exportEPS(...)` auf eine Vielzahl von Methoden zu, welche verschieden Formatierungen vornehmen und die Graphik schrittweise aufbauen.

3.4.5 Die Klasse `DrawMode`

Diese Klasse definiert Zeichenmodi für Kurven anhand der Attribute `Paint`, `BasicStroke`, einen Symboltyp und dessen Größe. `Paint` und `BasicStroke` sind Java-Klassen zur Charakterisierung einer Fülleigenschaft, hier meist eine Farbe bzw. eine Linienform. `DrawMode` enthält einige vordefinierte Linienformen wie unsichtbare Linie, durchgezogene Linie sowie verschiedene gepunktete und gestrichelte Linien, welche über statische Konstanten verwendet werden können. Weitere statische Konstanten definieren Symbole wie Kreuze, (gefüllte) Kreise und (gefüllte) Quadrate, welche am Ort von Kurvenpunkten mit den Koordinaten `x` und `y` durch Aufruf der Methode `drawSymbol(Graphics2D g2, float x, float y)` gezeichnet werden. `Graphics2D g2` ist dabei der Graphikkontext, welcher die interne graphische Darstellung aller Bildschirmkomponenten enthält. Er wird hier durch die Klassen `SinglePanel` und `MultiPanel` festgelegt. Damit graphische Elemente in `g2` in der gewünschten Art und Weise dargestellt werden, muss der Zeichenmodus mit `DrawMode.applyTo(g2)` eingestellt werden.

3.4.6 Frei positionierbare Diagrammelemente

Diagramme enthalten Darstellungselemente wie Achsen (`AbscissaAxis` und `OrdinateAxis`), deren Bezeichnungen (`AxisTitle`) und die Legende (`DiagramLegend`), welche mit der Maus verschoben werden können. Abbildung 3.11 zeigt ein Multidiagramm mit positionierbaren Diagrammelementen, die durch einen gepunkteten Rahmen graphisch hervorgehoben wurden, der normalerweise nicht sichtbar ist.

Die Zusammenhänge der verwendeten Klassen sind in Abbildung 3.12 dargestellt. Alle Klassen dieser Darstellungselemente leiten sich von der Klasse `FloatingDiagramPanel` ab, welche ihrerseits von der Klasse `JPanel` abgeleitet ist.

`FloatingDiagramPanel` implementiert das Interface `MouseMotionListener`, mit dem

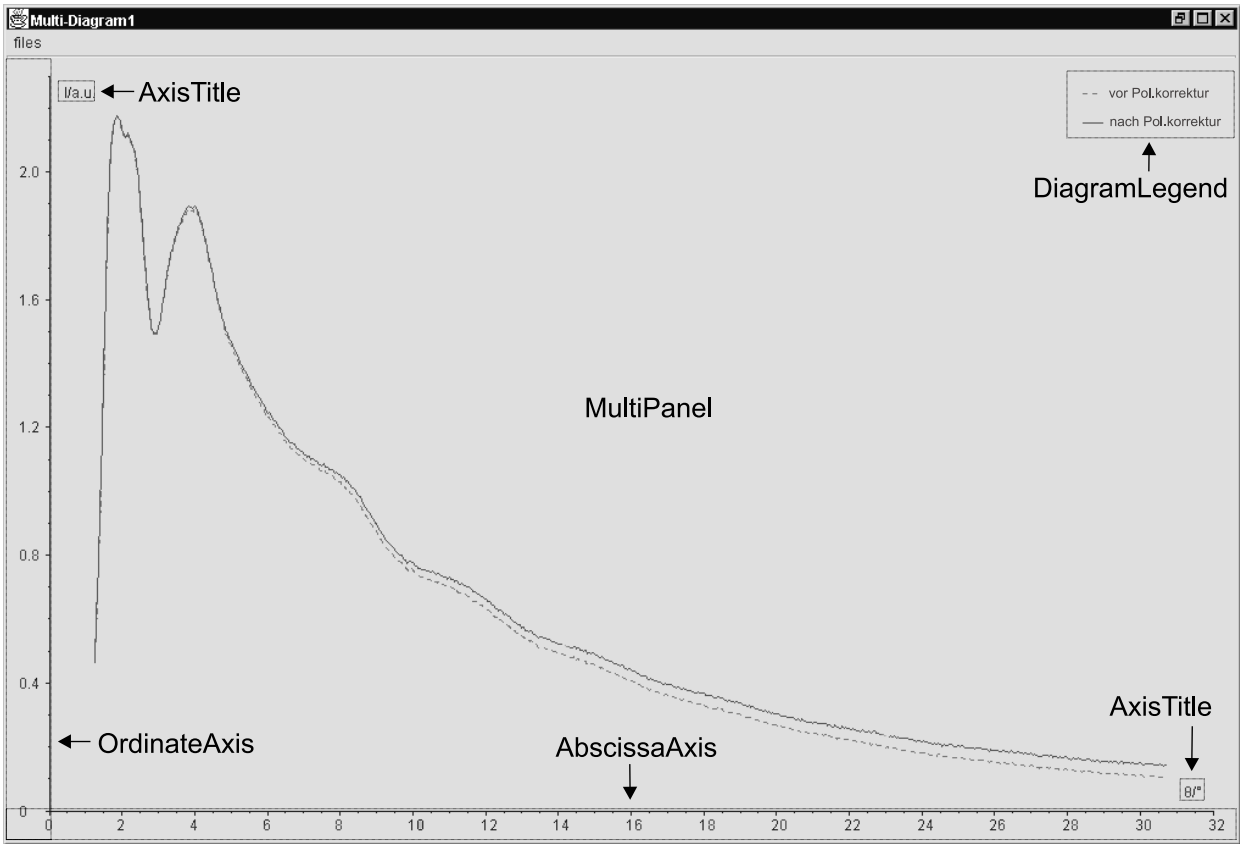


Abbildung 3.11: Multidiagramm mit positionierbaren Diagrammelementen

eine Bewegung der Maus registriert werden kann. Die Klasse `FloatListener`, die sich von der Java-Klasse `MouseAdaper` ableitet, registriert die Auswahl eines Darstellungselementes durch einen Mausklick.

Die Klasse `FloatingDiagramPanel` überschreibt die Methoden `setBounds(...)` und `setLocation(...)` der AWT-Stammklasse `Component`, die von den Layoutmanagern verwendet werden, um Kontainerelemente automatisch zu positionieren. Zur Verhinderung der Positionierung durch Layoutmanager verwendet diese Klasse eigene Positionierungsparameter, welche als Koordinaten bezüglich des Diagramms abgelegt sind. Die Umrechnung in die für die graphische Darstellung erforderlichen Bildschirmkoordinaten erfolgt durch Instanzen der Klasse `LinearTransformation`. Es handelt sich dabei um Referenzen auf die selben Transformationen, mit denen auch die Klasse `DiagramPanel` arbeitet. Das Objekt der Klasse `DiagramPanel` enthält Objekte der Subklassen von `FloatingDiagramPanel`, welche die jeweiligen Diagrammelemente repräsentieren.

Die einzelnen Diagrammelemente unterscheiden sich unter anderem in der Lage ihres Koordinatenursprungs, was durch die Instanzvariablen `xOffset` und `yOffset` berücksichtigt wird. Beispielsweise liegt der Referenzpunkt der Ordinatenachse auf eben dieser, während der Koordinatenursprung ihres `JPanel`s links davon liegt, weil die Achskoordinaten links von der Achse gezeichnet werden. Für die positionierbaren Diagrammelemente kann festgelegt

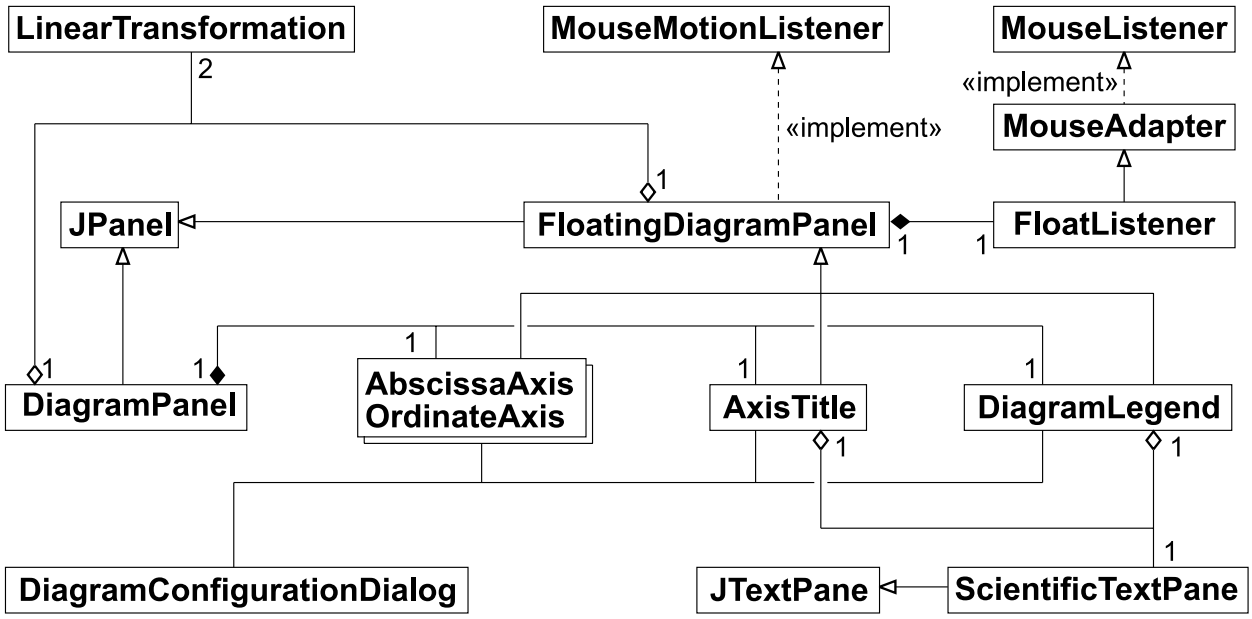


Abbildung 3.12: Verwendung positionierbarer Diagrammelemente

werden, ob sie vom Programmbenutzer tatsächlich verschoben werden können, wobei die Achsen nur orthogonal zu sich selbst verschoben werden können. Wird die Verschiebbarkeit der Diagrammelemente zugelassen, so sind sie als eigenständige GUI-Elemente im Diagramm enthalten, das als Kontainer fungiert. Beim Zeichnen des Diagramms wird dann wie üblich automatisch die Zeichenmethode enthaltener Elemente aufgerufen. Andernfalls existieren die Objekte der entsprechenden Klasse nur in Form von Referenzen, an welche der Graphikkontext explizit zum Zeichnen übergeben werden muss.

3.4.7 Die Klasse ScientificTextPane

Diese Klasse stellt eine Erweiterung der Klasse `JTextPane` dar, welche zur Darstellung formatierten Textes geeignet ist. Sie definiert verschiedene Schriftmodi für normalen Text sowie hoch- und tiefgestellten Text zur Darstellung von Exponenten und Indices. Der darzustellende Text wird einer Instanz als Zeichenkette übergeben, welche Formatierungen enthalten kann, deren Kodierung sich an LaTeX anlehnt: Tiefgestellte Zeichen werden mit einem Unterstrich '_' eingeleitet und hochgestellte Zeichen mit dem Zirkumflex '^'. Sollen mehrere Zeichen hoch- oder tiefgestellt werden, so müssen sie in geschweifte Klammern eingeschlossen werden. So ergibt z. B. `c(C_6H_{12}O_6)` die Zuckerkonzentration $c(C_6H_{12}O_6)$ und `V/m^3` das Volumen in Kubikmetern V/m^3. Da Java mit Unicode arbeitet, können damit auch alle Sonderzeichen dargestellt werden.

Der darzustellende Text wird durch `append(...)`-Methoden aufgebaut, z. B. hängt die Methode `void append(String txt)` formatierten Text an die bestehende oder leere Ausgabe an. Zum Zeichnen der Legende wird die Methode `void append(Component comp)` benötigt, welche z. B. Liniensymbole in die Beschriftung einfügt.

3.4.8 Das Speichern und Laden von Diagrammen

Das Speichern und Laden von Diagrammen erfolgt ausgehend von der Klasse `DiagramIO`. Als Formate stehen das ASCII-Format und das XML-Format zur Verfügung. Im ASCII-Format werden durch die Methode `exportASCII(Diagram dia, String filename)` nur die Abszissen- und Ordinatenwerte in zwei durch ein Tabulatorzeichen getrennten Spalten abgespeichert. Multidiagramme enthalten als Kopfzeile außerdem eine Angabe über die Anzahl an einzelnen Datensätzen und die fortlaufende Nummer des jeweiligen Datensatzes in der Datei. Die einzelnen Datensätze werden durch `exportASCII(Data data, BufferedWriter bw)` in den mit `filename` erzeugten Ausgabestrom geschrieben. Umgekehrt werden ASCII-Dateien mit der Methode `Diagram importASCII(String filename)` wieder eingelesen und daraus ein Diagramm erzeugt. Ein einzelner Datensatz wird über die Methode `Data importASCIIData(String filename)` und die Methode `load_ASCII(String filename, List lx, List ly)` in der Klasse `ASCII_io` eingelesen. Die Methode `Diagram importASCII(...)` erkennt mit `exportASCII(...)` gespeicherte Daten von Multidiagrammen und liest diese mit `List importMultiASCII(String filename)` ein.

Sowohl `load_ASCII(...)` als auch `importMultiASCII(...)` arbeiten intern mit der Klasse `java.io.StreamParser`, welche einen Eingabestrom in Tokens zerlegt. Die Einleseroutine wurde so formuliert, dass Datenformate als gültig erkannt werden, die abwechselnd Dezimalzahlen[9] als Abszissen- und Ordinatenwerte enthalten, die durch Trennzeichen (ASCII-Zeichen 0 bis $32_{(10)}$) getrennt sind. Zeilen, welche nicht diesem Format entsprechen, werden als Kommentarzeilen betrachtet und übersprungen. Dadurch lässt sich eine große Vielfalt an gängigen Datenformaten einlesen.

Für die Speicherung weiterer Informationen, die in den Diagrammen enthalten sind, wird wie für die Serialisierung der Registratur das XML-Format verwendet. Während jedoch die XML-Darstellung der `Registry` eine rekursive Anordnung einer Grundstruktur ist, in denen die Namen der Tags beliebig gewählt werden können und deren Daten vollständig in Attributen enthalten sind, liegt hier eine XML-Struktur mit einer Vielzahl verschiedener Elemente vor. Deren Grammatik lautet:

```
Diagramm:      "<diagram>" (Multi|Kurve) Abszisse Ordinate
               [Einsätze] [Legende] [Sonstige] "</diagram>".
Multi:         "<multi>" Kurve {Kurve} Titel "</multi>".
Kurve:         Daten Graph.
Daten:         "<data>" Titel x-Werte y-Werte Properties "</data>".
Titel:         "<title>" Zeichenkette "</title>".
x-Werte:       "<xValues>" Integer ";" {Double ","} "</xValues>".
y-Werte:       "<yValues>" Integer ";" {Double ","} "</yValues>".
Properties:    "<properties>" Registry "</properties>".
Registry:      "<" Bezeichnung " value = \"" Zeichenkette "\"" ["comment
               = \"" Zeichenkette "\""] ">" Registry "</" Bezeichnung ">".
Graph:         "<graph>" Farbe Linie Symbol "</graph>".
Farbe:         "<color red=\"" Farbwert "\" green = \""
               Farbwert "\" blue=\"" Farbwert "\"/>".
Linie:         "<line style=\"" Linienstil "\" width=\"" Linienbreite "\"/>".
Symbol:        "<symbol type=\"" Symboltyp "\" size=\"" Symbolgröße "\"/>".
```

[9] in englischer Schreibweise mit Dezimalpunkt

```
Abszisse:      "<abscissa>" [Achsenbezeichnung] [Anzeigeformat] "</abscissa>".
Ordinate:      "<ordinate>" [Achsenbezeichnung] [Anzeigeformat] "</ordinate>".
Achsenbez.:    "<title x=\"" Double "\" y=\"" Double "\">"
               Zeichenkette "</title>".
Anzeigeformat: "<display>" Darstellungsbereich Achsenabschnitt "</display>".
Darstellungs.: "<range min=\"" Double "\" max=\"" Double "\" primaryInterval
               =\"" Double "\" secondaryInterval=\"" Double "\"/>".
Achsenabs.:    "<intersection value=\"" Double "\"/>.
Einsätze:      <insets left=\"" Integer "\" right=\"" Integer "\" top=\""
               Integer "\" bottom=\"" Integer "\"/>.
Legende:       "<legend" [" x=\"" Double "\" y=\"" Double "\""] "/>".
Sonstige:      ["<printborder/>"] ["<floatsFixed/>"].
```

In dieser Grammatik stehe die Escape-Sequenz \" für ein Anführungszeichen innerhalb von Anführungszeichen eines Terminalelementes, die beiden Ausdrücke `Bezeichnung` im Ausdruck `Registry` seien identische Zeichenketten, `Integer` wie in Abschnitt 3.3.1 definiert und `Double` ein `Fließkommaliteral` doppelter Genauigkeit. `Farbwert` ist eine ganze Zahl im Intervall `[0; 255]`, welche den Wert einer Farbkomponente in der RGB-Darstellung repräsentiert. `Linienstil` und `Symboltyp` sind in der Klasse `DrawMode` definierte Ganzzahlkonstanten und `Linienbreite` und `Symbolgröße` sind Fließkommakonstanten mit der Genauigkeit des Typs `float`. Im Gegensatz zur oben beschriebenen XML-Darstellung der `Registry` sind hier auch Daten zwischen den Tags eingeschlossen. Die Struktur der Registratur ist erkennbar in diejenige der Diagramme eingebettet.

Die Speicherung der Daten im XML-Format ist vergleichsweise einfach: Die Klassen, welche über die zugehörigen Informationen verfügen, implementieren die Methode `String getXMLString(int indentLevel)` und bauen den XML-Code unter Verwendung einer gepufferten Zeichenkette der Klasse `StringBuffer` schrittweise durch rekursiven Aufruf nachgeordneter Objekte auf. Es sind dies die Klassen `SingleDiagram`, `MultiDiagram`, `DiagramPanel`, `Data`, `DrawMode` und `Registry`.

Wesentlich komplizierter ist die Rekonstruktion formatierter Diagramme aus den XML-Daten durch die Klasse `DiagramParser`. Das Grundprinzip entspricht wieder demjenigen, welches auch für das Einlesen der Registratur angewandt wurde. Zusätzlich werden jedoch Klassen benötigt, welche die oben angegebene Grammatik interpretieren. Abbildung 3.13 zeigt, wie sich die Grammatik in eine Klassenstruktur übertragen lässt. Ausgehend von der Klasse `DiagramIO` wird die statische Methode `Diagram getDiagram(InputStream in)` in `DiagramParser` aufgerufen, welche eine Instanz dieser Klasse erzeugt und den Interpretationsvorgang einleitet. Diese Klasse umschließt — mit Ausnahme von `ERegistry` — die inneren „`EKlassen`“, welche mit `E` (für Expression) beginnen und sich von `XMLExpression` ableiten. Sie überschreiben die in Abbildung 3.7 aufgeführten Methoden. Die einzelnen Klassen sind durch Aggregationsbeziehungen entsprechend obiger Grammatik verbunden und übernehmen die Interpretation der nachfolgend vom `SAXParser` ausgehenden Tags, Attribute und Daten.

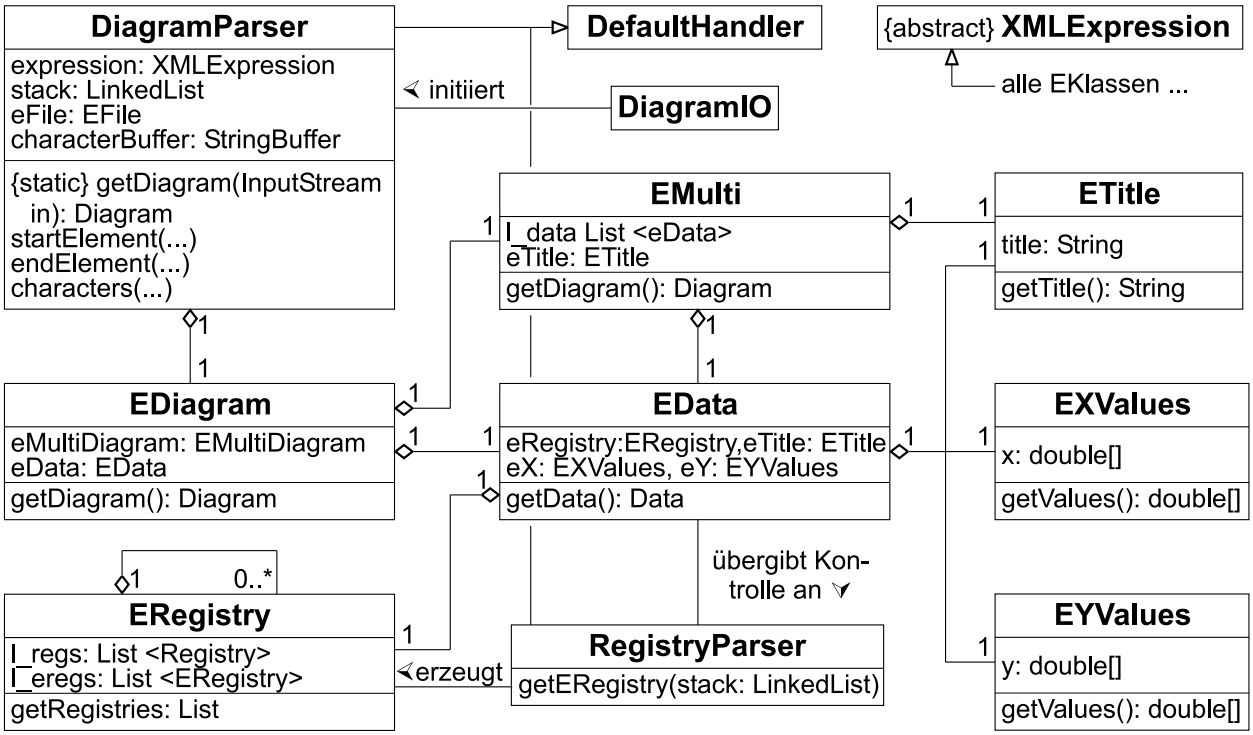


Abbildung 3.13: Interpreter zum Laden von Diagrammen (Ausschnitt)

Jeder `EKlasse` entspricht ein Tag in der XML-Grammatik und jede dieser Klassen gibt in der Methode `XMLExpression interpret(String key)` eine Instanz derjenigen Klassen zurück, deren Tags der Grammatik entsprechend nachfolgend erlaubt sind. Beispielsweise ist aus der Produktion `Multi` ersichtlich, dass auf das Tag `<multi>`, welches der Klasse `EMulti` entspricht, nur die Tags `<data>`, `<title>` und `<graph>` folgen dürfen, die in den gleichnamigen Ausdrücken stehen. Entsprechend sind die Rückgabeobjekte der Methode `EMulti::interpret(...)` Instanzen der Klassen `EData`, `ETitle` und `EDrawMode`, welche die weitere Interpretation übernehmen. Da ein Multidiagramm mehrere Datensätze enthält, werden die entsprechenden Objekte der Klasse `EData` vom Objekt der Klasse `EMulti` referenziert. Die weiteren Methoden `setData(...)` und `setAttributes(...)` der abstrakten Superklasse `XMLExpression` dienen in der jeweiligen Implementierung der `EKlassen` der Übergabe von Daten und Attributen, die dort in geeigneter Weise verarbeitet werden.

Tritt das Tag `<properties>` auf, so wird die Kontrolle an die schon oben dokumentierte Klasse `ERegistry` übergeben, von der eine Instanz über die statische Methode `getERegistry(...)` der Klasse `RegistryParser` zugänglich ist. Hier befindet sich also die Schnittstelle für die Kooperation der beiden Parser.

Wenn der Interpretationsvorgang beendet ist, wird über die Methode `getDiagram()` der Klasse `DiagramParser` eine weitere Rekursion eingeleitet, in welcher aus den `E`-Objekten die zugehörigen Datenobjekte erzeugt und verknüpft werden. Zur vollständigen Interpretation der XML-Datei werden 12 weitere Klassen benötigt, welche den Produktionen der obigen Grammatik entsprechen und in der Abbildung aus Platzgründen nicht enthalten sind.

In dieser Arbeit werden zwei Ansätze verwendet, um Datenstrukturen zu parsen. Der eine Ansatz wird zur Analyse von chemischen und mathematischen Formelausdrücken verwendet und basiert auf einer prozeduralen Vorgehensweise. Der andere Ansatz dient der Interpretation von XML-Daten und basiert auf dem objektorientierten Konzept. Während ersterer leichter zu verstehen und anzuwenden ist, bietet letzterer eine größere Flexibilität und verfügt über eine klarere Strukturierung bezüglich der Umsetzung einer Grammatik. Die Verwendung des Interfaces `ContentHandler` durch den `SAXParser` erzwingt zudem das objektorientierte Konzept dadurch, dass der Interpreter immer über eine einheitliche Schnittstelle angesteuert wird, welche der prozedurale Ansatz nicht bietet.

3.5 Dialoge, Menüs und die Symbolleiste

Wird ein Java-Programm mit Benutzeroberfläche gestartet, so wird zunächst der Bildschirmaufbau vorgenommen, d. h. die in Abschnitt 2.3.7 beschriebene Kontainerhierarchie erzeugt, die in den zugehörigen Klassen enthaltenen Daten initialisiert und die `paint()`-Methode darzustellender Komponenten rekursiv aufgerufen. Diese Methodenaufrufe erfolgen vom Hauptthread ausgehend. Sind alle diese Methoden abgearbeitet, so befindet sich das Programm in einem Wartezustand. Erst wenn vom Betriebssystem ein Ereignis eintrifft, wird ein neuer Thread, der Ereignisbehandlungsthread (Event dispatching thread), gestartet. In den meisten Fällen handelt es sich dabei um einen Mausklick oder eine Tastatureingabe. Die Kontainerklassen ordnen den Koordinaten, an denen der Mausklick stattgefunden hat, durch rekursiven Abstieg diejenige Komponente zu, die sich an dieser Stelle befindet. Für diese Komponente wird die Ereignisbehandlung durchgeführt, sofern für sie ein Ereignisbehandler registriert wurde. Wie im Abschnitt 2.3.7 ausgeführt wurde, existiert für die meisten Eingabeelemente die Methode `addActionListener(ActionListener al)`, der ein Objekt `al` übergeben wird, dessen Klasse die Methode `actionPerformed(ActionEvent evt)` des Interfaces `ActionListener` implementiert. Aus dieser Methode heraus können dann weitere Methoden aufgerufen werden, welche die Ereignisbehandlung durchführen und z. B. Berechnungen einleiten.

Die wichtigsten Komponenten mit denen der Benutzer über die Maus in Interaktion mit dem Programm tritt, sind Menüs. Im Verlauf der durch sie eingeleiteten Interaktionsmechanismen müssen häufig Daten eingegeben werden. Diese Eingaben erfolgen über Dialoge.

3.5.1 Aktivierung von Komponenten

In einem kommandozeilenorientierten Programm wird der Anwender durch verschiedene Menüs geführt, die ihm die zu diesem Bearbeitungszeitpunkt verfügbaren Operationen zur Auswahl bieten. Bei einer Benutzeroberfläche hingegen liegt normalerweise eine feststehende Anordnung von Komponenten wie Menüs und Schaltflächen vor. Es ist jedoch nicht zu jedem Zeitpunkt möglich, auf alle Operationen zuzugreifen, die diesen Komponenten zugeordnet sind. Beispielsweise muss zunächst ein Diagramm vorliegen, bevor es ausgedruckt werden kann und eine Fouriertransformation kann nicht auf ein Diagramm mit mehreren

Datensätzen angewandt werden. Daher können Komponenten aktiviert (enabled) und desaktiviert werden. Es liegen demnach zum einen Komponenten mit den beiden möglichen Zuständen aktiv und inaktiv vor und zum anderen Bedingungen, die erfüllt sein müssen, damit diese Komponenten aktiv sind. Mit wachsender Anzahl an Komponenten und Bedingungen entsteht schnell ein unübersichtliches Beziehungsgeflecht. Zur Verwaltung dieser Beziehungen wurde daher eine Klassenstruktur entwickelt, die in Abbildung 3.14 dargestellt ist.

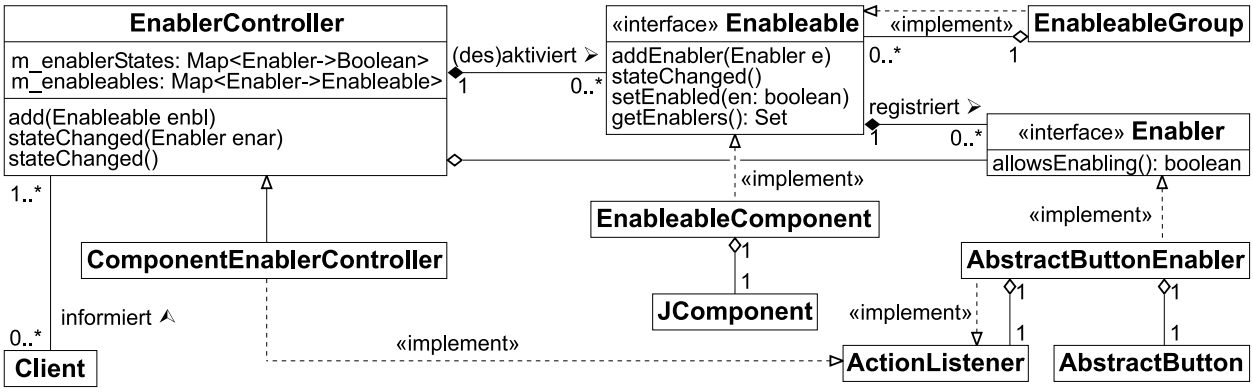


Abbildung 3.14: Aktivierung von Komponenten

Die grundlegende Struktur dieser Klassen besteht aus den Interfaces `Enableable` und `Enabler` und der Klasse `EnablerController`, die auf Objekten vom Typ dieser Interfaces operiert. Das Interface `Enableable` definiert dabei eine zu aktivierende Komponente, für welche eine oder mehrere Bedingungen registriert werden können, die durch die Methode `boolean allowsEnabling()` des Interfaces `Enabler` definiert sind. Eine Instanz der Klasse `EnablerController` verwaltet die `Enabler`-Objekte und die `Enableable`-Objekte, die den Bedingungen der `Enabler` zugeordnet sind. Erfährt das Kontrollobjekt über die Methode `stateChanged(Enabler enar)` von einem Klienten `Client` über eine Zustandsänderung einer bestimmten Bedingung, so werden die dieser Bedingung zugeordneten Objekte vom Typ `Enableable` durch Aufruf deren Methode `stateChanged()` informiert. Haben sich z. B. durch den Wechsel des aktuellen Diagrammtyps mehrere Bedingungen geändert oder weiß der Klient nur, dass sich Bedingungen geändert haben könnten, aber nicht welche dies sind, so kann er die Methode `EnablerController::stateChanged()` aufrufen. Diese Konstruktion ist soweit noch völlig allgemein und unabhängig von Klassen der Java-Bibliothek.

Bei der konkreten Anwendung auf Benutzeroberflächen spielen in diesem Zusammenhang zwei Klassen eine besondere Rolle: Die Klasse `JComponent` als Stammklasse aller Swing-Komponenten und die Klasse `AbstractButton` als Stammklasse vieler Schaltelemente wie `JButton`, `JMenuItem`, `JCheckBox` und `JRadioButton`. Die Klasse `JComponent` verfügt zwar über die Methode `setEnabled(boolean en)`, implementiert jedoch nicht das Interface `Enableable`. Dazu wurde die Klasse `EnableableComponent` eingeführt, welche eine Referenz auf ein Objekt vom Typ `JComponent` kapselt. In gleicher Weise wurde für `AbstractButton` die Kapselklasse `AbstractButtonEnabler` formuliert, die auf die Methode `boolean Ab-`

`stractButton::isSelected()` zugreifen kann. Die Klasse `AbstractButtonEnabler` spielt jedoch zugleich eine weitere Rolle, nämlich diejenige des Klienten. Wird nämlich z. B. eine als `Enabler` registrierte `JCheckBox` angeklickt, so ändert sich damit gleichzeitig ein Zustand, von dem das Kontrollobjekt informiert werden muss. Die Übermittlung dieser Information erfolgt über ein `ActionEvent`, das zunächst vom `AbstractButtonEnabler` abgefangen und dann an ein Objekt vom Typ `ComponentEnablerController` weitergeleitet wird, dessen Klasse sich von `EnablerController` ableitet.

In der Praxis ist es schließlich eine Vereinfachung, mehrere `Enableable`-Objekte zu einer `EnableableGroup` zusammenzufassen, welche selbst wieder dieses Interface implementiert. Bei der Zusammenfassung von Objekten zu einer `ButtonGroup`[10] muss beachtet werden, dass nur ein Objekt einen Zustand repräsentiert, jedoch alle Objekte Klient sind. Daher muss für sämtliche dieser Klienten das jeweilige Objekt der Klasse `AbstractButtonEnabler` als `ActionListener` registriert werden.

3.5.2 Dialoge

Für die Arbeit mit Dialogen unter Java bei Verwendung der Swing-GUI[11] steht die Klasse `JDialog` zur Verfügung, die für eigene Dialoge abgeleitet werden kann. Für einfache Abfragen wie *Yes/No* oder *OK/Cancel* stehen mit der Klasse `JOptionPane` konfigurierbare Standarddialoge bereit. Dialoge sind ebenso wie Fenster Komponenten, die den äußersten Rahmen für weitere Komponenten darstellen (sogenannte Top-Level-Komponenten). Man unterscheidet modale und nicht-modale Dialoge, wobei hier nur modale Dialoge zur Anwendung kommen. Diese sind dadurch gekennzeichnet, dass eine Methode, die einen modalen Dialog erzeugt und mit der Methode `setVisible(true)` angezeigt wird, in ihrer Ausführung unterbrochen wird, bis der Dialog wieder durch eine Methode innerhalb des Dialogs mit `setVisible(false)` geschlossen wird. Gleichzeitig werden Ereignisse für das zugeordnete Fenster und darin eingebettete Komponenten blockiert, d. h. der Benutzer kann nur noch über diesen Dialog und seine nachgeordneten Komponenten mit dem Programm kommunizieren.

Dialoge sollen für den Benutzer eine möglichst einfache und übersichtliche Schnittstelle zur Kontrolle und Eingabe von Parametern bieten. Gleichzeitig verbirgt sich hinter ihrer Benutzeroberfläche eine Vielzahl von Aufgaben, die sich aus Zusammenhängen zwischen mehreren Eingabekomponenten ergeben. Dialogklassen stellen dadurch einen großen Anteil am Programmtext einer Benutzeroberfläche. Daher soll als Beispiel ein Dialog zur Eingabe der Formatierungsparameter für Diagramme vorgestellt werden.

Abbildung 3.15 zeigt die Ansicht eines `DiagramConfigurationDialog`s. Die Klasse dieses Dialogs leitet sich von `StandardDialog` ab. Dem Konstruktor der Klasse werden Informationen über das zu konfigurierende `Diagram`, das Fenster, in dem der Dialog angezeigt werden soll, und die zugehörige Hilfe `Help` (siehe Abschnitt 3.6) übergeben. Die momentanen Darstellungsparameter werden vor Anzeige des Dialogs in einer tiefen Kopie gesichert, falls

[10] Diese Klasse definiert eine Gruppe von Schaltelementen, von denen jeweils nur eines aktiv sein kann
[11] unter AWT existieren entsprechende Komponenten

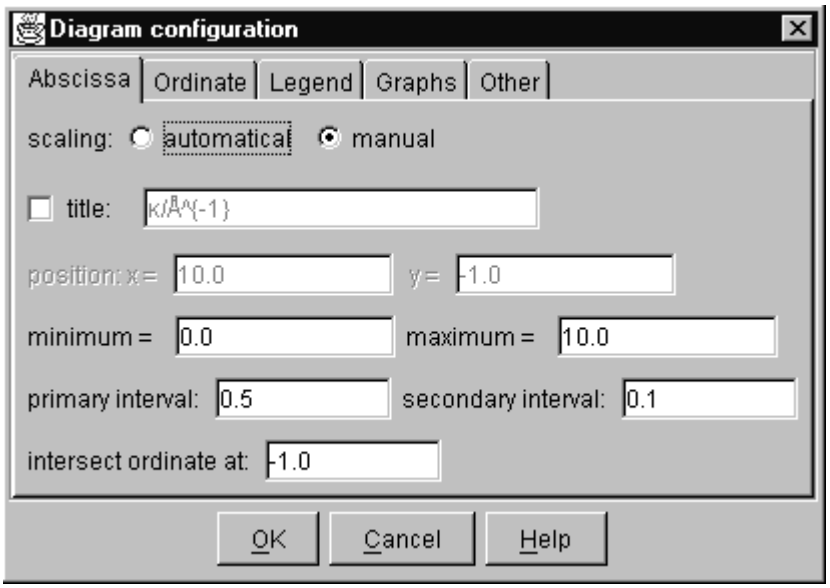


Abbildung 3.15: Dialog der Klasse `DiagramConfigurationDialog`, erste Ansicht

der Benutzer den Dialog mit *Cancel* abbricht und die von ihm vorgenommenen Änderungen daher nicht übernommen werden sollen.

Im nächsten Schritt wird das Aussehen des Dialogs festgelegt. Dazu werden die als Instanzvariablen enthaltenen Eingabeelemente wie Textfelder vom Typ `JTextField`, Kontrollkästchen vom Typ `JCheckBox` oder Radioknöpfe vom Typ `JRadioButton` in die Zeichenfläche vom Typ `JPanel` eingebettet und dort durch Layout-Manager positioniert. Falls zu den einzugebenden Parametern bereits Daten vorliegen, werden diese als Vorgabewerte eingetragen. Die Bedingungen für die Aktivierung von Eingabekomponenten werden festgelegt und mit den oben beschriebenen Klassen `EnableableComponent` und `AbstractButtonEnabler` kodiert. Beispielsweise dürfen die Eingabefelder und die zugehörigen Beschriftungen für die Eingabe der Position der Achsenbeschriftung nur dann aktiv sein, wenn sowohl der manuelle Skalierungsmodus gewählt wurde als auch eine Beschriftung angezeigt werden soll. Änderungen solcher Zustände werden durch Registrierung mit einem `ActionListener` detektiert.

Abbildung 3.16 zeigt eine weitere Ansicht eines Dialogs der Klasse `DiagramConfigurationDialog` zur Konfiguration der in einem Diagramm dargestellten Graphen. Die unterschiedlichen Ansichten dieses Dialogs rühren von einer Aufteilung der Eingabeelemente auf mehrere Reiterkarten vom Typ `JTabbedPane` her. Wird ein Multidiagramm konfiguriert, so stehen in einer Auswahlliste vom Typ `JComboBox` mehrere Graphen zur Auswahl, die konfigurierbar sind. Wählt der Benutzer einen bestimmten Graphen aus, so werden die zugehörigen Parameter in der Anzeige aktualisiert. Andere Dialoge enthalten zum Teil Berechnungen, die durch Auswahl von Schaltflächen eingeleitet werden und ggf. zu einer Veränderung des Diagramms führen.

Falls der Benutzer den Dialog mit *Cancel* abbricht, wird der ursprüngliche Parameter-

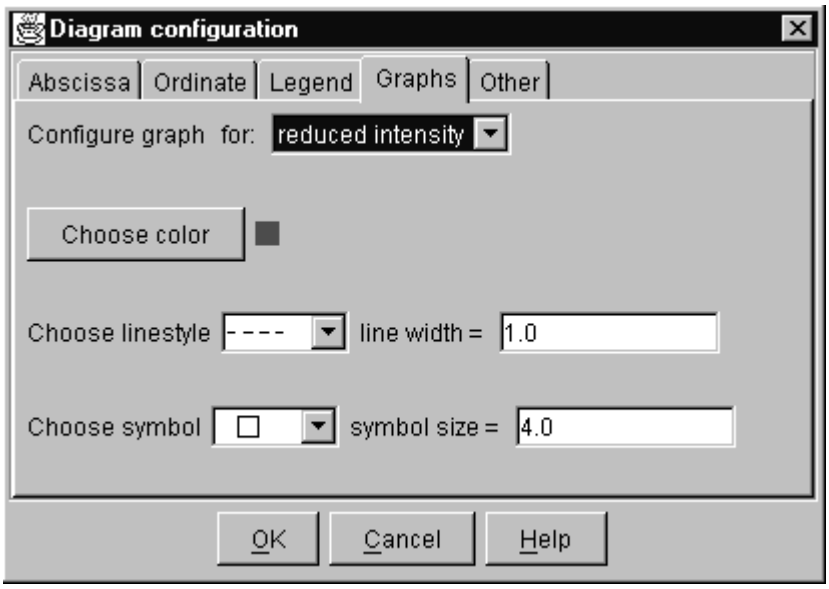


Abbildung 3.16: Dialog der Klasse `DiagramConfigurationDialog`, zweite Ansicht

satz mit Hilfe der oben erzeugten Kopie wieder hergestellt. Beendet er seine Eingaben durch Drücken der Schaltfläche *OK*, so werden zunächst alle Eingaben auf Plausibilität überprüft. Dabei muss unter anderem sichergestellt werden, dass Eingabefelder für numerische Werte tatsächlich Zahlen enthalten. Danach werden sinnlose Eingaben wie z. B. negative Linienbreiten registriert. Falls falsche Eingaben vorliegen, wird der Benutzer mit einem Hilfsdialog darauf aufmerksam gemacht. Sobald alle Eingaben formal richtig sind, werden sie in den aktuellen Parametersatz übernommen und der Dialog wird geschlossen. Die Auswahl der Schaltfläche *Help* führt zur Anzeige des Hilfedialogs, der in Abbildung 3.17 dargestellt ist.

Die Klasse `ParameterDialog`

In vielen Fällen haben Dialoge zur Eingabe von Parametern eine ähnliche Struktur. Dazu gehören eine einleitende Zeile zur Beschreibung und eine Tabelle, die auf der linken Seite eine Beschreibung der Parameter und auf der rechten Seite Eingabefelder für die Parameter, optional mit Vorgabewerten, enthält. Der Dialog schließt ab mit einer Leiste der Schaltflächen *OK*, *Cancel* und ggf. *Help*. Für solche Zwecke wurde die abstrakte Klasse `ParameterDialog` entwickelt, die von Unterklassen genauer spezifiziert wird. Im Konstruktor `ParameterDialog(Frame frame, String title, String kopfzeilen, String ausgabe, String vorgabe, int maxlength)` werden Zeichenketten für Kopfzeilen, Ausgabe (Beschreibung der Parameter) und Vorgabewerte übergeben. Die Zeichenketten können durch das Trennzeichen `'\n'` unterteilt sein, wodurch sie in mehrere Zeilen für die Darstellung zerlegt wird. Die Klasse enthält die abstrakte Methode `checkInput()`, in welcher die Eingabe in den Eingabefeldern überprüft wird. Die Eingabe ist gültig abgeschlossen, falls nach dem Drücken der Schaltfläche *OK* durch die Methode `checkInput()` keine Fehler festgestellt wurden. In diesem Fall wird der Inhalt der Eingabefelder in ein Array übertragen und kann aus diesem mit den Methoden `String[] getInput()` und `String getInput(int i)` ausgelesen werden. Eine den Dialog erzeugende Methode kann die Gültigkeit der Eingabe

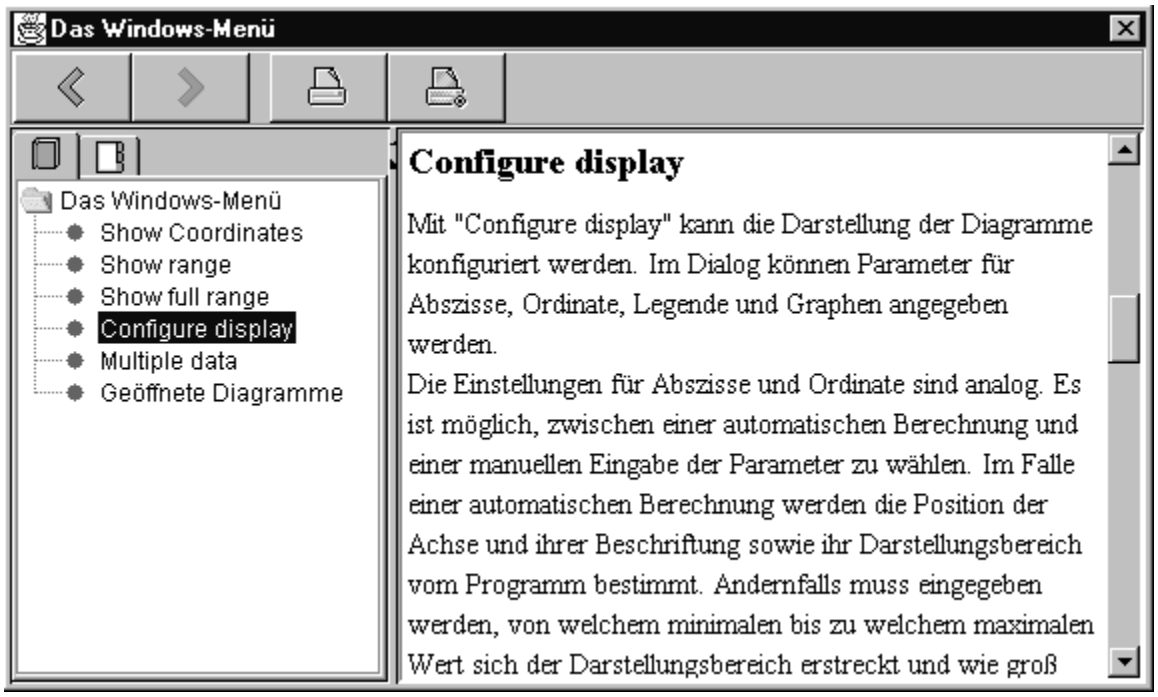


Abbildung 3.17: Bildschirmausdruck eines Hilfedialogs

mit der Methode `boolean hasValidInput()` abgefragen und geeignet auf diese Information reagieren. Abbildung 5.12 zeigt ein Beispiel für einen solchen Dialog.

Die Klasse `StandardDialog`

`StandardDialog` ist eine abstrakte Klasse mit einigen Methoden, die häufig benötigt werden. Die meisten der nachfolgend vorgestellten Dialoge[12] leiten sich von dieser Klasse ab. Die von Subklassen zu erzeugenden Eingabeelemente können mit der Methode `addComponent(Component comp)` in das zugleich über `getCenterPanel()` zugängliche `centerPanel` eingefügt werden, dessen Komponenten voreingestellt durch ein `BoxLayout` angeordnet werden. Mit der Methode `JPanel getFlowPanel()` erhält man eine konfigurierbare Eingabezeile mit `FlowLayout`, die an die bestehende Dialogausgabe angehängt wird. Nach der Festlegung der Ein- und Ausgabelemente muss die Methode `setVisible(true)` aufgerufen werden. Die Eingabe wird nach dem Drücken der Schaltfläche *OK* durch Aufruf der zu überschreibenden Methode `boolean checkInput()` überprüft.

Implementiert die Subklasse einen ActionListener, so muss in `actionPerformed(ActionEvent evt)` auch die gleiche Methode `StandardDialog::actionPerformed(...)` dieser Klasse aufgerufen werden. Die Schaltfläche *Help* wird nur angezeigt, wenn dem Konstruktor ein Objekt der Klasse `Help` übergeben wurde.

[12] z. B. in Abbildung 4.3

Die Klasse `ListDialog`

Ein `ListDialog` zeigt eine Liste von Objekten als Zeichenketten an, die sich durch Aufruf der Methode `Object::toString()` für diese Objekte ergeben und die dem Dialog im Konstruktor als `Vector` übergeben wurden. Der Benutzer kann aus dieser Liste ein oder mehrere Objekte auswählen. Die gewählten Objekte können mit `Vector getSelectedItems()` oder `int[] getIndizes()` abgefragt werden.

Die Klasse `OptionsDialog`

Mit Hilfe dieses Dialogs kann eine Vielzahl von Eingaben durchgeführt werden. Der Dialog zeigt eine zweispaltige Tabelle an, wobei die linke Spalte die Beschreibung eines Ausdrucks enthält, der in der rechten Spalte eingegeben werden soll. Die Richtigkeit der Eingabe wird mit Hilfe der Instanz einer Klasse überprüft, welche das Interface Checker implementiert und dem Dialog im Konstruktor übergeben wird. Es ist möglich, den Benutzer die Zahl der Eingabezeilen verändern zu lassen, indem die Schaltflächen *Insert row* und *Delete last row* eingefügt werden.

Das Interface `Checker` definiert die Methode `checkInput(String eingabe)`, mit welcher eingegebene Zeichenketten anhand in implementierenden Klassen zu definierender Kriterien überprüft werden.

Die Klassen `TextInfoDialog` und `PrintableTextPane`

Die Klasse `TextInfoDialog` dient der Anzeige einer Textseite, die in einem scrollbaren Bereich dargestellt wird. Die Textseite kann editiert und ausgedruckt werden. Sie ist eine Instanz der Klasse `PrintableTextPane`, die sich von der Klasse `JTextPane` ableitet und die Methode `int print(Graphics g, PageFormat pf, int pi)` des Interfaces `Printable` implementiert, über welches ein Ausdruck erfolgen kann.

3.5.3 Menüs

Menüs stellen die wichtigste Möglichkeit dar, Benutzeraktionen einzuleiten. Alle Menüs leiten sich von der Klasse `DiagramMenu` ab, welche ihrerseits von `JMenu` abgeleitet wurde. Ein Menü besteht aus einem oder mehreren Menüeinträgen, die durch Trennlinien gruppiert werden. Mehrere Menüs werden zu einer Menüleiste der Klasse `JMenuBar` zusammengefasst, die in diesem Programm von der Klasse `DiagramFrame` verwaltet wird.

Um eine flexible Verwendung von Menüs durch verschiedene Hauptprogramme zu ermöglichen, umfassen die Menüs neben der Struktur für die Darstellung gleichzeitig die mit den enthaltenen Menüeinträgen verbundene Funktionalität. Eine Ausnahme bilden die Klassen `DiagramFileMenu` und `DiagramWindowsMenu`, die fester Bestandteil der Klasse `DiagramFrame` sind, welche die entsprechenden Funktionen implementiert.

Jedes Menü umfasst eine Gruppe zusammengehöriger Operationen, deren Erläuterungen auch im Hilfesystem (siehe Abschnitt 3.6) zu entsprechenden Einheiten zusammengefasst sind. Daher ist jedem Menü durch eine Instanz der Klasse `Help` der zugehörige Teil des

Hilfesystems zugeordnet.

Die Menüklassen besitzen eine einheitliche Struktur: Ihre Einträge sind Instanzen der Klasse `DiagramMenuItem` oder `DiagramCheckBoxMenuItem`. Der graphische Aufbau wird in der Methode `void initGUI()` festgelegt, welche vom Konstruktor aufgerufen wird. Über die Methode `actionPerformed(...)` wird die Ereignisbehandlung für die einzelnen Menüeinträge eingeleitet, welchen jeweils eine eigene Methode zugeordnet ist, z. B. ruft diese Methode nach Auswahl des Menüeintrags *Kendig Pings* die Methode `kendigPings()` auf. Viele dieser Methoden überprüfen zunächst, ob über die allgemeinen Bedingungen (Single-Diagramm vorhanden, Datensatz geordnet, Abszissenwerte äquidistant) hinaus speziellere Bedingungen vorliegen müssen wie z. B. das Vorhandensein weiterer Datensätze oder ein bestimmtes Datenformat für die Fouriertransformation. Anschließend werden für Benutzereingaben die jeweiligen Dialoge erzeugt, denen über den Konstruktor die Daten des jeweils aktiven Diagramms und sie betreffende Teile der Registratur und des Hilfesystems übergeben werden. Führt der Dialog nur Berechnungen durch, die dem Benutzer direkt angezeigt werden, oder wird der Dialog mit *Cancel* abgebrochen, so wird die Ereignisbehandlung nach dem Schließen des Dialoges beendet. In den meisten Fällen schließen sich daran jedoch Transformationen an, für welche im Dialog Parameter eingegeben und auf Plausibilität überprüft wurden. Diese Parameter werden von der jeweiligen Dialogklasse erfragt und in Objekte der Klasse `ParameterSet` eingetragen. Die `Transformer` sind in der Regel als innere Klassen in den Menüklassen enthalten und werden mit dem Parametersatz zu einer Transformation verknüpft, die dann ausgeführt wird.

