WA ThEP
Fachbereich Physik
Johannes Gutenberg-Universität Mainz

Dissertation

zur Erlangung des Grades
„Doktor der Naturwissenschaften“

Algorithmische Methoden zur Berechnung von Vierbeinfunktionen

Richard Kreckel
geboren in Bingen/Rh.

Mainz, 2002

Erster Berichterstatter: Prof. Dr. Karl Schilcher
Zweiter Berichterstatter: Dr. habil. Stefan Scherer
Dritter Berichterstatter: Prof. Dr. Jos A. M. Vermaseren
Datum der mündlichen Prüfung: 19. Juli 2002

The practical scientist is trying to solve tomorrow's problem with yesterday's computer; the computer scientist, we think often has it the other way around.
Press et alii *[PTVF 1992, section 1.2]*

Inhaltsverzeichnis

Einleitung 1

I. Berechnung von Schleifenintegralen 9

1. Feynmandiagramme 11
- 1.1. Quantenfeldtheorien und das Standardmodell . . . 11
- 1.2. Standardmethoden zur Berechnung von Feynmandiagrammen . . . 12
- 1.3. Kinematische Abhängigkeiten . . . 17
- 1.4. Die bisher untersuchten Funktionen („Mainz I und Mainz II“) . . . 18
- 1.5. Beschränkte Integrationsgebiete nach Residuenintegration? . . . 20

2. Die skalaren Zweischleifen-Vierbeinfunktionen 25
- 2.1. Die Vierbeinfunktion („Mainz III“) . . . 25
- 2.2. Die einschränkenden Bedingungen . . . 34
- 2.3. Umformung der einschränkenden Bedingungen . . . 41
- 2.4. Ausblick: Das weitere Vorgehen . . . 49

II. Computeralgebra für Schleifenrechnungen 53

3. GiNaC: Motivation und Design 55
- 3.1. Die Motivation für GiNaC . . . 55
- 3.2. Das Design von GiNaC . . . 64

4. GiNaC: Implementierung 77
- 4.1. Die wichtigsten Klassen . . . 77
- 4.2. Kanonisierung von Produkten: die Klassen ‚mul' und ‚ncmul' . . . 78
- 4.3. Vereinfachungen in der Klasse ‚power' . . . 79
- 4.4. Die Numerik-Klasse . . . 83
- 4.5. Pseudofunktionen . . . 88
- 4.6. Laurentreihen: die Klasse ‚pseries' . . . 89
- 4.7. Die Matrix-Klasse . . . 98

5. Kritische Analyse des GiNaC-Ansatzes 113
- 5.1. Effizienz . . . 113

5.2. Handhabbarkeit 118
5.3. Erweiterbarkeit 123
5.4. Schlussfolgerungen und Ausblick 127

A. Hilfsmittel aus der komplexen Analysis 133
A.1. Der Cauchy'sche Residuensatz in einer Veränderlichen 133
A.2. Hauptwertintegrale 137
A.3. Schnitte, Umkehrungen elementarer Funktionen und all das 138

B. ‚pvegas': parallele MC-Integration 151
B.1. Vegas 151
B.2. Parallelisierung 152
B.3. Nebenintegrale 155
B.4. Parallele Zufallszahlen 156
B.5. Praktische Erfahrungen und Perspektiven 159

Glossar 165

Schlagwortverzeichnis 171

Literaturverzeichnis 177

Abbildungsverzeichnis

0.1. GRACE Systemflussdiagramm . . . 3
0.2. DIANA Systemflussdiagramm . . . 4
0.3. Symbolische Menüstruktur von CompHEP . . . 5
0.4. Struktur von *xloops* . . . 6

1.1. Schematische Gleichung als Beispiel partieller Integration . . . 16
1.2. Die gekreuzte Dreibeinfunktion . . . 18
1.3. Die Master- und daraus abgeleitete Topologien . . . 19
1.4. Alle Beiträge in den zugeordneten Impulsen werden endlich . . . 22
1.5. Gebiete der zugeordneten Impulse bei gekreuzten Funktionen . . . 23

2.1. Die Topologie II a) . . . 26
2.2. Mögliche skalare Vierbeintopologien . . . 27
2.3. Verbleibende Terme nach einer Residuenintegration . . . 35
2.4. Wahlfreiheit bei $\int dl\, 1/(P_1(l)P_2(l))$ . . . 35
2.5. Verbleibende Terme nach zwei Residuenintegrationen . . . 37
2.6. Graphische Darstellung eines faktorisierten Polynoms in θ-Funktionen . . . 46
2.7. Alternative Darstellung eines Polynoms in θ-Funktionen . . . 47
2.8. Terme nach zwei verschiedenen Integrationsmethoden zur planaren Box . . . 49
2.9. Graphische Darstellung der k_0-l_0-Gebiete bei der planaren Box . . . 50

3.1. Die Klassenhierarchie von GiNaC . . . 65
3.2. Mögliche unevaluierte Darstellung von $2d^3(4a + 6b - 3 - b)$ . . . 73
3.3. Naive Darstellung von $2d^3(4a + 5b - 3)$ . . . 73
3.4. Realistische Darstellungen von $2d^3(4a + 5b - 3)$ . . . 74
3.5. Baumdurchschreitung: Preorder und Postorder . . . 74
3.6. Laufzeiten für Denny Fliegners Konsistenztest . . . 76

4.1. Die Klassenhierarchie von CLN . . . 84
4.2. Laufzeiten der Multiplikation in CLN . . . 85
4.3. Laufzeiten der Multiplikation in verschiedenen Softwarepaketen . . . 86
4.4. Methodenaufruf bei Pseudofunktionen . . . 89
4.5. Laufzeiten für die Laurententwicklung von $\Gamma(x)|_{x=0}$ . . . 97
4.6. Schleifenumordnung bei der Matrix-Multiplikation . . . 102
4.7. Überflüssige Minorenberechnung bei Laplace-Entwicklung . . . 104
4.8. Partitionierungen zum Beweis der Sylvester-Identität . . . 108
4.9. Dreiecksmatrix vs. Staffelmatrix . . . 110