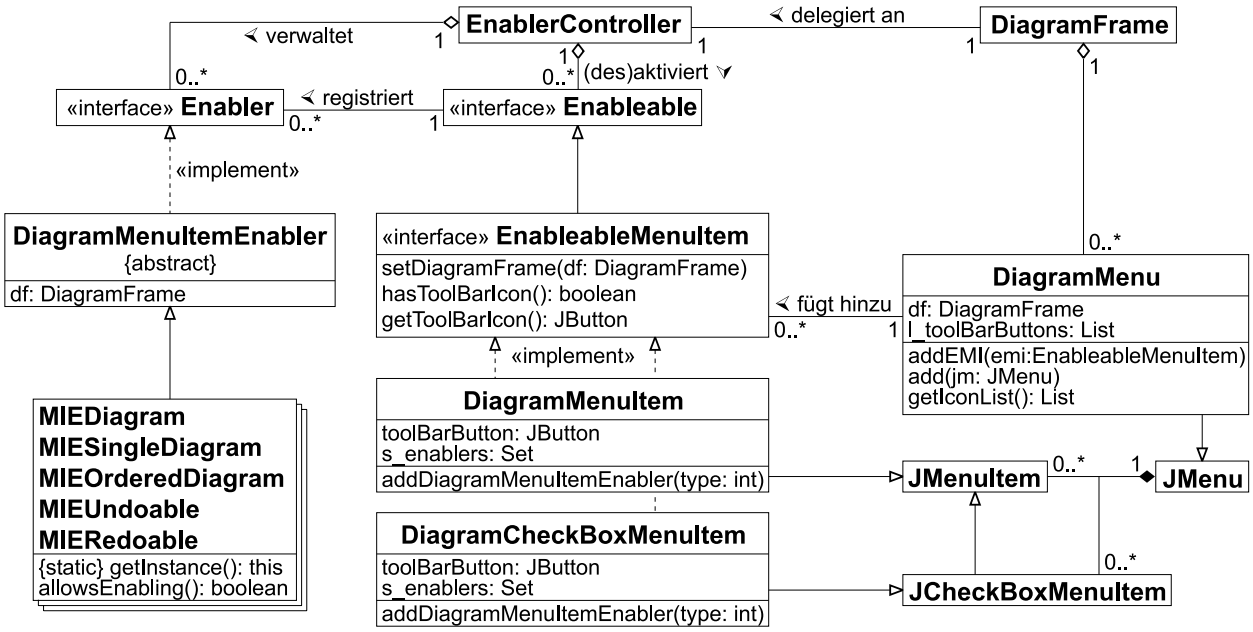


Abbildung 3.18: Klassen zur Arbeit mit Menüeinträgen

Abbildung 3.18 zeigt eine Übersicht der an der Programmierung von Menüs beteiligten Klassen.

Die Klassen `DiagramMenuItem` und `DiagramCheckBoxMenuItem`

Die einzelnen Menüeinträge werden bei Swing durch die Klassen `JMenuItem` und `JCheckBoxMenuItem` repräsentiert, wobei letztere Klasse einen sichtbaren Schalter in einem Menü darstellt. Zur Erweiterung der Funktionalität wurden sie jeweils zu den Klassen `DiagramMenuItem` und `DiagramCheckBoxMenuItem` abgeleitet, deren Aktivierung und Desaktivierung kontrolliert wird und die mit Icons der Toolbar assoziiert sind. Das Icon kann dem Konstruktor durch Angabe eines Namens übergeben werden, welcher mit dem Dateinamen des Icons im Verzeichnis `diagram/icons/` übereinstimmt und von dort mit der Methode `void setToolBarIcon(String iconName)` geladen wird. Die Methode `void createToolBarButton(Icon icon)` erzeugt daraus eine Schaltfläche für die Toolbar.

Zur (Des-)Aktivierung wird das Interface `Enableable` indirekt implementiert. Um die Klassen `DiagramMenuItem` und `DiagramCheckBoxMenuItem` unter einem gemeinsamen Typ ansprechen zu können, wurde als weiteres Interface `EnableableMenuItem` eingeführt, welches genannte Klassen direkt implementieren und das sich seinerseits von `Enableable` ableitet.

`EnableableMenuItem` definiert die Methoden `void setDiagramFrame(DiagramFrame diagramFrame)`, `boolean hasToolBarButton()` und `JButton getToolBarButton()`. Gleichzeitig wird von Klassen, die dieses Interface implementieren, angenommen, dass sie Subklassen von `JMenuItem` seien, die Methoden dieser Klasse werden jedoch nicht mehr explizit definiert. Die Klassen `DiagramMenuItem` und `DiagramCheckBoxMenuItem` erben somit einerseits Methoden der Klassen `JMenuItem` und `JCheckBoxMenuItem` und implementieren andererseits Methoden des Interfaces `EnableableMenuItem`.

Diese Konstruktion bildet die in Java nicht mögliche Mehrfachvererbung nach und bringt auch deren Probleme mit sich. Beispielsweise könnten zwei Methoden `JMenu::add(JMenuItem jmi)` und `DiagramMenu::add(EnableableMenuItem emi)`, wobei `DiagramMenu` eine Subklasse von `JMenu` ist, problemlos koexistieren. Leitet man jedoch die Klasse `DiagramMenu` weiter ab z. B. zu `DiagramWindowsMenu`, so ist dort ein Methodenaufruf `add(DiagramMenuItem)` mehrdeutig und der Compiler kann nicht entscheiden, welche Superklassenmethode aufgerufen werden soll. Daher verwendet die weiter unten beschriebene Klasse `DiagramMenu` die Methode `addEMI(EnableableMenuItem emi)`.

Das Interface `Enabler` definiert die Methode `boolean allowsEnabling()`. Diese Methode legt fest, ob das `Enableable`-Objekt aktiviert werden darf. Die abstrakte Klasse `DiagramMenuItemEnabler` implementiert dieses Interface und definiert die Konstanten `DIAGRAM`, `SINGLE` und `ORDERED` für die wichtigsten `Enabler` in Zusammenhang mit der Verwendung von Diagrammen. Des weiteren hält sie eine Referenz auf `DiagramFrame`, welche mit `setDiagramFrame(DiagramFrame df)` gesetzt wird. Von der Klasse `DiagramMenuItemEnabler` leiten sich schließlich `Enabler` wie `MIEDiagram`, `MIESingleDiagram` und `MIEOrderedDiagram`. Diese lassen eine Aktivierung zu, wenn das jeweils aktuelle Diagramm in der Klasse `DiagramFrame` existiert, den Typ `SingleDiagram` hat bzw. ein `SingleDiagram` mit geordnetem Datensatz ist.

Die Klassen `DiagramMenuItem` und `DiagramCheckBoxMenuItem` verwalten ein `Set` die-

ser `Enabler`. Erfahren sie über `stateChanged()` von einer Zustandsänderung, so überprüfen sie für jeden `Enabler`, ob dieser eine Aktivierung erlaubt. Für eine Aktivierung müssen sämtliche registrierten `Enabler` durch `allowsEnabling() == true` zustimmen.

Die Klasse `DiagramMenu`

Die abstrakte Klasse `DiagramMenu` leitet sich von `JMenu` ab und erweitert diese insbesondere um die Methode `void addEMI(EnableableMenuItem emi)`. Diese Methode registriert im `EnablerController` der Klasse `DiagramFrame` die aktivierbaren Menüeinträge und erzeugt eine Liste der Toolbar-Schaltflächen, die mit `List getIconList()` erhalten werden können. Für alle hinzugefügten Schaltflächen und Menüeinträge werden `ActionListener` registriert. Für eine spezifische Ereignisbehandlung muss zugleich die Methode `actionPerformed(ActionEvent evt)` von den Subklassen implementiert werden. Die vollständigen Menüs können dem `DiagramFrame` mit dessen Methode `void addDiagramMenu(DiagramMenu dm)` bekannt gemacht werden, welche die jeweiligen Elemente der `JMenuBar` bzw. `JToolBar` hinzufügt. Die Methode `DiagramMenu.add(JMenu jm)` erlaubt entsprechend der Methode `JMenu.add(JMenuItem jmi)` die Erzeugung von Untermenüs, wobei Elemente vom Typ `EnableableMenuItem` gesondert behandelt werden.

3.6 Das Hilfesystem

Das JavaHelp-API

Für Java existiert mit JavaHelp ein eigenes API zur Programmierung von Hilfesystemen, das im JavaHelp-Userguide [31] dokumentiert ist. Es handelt sich dabei um das Erweiterungspaket `javax.help`, das nicht zum Standardumfang von Java gehört und zusätzlich installiert werden muss (siehe Java-Tutorial [17], Extension-Mechanism).

Das Hilfesystem besteht zum einen aus der genannten Klassenbibliothek zur Verwaltung und Anzeige der Hilfetexte und zum anderen aus einer Sammlung von Dateien mit den Hilfetexten und Verweisstrukturen. Die Hilfetexte selbst sind in der Hypertext Markup Language HTML (siehe Sprachspezifikation des W3C-Konsortiums [28] und SELFHTML von S. Münz [32]) kodiert. Daneben gibt es ein Verwaltungssystem, das in XML kodiert ist.

Ein Hilfesystem ist in einzelne Hilfesätze mit Informationen zu einem Thema gegliedert, welche mit der Java-Klasse `HelpSet` verwaltet werden. Auf der Dateiebene ist jedem Hilfesatz eine Verwaltungsdatei `hilfesatz.hs` zugeordnet, welche Verweise auf verschiedene Anzeigeelemente wie das Stichwortverzeichnis (Index: `hilfesatzIndex.xml`), das Inhaltsverzeichnis (Table of Contents TOC: `hilfesatzTOC.xml`) und ein Verzeichnis mit Informationen für eine Volltextrecherche enthält. Stichwortverzeichnis und Inhaltsverzeichnis enthalten die Einträge zu verschiedenen Themen und eine ID. Die Zuordnung von IDs zu Hilfetexten befindet sich in einer Zuordnungsdatei (Java help map) `hilfesatz.jhm`, in welcher sich die Hyperlinks zu den Hilfeseiten befinden. Schließlich gibt es eine beliebige Zahl von Hilfeseiten mit den Hilfetexten.

Realisierung im Kontext

Im vorliegenden Programmpaket werden die Hilfesätze den Menüs zugeordnet. In der Klasse `DiagramFrame` wird ein Haupthilfesatz verwaltet, welchem durch Hinzufügen von Menüs, die durch die Klasse `DiagramMenu` repräsentiert werden, gleichzeitig jeweils ein Hilfesatz hinzugefügt wird, der Informationen zur Funktionalität enthält, die mit den Menüeinträgen verknüpft ist. Mit dieser durch JavaHelp unterstützten Technik (dynamic merging) ist es möglich, aus bestehenden Bausteinen Programme zu erzeugen, welche nur spezifische Hilfeinformationen enthalten.

Für das in dieser Arbeit vorgestellte Programm wurden sämtliche Menüeinträge und Dialoge mit den einzugebenden Parametern anhand dieser Online-Hilfe ausführlich dokumentiert. Abbildung 3.17 zeigt einen Dialog zur Darstellung von Hilfetexten.

Die Klasse `Help` und das Interface `Helpful`

Zur Unterstützung der Arbeit mit dem Hilfesystem wurden die Klasse `Help` und das Interface `Helpful` entwickelt. In der Klasse `Help` wird eine Instanz von `HelpSet` und eine ID zur Lokalisierung einer spezifischen Information gekapselt. Die statische Methode `loadHelpSet(String filename)` lädt einen in einer Datei `hilfesatz.hs` enthaltenen Hilfesatz über dessen Dateinamen bezüglich des Klassenpfades und erzeugt aus ihm eine Instanz der Klasse `HelpSet`. Die Methoden `enableHelp(MenuItem mi)`, `enableHelp(Component c)`, `enableHelpOnButton(MenuItem mi)` und `enableHelpOnButton(Component c)` delegieren ihre Aufgaben an die gleichnamigen Methoden des mit dem `HelpSet` erzeugten `HelpBroker`s. Das Interface `javax.help.HelpBroker` deklariert Methoden zur Anzeige und Verwaltung von Hilfesätzen sowie zur Kooperation mit Klassen der Benutzeroberfläche. Die Anwendung der Methoden `enableHelpOn...(...)` auf einen `Button`, einen `JButton` oder ein `MenuItem` ordnet der jeweiligen Komponente einen spezifischen Hilfetext zu. Von dieser Funktionalität wird vielfach Gebrauch gemacht, wenn Angaben zum Hilfesystem Dialogen übergeben werden und dort den Hilfe-Schaltflächen zugeordnet werden.

Das Interface `Helpful` dient der Markierung von Klassen, welche über ein `HelpSet` verfügen. Mit der in diesem Interface definierten Methode `HelpSet getHelpSet()` wird der Zugriff auf ein Objekt vom Typ `HelpSet` deklariert. Die Klasse `DiagramFrame` überprüft für hinzugefügte `DiagramMenu`s, ob sie dieses Interface implementieren und fügt in diesem Fall das jeweilige `HelpSet` dem Haupthilfesatz hinzu.

3.7 Das Hauptprogramm

Wie bereits ausgeführt wurde, enthalten die Menüs gleichzeitig die von ihnen ausgehende Funktionalität oder verwenden Klassen, in welche umfangreichere Berechnungen ausgelagert werden. Die Menüs werden von der Klasse `DiagramFrame` verwaltet, die auch die Aktivitäten der übrigen Klassen koordiniert, soweit diese nicht ohnehin schon zusammenarbeiten. Die Hauptklasse des Programms *X-mas* heißt `Xmas` und leitet sich von der Klasse `DiagramFrame` ab. Ihre einzige Aufgabe besteht darin, die für dieses Programm benötigten Menüs zu erzeugen und mit `DiagramFrame::addDiagramMenu(DiagramMenu dm)` der Klasse `DiagramFrame`

zu übergeben.

Im einzelnen handelt es sich um folgende Menüs: Die Menüs *File* und *Windows* der Klassen `DiagramFileMenu` und `DiagramWindowsMenu` sind Teil jedes auf der Klasse `DiagramFrame` basierenden Programms und werden automatisch hinzugefügt. Sie übernehmen die Grundfunktionen wie das Laden, Speichern, Importieren, Exportieren, Drucken und Schließen von Diagrammen und Datensätzen sowie die Anzeige von Koordinaten, Konfiguration der Diagrammdarstellung, Wahl des Darstellungsbereiches und Verknüpfung von Single-Diagrammen zu Multi-Diagrammen. Das Menü *Edit* der Klasse `DiagramEditMenu` enthält Methoden zum Editieren und Duplizieren von Datensätzen sowie die Undo- und Redo-Operationen. Das Menü *Calculate* der Klasse `DiagramCalculationMenu` enthält eine Reihe mathematischer Operationen wie allgemeine Transformationen der Abszissen- und Ordinatenwerte, Berechnung der Tangentensteigungsfunktion und Flächeninhaltsfunktion, symbolische Ableitungen, Glättung, Verknüpfungen mehrerer Datensätze, Intrapolation, Fouriertransformationen, Optimierungsroutinen und statistische Berechnungen. Das Menü *Tools* der Klasse `XRDToolsmenu` enthält Hilfsfunktionen zur Unterstützung der Auswertung von Röntgendiffraktogrammen und das Menü *Evaluate* der Klasse `XRDEvaluationMenu` die Operationen zur Auswertung der Diffraktogramme. Das Menü *Properties* der Klasse `DiagramPropertiesMenu` dient der Anzeige, Speicherung, Übernahme, dem Laden und Ausdruck der einem Datensatz zugeordneten Parameter wie z. B. die Probenzusammensetzung. Über das Menü *Help* der Klasse `DiagramHelpMenu` können die Online-Hilfe und die Programminformationen angezeigt werden. Die für diese Operationen benötigten Klassen und ihre Anwendung werden in den folgenden Kapiteln dokumentiert.

Kapitel 4

Auswertung von Röntgendiffraktogrammen

Dieses Kapitel stellt Menüs, Dialoge, Klassen und Methoden vor, welche für die Auswertung von Röntgendiffraktogrammen von Bedeutung sind. Das in diesem Zusammenhang beschriebene System ist eine Schmelze von Bismut(III)chlorid $BiCl_3$. Diese Substanz wurde von Heusel [33] am ESRF in Grenoble untersucht. Es sei darauf hingewiesen, dass die folgenden Ausführungen allein dem Zwecke dienen, die Möglichkeiten zur Auswertung durch das Programm darzustellen. Bezüglich einer Diskussion des Systems sei auf die genannte Literatur verwiesen. Die folgende Tabelle enthält die Messbedingungen:

Messbedingungen	
Substanz S	$BiCl_3$ 99,999% (Fa. Aldrich)
Dichte (S)	3,8 g/cm^3
Winkelbereich	0,7° bis 30°
Winkelauflösung	0,05°
Strahlungsquelle	Synchrotron ESRF, Messstand ID15A
Strahlungsenergie	80 keV
Strahlbreite	0,3 mm
Halbwertsbreite Monochromator	50 eV
Kapillare K	NMR-Röhrchen 322-pp-8" Fa. Wilmad
Material (K)	Borosilikatglas Pyrex 7740 Fa. Spintec
Zusammensetzung (K)	80,5% SiO_2, 12,9% B_2O_3, 3,8% Na_2O 2,2% Al_2O_3, 1,2% Li_2O, 0,4% K2O
Dichte (K)	2,23 g/cm^3
Innendurchmesser (K)	2,413 mm +0,013 mm/-0 mm
Außendurchmesser (K)	3,000 mm +0 mm/-0,013 mm
Temperatur	270 °C

4.1 Die Berechnung von Hilfsgrößen zur Auswertung von Röntgendiffraktogrammen

Neben der eigentlichen Auswertung der Diffraktogramme stehen etliche Methoden zur Unterstützung der Arbeit mit Röntgendiffraktogrammen zur Verfügung, welche eine Optimierung der Versuchsanordnung erlauben oder für weitere Berechnungen benötigt werden. Diese Funktionen sind über das Menü *Tools* zugänglich.

4.1.1 Eingabe der Messparameter

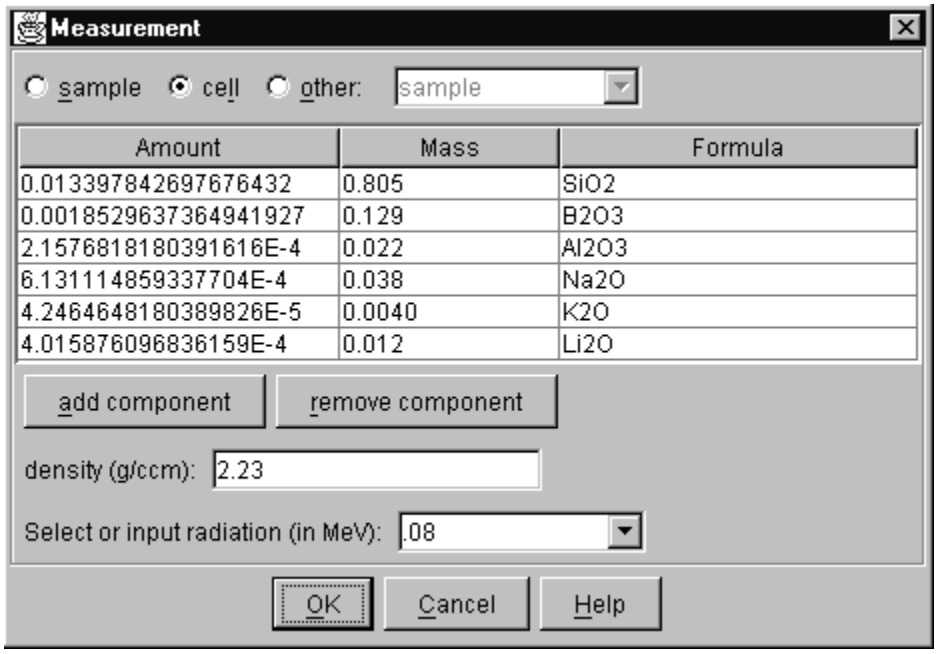


Abbildung 4.1: Eingabe der Messparameter für die Messzelle aus Glas

Viele Programmteile zur Auswertung oder Berechnung von Hilfsgrößen erlauben die Verwendung von Parametern, die zentral eingegeben wurden. Diese Eingabe erfolgt über einen in Abbildung 4.1 gezeigten Dialog der Klasse `MeasurementDialog`. Zunächst muss angegeben werden, auf welche Substanz sich die weiteren Eingaben beziehen. Das Programm verwendet zwei Substanzen mit festen Bezeichnungen: *sample* für die untersuchte Probe und *cell* für das Material der Messzelle. Darüber hinaus können im Eingabefeld hinter *other* variable Bezeichnungen für weitere Substanzen definiert werden. In der Tabelle können Komponenten der Mischungen durch Angabe ihrer chemischen Summenformel und ihres absoluten Anteils in Gramm oder Mol eingegeben werden. Abbildung 4.1 zeigt die Eingabe für eine Glaskapillare.

Für die vielfach benötigte Eingabe der Strahlungsenergie wurde die Klasse `RadiationComboBox` entwickelt. Sie bietet als Auswahl die Liste der in der Klasse `PSE` aufgeführten Strahlungstypen (z. B. Mo Ka für die K_α-Strahlung einer Molybdän-Anode). Entsprechend der verfügbaren Daten bei der Berechnung der Absorption ist auch die numerische Angabe

einer Strahlungsenergie im Bereich von 1 keV bis 100 GeV zulässig. Alle eingegebenen Größen werden nach bestätigendem Abschluss des Dialogs überprüft und in die Registratur eingetragen.

4.1.2 Die Berechnung stöchiometrischer Größen

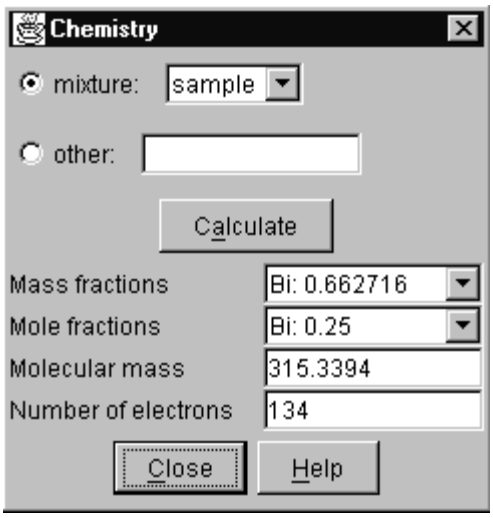


Abbildung 4.2: Berechnung stöchiometrischer Größen

Die Berechnung von Molmasse, Elektronenzahl und Zusammensetzung in Molen- und Massenbrüchen erfolgt in der in Abschnitt 3.3 beschriebenen Klasse `Chemistry`. Abbildung 4.2 zeigt einen Dialog der Klasse `ChemistryDialog`. Der Benutzer kann die Berechnungen für eine mit dem Dialog *Measurement* definierte Mischung durch Auswahl deren Bezeichnung oder für eine unter *other* einzugebende Substanz durch Drücken der Schaltfläche *Calculate* durchführen.

4.1.3 Berechnung von Absorptionskoeffizienten

Die Absorption von Röntgenstrahlung erfolgt analog zum Lambert-Beer-Gesetz nach

$$I = I_0 \cdot e^{-\mu x} \tag{4.1}$$

Darin ist μ der lineare Absorptionskoeffizient und x die durchstrahlte Strecke. Der Absorptionskoeffizient ergibt sich aus

$$\mu = \rho \cdot \sum_i \frac{m_i}{m} \cdot \left(\frac{\mu}{\rho}\right)_i \text{ mit } m = \sum_i m_i \tag{4.2}$$

und der Massendichte ρ als Summe aus den Massenabsorptionskoeffizienten $(\mu/\rho)_i$ der einzelnen Atomsorten entsprechend deren Massenanteil. In der Klasse `MassAttenuation` werden die Massenabsorptionskoeffizienten für Atomsorten, Verbindungen oder für die in den `properties` definierten Gemischen berechnet.

Die zugrundeliegenden Daten befinden sich in der Datei `xcom.dat`, welche der Datenbank DABAX [34] des ESRF entnommen wurde. Die enthaltenen Werte beruhen auf Berechnungen von M. J. Berger und J. H. Hubbell [35]-[43] mit dem Programm XCOM, welches die Streuquerschnitte für die Elemente mit den Ordnungszahlen 1-100 als Summe der verschiedenen Beiträge (inkohärente Streuung, kohärente Streuung, photoelektrische Absorption, Paarbildung in den Feldern des Atomkerns und der Elektronen) für Energien von 1 keV bis 100 GeV berechnet. Die Datei enthält zugleich die Einzelbeiträge für die genannten Prozesse. Die Werte beziehen sich auf ungeladene Atome und sind in logarithmischer Schrittweite tabelliert. Zusätzlich sind die Werte an den Absorptionskanten gegeben. Einheit für die tabellierten Streuquerschnitte σ_t ist barn/Atom mit 1 barn = 100 fm^2.

Die Methode `MassAttenuation.loadXCOM(int number, List l_x, List l_y)` lädt die Datei, durchsucht sie auf das Element mit Kernladungszahl `number` und trägt die Strahlungsenergie in MeV in die Liste `l_x` und den Streuquerschnitt in barn/Atom in die Liste `l_y` ein. Mit `double getMassAttenuation(String symbol, double d_radiation)` wird der Massenabsorptionskoeffizient in cm^2/g für ein bestimmtes Element mit Symbol `symbol` bei einer bestimmten Strahlungsenergie in MeV berechnet. Ist für die Strahlung ein Typ wie z. B. *„Mo Ka“* für die K_α-Linie von Molybdän angegeben, so bestimmt die Methode `double getSingleAttenuation(String symbol, String s_radiation)` zunächst die Strahlungsenergie für den Typ `s_radiation` und ruft weiter die vorige Methode auf. Die Umrechnung der tabellierten totalen Streuquerschnitte σ_t in die Massenabsorptionskoeffizienten μ/ρ erfolgt mit

$$\mu/\rho = \sigma_t \cdot \frac{N_A}{M} \tag{4.3}$$

über die Avogadro-Konstante N_A und die Atommasse M.

Für Energien x zwischen zwei tabellierten Werten x_1 und x_2 mit Streuquerschnitten y_1 bzw. y_2 wird über die Methode `double interpolateExponential(double x, double x1, double y1, double x2, double y2)` exponentiell intrapoliert nach

$$y = \exp\left(\frac{\ln y_2 - \ln y_1}{\ln x_2 - \ln x_1} \cdot (\ln x - \ln x_1) + \ln y_1\right) \tag{4.4}$$

Für die Energien an den Absorptionskanten wird der höhere Massenabsorptionskoeffizient oberhalb der Kante zurückgegeben. Mit `double getMassAttenuation(Composition comp, double density, double radiation)` kann der Massenabsorptionskoeffizient für Mischungen mit einer Zusammensetzung `comp` berechnet werden. Dazu werden die Massenanteile der Atomsorten mit `Chemistry.getMassFractions(Composition comp)` bestimmt und die Massenabsorptionskoeffizienten nach 4.2 aufsummiert.

Die Berechnung der Massenabsorption ist für den Benutzer über einen Dialog der Klasse `MassAttenuationDialog` möglich, wie er in Abbildung 4.3 dargestellt ist. Der Dialog verwendet eine zuvor definierte Substanz *sample* — in diesem Fall handelt es sich um $BiCl_3$ — und deren Dichte und berechnet für eine eingegebene durchstrahlte Weglänge die Massenabsorption und das Intensitätsverhältnis zwischen ein- und ausfallender Strahlung.

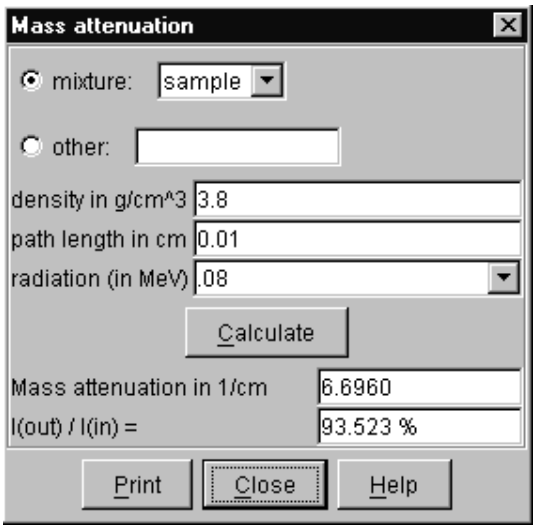


Abbildung 4.3: Berechnung der Massenabsorption

4.1.4 Die Berechnung von Röntgenspektren

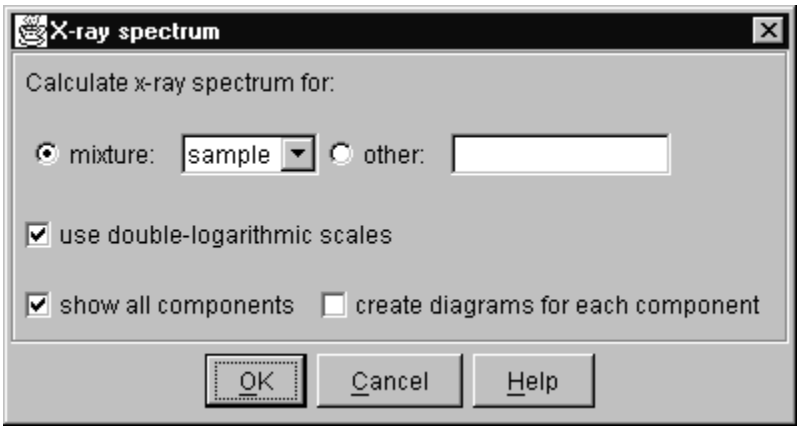


Abbildung 4.4: Berechnung von Röntgenspektren

Röntgenspektren stellen die Abhängigkeit des Absorptionskoeffizienten bzw. der Extinktion von der Wellenlänge dar. Durch Auswertung der Feinstruktur des Nahkantenbereiches in XAFS-Spektren lassen sich Strukturinformationen über die absorbierende Probe erhalten. Die hier berechneten Spektren enthalten hingegen ein Spektrum über einen sehr großen Energiebereich mit niedriger Energieauflösung. Falls für die Aufnahme von Röntgendiffraktogrammen eine Synchrotronstrahlungsquelle zur Verfügung steht, so lässt sich mit ihrer Hilfe abschätzen, in welchem Energiebereich geeignete Absorptionskoeffizienten vorliegen. Weiterhin sind die Kanten der enthaltenen Elemente sichtbar, wodurch die Auswahl einer geeigneten Strahlung möglich ist, die keine Fluoreszenz anregt.

Abbildung 4.4 zeigt einen Dialog der Klasse `AbsorptionDialog` mit den einzugeben-

den Parametern. Für die ausgewählte Substanz kann optional die Verwendung doppelt-logarithmischer Skalen angegeben werden. Das zu erzeugende Diagramm enthält die massengewichtete Summe der Massenabsorptionskoeffizienten aller enthaltenen Atomsorten. Optional werden die Beiträge der enthaltenen Elemente dargestellt und getrennte Diagramme für jedes Element erstellt.

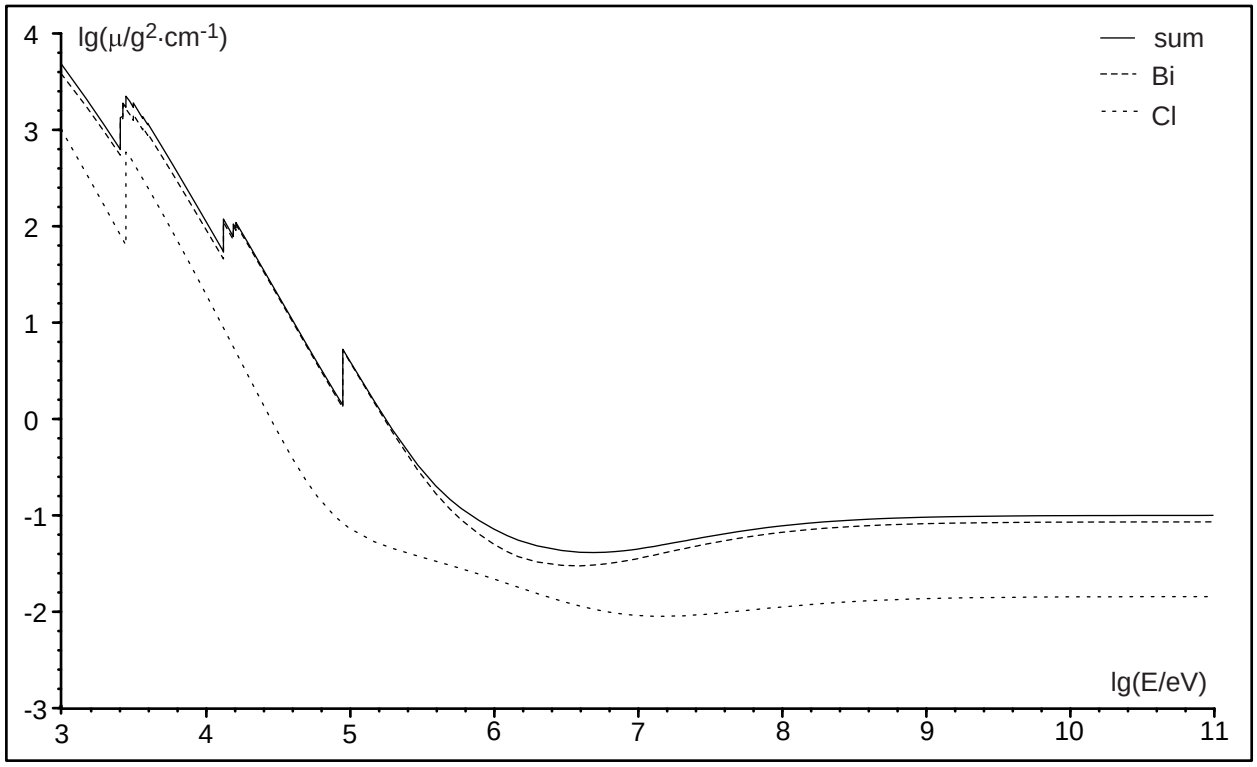


Abbildung 4.5: Röntgenspektrum von $BiCl_3$

Abbildung 4.5 zeigt das erzeugte Diagramm mit dem Röntgenspektrum von $BiCl_3$ (*sum*) sowie seiner Komponenten Bi und Cl im gesamten zur Verfügung stehenden Datenbereich in doppelt-logarithmischer Auftragung. Die Spektren werden mit der Methode `getAbsorptionDiagram(Map m_massFractions, DoubleArray dax, DoubleArray day, List componentList)` der Klasse `MassAttenuation` unter Verwendung der in der Klasse `Chemistry` bestimmten Massenbrüche berechnet. In das `DoubleArray dax` werden die Energien in eV eingetragen, ins `DoubleArray day` die Massenabsorptionskoeffizienten in cm^2/g der Substanz oder Mischung und in die `componentList` die Massenabsorptionskoeffizienten der einzelnen Elemente entsprechend ihrer Massenanteile. Die Daten für die einzelnen Elemente lassen sich nicht unmittelbar aufsummieren, da deren Abszissenwerte wegen der Angabe von Absorptionskanten nicht übereinstimmen. Mit `List combineLists(List l)` und `void combineDoubleLists(List baseList, List insertList)` wird zunächst ein aufsteigend geordneter Satz von Abszissenwerten als Vereinigungsmenge der Abszissenwerte der in den Listen enthaltenen Elemente erzeugt. Im nächsten Schritt erzeugt `getAbsorptionDiagram(...)` für sämtliche Elemente die zugehörigen Ordinatenwerte durch exponentielle Intrapolation. Schließlich werden die Einzelbeiträge entsprechend ihrer Massenbrüche aufsummiert. In `absorptionDiagram()` werden dann entsprechend der im `AbsorptionDialog`

gewählten Optionen die Daten ggf. logarithmiert, geeignete Diagramme erzeugt und dem `DiagramFrame` hinzugefügt.

4.1.5 Die Berechnung von Absorptionskanten

Aus EXAFS[1]-Messungen lässt sich die Nahordung von Atomen um ein bestimmtes Atom bestimmen, welches oberhalb einer bestimmten Energie, der Absorptionskante, Röntgenstrahlung absorbiert [44]. Um die Qualität einer EXAFS-Messung im Vorfeld abzuschätzen, kann man den Kantenhub an der zu untersuchenden Kante berechnen. Als Signal-Rausch-Verhältnis r_{sn} sei definiert

$$r_{sn} = \frac{\mu_{up} - \mu_{low}}{\mu_{low}} \tag{4.5}$$

Darin sind μ_{up} die Massenabsorption der Substanz oberhalb der Kante und μ_{low} die Massenabsorption der Substanz unterhalb der Kante. In der Praxis wird häufig eine als „Jump“

$$j = E_{up} - E_{low} \tag{4.6}$$

bezeichnete Größe verwendet, welche die Differenz der Extinktionswerte $E = \mu \cdot x$ oberhalb und unterhalb der Kante darstellt. Das Signal-Rausch-Verhältnis ist jedoch aussagekräftiger, da es nicht von der Probendicke abhängt. Davon getrennt zu betrachten ist die Extinktion, welche eine Aussage über die maximal sinnvolle Probendicke liefert.

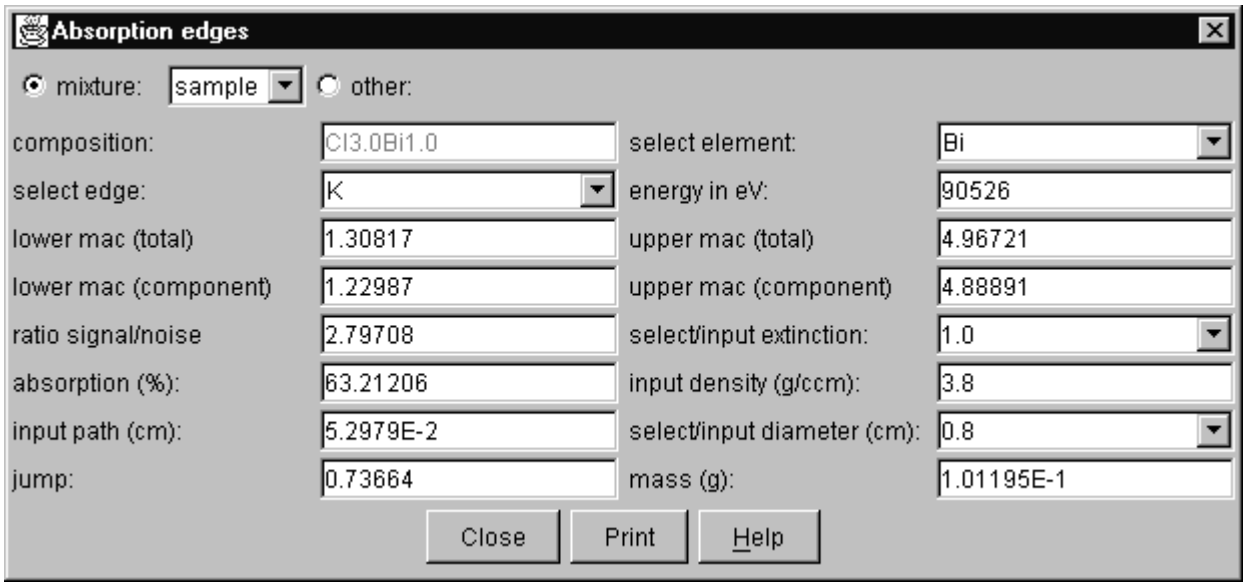


Abbildung 4.6: Berechnung von Absorptionskanten

Die Berechnung kann für eine zuvor definierte Mischung oder eine einzugebende Substanz durchgeführt werden. Aus den enthaltenen Atomsorten kann das Element für die zu betrachtende Kante sowie der Typ (K, LI, LII, ...) der Kante ausgewählt werden. Die Energie der

[1]Extended x-ray absorption fine structure

Kante in eV wird dann angezeigt. Für die weiteren Berechnungen muss die Energie entsprechend der verwendeten Daten für die Massenabsorptionskoeffizienten im Bereich zwischen 1 keV und 100 GeV liegen. Wenn dies der Fall ist[2], werden die Massenabsorptionskoeffizienten in cm^2/g unter- und oberhalb der Kante für das Kantenelement und die Substanz sowie das Signal-Rausch-Verhältnis berechnet. Für die Berechnung von j können die gewünschte Absorption, die Extinktion oder die Schichtdicke für das Experiment vorgegeben werden, aus denen die jeweils anderen Größen automatisch berechnet werden. Nach Auswahl oder Eingabe des Durchmessers eines zylindrischen Probenpresslings wird die einzuwiegende Masse der Substanz berechnet.

Abbildung 4.6 zeigt einen Dialog der Klasse `AbsorptionEdgeDialog`, mit dessen Hilfe die Berechnungen nach obigen Formeln durchgeführt werden. Der Massenabsorptionskoeffizient wird aus Platzgründen mit *mac* abgekürzt und für das Kantenelement *component* und die gesamte Probe *total* oberhalb *upper* und unterhalb *lower* der Kante berechnet.

Die Kantenenergien sind über die Klasse `AbsorptionEdges` zugänglich. Diese Klasse lädt die Daten aus der Datei `xrd/absedgeenergy.dat` und stellt Methoden zur Verfügung, welche die für ein Element verfügbaren Kanten und die Energien für ein bestimmtes Element an einer bestimmten Kante zurückgeben. Die verwendeten Daten sind einer Zusammenfassung von Williams [45] entnommen, der Werte von Bearden und Burr [46], korrigierte Werte von Cardona und Ley [47] und Fuggle und Mårtensson [48] verwendet.

Mit Hilfe der Schaltfläche *Print* werden die eingegebenen und berechneten Daten in einem eigenen Dialog angezeigt und können dort editiert und ausgedruckt werden. Abbildung 4.7 zeigt diesen Dialog.
Zahlreiche in der Arbeitsgruppe durchgeführte Messungen bestätigten eine gute Übereinstimmung der mit diesem Programm berechneten Abschätzungen mit den erhaltenen Messwerten.

4.1.6 Die Berechnung von Wichtungsfaktoren

Die Gesamtkorrelationsfunktion setzt sich aus der gewichteten Summe der Atompaarkorrelationsfunktionen zusammen. Die Wichtungsfaktoren $w_{i,j}$ für eine Atompaarkorrelationsfunktion der Elemente i und j sind

$$w_{i,j} = \frac{n_i n_j Z_i Z_j}{\left(\sum_k n_k Z_k\right)^2} \tag{4.7}$$

wobei die Elektronenzahlen Z als Näherung für die mittlere effektive Elektronenzahl verwendet werden. Mit Hilfe des in Abbildung 4.8 dargestellten Dialogs der Klasse `WeightDialog` lassen sich die Faktoren für eine definierte Zusammensetzung berechnen. Der Dialog enthält eine Tabelle mit den Wichtungsfaktoren der Atompaarkorrelationsfunktionen, wobei für $i \neq j$ die identischen Werte $w_{i,j}$ und $w_{j,i}$ zu einem Wert $2w_{i,j}$ zusammengefasst sind. In der Liste unter der Tabelle sind die Wichtungsfaktoren nach absteigender Größe sortiert aufgeführt.

[2] die K-Kanten der Elemente Wasserstoff bis Neon liegen nicht in diesem Bereich

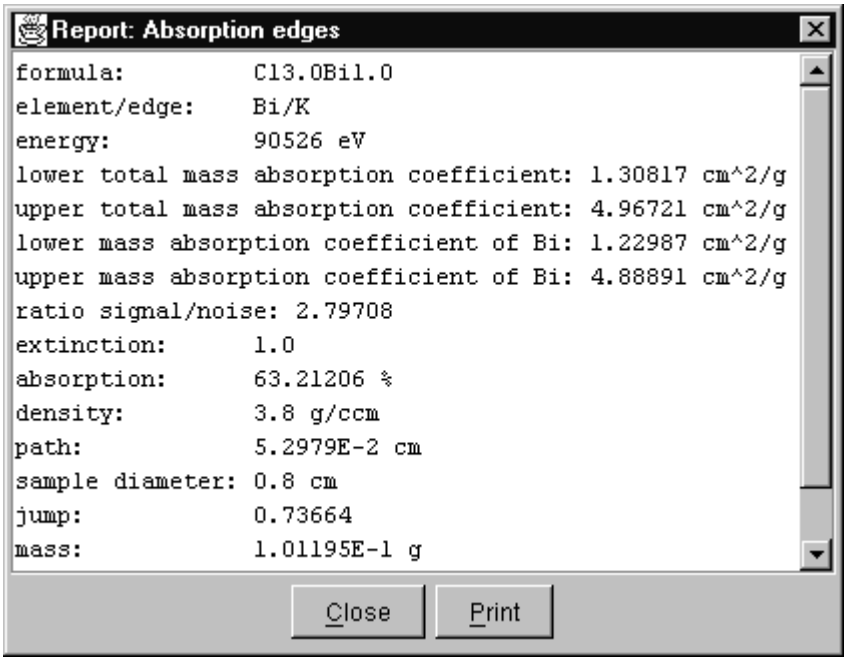


Abbildung 4.7: Ausdruck der berechneten Größen einer Absorptionskante

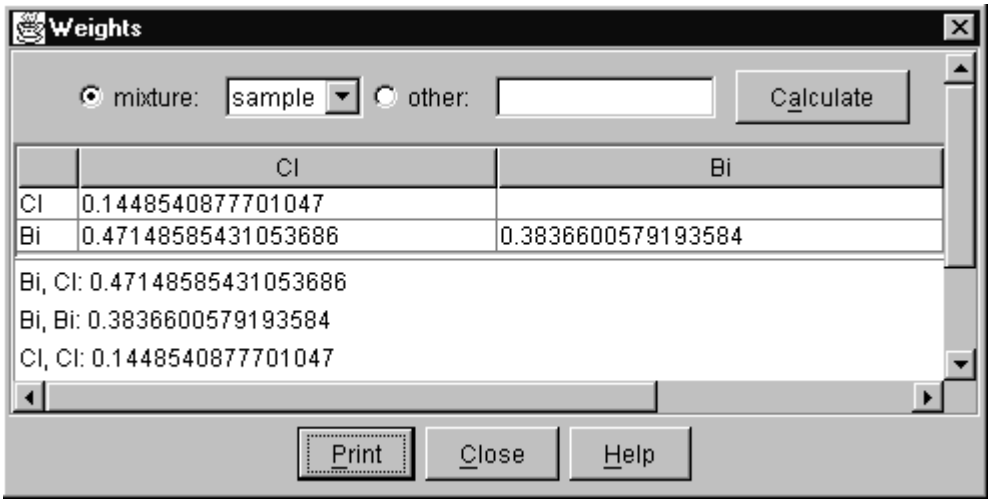


Abbildung 4.8: Berechnung von Wichtungsfaktoren

4.1.7 Das Laden von Messdateien des Seifert-Diffraktometers

Wie bereits ausgeführt wurde, erlaubt das allgemeine Import-Filter das Laden gängiger ASCII-Formate. Das im Arbeitskreis verwendete Messprogramm `XDAL 3000` zum Zweikreisdiffraktometer Isodebyeflex 2002 der Firma R. Seifert verwendet jedoch ein inkompatibles Format, welches über Kopfzeilen und unter Umständen mehrere aneinandergehängte Messintervalle verfügt. Die einzelnen Messbereiche werden über die Methode `importSeifert()` nacheinander eingeladen. Der Benutzer bekommt Informationen zu den Bereichen angezeigt und muss die jeweilige Messzeit pro Messwinkel eingeben, damit absolute Zählraten berech-

net werden können, die gegebenenfalls zu anderen Messungen addiert werden können. Für jeden Messbereich wird ein eigenes `SingleDiagram` erzeugt. Wurden mehrere Messungen im gleichen Winkelbereich durchgeführt, so können diese mit dem Menüeintrag *Calculate/Column operations* (siehe Abschnitt 6.4) aufsummiert werden.

4.2 Gang der Auswertung

In diesem Abschnitt wird eine komplette Auswertung des Röntgendiffraktogramms einer Schmelze von Bismut(III)chlorid $BiCl_3$ beschrieben.

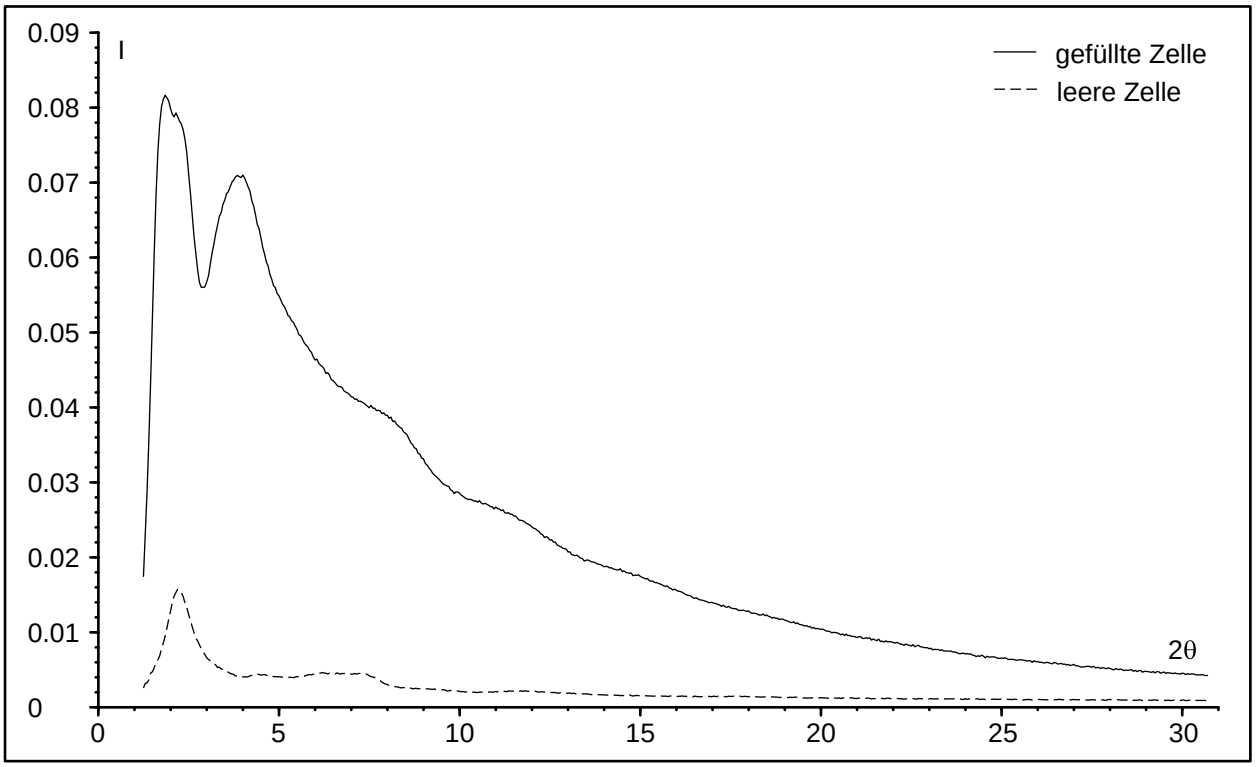


Abbildung 4.9: Diffraktogramme der leeren und mit $BiCl_3$ gefüllten Messzelle nach Abzug der Luftstreuung

Abbildung 4.9 zeigt die Diffraktogramme der leeren und mit $BiCl_3$ gefüllten Messzelle jeweils nach Abzug der Luftstreuung.

4.2.1 Die Absorptionskorrektur

Auf ihrem Weg von der Strahlungsquelle zum Detektor wird die Röntgenstrahlung von der Probe (s), der Messzelle (c) und der sie umgebenden Luft absorbiert. Abbildung 4.10 zeigt den Strahlverlauf durch eine zylindrische Kapillare in der Aufsicht. Die Absorption durch Luft mindert die Strahlintensität und verschlechtert damit das Signal-Rausch-Verhältnis. Eine Verbesserung kann durch Verwendung von Strahlrohren erzielt werden, die mit schwächer absorbierendem Heliums gefüllt sind. Entscheidend ist jedoch, dass die Strahlintensität beim

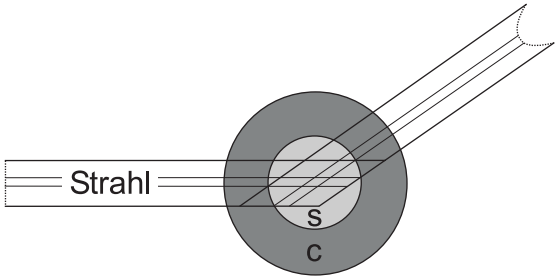


Abbildung 4.10: Absorption des Röntgenstrahls

Verlauf durch die Probe und Messzelle an verschiedenen Positionen bezüglich des Strahlquerschnitts unterschiedlich geschwächt wird und dass diese Schwächung weiterhin eine Funktion der Strahllage und des Beugungswinkels ist. Für die Auswertung muss dies durch eine Absorptionskorrektur berücksichtigt werden. In der Literatur wurden Absorptionskorrekturen für zahlreiche Strahl- und Probengeometrien beschrieben. Für diese Arbeit wurden zwei Strahlgeometrien ausgewählt, die in der Praxis besonders häufig auftreten: Die Absorptionskorrektur nach Kendig und Pings für gefüllte zylindrische Kapillaren und die Absorptionskorrektur nach Debye und Menke für die Reflexion an planaren Proben. Die Verwendung einer exakten Absorptionskorrektur ist entscheidend für den Erfolg der sich anschließenden Normierung.