5.1. Zeitliche Entwicklung der Effizienz von GiNaC 114
5.2. Speicherbedarf verschiedener CA-Systeme 116
5.3. GiNaCs Darstellungsgraph von $2\sinh(x)^2/\cosh(x)^3 - 1/\cosh(x)$ 117
5.4. STL Template Speicherallozierung 118

A.1. Zum Deformationssatz 133
A.2. Integrationswege bei der Hauptwertintegration 138
A.3. Die Blätter von $\sqrt{z}$ und $\sqrt[3]{z}$ 139
A.4. Der Imaginärteil von $\log(z)$ 140
A.5. Real- und Imaginärteil des Arcustangens 142
A.6. Real- und Imaginärteil des Arcussinus 145
A.7. Real- und Imaginärteil des Arcuscosinushyperbolicus 146
A.9. Real- und Imaginärteil des Dilogarithmus 147
A.8. Integrationsweg zum Schnitt von $\mathrm{Li}_2(z)$ 147

B.1. Sampling-Methoden 152
B.2. Konvergenzvergleich dreier verschiedener Parallelisierungsansätze 153
B.3. Schieberegister-, Tausworthe und Kirkpatrick-Stoll-Reihe 157
B.4. Effizienz von `pvegas` 159
B.5. Laufzeitverhalten von `pvegas` 160
B.6. Delaunay-Triangulation des 3-Würfels 160
B.7. Vergleich der Gitterstruktur von Vegas und ParInt 161
B.8. Unterteilungsbaum eines Gitters 161

Tabellenverzeichnis

1.1. Anzahl $\sharp$ der externen Parameter der n-Bein-Funktionen in $D = 4$ Dimensionen 18
1.2. Die zwei möglichen Kombinationen von Bedingungen 22

4.1. Laufzeiten zur Generierung und Ausmultiplikation symbolischer Determinanten 104
4.2. Anzahl der elementaren Rechenoperationen zur Determinantenberechnung . . 107

5.1. Eckpunkte in der zeitlichen Entwicklung der Effizienz von GiNaC 114
5.2. Vergleich symbolischer Pakete nach Robert Lewis und Michael Wester 115

A.1. Auflistung der Schnitte in der komplexen Ebene 149

B.1. `pvegas` ist auf allen derzeit gängigen Parallelrechnern lauffähig. 158
B.2. Argumente im `pvegas` Prototyp 163

Einleitung

There are two ways of constructing a software design: One way is to make it so simple that there are obviously *no deficiencies, and the other way is to make it so complicated that there are no* obvious *deficiencies. The first method is far more difficult.*
Charles A.R. Hoare [Hoar 1981]

Die Durchführung von physikalischen Berechnungen sprengt seit einiger Zeit nicht nur in numerischer Hinsicht den Rahmen dessen, was mit Bleistift auf Papier durchführbar ist. Auch symbolische Umformungen verlangen immer mehr nach einer automatischen Überprüfung oder können gar vollautomatisiert durchgeführt werden. So ergab es sich, dass die theoretische Physik beträchtlich zur Entwicklung der Computeralgebra beitragen konnte. Erste Computeralgebrasysteme (CAS) in den 1960er Jahren waren hauptsächlich Lisp-basierte Pakete zur Manipulation von Polynomen und rationalen Gleichungen und zum heuristischen Integrieren auf dem Niveau eines Studenten in den ersten Studiensemestern. Rechnungen in Quantenfeldtheorien stellten damals schon äußerste Ansprüche in Bezug auf Rechengeschwindigkeit und Skalierbarkeit. Das erste erfolgreiche System, das direkt aus der Hochenergie kam – das 1963 von Martinus Veltman entwickelte Schoonschip – war konsequenterweise direkt in der Maschinensprache des Rechners IBM 7094 geschrieben. Erst nachdem Schoonschip 1983 in der Maschinensprache der populären Motorola 68000 Prozessorfamilie neu geschrieben worden war gewann es eine gewisse Verbreitung [VeWi 1993]. 1968 stellte Anthony Hearn mit REDUCE ein interaktives Lisp-basiertes System vor, welches speziell für physikalische Rechnungen entwickelt worden, aber weit universeller einsetzbar war als Schoonschip. Es erlang rasch eine breite Beliebtheit. Weitere Systeme, die direkt aus der Hochenergiephysik kamen waren SMP – das von Stephen Wolfram geschriebene Vorläufersystem des seit 1988 kommerziellen Mathematica – und FORM von Jos Vermaseren. Diese beiden Systeme sind in der Sprache C implementiert,[1] was wohl hauptsächlich die wachsende Popularität und Verbreitung dieser Sprache in den 1970er und 80er Jahren wiederspiegelt.

Die Berechnung von Strahlungskorrekturen in Quantenfeldtheorien insbesondere erfordert umfangreiche Berechnungen. In den letzten Jahrzehnten hat sich gezeigt, dass die zugrundeliegenden Verfahren häufig einer algorithmischen Behandlung zugänglich sind. Die Formulierung einer Übergangsamplitude anhand ihres Feynmandiagrammes führt zum Beispiel auf Schleifenintegrale, die ausgeführt werden müssen. Aber auch schon auf Baumebene sind rechnergestützte Verfahren unabdingbar, da die Anzahl der Diagramme bestenfalls exponentiell, meist mit der Fakultät der äußeren Beine anwächst: In einer ϕ^3-Theorie beträgt die Anzahl der

[1] Die erste Version von FORM war allerdings noch in FORTRAN geschrieben.

Feynmandiagramme, die auf Baumebene zu einem Prozess mit n äußeren Beinen beitragen, beispielsweise $F_\Gamma(n) = (2n-5)!! = (2n-5)\cdots 5\cdot 3\cdot 1$.

Eine ganze Reihe von Softwarepaketen zur Berechnung von Amplituden sind in den letzten zehn Jahren vorgestellt worden. Hier soll und kann kein Überblick mit Anspruch auf Vollständigkeit gegeben werden. Einen solchen gewährt beispielsweise [HaSt 1998]. Lediglich anhand einiger wichtiger Pakete möchte ich hier Betrachtungen aufzeigen, die für das Design meines Erachtens nach eine prominente Rolle spielen sollten.