Die Absorptionskorrektur nach Kendig und Pings

Diese Absorptionskorrektur dient der Korrektur einer zylindrischen Proben- und Kapillarsymmetrie bei unterschiedlichen Strahllagen und Strahlbreiten und wird detailliert im Abschnitt 5.1 beschrieben.
Abbildung 4.11 zeigt die Eingaben für die im Experiment verwendete Strahlgeometrie: Die Mitte des 0,3 mm breiten Strahls verläuft zentral durch die Kapillare mit einem Innendurchmesser von 2,413 mm und einem Außendurchmesser von 3,0 mm. Für die Werte der leeren Messzelle kann ein Diagramm ausgewählt werden. Falls für die entsprechende Geometrie und Absorptionskoeffizienten bereits Werte für die Absorptionsfaktoren vorliegen, können diese aus einer Datei geladen werden. Andernfalls werden diese Faktoren berechnet und können in eine Datei gespeichert werden. Abbildung 4.15 zeigt das Diffraktogramm von $BiCl_3$ nach der Absorptionskorrektur.

Die Absorptionskorrektur für die planare Reflexionsgeometrie

Eine weitere häufig auftretende Anordnung von Beugungsexperimenten ist die in Abbildung 4.12 dargestellte Reflexion an planaren Proben. Für Proben, welche den einfallenden Strahl vollständig absorbieren, gilt die Formel von Debye und Menke [4], [5]:

$$I_{korr} = \frac{I}{A\,(2\theta)} \text{ mit } A\,(2\theta) = \frac{1}{\mu}\frac{sin\,(2\theta - \alpha)}{\sin\alpha + \sin\,(2\theta - \alpha)} \tag{4.8}$$

Darin ist α der Winkel des einfallenden Strahls gegen die Probenebene. Abbildung 4.13 zeigt einen Dialog zur Eingabe der Parameter für diese Korrektur. Zusätzlich können

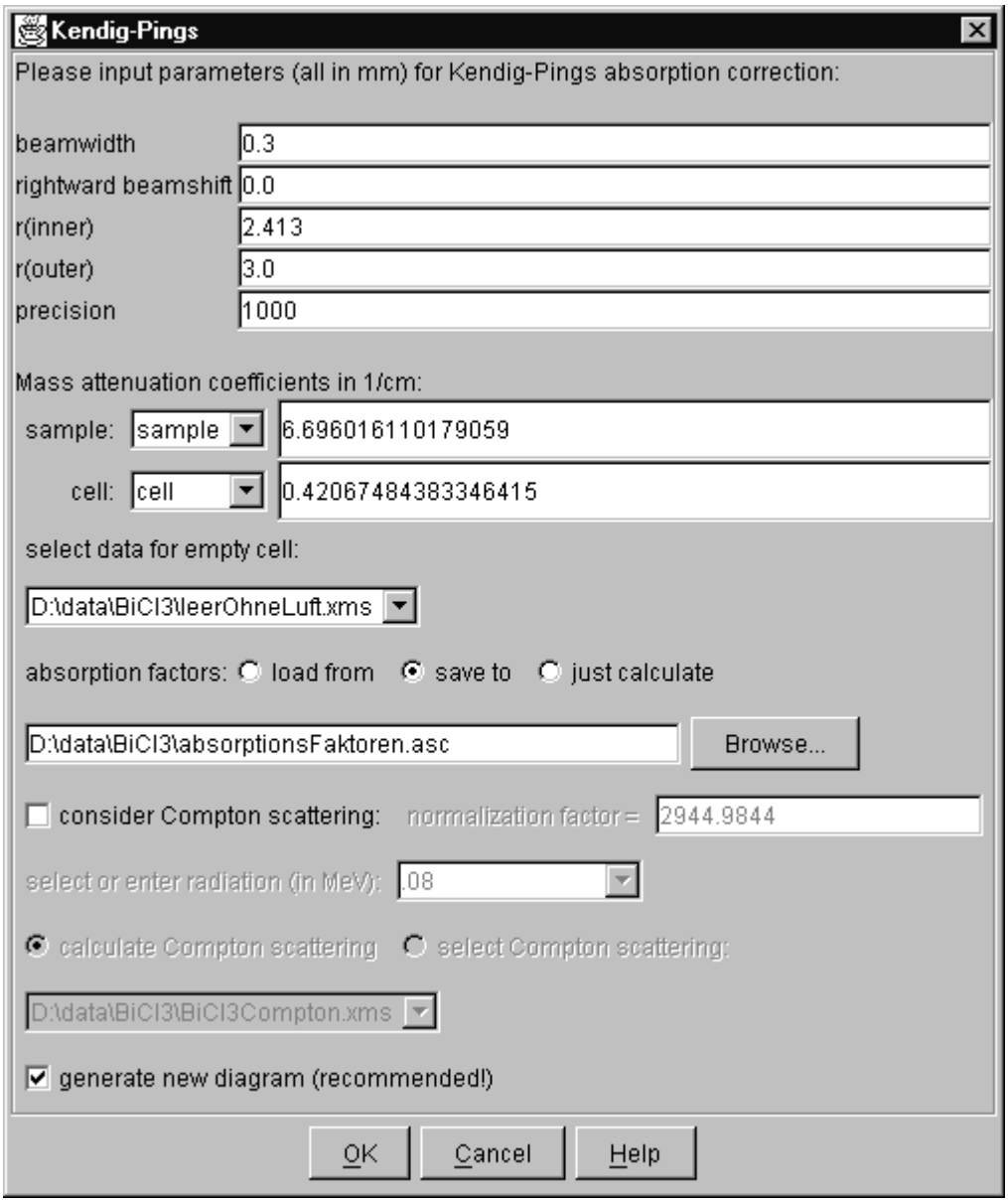


Abbildung 4.11: Absorptionskorrektur nach Kendig und Pings

durch Drehen der Probe symmetrische Reflexionsbedingungen nach Bragg-Brentano gewählt werden. Dann ist der Winkel des einfallenden Strahls gleich dem Winkel des gestreuten Strahls zur Probenebene: $\alpha = \theta$. Die innere Klasse `XRDEvaluationMenu$PlanarReflectionAbsorptionCorrector` berechnet diese Transformation.

4.2.2 Die Polarisationskorrektur

Die Polarisation der Röntgenstrahlung ändert sich bei der Streuung an Probe und Monochromator und damit auch die gemessene Intensität. Dieser Effekt wird durch eine Polarisationskorrektur berücksichtigt. Azaroff, Kerr et al. [49], [50] beschreiben eine Korrektur für unpolarisierte Primärstrahlung einer Röntgenröhre, die durch einen Monochromatorkristall

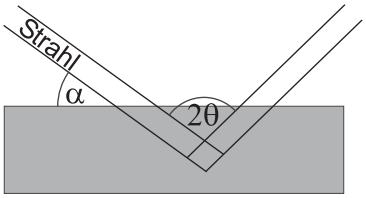


Abbildung 4.12: Die planare Reflexionsgeometrie

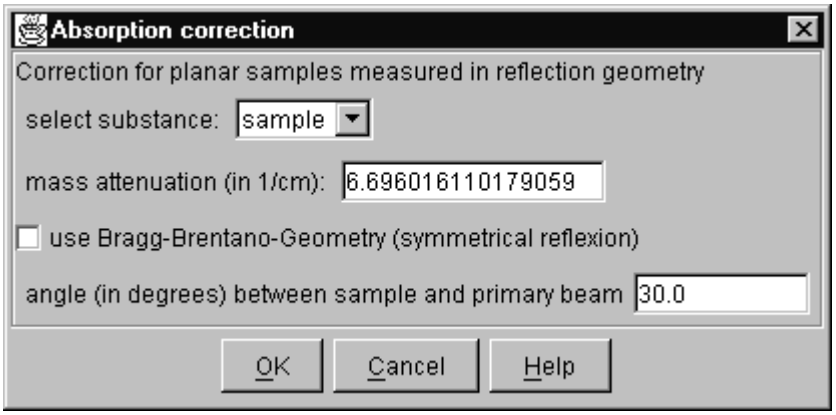


Abbildung 4.13: Dialog zur Absorptionskorrektur bei planarer Reflexionsgeometrie

teilpolarisiert wird:

$$I_{korr} = \frac{I}{P(2\theta)} \text{ mit } P(2\theta) = \frac{1 + \cos^2 2\alpha \cos^2 2\theta}{1 + cos^2 2\alpha} \tag{4.9}$$

Hierin ist α der Bragg-Winkel des Monochromatorkristalls. Falls kein Monochromator verwendet wird, vereinfacht sich diese Formel mit $\alpha = 0$ zu

$$P(2\theta) = \tfrac{1}{2}\left(1 + \cos^2 2\theta\right). \tag{4.10}$$

Wird mit Synchrotronstrahlung gearbeitet, so ist zu beachten, dass diese bereits linear in der Ringebene des Synchrotrons polarisiert ist. In diesem Falle kommt eine von Klein und Nishina eingeführte Formel [51] zur Anwendung:

$$P(2\theta) = \tfrac{1}{2}\left(2 - \sin^2 2\theta + \sin^2 2\theta \cdot \pi(E)\right). \tag{4.11}$$

Hierin ist $\pi(E)$ der Polarisationsgrad, welcher für die horizontal linear polarisierte Synchrotronstrahlung den Wert -1 hat. Damit vereinfacht sich diese Formel zu

$$P(2\theta) = \cos^2(2\theta). \tag{4.12}$$

Abbildung 4.14 zeigt einen Dialog der Klasse `PolarizationDialog` zur Eingabe der Parameter für die Polarisationskorrektur. Die Transformation wird durch die innere Klasse

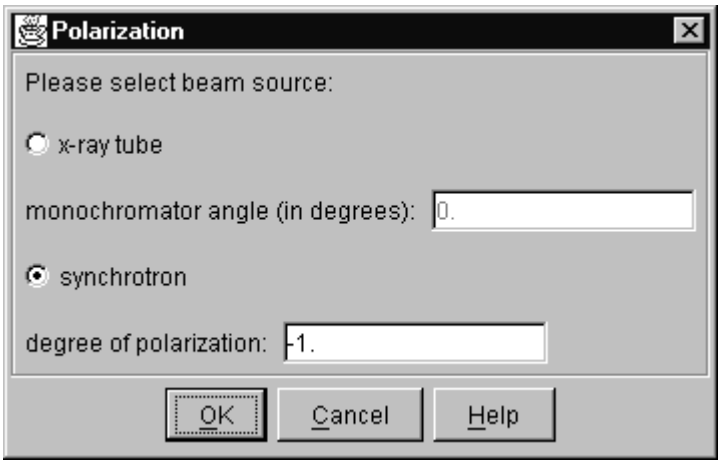


Abbildung 4.14: Dialog zur Polarisationskorrektur

`XRDEvaluationMenu$Depolarizator` berechnet. Abbildung 4.15 zeigt das auf Polarisation korrigierte Diffraktogramm von $BiCl_3$. Die Polarisationskorrektur wurde nach Klein-Nishina berechnet mit $\pi(E) = -1$.

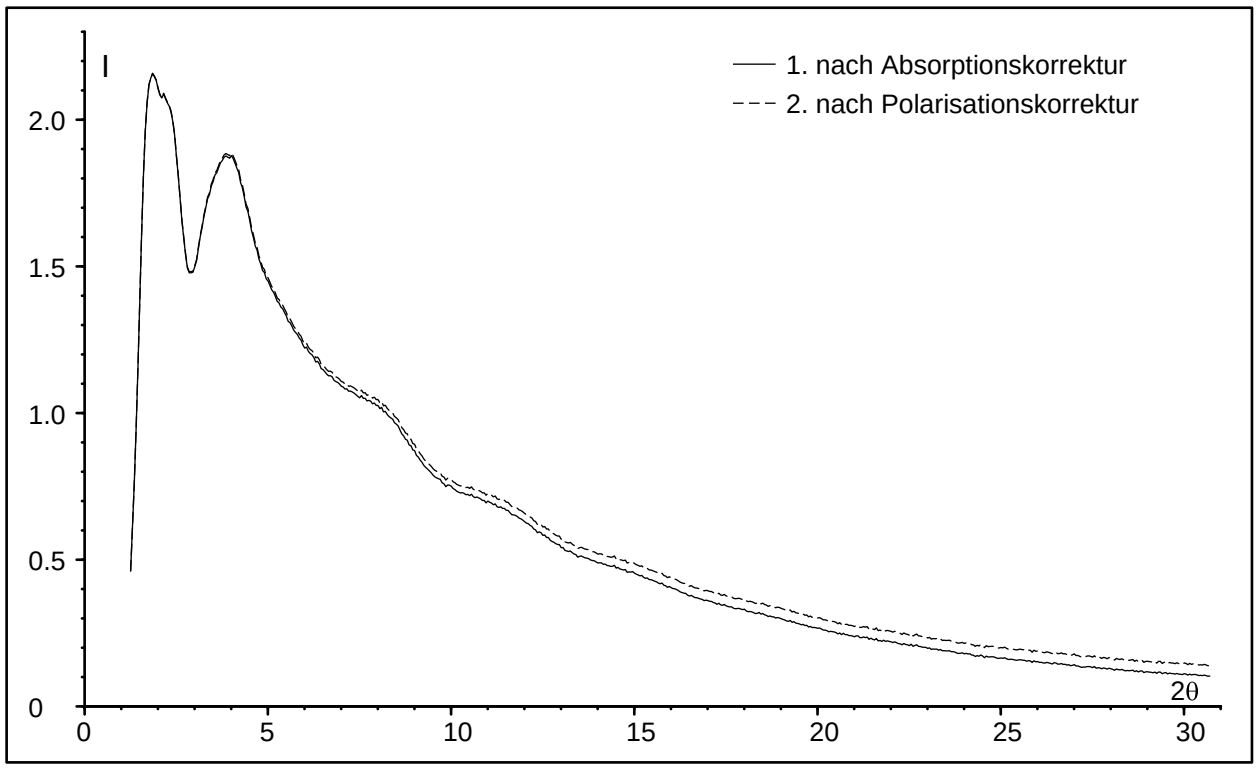


Abbildung 4.15: Diffraktogramm von $BiCl_3$ nach Absorptions- und Polarisationskorrektur

4.2.3 Die Umrechnung der Winkel in Beträge der Wellenvektoren

Die nachfolgenden Berechnungen finden in Einheiten der Wellenvektoren statt. Daher muss an dieser Stelle eine Umrechnung nach

$$\kappa = \frac{4\pi \sin\theta}{\lambda} \tag{4.13}$$

durchgeführt werden, d. h. die Abszissenwerte werden transformiert, wodurch die Abszissenwerte gegebenenfalls ihre Äquidistanz verlieren. Für die spätere Fouriertransformation werden jedoch äquidistante Werte benötigt, darüberhinaus müssen alle Abszissenwerte ein Vielfaches der Schrittweite darstellen.

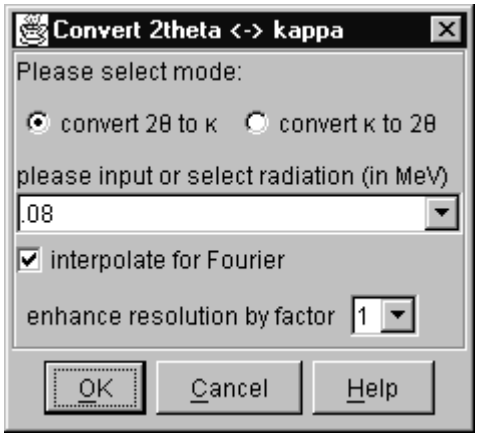


Abbildung 4.16: Dialog zur Umrechnung der Winkel in Beträge der Wellenvektoren

Die Parameter für diese Umrechnung können in einen Dialog der Klasse `ConvertDialog` eingegeben werden, der in Abbildung 4.16 dargestellt ist. Die Intrapolation kann optional durchgeführt werden. Die Grenzen des neuen Feldes ergeben sich aus der Umrechnung der Feldgrenzen des ursprünglichen Feldes. Als Schrittweite für die Intrapolation wird die minimale auftretende Schrittweite im konvertierten Feld gewählt. Soll eine kleinere Schrittweite verwendet werden, so lässt sich mit *enhance resolution by factor* ein ganzzahliger Faktor auswählen, um welchen die Schrittweite verkleinert wird.

Die Umrechnung erfolgt durch die Klasse `ThetaToKappaConverter` und die Intrapolation als davon unabhängige zweite Transformation mit dem Transformer `LinearInterpolation`. Da die Messwerte üblicherweise mit statistischem Rauschen behaftet sind, ist eine lineare Intrapolation nicht nur die einfachste sondern auch die sicherste Intrapolationsmethode.

4.2.4 Berechnung der Atomstreuung und des mittleren Atomformfaktors

Abbildung 4.18 zeigt einen Dialog der Klasse `AtomicScatteringDialog`. Mit seiner Hilfe können sowohl die Atomstreuung als auch der mittlere Atomformfaktor als Funktion des Streuvektorbetrages berechnet werden. Die genannten Größen können entweder für eine zuvor definierte Mischung, eine durch ihre Formel zu definierende Substanz, ein Atom oder

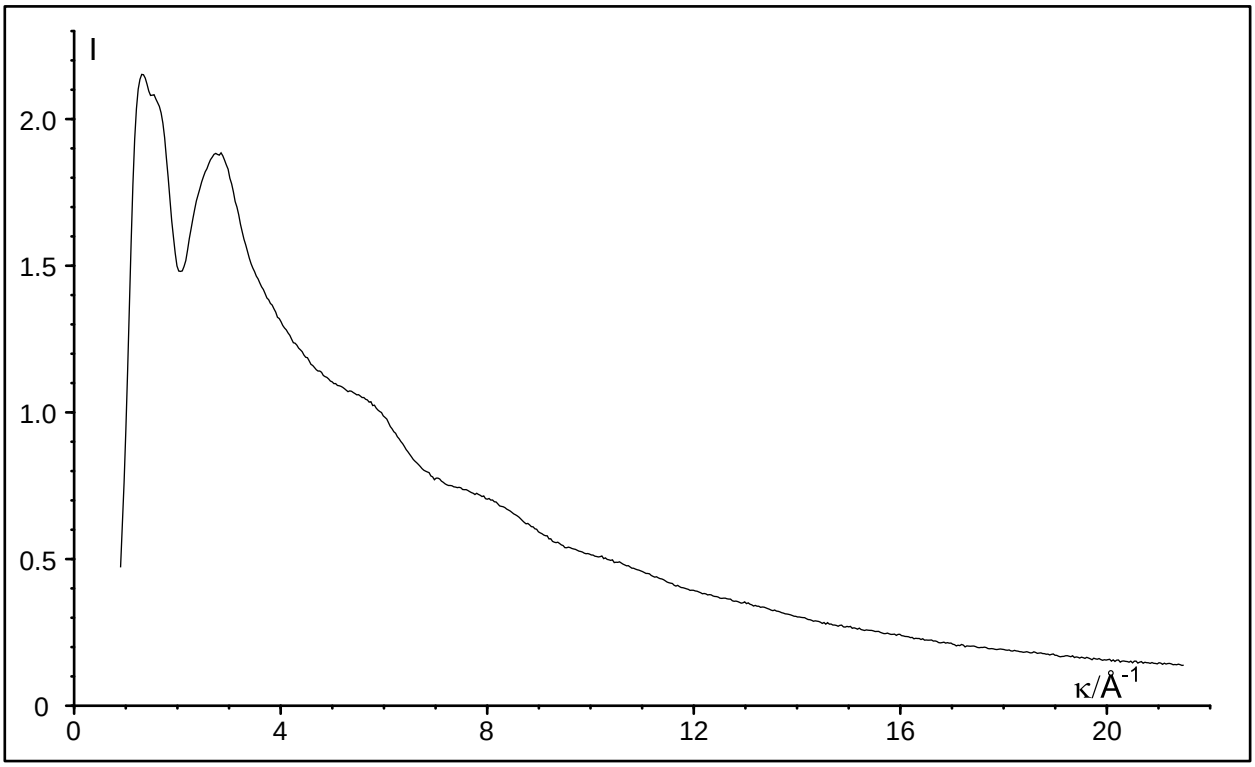


Abbildung 4.17: Diffraktogramm von $BiCl_3$ als Funktion von κ

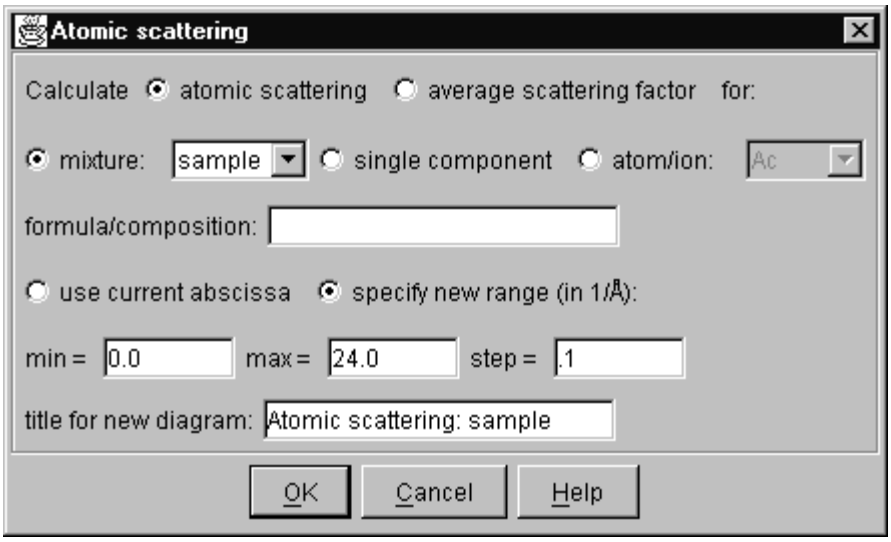


Abbildung 4.18: Dialog zur Berechnung der Atomstreuung

ein Ion berechnet werden. Als Abszissenwerte können die Werte des aktuellen Diagramms oder diejenigen eines durch Angabe der Parameter *min*, *max* und *step* neu zu definierenden Datenbereiches verwendet werden.

Die Daten werden in der Klasse `AtomicScattering` berechnet. Der statische Konstruktor dieser Klasse lädt die erforderlichen Parameter mit der Methode `loadParameters()` aus der Datei `atom_wk.dat` und trägt sie in eine `Map` ein. Die Werte in dieser Datei gehen auf Waasmeier und Kirfel [52] zurück. Sie entstanden durch eine Parametrisierung der Werte von Doyle/Turner/Coulthard/Cromer/Waber/Mann in den International Tables of Crystallography [30] und sind für die Elemente mit den Ordnungszahlen bis 98 (Californium) im Bereich $0 \leq \kappa < 24\pi$ tabelliert. Neben den Atomen sind auch Werte für eine Vielzahl von Ionen tabelliert. Aus den gegebenen Parametern p_0 bis p_{11} (der Parameter $p_{11} = 0$ existiert nur formal) und dem Debye-Waller-Faktor D berechnet die Methode `addAtomicScattering(...)` die Atomstreufaktoren für ein Element nach

$$f_0(\kappa) = \exp\left(-2D\kappa^2\right) \sum_{i=0}^{5} p_i \exp\left(\frac{-p_{i+6}\kappa^2}{16\pi^2}\right) \tag{4.14}$$

als Summe von Gaussfunktionen und daraus die Atomstreuung in Elektroneneinheiten (e. u.)

$$A(\kappa) = f_0^2(\kappa) \tag{4.15}$$

und addiert sie gegebenenfalls zu einer bereits berechneten Atomstreuung entsprechend ihrer molaren Menge. Beispielsweise setzt sich die Atomstreuung von Zinkchlorid zusammen aus

$$A_{\mathrm{ZnCl_2}} = A_{\mathrm{Zn}} + 2 \cdot A_{\mathrm{Cl}} \ . \tag{4.16}$$

Entsprechend wird für Mischungen verfahren und es gilt:

$$A_{\mathrm{mix}} = \sum_i n_i A_i \ . \tag{4.17}$$

Aus denselben Daten lässt sich der mittlere Atomformfaktor berechnen gemäß

$$f_e(\kappa) = \frac{\sum_i n_i \cdot f_{0,i}(\kappa)}{\sum_i n_i Z_i} \tag{4.18}$$

mit der Elektronenzahl Z_i der in der molaren Menge n_i auftretenden atomaren Komponente i. Die Elektronenzahl wird mit der Methode `getNumberOfElectrons(String symbol)` insbesondere auch für Ionen bestimmt wird. Der mittlere Atomformfaktor wird mit der Methode `double[] getAverageAtomicFormFactors(double[] kappa, CompositionInterface compi, double dw)` berechnet. Das in dieser Methode verwendete Interface `CompositionInterface` definiert die Methoden `int size()`, `String getSymbol(int i)`, `double getAmount(int i)` und `double getAmount(String symbol)` und wird von der bereits im Abschnitt 3.3.6 vorgestellten Klasse `Composition` implementiert. Im Gegensatz zu jener Klasse erlaubt die in `AtomicScatteringDialog` verwendete Klasse `FlexibleComposition`, welche ebenfalls dieses Interface implementiert, die Änderung der Zusammensetzung nach der Initialisierung und die Verwendung von Symbolen, die nicht in der Klasse `PSE` definiert sind, aber für spezielle Berechnungen benötigt werden wie z. B. `K+` für die Berechnung der Atomstreuung des Kaliumions.

In der Methode `XRDEvaluationMenu::atomicScattering()` wird aus den im Dialog eingegebenen Größen die entsprechende Kurve berechnet und ein neues `SingleDiagram` erzeugt.
Abbildung 4.19 zeigt ein Diagramm mit der Atomstreuung und dem mittleren Atomformfaktor von $BiCl_3$.

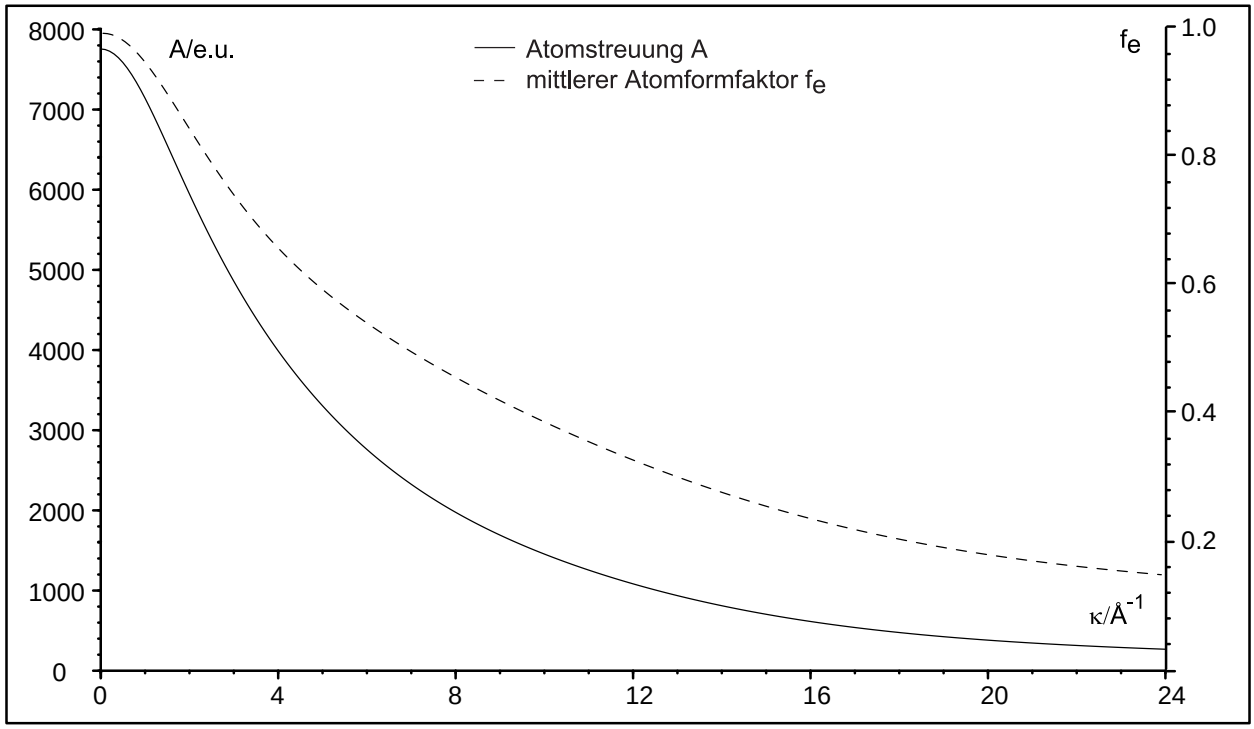


Abbildung 4.19: Atomstreuung und mittlerer Atomformfaktor von $BiCl_3$

4.2.5 Berechnung der Comptonstreuung

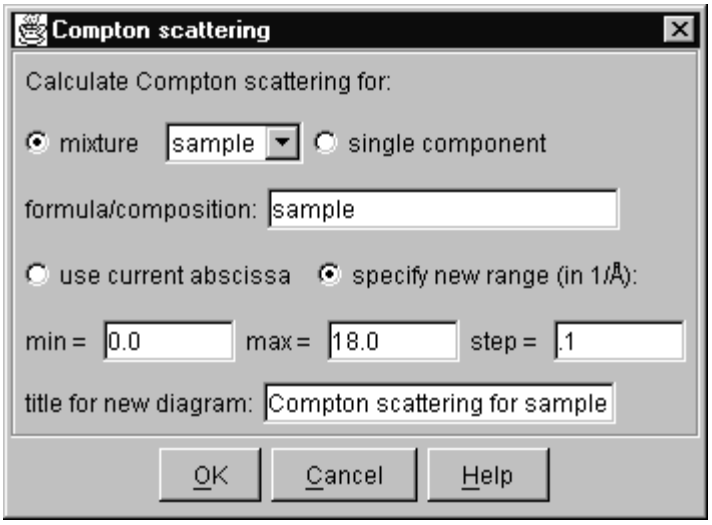


Abbildung 4.20: Dialog zur Berechnung der Comptonstreuung

Abbildung 4.20 zeigt einen Dialog der Klasse `ComptonScatteringDialog` zur Eingabe von Parametern für die Berechnung der Comptonstreuung. Die Berechnung der Comptonstreuung verläuft bezüglich des Programms in vielfacher Hinsicht analog zur Berechnung der Atomstreuung. Die Klasse `ComptonScattering` lädt Parameter aus der Datei `comp-`

`ton_by.dat`, welche auf einer Parametrisierung der Werte von Cromer und Mann ([53], [54]) durch Balyuzi [55] beruhen. Die Werte sind für die Elemente 1-92 im Bereich $0 \leq \kappa < 6\pi$ in Form von Parametern p_1 bis p_{11} tabelliert. Die Methode `addComptonScattering(int Z, double[] x, double[] y, double faktor)` berechnet die Comptonstreuung daraus mit

$$C(\kappa) = Z - \sum_{i=1}^{5} p_i \exp\left(\frac{p_{i+6}\kappa^2}{16\pi^2}\right) \tag{4.19}$$

wobei Z die Ordnungszahl des jeweiligen Elementes ist und die Comptonstreuung entsprechend der Molzahl der Atomsorte addiert wird:

$$C_{mix} = \sum_i n_i C_i \tag{4.20}$$

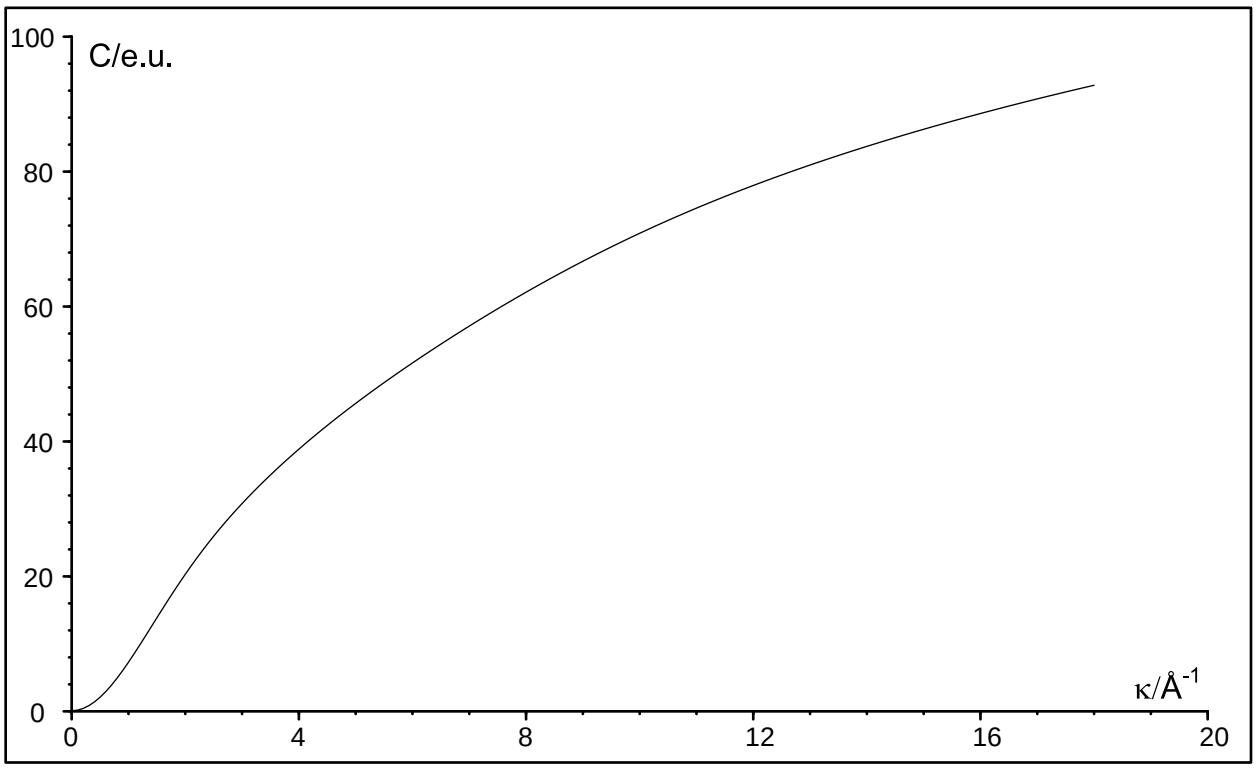


Abbildung 4.21: Comptonstreuung von $BiCl_3$

Abbildung 4.21 zeigt die Comptonstreuung von $BiCl_3$ als Funktion des Streuvektorbetrages κ.

4.2.6 Extrapolation und Normierung

Wurden die bisherigen Intensitäten in willkürlichen Einheiten (arbitrary units, a.u.) dargestellt, so ist für eine quantitative Auswertung der Ordinatenwerte in den nachfolgenden

Auswertungsschritten eine Normierung auf absolute Einheiten erforderlich. Die gesamte gestreute Intensität I setzt sich additiv aus der kohärent und der inkohärent (Comptonstreuung C) gestreuten Intensität zusammen. Der kohärente Anteil ist wiederum die Summe der Streuung von Elektronen desselben Atoms (Atomstreuung A) und verschiedener Atome. Letzterer Anteil verschwindet für große Werte von κ, d. h. die gesamte normierte Intensität konvergiert für große κ gegen die Summe der Atom- und Comptonstreuung. Da die Gesamtzahl der gestreuten Photonen unabhängig von der Anordnung der Elektronen in der Probe ist, verursacht die interatomare Elektronenverteilung gegenüber der Atomstreuung nur eine Umstrukturierung des Interferenzmusters, während die integrale Intensität der Streustrahlung konstant bleibt. Dadurch sind die Flächen unter der normierten Gesamtintensität einerseits und der Summe aus Atom- und Comptonstreuung andererseits gleich groß. Damit ergeben sich zwei Normierungsverfahren:

Verfügt man über qualitativ hochwertige Daten, die bis zu einem hinreichend großen Wert von κ vorliegen, so ist es möglich, die Normierungskonstante durch visuelle Kontrolle so zu wählen, dass die genannten Kurven für große Werte von κ konvergieren. Die Konstanz der integralen Intensität lässt sich verwenden, indem man die Summen der Abszissenwerte von gemessener Intensität einerseits und der Summe der Atom- und Comptonstreuung andererseits berechnet. Der Quotient entspricht dann der Normierungskonstanten α. Nimmt man an, dass beispielsweise durch Fluoreszenz ein konstanter Untergrund B vorliegt, so lässt sich dieses Verfahren dahingehend erweitern, dass man ansetzt

$$\int_{\kappa=0}^{\kappa_{max}} I(\kappa)\, d\kappa = B \cdot \kappa_{max} + \alpha \cdot \int_{\kappa=0}^{\kappa_{max}} \left[A_{mix}(\kappa) + C_{mix}(\kappa)\right] d\kappa \tag{4.21}$$

Betrachtet man I als Funktion der Summe $A_{mix}(\kappa) + C_{mix}(\kappa)$, so wird deutlich, dass Normierungskonstante und konstanter Untergrund durch lineare Regression der Datenpaare dieser Funktion bestimmt werden können, d. h. diese Größen werden so bestimmt, dass die Abweichungsquadrate der normierten Intensität von der Summe aus Atom- und Comptonstreuung minimal werden. In der Praxis eignet sich diese Methode jedoch nur für Abschätzungen der Normierungskonstante.

Bewährt hat sich hingegen ein von Krogh-Moe [56] vorgeschlagenes Verfahren, das unabhängig davon auch von Norman entwickelt wurde [57] - [59]. Die Normierungskonstante α lässt sich demnach durch folgende Formel berechnen:

$$-2\pi^2\rho_0^2\frac{V}{N} = \alpha \int_{\kappa=0}^{\kappa_{max}} I(\kappa)\,\kappa^2 d\kappa - \int_{\kappa=0}^{\kappa_{max}} \left[A_{mix}(\kappa) + C_{mix}(\kappa)\right]\kappa^2 d\kappa \tag{4.22}$$

mit dem Faktor „rhone“

$$rhone = \rho_0^2\frac{V}{N} = \bar{\rho}\cdot\bar{Z}^2 = \bar{Z}^2\cdot\rho\cdot\frac{N_A}{M}\ , \tag{4.23}$$

der Elektronenzahl

$$\bar{Z} = \sum_i n_i Z_i \tag{4.24}$$

des betrachteten Systems, der Massendichte ρ, der Avogadrokonstante N_A, der Molmasse M, der mittleren Teilchendichte $\bar{\rho} = N/V$ und der mittleren Ladungsdichte $\rho_0 = \bar{Z} \cdot N/V$. Krogh-Moe betont, dass die Herleitung dieser Gleichung unter der Annahme erfolgte, dass sich die Elektronendichteverteilungen der beteiligten Atome nicht überlappen. Der Fehler, den diese Annahme mit sich bringt, sei jedoch für hinreichend große Werte von κ zu vernachlässigen.

Mit gewöhnlichen Röntgendiffraktometern ist es nicht möglich, die Streuung bei kleinen Winkeln zu messen. Weidner, Geisenfelder und Zimmermann [11] schlagen daher vor, eine Extrapolation zu kleinen Werten von κ mit der empirischen Formel

$$I_{coh}(\kappa) = a_0 + a_2\kappa^2 + a_4\kappa^4 \tag{4.25}$$

durchzuführen. Die Konstante a_0 kann ebenso wie die Konstanten a_2 und a_4 durch Anpassung des Polynoms an die Messdaten erhalten werden, sofern diese bis zu hinreichend kleinen Werten von κ vorliegen. Häufig ist dies jedoch nicht der Fall. Aus der statistischen Mechanik ([60], [61]) ist bekannt, dass sich die Nullstreuung

$$I_{coh}(0) = a_0 = \frac{\rho}{M} \cdot R \cdot \chi_T \cdot T \cdot \bar{Z}^2 \tag{4.26}$$

aus der isothermen Kompressibilität χ_T berechnen lässt, die mit der adiabatischen Kompressibilität χ_S über

$$\chi_T = \chi_S + \frac{\beta^2 \cdot T \cdot M}{C_p \cdot \rho} \tag{4.27}$$

zusammenhängt. Darin sind β der thermischen Ausdehnungskoeffizient und C_p die Molwärme bei konstantem Druck. Tabelliert ist häufig auch der Kompressionsmodul $K = 1/\chi_T$. Der Kompressionsmodul K_l von Flüssigkeiten lässt sich aus Messungen der Schallgeschwindigkeit c berechnen:

$$K_l = c^2 \cdot \rho \tag{4.28}$$

In den Kompressionsmodul K_s von Festkörpern gehen sowohl die longitudinale Schallgeschwindigkeit c_l als auch die transversale Schallgeschwindigkeit c_t ein:

$$K_s = \rho \cdot \left(c_l^2 - \tfrac{4}{3}c_t^2\right) \tag{4.29}$$

Abbildung 4.22 zeigt einen Dialog der Klasse `NormDialog` zur Eingabe von Parametern für die Normierung. Die für die Normierung benötigten Daten für Atom-, Comptonstreuung und mittlere Atomformfaktoren werden wie oben beschrieben berechnet oder können als Datensatz ausgewählt werden. Dazu muss in letzterem Fall ein anderes Diagramm mit geeigneten Daten und identischen Abszissenwerten vorliegen. Es kann ein konstanter oder als Diagramm vorliegender Untergrund gewählt werden. Für $I_{coh}(0)$ kann ein Wert vorgegeben werden oder durch lineare Optimierung bestimmt werden. Für die Extrapolation kann die Anzahl der zu verwendenden Datenpunkte angegeben werden. Optional werden diese extrapolierten Werte in die Normierung einbezogen (Vorgabeeinstellung). Mehrere Kontrollkästchen erlauben die Auswahl des gewünschten Normierungsverfahrens. Für eine manuelle Normierung bei hinreichender Konvergenz für große Werte von κ lässt sich der Normierungsfaktor eingeben und über die Schaltflächen +/– in kleinen Schritten variieren. Die Normierung nach Krogh-Moe

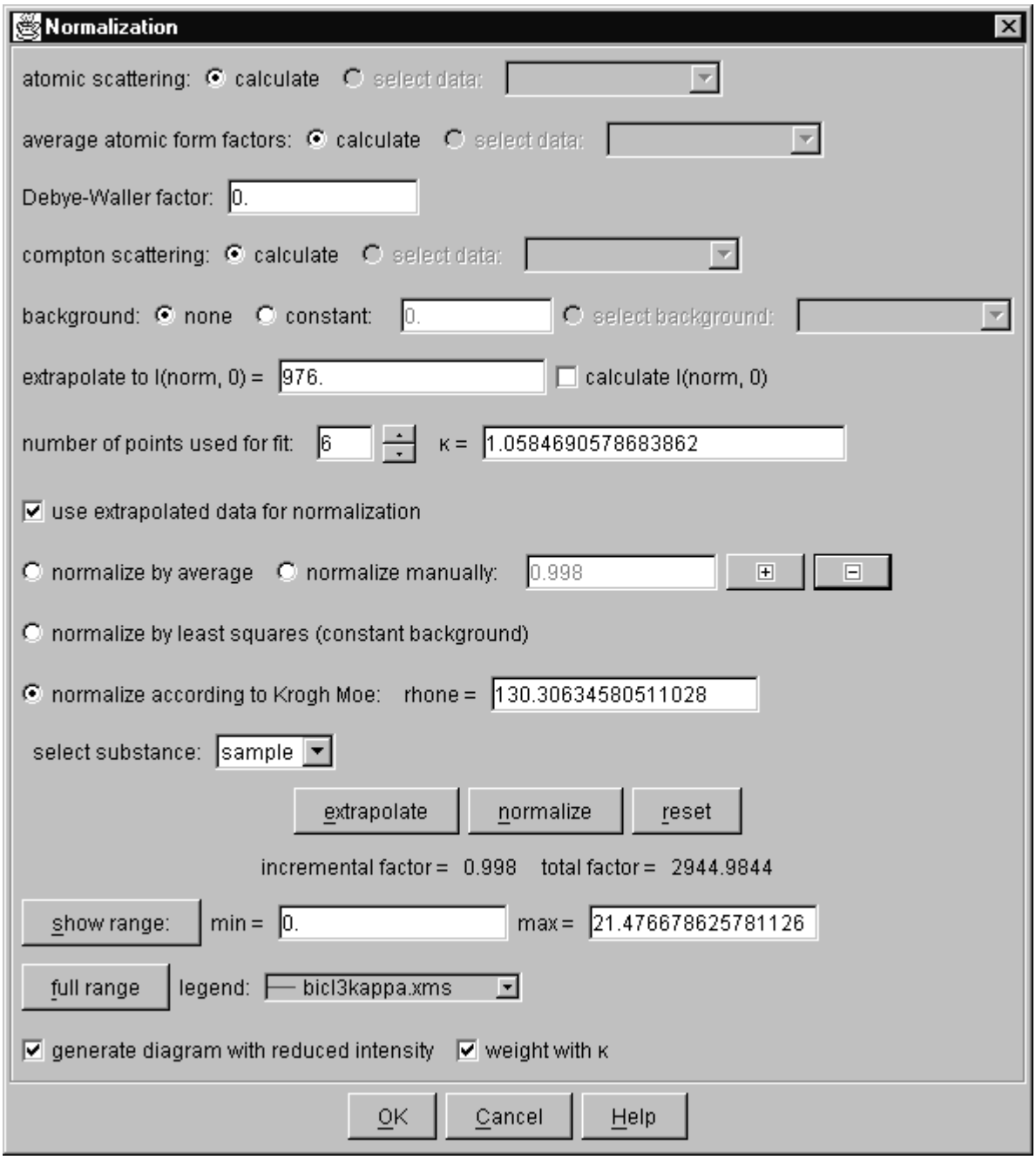


Abbildung 4.22: Dialog zur Normierung

erfordert die Angabe des „rhone“-Faktors, welcher gegebenenfalls aus den Registraturdaten für eine auszuwählende Substanz berechnet wird. Mit der Schaltfläche *extrapolate* wird nur eine Extrapolation durchgeführt, mit *normalize* wird die Normierung berechnet und mit *reset* wird der letzte Normierungsschritt rückgängig gemacht. Die Angabe *incremental factor* zeigt den zuletzt berechneten inkrementellen Normierungsfaktor an und der Wert *total factor* den gesamten Normierungsfaktor bei mehreren Normierungsschritten. Da sich bei der Normierung die für die Extrapolation verwendeten Daten verändern, handelt es sich

bei der Normierung um ein iteratives Verfahren, das jedoch meist nach wenigen Iterationen konvergiert, d. h. es wird *incremental factor = 1.0*. Um die Qualität der Normierung zu kontrollieren, kann mit der Schaltfläche *show range* und den beiden Textfeldern *min* und *max* oder der Schaltfläche *full range* der anzuzeigende Bereich des Diagramms bestimmt werden. Schließlich kann angegeben werden, welche Diagramme als Ergebnis erzeugt werden sollen: die normierte Intensität und/oder die reduzierte Intensität mit oder ohne κ-Wichtung.

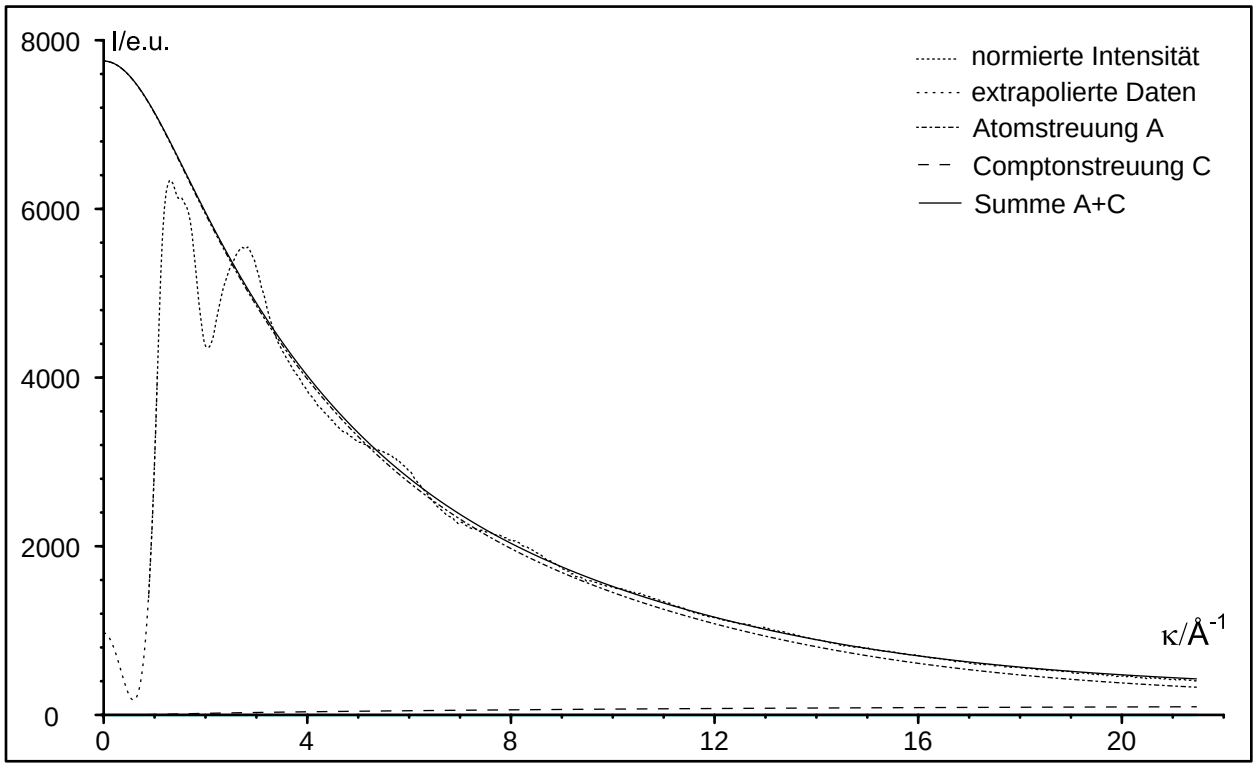


Abbildung 4.23: Normierung des korrigierten Diffraktogramms von $BiCl_3$

Abbildung 4.23 zeigt die Normierung von $BiCl_3$. Man erkennt die gute Konvergenz der normierten Intensität gegen die Summe von Atom- und Comptonstreuung. Zur Extrapolation wurden die ersten 10 Messwerte verwendet. Da die isotherme Kompressibilität von $BiCl_3$ bei 270 °C nicht bekannt ist, wurde ein für Flüssigkeiten typischer mittlerer Wert von $1 \cdot 10^{-9} \frac{1}{Pa}$ verwendet, der mit den weiteren gegebenen physikalischen Daten eine berechnete Nullstreuung von 976 Elektroneneinheiten ergibt. Die Normierung wurde nach Krogh-Moe berechnet und zur Verbesserung der Konvergenz geringfügig (Faktor 0,998) manuell skaliert.

Im Anschluss an die Normierung wurde entsprechend der Ausführungen in Abschnitt 5.1.3 eine erneute Absorptionskorrektur nach Kendig-Pings berechnet, welche die abweichende Absorption der Comptonstreuung aufgrund ihrer Wellenlängenverschiebung berücksichtigt. Da aufgrund der hohen Strahlungsenergie nur bis zu kleinen Beugungswinkeln gemessen wurde, bei welchen die Wellenlängenverschiebung relativ klein ausfällt, unterscheidet sich das so erhaltene Ergebnis nur kaum von dem in Abbildung 4.23 dargestellten. Dieses Ergeb-

nis wurde jedoch für die weiteren Berechnungen verwendet.

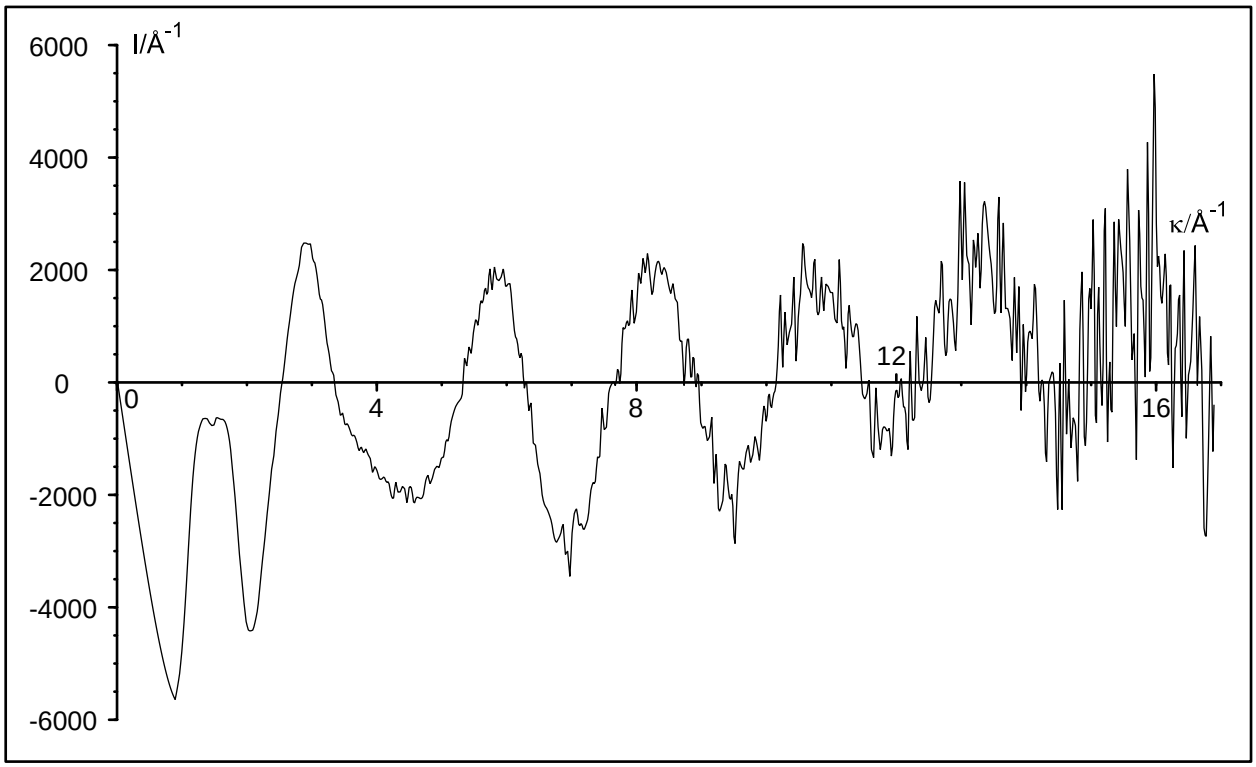


Abbildung 4.24: κ-gewichtete reduzierte Intensität von $BiCl_3$

Abbildung 4.24 zeigt die κ-gewichtete reduzierte Intensität von $BiCl_3$. Erkennbar ist das durch die Wichtung der kleinen Differenz zwischen normierter Kurve und Summe von Atom- und Comptonstreuung zunehmende Rauschen für große Werte von κ. Die Kurve wurde daher auf den angezeigten Bereich beschnitten.

Die Ausführung der Normierung im Programm ist auf mehrere Klassen verteilt. Einfachere Berechnungen werden direkt in der Klasse `NormDialog` durchgeführt. Atom- und Comptonstreuung werden mit Hilfe der oben beschriebenen Klassen berechnet. Die Normierung nach Krogh-Moe ist in die Klasse `KroghMoe` ausgelagert. Die Integralnormierungen finden in der Klasse `Statistics` statt. Die Bestimmung der Extrapolationsparameter durch lineare Optimierung erfolgt in der Klasse `LinearFit` (siehe Abschnitt 5.5). Die Klassen `LegendRenderer` und `IntegerSelector` dienen der Anzeige der Legende und der Auswahl ganzer Zahlen mit Pfeiltasten.

4.2.7 Die Fouriertransformation

Wie bereits im Kapitel Grundlagen ausgeführt wurde, ist die korrigierte und normierte gemessene Intensität mit der Gesamtkorrelationsfunktion über eine Fouriertransformation ver-

knüpft. Formel 2.5

$$G(r) = \frac{\sum\limits_{i,j} n_i n_j \bar{z}_i \bar{z}_j g_{ij}(r)}{\left(\sum\limits_i n_i Z_i\right)^2} = 1 + \frac{1}{2\pi^2 r \bar{\rho} \left(\sum\limits_i n_i Z_i\right)^2} \int\limits_0^\infty \kappa \cdot \underbrace{\frac{I(\kappa) - I_{Atom}(\kappa)}{f_e^2(\kappa)}}_{I_{red}} \sin(\kappa r)\, d\kappa$$

lässt sich durch Substitution auf eine Sinustransformation zurückführen. Die theoretischen Grundlagen der Sinustransformation und der Fouriertransformation und die dafür verwendeten Methoden werden im Kapitel 5.2 behandelt. Anstelle der mittleren effektiven Elektronenzahlen $\bar{z}$ werden die Elektronenzahlen als gute Näherung verwendet.

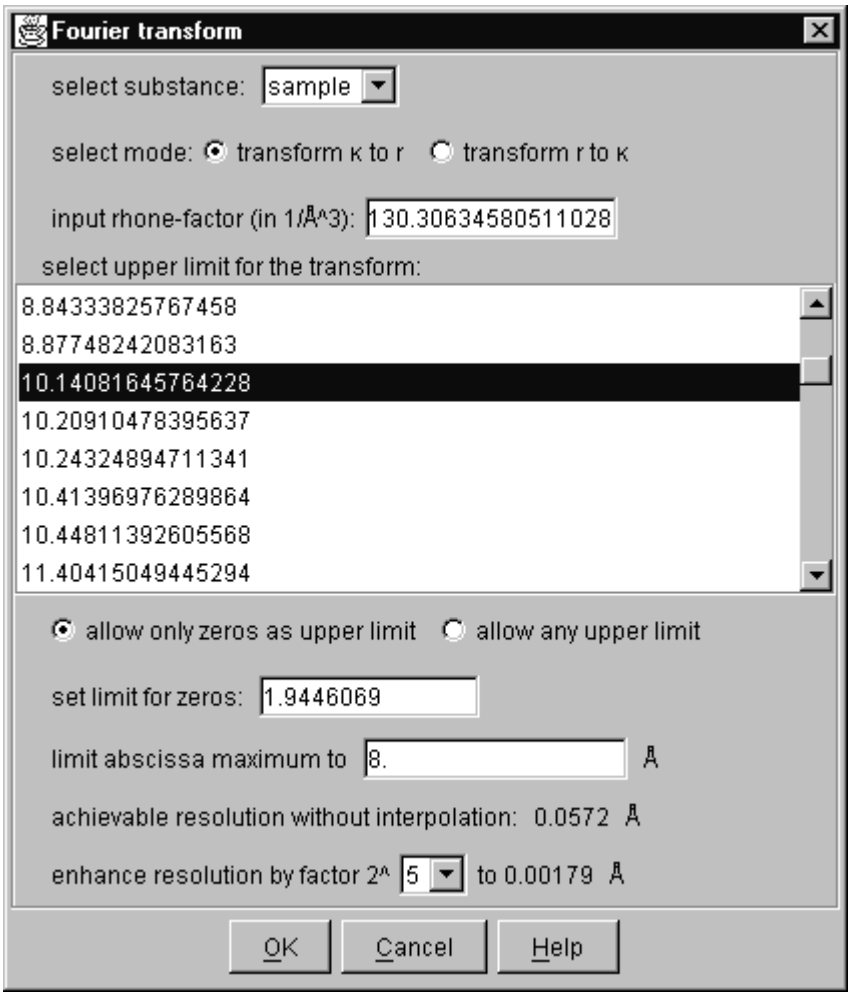


Abbildung 4.25: Dialog zur Transformation der reduzierten Intensität

Abbildung 4.25 zeigt einen Dialog der Klasse `FourierDialog`, mit welchem sich die Fouriertransformation der reduzierten Intensität kontrollieren lässt. Eingabeparameter sind die Transformationsrichtung, der in voriger Gleichung auftretende „rhone“-Faktor, der optional für eine zu wählende Probe berechnet wird und die obere Integrationsgrenze. Im vorliegenden Fall wurde dafür eine Nullstelle bei $\kappa_{max} = 10{,}14\ \text{Å}^{-1}$ gewählt. Als Integrationsgrenzen

können aus einer Liste entweder beliebige Werte oder Nullstellen ausgewählt werden, wobei als Nullstellen Datenpaare gelten, nach denen ein Vorzeichenwechsel erfolgt oder die von null maximal um eine Konstante ϵ abweichen. Der voreingestellte Wert von ϵ beträgt 0,1 % der maximalen Schwankung der Ordinatenwerte und kann über ein Textfeld verändert werden (*set limit for zeros*). Die gewählte Grenze wird dabei im zu transformierenden Diagramm angezeigt. Die Wahl der Integrationsgrenze unterliegt einer gewissen Willkür, hat aber einen deutlichen Einfluss auf das Ergebnis der Transformation. Folgende Kriterien sind zu berücksichtigen: Um Abbrucheffekte durch die Fouriertransformation zu minimieren, sollte eine Nullstelle der reduzierten Intensität gewählt werden. Wählt man den Wert zu groß, so transformiert man keine Daten, sondern Rauschen, welches durch die Wichtung mit κ verstärkt wurde. Auf der anderen Seite sollte der Wert so groß wie möglich gewählt werden, da sich aus dem Abtasttheorem der Fouriertransformation ergibt, dass die maximale Auflösung Δ_r der transformierten Funktion über $\Delta_r = 1/\kappa_{max}$ mit der oberen Integrationsgrenze κ_{max} zusammenhängt. Diese ist nach Gleichung 2.1 mit der Wellenlänge λ und dem maximalen Beugungswinkel θ_{max} verknüpft. Es gilt also:

$$\Delta_r = \frac{\lambda}{4\pi \sin \theta_{max}} \tag{4.30}$$

Für den theoretisch maximal erreichbaren Winkel $2\theta = 180°$ sind in der folgenden Tabelle maximal erreichbare Auflösungen einiger Strahlungstypen angegeben:

Strahlung	Wellenlänge	Auflösung
Cu K_α	1,542 Å	0,12 Å
Mo K_α	0,7109 Å	0,057 Å
Ag K_α	0,5610 Å	0,045 Å
80 keV	0,155 Å	0,012 Å

Die angegebenen Auflösungen sind die bestmöglichsten, die aufgrund der Fouriertransformation zu erreichen sind. Die tatsächlichen Auflösungen liegen jedoch meist deutlich niedriger. Im Fall der Messung von $BiCl_3$ erhält man mit κ_{max} =10,14 $Å^{-1}$ eine Auflösung von 0,098 Å. Im Dialog ist die sich numerisch ergebende Auflösung von ca. 0,0572 Å angezeigt, die sich daraus ergibt, dass bei der Schnellen Fouriertransformation stets Feldlängen verwendet werden, welche Zweierpotenzen entsprechen. Entspricht der Integrationsgrenze keine entsprechende Feldlänge, so werden die Felder auf eine geeignete Länge vergrößert, wobei die Ordinatenwerte gleich null gesetzt werden. Mathematisch entspricht dieses Auffüllen (zero padding) einer trigonometrischen Intrapolation nach der Transformation und führt zu einer besseren virtuellen Auflösung. Dieser Effekt kann auch gezielt herbeigeführt werden. Im Dialog kann dazu eine Zweierpotenz angegeben werden, um welche die Auflösung verbessert werden soll.

In gleicher Weise gilt nach dem Abtasttheorem für den maximal bestimmbaren Abstand bei der Sinustransformation

$$r_{max} = \frac{1}{\Delta_\kappa} \text{ und damit } \sin \Delta_\theta = \frac{\lambda}{4\pi r_{max}} \tag{4.31}$$

Wählt man $r_{max} = 8$ Å als einen Abstand, bis zu dem maximal auswertbare Informationen vorliegen können, so ist hierfür bei Verwendung der in dieser Hinsicht relativ ungünstigen Ag K_α-Strahlung eine Geräteauflösung von ca. $\Delta_{2\theta}$ =0,64° erforderlich, die mit gängigen Diffraktometern problemlos erreicht wird. Aufgrund der Intrapolation der Werte, die nach der Umrechnung in den κ-Bereich durchgeführt wird, erhält man vor der Transformation eine hohe Auflösung, welche nach der Transformation zu einem großen Bereich mit Nullen oder Rauschen führt und keine physikalische Relevanz besitzt. Im Dialog kann daher der Bereich auf einen sinnvollen Wert beschränkt werden (*limit abscissa*).

Die genannten Parameter werden der inneren Klasse `XRDEvaluationMenu$XRDSineTransformer` in einem `ParameterSet` übergeben, in welcher die Transformation ausgeführt wird.

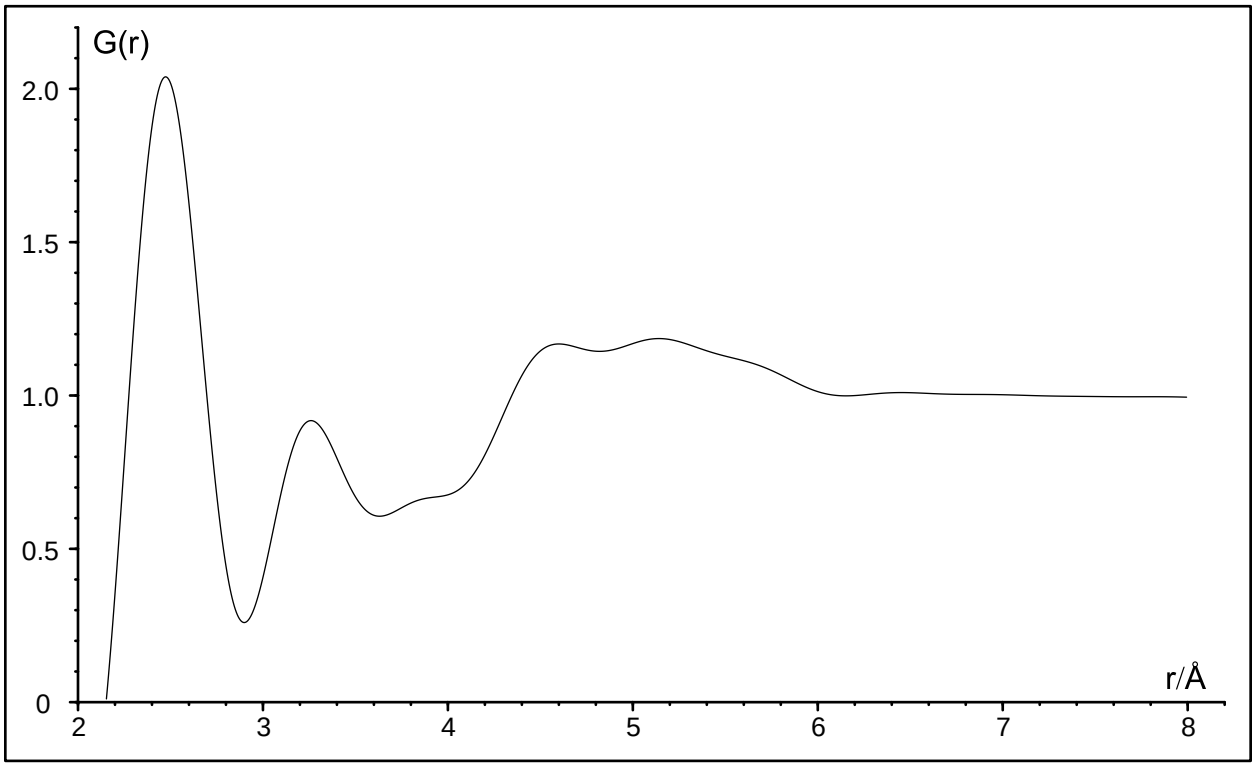


Abbildung 4.26: Gesamtkorrelationsfunktion von $BiCl_3$

In Abbildung 4.26 ist die Gesamtkorrelationsfunktion von $BiCl_3$ dargestellt. Der Bereich für $r < 2$ Å wird von Abbruchfehlern der Fouriertransformation überlagert und wurde entfernt.

4.2.8 Die Bestimmung von Koordinationszahlen

Durch Anpassung einer Summe von Gaussfunktionen an die radiale Verteilungsfunktion RDF ist die Bestimmung von Koordinationszahlen möglich:

$$RDF\,(r) = 4\pi r^2 G\,(r) = \sum_i a_i' \cdot e^{-\left(\frac{r-b_i}{c_i}\right)^2} \tag{4.32}$$

Alternativ lässt sich an die Gesamtkorrelationsfunktion $G(r)$ eine Summe von $1/r^2$-gewichteten Gaussfunktionen

$$f_i^G(r) = a_i \frac{1}{r^2} \cdot e^{-\left(\frac{r-b_i}{c_i}\right)^2} \tag{4.33}$$

anpassen:

$$G(r) = \sum_i f_i^G \tag{4.34}$$

Die Gesamtkorrelationsfunktion ist eine gewichtete Summe von Atompaarkorrelationsfunktionen

$$G(r) = \frac{\sum_{i,j} n_i n_j \bar{z}_i \bar{z}_j g_{ij}(r)}{\left(\sum_k n_k Z_k\right)^2} \tag{4.35}$$

$$= \sum_{i,j} w_{ij} g_{ij} \tag{4.36}$$

mit den Wichtungsfaktoren w_{ij}. Für eine Atompaarkorrelationsfunktion gilt

$$g(r) = \frac{K}{s\bar{\rho}\sqrt{32\pi^3}} \frac{1}{r^2} \cdot \exp\left(-\frac{(r-r_0)^2}{2s^2}\right) \tag{4.37}$$

mit der Koordinationszahl K, der mittleren Teilchendichte $\bar{\rho} = \rho \cdot N_A/M$, der Schwingungsamplitude s und dem Abstand der Atomzentren r_0. Gelingt es nun, ein Maximum der Gesamtkorrelationsfunktion eindeutig einem Maximum einer Atompaarkorrelationsfunktion zuzuordnen, so kann man aus den Parametern a_i, b_i und c_i der gewichteten Gaussfunktion und dem Wichtungsfaktor die Koordinationszahl bestimmen:

$$K = \frac{4a_i \cdot c_i \cdot \bar{\rho} j \sqrt{\pi^3}}{w} \tag{4.38}$$

Darin ist j der Index, mit dem eine Element in einer Korrelationsfunktion auftritt z. B. $j = 3$ für Cl in der (Bi, Cl)-Paarkorrelationsfunktion von $BiCl_3$.

Abbildung 4.27 zeigt einen Dialog der Klasse `CoordinationNumberDialog` zur Bestimmung von Koordinationszahlen. Beim Drücken der Schaltfläche *Search peaks* werden durch Analyse der Kurvenform Startwerte für die Anpassung der Gaussfunktionen ermittelt und in eine Tabelle eingetragen. Mit *Add peak* kann der Tabelle ein weiteres Maximum hinzugefügt werden, *Remove peaks* entfernt in der Liste ausgewählte Maxima und *Clear table* löscht alle Tabelleneinträge. Sind die Maxima deutlich voneinander getrennt, so können aus den Startparametern mit Hilfe der Schaltfläche *fit* sofort gute Näherungswerte erhalten werden. Enthält die Kurve hingegen Schultern oder schwach ausgeprägte Maxima, so müssen die Startwerte zunächst interaktiv optimiert werden, um später eine Anpassung erfolgreich berechnen zu können. Dazu können die Werte in der Tabelle editiert werden. Die der gewählten Reihe entsprechende Gaussfunktion wird dabei im zugehörigen Diagramm angezeigt. Da die Parameter Amplitude, Position und Breite der gewichteten Gaussfunktion nicht auf einfache Weise mit

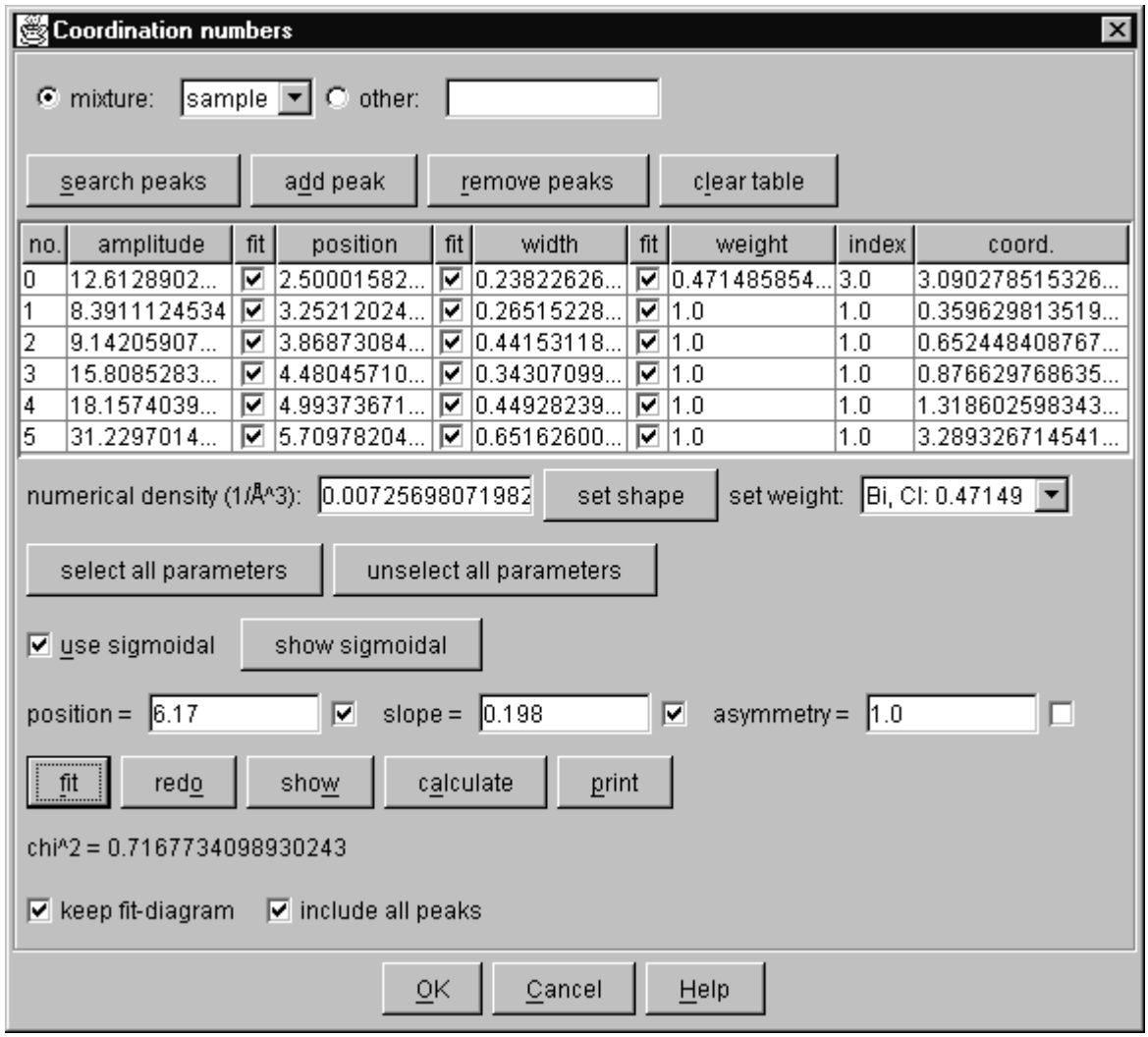


Abbildung 4.27: Dialog zur Bestimmung von Koordinationszahlen

der Form der Funktion zusammenhängen, können stattdessen nach Drücken der Schaltfläche *Set shape* die entsprechenden Werte für die Form einer nicht-gewichtete Gaussfunktion eingegeben werden, die dann näherungsweise umgerechnet werden. Für größere Werte von r ist eine Vielzahl schlecht aufgelöster Minima und Maxima zu erwarten, die um den Grenzwert 1 oszillieren. Da sich diese nur schlecht anpassen lassen, kann dafür optional eine Sigmoide $f^S(r)$ der Form

$$f^S(r) = \left(1 + \exp\left(\frac{a-x}{c}\right)\right)^{-c} \tag{4.39}$$

verwendet werden, deren Parameter die Lage (position a), Steigung (slope b) und Asymmetrie (asymmetry c) angeben. Für jeden Parameter der Gausskurven und der Sigmoide kann angegeben werden, ob er für die Anpassung variiert werden soll. Die Schaltfläche *Show* berechnet die Summe aller angepassten Kurven mit den aktuellen Parametern und stellt diese im Diagramm dar. Eine mit *fit* durchgeführte Optimierung kann mit *undo* rückgängig gemacht werden. Der ausgegebene Faktor *chi^2* gibt die Genauigkeit der Anpassung als

Summe quadratischer Abweichungen an. Mit *calculate* wird schließlich die Koordinationszahl berechnet. Dazu muss eine Zuordnung eines Maximums zu einer Paarkorrelation gemacht werden. Mit *set weight* kann für die gewählte Substanz und für ein Maximum der Wichtungsfaktor eines Atompaars ausgewählt werden oder direkt in die Tabelle eingetragen werden. Unter *index* muss weiter der Index des zu betrachtenden Ligandenatoms angegeben werden. Die mittlere Teilchendichte *numerical density* wird aus den Substanzdaten berechnet oder kann eingegeben werden. Ein Diagramm mit den angepassten Kurven kann erzeugt werden.

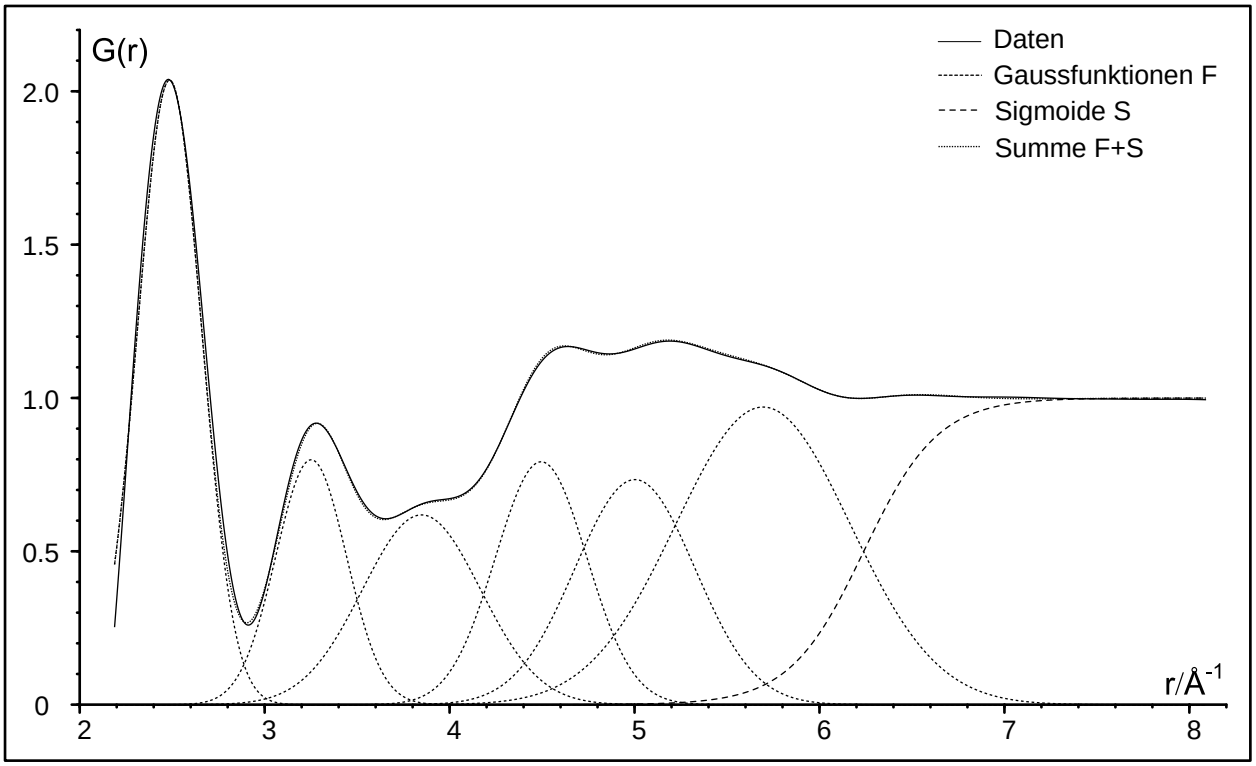


Abbildung 4.28: Anpassung von Gaussfunktionen an die Daten

Ein solches Diagramm ist in Abbildung 4.28 dargestellt. Die Summe der Anpassungskurven stimmt sehr gut mit den Daten überein. Trotz der Vielzahl an Parametern, die erforderlich sind, um die einzelnen Maxima zu modellieren, konvergieren Optimierungen für verschiedene Startparameter zum gezeigten Ergebnis, dessen Parameter in Abbildung 4.27 angezeigt werden. Im Bereich größerer Werte von r besteht jedoch eine gewisse Mehrdeutigkeit. Das erste Maximum ist zwar eindeutig zu modellieren, jedoch auch von Abbrucheffekten beeinflusst und weicht erkennbar von der Gaussfunktion ab.

Die Berechnungen werden in der Klasse `CoordinationNumberDialog` nach den oben angegebenen Formeln durchgeführt. Die Parameter der gewichteten Gaussfunktionen werden in der Klasse `LevenbergMarquardt` optimiert (siehe Abschnitt 5.6). Die innere Klasse `CoordinationNumberDialog$GaussFunctions` kapselt die gewichteten Gaussfunktionen $f_i^G(r)$, die Sigmoide $f^S(r)$ und deren partielle Ableitungen nach den Parametern. Die

Funktionen werden als Objekte vom Typ `VectorFunctionPointer` und `Parametrized-FunctionPointer` mit den tabellierten Startparametern und einem Feld mit Angabe der zu optimierenden Parameter dem Konstruktor der Klasse `LevenbergMarquardt` übergeben. Die verbesserten Parameter können dort mit der Methode `getParameters()` abgerufen werden.

In Abbildung 4.27 wurde die Koordinationszahl für das erste Maximum bei 2,5 Å bestimmt. Es wurde einer Bi-Cl-Korrelation zugeordnet. Der zugehörige Wichtungsfaktor beträgt 0,4715 und der Index für eine Koordination von Chlor um Bismut 3. Daraus ergibt sich nach Gleichung 4.38 mit den dargestellten Anpassungsparametern eine Koordinationszahl von 3,1. Diese Ergebnisse stehen in guter Übereinstimmung mit den meisten der aus der Literatur bekannten Ergebnissen:

Literatur	Zustand	Methode	Abstand	CN
Lemke [62]	Schmelze 240 °C	EDXD	2,70 Å	5,8
Fukushima/Suzuki [63]	Schmelze 270 °C	ND	2,50 Å	3,0
Price et al. [64]	Schmelze 300 °C	ND	2,5 Å	3,0
Nyburg et al. [65]	Kristall (orthorhomb.)	ADXD	2,46 Å/2,51 Å	1,0/2,0
Bartl [66]	Kristall (orthorhomb.)	ND	2,50 Å	3,0
Heusel/Eberhardinger	Schmelze 270 °C	ADXD	2,5 Å	3,1

Es sind mit ND Neutronenbeugung (Neutron diffraction), mit ADXD winkeldispersive Röntgenbeugung (Angular dispersive x-ray diffraction), mit EDXD energiedispersive Röntgenbeugung (Energy dispersive x-ray diffraction) und mit CN die Koordinationszahl bezeichnet.

Kapitel 5

Numerische Klassen

5.1 Die Absorptionskorrektur nach Kendig und Pings

Mit dieser Korrektur von Kendig und Pings [67] können Röntgendiffraktogramme flüssiger und fester amorpher Proben in zylindischen Kapillaren bei variabler Strahlbreite und Geometrie auf Absorption korrigiert werden. Ziel der Korrektur ist es, aus einer Messung der gefüllten Kapillare und einer weiteren Messung der leeren Kapillare denjenigen Anteil numerisch zu isolieren, der von einer Probe ohne Eigenabsorption und umgebende Kapillare herrühren würde. Bereits von Paalman und Pings [68] wurde eine solche Korrektur ermittelt, deren Anwendbarkeit sich jedoch auf Fälle beschränkt, in denen die Messzelle vollständig bestrahlt wird. Zwar ist es wünschenswert, mit einem schmalen Strahl und entsprechend einer dünnen Kapillare zu arbeiten, doch lässt sich dies technisch nicht immer realisieren.

5.1.1 Formalismus der Absorptionskorrektur

Da der dieser Korrektur zugrundeliegende Artikel [67] mehrere Fehler enthält, sollen Teile der Herleitung hier nachvollzogen oder wiederholt werden.

Eine grundlegende Formel wurde bereits von Paalman und Pings [68] hergeleitet:

$$I_s(\kappa) = \frac{1}{A_{s,sc}(\kappa)} \left[I^E_{c+s}(\kappa) - \frac{A_{c,sc}(\kappa)}{A_{c,c}(\kappa)} I^E_c(\kappa) \right]. \tag{5.1}$$

Hierin sind

$I_s(s)$	gesuchte kohärente Streuintensität der Probe ohne Zelle
$A_{s,sc}(\kappa)$	Absorptionsfaktor für Streuung in der Probe und Absorption in Probe und Zelle
$A_{c,sc}(\kappa)$	Absorptionsfaktor für Streuung in der Zelle und Absorption in Probe und Zelle
$A_{c,c}(\kappa)$	Absorptionsfaktor für Streuung und Absorption in der Zelle
$I^E_{c+s}(\kappa)$	gemessene Streuintensität der Zelle mit Probe
$I^E_c(\kappa)$	gemessene Streuintensität der leeren (evakuierten) Zelle

Vorige Gleichung ist exakt, falls man wie üblich davon ausgeht, dass langreichweitige Korrelationen zwischen Probe und Zelle auszuschließen sind. Mit dem Lambert-Beer-Gesetz für Röntgenabsorption 4.1 folgt für die Absorptionsfaktoren:

$$A_{s,sc}(\kappa) = \frac{1}{V_s}\int_{V_s} \exp\left(-\mu_s l_s(r,\theta) - \mu_c l_c(r,\theta)\right) d\vec{r} \tag{5.2}$$

$$A_{c,sc}(\kappa) = \frac{1}{V_c}\int_{V_c} \exp\left(-\mu_s l_s(r,\theta) - \mu_c l_c(r,\theta)\right) d\vec{r} \tag{5.3}$$

$$A_{c,c}(\kappa) = \frac{1}{V_c}\int_{V_c} \exp\left(-\mu_c l_c(r,\theta)\right) d\vec{r} \tag{5.4}$$

Hierin sind

μ_s und μ_c	lineare Absorptionskoeffizienten von Probe s und Zelle c
V_s und V_c	Integrationsvolumina von Probe und Zelle
θ	halber Streuwinkel
$\vec{r}$	Position, an welcher die Streuung stattfindet (Integrationsvariable)
l_s und l_c	Weglängen der Strahlung durch Probe und Zelle

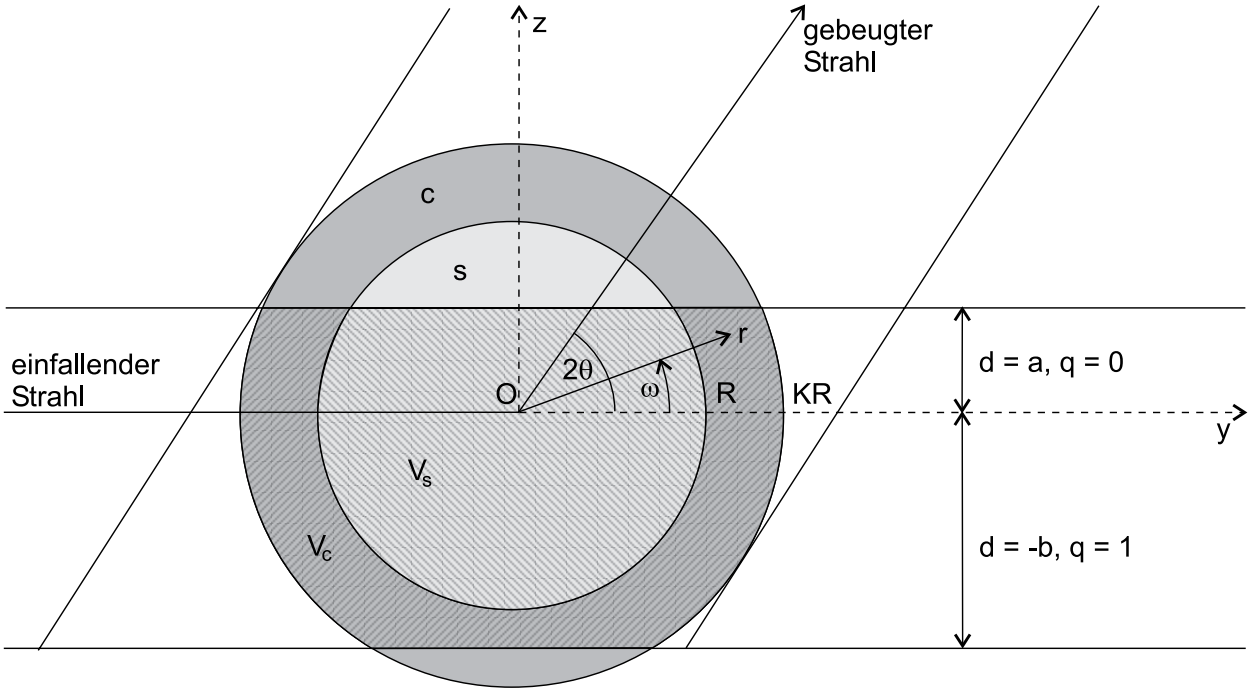


Abbildung 5.1: Strahlengang bei der Absorptionskorrektur nach Kendig-Pings

Dabei befinden sich die in Abbildung 5.1 schraffiert dargestellten Projektionen der Volumina V_s und V_c im einfallenden Strahl und tragen so durch Streuung zur gemessenen Intensität bei, während sich die Weglängen l_s und l_c auf den gesamten durchstrahlten Bereich, d. h. des ein- und ausfallenden Strahl beziehen. In diesen Ausdrücken ist eine Separation der Abhängigkeit

von s in einen wellenlängenabhängigen Teil $\mu(\lambda)$ und einen winkelabhängigen Teil $l(r,\theta)$ zu erkennen. Es wird angenommen, dass die Strahlung kollimiert und ihre Intensitätsverteilung über die Breite konstant sei. Die Höhe der zylinderförmigen Kapillare sei groß genug, um vollständig im Strahl zu liegen.

Zur weiteren Beschreibung wird ein kartesisches Koordinatensystem verwendet, dessen Ursprung sich auf der Kapillarachse befindet, welche die x-Achse definiert. Es genügt dann, geometrische Betrachtungen in einer Ebene senkrecht zur Kapillarachse anzustellen. Es sind der Nomenklatur der Originalliteratur folgend R der Innenradius und KR der Außenradius der Kapillare, a und b die Begrenzungen des Strahls in Richtung der z-Achse in den Grenzen $a > b$, $\infty > a > -R$ und $R > b > -\infty$.

Die Integration über die Volumina V_s und V_c bzw. deren Projektion in die genannte Ebene wird sinnvollerweise in Polarkoordinaten ausgeführt, wobei $\omega = 0$ die Richtung des einfallenden Strahls kennzeichne. Durch die Wahl der Werte a, b, R und KR ergeben sich 15 unterschiedliche Fälle für die zu lösenden Integrale. Die Fallunterscheidung ist in der folgenden Tabelle angegeben. Diese und weitere Tabellen und Formeln dieses Abschnitts sind im Anhang in einer größeren Schrift enthalten.

Fall	Bereiche von a und b	Ausdrücke für Absorptionsfaktoren
1	$\infty > a \geq KR$	$A_{s,sc} = \{2A_s(R)\}^{-1}\left\{SR(R,0)_{s,sc} + SR(R,1)_{s,sc}\right\}$
	$-KR \geq b > -\infty$	$A_{c,sc} = \{2A_{c1}(KR)\}^{-1}\left\{CR(KR,0)_{c,sc} + CR(KR,1)_{c,sc}\right\}$
		$A_{c,c} = \{2A_{c1}(KR)\}^{-1}\left\{CR(KR,0)_{c,c} + CR(KR,1)_{c,c}\right\}$
2	$\infty > a \geq KR$	$A_{s,sc} = \{2A_s(R)\}^{-1}\left\{SR(R,0)_{s,sc} + SR(R,1)_{s,sc}\right\}$
	$-R \geq b > -\infty$	$A_{c,sc} = \{A_{c1}(KR) + A_{c1}(b)\}^{-1}\left\{CR(KR,0)_{c,sc} + CR(b,1)_{c,sc} + CT(b,1)_{c,sc}\right\}$
		$A_{c,c} = \{A_{c1}(KR) + A_{c1}(b)\}^{-1}\left\{CR(KR,0)_{c,c} + CR(b,1)_{c,c} + CT(b,1)_{c,c}\right\}$
3a	$\infty > a \geq KR$	$A_{s,sc} = \{A_s(R) + A_s(b)\}^{-1}\left\{SR(R,0)_{s,sc} + SR(b,1)_{s,sc} + ST(b,1)_{s,sc}\right\}$
	$0 \geq b > -R$	$A_{c,sc} = \{A_{c1}(KR) + A_{c2}(b)\}^{-1}\left\{CR(KR,0)_{c,sc} + CT(b,1)_{c,sc}\right\}$
		$A_{c,c} = \{A_{c1}(KR) + A_{c2}(b)\}^{-1}\left\{CR(KR,0)_{c,c} + CT(b,1)_{c,c}\right\}$
3b	$\infty > a \geq KR$	$A_{s,sc} = \{A_s(R) - A_s(b)\}^{-1}\left\{SR(R,0)_{s,sc} - SR(b,0)_{s,sc} - ST(b,0)_{s,sc}\right\}$
	$R > b > 0$	$A_{c,sc} = \{A_{c1}(KR) - A_{c2}(b)\}^{-1}\left\{CR(KR,0)_{c,sc} - CT(b,0)_{c,sc}\right\}$
		$A_{c,c} = \{A_{c1}(KR) - A_{c2}(b)\}^{-1}\left\{CR(KR,0)_{c,c} - CT(b,0)_{c,c}\right\}$
4	$KR > a \geq R$	$A_{s,sc} = \{2A_s(R)\}^{-1}\left\{SR(R,0)_{s,sc} + SR(R,1)_{s,sc}\right\}$
	$-KR \geq b > -\infty$	$A_{c,sc} = \{A_{c1}(a) + A_{c1}(KR)\}^{-1}\left\{CR(a,0)_{c,sc} + CT(a,0)_{c,sc} + CR(KR,1)_{c,sc}\right\}$
		$A_{c,c} = \{A_{c1}(a) + A_{c1}(KR)\}^{-1}\left\{CR(a,0)_{c,c} + CT(a,0)_{c,c} + CR(KR,1)_{c,c}\right\}$
5	$KR > a \geq R$	$A_{s,sc} = \{2A_s(R)\}^{-1}\left\{SR(R,0)_{s,sc} + SR(R,1)_{s,sc}\right\}$
	$-R \geq b > -KR$	$A_{c,sc} = \{A_{c1}(a) + A_{c1}(b)\}^{-1}\left\{CR(a,0)_{c,sc} + CT(a,0)_{c,sc} + CR(b,1)_{c,sc} + CT(b,1)_{c,sc}\right\}$
		$A_{c,c} = \{A_{c1}(a) + A_{c1}(b)\}^{-1}\left\{CR(a,0)_{c,c} + CT(a,0)_{c,c} + CR(b,1)_{c,c} + CT(b,1)_{c,c}\right\}$
6a	$KR > a \geq R$	$A_{s,sc} = \{A_s(R) + A_s(b)\}^{-1}\left\{SR(R,0)_{s,sc} + SR(b,1)_{s,sc} + ST(b,1)_{s,sc}\right\}$
	$0 \geq b > -R$	$A_{c,sc} = \{A_{c1}(a) + A_{c2}(b)\}^{-1}\left\{CR(a,0)_{c,sc} + CT(a,0)_{c,sc} + CT(b,1)_{c,sc}\right\}$
		$A_{c,c} = \{A_{c1}(a) + A_{c2}(b)\}^{-1}\left\{CR(a,0)_{c,c} + CT(a,0)_{c,c} + CT(b,1)_{c,c}\right\}$
6b	$KR > a \geq R$	$A_{s,sc} = \{A_s(R) - A_s(b)\}^{-1}\left\{SR(R,0)_{s,sc} - SR(b,0)_{s,sc} - ST(b,0)_{s,sc}\right\}$
	$R > b > 0$	$A_{c,sc} = \{A_{c1}(a) - A_{c2}(b)\}^{-1}\left\{CR(a,0)_{c,sc} + CT(a,0)_{c,sc} - CT(b,0)_{c,sc}\right\}$
		$A_{c,c} = \{A_{c1}(a) - A_{c2}(b)\}^{-1}\left\{CR(a,0)_{c,c} + CT(a,0)_{c,c} - CT(b,0)_{c,c}\right\}$
7a	$R > a \geq 0$	$A_{s,sc} = \{A_s(a) + A_s(R)\}^{-1}\left\{SR(a,0)_{s,sc} + ST(a,0)_{s,sc} + SR(R,1)_{s,sc}\right\}$
	$-KR \geq b > -\infty$	$A_{c,sc} = \{A_{c2}(a) + A_{c1}(KR)\}^{-1}\left\{CT(a,0)_{c,sc} + CR(KR,1)_{c,sc}\right\}$
		$A_{c,c} = \{A_{c2}(a) + A_{c1}(KR)\}^{-1}\left\{CT(a,0)_{c,c} + CR(KR,1)_{c,c}\right\}$
7b	$0 > a > -R$	$A_{s,sc} = \{A_s(R) - A_s(a)\}^{-1}\left\{SR(R,1)_{s,sc} - SR(a,1)_{s,sc} - ST(a,1)_{s,sc}\right\}$
	$-KR \geq b > -\infty$	$A_{c,sc} = \{A_{c1}(KR) - A_{c2}(a)\}^{-1}\left\{CR(KR,1)_{c,sc} - CT(a,1)_{c,sc}\right\}$
		$A_{c,c} = \{A_{c1}(KR) - A_{c2}(a)\}^{-1}\left\{CR(KR,1)_{c,c} - CT(a,1)_{c,c}\right\}$
8a	$R > a \geq 0$	$A_{s,sc} = \{A_s(a) + A_s(R)\}^{-1}\left\{SR(a,0)_{s,sc} + ST(a,0)_{s,sc} + SR(R,1)_{s,sc}\right\}$
	$-R \geq b > -KR$	$A_{c,sc} = \{A_{c2}(a) + A_{c1}(b)\}^{-1}\left\{CT(a,0)_{c,sc} + CR(b,1)_{c,sc} + CT(b,1)_{c,sc}\right\}$
		$A_{c,c} = \{A_{c2}(a) + A_{c1}(b)\}^{-1}\left\{CT(a,0)_{c,c} + CR(b,1)_{c,c} + CT(b,1)_{c,c}\right\}$
8b	$0 > a > -R$	$A_{s,sc} = \{A_s(R) - A_s(a)\}^{-1}\left\{SR(R,1)_{s,sc} - SR(a,1)_{s,sc} - ST(a,1)_{s,sc}\right\}$
	$-R \geq b > -KR$	$A_{c,sc} = \{A_{c1}(b) - A_{c2}(a)\}^{-1}\left\{CR(b,1)_{c,sc} + CT(b,1)_{c,sc} - CT(a,1)_{c,sc}\right\}$
		$A_{c,c} = \{A_{c1}(b) - A_{c2}(a)\}^{-1}\left\{CR(b,1)_{c,c} + CT(b,1)_{c,c} - CT(a,1)_{c,c}\right\}$
9a	$R > a \geq 0$	$A_{s,sc} = \{A_s(a) + A_s(b)\}^{-1}\left\{SR(a,0)_{s,sc} + ST(a,0)_{s,sc} + SR(b,1)_{s,sc} + ST(b,1)_{s,sc}\right\}$
	$0 \geq b > -R$	$A_{c,sc} = \{A_{c2}(a) + A_{c2}(b)\}^{-1}\left\{CT(a,0)_{c,sc} + CT(b,1)_{c,sc}\right\}$
		$A_{c,c} = \{A_{c2}(a) + A_{c2}(b)\}^{-1}\left\{CT(a,0)_{c,c} + CT(b,1)_{c,c}\right\}$
9b	$R > a > 0$	$A_{s,sc} = \{A_s(a) - A_s(b)\}^{-1}\left\{SR(a,0)_{s,sc} + ST(a,0)_{s,sc} - SR(b,0)_{s,sc} - ST(b,0)_{s,sc}\right\}$
	$R > b > 0$	$A_{c,sc} = \{A_{c2}(a) - A_{c2}(b)\}^{-1}\left\{CT(a,0)_{c,sc} - CT(b,0)_{c,sc}\right\}$
		$A_{c,c} = \{A_{c2}(a) - A_{c2}(b)\}^{-1}\left\{CT(a,0)_{c,c} - CT(b,0)_{c,c}\right\}$
9c	$0 > a > -R$	$A_{s,sc} = \{A_s(b) - A_s(a)\}^{-1}\left\{SR(b,1)_{s,sc} + ST(b,1)_{s,sc} - SR(a,1)_{s,sc} - ST(a,1)_{s,sc}\right\}$
	$0 > b > -R$	$A_{c,sc} = \{A_{c2}(b) - A_{c2}(a)\}^{-1}\left\{CT(b,1)_{c,sc} - CT(a,1)_{c,sc}\right\}$
		$A_{c,c} = \{A_{c2}(b) - A_{c2}(a)\}^{-1}\left\{CT(b,1)_{c,c} - CT(a,1)_{c,c}\right\}$

Die in dieser Tabelle verwendeten Elementarintegrale sind:

$$SR(d,q)_{s,sc} = \int_0^{|d|}\int_0^{\pi} \exp\left(-\mu_s l_s(r,\omega+q\pi,\theta) - \mu_c l_c(r,\omega+q\pi,\theta)\right) d\omega r dr \qquad (5.5)$$

$$ST(d,q)_{s,sc} = \int_{|d|}^{R}\int_0^{\arcsin(|d|/r)} \exp(-\mu_s l_s(r,\omega+q\pi,\theta) - \mu_c l_c(r,\omega+q\pi,\theta))$$

$$+\exp(-\mu_s l_s(r,\pi-\omega+q\pi,\theta)-\mu_c l_c(r,\pi-\omega+q\pi,\theta))\,d\omega r dr \quad (5.6)$$

$$CR(d,q)_{c,sc} = \int_R^{|d|}\int_0^{\pi} \exp(-\mu_s l_s(r,\omega+q\pi,\theta)-\mu_c l_c(r,\omega+q\pi,\theta))\,d\omega r dr \quad (5.7)$$

$$CT(d,q)_{c,sc} = \int_{R_{low}}^{KR}\int_0^{\arcsin(|d|/r)} \exp(-\mu_s l_s(r,\omega+q\pi,\theta)-\mu_c l_c(r,\omega+q\pi,\theta))$$
$$+\exp(-\mu_s l_s(r,\pi-\omega+q\pi,\theta)-\mu_c l_c(r,\pi-\omega+q\pi,\theta))\,d\omega r dr \quad (5.8)$$

$$CR(d,q)_{c,c} = \int_R^{|d|}\int_0^{\pi} \exp(-\mu_c l_c(r,\omega+q\pi,\theta))\,d\omega r dr \quad (5.9)$$

$$CT(d,q)_{c,c} = \int_{R_{low}}^{KR}\int_0^{\arcsin(|d|/r)} \exp(-\mu_c l_c(r,\omega+q\pi,\theta))$$
$$+\exp(-\mu_c l_c(r,\pi-\omega+q\pi,\theta))\,d\omega r dr \quad (5.10)$$

Dabei ist d der durch die vorige Tabelle als a oder b gegebene Abstand der Strahlbegrenzung von der Kapillarachse auf der Seite q, wobei der betrachtete Punkt vom einfallenden Strahl aus gesehen links ($q=0$) oder rechts ($q=1$) der Kapillarachse liegt (siehe Abbildung 5.1). Der in den Integralen CT auftretende Wert R_{low} ist

$$R_{low} = \begin{cases} d & \text{für} \quad KR \geq d \geq R \\ R & \text{für} \quad R > d > 0 \end{cases} \quad (5.11)$$

Es sind weiter

$$A_s(d) = (\pi/2)R^2 - R^2\arccos(|d|/R) + |d|\sqrt{R^2-d^2} \quad (5.12)$$

$$A_{c1}(d) = (\pi/2)(K^2R^2-R^2) - K^2R^2\arccos(|d|/KR) + |d|\sqrt{K^2R^2-d^2}$$
$$\text{mit } |d| \geq R \quad (5.13)$$

$$A_{c2}(d) = (\pi/2)(K^2R^2-R^2) - K^2R^2\arccos(|d|/KR) + |d|\sqrt{K^2R^2-d^2}$$
$$+R^2\arccos(|d|/R) - |d|\sqrt{R^2-d^2} \text{ mit } |d| < R \quad (5.14)$$

Nun werden noch Ausdrücke für die Weglängen in den Elementarintegralen benötigt. Für die verschiedenen Verläufe des Strahls durch Probe (s) und Zelle (c) und den Ort (sc), an welchem die Streuung stattfindet, ergeben sich für die Polarkoordinate r verschiedene Weglängen l_s und l_c und Integrationsbereiche für ω:

Fall	Bereich r	Bereich ω bei gegebenem r	Strahlverlauf	Weglängen
1	$R \geq r > 0$	$2\pi > \omega \geq 0$	c, s, sc, s, c	$l_s = \nu+\eta-\sigma$ $l_c = \alpha+\beta-\nu-\eta$
2a	$R/\cos\theta > r > R$	$2\theta+\pi-\arcsin(R/r) \geq \omega \geq \arcsin(R/r)$	c, sc, c	$l_s = 0$
2b	$KR \geq r \geq R/\cos\theta$	$2\theta+\pi-\arcsin(R/r) \geq \omega \geq \arcsin(R/r)$ und		$l_c = \alpha+\beta-\sigma$
2c		$2\pi-\arcsin(R/r) \geq \omega \geq 2\theta+\pi+\arcsin(R/r)$		
3a	$R/\cos\theta > r > R$	$2\pi-\arcsin(R/r) \geq \omega > 2\theta+\pi-\arcsin(R/r)$	c, sc, c, s, c	$l_s = 2\eta$
3b	$KR \geq r \geq R/\cos\theta$	$2\theta+\pi+\arcsin(R/r) > \omega > 2\theta+\pi-\arcsin(R/r)$		$l_c = \alpha+\beta-\sigma-2\eta$
4a	$R/\cos\theta > r > R$	für $\pi+2\theta+\arcsin(R/r) \leq 2\pi$: $2\pi > \omega \geq \pi+2\theta+\arcsin(R/r)$ und $\arcsin(R/r) > \omega > 0$	c, s, c, sc, c	$l_s = 2\nu$ $l_c = \alpha+\beta-\sigma-2\nu$
4b		für $\pi+2\theta+\arcsin(R/r) > 2\pi$: $\arcsin(R/r) > \omega \geq 2\theta-\pi+\arcsin(R/r)$		
4c	$KR \geq r \geq R/\cos\theta$	$\arcsin(R/r) > \omega \geq 0$ und $2\pi > \omega > 2\pi-\arcsin(R/r)$		
5a	$R/\cos\theta > r > R$	für $\pi+2\theta+\arcsin(R/r) \leq 2\pi$: $\pi+2\theta+\arcsin(R/r) > \omega > 2\pi-\arcsin(R/r)$ für $\pi+2\theta+\arcsin(R/r) > 2\pi$: $2\pi > \omega > 2\pi-\arcsin(R/r)$ und $\pi+2\theta+\arcsin(R/r) > \omega \geq 0$	c, s, c, sc, c, s, c	$l_s = 2\nu+2\eta$ $l_c = \alpha+\beta-\sigma-2\nu-2\eta$
5b	$KR \geq r \geq R/\cos\theta$	Bereich tritt nicht auf		

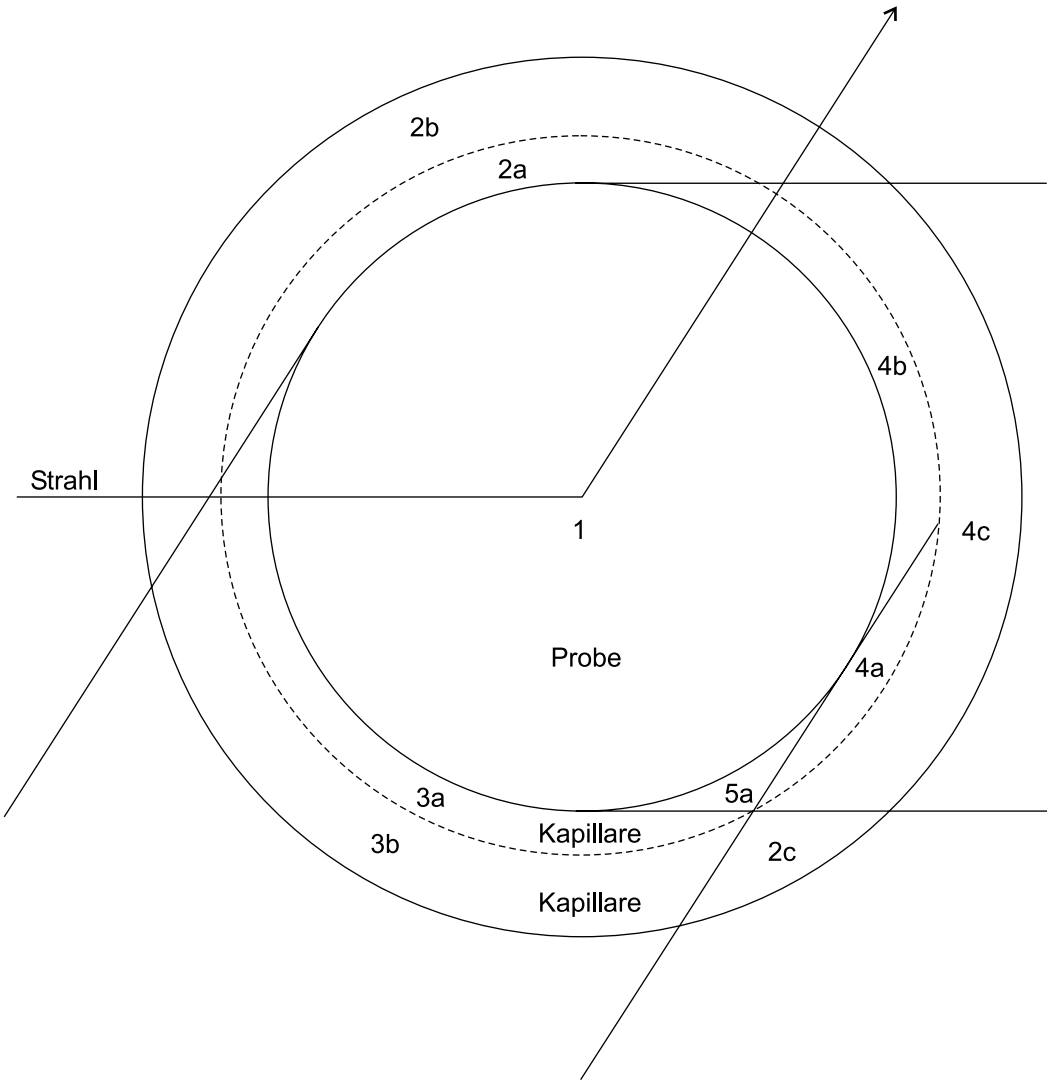


Abbildung 5.2: Fallunterscheidung der Integrationsintervalle für die Kendig-Pings Absorptionskorrektur

Abbildung 5.2 zeigt die in der Tabelle aufgeführten Teilbereiche, über welche integriert wird. In der vorigen Tabelle wurden für die Weglängen folgende Hilfsgrößen verwendet:

$$\alpha(r,\omega) = \sqrt{K^2R^2 - r^2\sin^2\omega} \tag{5.15}$$

$$\beta(r,\omega,\theta) = \sqrt{K^2R^2 - r^2\sin^2(\omega - 2\theta)} \tag{5.16}$$

$$\nu(r,\omega) = \sqrt{R^2 - r^2\sin^2\omega} \tag{5.17}$$

$$\eta(r,\omega,\theta) = \sqrt{R^2 - r^2\sin^2(\omega - 2\theta)} \tag{5.18}$$

$$\sigma(r,\omega,\theta) = 2r\sin(\omega - \theta)\sin\theta \tag{5.19}$$

5.1.2 Numerische Auswertung der Integrale

Um die Elementarintegrale numerisch auswerten zu können, wurden von Kendig und Pings Näherungen eingeführt, mit denen die Integrale durch zweimalige Anwendung der Trapezformel unter Weglassung der Werte an den Integrationsgrenzen als Summation berechnet werden können. Die Flächenelemente der Ringsegmente werden dabei anhand von Unterteilungsgraden m_s so gewählt, dass sie etwa gleich groß sind.