Grace [IKKKST 1993] und CompHEP [BDIPS 1994] beschränken sich im Wesentlichen auf Baumebene und vermögen nur spezielle Einschleifenintegrale zu berechnen. Grace bedient sich hierfür ausnahmslos der Feynmanparametrisierung, wobei im besten Falle eine Integration numerisch auszuführen bleibt. Beide Systeme können dort aber viele komplette Prozesse im Standardmodell inklusive der nötigen Phasenraumintegrationen durchführen.

Das in Mainz entwickelte *xloops* basiert auf einer Serie von Publikationen [Krei 1991, Krei 1992a, Krei 1992b, CKK 1994], in denen erstmals die Zerlegung in Parallel- und Othogonalraum für Zweischleifen-Integrale ausgedehnt wurde – zuvor wurde diese Technik schon von verschiedenen Gruppen erfolgreich auf Ein-Schleifen-Integrale angewendet. Eine Implementierung dieser Methoden als Sammlung von Maple-Routinen beschränkte sich zunächst auf Ein-Schleifen-Integrale [BFK 1995] und wurde später erfolgreich auf Zweischleifen-Integrale bis maximal zwei äußere Beinen ausgedehnt [BFK 1998].

Pakete zur Schleifenrechnung werden üblicherweise an der erfolgreichen Durchführung einer oder mehrerer Rechnungen bewertet. Im Laufe meiner Arbeit im Umfeld des *xloops*-Projektes bin ich zu der Überzeugung gelangt, dass dieser Ansatz jedoch zu bequem ist. Ein Softwaresystem kann punktuell noch so erfolgreich sein – es ist von geringem Wert, wenn es nicht erweiterbar ist, von Fremden nicht benutzt und von neuen Diplomanden und Doktoranden kaum durchschaut werden kann. Unbrauchbar wird es dann, wenn es so verschachtelt ist, dass keine Komponente geändert werden kann, ohne dass dies unerwartete Auswirkungen an anderen Stellen hat. Gefährlich wird es, wenn solche unerwartete Auswirkungen unbemerkt bleiben, weil versäumt worden ist, bekannte und mit unabhängigen Methoden verifizierte Ergebnisse durch automatisierte Regressionstests abzusichern gegen strukturelle Fehler die sich beim vermeintlichen Reparieren von Programmierfehlern einschleichen.

So ist es bislang nicht gelungen, den allgemeinen Fall der Dreibeinfunktion auf Zweischleifen-Niveau in *xloops* zu integrieren. Abgesehen von zahlreichen neuen numerischen Instabilitäten, die in dieser Methode aufzutreten scheinen [Frin 2000] lag das Hauptproblem an der wenig durchsichtigen Architektur von *xloops*. Diese wiederum ist eine Konsequenz der veralteten symbolischen Sprache von Maple, die zusammen mit Tcl/Tk, welches auch nicht als Vorbild strukturierter Programmiersprachen herhalten kann, als Vehikel für die gesamten Berechnungen dienen musste. Dies führte zu einem bedauerlichen Stillstand des Projektes.

Leider scheint dies in Softwarepaketen aus der theoretischen Hochenergiephysik ein allgemein verbreiteter Zustand zu sein (die Experimentalphysiker gehen häufig professioneller vor). Ein paar Blicke auf Softwareengineering können hier Besserung verschaffen. Ein besonderes Problem bildet die Tatsache, dass die bekannten Pakete in der Anzahl der benutzten externen Programmiersprachen und damit auch Programmierparadigmen proliferieren. Dies zwingt zwar zu Modularität, aber zu einer Ungewollten. Und es erschwert den Einstieg in die Arbeit an einem solchen System unnötig. Abbildung 0.1 zeigt dies im Falle des Systems Grace. Die

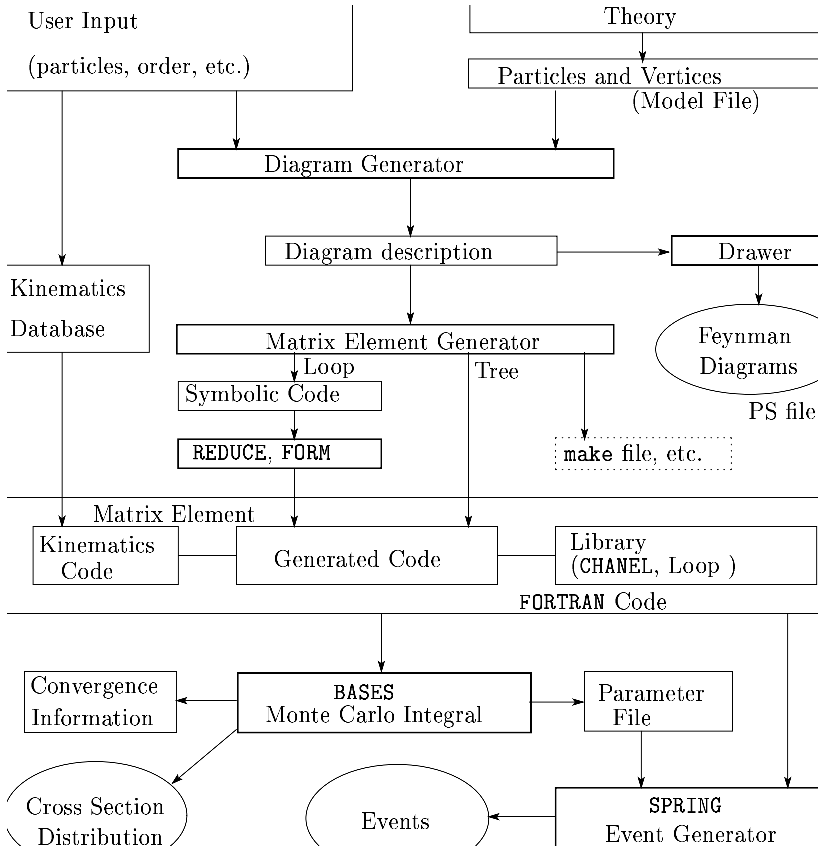


Abbildung 0.1.: *Das Systemflussdiagramm von GRACE (aus [Bèla 1999]).*

Sprachen FORTRAN und C dominieren hier, jedoch müssen zur symbolischen Rechnung auch REDUCE und FORM herangezogen werden. In *xloops* hat Maple diese Rolle übernommen.

Die meisten Systeme mit dem Baustein FORM als symbolischer Maschine leiden darunter, dass dieses System wirkliche Programmierung praktisch nicht unterstützt. Die Struktur einer FORM-Datei ist immer die Folgende: **Deklarationen** (Symbole, z.B. `Symbol a;`), **Spezifizierungen**, **Definitionen** (z.B. `Local F;`), **Anweisungen** (atomare Instruktionen wie `.sort` aber auch Flusskontrolle wie `if()` und `while()`) und **Ausgabeanweisung** (wie `.print`).[2] Da ein Programmieren ohne echte Programmierumgebung jedoch undenkbar ist, ziehen Softwarepakete zur Schleifenberechnung für die fehlenden Konstrukte andere nicht-symbolische Systeme auf einer höheren Ebene heran. Im klassischen *xloops* war dies zum Beispiel Tcl/Tk – obwohl die Unterstützung zur Programmierung im dort verwendeten Symbolikpaket MapleV weit über diejenige von FORM hinausgeht.

Auch das System DIANA [FlTe 1999a] bietet eine graphische Benutzeroberfläche[3] [FlTe 2000]. DIANA ist im Wesentlichen ein Graphengenerator der über die rein topologische Generierung hinausgeht und FORM-Programme erzeugt die alle Graphen zu einem gegebenen Prozess

[2] Es gibt zwar „Prozeduren“ in FORM, diese werden jedoch nicht aufgerufen mit Parameterübergabe auf einem Stack, sondern sie werden lediglich von einem Präprozessor expandiert. „Makros“ wäre eigentlich eine treffendere Bezeichung.

[3] DIANA's Benutzerschnittstelle ist übrigens derjenigen des alten *xloops* nicht ganz unähnlich.

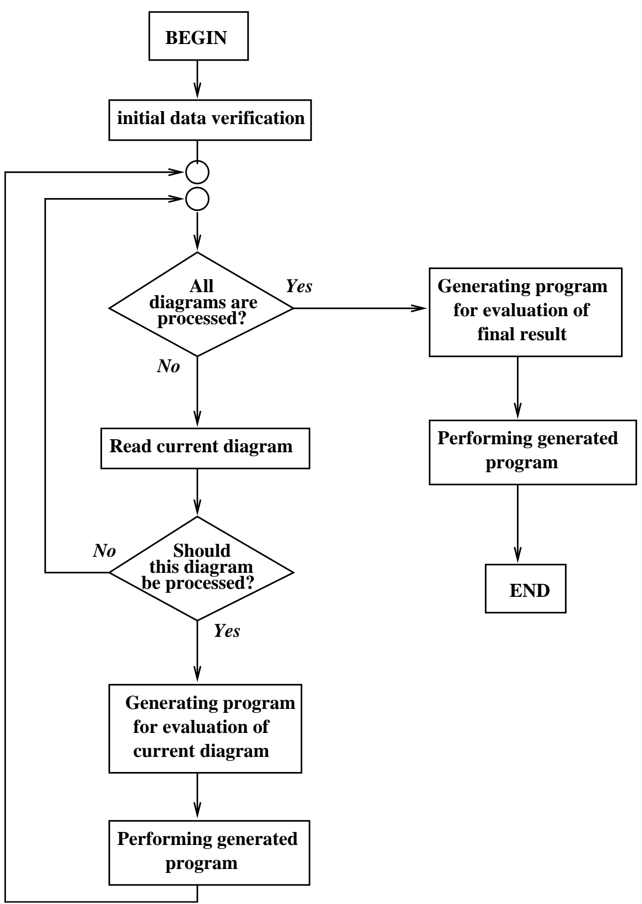


Abbildung 0.2.: *Das Systemflussdiagramm von DIANA (aus [FlTe 1999a]).*

darstellen. Es beansprucht vor allem dort Fähigkeiten, wo *xloops* bis dato wenig zu bieten hat: auf der automatischen Summierung von Feynmangraphen für einen gegebenen physikalischen Prozess. Allerdings müssen die einzelnen Graphen dann von anderen Programmen berechnet werden. Hierzu bedient es sich der eigens für diesen Zweck geschaffenen Sprache TM [FlTe 1999b]. Es sei dahingestellt ob die Einführung einer weiteren Programmiersprache wirklich eine Erleichterung darstellt und ob die von ihr ausgeführte Tätigkeit der Makroexpansion nicht durch vorhandene, weit verbreitete und getestete Makrosprachen wie etwa dem allgegenwärtigen m4 bewerkstelligt werden könnte.

Ein zukünftiges *xloops*-Paket sollte nach Möglichkeit Mehrsprachigkeit soweit wie möglich vermeiden. Als algebraisches System kann GiNaC alle notwendigen symbolischen Manipulationen direkt in C++ erledigen. Ob darauf aufbauend eine graphische Benutzerschnittstelle wirklich sinnvoll ist kann in mittlerer oder ferner Zukunft entschieden werden. Als ungemein wertvoll hat sich erwiesen, dass jede nichttriviale Funktionalität von automatischen Regressionstests stets wieder auf Korrektheit überprüft wird. Ansonsten ist die Software stets der absolut nicht unrealistischen Gefahr ausgesetzt, dass eine Änderung an einem Modul unbemerkt die Korrektheit eines anderen Moduls zerbricht. Alle bekannten Pakete zur Schleifenberechnung weisen meines Erachtens hier große Defizite auf. Auch das bisherige *xloops* ist hierfür ein trauriges Beispiel. Solche Regressionstests sollten möglichst die gesamte Funktionalität abdecken