Für die begrenzenden Werte der Probensegmente erhält man damit

$$r = R \cdot (m/m_s) \text{ mit } m = 1, 2, 3, \ldots, m_s \tag{5.20}$$
$$\omega = \pi n/(3m) \text{ mit } n = 1, 2, 3, \ldots, 6m \tag{5.21}$$

Zur Auswertung der Integrale wird der Integrand in der Mitte dieser Segmente berechnet, also an den Stellen

$$r_m = (R/m_s)\left(m - \tfrac{1}{2}\right) \text{ mit } m = 1, 2, 3, \ldots, m_s \tag{5.22}$$
$$\omega_{mn} = (\pi/(3m))\left(n - \tfrac{1}{2}\right) \text{ mit } n = 1, 2, 3, \ldots, 6m \tag{5.23}$$

In gleicher Weise unterteilt man die Messzelle mit Dicke $(K-1)R$ so in Segmente, dass diese etwa dieselbe Dicke haben wie die Probensegmente

$$m_c = [(K-1)m_s] + 1. \tag{5.24}$$

Darin ist [ ] die auch im Folgenden mehrfach verwendete Gaußklammerfunktion, welche eine Dezimalzahl auf eine ganze Zahl abrundet. Damit ergibt sich für die begrenzenden Werte und die Werte des Integranden in der Messzelle

$$r = R + R \cdot (K-1)\,(m/m_c) \text{ mit } m = 1, 2, 3, \ldots, m_c \tag{5.25}$$
$$\omega = \pi n/\,(3m_s + 3m) \text{ mit } n = 1, 2, 3, \ldots, 6(m_s + m) \tag{5.26}$$
$$r_m = R + R \cdot (K-1)\left(m - \tfrac{1}{2}\right)/m_c \text{ mit } m = 1, 2, 3, \ldots, m_c \tag{5.27}$$
$$\omega_{mn} = \pi\left(n - \tfrac{1}{2}\right)/\,(3m_s + 3m) \text{ mit } n = 1, 2, 3, \ldots, 6(m_s + m) \tag{5.28}$$

Aus den elementaren Doppelintegralen erhält man dadurch folgende Doppelsummen:

$$\begin{aligned} SR(d,q)_{s,sc} = {} & \frac{\pi R^2}{3m_s^2} \sum_{m=1}^{m_d} (1 - 1/(2m)) \sum_{n=1}^{3m} \exp\{-\mu_s l_s(r_m, \omega_{mn} + q\pi, \theta) \\ & -\mu_c l_c(r_m, \omega_{mn} + q\pi, \theta)\} \text{ mit } m_d = \left[m_s(|d|/R) + \tfrac{1}{2}\right] \end{aligned} \tag{5.29}$$

$$\begin{aligned} ST(d,q)_{s,sc} = {} & \frac{\pi R^2}{3m_s^2} \sum_{m=m_d+1}^{m_s} (1 - 1/(2m)) \sum_{n=1}^{n_d} (\exp\{-\mu_s l_s(r_m, \omega_{mn} + q\pi, \theta) \\ & -\mu_c l_c(r_m, \omega_{mn} + q\pi, \theta)\} + \exp\{-\mu_s l_s(r_m, \pi - \omega_{mn} + q\pi, \theta) \\ & -\mu_c l_c(r_m, \pi - \omega_{mn} + q\pi, \theta)\}) \\ & \text{mit } n_d = \left[(3m/\pi)\arcsin(|d|/r_m) + \tfrac{1}{2}\right] \end{aligned} \tag{5.30}$$

$$CR(d,q)_{c,sc} = \sum_{m=1}^{m_d} \frac{\pi R^2}{3(m_s + m)} \left\{ \frac{K-1)^2}{m_c^2}(m - \tfrac{1}{2}) + \frac{K-1}{m_c} \right\}$$

$$\sum_{n=1}^{3(m_s+m)} \exp\{-\mu_s l_s(r_m, \omega_{mn}+q\pi, \theta) - \mu_c l_c(r_m, \omega_{mn}+q\pi, \theta)\}$$

$$\text{mit } m_d = \begin{cases} \left[\frac{m_c}{K-1}\left(\frac{|d|}{R}-1\right)+\frac{1}{2}\right] & \text{für} \quad d \geq R \\ 0 & \text{für} \quad d < R \end{cases} \tag{5.31}$$

$$CT(d,q)_{c,sc} = \sum_{m=m_d+1}^{m_c} \frac{\pi R^2}{3(m_s+m)} \left\{ \frac{(K-1)^2}{m_c^2}(m-\tfrac{1}{2}) + \frac{K-1}{m_c} \right\}$$

$$\sum_{n=1}^{n_d} \exp\{-\mu_s l_s(r_m, \omega_{mn}+q\pi, \theta) - \mu_c l_c(r_m, \omega_{mn}+q\pi, \theta)\}$$

$$+ \exp\{-\mu_s l_s(r_m, \pi-\omega_{mn}+q\pi, \theta) - \mu_c l_c(r_m, \pi-\omega_{mn}+q\pi, \theta)\}$$

$$\text{mit } n_d = \left[\frac{3(m_s+m)}{\pi} \arcsin\left(\frac{|d|}{r_m}\right) + \frac{1}{2}\right]$$

$$\text{und } m_d \text{ aus Gleichung 5.31} \tag{5.32}$$

$$CR(d,q)_{c,c} = \sum_{m=1}^{m_d} \frac{\pi R^2}{3(m_s+m)} \left\{ \frac{(K-1)^2}{m_c^2}(m-\tfrac{1}{2}) + \frac{K-1}{m_c} \right\}$$

$$\sum_{n=1}^{3(m_s+m)} \exp\{-\mu_c l_c(r_m, \omega_{mn}+q\pi, \theta)\}$$

$$\text{mit } m_d \text{ aus Gleichung 5.31} \tag{5.33}$$

$$CT(d,q)_{c,c} = \sum_{m=m_d+1}^{m_c} \frac{\pi R^2}{3(m_s+m)} \left\{ \frac{(K-1)^2}{m_c^2}(m-\tfrac{1}{2}) + \frac{K-1}{m_c} \right\}$$

$$\sum_{n=1}^{n_d} (\exp\{-\mu_c l_c(r_m, \omega_{mn}+q\pi, \theta) - \mu_c l_c(r_m, \pi-\omega_{mn}+q\pi, \theta)\})$$

$$\text{mit } m_d \text{ und } n_d \text{ aus Gleichungen 5.31 und 5.32} \tag{5.34}$$

5.1.3 Berücksichtigung der Comptonstreuung

Strenggenommen muss bei der Absorptionskorrektur berücksichtigt werden, dass der inkohärent gestreute Anteil eine größere Wellenlänge hat und somit meist stärker absorbiert wird. Der Anteil der Comptonstreuung ist zwar zunächst nicht bekannt, wenn die Absorptionskorrektur durchgeführt wird, er lässt sich jedoch nachträglich für die normierte Intensität berechnen.

Zunächst wird die Comptonstreuung als Funktion des Streuwinkels 2θ berechnet oder ein bestehender Datensatz auf die Winkelskala umgerechnet. Dabei muss jeweils die Änderung $\Delta\lambda$ der Wellenlänge

$$\lambda' = \lambda + \Delta\lambda \text{ mit } \Delta\lambda = \frac{2h}{m_e c} \sin^2\theta \tag{5.35}$$

berücksichtig werden. Für die Umrechnung von κ nach 2θ ergibt sich daraus

$$2\theta = 2 \cdot \arcsin \frac{2\pi - \sqrt{4\pi^2 - \frac{2h}{m_e c}\kappa^2\lambda}}{\frac{2h}{m_e c}\kappa}. \tag{5.36}$$

Nach der Umrechnung werden die Daten linear intrapoliert. Im nächsten Schritt wird die Kendig-Pings-Korrektur für die Comptonstreuung rückgängig gemacht. Es ist zu beachten, dass sich die Wellenlänge der Strahlung während des Streuprozesses ändert. Die Strahlwege müssen daher in eine Strecke vor der Streuung und eine Strecke nach der Streuung unterteilt werden, wie dies in folgender Tabelle dargestellt ist:

	Probe		Messzelle	
	vorher	nachher	vorher	nachher
1	$\nu + r \cdot \cos\omega$	$\eta - r \cdot \cos(\omega - 2\theta)$	$\alpha - \nu$	$\beta - \eta$
2	0	0	$\alpha + r \cdot \cos\omega$	$\beta - r \cdot \cos(\omega - 2\theta)$
3	0	2η	$\alpha + r \cdot \cos\omega$	$\beta - r \cdot \cos(\omega - 2\theta) - 2\eta$
4	2ν	0	$\alpha + r \cdot \cos\omega - 2\nu$	$\beta - r \cdot \cos(\omega - 2\theta)$
5	2ν	2η	$\alpha + r \cdot \cos\omega - 2\nu$	$\beta - r \cdot \cos(\omega - 2\theta) - 2\eta$

Für jeden Winkel muss die Energie nach der Streuung berechnet werden gemäß

$$E' = \frac{hc}{\lambda'} = \frac{E m_e c^2}{m_e c^2 + 2E \sin^2\theta} \tag{5.37}$$

und für diese Energien jeweils die neuen Massenabsorptionskoeffizienten. Die so korrigierte Comptonstreuung entspricht demjenigen Anteil, den die Comptonstreuung an der gemessenen Intensität hat. Diese Streuung wird von der gesamten gemessenen Intensität abgezogen, die mit dem Normierungsfaktor multipliziert wurde, der als konstanter Faktor vor die Absorptionskorrektur gezogen werden kann. Die verbleibende Intensität enthält nur den kohärenten Streuanteil und wird erneut auf Absorption korrigiert. Anschließend werden wieder die kohärenten und inkohärenten Anteile addiert.

Der relative Beitrag der Compton-Korrektur ist umso größer, je größer der Streuwinkel ist und je kleiner die Wellenlänge der verwendeten Strahlung ist, da die Wellenlängenänderung nicht von der Strahlungsenergie abhängt. Gleichzeitig nimmt mit dem Winkel der Anteil der Comptonstreuung an der Gesamtintensität zu. Für das Maximum $2\theta = 180°$ ergibt sich eine Wellenlängenänderung von $\Delta\lambda = 0,0485\mathring{A}$. Die Massenabsorptionskoeffizienten steigen mit der dritten Potenz der Wellenlänge an und gehen exponentiell in die Absorption ein, wodurch sich bei stark absorbierenden Proben der Einfluss der Comptonstreuung in der Absorptionskorrektur bemerkbar macht. Meist ist diese Korrektur jedoch vernachlässigbar, weil entweder bei hohen Energien und bis zu kleinen Winkeln oder bei niedrigen Energien und bis zu großen Winkeln gemessen wird.

5.1.4 Realisierung der Absorptionskorrektur durch Klassen und Methoden

In einen Dialog der Klasse `KendigPingsDialog` können folgende Parameter eingegeben werden: Die Strahlbreite $d = a + b$, die rechtsseitige Verschiebung v der Strahlmitte gegenüber der Kapillarachse, der Innenradius R und der Außenradius KR der Kapillare und die dem obigen Wert m_s entsprechende Rechengenauigkeit. Die oben eingeführten Parameter $a = v + d/2$ und $b = v - d/2$ können daraus berechnet werden. Weiterhin können Werte für den linearen Absorptionskoeffizienten in $1/cm$ von Probe und Messzelle eingegeben

werden. Liegen in der Registratur geeignete Eingaben zu Dichte, Strahlungsenergie und Zusammensetzungen bereits vor, wird dieser Wert automatisch eingetragen. Die Werte für die gefüllte Messzelle werden dem aktuellen Datensatz entnommen, für die Daten der leeren Kapillare kann ein Diagramm ausgewählt werden. Die Absorptionsfaktoren können wahlweise aus einer anzugebenden Datei geladen, neu berechnet und/oder in eine Datei gespeichert werden. Die gespeicherten Absorptionsfaktoren werden gegebenenfalls in der Methode `checkInput()` geladen und auf ihre Kompatibilität zum aktuellen Datensatz überprüft. Auf sie kann anschließend mit der Methode `double[][] KendigPingsDialog::getAbsorptionCoefficients()` zugegriffen werden. Weiter kann angegeben werden, ob in einer verfeinerten Korrektur die Comptonstreuung berücksichtigt werden soll. In diesem Fall müssen die Intensitäten sowohl der leeren als auch der gefüllten Messzelle auf Polarisation korrigiert sein. Benötigte Parameter sind der Normierungsfaktor, die Strahlungsenergie und ein Datensatz mit der Comptonstreuung als Funktion von κ, sofern die Daten nicht unter Verwendung der Registratur berechnet werden sollen. Diese Berechnung ist zu empfehlen, da hier die Intrapolation entfällt.

In der Methode `void XRDEvaluationMenu::kendigPings()` werden die erforderlichen Parameter in ein `ParameterSet` eingetragen und der `Transformer XRDEvaluationMenu$KendigPingsI` aufgerufen, falls die Absorptionsfaktoren neu berechnet werden müssen bzw. der `Transformer XRDEvaluationMenu$KendigPingsII` aufgerufen, falls Absorptionsfaktoren aus einer Datei verwendet werden. Die Methode `transform(...)` in `KendigPingsII` berechnet die Absorptionskorrektur nach Formel 5.1. Mit Hilfe von `KendigPingsIII` kann eine Absorptionskorrektur unter Berücksichtigung der Comptonstreuung nach dem oben beschriebenen Algorithmus durchgeführt werden.

Die Klassen `KendigPingsI` und `KendigPingsIII` leiten sich von der abstrakten Klasse `ExpensiveTransformer` für rechenintensive Operationen (siehe Abschnitt 3.1.7) ab. Deren abstrakte Methode `transform(Data data, ParameterSet ps, Progress prog)` wird implementiert und ruft die Methoden `void calculateCorrection(Data data, ParameterSet ps, Progress prog)` bzw. `void calculateInverseComptonCorrection(Data data, ParameterSet ps, Progress prog)` auf, welche die jeweilige Berechnung über die Konstruktoren der Klasse `KendigPings` und die Berechnung der Koeffizienten in der Methode `void calcKoeffs()` einleiten. In dieser Methode wird zunächst die Fallunterscheidung bezüglich der Bereiche von a und b durchgeführt. Für den gewählten Fall werden die Elementarintegrale für alle Winkel in den Methoden

```
double calcAs(double d)
double calcAc1(double d)
double calcAc2(double d)
double calcSR_ssc(double d, int q, double theta)
double calcST_ssc(double d, int q, double theta)
double calcCR_csc(double d, int q, double theta)
double calcCT_csc(double d, int q, double theta)
double calcCR_cc(double d, int q, double theta)
double calcCT_cc(double d, int q, double theta)
```

berechnet. Diese Methoden rufen weiterhin die Berechnung der Integrandenwerte in den Methoden

```
double calcSR_m(int m)
double calcSOmega_mn(int m, int n)
double calcCR_m(int m)
double calcCOmega_mn(int m, int n)
```

auf. Für jeden dieser Werte `omega` und `rr` können dann die Extinktionen berechnet werden mit `Extinction calcExtinction(double rr, double omega, double theta)`, wo die Fallunterscheidung für die unterschiedlichen Strahlwege getroffen wird. Dabei ist `Extinction` eine innere Klasse von `KendigPings`, welche die gleichzeitig berechneten Werte e_s und e_c für die Extinktionen in Probe und Messzelle aufnimmt. Die in den Extinktionen enthaltenen Weglängen werden wie beschrieben aus Hilfsgrößen zusammengesetzt, welche mit den Methoden

```
double calcAlpha(double rr, double omega)
double calcBeta(double rr, double omega, double theta)
double calcNy(double rr, double omega)
double calcEta(double rr, double omega, double theta)
double calcSigma(double rr, double omega, double theta)
double calcTau(double rr, double omega, double theta)
```

bestimmt werden. Gegebenenfalls werden die Wege vor und nach der Streuung getrennt berechnet. Die Elementarintegrale werden in `void calcKoeffs()` zu den Absorptionskoeffizienten verknüpft. Dort erfolgt schließlich auch die eigentliche Absorptionskorrektur nach Formel 5.1.

5.2 Die Fourier-Transformation

Einen guten Überblick zu diesem Thema gibt Numerical Recipes [69], umfassende Informationen findet man bei Brigham [70]. Viele physikalische Prozesse können in unterschiedlichen Domänen dargestellt werden, die untereinander paarweise verknüpft sind. Im Falle von Beugungsexperimenten sind dies der Impulsraum κ und der Ortsraum r, für spektroskopische Anwendungen die Zeitdomäne t und die Frequenzdomäne f. Im Folgenden wird das in der Literatur häufiger beschriebene Paar Frequenz/Zeit verwendet. Beide Domänen sind über Fouriertransformationen verknüpft:

$$H(f) = \int_{-\infty}^{\infty} h(t)\, e^{2\pi i f t} dt \tag{5.38}$$

$$h(t) = \int_{-\infty}^{\infty} H(f)\, e^{-2\pi i f t} df \tag{5.39}$$

Letztere Transformation wird als inverse Fouriertransformation bezeichnet. Es sei darauf hingewiesen, dass diese Definitionen in der Literatur nicht einheitlich sind. Unterschiede treten bezüglich der Normierungsfaktoren (Verwendung des Transformationspaares t/ω) und der Vorzeichen im Exponenten auf.

5.2.1 Die diskrete Fouriertransformation

Häufig liegen in einer Domäne diskrete Daten vor, z. B. sei die Funktion $h(t)$ angenähert durch die Messwerte

$$h_k = h(t_k) \text{ mit } t_k = k\Delta \text{ und } k = 0, 1, 2, \ldots, N-1$$

Es ist dann $1/\Delta$ die Abtastrate und f_c die Nyquist-Frequenz $f_c = \frac{1}{2\Delta}$, welche derjenigen Frequenz entspricht, in deren Periodendauer mindestens 2 Messwerte aufgenommen wurden. Nach dem Abtast-Theorem von Shannon ist eine Funktion $h(t)$ durch die Werte h_k vollständig bestimmt, wenn f_c die maximale Frequenz ist, d. h. $H(f) = 0$ für alle $|f| \geq f_c$. Man sagt, eine solche Funktion habe eine beschränkte Bandbreite. Existiert keine Bandbreitenbeschränkung, so bringt der Diskretisierungsprozeß zwangsläufig eine als Aliasing bezeichnete Verfälschung der transformierten Funktion mit sich. Diese besteht darin, dass Spektralinformationen, welche höheren Frequenzen entsprechen als denjenigen, die sich nach obiger Formel aus der Abtastrate ergeben, in den Bereich niedrigerer Frequenzen verschoben werden. Liegt hingegen eine Bandbreitenbeschränkung vor und wurde die Abstastrate hinreichend hoch gewählt, so klingt die Fouriertransformierte auf null ab.

Bei der Fouriertransformation bleibt die Anzahl erhaltener Werte gleich der Anzahl der Messwerte, es sind dies die Funktionswerte $H(f)$ an den Frequenzen

$$f_n = \frac{n}{N\Delta} \text{ mit } n = -\frac{N}{2}, -\frac{N-1}{2}, \ldots, \frac{N-1}{2}, \frac{N}{2}$$

wobei N der Einfachheit halber eine gerade Zahl sei und die Werte an den Frequenzen $f_{-N/2} = f_{N/2} = f_c$ identisch sind. Das Integral in (5.38) geht dann über in die Summe

$$H(f_n) = \int_{-\infty}^{\infty} h(t)\, e^{2\pi i f_n t} dt \approx \sum_{k=0}^{N-1} h_k e^{2\pi i f_n t_k} \Delta = \Delta \sum_{k=0}^{N-1} h_k e^{2\pi i k n/N}$$

Man bezeichnet

$$H_n = \sum_{k=0}^{N-1} h_k e^{2\pi i k n/N} \tag{5.40}$$

als diskrete Fouriertransformierte und kann damit schreiben

$$H(f_n) \approx \Delta H_n \tag{5.41}$$

Aufgrund der Periodizität von (5.40) bezüglich n mit Periodenlänge N ist $H_{N-n} = H_{-n}$, wobei n nun im Bereich $0, \ldots, N-1$ gewählt werden kann.
Für die diskrete inverse Fouriertransformierte gilt entsprechend:

$$h_k = \frac{1}{N} \sum_{n=0}^{N-1} H_n e^{-2\pi i k n/N} \tag{5.42}$$

Mit Einführung der komplexen Zahl

$$W = e^{2\pi i/N} \tag{5.43}$$

kann die Fouriertransformation als Matrix-Vektor-Produkt dargestellt werden:

$$H_n = \sum_{k=0}^{N-1} h_k W^{nk} \text{ für alle n} \tag{5.44}$$

Die Matrix enthält dabei als Elemente Potenzen von W, im Vektor sind die Werte h_k der Zeitdomäne untergebracht und als Ergebnis erhält man die Werte H_n in der Frequenzdomäne. Man sieht, dass diese Multiplikation mit einer Ordnung von N^2 Operationen sehr rechenintensiv ist.

5.2.2 Die Schnelle Fouriertransformation

Eine wesentlich schnellere Berechnung ermöglicht die Schnelle Fouriertransformation (Fast Fourier Transform, FFT). Das Danielson-Lanczos-Lemma erlaubt eine Zerlegung der Summe (5.40) mit Länge N in zwei Teilsummen der Länge $N/2$, wobei die erste Summe die Summanden mit geradzahligen (o) Indizes enthält und die zweite Summe die Summanden mit ungeradzahligen (i) Indizes:

$$\begin{aligned}
H_n &= \sum_{k=0}^{N-1} h_k e^{2\pi ikn/N} && (5.45)\\
&= \sum_{k=0}^{N/2-1} h_{2k} e^{2\pi in(2k)/N} + \sum_{k=0}^{N/2-1} h_{2k+1} e^{2\pi in(2k+1)/N} && (5.46)\\
&= \sum_{k=0}^{N/2-1} h_{2k} e^{2\pi ink/(N/2)} + W^n \sum_{k=0}^{N/2-1} h_{2k+1} e^{2\pi ink/(N/2)} && (5.47)\\
&= H_n^o + W^n H_n^i && (5.48)
\end{aligned}$$

Diese Zerlegung lässt sich rekursiv anwenden. Eine weitere Zerlegung führt zu den Summen H_n^{oo} und H_n^{oi} bzw. H_n^{io} und H_n^{ii} usw. Falls die Anzahl der Daten N eine Zweierpotenz ist, was notfalls durch Auffüllen von Feldern mit Nullen erreicht wird, ist eine Zerlegung in Summen mit einem Element möglich. Als elementare Operation erhält man dann die Fouriertransformation eines einzigen Wertes, nämlich die Identitätsoperation. Dies bedeutet, dass jede durch $H_n^{\cdots}$ beschriebene Summe in der letzten Rekursion zu einem Wert h_k führt. Man sieht, dass sich für die Zerlegung in Summen eine Binärkodierung H_n^{binaer} ergibt. Vergleicht man diese Binärkodierung mit der Binärdarstellung der Zahl n, so erkennt man, dass sich die beiden Darstellungen dadurch ineinander umwandeln lassen, dass man sie rückwärts liest.

Abbildung 5.3 zeigt diese Zerlegung für ein Feld mit 8 Elementen. Im Ausgangszustand sind die Indizes der Feldelemente in dezimaler und Binärdarstellung in ihrer natürlichen Reihenfolge gegeben. Die erste Zerlegung nach geraden und ungeraden Indizes entspricht einer Sortierung nach dem letzten Bit und führt zu zwei durch unterschiedliche Schattierungen gekennzeichneten Teilbereichen. Diese werden im zweiten Schritt wieder nach geraden

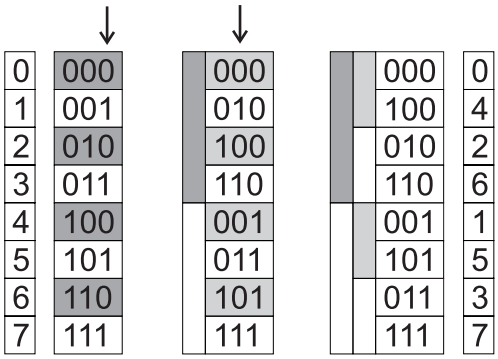


Abbildung 5.3: Bitumkehr bei der FFT

und ungeraden Indizes der Teilbereiche, also jeweils 0 bis 3, weiter aufgeteilt. Bezüglich der Indizes aller Elemente entspricht dies einer Sortierung nach dem vorletzten Bit. Für den Fall von 8 Elementen ist die Zerlegung damit auch schon abgeschlossen. Vergleicht man die Binärdarstellungen von Ausgangs- und Endzustand, so erkennt man die besagte Bitumkehr.

Bei der Berechnung der diskreten Fouriertransformation nach Cooley/Tukey [71] werden die Feldelemente h_k zunächst nach dem Verfahren der Bitumkehr umsortiert, indem jedes Feldelement mit demjenigen vertauscht wird, welches die umgekehrte Bitdarstellung besitzt. Anschließend werden die Elementartransformationen durch paarweise Verknüpfung zu den einzelnen Werten H_n zusammengefügt, wobei sämtliche Berechnungen speichersparend im übergebenen Feld durchgeführt werden können. Die relativ komplexen Verknüpfungen werden anhand von Signalflussdiagrammen in [70] ausführlich erörtert.

Die Verknüpfung der N Werte verläuft über $\log_2 N$ Stufen, damit ergibt sich für den Cooley-Tukey-Algorithmus ein Aufwand der Ordnung $N \log N$, die erheblich unter der Rechenordnung N^2 des Matrix-Vektor-Produktes liegt.

5.2.3 FFT für reelle Messwerte

Wie oben dargestellt wurde, handelt es sich bei der Fouriertransformation um eine Abbildung einer komplexen Funktion auf eine andere komplexe Funktion. Die zu transformierenden Werte werden dabei in Feldern gespeichert, die sowohl den Real- als auch Imaginärteil enthalten. In der Praxis sind die zu transformierenden Daten jedoch häufig reellwertig, man könnte also die reellen Messwerte als komplexe Zahlen speichern, deren Imaginärteil null ist. Dies erhöht jedoch unnötig den rechnerischen Aufwand. Eine bessere Lösung besteht darin, alle reellen Werte in geeigneter Weise als komplexe Zahlen im Feld abzulegen.

Dazu wird die zu transformierende Funktion $h(k)$ bzw. deren diskrete Werte in zwei Teilfunktionen zerlegt gemäß

$$\left.\begin{array}{rcl} h_1(k) & = & h(2k) \\ h_2(k) & = & h(2k+1) \end{array}\right\} k = 0, 1, \ldots, N-1.$$

Die diskrete Fouriertransformierte[1] lautet dann:

$$\begin{aligned} H(n) &= \sum_{k=0}^{2N-1} h(k)\, e^{\pm 2\pi i k n/2N} \\ &= \sum_{k=0}^{N-1} h(2k)\, e^{\pm 2\pi i \{2k\} n/2N} + \sum_{k=0}^{N-1} h(2k+1)\, e^{\pm 2\pi i \{2k+1\} n/2N} \\ &= \sum_{k=0}^{N-1} h(2k)\, e^{\pm 2\pi i k n/N} + e^{\pm \pi i n/N} \sum_{k=0}^{N-1} h(2k+1)\, e^{\pm 2\pi i k n/N} \\ &= \sum_{k=0}^{N-1} h_1(k)\, e^{\pm 2\pi i k n/N} + e^{\pm \pi i n/N} \sum_{k=0}^{N-1} h_2(k)\, e^{\pm 2\pi i k n/N} \\ &= H_1(n) + e^{\pm \pi i n/N} H_2(n) \end{aligned} \tag{5.49}$$

mit den (inversen) Fouriertransformierten $H_1(n)$ und $H_2(n)$ von $h_1(k)$ und $h_2(k)$. Um mit dem Algorithmus für die Schnelle Fouriertransformation arbeiten zu können, erzeugt man formal eine komplexwertige Funktion

$$f(k) = h_1(k) + i h_2(k)$$

Deren Fouriertransformierte

$$F(n) = H_1(n) + i H_2(n) \text{ mit } n = 0, 1, \ldots, N-1$$

setzt wiederum aus den komplexen Fouriertransformierten H_1 und H_2 zusammen. Andererseits kann man auch schreiben

$$F(n) = F_r(n) + i F_i(n)$$

Hierin sind sowohl der Realteil F_r als auch der Imaginärteil F_i von F reelle Funktionen.
Mit $F(n) = F_g(n) + F_u(n)$ lässt sich jede Funktion $F(n)$ in eine gerade $F_g(n)$ und eine ungerade $F_u(n)$ Funktion zerlegen gemäß $F_g(n) = \frac{1}{2}[F(n) + F(-n)]$ und $F_u(n) = \frac{1}{2}[F(n) - F(-n)]$.

Demnach gilt:

$$F(n) = \underbrace{\frac{F_r(n) + F_r(-n)}{2}}_{F_{rg}} + \underbrace{\frac{F_r(n) - F_r(-n)}{2}}_{F_{ru}} + i\underbrace{\frac{F_i(n) + F_i(-n)}{2}}_{iF_{ig}} + i\underbrace{\frac{F_i(n) - F_i(-n)}{2}}_{iF_{iu}} \tag{5.50}$$

Gleichzeitig kann man unter Berücksichtigung der Symmetrieeigenschaften der trigonometrischen Funktionen schreiben:

$$\begin{aligned} F(n) &= \sum_{k=0}^{N-1} [h_1(k) + i h_2(k)]\, e^{\pm 2\pi i k n/N} \\ &= \underbrace{\sum_{k=0}^{N-1} h_1(k) \cos\left(\frac{2\pi k n}{N}\right)}_{F_{rg}} \underbrace{\mp \sum_{k=0}^{N-1} h_2(k) \sin\left(\frac{2\pi k n}{N}\right)}_{F_{ru}} \\ &\quad \underbrace{\pm i \sum_{k=0}^{N-1} h_1(k) \sin\left(\frac{2\pi k n}{N}\right)}_{iF_{iu}} \underbrace{+ i \sum_{k=0}^{N-1} h_2(k) \cos\left(\frac{2\pi k n}{N}\right)}_{iF_{ig}} \end{aligned}$$

[1] das untere Vorzeichen bezieht sich ggf. im Folgenden auf die inverse Fouriertransformierte

oder kurz

$$\begin{array}{rrcl} & H_1(n) & = & F_{rg}(n) + iF_{iu}(n) \\ & iH_2(n) & = & F_{ru}(n) + iF_{ig}(n) \\ \Leftrightarrow & H_2(n) & = & F_{ig}(n) - iF_{ru}(n) \end{array} \tag{5.51}$$

Für $H(n)$ erhält man mit 5.49:

$$\begin{array}{rcl} H(n) & = & F_{rg}(n) + iF_{iu}(n) + e^{\pm\pi in/N}\left[F_{ig}(n) - iF_{ru}(n)\right] \\ & = & F_{rg}(n) + \cos\left(\frac{\pi n}{N}\right) F_{ig}(n) \pm \sin\left(\frac{\pi n}{N}\right) F_{ru}(n) \\ & + & i\left[F_{iu}(n) \pm \sin\left(\frac{\pi n}{N}\right) F_{ig}(n) - \cos\left(\frac{\pi n}{N}\right) F_{ru}(n)\right] \\ & = & H_r(n) + iH_i(n) \end{array} \tag{5.52}$$

Einsetzen von 5.50 ergibt:

$$\begin{array}{rcl} H_r(n) & = & \frac{F_r(n)+F_r(-n)}{2} + \frac{F_i(n)+F_i(-n)}{2}\cos\left(\frac{\pi n}{N}\right) \pm \frac{F_r(n)-F_r(-n)}{2}\sin\left(\frac{\pi n}{N}\right) \\ H_i(n) & = & \frac{F_i(n)-F_i(-n)}{2} \pm \frac{F_i(n)+F_i(-n)}{2}\sin\left(\frac{\pi n}{N}\right) - \frac{F_r(n)-F_r(-n)}{2}\cos\left(\frac{\pi n}{N}\right) \end{array} \tag{5.53}$$

Demnach lässt sich die gesuchte (inverse) Fouriertransformierte $H(n)$ der ursprünglichen Funktion $h(k)$ aus dem Real- und Imaginärteil der (inversen) Fouriertransformierten $F(n)$ der neu gebildeten Funktion $f(k)$ rekonstruieren.

Für die numerische Auswertung benötigt man weiterhin die sich aus der Periodizität ergebenden Beziehungen $F(-n) = F(N-n)$ und insbesondere $F(N) = F(0)$.

5.2.4 Die Schnelle Diskrete Sinustransformation DST

Die Fouriertransformierte des ungeraden Anteils einer Funktion $f(k)$ wird als Sinusfouriertransformierte oder Sinustransformierte dieser Funktion bezeichnet. In ihrer diskreten Schreibweise sei sie definiert als

$$F_n = \sum_{k=1}^{N-1} f_k \sin(\pi kn/N) \tag{5.54}$$

Cooley, Lewis und Welch [72] schlagen einen Algorithmus vor, welcher die Sinustransformation auf die FFT zurückführt. Die im Folgenden beschriebene Vorgehensweise von Temperton [73] verwendet den Algorithmus der bis auf einen Faktor zu sich selbst inversen Sinustransformation in umgekehrter Reihenfolge, um Rundungsfehler zu minimieren.

Zur Berechnung wird zunächst eine Substitution durchgeführt mit

$$h_k = \sin(k\pi/N)(f_k + f_{N-k}) + \frac{1}{2}(f_k - f_{N-k}) \tag{5.55}$$

$$\text{mit } k = 1, 2, \ldots, N-1 \text{ und } h_0 = h_N = 0$$

Diese Werte bilden ein Feld reeller Elemente, die mit dem im vorigen Abschnitt beschriebenen Algorithmus transformiert werden können. Die Fouriertransformierte lautet

$$H(n) = \sum_{k=0}^{N-1} h_k e^{2\pi ikn/N} = \sum_{k=0}^{N-1} h_k \cos(2\pi kn/N) + i\sum_{k=0}^{N-1} h_k \sin(2\pi kn/N)$$

Für den Realteil ergibt sich nach der Substitution:

$$H_n^{\Re} = \sum_{k=0}^{N-1} \sin(k\pi/N)\,(f_k + f_{N-k}) \cos(2\pi kn/N) \tag{5.56}$$

$$+ \frac{1}{2} \sum_{k=0}^{N-1} (f_k - f_{N-k}) \cos(2\pi kn/N) \tag{5.57}$$

Zerlegt man die zweite Summe voriger Gleichung in zwei Teilsummen

$$\sum_{k=0}^{N-1} f_k \cos(2\pi kn/N) - \sum_{k=0}^{N-1} f_{N-k} \cos(2\pi kn/N)$$

und substituiert in der zweiten Teilsumme mit $l = N - k$

$$\sum_{k=0}^{N-1} f_k \cos(2\pi kn/N) - \sum_{l=1}^{N} f_l \cos(2\pi\,(N-l)\,n/N)$$

so ergibt sich nach Anwendung des Additionstheorems $\cos(\alpha - \beta) = \cos\alpha\cos\beta + \sin\alpha\sin\beta$:

$$\sum_{k=1}^{N} f_k \cos(2\pi kn/N) - \sum_{l=1}^{N} f_l \cos(2\pi ln/N) = 0\ ,$$

d. h. die Teilsumme (5.57) verschwindet.

Für den Realteil folgt damit

$$\begin{aligned} H_n^{\Re} &= \sum_{k=0}^{N-1} \sin(k\pi/N)\,(f_k + f_{N-k}) \cos(2\pi kn/N) \\ &= \sum_{k=0}^{N-1} f_k \sin(k\pi/N) \cos(2\pi kn/N) \\ &+ \sum_{k=0}^{N-1} f_{N-k} \sin(k\pi/N) \cos(2\pi kn/N) \end{aligned}$$

Substitutiert man in der zweiten Summe voriger Formel mit $l = N - k$, so ergibt sich für diese

$$\sum_{l=1}^{N} f_l \sin\frac{(N-l)\,\pi}{N} \cos\frac{2\pi\,(N-l)\,n}{N}$$

Durch Anwendung der Additionstheoreme $\cos(\alpha - \beta) = \cos\alpha\cos\beta + \sin\alpha\sin\beta$ und $\sin(\alpha - \beta) = \sin\alpha\cos\beta - \cos\alpha\sin\beta$ erhält man weiter

$$\sum_{l=1}^{N} f_l \sin(l\pi/N) \cos(2\pi ln/N)$$

Der Realteil wird damit

$$H_n^{\Re} = \sum_{k=0}^{N-1} 2 f_k \sin(k\pi/N) \cos(2\pi kn/N)$$

Mit dem Additionstheorem $2\cos\alpha\sin\beta = \sin(\alpha+\beta) - \sin(\alpha-\beta)$ folgen mit

$$\begin{aligned} H_n^{\Re} &= \sum_{k=0}^{N-1} f_k \left(\sin\frac{(2n+1)\,k\pi}{N} - \sin\frac{(2n-1)\,k\pi}{N} \right) \\ &= F_{2n+1} - F_{2n-1} \end{aligned}$$

schließlich die Elemente der Sinustransformierten F_n.

In ähnlicher Weise berechnet man für den Imaginärteil

$$\begin{aligned} H_n^{\Im} &= \sum_{k=0}^{N-1} \sin(k\pi/N)\,(f_k + f_{N-k}) \sin(2\pi kn/N) \\ &+ \sum_{k=0}^{N-1} \frac{1}{2}\,(f_k - f_{N-k}) \sin(2\pi kn/N) \end{aligned}$$

Für den Imaginärteil entfällt die erste Teilsumme, und man erhält mit

$$\begin{aligned} H_n^{\Im} &= \sum_{k=0}^{N-1} \frac{1}{2}\,(f_k - f_{N-k}) \sin(2\pi kn/N) \\ &= \sum_{k=0}^{N-1} f_k \sin(2\pi kn/N) \\ &= F_2 n \end{aligned}$$

weitere Elemente der Sinustransformierten.

Damit ergibt sich für die gesamte Sinustransformierte

$$F_{2n} = H_n^{\Im} \text{ und } F_{2n+1} = F_{2n-1} + H_n^{\Re} \text{ mit } n = 0, 1, \ldots, N/2-1 \tag{5.58}$$

Die ungeradzahligen Feldelemente werden also über eine Rekursion berechnet, deren erstes Element sich über die Symmetriebeziehung $F_1 = -F_{-1}$ zu $F_1 = \frac{1}{2}R_0$ ergibt.

Der Algorithmus umfasst insgesamt drei Schritte:

1. Substitution des zu transformierenden Feldes nach Gleichung 5.55
2. Fouriertransformation des sich ergebenden reellen Feldes
3. Berechnung der Sinustransformierten nach Gleichung 5.58

wobei man für die Substitution im ersten Schritt unter Beachtung von

$$\begin{aligned} h_k &= \sin(k\pi/N)\,(f_k + f_{N-k}) + \frac{1}{2}\,(f_k - f_{N-k}) \\ h_{N-k} &= \sin(k\pi/N)\,(f_k + f_{N-k}) - \frac{1}{2}\,(f_k - f_{N-k}) \end{aligned}$$

die Zahl der Rechenschritte halbiert.

5.2.5 Testfunktionen für die Transformationen

Zur Überprüfung der Transformationen wurden numerische Ergebnisse mit analytisch berechneten Funktionen verglichen. Als zu transformierende Funktion wurde eine Rechteckfunktion gewählt mit

$$h(t) = \begin{cases} A & \text{für } t \leq B \\ 0 & \text{für } t > B \end{cases} \tag{5.59}$$

Diese Funktion ist leicht darstellbar und besitzt charakteristische Transformierte, bei denen Abweichungen deutlich erkennbar sind. Bei einer numerischen Darstellung dieser Funktion muss die Dirichlet-Bedingung beachtet werden, nach der an Sprungstellen t_S der Mittelwert der Grenzwerte beider Seiten einzusetzen ist:

$$h(t_S) = \frac{1}{2}\left(\lim_{t^+ \to t_S} h(t) + \lim_{t^- \to t_S} h(t)\right) \tag{5.60}$$

Hier handelt es sich um die beiden Sprungstellen bei 0 und bei B. Die Dirichlet-Bedingung ist übrigens bei allen Transformationen zu berücksichtigen, bei denen eine Funktion von einem endlichen Wert bei $h(0)$ auf $h(N-1) = 0$ abklingt. In den nachfolgenden Beispielen wurde die Funktion durch ein Feld mit 2048 Datenpaaren dargestellt mit $A = 2$, $B = 0.6$ und den Sprungstellen $h(0) = h(.6) = 1.$. Für die Fouriertransformierte ergibt sich

$$\begin{aligned} H(f) &= \int_{-\infty}^{+\infty} h(t)\, e^{2\pi i f t}\, dt = \int_{0}^{B} A \cdot e^{2\pi i f t}\, dt \\ &= \frac{A}{2\pi f}\left(1 - \cos(2\pi f B)\right) i + \frac{A}{2\pi f}\sin(2\pi f B) \end{aligned}$$

Für die inverse Fouriertransformierte der Rechteckfunktion erhält man

$$\begin{aligned} h(t) &= \int_{-\infty}^{+\infty} H(f)\, e^{-2\pi i f t}\, df = \int_{0}^{B} A \cdot e^{-2\pi i f t}\, df \\ &= \frac{A}{2\pi t}\left(\cos(2\pi t B) - 1\right) i + \frac{A}{2\pi f}\sin(2\pi f B) \end{aligned}$$

Abbildung 5.4 zeigt die Realteile der Fouriertransformierten der Rechteckfunktion 5.59 in analytischer Form und als Ergebnis der Algoritmen DFT und FFT. Im vorderen Bereich ist die Übereinstimmung sehr gut, im hinteren Bereich klingen beide numerisch transformierten Funktionen gegenüber der analytischen Funktion zu stark ab. Die Ursache liegt in den erniedrigten Werten der beiden Sprungstellen. Werden diese jedoch nicht getrennt berücksichtigt, so ergibt sich eine deutlich verschobene Funktion. Die Abweichung wird hingegen bei größeren Abtastraten zunehmend geringer. Dieses Problem besteht bei der diskreten Transformation prinzipiell und ist auch durch Verwendung anderer Algorithmen nicht zu lösen. Die Realteile der inversen Fouriertransformierten sind mit denjenigen der normalen Fouriertransformation identisch und sind daher nicht dargestellt.

Die Sinustransformierte der Rechteckfunktion lautet

$$H(f) = \int_{0}^{+\infty} h(t)\, \sin(\pi f t)\, dt = \int_{0}^{B} A \cdot \sin(\pi f t)\, dt$$

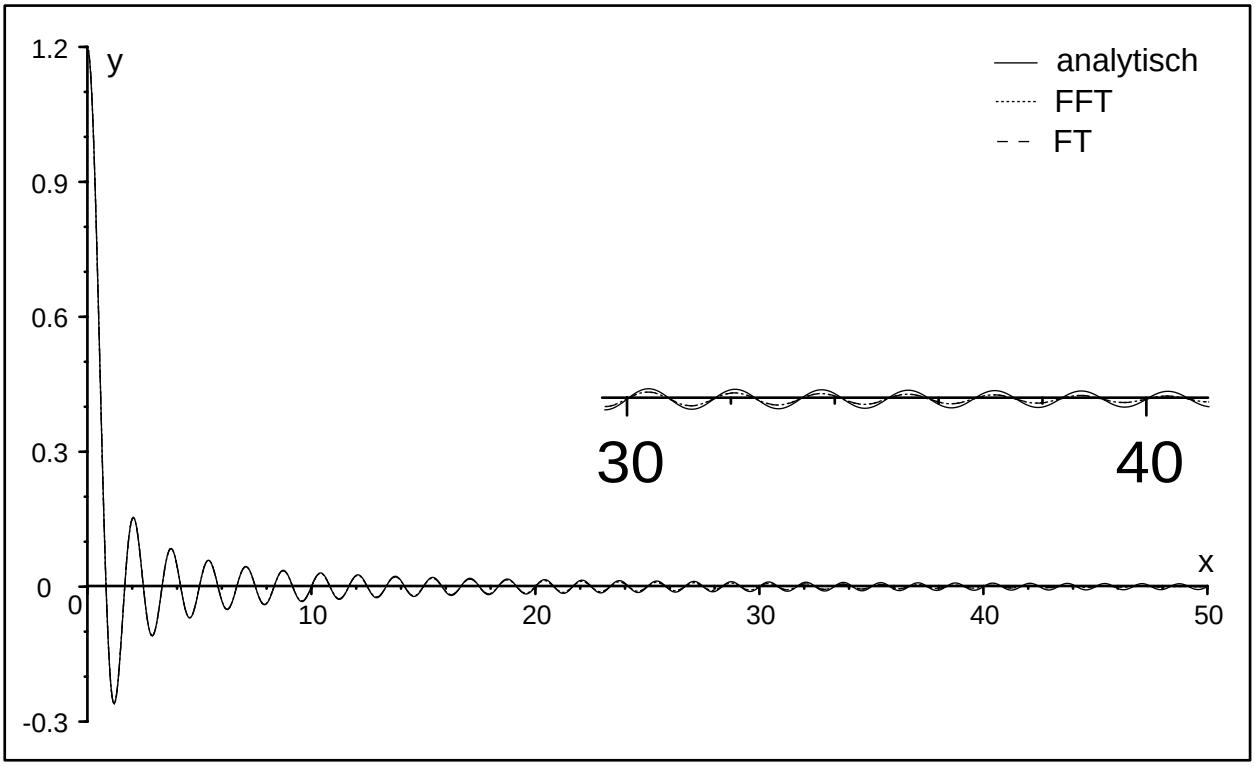


Abbildung 5.4: Test der Fouriertransformation

$$= \frac{A}{\pi f}\left(1 - \cos\left(\pi f B\right)\right)$$

Abbildung 5.5 zeigt die Sinustransformierte der Rechteckfunktion in analytischer Form und nach numerischer Transformation. Auch hier sind die oben beschriebenen Abweichungen zu beobachten.

5.2.6 Klassen und Methoden zur Berechnung der Fouriertransformation

Durch Anwahl des Menüeintrags *Fourier Transform...* im `DiagramCalculationMenu` können die reellen Daten des aktuellen Datensatzes transformiert werden. Die Voraussetzungen dafür werden in der Methode `boolean readyForFourier()` überprüft: Alle Abszissenwerte müssen aufsteigend angeordnet, äquidistant und positiv sein. Beginnen sie nicht mit Null, so wird der Datensatz später mit Nullen im Ordinatenbereich aufgefüllt. Außerdem muss der erste Abszissenwert ein Vielfaches der Schrittweite sein, andernfalls ist eine vorhergehende Intrapolation erforderlich. Sind diese Voraussetzungen gegeben, so wird in der Methode `fourier()` ein `FourierDialog` geöffnet, welcher eine Auswahl zwischen verschiedenen Algorithmen (gewöhnliche Fouriertransformation, FFT und Sinustransformation), dem zurückgebenden Anteil (real/imaginär) und dem Modus (normal/invers) bietet. Die gewählten Optionen werden in ein `ParameterSet` eingetragen, die jeweiligen Transformationen erzeugt und ausgeführt. Die dazu verwendeten `Transformer` sind: `DiagramCalculationMe-`

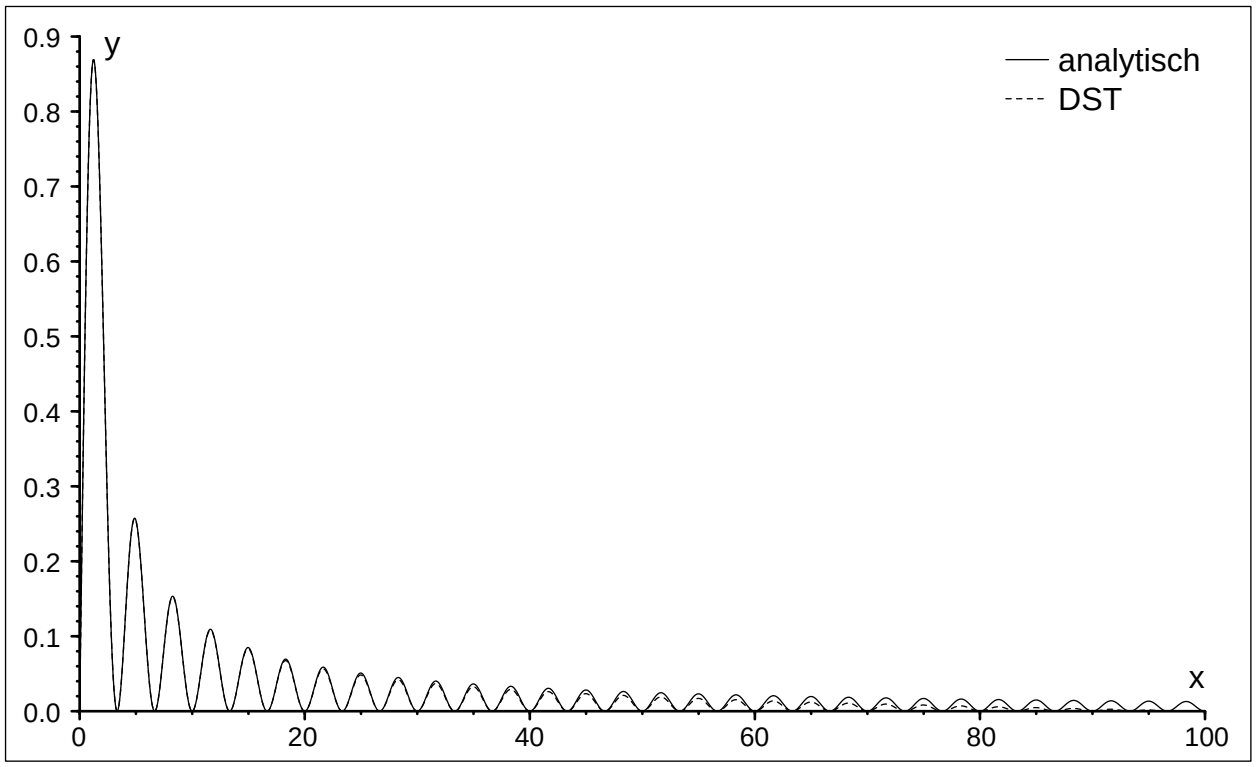


Abbildung 5.5: Test der Sinustransformation

`nu$FourierTransformer`, `DiagramCalculationMenu$FastFourierTransformer` und `DiagramCalculationMenu$SineTransformer`.

Methoden zur gewöhnlichen Fouriertransformation

Da die gewöhnliche Fouriertransformation für die Transformation längerer Datensätze sehr rechenintensiv ist, wurde der `FourierTransformer` als Subklasse von `ExpensiveTransformer` implementiert. Die Berechnung der Transformation erfolgt dort unter Aufruf der Methode `Fourier.arrangeRealFT(...)`, in welcher die Arrays des Datensatzes in `DoubleArrays` übergeben werden. In `arrangeRealFT(...)` wird das übergebene Feld reeller Werte mit `Complex.convertRealToComplex(...)` in ein Feld komplexer Zahlen umgewandelt, deren Imaginärteil null ist. Damit wird die Methode `Fourier.arrangeComplexFT(...)` aufgerufen. Hier werden zunächst mit `Fourier.prepareStartSequence(...)` die Felder ggf. mit äquidistanten Stützstellen aufgefüllt, falls die Abszissenwerte nicht mit 0 beginnen. Die eigentliche Transformation erfolgt schließlich nach Gleichung 5.40 in `Fourier.ft(...)`. Daraufhin werden noch die Abszissen- und Ordinatenwerte in `Fourier.arrangeComplexFT(...)` skaliert und in `FourierTransformer` der Real- oder Imaginärteil entsprechend der gewünschten Eingabe verwendet. Die inverse Transformation erfolgt entsprechend.

Methoden zur Schnellen Fouriertransformation

Für die Schnelle Fouriertransformation wird die Klasse `FastFourierTransformer` verwendet. In der Methode `Fourier.arrangeRealFFT(...)` werden nicht nur ggf. führende Stütz-

stellen eingefügt, sondern auch abschließende Stützstellen mit dem Funktionswert 0, da der oben beschriebene Algorithmus Feldlängen benötigt, welche Potenzen von zwei sind. Eine geeignete Feldlänge wird mit `getSuitedSize(int i)` berechnet. Mit `fftReal(...)` wird die Transformation reeller Werte auf die Schnelle Fouriertransformation zurückgeführt. Die Schnelle Fouriertransformation komplexer Werte schließlich findet in `fft(Complex[] z, boolean inverse)` statt, worin weiterhin die Methoden `reverse(int k, int t)` zur Bitumkehr einer Zahl und `reverse(Complex[] z)` zur Umsortierung eines Feldes komplexer Zahlen entsprechend der Bitumkehr ihrer Indices aufgerufen werden. Letztere Methoden wurden unter Anlehnung an die bei van Loan [74] formulierten Algorithmen kodiert. Zum Rechnen mit komplexen Zahlen wird die Klasse `Complex` verwendet. Die weitere Bearbeitung der Felder verläuft analog zur gewöhnlichen Fouriertransformation.

Methoden zur Sinustransformation

Die Sinustransformation wird in der Klasse `SineTransformer` und der Methode `Fourier.arrangeDST(...)` vorbereitet. Auch hier wird das Feld mit führenden und folgenden Stützstellen ergänzt. In `Fourier.dst(...)` wird die Sinustransformation entsprechend dem oben beschriebenen Algorithmus auf die Schnelle Fouriertransformation reeller Werte zurückgeführt, deren Methoden im vorigen Abschnitt aufgeführt wurden.

5.3 Das Paket `function`

Zur Auswertung von Zeichenketten, die Funktionsterme wie `sin(x^2)+x*5+cos(-x/3)` enthalten, wird ebenso wie für die in Abschnitt 3.3.3 beschriebenen chemischen Formeln ein Parser benötigt. Dieser Parser befindet sich in der Klasse `Function`. Während für chemische Formeln nur die Zusammensetzung als weiter verarbeitbare Information von Bedeutung ist, welche in einer `Map` abgelegt werden kann, müssen mathematische Formeln so abgelegt werden, dass sie als Operationen auf Variablen und Parameter angewandt werden können. Dies geschieht mit Hilfe der Klasse `FunctionNode`. Die Variablen und Parameter befinden sich in der Klasse `FunctionVariables`.
Abbildung 5.6 zeigt einen Überblick über die bei der Arbeit mit Funktionstermen auftretenden Klassen. Die enthaltenen Methoden werden in den folgenden Abschnitten beschrieben. Neben den eingezeichneten Klassenbeziehungen bestehen zwischen den speziellen Klassen, die hier zu Gruppen zusammengefasst wurden, zahlreiche weitere Beziehungen.

5.3.1 Die Syntax von Funktionstermen

Die Grammatik mathematischer Ausdrücke lautet in der EBNF-Notation:

```
Ausdruck:       ["+" | "-"] Summand {("+" | "-") Summand}.
Summand:        Faktor {("*" | "/") Faktor}.
Faktor:         Element {"^" Element}.
Element:        "(" Ausdruck ")" | Variable | Funktion "(" Ausdruck ")"
                | Konstante.
Variable:       "x" [Integer] | "y" | "z" | "t".
Konstante:      "pi" | "e" | "k" [Integer] | Fließkommaliteral.
Funktion:       "sin" | "cos" | "tan" | "asin" | "acos" | "atan"
```

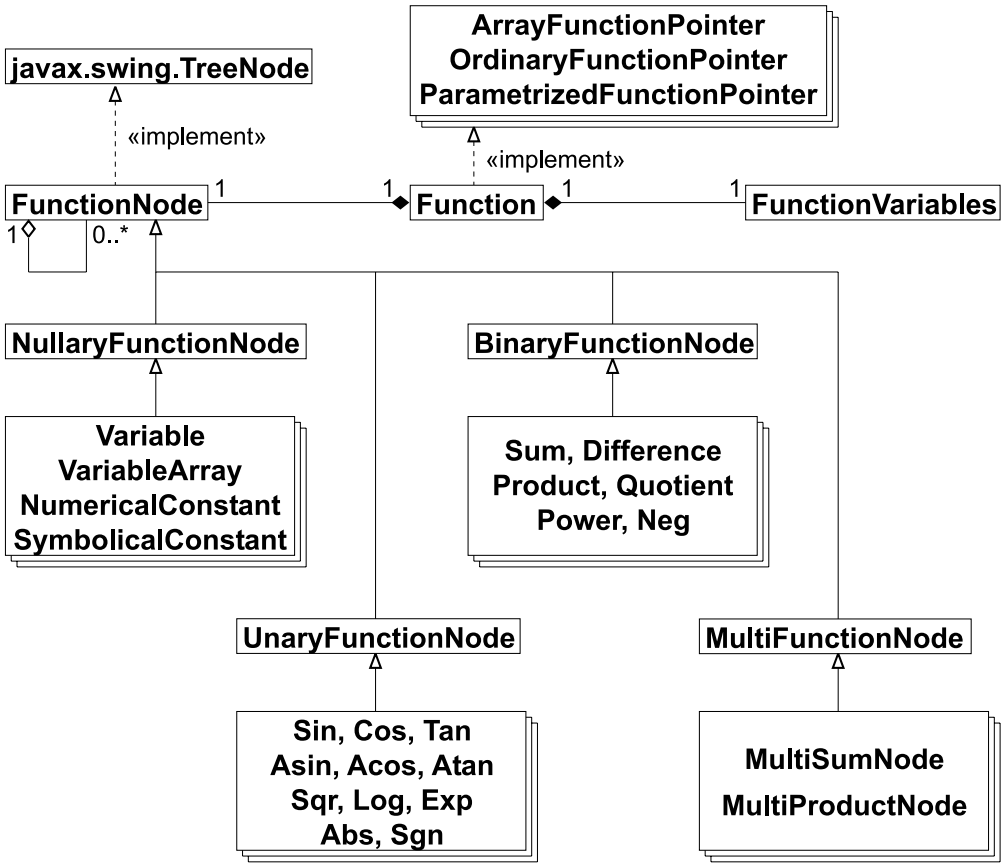


Abbildung 5.6: Klassen zur Arbeit mit Funktionstermen

```
| "sqr" | "sqrt" | "exp" | "log" | "abs" | "sgn".
```

Die Grammatiken für `Integer` und `Fließkommaliteral` wurden bereits im Abschnitt 3.3.1 angegeben. Es sind `sqr()` und `sqrt()` die Quadratwurzelfunktion, `abs()` die Betragsfunktion und `sgn()` die Vorzeichenfunktion mit

$$\mathrm{sgn}(x) = \begin{cases} -1 & \text{für} \quad x < 0 \\ 0 & \text{für} \quad x = 0 \\ +1 & \text{für} \quad x > 0 \end{cases}$$

Der Potenzierungsoperator ∧ wird wie alle anderen binären Operatoren linksassoziativ verwendet[2], d. h. `e∧x∧2` wird interpretiert als `(e∧x)∧2` und nicht als `e∧(x∧2)`. Für x-Variablen und Konstanten k sind ganzzahlige Indices erlaubt, soweit die Klasse `Function` entsprechend konfiguriert wurde (s. u.).

5.3.2 Die Klasse `FunctionNode` und ihre Subklassen

Ein Funktionsterm wie `sin(x∧2)+x*5+cos(-x/3)` lässt sich als arithmetischer Baum darstellen, wie er in Abbildung 5.7 gezeigt ist.

[2]im Gegensatz z. B. zu Fortran

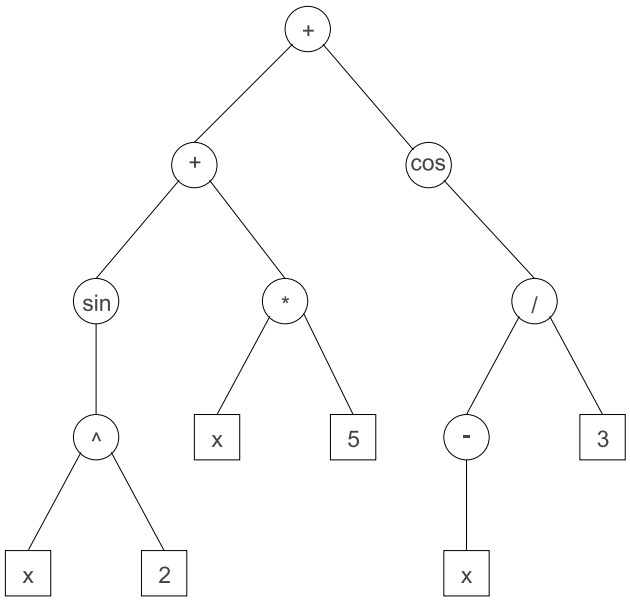


Abbildung 5.7: Beispiel eines arithmetischen Baums

Man erkennt, dass sich dieser Ausdruck unter Beachtung der Prioritätsregeln in Unterausdrücke zerlegen lässt. Er lautet in linearisierter Schreibweise:

$$((sin(x \wedge 2)) + (x * 5)) + (cos((-x)/3)).$$

Ein Baum besteht aus Knoten (beschriftete Elemente) und Kanten. Man unterscheidet verschiedene Typen von Knoten:

- binäre Knoten mit zwei Nachfolgern, das sind die Operatoren $+$, $*$, $-$, $/$ und $\wedge$
- unäre Knoten mit nur einem Nachfolger, das sind mathematische Funktionen wie *sin* und *cos* sowie der Negierungsoperator $-$
- Knoten ohne Nachfolger, die Blätter. Hier sind weiterhin zu unterscheiden
  - numerische Konstanten (beliebige Dezimalzahlen)
  - symbolische Konstanten (im Programm sind es k1, k2, ...)
  - Variablen (t, x, y, z, x1, x2, x3, ...)

Diese Aufteilung in Knotentypen spiegelt sich in den Klassen `BinaryFunctionNode`, `UnaryFunctionNode` und `NullaryFunctionNode` in Abbildung 5.6 wieder. Erkennbar ist auch die für Baumstrukturen typische rekursive Verwendung der Klasse `FunctionNode`, die das Interface `javax.swing.TreeNode` implementiert.

Die abstrakte Basisklasse `FunctionNode` verwaltet Referenzen auf den Vorgängerknoten `parent` und die Nachfahren `leftChild` und `rightChild`. Ihre abstrakte Methode `double evaluate()` wertet den in diesem Knoten aufgehängten Unterbaum durch Traversierung aus.

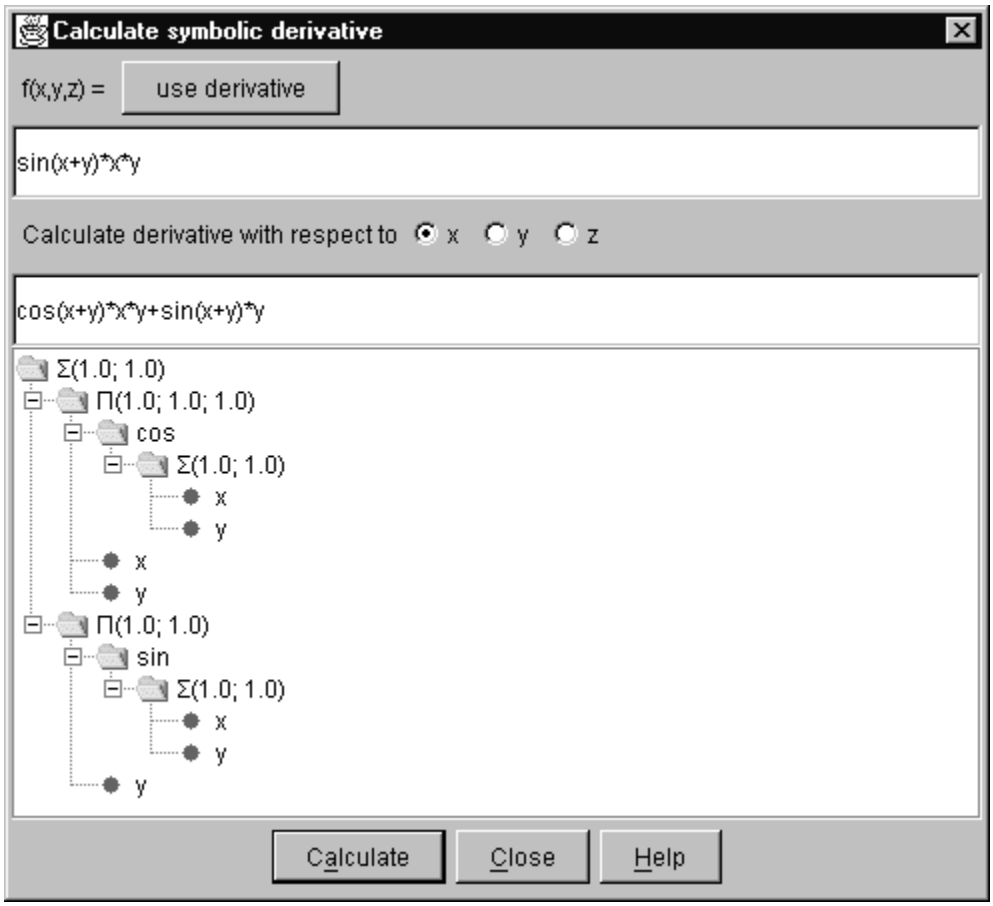


Abbildung 5.8: Dialog zur Berechung symbolischer Ableitungen

Beispielsweise sieht eine Implementierung dieser Methode in der Subklasse `Produkt` so aus, dass als Wert das Produkt `leftChild.evaluate() * rightChild.evaluate()` zurückgegeben wird. Die abstrakte Methode `FunctionNode getDerivative(String var)` definiert die Berechnung der Ableitung nach der Variablen `var`. Tritt beispielsweise in der Traversierung ein Objekt der Klasse Produkt auf, so wird die Ableitung nach `var = x` nach der Produktregel berechnet:

```
new Sum(new Product(leftChild.getDerivative("x"), rightChild.deepCopy()),
  new Product(leftChild.deepCopy(), rightChild.getDerivative("x")));
```

Darin erzeugt die in dieser Klasse definierte Methode `deepCopy()` eine tiefe Kopie des jeweiligen Knotens, indem diese Methode rekursiv für alle Nachfahren aufgerufen wird. Abbildung 5.8 zeigt einen Dialog zur Berechnung symbolischer Ableitungen und stellt die angezeigte Ableitung als arithmetischen Baum dar.

Um eine Funktion wieder in einer für den Anwender lesbaren Form ausgeben zu können, müssen die Knoten in Zeichenketten übersetzt werden. Für den Knoten an sich gibt die Methode `String getString()` eine Zeichenkette zurück z. B. „*“ für die Klasse `Produkt`. Die Methode `toString()` gibt das gesamte Produkt mit den Faktoren zurück, deren Zeichenket-

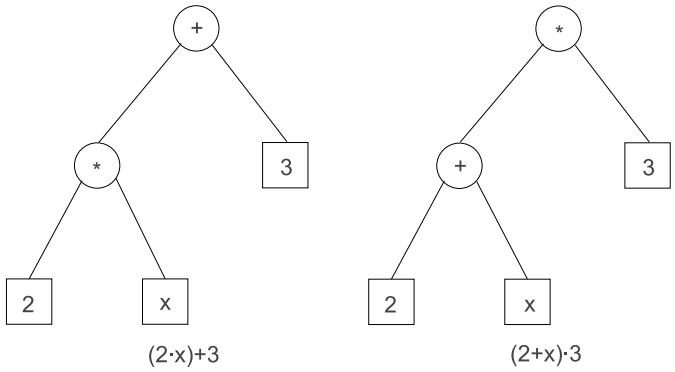


Abbildung 5.9: Linearisierung von arithmetischen Bäumen

ten durch die entsprechende Methode der Nachfahren erzeugt wurde. Bei der Auswertung mathematischer Terme sind Prioritätsregeln zu beachten. Diese sind in einer Baumstruktur und den verwendeten Methoden implizit enthalten, da immer erst die Nachfahren eines Knotens ausgewertet werden. In Abbildung 5.9 ist dargestellt, wie zwei Bäume wieder in die lineare Schreibweise überführt werden, indem die Ausdrücke der Nachfahren in Klammern geschrieben werden. Obwohl die Bäume jedoch strukturell identisch sind, werden nur für den rechten Baum Klammern benötigt, da die Prioritäten der Operatoren nicht mit denjenigen des Baumes übereinstimmt. Die dafür erforderlichen Prioritäten der Knotenklassen sind über die Methode `int getPriority()` zugänglich. Aber auch nach Beachtung der Prioriotätsregeln sind weiterhin zahlreiche Einzelfallentscheidungen erforderlich. Beispielsweise hat der Negierungsoperator eine höhere Priorität als der Additionsoperator, es muss also in `-(1-x)` eine Klammer gesetzt werden. Umgekehrt ist jedoch ein Ausdruck wie `x+-1` nicht zulässig.

Bei formaler Anwendung von Ableitungsregeln ergeben sich häufig Terme wie `1*x` oder `0+x`. Die Methode `simplify()` erlaubt die Vereinfachung solcher Ausdrücke. Mit `simplifyAll()` wird `simplify()` rekursiv zunächst für die Nachfahren aufgerufen. Dies ist sehr viel komplizierter, als es auf den ersten Blick erscheint. Arbeitet man mit binären Knoten, so wird ein Term wie `x+x^2+2*x` intern z. B. entsprechend `x+((x^2)+(2*x))` dargestellt. Um eine solche Baumstruktur zu vereinfachen, müssen zunächst Terme gleicher Ordnung zusammengefasst werden und in einem Knoten mit einer beliebigen Anzahl an Nachfahren angeordnet werden. Dazu dienen die abstrakte Klasse `MultiFunctionNode` und ihre Subklassen `MultiSumNode` und `MultiProductNode`. Diese Klassen leiten sich ebenfalls von `FunctionNode` ab und implementieren die genannten Methoden zur Auswertung, Ableitung, Zeichenkettengenerierung, Vereinfachung und Zusammenfassung von Ausdrücken. Dieser Ansatz löst eine Vielzahl von Problemen. Für komplexere Ausdrücke werden jedoch sehr viel spezialisiertere Algorithmen benötigt, die hier nicht zur Anwendung kommen.

Ebenso implementieren die übrigen in Abbildung 5.6 aufgeführten Klassen die abstrakten Methoden von `FunctionNode` und deren abstrakten Subklassen `NullaryFunctionNode`, `UnaryFunctionNode` und `BinaryFunctionNode`. Eine detaillierte Dokumentation kann an dieser Stelle aus Platzgründen nicht erfolgen.

5.3.3 Die Klassen `Function` und `FunctionVariables`

Diese Klasse erlaubt die Arbeit mit vom Benutzer in Form von Zeichenketten eingegebenen Funktionen ähnlich wie mit numerischen Methoden. Über den Konstruktor `FunctionNode(String term)` lassen sich Funktionen gemäß `Function fun = new Function("x^2")` erzeugen. Durch Verwendung der im Interface `OrdinaryFunctionPointer` definierten Methode `double f(double x)` lassen sich Funktionswerte berechnen, z. B. `y = fun.f(x)`. In diesem Interface und den Interfaces `ArrayFunctionPointer` und `ParametrizedFunctionPointer` werden weitere Funktionen definiert, welche als Parameter der Methoden Teilmengen der Variablen und Konstanten x, y, z, t, x0, x1, ..., k0, k1, ... zulassen. Der Konstruktor `Function(String term, FunctionVariables fvars)` erlaubt die Angabe zulässiger Variablen `fvars`. Der in dieser Klasse gekapselte Parser erzeugt unter Verwendung der oben gegebenen Grammatik und mit Hilfe der Klasse `StringParser` aus der Zeichenkette `term` einen Baum mit Objekten vom Typ `FunctionNode`. Methoden wie `FunctionNode element(StringParser sp)` zerlegen die Zeichenkette in Teilausdrücke und erzeugen durch zahlreiche Fallunterscheidungen die für den Baum benötigten Objekte.

Die Variablen und Parameter, auf denen Funktionsauswertungen operieren, sind an die Klasse `FunctionVariables` delegiert. Dadurch können sich Funktionenscharen einen gemeinsamen Satz von Variablen teilen, wie dies z. B. bei der numerischen Lösung von Differentialgleichungssystemen geschieht.

5.4 Numerische Integration

Im Folgenden sollen Verfahren angegeben werden, mit denen das Integral einer Funktion numerisch berechnet werden kann, deren Werte tabelliert sind. Im Gegensatz dazu stehen Verfahren wie z. B. die Romberg-Integration zur numerischen Berechnung von Stammfunktionen, deren Ableitung in analytischer Form vorliegt.

Liegen N tabellierte Datenpaare x_i, y_i mit äquidistanten Abszissenwerten und $\Delta = x_i - x_{i-1}$ vor, so lassen sich deren Integrale f_N nach den Newton-Cotes-Formeln berechnen:

$$f_1 = 0 \tag{5.61}$$

$$f_2 = \frac{1}{2}\Delta\,(y_0 + y_1) \tag{5.62}$$

$$f_3 = \frac{1}{3}\Delta\,(y_0 + 4y_1 + y_2) \ \text{(Simpson-Regel)} \tag{5.63}$$

$$f_4 = \frac{3}{8}\Delta\,(y_0 + 3y_1 + 3y_2 + y_3) \ \text{(3/8-Regel)} \tag{5.64}$$

$$f_5 = \frac{4}{90}\Delta\,(7y_0 + 32y_1 + 12y_2 + 32y_3 + 7y_4) \tag{5.65}$$

$$f_6 = \frac{5}{288}\Delta\,(19y_0 + 75y_1 + 50y_2 + 50y_3 + 75y_4 + 19y_5) \tag{5.66}$$

Für größere Werte von N wird die erweiterte geschlossene Simpson-Regel verwendet:

$$f_N = \Delta\left[\frac{1}{3}y_0 + \frac{4}{3}y_1 + \frac{2}{3}y_2 + \frac{4}{3}y_3 + \cdots + \frac{2}{3}y_{N-3} + \frac{4}{3}y_{N-2} + \frac{1}{3}y_{N-1}\right] \tag{5.67}$$

Man erkennt, dass diese Formel nur für ungeradzahlige Werte von N anwendbar ist. Für geradzahlige Werte wird diese Formel mit der 3/8-Regel verknüpft, welche auf die letzten 4 Summanden angewandt wird:

$$f_4 = \frac{3}{8}\Delta\left(y_0 + 3y_1 + 3y_2 + y_3\right) \tag{5.68}$$

Die Verknüpfung ergibt dann

$$f_N = \Delta\left[\frac{1}{3}y_0 + \frac{4}{3}y_1 + \frac{2}{3}y_2 + \cdots + \frac{4}{3}y_{N-5} + \frac{17}{24}y_{N-4} + \frac{9}{8}y_{N-3} + \frac{9}{8}y_{N-2} + \frac{3}{8}y_{N-1}\right] \tag{5.69}$$

Verfügt man nicht über äquidistante Abszissenwerte, so lässt sich die Trapezregel anwenden, die allerdings ungenauer ist. Die summierte Trapezregel lautet:

$$\int_{x_0}^{x_N} f(x)dx \approx \frac{x_1 - x_0}{2} f\left(x_0\right) + \frac{1}{2}\sum_{i=1}^{N-1}\left(x_{i+1} - x_{i-1}\right) f\left(x_i\right) + \frac{x_N - x_{N-1}}{2} f\left(x_N\right) \tag{5.70}$$

Berechnet man nicht das Integral über einen Datensatz, sondern möchte dessen Flächeninhaltsfunktion berechnen, so setzt man den ersten Wert gleich 0, berechnet die folgenden Werte nach den Newton-Cotes-Formeln und kann dann für geradzahlige Werte weiter folgende Rekursionsformel anwenden:

$$f_i = f_{i-2} + \frac{1}{3}dx\left(y_{i-2} + 4y_{i-1} + y_i\right) \tag{5.71}$$

An ungeradzahligen Stellen ergibt sich durch Anwendung der 3/8-Regel

$$f_i = f_{i-3} + \frac{3}{8}dx\left(y_{i-3} + 3y_{i-2} + 3y_{i-1} + y_i\right) \tag{5.72}$$

Liegen keine äquidistanten Abszissenwerte vor, so ergibt sich unter Verwendung der Trapezformel

$$f_0 = 0\ , \tag{5.73}$$

$$f_1 = \int_{x_0}^{x_1} f\left(x\right) dx \approx \frac{1}{2}\left(x_1 - x_0\right)\left(y_1 + y_0\right) \tag{5.74}$$

und

$$f_N = \int_{x_0}^{x_N} f(x)dx \approx f_{N-1} + \frac{1}{2}\left(x_N - x_{N-1}\right)\left(f\left(x_N\right) + f\left(x_{N-1}\right)\right)\ . \tag{5.75}$$

Methoden zur Auswertung dieser Formeln befinden sich in der Klasse `Numerical`.

5.5 Allgemeine Methode der kleinsten Quadrate

Mit der allgemeinen Methode kleinster Quadrate kann eine Linearkombination beliebiger Funktionen

$$y\left(x\right) \approx F\left(x\right) = \sum_{j=1}^{M} a_j f_j\left(x\right) \tag{5.76}$$

an einen Datensatz angepasst werden. Die Fehlerquadratfunktion

$$\chi^2 = \sum_{i=1}^{N} \left(\frac{y_i - F(x_i)}{\sigma_i} \right)^2 \tag{5.77}$$

mit der Standardabweichung σ muss dazu minimiert werden. Definiert man die Matrix A und den Vektor $\vec{b}$ mit den Elementen

$$A_{ij} = \frac{f_j(x_i)}{\sigma_i} \text{ und } b_i = \frac{y_i}{\sigma_i} \ , \tag{5.78}$$

so kann man mit dem Vektor $\vec{a}$ der Werte a_j auch schreiben

$$\chi^2 = \left| A\vec{a} - \vec{b} \right| \ . \tag{5.79}$$

Die Lösung dieses linearen Gleichungssystemes führt häufig zu singulären oder numerisch annähernd singulären Matrizen. Solche Systeme lassen sich mit Hilfe der Singulärwertzerlegung (Singular Value Decomposition, SVD) lösen. Jede rechteckige Matrix A lässt sich zerlegen nach

$$A[N, M] = U[N, M] \cdot W[M, M] \cdot V^T[M, M] \tag{5.80}$$

mit den Matrixdimensionen M und N. Die Spalten von U und V sind orthonormal, ebenso die Zeilen von V. W ist eine Diagonalmatrix mit den Diagonalelementen w_i. Zur Bestimmung des Vektors $\vec{a}$, der χ^2 minimiert, berechnet man die inverse Matrix

$$A^{-1} = V \cdot W^{-1} U^T \ . \tag{5.81}$$

Die Diagonalmatrix lässt sich einfach invertieren, indem man die Diagonalelemente durch ihre Kehrwerte ersetzt. Ist A keine singuläre Matrix, so existiert eine optimale Lösung mit $\chi^2 = 0$, die man mit den Spaltenvektoren U_i und V_i der Matrizen berechnen kann nach

$$\vec{a} = \sum_{i=1}^{M} \left(\frac{U_i \cdot \vec{b}}{w_i} \right) V_i \ . \tag{5.82}$$

Für singuläre Matrizen werden manche der Diagonalelemente $w_i = 0$ oder sehr klein, wodurch über die Kehrwertbildung große Beiträge von Funktionen entstehen, deren Funktionswerte sich gegenseitig aufheben. Es lässt sich zeigen, dass ein Nullsetzen der Kehrwerte zu Werten führt, welche die Fehlerquadratfunktion minimieren.

Die Klasse `LinearFit` stellt verschiedene Methoden zur Verfügung, mit denen eine solche Parameteroptimierung vorgenommen werden kann. Die Berechnung der Singulärwertzerlegung ist in die Klasse `SingularValueDecomposition` des im Abschnitt 5.9 vorgestellten Pakets `jamax` ausgelagert.

Abbildung 5.10 zeigt einen Dialog der Klasse `LinearFitDialog` zur Eingabe von Parametern für die lineare Optimierung. In der Tabelle können Funktionen angegeben werden, deren Linearkombination optimiert werden soll. Die Koeffizienten können vorgegeben, berechnet oder konstant gehalten werden. Für jeden berechneten Koeffizienten wird dessen Varianz angegeben. Mit Hilfe der Koeffizienten und Funktionen kann in einem zu spezifizierenden Intervall ein neuer Datensatz berechnet werden, der als neues Diagramm angezeigt wird.

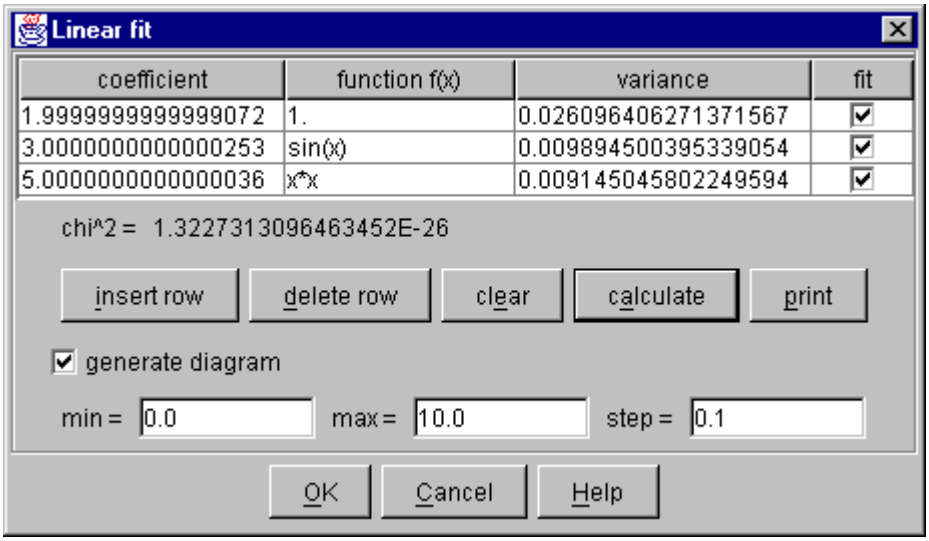


Abbildung 5.10: Dialog zur linearen Optimierung

5.6 Nichtlineare Optimierung nach Levenberg-Marquardt

Das Verfahren von Levenberg und Marquardt [75] gilt als Standardmethode für die nichtlineare Optimierung von Funktionsparametern. Für eine gegebene Funktion $f(x, \vec{a})$ sind diejenigen Parameter $\vec{a} = (a_1, a_2, \ldots, a_N)$ gesucht, für welche diese Funktion einen gegebenen Datensatz optimal modelliert.

Die Herleitung der verwendeten Formel soll hier nur skizziert werden. Auch im nichtlinearen Fall lautet die Fehlerquadratfunktion

$$\chi^2(\vec{a}) = \sum_{i=1}^{N} \left(\frac{y_i - f(x_i, \vec{a})}{\sigma_i} \right)^2 \tag{5.83}$$

Diese lässt sich an der Stelle $\vec{a}^*$ in eine Taylorreihe bis zur zweiten Ordnung entwickeln durch

$$\chi^2(\vec{a}) = \chi^2(\vec{a}^*) + \sum_i \frac{\partial \chi^2(\vec{a}^*)}{\partial a_i}(a_i - a_i^*) + \tfrac{1}{2} \sum_{i,j} \frac{\partial^2 \chi^2(\vec{a}^*)}{\partial a_i \partial a_j}(a_i - a_i^*)(a_j - a_j^*) + \cdots \tag{5.84}$$

Unter Verwendung der Matrix M und der Vektoren $\vec{b}$, $\vec{c}$ und $\vec{d}$ mit

$$[M]_{ij} = \frac{\partial^2 \chi^2(\vec{a}^*)}{\partial a_i \partial a_j},\ \vec{b} = -\nabla \chi^2(\vec{a}^*),\ \vec{c} = \chi^2(\vec{a}^*) \text{ und } \vec{d} = \vec{a} - \vec{a}^* \tag{5.85}$$

kann man auch schreiben:

$$\chi^2(\vec{a}) = \vec{c} - \vec{b}\vec{d} + \tfrac{1}{2}\vec{d}M\vec{d}. \tag{5.86}$$

Der Gradient dieser Funktion ist

$$\nabla \chi^2(\vec{a}) = -\vec{b} + M\vec{d}. \tag{5.87}$$

In der Nähe des Minimums der Fehlerquadratfunktion wird dieser Gradient $\nabla\chi^2(\vec{a}) = 0$ und es folgt

$$\vec{d} = M^{-1}\vec{b}. \tag{5.88}$$

Diese Gleichung entspricht dem Newton-Verfahren zur Suche der Nullstellen der Gradientenfunktion bzw. der Minima von $\chi^2(\vec{a})$ und heißt Quasi-Newton-Verfahren. Ebenso wie das Newton-Verfahren konvergiert eine Iteration von $\vec{a}$ zu $\vec{a}^*$ sehr schnell, die gemachten Näherungen gelten jedoch nur in der Nähe des Minimums. Für Parametervektoren $\vec{a}$, die vom Minimum weiter entfernt sind, nähert man sich diesem, indem man dem Gradienten in kleinen Schritten *const* folgt (Methode des steilsten Abstiegs, steepest descent):

$$\vec{d} = const \cdot \vec{b}. \tag{5.89}$$

Die in den obigen Gleichungen auftretenden partiellen Ableitungen sind:

$$\frac{\partial\chi^2}{\partial a_k} = \frac{-2}{\sigma_i^2}\sum_{i=1}^{N}\left(y_i - f(x_i)\right)\frac{\partial f(x_i)}{\partial a_k} \tag{5.90}$$

und

$$\frac{\partial^2\chi^2}{\partial a_k \partial a_l} = \frac{2}{\sigma_i^2}\sum_{i=1}^{N}\frac{\partial f(x_i)}{\partial a_k}\frac{\partial f(x_i)}{\partial a_l} - \left(y_i - f(x_i)\right)\cdot\frac{\partial^2 f(x_i)}{\partial a_k \partial a_l} \tag{5.91}$$

Man definiert

$$\beta_k = -\frac{1}{2}\frac{\partial\chi^2(\vec{a}^*)}{\partial a_k} \text{ und } \alpha_{kl} = \frac{1}{2}\frac{\partial^2\chi^2(\vec{a}^*)}{\partial a_k \partial a_l} \approx \sum_{i=1}^{N}\frac{\partial f(x_i)}{\partial a_k}\frac{\partial f(x_i)}{\partial a_l} \tag{5.92}$$

unter Vernachlässigung des Terms der zweiten Ableitungen in Gleichung 5.91 für α_{kl}. Diese Näherung ist zulässig, da sich die Abweichungen $y_i - f(x_i)$ bei statistischer Abweichung der anzupassenden Werte herausmitteln. Levenberg fand, dass ein sinnvoller Wert für die Schrittweite *const* in Gleichung 5.89 mit Hilfe des Faktors λ kontrolliert werden kann:

$$const = \frac{1}{\lambda\alpha_{ll}} \tag{5.93}$$

Definiert man weiter eine neue Matrix mit den Elementen

$$\alpha'_{kl} = \begin{cases} \alpha_{kl}\cdot(1+\lambda) & \text{für } k = l \\ \alpha_{kl} & \text{für } k \neq l \end{cases}, \tag{5.94}$$

so kann man mit den Vektorelementen d_l des Vektors $\vec{d}$ schreiben:

$$\sum_{l=1}^{M}\alpha'_{kl}d_l = \beta_k. \tag{5.95}$$

Diese Gleichung geht für kleine Werte von λ in das Quasi-Newton-Verfahren nach Gleichung 5.88 über, das in der Nähe des Minimums Anwendung findet. Für große Werte von λ hingegen, die einer größeren Entfernung vom Minimum entsprechen, wird die Matrix α' durch ihre Diagonalelemente dominiert und dieses Verfahren geht in das Verfahren des steilsten Abstiegs nach Gleichung 5.89 über.

Marquardt entwickelte ein iteratives Verfahren zur Berechnung optimaler Koeffizienten durch Variation des Faktors λ. In jedem Rechenschritt werden die Koeffizienten $\vec{a}$ um den Vektor $\vec{d}$ verändert, der durch Lösung des linearen Gleichungssystems 5.95 berechnet wird. Falls dabei der Wert der Fehlerquadratfunktion abnimmt, wird λ um eine Größenordnung verkleinert, andernfalls wird er um eine Größenordnung vergrößert. Die Iteration wird abgebrochen, falls nur noch minimale absolute oder relative Verbesserungen erreicht werden oder eine bestimmte Iterationstiefe überschritten wird. Der Erfolg der Optimierung hängt entscheidend von der Wahl der Anfangsparameter ab. Generell auftretende Probleme bei der nichtlinearen Optimierung liegen in der Konvergenz gegen lokale Minima, einer sehr langsamen Konvergenz oder einer Divergenz.

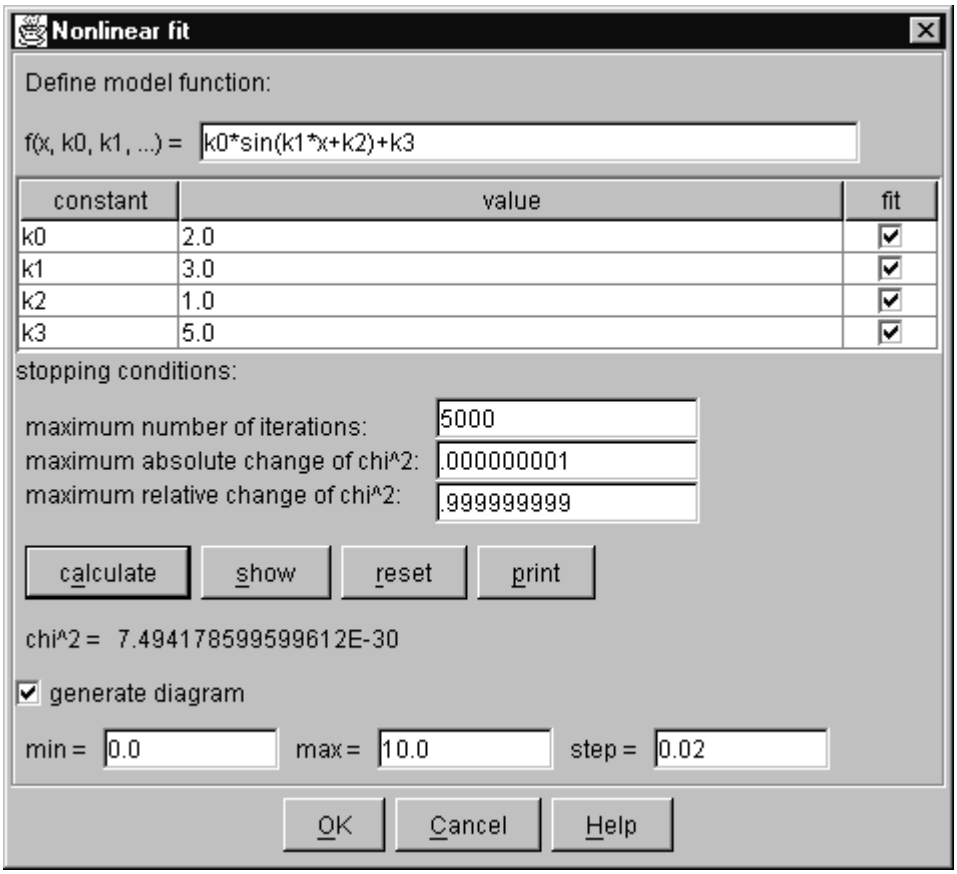


Abbildung 5.11: Dialog zur nichtlinearen Optimierung

Neben der erwähnten Verwendung dieses Verfahrens bei der Bestimmung von Koordinationszahlen kann der Benutzer nichtlineare Optimierungen von Koeffizienten für eine beliebige parametrisierte Funktion $f(x, k_0, k_1, ...)$ mit Hilfe des in Abbildung 5.11 dargestellten Dialogs der Klasse `NonlinearFitDialog` durchführen. Die in dieser Funktion auftretenden Parameter *constant* werden in eine Tabelle eingetragen und müssen dort mit Startwerten versehen werden. Für jeden Parameter kann unter *fit* angegeben werden, ob er

variiert oder konstant gehalten werden soll. Weitere Eingabeparameter sind die genannten Abbruchkriterien des Verfahrens. Falls eine Berechnung zu unsinnigen Werten divergiert hat, können mit *reset* die Lösungsparameter zurückgesetzt werden. Anzupassende Werte und die Modellfunktion werden optional gemeinsam in einem Multi-Diagramm angezeigt.

Die Berechnungen werden nach den oben angegebenen Formeln in der Klasse `LevenbergMarquardt` durchgeführt. Dem Konstruktor der Klasse werden die anzupassenden Werte, die Modellfunktion, Startparameter und ggf. Standardabweichungen und partielle Ableitungen übergeben. Werden keine Standardabweichungen angegeben, so wird für sie der Vorgabewert 1. verwendet. Die partiellen Ableitungen können als Objekt vom Typ `VectorFunctionPointer` übergeben werden. Wird wie z. B. durch den Dialog der Klasse `NonlinearFitDialog` nur die Modellfunktion übergeben, so werden die symbolischen Ableitungen der Modellfunktion mit Hilfe der Klasse `Function` berechnet. Numerische Ableitungen sind hierzu aufgrund ihrer Ungenauigkeit ungeeignet. Für die Abbruchkriterien werden Vorgabewerte verwendet, die mit entsprechenden Methoden ersetzt werden können. Die Parameter werden in den Methoden `iterate()` und `calculate()` berechnet. Das lineare Gleichungssystem 5.95 wird mit Hilfe der Methode `Matrix.solve()` des `jamax`-Paketes gelöst. Die optimierten Parameter sind über die Methode `double[] getParameters()` zugänglich.

5.7 Glättung von Daten nach dem Savitzky-Golay-Verfahren

Das Verfahren von Savitzky und Golay [76], das auch in den Numerical Recipes [69] beschrieben ist, wurde zur Bestimmung der Breite und Höhe von Maxima spektroskopischer Daten entwickelt. Es arbeitet als digitales Filter für Werte $y_i = f(x_i)$ mit äquidistanten x_i durch Bildung eines Mittelwertes

$$z_i = \sum_{n=-n_l}^{n_r} c_n y_{i+n} \tag{5.96}$$

Darin ist n_l die Zahl der Werte links des aktuellen Wertes und n_r die Zahl der Werte rechts des aktuellen Wertes. Im einfachsten Falle könnte man konstante Koeffizienten c_n verwenden, das bedeutet, man ersetzt einen Wert durch einen gleichgewichteten Mittelwert der Werte y_{i-n_l} bis y_{i+n_r}. Mathematisch ist diese Operation äquivalent mit dem Zeichnen einer Gerade, welche nach der Methode der kleinsten Fehlerquadrate optimal an diese Werte angepasst ist und die Berechnung eines Wertes y'_i auf der Gerade an der Stelle x_i. Dieses Verfahren lässt sich im Allgemeinen verbessern, indem man nicht an eine Gerade, sondern an ein Polynom vom Grad M anpasst, für dieses die Koeffizienten bestimmt und einen Wert an der gesuchten Stelle berechnet. Angewandt auf obige Formel ergeben sich dadurch unabhängig von der Kurve Koeffizienten c_n als Funktion von M, n_l und n_r. Mit diesem Algorithmus lassen sich zugleich die Ableitungen der Kurve berechnen, die sich bei einer Anpassung an die Punkte nach der Methode der kleinsten Quadrate ergäbe.

Als Faustregel für die Filterbreite $n_l + n_r$ wird eine Anzahl von Werten empfohlen, welche dem ein- bis zweifachen der Halbwertsbreite zu untersuchender Maxima entspricht. Höhere

(4 oder 6) Ordnungen von M sind besser geeignet, schmale Maxima zu reproduzieren, gleichzeitig geht dadurch jedoch die glättende Wirkung verloren. Es sei darauf hingewiesen, dass Glättungen mit Vorsicht eingesetzt werden sollten und nur dazu dienen sollten, eine graphische Darstellung übersichtlicher zu gestalten. Keinesfalls sollten sie in Zwischenschritten numerischer Auswertungen angewandt werden, da sie stets mit einer Verfälschung von Originaldaten einhergehen und mit Informationsverlust verbunden sind. Insbesondere ist es sinnlos, Daten zu glätten, die anschließend einer Fouriertransformation unterworfen werden, da diese durch Separation hochfrequenten Rauschens mit einer Glättung einhergeht. Auch die Savitzky-Golay-Filterung entspricht mathematisch einer Faltung, also der Fouriertransformation des Produktes zweier Funktionen.

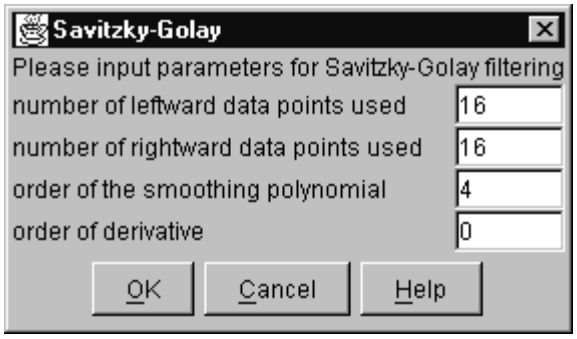


Abbildung 5.12: Dialog Savitzky-Golay-Filterung

Die Parameter für eine Savitzky-Golay-Filterung können in einen Dialog der Klasse `Savgol-Dialog` eingegeben werden, welcher sich von `ParameterDialog` ableitet. Er ist in Abbildung 5.12 dargestellt. Die Transformation wird durch die Klasse `SavitzkyGolay` berechnet. In der Methode `void Numerical.savgol(double[] c, int nl, int nr, int ld, int m)` werden die Koeffizienten c_n berechnet, wobei l_d der Grad der zu berechnenden Ableitung ist. Im Falle einer reinen Glättung ist also $l_d = 0$. Die Bestimmung der Polynomkoeffizienten führt über die Methode der kleinsten Quadrate zu einem linearen Gleichungssystem, das mit Hilfe einer LU-Zerlegung gelöst wird. Eine Komplikation ergibt sich dadurch, dass an den Intervallgrenzen der zu glättenden Daten nicht ausreichend viele Daten auf der linken bzw. rechten Seite zur Verfügung stehen. In diesen Fällen werden Koeffizienten für kleinere Werte von n_l oder n_r berechnet, wodurch sich an den Rändern asymmetrische Glättungen ergeben.

5.8 Die Klasse `Complex`

Bislang umfasst die Klassenbibliothek von Java keine Klasse zum Rechnen mit komplexen Zahlen. Diese Klasse stellt daher einige Methoden zur Verfügung, die häufig benötigt werden: Addition, Subtraktion, Multiplikation, Division, Abfragen von Real- und Imaginärteil, Norm, Betrag und Phase. Viele dieser Operationen wurden in zwei Varianten implementiert: als Klassenmethoden der Form `static Complex multiply(c1, c2)` und als Instanzmethoden der Form `Complex multiply(Complex c)`. Während die statischen Methoden die

beteiligten Objekte unverändert lassen, operieren die Instanzmethoden auf dem aktuellen Objekt. So sind Methodenaufrufe wie `Complex c = c1.add(c2).multiply(c3)` möglich, welche die komplexe Zahl `c1` durch das Ergebnis von $(c_1+c_2)\cdot c_3$ ersetzen und an c übergeben.

In [69] wird darauf hingewiesen, dass die Implementierung der Betragsbildung

$$|a+ib| = \sqrt{a^2+b^2} \tag{5.97}$$

leicht zu Speicherüber oder -unterläufen führt. Stattdessen berechnet man besser

$$|a+ib| = \begin{cases} |a|\sqrt{1+(b/a)^2} & |a| \geq |b| \\ |b|\sqrt{1+(a/b)^2} & |a| < |b| \end{cases} \tag{5.98}$$

Ähnliches gilt für die Berechnung von Quotienten:

$$\frac{|a+ib|}{|c+di|} = \frac{ac+bd}{c^2+d^2} + i\frac{bc-ad}{c^2+d^2} = \begin{cases} \frac{[a+b(d/c)]+i[b-a(d/c)]}{c+d(d/c)} & |c| \geq |d| \\ \frac{[a(c/d)+b]+i[b(c/d)-a]}{c(c/d)+d} & |c| < |d| \end{cases} \tag{5.99}$$

Auch hier ist die Auswertung des rechten Ausdrucks derjenigen des mittleren Ausdrucks zur Vermeidung von Rundungsfehlern vorzuziehen.