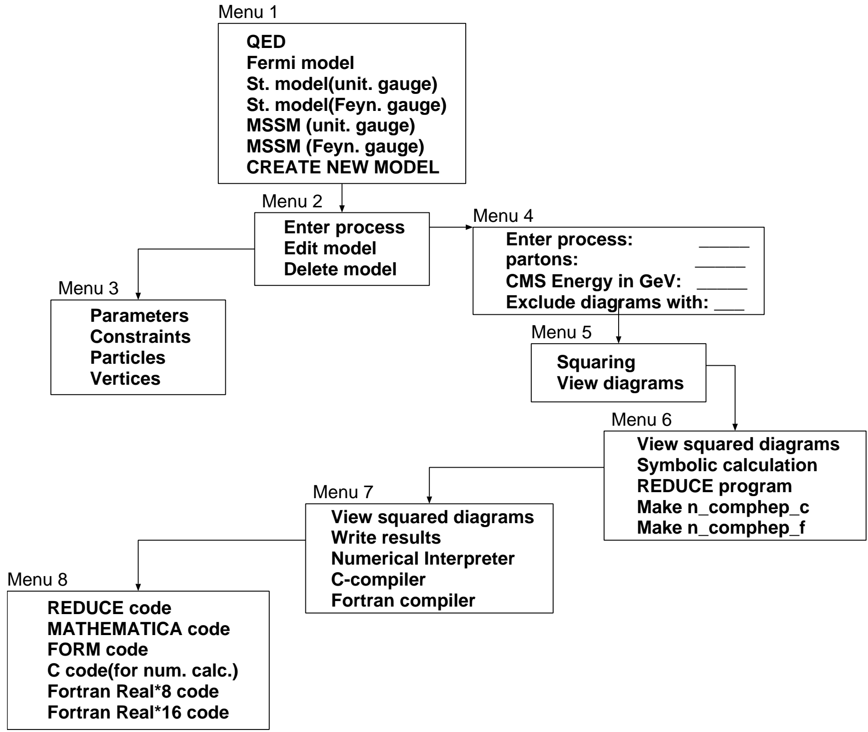


Abbildung 0.3.: *Symbolische Menüstruktur von CompHEP (aus [Pukh 1999]).*

und orthogonal gestaltet sein, um selbst nicht übermäßig umfangreich zu werden.

Benutzer von *xloops* mussten bislang für einen bestimmten kinematischen Fall einen Knopf mit der betreffenden Topologie anklicken und darin die Teilchenarten sowie die äußeren Impulse eingeben. Alsdann wurden die divergenten von den konvergenten Anteilen (in dimensionaler Regularisierung und unter Berücksichtigung von Abzugstermen) abgetrennt, die divergenten Anteile symbolisch, also exakt, berechnet und eine Zweifachintegraldarstellung für den konvergenten Anteil generiert. Auf Wunsch (sprich: Knopfdruck) konnte dieser dann numerisch von einem Monte-Carlo- (MC-) Integrator approximiert werden. Ein solches menügesteuertes Vorgehen erschwert aber die Berechnung von physikalischen Prozessen, in denen im Zweischleifen-Fall manchmal viele Hundert Diagramme zu berechnen sind. Dann wird die vermeintliche Vereinfachung für den Benutzer nämlich zu einem unüberwindbaren Hindernis. CompHEP leidet mit seinen tief verschachtelten Menüs unter demselben Problem. Hier muss ein solider Graphengenerator zur Verfügung gestellt werden, der abhängig von einem auswechselbaren physikalischen Modell (spezifiziert als Feynmanregeln) alle Diagramme zu einem gegebenen Prozess mitsamt Symmetriefaktoren erzeugt.

Für ein zukünftiges *xloops* wäre also auch eine neue Aufarbeitung des Themas Graphengenerierung empfehlenswert. Das bisherige Vorgehen ist weitgehend heuristischer Natur und funktioniert im Wesentlichen durch Anhängen äußerer Beine an die Einschleifen- und Zweischleifen-Mastertopologien. Die bisherigen Implementierungen sind recht unbefriedigend – und können auch kaum auf Dreischleifendiagramme fortgesetzt werden. Ein sehr effizienter graphentheoretischer Ansatz wurde in [Nogu 1993] vorgestellt. Die begleitende FORTRAN-Implementierung QGRAF lässt aber insofern sehr zu wünschen übrig, als dass sie keine Schnittstellen zur Wei-

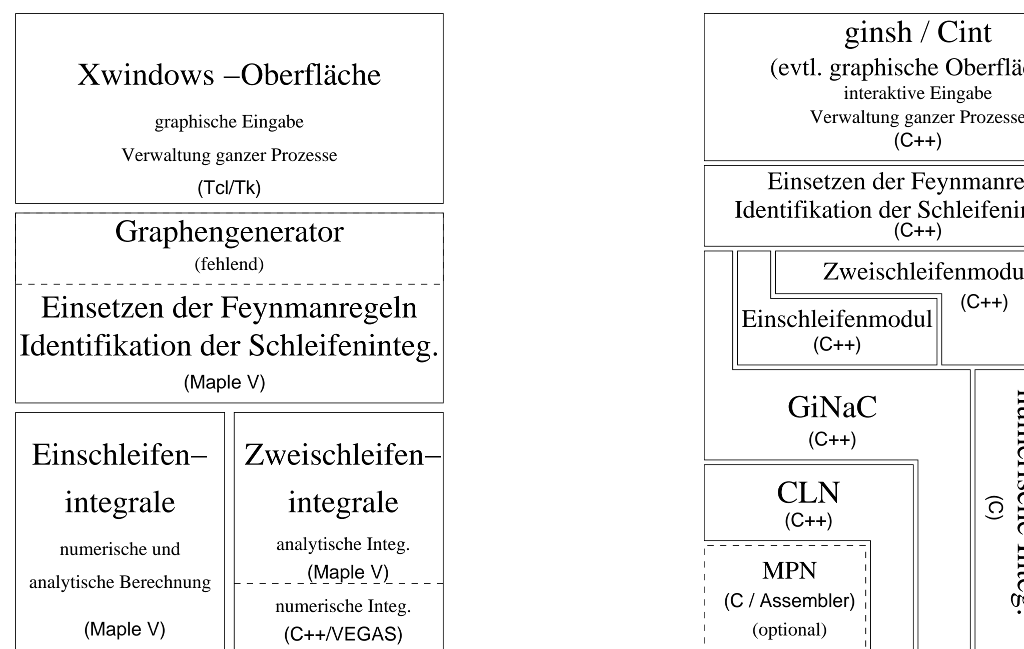


Abbildung 0.4.: *Die Struktur des alten xloops (aus [Brue 1997]) und eine Vision des Neuen.*

terverarbeitung der generierten Graphen besitzt – lediglich Textausgabe ist implementiert. Für GRACE wurde der dort benutzte Algorithmus auch reimplementiert [Kane 1995] – das Modul heißt dort GRC. Der Aufwand ist recht überschaubar und setzt nicht einmal symbolische Manipulationen voraus. Die Spezifizierung physikalischer Modelle erfolgt sowohl in QGRAF als auch in GRC jeweils in Form einer Modelldatei, die eingelesen und in eine interne Darstellung umgewandelt wird. Möglicherweise wird die Zunkunft hier eine Vereinheitlichung bringen in Form von als XML spezifizierten Feynmanregeln. Entsprechende Bestrebungen wurden angekündigt, es liegt aber zu diesem Datum noch kein konkreter Vorschlag vor.

xloops könnte auch von einer allgemeinen Verfügbarkeit profitieren so dass es leichter zu testen ist. Die Abhängigkeit von Maple in einer bestimmten Versionsnummer hat sich hier als sehr störend herausgestellt. Das war einer der Gründe für die Entwicklung von GiNaC. Heute ist dies interessanterweise nicht mehr ganz so dringend wie 1998: Nach dem Erscheinen von GiNaC wurden noch ein paar weitere bedeutende Pakete, die Computeralgebra tangieren, von einer restriktiven Lizenz beziehungsweise nicht einsehbaren Quellcodes zu einer Lizenz unter dem GNU-Modell gestellt: NTL (April 2000), Pari (November 2000) und Singular (April 2001). Zeitgeist, Zufall oder Memreplikation?

Abbildung 0.4 vergleicht das bisherige *xloops* und wagt eine Zunkunftsvision, wie die strukturellen Abhängigkeiten in einer zukünftigen Version aussehen könnten beziehungsweise sich derzeit abzeichnen. Besonders hervorzuheben ist meines Erachtens, dass symbolische Vehikel nur dann herangezogen werden sollten, wenn dies unabdingbar ist. Wir haben zwar mit GiNaC nun ein System, welches leistungsfähig genug ist, die in *xloops* vorkommenden Rechnungen zu unterstützen, jedoch sind wir nicht gefeit vor selbstgemachten ausufernden Abhängigkeiten, die die Wartbarkeit nur unnötig erschweren. Die symbolische Domäne ist ungleich aufwändiger als die numerische und es lohnt sich ihr so weit wie möglich aus dem Wege zu gehen.

Zum Aufbau dieser Arbeit: Das erste Kapitel steckt die Bühne, auf der sich die hier ent-

wickelten Berechnungsmethoden abspielen sollen, ab und gibt einen Überblick über das in Mainz entwickelte und bisher praktizierte Repertoire. Das zweite erweitert dieses Repertoire um ein neues Stück – die Zweischleifen-Vierbeinfunktionen. Im dritten Kapitel werden dann die Sprechübungen nachgeholt, indem die bisher benutzten Computeralgebrasysteme einer gründlichen Untersuchung unterzogen werden. Dabei wird sich herausstellen, dass die bisher benutzten Sprachen für die Inszenierung größerer Stücke ungeeignet sind und GiNaC wird als Ersatz vorgestellt. In Kapitel vier wird der Fundus durchstöbert wobei aus der Vielzahl der Requisiten nur ein paar bislang nicht oder nur unvollständig beschriebene ausgewählt werden. Dabei wird die Absicht weniger sein, eine Anleitung für die Benutzung der GiNaC-Bibliothek zu geben – hierfür ist die offen verfügbare Dokumentation zuständig. Vielmehr sollen einige Implementationsentscheidungen motiviert werden, für deren Darlegung es sonst kein geeignetes Forum gibt. Das fünfte Kapitel versucht von einer – soweit es mir möglich ist – distanzierten Sichtweise, dieses System zwischen vergleichbaren Systemen einzuordnen. In zwei Anhängen werden ein paar Hilfsmittel skizziert, in einem Glossar einige für Physiker nicht allzu gebräuchliche Begriffe erläutert.

Wenn es bisweilen den Anschein hat, dass hier mehr Fragen aufgeworfen als beantwortet werden, dann ist dies weder Zufall, noch beabsichtigt, sondern unvermeidlich.

Teil I.

Berechnung von Schleifenintegralen

1. Feynmandiagramme

Science is what we understand well enough to explain to a computer.
Art is everything else we do.
Donald E. Knuth, Reader's Digest, Juli 1987

1.1. Quantenfeldtheorien und das Standardmodell

Die Notwendigkeit, quantisierte Felder der Beschreibung relativistischer Teilchen zugrunde zu legen, erwächst aus zwei Gründen: Erstens zwingt uns die Einstein'sche Masse-Energie-Beziehung, den Ein-Teilchen-Standpunkt aufzugeben. Zweitens sind die Träger von Übergangsamplituden $U(t) = \langle \boldsymbol{x} | e^{-iHt} | \boldsymbol{y} \rangle$ für einzelne freie Teilchen akausal – sie können erst durch in entgegengesetzte Richtung propagierende Antiteilchen außerhalb des Lichtkegels exakt zum Verschwinden gebracht werden (siehe zum Beispiel [PeSc 1995, Abschnitt 2.1]).

Renormierbare Quantenfeldtheorien werden durch eine Lagrangedichte $\mathcal{L}(\phi, \partial_\mu \phi)$, abhängig von verschiedenen Feldern $\phi(x^\mu)$ und ihren Ableitungen, vollständig beschrieben. Massen und Kopplungskonstanten sind Parameter dieser Lagrangedichte. Die die Dynamik beschreibenden Feldgleichungen können aus dem Extremalprinzip $\delta S = \delta \int \mathcal{L} d^4x = 0$ daraus abgeleitet werden. Quantisiert werden solche Feldtheorien entweder kanonisch durch Postulierung von Vertauschungs- beziehungsweise Antivertauschungsrelationen oder durch den Pfadintegralformalismus. In beiden Fällen kann durch Entwicklung in den Kopplungskonstanten des Wechselwirkungsterms eine Störungsreihe konstruiert werden, deren Elemente auf intuitive Weise mit Feynmangraphen korrespondieren – dies vereinfacht eine systematische Buchhaltung. Wechselwirkungen können auch weniger ad hoc durch lokale Eichtransformationen in die Theorie eingeflochten werden. Der dabei notwendig werdende Zusammenhang A_μ, der die Eichinvarianz des $\partial_\mu \phi$-abhängigen Teils in der Lagrangedichte garantiert, wird dann als Träger der Wechselwirkung interpretiert. Dabei stellt sich in Abwesenheit spontaner Symmetriebrechung heraus, dass die Masse der zugehörigen Austauschteilchen exakt verschwinden muss – genau wie dies für Photonen und Gluonen auch der Fall ist.