Weiterhin wurden einige Methoden zur Arbeit mit Arrays komplexer Zahlen implementiert. Neben der naheliegenden Möglichkeit, entsprechend `Complex[] z = new Complex[len]` ein Array mit Instanzen der Klasse `Complex` anzulegen, arbeiten viele in der Literatur beschriebene numerische Methoden für komplexe Zahlen mit Arrays von Fließkommazahlen, die in der Form `[re0, im0, re1, im1, ...]` abgelegt sind. Dazu wurden statische Methoden formuliert, die solche Felder erzeugen oder aus ihnen Real- und Imaginärteile extrahieren.

5.9 Das Paket `jamax`

Dieses Paket ist fast identisch mit dem Java Matrix Package JAMA. Das JAMA-Paket wurde gemeinsam von der MathWorks-Gruppe und vom National Institute of Standards and Technology (NIST) [77] entwickelt. Dieses Paket enthält grundlegende Funktionen zur Linearen Algebra mit reellwertigen dichten Matrizen. Es umfasst folgende sechs Klassen: `Matrix`, `CholeskyDecomposition`, `LUDecomposition`, `QRDecomposition`, `SingularValueDecomposition` und `EigenvalueDecomposition`. Obwohl sich Vektoren prinzipiell als einzeilige bzw. einspaltige Matrizen des JAMA-Paketes darstellen lassen, wurde zum vereinfachten Umgang mit Vektoren zusätzlich die Klasse `Vector` entwickelt und bestehende Klassen teilweise um Methoden ergänzt, die auf Vektoren operieren. Die genannten Klassen arbeiten mit Fließkommavariablen doppelter Genauigkeit. Die `Matrix`-Klasse enthält Methoden zur Addition, Multiplikation und Normierung von Matrizen sowie für den Zugriff auf Submatrizen und Matrixelemente. Die Zerlegungsklassen erlauben unter anderem die Lösung linearer Gleichungssysteme, die Berechnung von Determinanten und inversen Matrizen.

Kapitel 6

Allgemeine Datenverarbeitung

Neben spezifischen Funktionen zur Auswertung der Diffraktogramme stellt das Programm viele Möglichkeiten zur Verfügung, mit denen die Auswertung unterstützt wird oder Daten auf geeignete Weise bearbeitet werden können.

6.1 Grundfunktionen

Sämtliche Grundfunktionen wie das Erzeugen neuer Diagramme durch Angabe eines Datenintervalls, das Laden und Speichern im programmspezifischen Format (siehe Abschnitt 3.4.8) der Import von ASCII-Dateien und der Export von ASCII-Dateien sowie Diagrammen im EPS-Format werden unterstützt und sind über das Menü *File* zugänglich. Der mehrfarbige Ausdruck von Diagrammen als hochauflösende Vektorgraphik ist möglich. Durch Auswahl eines Bereiches mit der Maus oder durch Angabe entsprechender Werte wird das gewünschte Datenintervall auf der gesamten zur Verfügung stehenden Breite dargestellt. Für die aktuelle Position des Mauszeigers können die Koordinaten im Datenraum angegeben werden. Mit Hilfe eines Dialogs der Klasse `DiagramConfigurationDialog` (siehe Abbildung 3.15) können die Darstellungsparameter von Diagrammen konfiguriert werden.

6.2 Mathematische Transformationen

In Kapitel 5 wurden verschiedene numerische Klassen vorgestellt, die im Zusammenhang mit Auswertungsschritten der Diffraktogramme von Bedeutung sind. Zum Teil haben sie zudem eine eigenständige Bedeutung und können über Menüeinträge des Menüs *Calculate* der Klasse `DiagramCalculationMenu` verwendet werden. Die eigentlichen Transformationen finden in inneren Klassen dieser Klasse statt bzw. werden dort eingeleitet.

Die einfachste Möglichkeit Daten zu verändern besteht in der Anwendung einer Funktion $x \mapsto f(x)$ auf die Abszissenwerte x über den Menüeintrag *Transform x* oder einer Funktion $y \mapsto f(x, y)$ auf die Ordinatenwerte y über den Menüeintrag *Transform y*.
Abbildung 6.1 zeigt einen Dialog der Klasse `FormulaInputDialog` zur Transformation der Ordinatenwerte. Ein entsprechender Dialog existiert zur Transformation der Abszissenwerte. Beide Dialoge ermöglichen die Eingabe einer Funktionsvorschrift, welche der in der Klasse `Function` definierten Syntax genügt (siehe Abschnitt 5.3.1).

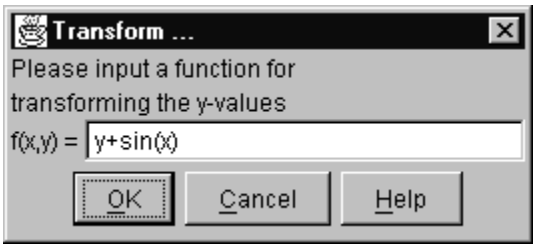


Abbildung 6.1: Dialog zur Transformation der Ordinatenwerte

Durch Wahl des Menüeintrags *Fourier transform* öffnet sich ein Dialog der Klasse `FourierDialog`, mit dem eine Fouriertransformation durchgeführt werden kann. Zur Auswahl stehen verschiedene Modi (normal/invers), Algorithmen (DFT/FFT/DST) und Anteile (real/imaginär). Mit *Integrate* wird die Flächeninhaltsfunktion berechnet, *Get area* berechnet die Fläche unter einer Kurve, *Derive* berechnet die Tangentensteigungsfunktion und *Symbolic derivation* die symbolische Ableitung einer einzugebenden Funktionsgleichung. Die Berechnung numerischer Ableitungen ist sehr ungenau und wird nur als Umkehroperation zur Bildung der Flächeninhaltsfunktion angeboten. Die Glättung von Daten mit Hilfe der Savitzky-Golay-Filterung kann über den Menüeintrag *Smooth* berechnet werden.

Zur Generierung eines normalverteilten Rauschens kann *Generate noise* eingesetzt werden. Das Rauschen setze sich zusammen aus einem Untergrundrauschen s_0 und einem Rauschen s_n in Abhängigkeit der Signalintensität i. Als Signalintensität wird die Differenz zwischen dem größten und kleinsten Ordinatenwert eines Datensatzes aufgefasst. Es wird weiter angenommen, dass die verrauschten Messwerte eine Gaußverteilung um die idealen Werte annehmen. Die Verteilung um einen Erwartungswert null ist dann

$$f(t) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{t}{\sigma}\right)^2}$$

mit der Standardabweichung σ. Die Integration dieser auf 1 normierten Funktion ergibt die Fehlerfunktion

$$x = \operatorname{erf}(y) = \int_{t=-\infty}^{y} f(t)\, dt$$

Dieses Integral ist nur numerisch lösbar und wird in der Klasse `Noise` berechnet. Unter Ausnutzung von $\operatorname{erf}(0) = \frac{1}{2}$ und Punktsymmetrie zu diesem Kurvenpunkt werden die Werte $\operatorname{erf}(y)$ durch Integration im Bereich von $t = 0$ bis $t_{max} = 10\sigma$ berechnet und in Feldern tabelliert. Dabei wird für den Zusammenhang einzugebender Parameter mit der Standardabweichung σ des Rauschens $\sigma = i/s_n + s_0$ angenommen. Mit Hilfe der Funktion `Math.random()` wird eine Zufallszahl x_{rnd} im Intervall $[0; 1[$ berechnet und mit `Numerical.locateOrdered(double x[], double xrnd)` derjenige Wert $x = \operatorname{erf}(y)$ bestimmt, welcher der Zufallszahl am nächsten liegt. Über den zugehörigen y-Wert an derselben Position erhält man einen normalverteilten Fehler mit einem maximalen Betrag von t_{max}, der zum Ordinatenwert des Datensatzes addiert wird.

Weitere Funktionen sind die Interpolation, die Vertauschung der Abszissenwerte gegen die Ordinatenwerte und die Sortierung von Daten nach aufsteigenden Abszissenwerten. Die Sortierroutine des Transformers `Sort` befindet sich in der Klasse `DoublePair` im Paket `pointer`. In den Java-Collections befinden sich verschiedene Sortierroutinen für Standarddatentypen, jedoch können keine Datenpaare sortiert werden. Dazu wurde die Klasse `DoublePair` implementiert, welche zwei doppelt genaue Fließkommakonstanten kapselt. Zur Sortierung wird das Java-Interface `Comparable` implementiert. Nach Umwandlung zweier Arrays `double[] x` und `double[] y` in ein Array `DoublePair[] dp` in der Methode `DoublePair[] toDoublePairArray(double[] x, double[] y)` kann dieses in `void sortByAscendingX(double[] x, double[] y)` über einen Java-Algorithmus (modifizierter Merge-Sort) sortiert werden und mit `void toDoubleArrays(DoublePair[] dpArr, double[] x, double[] y)` in die alten Arrays zurück übertragen werden.

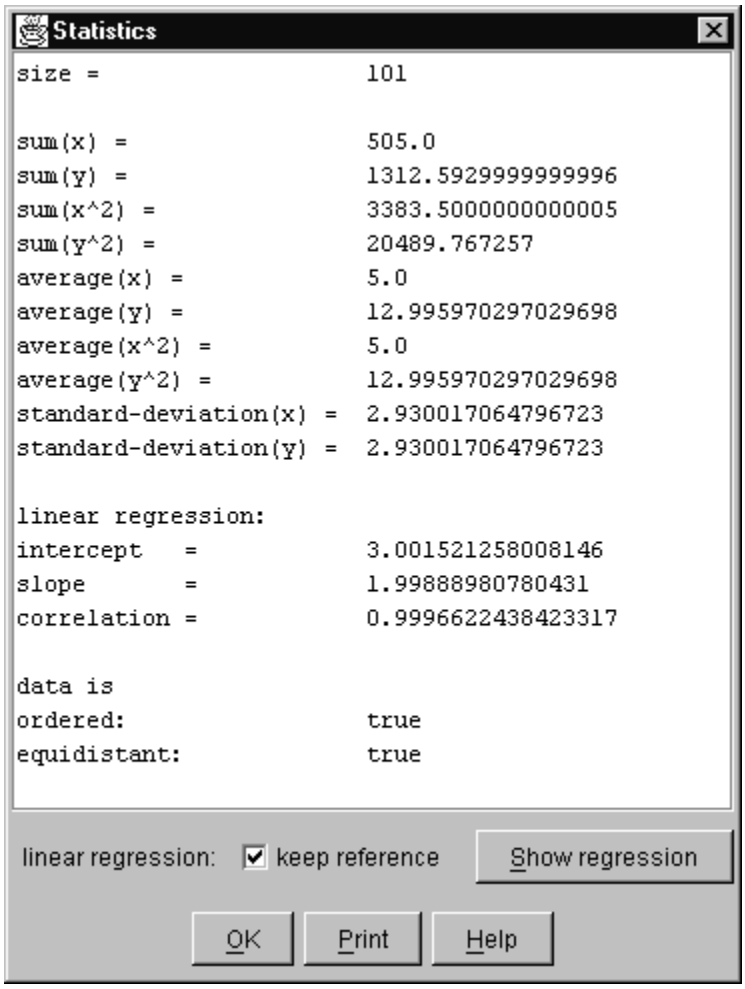


Abbildung 6.2: Dialog zur Berechnung statistischer Daten und linearen Regression

Abbildung 6.2 zeigt einen Dialog der Klasse `StatisticsDialog`, über den grundlegende statistische Daten berechnet werden können, darunter die Parameter für lineare Regression. Die

sich aus den Regressionsparametern ergebende Gerade kann zusammen mit den betrachteten Daten in einem Diagramm der Klasse `RegressionDiagram` dargestellt werden, die sich von der Klasse `MultiDiagram` ableitet. Das Diagramm wird bei Wahl der Option *keep reference* aktualisiert, sobald sich der Basisdatensatz ändert, d. h. es wird eine neue Regressionsgerade berechnet, deren Parameter über ein Menü dieses Diagramms angezeigt werden können.

6.3 Rückgängigmachung und Wiederherstellung von Transformationen

Im Abschnitt 3.1 wurde ein Mechanismus vorgestellt, mit dem Transformationen wiederholt, rückgängig gemacht und wiederhergestellt werden können. Im Menü *Edit* befinden sich die Einträge *undo*, *undolist*, *redo*, *redolist*, *repeat* und *repeatlist*, mit denen einzelne oder mehrere Transformationen in dieser Weise angewandt werden können. Wie bereits ausgeführt wurde, werden Transformationen eigentlich nicht rückgängig gemacht, sondern ab einem Referenzzustand neu ausgeführt. Es ist daher ungünstig, mehrere einzelne Undo-Operationen hintereinander auszuführen. Für die Ausführung einer Gruppe von Transformationen stehen Dialoge der Klasse `HistoryDialog` zur Verfügung, von denen ein Vertreter in Abbildung 6.3 dargestellt ist. Er enthält eine Liste bisher ausgeführter Transformationen und eine zweite Liste mit Parametern, die jeweils zu diesen Transformationen gehören. Mit *reset history* kann der Refererenzzustand festgelegt werden, hinter den nicht zurückgegangen werden kann.

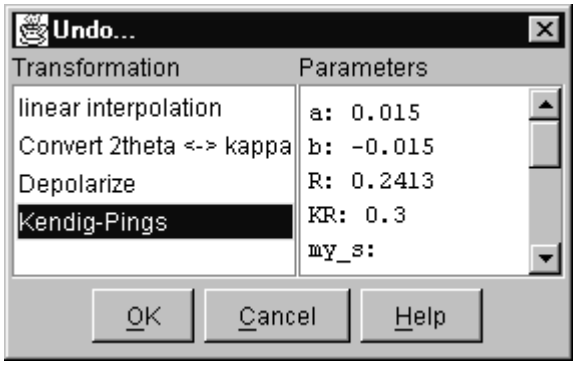


Abbildung 6.3: Dialog zur Ausführung von Gruppentransformationen

Die genannten Menüeinträge werden von der Klasse `DiagramEditMenu` verwaltet, während der Undo/redo-Mechanismus in der Klasse `Data` lokalisiert ist. Die Aktivierung bzw. Desaktivierung der Menüeinträge erfolgt über die inneren Klassen `DiagramEditMenu$MIEUndoable` und `DiagramEditMenu$MIERedoable`, welche den Zustand erfragen, ob der durch das aktuelle `SingleDiagram` referenzierte Datensatz die Operationen Undo bzw. Redo zulässt.

6.4 Das Arbeiten mit mehreren Datensätzen

Über den Menüeintrag *Duplicate data* lässt sich ein Datensatz duplizieren. Diese Operation wird beispielsweise benötigt, um Datensätze vor und nach einer Transformation vergleichen zu können. Mit Hilfe des Menüeintrags *Multiple data* lassen sich mehrere Diagramme zu einem Multidiagramm zusammenfassen, in welchem mehrere Datensätze gleichzeitig angezeigt werden. Auch diese Diagramme lassen sich skalieren, konfigurieren, laden, speichern und drucken. Mit *Extract data* lassen sich umgekehrt einzelne Datensätze aus einem Multi-Diagramm extrahieren und daraus neue Single-Diagramme erzeugen.

Mit *Combine* werden mehrere Datensätze zu einem einzigen neuen Datensatz kombiniert. Diese Operation kann z. B. verwendet werden, wenn ein Messbereich aus experimentellen Gründen in mehrere Teilbereiche zerlegt wurde, die wieder aneinandergehängt werden sollen. Im Überlappungsbereich wird für Daten mit identischen Abszissenwerten der Mittelwert der Ordinatenwerte gebildet. Datenpaare mit unterschiedlichen Abszissenwerten werden in der richtigen Reihenfolge zusammengefügt. Die Kombination erfolgt über einen Dialog der Klasse `CombineDialog`. Es ist auch möglich, Datensätze zu verwenden, deren Abszissenwerte nicht aufsteigend sortiert sind. In diesem Fall werden die Daten beider Datensätze aneinandergehängt.

Mit dem Menüeintrag *Column operations* wird ein Dialog der Klasse `MultiColumnDialog` geöffnet, wie er in Abbildung 6.4 dargestellt ist[1]. Der Dialog enthält eine Tabelle mit den Abszissen- und Ordinatenwerten aller Datensätze, welche die gleiche Länge haben wie der aktuelle Datensatz. Die Liste *Assignments* enthält für jeden dieser Datensätze jeweils die Abszissen- und Ordinatenwerte und ordnet diesen jeweils eine indizierte Variable x_i zu. Nur die ausgewählten Zeilen werden in der Tabelle dargestellt. Es ist nun möglich, die Daten mehrerer Spalten über eine Funktion zu verknüpfen. Unter Verwendung der Tabellenspalten x_i können durch eine Funktion $x_j = f(x_0, x_1, ...)$ neue Spalten definiert werden, die zur Erzeugung eines neuen Datensatzes verwendet werden.

Ein Beispiel für die Anwendung dieses Dialoges wäre die Berechnung der Atomstreuung von $BiCl_3$ bei der Betrachtung einer ionischen Struktur. Die Atomstreubeiträge der Ionen Bi^{3+} und Cl^- lassen sich wie im Abschnitt 4.2.4 gezeigt berechnen. Mit Hilfe der hier vorgestellten Operation lässt sich dann die Atomstreuung als Summe der Einzelbeiträge berechnen. Weitere Anwendungsmöglichkeiten sind z. B. die Aufsummierung oder Mittelwertbildung von Mehrfachmessungen. Diese häufig benötigten Spezialfälle wurden in die Dialogoptionen aufgenommen.

6.5 Editieren von Daten

In das Programm wurde ein Editor integriert, mit dem einzelne Werte und Datenbereiche angezeigt und verändert werden können.
Abbildung 6.5 zeigt einen Dialog der Klasse `DataEditorDialog`, mit dem Daten editiert

[1] Aus Platzgründen wurden in der abgebildeten Tabelle einige Datenzeilen entfernt

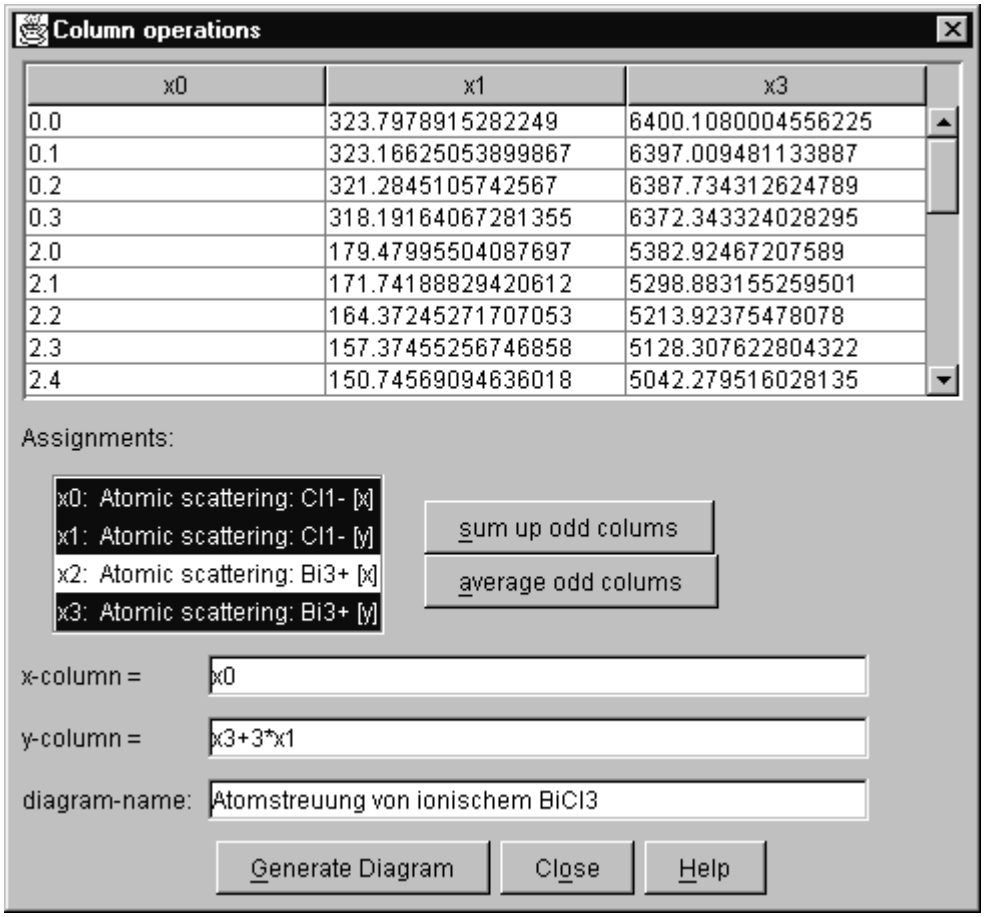


Abbildung 6.4: Dialog für Spaltenoperationen

werden können, wobei gleichzeitig die Darstellung des zugehörigen Diagramms aktualisiert wird. Es ist möglich, die Zahlenwerte einzelner Daten in der Tabelle zu verändern, wobei jede dieser Editieroperation als Transformation ausgeführt wird. Für die Transformationen von Teilbereichen stehen die im Abschnitte 6.2 vorgestellten Operationen *Transform x*, *Transform y*, *Interpolate*, *Smooth* und *Sort all* zur Verfügung. Beispielsweise lassen sich mit *Interpolate* an einzelnen Stellen Ausreißer wie Reflexe kristalliner Messzellen bequem entfernen. Ein hypothetisches Beispiel dafür ist in der Abbildung dargestellt. Die Operationen *Copy*, *Cut* und *Paste* zeigen das erwartete Verhalten zur Entfernung, Verschiebung und Duplizierung von Teilbereichen. *Trim* ist die zu *Cut* komplementäre Operation und entfernt alle nicht-selektierten Datenzeilen. Mit *Insert* lassen sich zu generierende Datenbereiche einfügen. Mit *Extract* wird ein Teilbereich eines Diagramms entfernt und daraus ein neues Diagramm erzeugt. Es kann also in Umkehrung zur Operation *Combine* im vorhergehenden Abschnitt eine Zerlegung in Teildiagramme durchgeführt werden. Die Abkürzung *NaN* steht in Java für „Not a number“ und dient der Kennzeichnung von Werten, die bei nicht erlaubten Operationen entstanden wie z. B. 1/0. Java kann mit diesen Werten arbeiten, doch können diese Probleme bereiten, wenn sie in Datensätzen enthalten sind. Die Operationen *Find NaNs* bzw. *Replace NaNs* suchen bzw. ersetzen diese Werte. Alle mit diesem Dialog gemachten Änderungen sind zunächst lokal und können einzeln rückgängig gemacht werden.

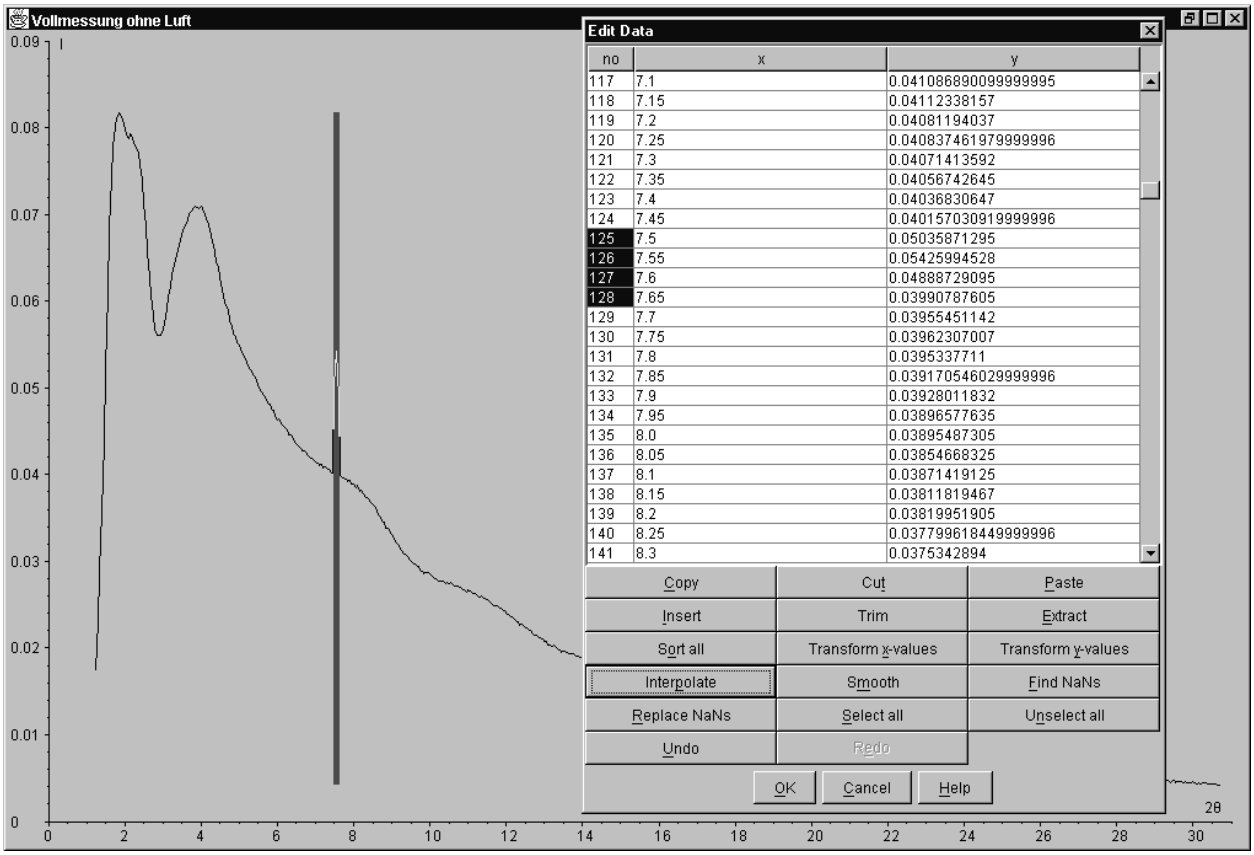


Abbildung 6.5: Das Editieren von Daten

Bei bestätigender Beendigung des Dialogs werden diese Operationen zu einer Gruppenoperation vom Typ `CompoundTransformer` zusammengefasst und als solche auf den Originaldaten ausgeführt.

Kapitel 7

Anwendung

7.1 Systemvoraussetzungen

Die Anwendung benötigt als Laufzeitumgebung die Java Virtual Machine der Java Platform 2 Standard Edition Version 1.3 oder höher. Diese stellt unter anderem die Firma Sun für folgende Betriebssysteme zur Verfügung: Solaris SPARC mit x86 Architektur, Linux mit x86-Architektur und Microsoft Windows 95/98/ME/2000 und NT 4.0 mit Intel-Prozessoren. Sowohl die Programmentwicklungsumgebung (Java-SDK) als auch die Laufzeitumgebung (JRE) können unter der Adresse `http://java.sun.com/j2se/` kostenlos heruntergeladen werden. Entsprechende Umgebungen sind auch von anderen Firmen für diese und viele weitere Betriebssysteme erhältlich.

Die Laufzeitumgebung für Windows, die für Entwicklung und Test unter Windows NT 4.0 verwendet wurde, benötigt mindestens einen Pentium 166 MHz-Prozessor und 32 MB RAM, wobei 48 MB RAM dringend empfohlen werden. Die Entwicklungsumgebung erfordert mindestens 65 MB Plattenspeicher und deren Dokumentation 120 MB.

7.2 Installation und Ausführung des Programms

Neben der genannten Laufzeitumgebung benötigt das Programm einige Pakete, die derzeit nicht zum Standardumfang von Java gehören. Es sind dies die Dateien `jh.jar` für das Java-Hilfesystem JavaHelp und die Dateien `crimson.jar`, `xalan.jar` und `jaxp.jar` für die Arbeit mit xml-Dateien. Letztere Dateien sollen ab Version 2-1.4 Teil der Java-Umgebung sein. Unter der Annahme, dass `jdk1.3` das Hauptverzeichnis der Java-Installation sei, müssen diese Dateien ins Verzeichnis `jdk1.3\jre\lib\ext` kopiert werden. Der Java-Erweiterungsmechanismus sieht vor, dass diese Dateien automatisch erkannt werden, was jedoch in der getesteten Version 1.3 von Java noch nicht funktioniert. Es ist daher notwendig, diese Dateien explizit in die Umgebungsvariable `CLASSPATH` aufzunehmen. Siehe dazu in der Dokumentation für das jeweilige Betriebssystem unter *Setting the classpath*. Unter Windows NT muss zu diesem Zweck unter `Start/Systemsteuerung/System/Umgebung` die Benutzervariable (persönliche Installation) oder Systemvariable (allgemeine Installation) `CLASSPATH` erzeugt oder verändert werden, so dass sie die Pfade `.;c:\jdk1.3\jre\lib\ext\jh.jar;c:\jdk1.3\jre\lib\ext\crimson.jar`

usw. für genannte Dateien enthält.

Das Programm X-mas kann danach unter Windows mit der Eingabe `java -jar Xmas.jar` mit der Eingabeaufforderung (DOS-Fenster) gestartet werden. Zur weiteren Bedienung des Programms steht die Online-Hilfe im Menü *Help* zur Verfügung.

7.3 Kompilierung und Archivierung

Da Java eine plattformunabhängige Programmiersprache ist, wird es bei einer Portierung nicht erforderlich, den Quelltext neu zu kompilieren, solange man keine Änderungen an ihm vorgenommen hat. Stattdessen genügt es, den Verzeichnisbaum mit den kompilierten Dateien zu kopieren und das Zielverzeichnis in den Klassenpfad `CLASSPATH` einzubinden. Neben den Klassendateien mit der Endung `.class` werden die html- und xml-Dateien des Hilfesystems benötigt sowie Bilddateien für Icons und Dateien mit Daten zur Röntgenabsorption (`xcom.dat`), zu den Absorptionskanten (`absegeenergy.dat`), zur Atomstreuung (`atom_wk.dat`) und zur Comptonstreuung (`compton_by.dat`). Wurden Änderungen am Quelltext vorgenommen, so sollten alle Klassendateien gelöscht werden, z. B. unter Windows mit der Batch-Datei `delclass.bat`. Bei *einzelnen* Änderungen genügt das Löschen der betroffenen Klassendateien. Anschließend wird der Übersetzer für das Hauptprogramm aufgerufen, was unter Windows mit der Syntax `javac Xmas.java` geschieht.

In Java steht ein spezielles Archivformat (Java Archive `JAR`) zur Verfügung, mit dem alle zu einem Programm gehörigen Dateien (`.class`-Dateien, Daten, Bilder, Hilfetexte, usw.) in eine einzige ausführbare Datei gepackt werden können. Für das Betriebssystem Windows wurde die Batch-Datei `makexmas.bat` geschrieben, welche die Kompilierung und Archivierung der über 450 Dateien der Anwendung automatisiert. Sie nutzt zugleich den Erweiterungsmechanismus von Java aus, welcher es ermöglicht, `jar`-Dateien in das Verzeichnis `jre\bin\ext` der Java-Installation zu platzieren, um zukünftig auf die enthaltenen Klassen in gleicher Weise wie auf Java-Kernklassen zugreifen zu können.

Kapitel 8

Zusammenfassung

Das in dieser Arbeit vorgestellte Programm unterstützt sämtliche Auswertungsschritte, die zur Auswertung von Röntgendiffraktogrammen ungeordneter Systeme benötigt werden, angefangen vom Import gängiger Datenformate über Korrekturverfahren zu Absorption und Polarisation, Normierung und Fouriertransformation bis zur Gewinnung von Strukturinformationen. Röntgenspezifische, stöchiometrische und mathematische Funktionen unterstützen die Vorbereitung der Messungen und begleiten den Auswertungsgang. Neben der eigentlichen Auswertung wurden zahlreiche numerische Verfahren zur Aufbereitung von Datensätzen sowie zur Dateninterpretation durch Optimierung integriert. Die Diagramme können farbig und hochauflösend ausgedruckt werden oder im gängigen EPS-Format exportiert werden, um in andere Anwendungen wie Textverarbeitungssysteme integriert zu werden.

Durch die Verwendung verbesserter Algorithmen und neuerer Röntgendaten werden gegenüber früheren Programmen genauere Auswertungen möglich. An die Stelle einer Vielzahl kleinerer Programme, die oft nur auf bestimmten Betriebssystemen ausführbar sind, tritt ein betriebssystemunabhängiges integriertes Programmpaket, dessen Benutzeroberfläche einen interaktiven Umgang mit den Daten ermöglicht. Die gleichzeitige und vergleichende Bearbeitung mehrerer Datensätze ist problemlos möglich. Anwender des Programms können dessen Bedienung durch die intuitiv gestaltete Oberfläche leicht erlernen und werden bei Bedarf in ihrer Arbeit durch ein Online-Hilfesystem unterstützt. Wurden für ältere Programme manuell zu konfigurierende Steuerdateien benötigt, die mit der Generierung redundanter Informationen einhergingen und anfällig für Übertragungsfehler waren, so verwendet das in dieser Arbeit vorgestellte Programm eine Registratur, deren internes Datenformat die neue universelle Datensprache XML unterstützt. Mit Hilfe der Registratur können die Daten zentral eingegeben, kontrolliert und abgespeichert werden.

Durch zahlreiche Plausiblititätskontrollen und ausgiebigen Gebrauch von Fehlerbehandlungsmechanismen entstand ein stabil laufendes Programm. Unter Einsatz von aktuellen Konzepten zur Objektorientierung und Entwurfsmustern wurde ein Programmpaket erzeugt, das flexibel eingesetzt und erweitert werden kann. Das Programm wurde auch im Hinblick auf die Erweiterbarkeit durch diese Arbeit und die Verwendung von javadoc-Kommentaren ausführlich dokumentiert.

Insgesamt dient das in dieser Arbeit vorgestellte Programm dazu, die Auswertung von

Röntgendiffraktogrammen wesentlich zu erleichtern und zu verbessern. Durch Anwendung fortgeschrittener Programmiertechniken wurde ein Programm entwickelt, mit dem bestehende Problemstellungen bearbeitet werden können und das zugleich eine solide Grundlage für kommende Erweiterungen bietet.

Kapitel 9

Abstract: Development of a graphical user interface for the evaluation of x-ray diffractograms of disordered systems

9.1 Introduction

In course of his work on cathode ray tubes in 1895 Wilhelm Conrad Röntgen discovered the generation of a new radiation, which was emitted from thin metal foils that were irradiated by electrons accelerated by high voltage. Röntgen recognized that this x-radiation was able to penetrate matter almost unhinderedly. Laue, Friedrich and Knipping exposed crystals of copper sulfate to x-rays and found regular point patterns on photographic plates coming from the radiation outgoing from the sample. They formulated the correlation between the crystal structure and its interference pattern. Debye extended this method for structure determination to the examination of disordered systems such as liquids and glasses. Theory and practice of this method were described in numerous articles. At the present stage, x-ray diffraction is an important method for the research on condensed matter and is used intensely in physical chemistry and applied sciences.

This work presents a graphical user interface (GUI), shown in figure 9.1, which can be used for the evaluation of x-ray diffractograms of disordered systems. This application was written in the new programming language Java, which was developped by Sun Microsystems in a group around Bill Joy and James Gosling, who published the Java Language Specification in 1996. Java is often characterized by the following catchwords: simple, object-oriented, distributed, interpreted, robust, secure, architecture-neutral, portable, high-performant, multithreaded, dynamic and secure. Two of these properties were especially important for the decision to use this language: architecture neutrality, which allows users to run compiled Java programs on various operating systems and the extensive support of object-oriented techniques, which are best suited for the development of large and complex applications. Java programs are running on a Java platform, which consists of the so-called Java Virtual Machine (JVM) and the large Java Application Programming Interface (Java-API).

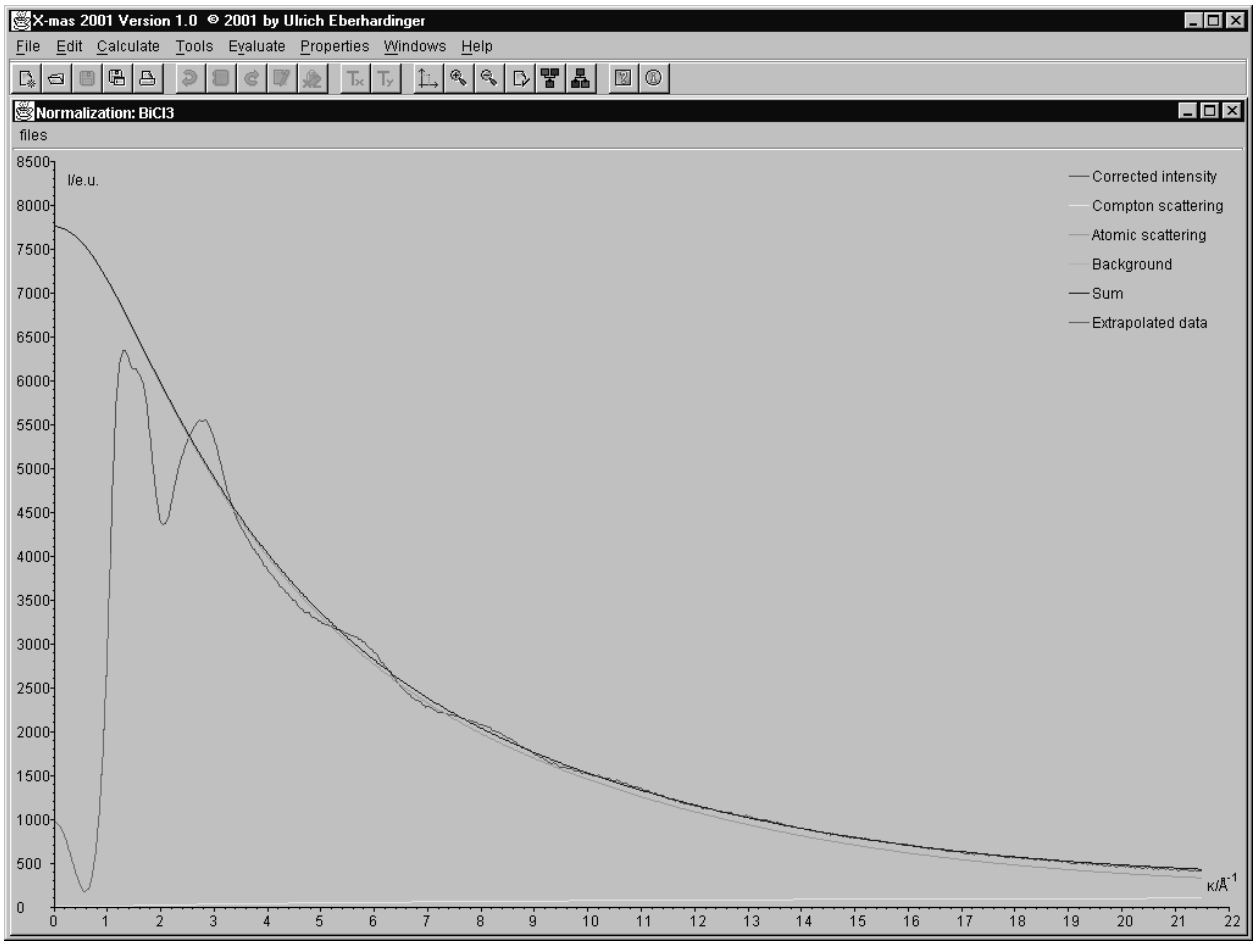


Figure 9.1: Graphical user interface

The JVM comprises an interpreter, a just-in-time compiler and the runtime environment. The API is a program library, which offers solutions for many common problems as data structures, GUI-elements, numerical methods, event-handling, exception-handling and so on.

Along with the use of Java, object-oriented techniques were introduced to extend the methodology used in the present working group. While former programs were written in procedural languages as Fortran 77 and C, object oriented concepts allowed the development of modules for many specialised tasks which were integrated into one comprehensive application with a user-friendly graphical front-end. The designation of object-oriented analysis and design is to map a complex system to a system of classes and objects. Objects are concrete instances of classes, which describe the state and its possible changes of a system and are coded by the use of a program language such as Java or C++.

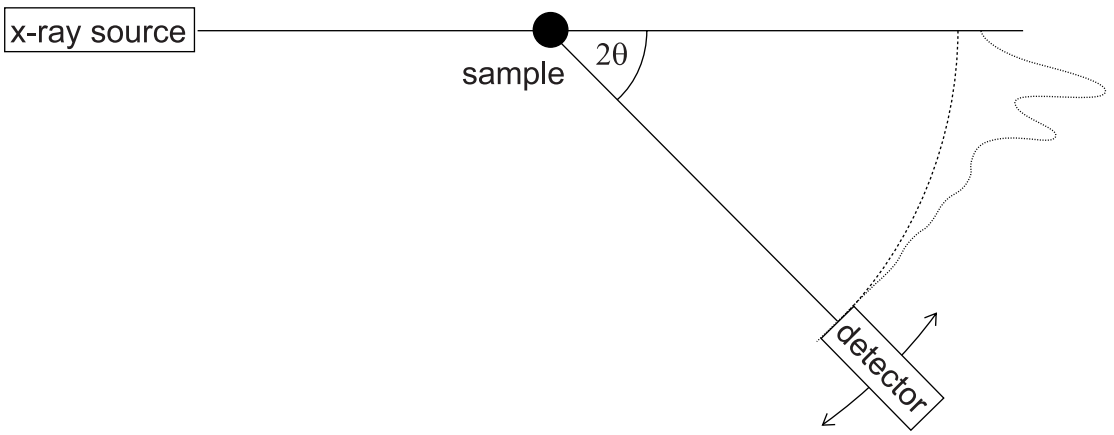


Figure 9.2: sketch of an x-ray diffraction experiment

9.2 Evaluation of x-ray diffractograms

Figure 9.2 shows a sketch of an angular dispersive x-ray diffraction experiment: An x-ray emitted by an x-ray source like an x-ray tube or a synchrotron is monochromatized[1] and subsequently diffracted by the electrons of a sample. The intensity (shown as a dotted line) of the diffracted radiation is measured by a detector as a function of the diffraction angle 2θ. In the evaluation process the electronic structure of the examined system is calculated from the measured intensity. The intra-atomic electronic structure, which is known from quantum mechanical calculations, results in its correlated atomic scattering and can be separated from the total scattering to obtain the atomic structure. The relation of the scattered intensity I and the total correlation function $G(r)$, which describes the atomic structure, is given by

$$G\left(r\right) = \frac{\sum\limits_{i,j} n_i n_j \bar{z}_i \bar{z}_j g_{ij}\left(r\right)}{\left(\sum\limits_i n_i Z_i\right)^2} = 1 + \frac{1}{2\pi^2 r \bar{\rho} \left(\sum\limits_i n_i Z_i\right)^2} \int\limits_0^\infty \kappa \cdot \underbrace{\frac{I\left(\kappa\right) - I_{atom}\left(\kappa\right)}{f_e^2\left(\kappa\right)}}_{I_{red}} \sin\left(\kappa r\right) d\kappa \quad (9.1)$$

with

r	distance	κ	length of scattering vector
n	amount	x	mole fraction
$g_{i,j}(r)$	atomic pair correlation function	I_{atom}	atomic scattering
I	corrected measured intensity	I_{red}	reduced intensity
$\bar{\rho}$	number density	f_e	average atomic form factor
$\bar{z}_i\left(\kappa\right)$	average effective electron number	Z_i	proton count

The radiation is scattered as well elastically as inelastically whereas the latter is called Compton scattering. The scattering of radiation is accompanied by a polarization of the beam. Finally, the absorption of the x-rays by the sample and where appropriate by a

[1] the monochromator is omitted in the figure for simplicity

surrounding capillary has to be taken into account.

A typical evaluation will be shown in the following for the sample and measurement conditions described in this table:

Measurement conditions	
sample s	$BiCl_3$ 99.999% (Aldrich)
density (s)	3.8 g/cm^3
angular range	0.7° bis 30°
angular resolution	0.05°
beam source	ESRF synchrotron, beamline ID15A
beam width	0.3 mm
radiation energy	80 keV
resolution monochromator	50 eV (FWHM)
capillary c	NMR-capillary 322-pp-8" (Wilmad)
substance (c)	boro silicate glass Pyrex 7740 (Spintec)
composition (c)	80.5% SiO_2, 12.9% B_2O_3, 3.8% Na_2O 2.2% Al_2O_3, 1.2% Li_2O, 0.4% K2O
density (c)	2.23 g/cm^3
inner diameter (c)	2.413 mm +0.013 mm/-0 mm
outer diameter (c)	3.000 mm +0 mm/-0.013 mm
temperature	270 °C

This measurement has been carried out by G. Heusel and is described in his PhD-thesis. The focus of this section is to show the possibilities of the program with regard to the evaluation. It refers to the German-language section 4.2, where more detailed information is given about used formulae, program structures, diagrams and screen shots of the used dialogs.

Figure 9.3 shows the diffractograms of the cell filled with $BiCl_3$ and the empty cell after subtracting the air scattering.
Figure 9.4 shows the geometry for the absorption of the beam by a cylindrical capillary c filled with a sample s. Kendig and Pings have derived a formalism for correcting the measured beam intensity with respect to absorption. This formalism is described in detail in section 5.1 together with the corresponding parts of the program. Figure 4.11 shows a dialog, which allows to enter parameters for this correction. Another dialog and corresponding numerical methods have been provided for planar reflexion geometry according to Debye and Menke.

The correction for absorption is followed by a correction for polarization. Formulae for several x-ray sources have been used: The formulae of Azaroff, Kerr et al. for x-ray tubes with and without use of a monochromator and a formula of Klein and Nishina for linearly polarized synchrotron radiation. Figure 9.5 shows the diffractograms of $BiCl_3$ after the corrections for absorption and polarization. In this case, the Klein-Nishina formula was

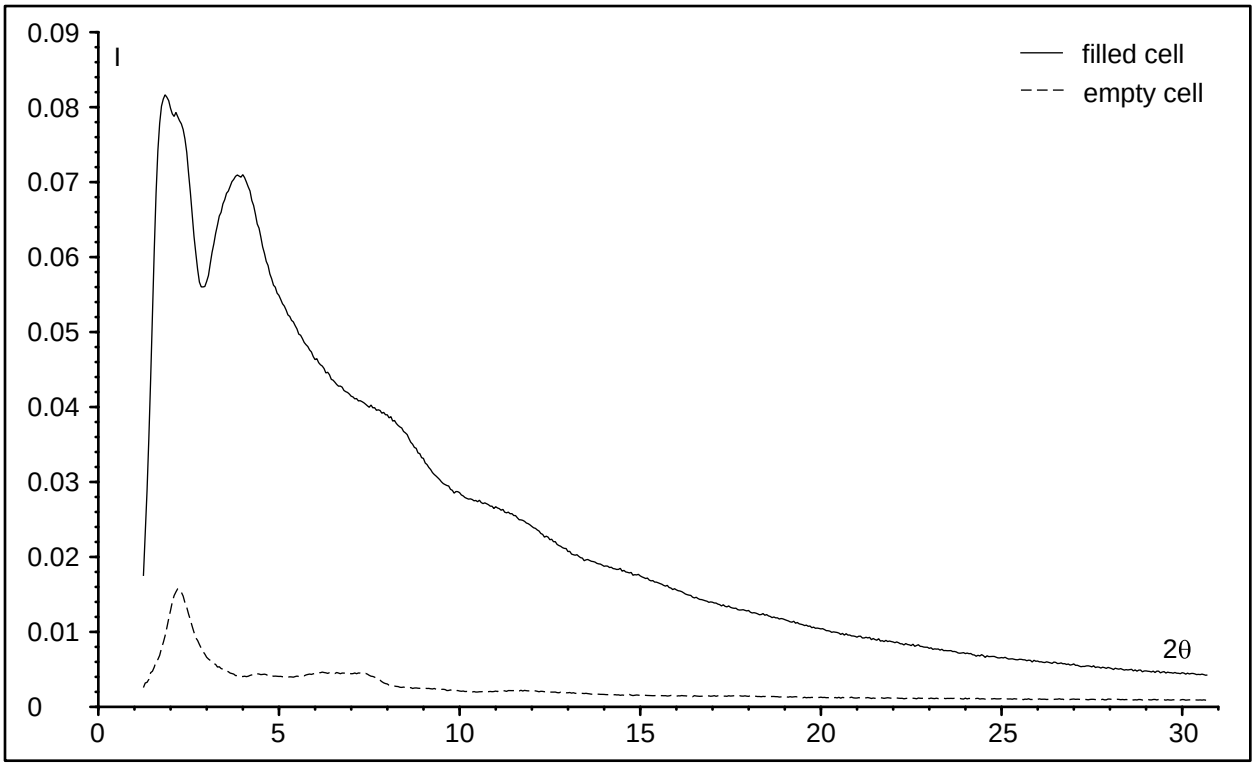


Figure 9.3: diffractograms of the cell filled with $BiCl_3$ and the empty cell after subtracting the air scattering

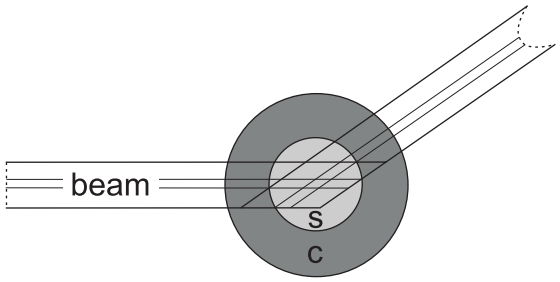


Figure 9.4: absorption of the x-ray beam

applied for the case of synchrotron radiation.

After these corrections, the scale of the abscissa has to be converted from angular units to units of the scattering vector, which are used for the subsequent Fourier transform. Because equidistant abscissa values are needed, this operation is combined with an interpolation. For a quantitative interpretation of the total correlation function, which is obtained by the Fourier transform, the ordinate values must be normalized from arbitrary units (photon counts) to an absolute scale (electron units, e. u.). This normalization is best performed by a procedure proposed by Krogh-Moe. It needs the atomic scattering and the Compton scattering of the observed system. Their values can be calculated from tabulated parameters

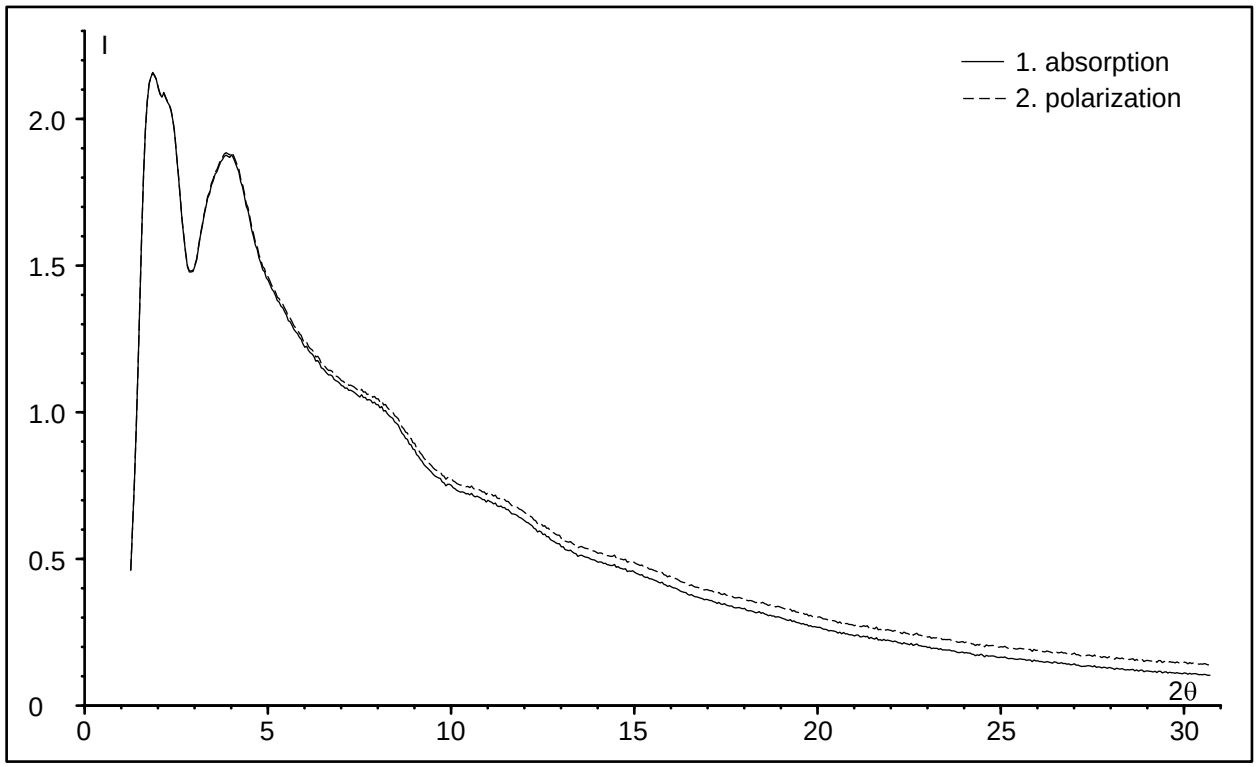


Figure 9.5: diffractograms of $BiCl_3$ after corrections for absorption and polarization

of quantum mechanical calculations. The program presented in this work uses parameterizations of Waasmeier and Kirfel of data from Doyle/Turner/Coulthard/Cromer/Waber/Mann which are tabulated in the International Tables of Crystallography for the atomic scattering and parameterizations of Balyuzi for the data of Cromer and Mann for the Compton scattering. For the calculation of the integrals in the Krogh-Moe formula 4.22, data are needed beginning from $\kappa = 0$. These data are difficultly to be obtained from experiments and can be calculated from the isothermal compressibility and an extrapolation suggested by Weidner, Geisenfelder and Zimmermann.