Das Standardmodell der Elementarteilchenphysik lässt sich als Quantenfeldtheorie mit spontaner Symmetriebrechung beschreiben. Die volle Symmetriegruppe lautet $\mathrm{SU}(3)_c \times \mathrm{SU}(2)_L \times \mathrm{U}(1)_Y$, worin $\mathrm{SU}(3)_c$ die Symmetriegruppe der QCD-Farbladung, $\mathrm{SU}(2)_L$ diejenige des schwachen Isospins und $\mathrm{U}(1)_Y$ die der schwachen Hyperladung ist. Der Grundzustand ist nicht invariant unter $\mathrm{SU}(2)_L \times \mathrm{U}(1)_Y$, sondern nur unter der Untergruppe $\mathrm{U}(1)_e$, der Symmetriegruppe der Quantenelektrodynamik – weshalb das Photon im Gegensatz zu den anderen Eichbosonen des elektroschwachen Sektors $W^\pm$ und Z^0 masselos bleibt. Im einfachsten Fall, dem

Weinberg-Salam-Modell, wird hierfür als Higgs-Sektor ein skalares Isospin-Dublett eingeführt und nebenbei verleiht das Higgs-Teilchen auch den Fermionen über eine einfache Yukawa-Kopplung ihre Masse.

Das Standardmodell setzte sich durch, als auf theoretischer Seite in einer der Sternstunden der Computeralgebra seine Renormierbarkeit [t'Ho 1971a, t'Ho 1971b] und seine Anomaliefreiheit (siehe beispielsweise [ItZu 1993, Abschnitt 6-2-4]) gezeigt werden konnten. Außerdem vermag es auf experimenteller Seite den gesamten wenig systematischen „Teilchenzoo“ und die darin unmotiviert vorkommenden Quantenzahlen (*strangeness*, etc. . .) erfolgreich zu erklären und anschauungsmäßig zu ersetzen. Dennoch ist das Standardmodell nicht vollständig befriedigend. Unschön ist, dass es bisher nicht gelungen ist, die Gravitation mit einzubeziehen und auch die große Anzahl der 25 freien Parameter[1] ist nicht gerade ein Zeichen von Eleganz. Es erklärt ebenso wenig die Zweiteilung der Materie in einen stark wechselwirkenden und einen nicht stark wechselwirkenden Sektor wie die augenfällige Hierarchie der Teilchenmassen. Außerdem wird darin die schwache Hyperladungen etwas ad hoc ins Spiel gebracht um sie an die experimentell beobachtbaren Ladungen anzupassen. Die drittelzahligen Quarkladungen und die Gleichheit von Proton- und Elektronladung bleibt damit unerklärt.

Aufgrund dieser Unzulänglichkeiten und der schweren Zugänglichkeit höchster Energien wachsen die Ansprüche an den Vergleich von Theorie und Experiment stetig an. Ein Paradebeispiel ist die in den letzten Jahren stattgefundene indirekte Eingrenzung der notorisch unzugänglichen (weil nur logarithmisch in Schleifenrechnungen eingehende) Masse des Higgs-Bosons, obwohl seine Entdeckung immer noch aussteht. Die Berechnung von höheren Termen in der Störungsreihe muss automatisiert werden, um sie einerseits überhaupt erst durchführbar und andererseits vertrauenswürdig zu machen. Die danach möglichen Vergleiche mit experimentellen Messungen bergen ein umfangreiches physikalisches Potenzial. Eine kleine Auswahl aus den üblichen Werkzeugen für diese Berechnungen wird im Folgenden beschrieben.

1.2. Standardmethoden zur Berechnung von Feynmandiagrammen

Der Vergleich von Experiment auf der einen und Theorie auf der anderen Seite erfordert seit langem schon die Berechnung von geschlossenen Schleifen in Feynmandiagrammen – mindestens seit 1947, als das anomale magnetische Moment des Elektrons erstmalig gemessen worden ist. Dieser Fall wird seitdem in stets verfeinerter Messung der ebenso verbesserten Schleifenrechnung gegenübergestellt. Auf diesem Gebiet wurden Vertexkorrekturen bis hin zur Vierschleifen-Ordnung berechnet, teilweise mit immensem numerischem Aufwand. Auf anderen Teilgebieten der Teilchenphysik reichen weniger Schleifen aus, um schon an die Grenzen des Messbaren/Errechenbaren zu gelangen. So zum Beispiel in der vollen elektroschwachen Wechselwirkung, wo die Anwesenheit vieler Massenskalen kühne Rechnungen wie in der reinen QED vereitelt – bis vor wenigen Jahren reichte es häufig aus, einfach $\alpha = e^2/4\pi$ bei

[1] Es sind ohne die Gravitationskonstante 25, wenn man davon ausgeht, dass die kürzlich entdeckte Evidenz für Neutrinooszillationen einen kompletten zweiten CKM-Sektor mit drei Massen und vier Mischungswinkeln mitbringt.

$Q^2 = M_Z^2$ zu fixieren. Drei häufig praktizierte Methoden für solche Rechnungen werden in diesem Abschnitt kurz skizziert.

Feynmanparametrisierung

Mit diesem Verfahren können inverse Propagatoren, die wir im Folgenden immer mit $P_i(p^\mu) = p_\mu p^\mu - m_i^2 + i\rho$ notieren wollen, in Schleifenintegralen durch die derselbe Impuls fließt kombiniert werden. Dafür müssen aber zusätzliche Parameter eingeführt werden. Die Ausgangsidentität hierfür ist

$$\frac{1}{P_1 P_2 \cdots P_n} = \int_0^1 dx_1 \cdots dx_n \delta\big(\sum x_i - 1\big) \frac{(n-1)!}{\big(x_1 P_1 + \cdots + x_n P_n\big)}.$$