Figure 9.6 shows the normalized intensity extrapolated to $\kappa = 0$ as a function of κ together with the atomic scattering A and the Compton scattering C of $BiCl_3$. The normalized intensity converges to the sum $A + C$ as expected. A dialog (compare figure 4.22) allows extensive control of the parameters used for this normalization.

The reduced intensity calculated according to formula 9.1 is now Fourier transformed to the total correlation function $G(r)$. This function is shown in figure 9.7. Gauss functions can be used to model $G(r)$. A Levenberg-Marquardt routine allows to fit Gauss functions to $G(r)$ by optimizing parameters given by the user or estimated by the program. If a maximum can be assigned to a correlation between two defined elements, their distance and coordination number can be calculated. The assignment of the first maximum in figure 9.7 to a correlation

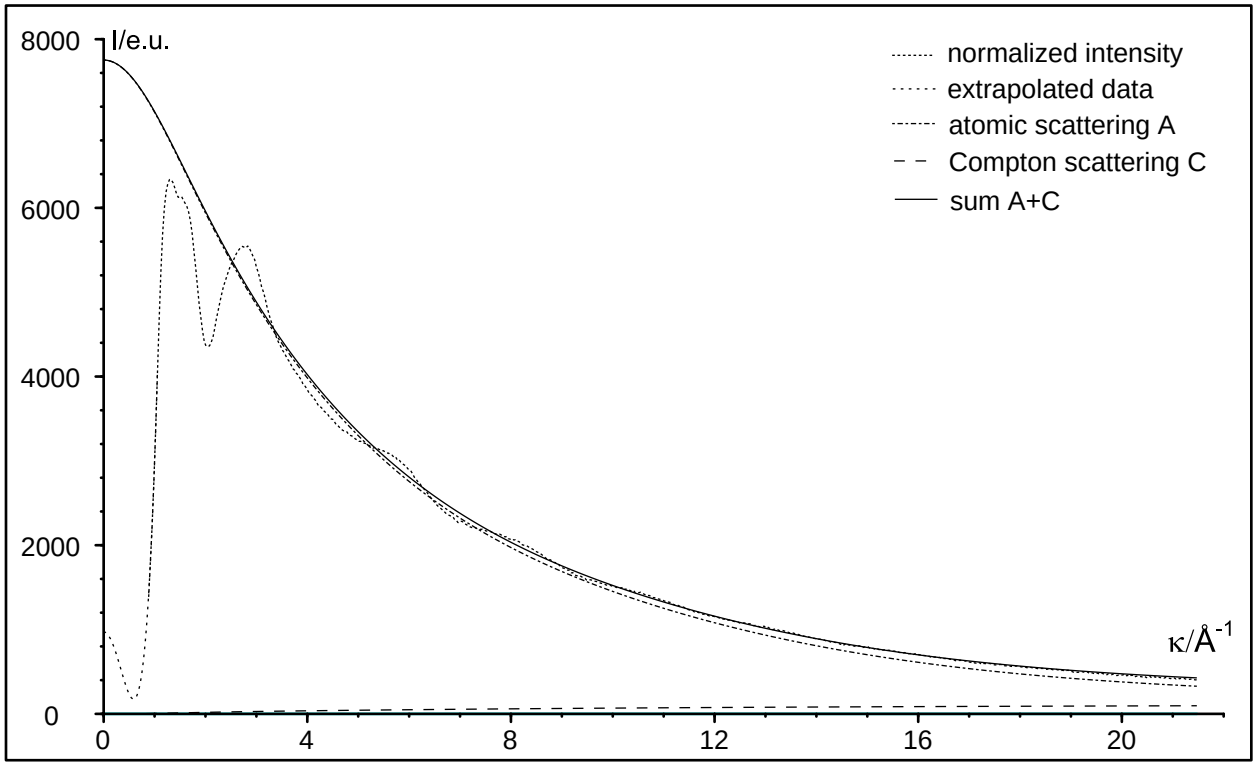


Figure 9.6: Normalization of the corrected diffractogram of $BiCl_3$

between Bi and Cl resulted in a distance of 2.5 Å and a coordination number of 3.1. These results are in good agreement with published values:

literature	state	method	distance	cn
Lemke	melt 240 °C	EDXD	2.70 Å	5.8
Fukushima/Suzuki	melt 270 °C	ND	2.50 Å	3.0
Price et al.	melt 300 °C	ND	2.5 Å	3.0
Nyburg et al.	crystal (orh)	ADXD	2.46 Å/2.51 Å	1.0/2.0
Bartl	crystal (orh)	ND	2.50 Å	3.0
Heusel/Eberhardinger	melt 270 °C	ADXD	2.5 Å	3.1

Abbreviations: EDXD: energy dispersive x-ray diffraction, ADXD: angular dispersive x-ray diffraction, ND: neutron diffraction, cn: coordination number, orh: orthorhombic

Additional to these operations, which are part of the evaluation process, different calculations can be performed separately: Calculation of weighting factors, atomic and Compton scattering, stoichiometrical calculations, calculation of absorption factors and absorption spectra. A dialog was provided to estimate the quality of EXAFS-spectra in advance.

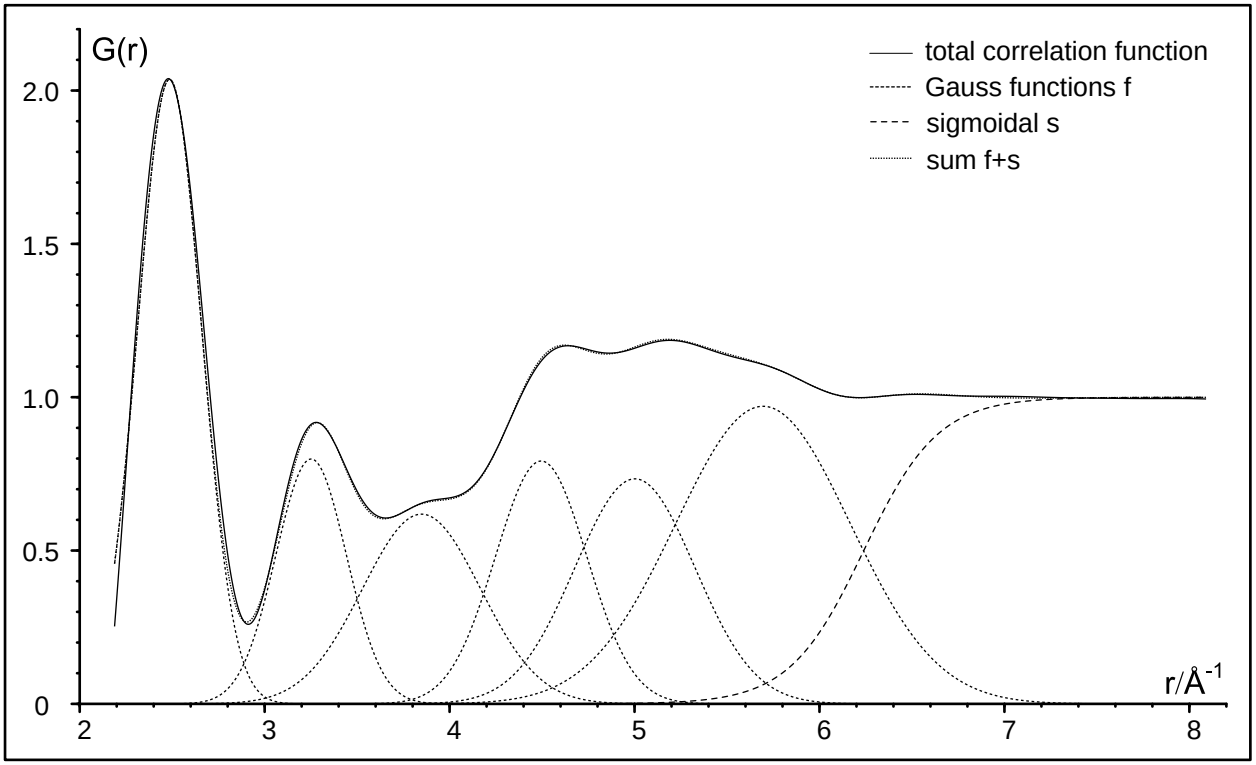


Figure 9.7: Fit of Gauss functions to data

9.3 Program

All parts of the program were newly developped by taking advantage in the use of the extensive Java-API. The fundamental principle of this program follows the model-view-controller architecture, which was introduced by Krasner and Pope and found broad application in the design of GUIs. The model unit corresponds to the class `Data`, which contains the measured or processed data. The graphical representation (view) of data is implemented by the class `DiagramPanel` and its specialized subclasses. Objects of this class are contained in the class `Diagram`, which allows control over representation and change of data. Diagrams can be loaded, saved, printed and exported in ASCII- or EPS-format. Data are changed by transformations, which are represented by the classes `Transformation`, `Transformer` and `ParameterSet`. This encapsulation permits undo/redo operations. The class `Registry` contains parameters which are used for the evaluation and build up a tree structure, which can be serialized. Interpreters were developped to load data and parameters, which are stored in files with XML-structure. Other interpreters permit the input and evaluation of chemical formulae and mathematical expressions.

Many menus and dialogs were developped for the interaction between user and program. Most of the dialogs were derived from the class `StandardDialog`, which supports graphic representation and functionality, that is common to many dialogs. Some classes were written to enable and disable components of menus and dialogs. Based on the `JavaHelp`-API, a help system was developped, which offers general and context-sensitive help to the user.

Many algorithms were integrated to enhance the precision of mathematical calculations. Fast fourier transform algorithms were included for complex and real data and the sine transform of real data, which is needed for the transformation of diffractograms. The class `FunctionNode` allows the calculation of symbolic derivations of mathematical terms. These derivations are used for the nonlinear optimization of arbitrary parametrized functions according to the procedure of Levenberg and Marquardt. The generalized least-squares method is used for the optimization of linear function sets. Savitzky-Golay filtering is used for the smoothing of data. These operations are part of other operations or are accessible via menu items and dialogs by themselves.

Data can furthermore be manipulated by the application of user-defined functions or the built-in editor, which allows editing and transformation under simultaneous visual control.

Multiple data records can be combined in various ways: They can be represented together in diagrams, their values can be accumulated or combined by mathematical transformations, e. g. by calculating average values.

9.4 Summary

The program presented in this work supports all steps of evaluation, which are needed for the analysis of x-ray diffractograms of disordered systems, starting with the import of common data formats via correction procedures for absorption and polarization, normalization and fourier transform up to the extraction of structure information. X-ray-specific, stoichiometric and mathematical functions support the preparation of measurements and accompany the evaluation progress. Additionally to operations which are part of the evaluation, many numeric procedures were included to process and interpret data. The diagrams can be printed out colored and in high resolution or be exported in the common EPS format, which is often used by other applications such as text processing systems.

By the use of improved algorithms and recent data for atomic form factors, Compton scattering and absorption factors, data analysis becomes more exact compared to former programs. This new application replaces numerous small programs which run on different and particular operating systems. Its GUI enables the user to process data interactively. Multiple records can be handled simultaneously and comparatively. Due to its intuitive user interface, this program is easy to be learned and users are supported in their work by extensive on-line help. Former programs were controlled by associated files which had to be configured manually. That procedure comes along with production of redundant information and is susceptible to transcription errors. This program uses a registry instead which allows central input, control and storage. This registry makes use of the new universal data format XML.

Extensive use was made of validity checks and exception handling to build a stably running program. Current concepts of object-oriented analysis and design patterns were

used to create a program package, which can be employed flexibly and be extended simply. With regard to usability and extensibility, the program was documented amply by this work and the usage of javadoc comments.

Altogether the program presented in this work suits the purpose to facilitate and improve the analysis of x-ray diffractograms substantially. By the use of advanced programming techniques, an application was developped, which allows the coverage of existent problems and provides a sustainable fundament for future extensions.

Kapitel 10

Anhang

10.1 Die Erzeugung von Postskriptdateien mit der Klasse `PSGraphics`

Diese Klasse leitet sich von der Klasse `Graphics2D` ab und dient der Erzeugung von Postscript-Code (PS) bzw. Encapsulated Postscript-Code (EPS) durch Überschreibung der grafischen Methoden der abstrakten Klassen `Graphics` und `Graphics2D` im Paket `java.awt`. Diese Klassen werden von Java in allen Methoden verwendet, in denen eine Ausgabe auf ein graphisches Gerät (graphical device) wie einen Bildschirm oder einen Drucker erfolgen soll. Die praktische Bedeutung der Klasse `PSGraphics` liegt in der Möglichkeit, mit in Java üblichen Methoden betriebssystemunabhängigen Quelltext für Vektorgraphiken zu erzeugen. Das (gekapselte) Postskriptformat wird von vielen Anwendungen unterstützt, in welchen entsprechende Dateien ausgedruckt, angezeigt oder in andere Dokumente, insbesondere LaTeX-Dateien, eingebunden werden können. So wurden beispielsweise die in dieser Arbeit abgebildeten Diagramme mit Hilfe dieser Klasse im Programm X-mas erzeugt und ggf. mit dem Programm Adobe Illustrator nachbearbeitet.

Sowohl die im Postskript Language Reference Manual [78] dokumentierte Sprache Postscript als auch die im Java2D Programming Guide [79] dokumentierten Java-Klassen beschreiben graphische Kontexte, in welche Vektorgraphiken und Bitmap-Graphiken eingebettet sind. Den jeweiligen Beschreibungen der Vektorgraphik in beiden Sprachen liegt ein gemeinsames Modell zugrunde, die Syntax der Sprachen unterscheidet sich jedoch erheblich. Postscript ist eine von der Firma Adobe entwickelte Skriptsprache, die durch Stackoperationen beschrieben wird.

Die vorliegende Implementierung der Klasse `PSGraphics` umgeht Berechnungen in Postscript weitgehend und führt diese in Java durch, so dass der erzeugte Code überwiegend konstante Größen und Operationen enthält. Dadurch ist es nicht möglich, implementierungsabhängige Parameter wie Eigenschaften von Schriftarten abzufragen, wofür in den genannten Superklassen jedoch auch keine Methoden deklariert sind, so dass sie von anderen Klassen des Java-API nicht genutzt werden können. Die häufigste Verwendung der `Graphics`-Klassen tritt in den Methoden `paint(Graphics g)` und `print(Graphics g)` auf.

`Graphics` und ihre Subklasse `Graphics2D` sind abstrakte Klassen, welche Methoden definieren, die einen Zugriff auf den graphischen Kontext erlauben und von `PSGraphics` implementiert werden müssen. Einige dieser Methoden werden weiter unten beschrieben. Die Methoden zum Zeichnen graphischer Objekte erzeugen unmittelbar Zeichenketten mit Postscript-Operationen, welche in einen Vektor eingetragen werden.

Der Zustand eines graphischen Kontextes wird in beiden Sprachen durch mehrere Parameter charakterisiert, welche in `PSGraphics` als Instanzvariablen und unter Postscript im Lexikon (dictionary) `graphics state` abgelegt sind. Zu Beginn des in der Methode `getPSCode()` erzeugten Postscript-Codes wird der aktuelle Zustand mit `gsave` abgespeichert und zum Schluss mit `grestore` wieder hergestellt. `PSGraphics` definiert folgende Parameter:

Transformation

Die in Zeichenbefehlen verwendeten Koordinaten beziehen sich auf einen Benutzerraum (user space). Bei der Ausgabe müssen sie in Gerätekoordinaten (device space) umgerechnet werden. Diese Umrechnung erfolgt über eine affine Transformation. Darüber hinaus erfordern manche graphische Elemente die Anwendung affiner Transformationen, beispielsweise können unter Postscript keine Ellipsen dargestellt werden und in beiden Sprachen keine Rechtecke gezeichnet werden, deren Kanten gegen die Koordinatenachsen verdreht sind. Stattdessen muss die Zeichenebene skaliert bzw. gedreht werden. Die entsprechenden Transformationen werden mit einer Grundtransformation verknüpft, welche auf alle Zeichenbefehle angewandt wird. Die affinen Transformationen der Ebene lassen sich durch eine 3x3-Matrix dargestellen, welche die Produktbildung zur Verkettung von Transformationen ermöglicht:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} m_{00}x + m_{01}y + m_{02} \\ m_{10}x + m_{11}y + m_{12} \\ 1 \end{pmatrix}$$

Die Koordinatensysteme im Benutzerraum von Java und Postscript unterscheiden sich in der Lage des Ursprungs, welcher in Java — wie für Bildschirmkoordinaten üblich — links oben liegt, d. h. die positive y-Achse zeigt nach unten. Bei Postscript liegt der Ursprung links unten und die positive y-Achse zeigt nach oben.

Um dies zu berücksichtigen, wird zu Beginn des Postscript-Codes eine Basistransformation festgelegt, welche sich aus der aktuellen Postscripttransformation (current transformation matrix, CTM, herrührend beispielsweise von einem Dokument, in welches die zu erzeugende Graphik eingebettet ist) durch Verknüpfung mit einer Transformation ergibt, welche das Java-Koordinatensystem in das Postscript-Koordinatensystem umrechnet. Zu dieser Umrechnung benötigt das Programm Angaben zur Höhe des darzustellenden Bereiches. Der darzustellende Bereich ist in der Instanzvariablen `Rectangle2D.Float boundingBox` gekapselt. Die Bounding-Box dient im EPS-Format als Information über Größe und Lage einer in eine Seite einzubindenden Graphik. Die Basistransformation wird in der PS-Variablen `AFFIN` abgelegt und kann bei Bedarf mit dem Postscript-Befehl `AFFIN setmatrix` in der Methode `addBasicTransform()` wieder hergestellt werden. Durch Festlegung dieser Grundtransformation kehrt sich jedoch gleichzeitig der Drehsinn um, was bei der Ausgabe von Schrift und

Angabe von Winkeln von Bedeutung ist. Vor und nach der Ausgabe von Schrift wird daher stets die y-Achse mit -1 skaliert. Auch in Methoden, die mit Winkeln arbeiten, musste diese Umkehr des Drehsinns berücksichtigt werden.

Die auf der Basistransformation aufbauenden Transformationen können mit verschiedenen Methoden verändert werden, die in `Graphics2D` definiert und in dieser Klasse implementiert wurden. Der zugehörige Postscript-Code wird durch weiteren Aufruf der privaten Methoden `addTranslateCode(float x, float y)`, `addRotateCode(float theta)`, `addScaleCode(float sx, float sy)` und `addTransformCode(float m00, float m10, float m01, float m11, float m02, float m12)` dem Code-Vektor hinzugefügt, der alle Postskriptanweisungen enthält.

Schriftart

In Java existieren mehrere Klassifikationen der Schriftart (Klasse `Font`), von denen in dieser Klasse die logische Schriftartbezeichnung (logical font name) verwendet wird. Für jedes postscriptfähige System sind folgende Schriftarten definiert, die mit Java-Schriftarten korrespondieren:

Postscript	Java
Times-Roman	Serif
Times-Bold	Serif, Bold
Times-Italic	Serif, Italic
Times-BoldItalic	Serif, Bold, Italic
Helvetica	SansSerif
Helvetica-Bold	SansSerif, Bold
Helvetica-Oblique	SansSerif, Italic
Helvetica-BoldOblique	SansSerif, Bold, Italic
Courier	Monospaced
Courier-Bold	Monospaced, Bold
Courier-Oblique	Monospaced, Italic
Courier-BoldOblique	Monospaced, Bold, Italic
Symbol	Symbol

Zur vereinfachten Verwendung dieser Schriftarten wurden statische Konstanten erzeugt, die mit `setFont(int fontConst, int size)` der Festlegung der aktuellen Schriftart dienen können. Da Java nicht direkt auf die Eigenschaften der Schriftarten des Druckers zugreifen kann, werden stattdessen in guter Näherung die Eigenschaften der entsprechenden Java-Schriftarten verwendet. Die Methode `addFontCode(Font font)` erzeugt den zu einer Schriftart gehörigen Postscript-Code. Wird keine Schriftart festgelegt, so wird als Vorgabe die Schriftart Times-Roman in der Größe 10 verwendet.

Füllstil

Der Füllstil wird in Java durch das Interface `Paint` festgelegt, welches von den Klassen `Color` (Farbe), `GradientPaint` (axiale lineare Farbgradienten) und `TexturePaint` (periodi-

sche Bitmap-Füllmuster) implementiert wird. Die derzeitige Implementierung von `PSGraphics` berücksichtigt nur die Verwendung der Klasse `Color`. Zur Farbkodierung verwendet diese Klasse das RGB-Modell. Die voreingestellte Farbe für den Vordergrund ist schwarz und für den Hintergrund weiß. Die Methode `addColorCode()` erzeugt den erforderlichen Postscript-Code.

Linienstil

Der Linienstil in Java wird durch das Interface `Stroke` bzw. eine Implementierung desselben in der Klasse `BasicStroke` festgelegt, welche in der Klasse `PSGraphics` ausschließlich verwendet wird. Sie enthält Attribute wie Linienbreite, Strichelungsmuster, Gehrungsfuge und -winkel, Verknüpfung von Liniensegmenten und Form von Linienenden. Sämtliche dieser Attribute finden auch unter Postscript Verwendung. Eine Zuordung zu den entsprechenden Kodierungen erfolgt in der Methode `addStrokeCode()`.

Darstellbarer Bereich (Clip)

Mit Hilfe eines Clips lässt sich der sichtbare Bereich einer Graphik beschränken. Dazu existiert einerseits in Java die Klasse `Clip` und andererseits ein entsprechender Befehl unter Postscript. In beiden Fällen ist der Clip als Pfad implementiert. Der Postscript-Code wird mit `addClipCode()` erzeugt, der Code des zugehörigen Pfades mit `addIteratorCode()`. Der voreingestellte Wert für den Clip entspricht der Bounding-Box.

Graphische Befehle

Zum eigentlichen Zeichen graphischer Elemente definiert Java eine Vielzahl von Zeichenbefehlen, für welche es unter Postscript Entsprechungen gibt. Wesentliche Zeichenelemente sind Zeichenketten, Strecken, Kreisbögen und Bézierkurven. Sämtliche geometrischen Objekte werden in Java unter dem Interface `Shape` zusammengefasst. Dazu gehören auch zusammengesetzte Objekte wie Polygone oder allgemeine offene und geschlossene, zusammenhängende und unzusammenhängende, gefüllte oder als Umrisse dargestellte Objekte, die aus genannten elementaren Objekten zusammengesetzt sind. Die Methode `addIteratorCode()` iteriert über die Elemente zusammengesetzter Objekte und erzeugt für diese den jeweiligen Postscriptcode. Für die einzelnen Elemente existieren Methoden wie `addMoveTo(...)` zur Bewegung eines virtuellen Stifts ohne zu zeichnen, `addLineTo(...)` zum Zeichnen einer Linie, `addQuadTo(...)` zum Hinzufügen eines Kurvensegmentes mit rechtwinkliger Begrenzung, `addCurveTo(...)` zum anhängen einer Bézier-Kurve und `addClosePath()` zum Schließen eines Pfades.

Über die Methoden `String getPSCode()` bzw. `String getEPSCode()` erhält man den Quelltext in der gewünschten Sprache, welcher z. B. in eine Datei ausgegeben werden kann. Solche Dateien können von anderen Programmen wie Adobe Illustrator, Corel Draw oder Latex-Kompilern verarbeitet und ausgedruckt werden.

10.2 Das Paket pointer

Wie schon im Kapitel Grundlagen erwähnt wurde, verwendet Java aus Sicherheitsgründen keine Zeiger sondern stattdessen das damit verwandte Konzept der Referenzen, mit dem alle Objekte und Arrays behandelt werden können. Daneben gibt es in Java primitive Datentypen (`boolean, char, int, double, ...`), die bei der Parameterübergabe nicht als Referenz, sondern als Wert übergeben werden. Dies hat zur Folge, dass eine Änderung eines Wertes in einer aufgerufenen Methode keine Auswirkung auf den entsprechenden Wert in der aufrufenden Methode hat. In C/C++-Programmen verwendet man in Fällen, in denen eine Änderung auch in der aufrufenden Methode erwünscht ist, Zeiger. In Java kapselt man stattdessen mehrere veränderliche Werte in Klassen. Die Werte dort werden durch geeignete Methoden verändert.

10.2.1 Die Klassen BooleanPointer, IntPointer und DoublePointer

Da diese Klassen weitgehende Analogien aufweisen, sollen sie hier gemeinsam beschrieben werden. Um numerische Routinen wie z. B. in Numerical Recipes [69] ohne große Veränderungen nach Java portieren zu können, wurden in diesem Paket spezielle Kapselklassen (Wrapper) entwickelt. Es sei erwähnt, dass auch das Java-API die Kapselklassen `Boolean`, `Character`, `Integer`, `Double` usw. bereitstellt, es handelt sich jedoch um unveränderliche Klassen, d. h. es existieren keine Methoden, um den Wert der entsprechenden Objekte nach deren Initialisierung zu verändern.

Sämtliche dieser Klassen enthalten eine Variable mit dem zu kapselnden einfachen Datentyp. Mit Konstruktoren wie beispielsweise `IntPointer(int i)` und `IntPointer(String str_i)` werden die Objekte angelegt, mit Methoden wie `void set(int i)` bzw. `void set(String str_i)` können sie verändert und mit `int intValue()` und `String toString()` wieder ausgelesen werden. Darüber hinaus wurden einige mathematische Grundmethoden wie `add(int j)`, `add(IntegerPointer ip)` oder spezifische Methoden wie die Inkrementierung durch `void IntegerPointer.inc()` implementiert.

10.2.2 Die Klassen IntArray und DoubleArray

Im Gegensatz zu den einfachen Datentypen werden Arrays als Referenz übergeben; ihre Elemente können in der aufgerufenen Methode in gewohnter Weise verändert werden. Hingegen ist es nicht möglich, die Größe eines einmal angelegten Arrays nachträglich zu verändern. In Java werden in solchen Fällen meist Vektoren (Klasse `java.util.Vector`) verwendet. Der Zugriff auf Vektoren ist jedoch bei Verwendung einfacher Datentypen unhandlich, da Vektoren nur Objekte enthalten können und die primitiven Datentypen dazu wieder in die genannten Java-Kapselklassen eingebettet werden müssen. Auch das Auslesen erfordert zunächst Typüberprüfungen und Konversionen. Eine Portierung numerischer C-Methoden wird dadurch erschwert, sodass aus praktischen Gründen für diesen Zweck die genannten Klassen entwickelt wurden.

10.2.3 Interfaces zur Übergabe mathematischer Funktionen

Ein weiteres Einsatzgebiet von Zeigern in C/C++ sind Funktionszeiger, welche dazu dienen können, beliebige Funktionen anderen Funktionen als Argumente zu übergeben. In Java arbeitet man stattdessen mit Interfaces. Die folgenden Interfaces wurden für den Fall einiger mathematischer Funktionen, wie sie z. B. für Transformationen benötigt werden, entwickelt.

`SingleFunctionPointer:` deklariert eine einfache Funktion `double w = f(double x)`

`OrdinaryFunctionPointer:` deklariert die Funktionen `double f(double x)`, `double f(double x, double y)` und `double f(double x, double y, double z)`

`ArrayFunctionPointer:` deklariert Funktionen, deren Argumente ein Array enthalten: `double f(double[] x) und double f(t, double[] x)`

`VectorFunctionPointer:` deklariert vektorwertige Funktionen, deren Argumente und Rückgabewerte Arrays enthalten bzw. sind: `double[] fv(double[] x)` und `double[] fv(double t, double[] x)`

`FunctionPointer:` erweitert sämtliche der hier genannten Interfaces

10.3 Wiederholung von Tabellen des Kendig-Pings-Formalismus

Die folgenden Seiten enthalten eine Wiederholung von Formeln aus Abschnitt 5.1 der Kendig-Pings Absorptionskorrektur unter Verwendung einer größeren Schrift.

Fall	Bereiche von a und b	Ausdrücke für Absorptionsfaktoren
1	$\infty > a \geq KR$	$A_{s,sc} = \{2A_s(R)\}^{-1}\left\{SR(R,0)_{s,sc} + SR(R,1)_{s,sc}\right\}$
	$-KR \geq b > -\infty$	$A_{c,sc} = \{2A_{c1}(KR)\}^{-1}\left\{CR(KR,0)_{c,sc} + CR(KR,1)_{c,sc}\right\}$
		$A_{c,c} = \{2A_{c1}(KR)\}^{-1}\left\{CR(KR,0)_{c,c} + CR(KR,1)_{c,c}\right\}$
2	$\infty > a \geq KR$	$A_{s,sc} = \{2A_s(R)\}^{-1}\left\{SR(R,0)_{s,sc} + SR(R,1)_{s,sc}\right\}$
	$-R \geq b > -\infty$	$A_{c,sc} = \{A_{c1}(KR) + A_{c1}(b)\}^{-1}\left\{CR(KR,0)_{c,sc} + CR(b,1)_{c,sc} + CT(b,1)_{c,sc}\right\}$
		$A_{c,c} = \{A_{c1}(KR) + A_{c1}(b)\}^{-1}\left\{CR(KR,0)_{c,c} + CR(b,1)_{c,c} + CT(b,1)_{c,c}\right\}$
3a	$\infty > a \geq KR$	$A_{s,sc} = \{A_s(R) + A_s(b)\}^{-1}\{SR(R,0)_{s,sc} + SR(b,1)_{s,sc} + ST(b,1)_{s,sc}\}$
	$0 \geq b > -R$	$A_{c,sc} = \{A_{c1}(KR) + A_{c2}(b)\}^{-1}\{CR(KR,0)_{c,sc} + CT(b,1)_{c,sc}\}$
		$A_{c,c} = \{A_{c1}(KR) + A_{c2}(b)\}^{-1}\{CR(KR,0)_{c,c} + CT(b,1)_{c,c}\}$
3b	$\infty > a \geq KR$	$A_{s,sc} = \{A_s(R) - A_s(b)\}^{-1}\{SR(R,0)_{s,sc} - SR(b,0)_{s,sc} - ST(b,0)_{s,sc}\}$
	$R > b > 0$	$A_{c,sc} = \{A_{c1}(KR) - A_{c2}(b)\}^{-1}\{CR(KR,0)_{c,sc} - CT(b,0)_{c,sc}\}$
		$A_{c,c} = \{A_{c1}(KR) - A_{c2}(b)\}^{-1}\{CR(KR,0)_{c,c} - CT(b,0)_{c,c}\}$
4	$KR > a \geq R$	$A_{s,sc} = \{2A_s(R)\}^{-1}\{SR(R,0)_{s,sc} + SR(R,1)_{s,sc}\}$
	$-KR \geq b > -\infty$	$A_{c,sc} = \{A_{c1}(a) + A_{c1}(KR)\}^{-1}\{CR(a,0)_{c,sc} + CT(a,0)_{c,sc} + CR(KR,1)_{c,sc}\}$
		$A_{c,c} = \{A_{c1}(a) + A_{c1}(KR)\}^{-1}\{CR(a,0)_{c,c} + CT(a,0)_{c,c} + CR(KR,1)_{c,c}\}$
5	$KR > a \geq R$	$A_{s,sc} = \{2A_s(R)\}^{-1}\{SR(R,0)_{s,sc} + SR(R,1)_{s,sc}\}$
	$-R \geq b > -KR$	$A_{c,sc} = \{A_{c1}(a) + A_{c1}(b)\}^{-1}\{CR(a,0)_{c,sc} + CT(a,0)_{c,sc} + CR(b,1)_{c,sc} + CT(b,1)_{c,sc}\}$
		$A_{c,c} = \{A_{c1}(a) + A_{c1}(b)\}^{-1}\{CR(a,0)_{c,c} + CT(a,0)_{c,c} + CR(b,1)_{c,c} + CT(b,1)_{c,c}\}$
6a	$KR > a \geq R$	$A_{s,sc} = \{A_s(R) + A_s(b)\}^{-1}\{SR(R,0)_{s,sc} + SR(b,1)_{s,sc} + ST(b,1)_{s,sc}\}$
	$0 \geq b > -R$	$A_{c,sc} = \{A_{c1}(a) + A_{c2}(b)\}^{-1}\{CR(a,0)_{c,sc} + CT(a,0)_{c,sc} + CT(b,1)_{c,sc}\}$
		$A_{c,c} = \{A_{c1}(a) + A_{c2}(b)\}^{-1}\{CR(a,0)_{c,c} + CT(a,0)_{c,c} + CT(b,1)_{c,c}\}$
6b	$KR > a \geq R$	$A_{s,sc} = \{A_s(R) - A_s(b)\}^{-1}\{SR(R,0)_{s,sc} - SR(b,0)_{s,sc} - ST(b,0)_{s,sc}\}$
	$R > b > 0$	$A_{c,sc} = \{A_{c1}(a) - A_{c2}(b)\}^{-1}\{CR(a,0)_{c,sc} + CT(a,0)_{c,sc} - CT(b,0)_{c,sc}\}$
		$A_{c,c} = \{A_{c1}(a) - A_{c2}(b)\}^{-1}\{CR(a,0)_{c,c} + CT(a,0)_{c,c} - CT(b,0)_{c,c}\}$

Tabelle 10.1: Fallunterscheidung für die Absorptionsfaktoren

Fall	Bereiche von a und b	Ausdrücke für Absorptionsfaktoren
7a	$R > a \geq 0$ $-KR \geq b > -\infty$	$A_{s,sc} = \{A_s(a) + A_s(R)\}^{-1} \{SR(a,0)_{s,sc} + ST(a,0)_{s,sc} + SR(R,1)_{s,sc}\}$ $A_{c,sc} = \{A_{c2}(a) + A_{c1}(KR)\}^{-1} \{CT(a,0)_{c,sc} + CR(KR,1)_{c,sc}\}$ $A_{c,c} = \{A_{c2}(a) + A_{c1}(KR)\}^{-1} \{CT(a,0)_{c,c} + CR(KR,1)_{c,c}\}$
7b	$0 > a > -R$ $-KR \geq b > -\infty$	$A_{s,sc} = \{A_s(R) - A_s(a)\}^{-1} \{SR(R,1)_{s,sc} - SR(a,1)_{s,sc} - ST(a,1)_{s,sc}\}$ $A_{c,sc} = \{A_{c1}(KR) - A_{c2}(a)\}^{-1} \{CR(KR,1)_{c,sc} - CT(a,1)_{c,sc}\}$ $A_{c,c} = \{A_{c1}(KR) - A_{c2}(a)\}^{-1} \{CR(KR,1)_{c,c} - CT(a,1)_{c,c}\}$
8a	$R > a \geq 0$ $-R \geq b > -KR$	$A_{s,sc} = \{A_s(a) + A_s(R)\}^{-1} \{SR(a,0)_{s,sc} + ST(a,0)_{s,sc} + SR(R,1)_{s,sc}\}$ $A_{c,sc} = \{A_{c2}(a) + A_{c1}(b)\}^{-1} \{CT(a,0)_{c,sc} + CR(b,1)_{c,sc} + CT(b,1)_{c,sc}\}$ $A_{c,c} = \{A_{c2}(a) + A_{c1}(b)\}^{-1} \{CT(a,0)_{c,c} + CR(b,1)_{c,c} + CT(b,1)_{c,c}\}$
8b	$0 > a > -R$ $-R \geq b > -KR$	$A_{s,sc} = \{A_s(R) - A_s(a)\}^{-1} \{SR(R,1)_{s,sc} - SR(a,1)_{s,sc} - ST(a,1)_{s,sc}\}$ $A_{c,sc} = \{A_{c1}(b) - A_{c2}(a)\}^{-1} \{CR(b,1)_{c,sc} + CT(b,1)_{c,sc} - CT(a,1)_{c,sc}\}$ $A_{c,c} = \{A_{c1}(b) - A_{c2}(a)\}^{-1} \{CR(b,1)_{c,c} + CT(b,1)_{c,c} - CT(a,1)_{c,c}\}$
9a	$R > a \geq 0$ $0 \geq b > -R$	$A_{s,sc} = \{A_s(a) + A_s(b)\}^{-1} \{SR(a,0)_{s,sc} + ST(a,0)_{s,sc} + SR(b,1)_{s,sc} + ST(b,1)_{s,sc}\}$ $A_{c,sc} = \{A_{c2}(a) + A_{c2}(b)\}^{-1} \{CT(a,0)_{c,sc} + CT(b,1)_{c,sc}\}$ $A_{c,c} = \{A_{c2}(a) + A_{c2}(b)\}^{-1} \{CT(a,0)_{c,c} + CT(b,1)_{c,c}\}$
9b	$R > a > 0$ $R > b > 0$	$A_{s,sc} = \{A_s(a) - A_s(b)\}^{-1} \{SR(a,0)_{s,sc} + ST(a,0)_{s,sc} - SR(b,0)_{s,sc} - ST(b,0)_{s,sc}\}$ $A_{c,sc} = \{A_{c2}(a) - A_{c2}(b)\}^{-1} \{CT(a,0)_{c,sc} - CT(b,0)_{c,sc}\}$ $A_{c,c} = \{A_{c2}(a) - A_{c2}(b)\}^{-1} \{CT(a,0)_{c,c} - CT(b,0)_{c,c}\}$
9c	$0 > a > -R$ $0 > b > -R$	$A_{s,sc} = \{A_s(b) - A_s(a)\}^{-1} \{SR(b,1)_{s,sc} + ST(b,1)_{s,sc} - SR(a,1)_{s,sc} - ST(a,1)_{s,sc}\}$ $A_{c,sc} = \{A_{c2}(b) - A_{c2}(a)\}^{-1} \{CT(b,1)_{c,sc} - CT(a,1)_{c,sc}\}$ $A_{c,c} = \{A_{c2}(b) - A_{c2}(a)\}^{-1} \{CT(b,1)_{c,c} - CT(a,1)_{c,c}\}$

Tabelle 10.2: Fallunterscheidung für die Absorptionsfaktoren (Fortsetzung)

$$
\begin{aligned}
SR\left(d,q\right)_{s,sc} &= \int_0^{|d|}\int_0^{\pi} \exp\left(-\mu_s l_s\left(r,\omega+q\pi,\theta\right)-\mu_c l_c\left(r,\omega+q\pi,\theta\right)\right) d\omega r dr \\
ST(d,q)_{s,sc} &= \int_{|d|}^{R}\int_0^{\arcsin(|d|/r)} \exp(-\mu_s l_s(r,\omega+q\pi,\theta)-\mu_c l_c(r,\omega+q\pi,\theta))+\exp(-\mu_s l_s(r,\pi-\omega+q\pi,\theta) \\
&\qquad -\mu_c l_c(r,\pi-\omega+q\pi,\theta))d\omega r dr \\
CR(d,q)_{c,sc} &= \int_R^{|d|}\int_0^{\pi} \exp(-\mu_s l_s(r,\omega+q\pi,\theta)-\mu_c l_c(r,\omega+q\pi,\theta))d\omega r dr \\
CT(d,q)_{c,sc} &= \int_{R_{low}}^{KR}\int_0^{\arcsin(|d|/r)} \exp(-\mu_s l_s(r,\omega+q\pi,\theta)-\mu_c l_c(r,\omega+q\pi,\theta))+\exp(-\mu_s l_s(r,\pi-\omega+q\pi,\theta) \\
&\qquad -\mu_c l_c(r,\pi-\omega+q\pi,\theta))d\omega r dr \\
CR(d,q)_{c,c} &= \int_R^{|d|}\int_0^{\pi} \exp(-\mu_c l_c(r,\omega+q\pi,\theta))d\omega r dr \\
CT(d,q)_{c,c} &= \int_{R_{low}}^{KR}\int_0^{\arcsin(|d|/r)} \exp(-\mu_c l_c(r,\omega+q\pi,\theta))+\exp(-\mu_c l_c(r,\pi-\omega+q\pi,\theta))d\omega r dr
\end{aligned}
$$

Tabelle 10.1: Elementarintegrale

Fall	Bereich r	Bereich ω bei gegebenem r	Strahlverlauf	Weglängen
1	$R \geq r > 0$	$2\pi > \omega \geq 0$	c, s, sc, s, c	$l_s = \nu + \eta - \sigma$ $l_c = \alpha + \beta - \nu - \eta$
2a	$R/\cos\theta > r > R$	$2\theta + \pi - \arcsin(R/r) \geq \omega \geq \arcsin(R/r)$	c, sc, c	$l_s = 0$ $l_c = \alpha + \beta - \sigma$
2b	$KR \geq r \geq R/\cos\theta$	$2\theta + \pi - \arcsin(R/r) \geq \omega \geq \arcsin(R/r)$ und		
2c		$2\pi - \arcsin(R/r) \geq \omega \geq 2\theta + \pi + \arcsin(R/r)$		
3a	$R/\cos\theta > r > R$	$2\pi - \arcsin(R/r) \geq \omega > 2\theta + \pi - \arcsin(R/r)$	c, sc, c, s, c	$l_s = 2\eta$ $l_c = \alpha + \beta - \sigma - 2\eta$
3b	$KR \geq r \geq R/\cos\theta$	$2\theta + \pi + \arcsin(R/r) > \omega > 2\theta + \pi - \arcsin(R/r)$		
4a	$R/\cos\theta > r > R$	für $\pi + 2\theta + \arcsin(R/r) \leq 2\pi$: $2\pi > \omega \geq \pi + 2\theta + \arcsin(R/r)$ und $\arcsin(R/r) > \omega > 0$	c, s, c, sc, c	$l_s = 2\nu$ $l_c = \alpha + \beta - \sigma - 2\nu$
4b		für $\pi + 2\theta + \arcsin(R/r) > 2\pi$: $\arcsin(R/r) > \omega \geq 2\theta - \pi + \arcsin(R/r)$		
4c	$KR \geq r \geq R/\cos\theta$	$\arcsin(R/r) > \omega \geq 0$ und $2\pi > \omega > 2\pi - \arcsin(R/r)$		
5a	$R/\cos\theta > r > R$	für $\pi + 2\theta + \arcsin(R/r) \leq 2\pi$: $\pi + 2\theta + \arcsin(R/r) > \omega > 2\pi - \arcsin(R/r)$ für $\pi + 2\theta + \arcsin(R/r) > 2\pi$: $2\pi > \omega > 2\pi - \arcsin(R/r)$ und $\pi + 2\theta + \arcsin(R/r) > \omega \geq 0$	c, s, c, sc, c, s, c	$l_s = 2\nu + 2\eta$ $l_c = \alpha + \beta - \sigma - 2\nu - 2\eta$
5b	$KR \geq r \geq R/\cos\theta$	Bereich tritt nicht auf		

Tabelle 10.3: Fallunterscheidung für Strahlwege

Literaturverzeichnis

[1] A. Brachner, M. Eckert, M. Blum, G. Wolfschmidt, Röntgenstrahlen: Entdeckung, Wirkung, Anwendung; Zum 100. Jubiläum der Entdeckung der X-Strahlen, Deutsches Museum München, 1995

[2] P. Debye, Ann. Physik 46 (1915) 809

[3] F. Zernike, I.A. Prins, Z. Physik 41 (1927) 184

[4] P. Debye, H. Menke, Phys. Z. 31 (1930) 797

[5] P. Debye, H. Menke, Ergebn. techn. Röntgenkunde 2 (1931) 1

[6] W.H. Zachariasen, J. chem. Physics 3 (1935) 158

[7] B.E. Warren, H. Crutter, O. Morningstar, J. Amer. Ceram. Soc. 19 (1936) 202

[8] B.E. Warren, J. appl. Physics 8 (1937) 645

[9] W.C. Pierce, J. chem. Physics 5 (1937) 717

[10] R.F. Kruh, Chem. Reviews 62 (1962) 319

[11] J.U. Weidner, H. Geisenfelder, H. Zimmermann, Ber. Bunsenges. Phys. Chem. 75 (1971) 800

[12] Diplomarbeit U. Eberhardinger, Universität Stuttgart, 1997

[13] J. Gosling, B. Joy, G.L. Steele, The Java Language Specification, Addison-Wesley Longman, Amsterdam, 1996

[14] K. Arnold, J. Gosling, The Java Programming Language, Addison-Wesley, 1996

[15] J. Gosling, H. McGilton, The Java Language Environment, `http://java.sun.com/docs/white/langenv/`

[16] D. Flanagan, Java in a Nutshell, O'Reilly, Köln, 1998

[17] M. Campione, K. Walrath, The Java Tutorial, Addison-Wesley Longman, Amsterdam, 1998, `http://java.sun.com/docs/books/tutorial/`

[18] M. Campione, K. Walrath, A. Huml et al., The Java Tutorial Continued, The Rest of the JDK, Addison-Wesley Longman, Amsterdam, 1999

[19] K. Walrath, M. Campione, The JFC Swing Tutorial, A Guide for Constructing GUIs, Addison-Wesley Longman, Amsterdam, 1999

[20] G. Booch, Objektorientierte Analyse und Design, Addison-Wesley, 1995

[21] T. Budd, An Introduction to Object-Oriented Programming, An introduction to the topic of object-oriented programming, as well as a comparison of C++, Objective C, SmallTalk, and Object Pascal, Addison-Wesley, Reading, Massachusetts, 1991

[22] B. Oesterreich, Objektorientierte Softwareentwicklung: Analyse und Design mit der Unified Modeling Language, Oldenbourg, 1998

[23] I. Sommerville, Software Engineering, Addison-Wesley, 1990

[24] E. Gamma, R. Helm, R.E. Johnson, J. Vlissides, Entwurfsmuster, Elemente wiederverwendbarer objektorientierter Software, Addison-Wesley Professional Computing Series, München, 1998

[25] G.E. Krasner, S.T. Pope, Journal of Object-Oriented Programming, 1(3) (1988) 26

[26] Using a Swing Worker Thread, New Ways to Perform Background Tasks, The Swing Connection ARCHIVE, Tech Topics, http://java.sun.com/products/jfc/tsc/archive/tech_topics_arch/swing_worker/swing_worker.htm

[27] Sprachspezifikation XML 1.0, Entwurf de W3C-Konsortiums vom 14.08.2000, http://www.w3.org/TR/2000/WD-xml-2e-20000814

[28] Sprachspezifikation HTML 4.01, Empfehlungen des W3C-Konsortiums vom 24.12.1999, http://www.w3.org/TR/REC-html40/

[29] E. Armstrong, Working with XML - The Java API for Xml Parsing (JAXP) Tutorial, 13.07.2000, http://java.sun.com/xml/jaxp-1.1/docs/tutorial/index.html

[30] International Tables for X-Ray Crystallography Vol. I-IV, Kluwer Academic Publishing, Dordrecht, 1992

[31] JavaHelp 1.1 User Guide, http://www.java.sun.com/products/javahelp/download_binary.html#userguide

[32] S. Münz, SELFHTML, http://www.teamone.de/selfaktuell/

[33] G. Heusel, Dissertation in Vorbereitung

[34] http://www.esrf.fr/computing/expg/subgroups/theory/DABAX/dabax.html

[35] J.H. Hubbell, Natl. Stand. Ref. Data Ser. 29 (1969)

[36] J.H. Hubbell, Radiat. Res. 70 (1977) 58

[37] J.H. Hubbell, H.A. Gimm., I. Overbo, J. Phys. Chem. Ref. Data 9 (1980) 1023

[38] J.H. Hubbell, W.J. Veigele, E.A. Briggs, R.T. Brown, D.T. Cromer, R.J. Howerton, J. Phys. Chem. Ref. Data 4, (1975) 471; erratum in 6 (1977) 615

[39] J.H. Hubbell, I. Overbo, J. Phys. Chem. Ref. Data 8 (1979) 69

[40] J.H. Scofield, Lawrence Livermore National Laboratory Rep. UCRL-51326 (1973)

[41] R.H. Pratt, Phys. Rev. 117 (1960) 1017-1028

[42] E.B. Saloman, J.H. Hubbell, Nucl. Instr. Meth. A255 (1987) 38

[43] E.B. Saloman, J.H. Hubbell, National Bureau of Standards Report NBSIR 86-3431 (1986)

[44] H. Bertagnolli, T.S. Ertel, Angew. Chem. 106 (1994) 15

[45] G. Williams, Electron binding energies in eV for the elements in their natural forms, 1995, `http://xray.uu.se/hypertext/EBindEnergies.html`

[46] J.A. Bearden and A.F. Burr, Rev. Mod. Phys. 39 (1967)

[47] M. Cardona and L. Ley, Eds., Photoemission in Solids I: General Principles, Springer-Verlag, Berlin, 1978

[48] J.C. Fuggle and N. Mårtensson, J. Electron Spectrosc. Relat. Phenom. 21 (1980) 275

[49] L.V. Azaroff, Acta Cryst. 8 (1955) 701

[50] D.A. Kerr, J.P. Ashmore, Acta Cryst. A30 (1974) 176

[51] O. Klein, Y. Nishina, ZS. f. Phys. 52 (1929) 853

[52] D. Waasmaier, A. Kirfel, Acta Cryst. A51 (1995) 416

[53] D.T. Cromer, J.B. Mann, J. Chem. Phys. 47 (1967) 1892

[54] D.T. Cromer, J. Chem. Phys. 50 (1969) 4857

[55] H.H.M. Balyuzi, Acta Cryst. A31 (1975) 600

[56] J. Krogh-Moe, Acta Cryst. 9 (1956) 951

[57] N. Norman, Department of Physics, University of Oslo, Thesis No. 219 (1954)

[58] N. Norman, Acta Cryst. 7 (1954) 462

[59] N. Norman, Acta Cryst. 10 (1957) 370

[60] P.A. Egelstaff, An introduction to the Liquid state, Clarendon Press Oxford (1994)

[61] S.A. Rice, P. Gray, The statistical mechanics of simple liquids - an introduction to the theory of equilibrium and non-equilibrium phenomena, Interscience Publ. New York (1965)

[62] Dissertation A. Lemke 1999, Stuttgart

[63] Y. Fukushima, K. Suzuki, Kakuriken Kenkyu Hokoku 9 (1967) 235

[64] D. Price, W. Howells, M. Tosi, Proc. Electrochem. Soc. 93-3 (1993)

[65] S. Nyburg, G. Ozin, J. Szymanski, Acta Cryst. B27 (1972) 2298

[66] H. Bartl, Z. Anal. Chem., 312 (1982) 17

[67] A.P. Kendig, C.J. Pings, J. Appl. Phys. 36 (1965) 1692

[68] H.H. Paalman, C.J. Pings, J. Appl. Phys. 33 (1962) 2635

[69] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, Numerical Recipes in C, The Art of Scientific Computing, Cambridge University Press, 1996

[70] E.O. Brigham, FFT-Anwendungen, Oldenbourg, München, Wien, 1997

[71] W. Cooley, J.W. Tukey, Math of Computation 19 (1965) 297

[72] J.W. Cooley, P.A.W. Lewis, P.D. Welch, J. Sound Vib. 12(3) (1970) 315

[73] C. Temperton, J. Comp. Phys. 34 (1980) 314

[74] C. Van Loan, Computational Frameworks for the Fast Fourier Transform, Society for Industrial and Applied Mathematics, Philadelphia, 1992

[75] D.W. Marquardt, Journal of the Society for Industrial and Applied Mathematics, vol. 11 (1963) 431

[76] A. Savitzky, M.J.E. Golay, Smoothing and Differentiation of Data by Simplified Least Squares Procedures, Analytical Chemistry 36 (1964) 1627

[77] J. Hicklin, R.F. Boisvert et al., The MathWorks and the National Institute of Standards and Technology (NIST), The Java Matrix Package JAMA, `http://math.nist.gov/javanumerics/jama/`

[78] Postscript Language Reference, Addison-Wesley Longman, Amsterdam, 1999

[79] Programmer's Guide to the Java 2D^{TM} API, Enhanced Graphics and Imaging for Java, Sun Microsystems, 1999

Lebenslauf

Ulrich Eberhardinger
Geboren am 30.10.1969 in Stuttgart

Schulbildung:

1976-1980	Eichendorff-Grundschule in Stuttgart
1980-1989	Elly-Heuss-Knapp-Gymnasium in Stuttgart
26.04.1989	Abitur

Wehrdienst:

01.06.1989-31.08.1990	Grundwehrdienst in Münsingen und Stuttgart

Studium:

01.10.1990-20.02.1997	Studiengang Diplom-Chemie an der Universität Stuttgart
12.08.1996-20.02.1997	Diplomarbeit am Institut für Physikalische Chemie in der Arbeitsgruppe von Prof. Dr. H. Bertagnolli zum Thema „Röntgenbeugung an modifizierten Gläsern“
20.02.1997-22.04.2002	Promotion am Institut für Physikalische Chemie in der Arbeitsgruppe von Prof. Dr. H. Bertagnolli zum Thema „Entwicklung einer graphischen Benutzeroberfläche zur Auswertung von Röntgendiffraktogrammen ungeordneter Systeme“