Sie kann durch Ableitung auch auf beliebige Potenzen $1/P_i^{\nu_i}$ erweitert werden. Nach einer solchen Kombination von Propagatoren unter dem Integral über l^μ ist es oft möglich, über die Lage der kausalen Pole in der komplexen l^0-Ebene Aussagen zu machen. Befinden sie sich beispielsweise im zweiten und vierten Quadranten, so kann durch die Wick-Rotation $l^0 \to il^0$ die Metrik in eine euklidische überführt und die Winkelintegration ausgeführt werden. Das verbleibende Integral über den Abstand $|l_0| =: l$ hat die Gestalt

$$\int_0^\infty dl \frac{l^\beta}{(l^2 + \Delta)^\alpha},$$

was für positives Δ leicht berechnet werden kann, siehe [Ryde 1985, PeSc 1995]:

$$\int_0^\infty dl \frac{l^\beta}{(l^2 + \Delta)^\alpha} = \frac{\Gamma\big(\frac{1+\beta}{2}\big)\Gamma\big(\alpha - \frac{1+\beta}{2}\big)}{2\Delta^{\alpha - \frac{1+\beta}{2}}\Gamma\big(\alpha\big)} = \tfrac{1}{2}\Delta^{\frac{1+\beta}{2}-\alpha} B\big(\tfrac{1+\beta}{2}, \alpha - \tfrac{1+\beta}{2}\big) \tag{1.1}$$

Ist jedoch Δ nicht mehr positiv definit, wie es bei Zweischleifen-Integralen der Fall ist, so ist Vorsicht an der Polstelle des Integranden geboten. Man kann dann

$$\int_0^\infty dl \frac{l^\beta}{(l^2 + \Delta \pm i\varepsilon)^\alpha} = \tfrac{1}{2}(-)^\alpha(\pm i)^{\beta+1}|\Delta|^{\frac{1+\beta}{2}-\alpha} B\big(\tfrac{1+\beta}{2}, \alpha - \tfrac{1+\beta}{2}\big)$$

benutzen.

Im Falle der Zweibeinfunktion bleiben auf Einschleifenebene damit Einfach-Integraldarstellungen übrig, bei Dreibeinfunktionen Zweifach-Integraldarstellungen und so weiter. In vielen Fällen können Feynmanparameter danach ausintegriert werden, besonders wenn wenige oder gar keine Massenskalen vorliegen. Auf Zweischleifenebene bleiben für die planare Dreibeinfunktion im allgemeinen Fall zum Beispiel fünf numerische Integrationen übrig, was je nach Konvergenzverhalten schon zu viel sein kann für eine brauchbare numerische Genauigkeit.

Das obige Integral über l ist aber für negatives $2\alpha - \beta - 1$ divergent. Ein altehrwürdiges Regularisierungsverfahren besteht denn auch in der Einführung eines Abschneideparameters Λ für l. Äquivalent dazu können Propagatoren im Integral derart abgewandelt werden, dass von masselosen Propagatoren ein Propagator mit einer fiktiven großen Masse Λ abgezogen wird (Pauli-Villars Regularisierung). Die Abhängigkeit der greenschen Funktionen von Λ müssen durch Renormierung der Lagrangedichte später so ausgeglichen werden, dass die physikalischen Größen nicht von diesem fiktiven Parameter abhängig sind.

Dimensionale Regularisierung und PO-Zerlegung

Die Regularisierung der Divergenzen mit einem Abschneideparameter hat sich als nicht allzu tragfähig herausgestellt. So verletzt sie insbesondere die Eichinvarianz. Die dimensionale Regularisierung geht davon aus, dass Gleichungen wie (1.1) wörtlich zu nehmen sind – die rechte Seite ist schließlich bis auf isolierte Pole in der komplexen l-Ebene wohldefiniert. Wenn man die Dimension D als freien Parameter betrachtet und die Abweichung von der physikalischen Raumzeit mit $D = 4 - 2\varepsilon$ parametrisiert, dann werden alle Schleifenintegrale zu meromorphen Funktionen mit Polen bei $\varepsilon = 0$ und können in Laurent-Reihen entwickelt werden. Dieses Verfahren erhält die Eichinvarianz, weshalb es sich als Standardmethode der Regularisierung durchgesetzt hat.

Das D-dimensionale Integral gehorcht – wie das gewöhnliche Integral auch – den vier Axiomen:

- **Linearität:**
  Für beliebige komplexe Zahlen α und β gilt

$$\int d^D x \, \big(\alpha f(\boldsymbol{x}) + \beta g(\boldsymbol{x})\big) = \alpha \int d^D x \, f(\boldsymbol{x}) + \beta \int d^D x \, g(\boldsymbol{x}).$$

- **Skalierung:**
  Für beliebiges $s \in \mathbb{C}$ können Skalenfaktoren mithilfe einer Verallgemeinerung der Jacobi-Determinante aus dem Integral herausgezogen werden:

$$\int d^D x \, f(s\boldsymbol{x}) = s^{-D} \int d^D x \, f(\boldsymbol{x}).$$

- **Translationsinvarianz:**
  Da sich das Integrationsintervall über den gesamten homogenen D-dimensionalen Raum erstreckt, gilt für alle Vektoren $\boldsymbol{y}$

$$\int d^D x \, f(\boldsymbol{x} + \boldsymbol{y}) = \int d^D x \, f(\boldsymbol{x}). \tag{1.2}$$

- **Rotationsinvarianz:**
  Der D-dimensionale Raum soll auch isotrop sein. Bei rotationssymmetrischen Integranden $f(\boldsymbol{x}^2)$ kann man das verallgemeinerte Oberflächenintegral daher ausführen:

$$\int d^D x \, f(\boldsymbol{x}^2) = \frac{2\pi^{D/2}}{\Gamma(D/2)} \int_0^\infty dx \, x^{D-1} f(x^2). \tag{1.3}$$

  Hierin ist $\frac{2\pi^{D/2}}{\Gamma(D/2)}$ der Flächeninhalt der Einheitssphäre in D Dimensionen.

Falls der Integrand nicht rotationssymmetrisch ist, kann das verallgemeinerte Oberflächenintegral (1.3) nicht vollständig ausgeführt werden. Eine solche Rotationsvarianz kommt in Schleifenintegralen durch Abhängigkeit von externen Impulsen zustande. Wenn die lineare Hülle der äußeren Impulse aber nicht den gesamten Raum aufspannt, dann kann jener aufgespalten werden in den $D_\parallel$-dimensionalen Parallelraum, der gerade als lineare Hülle der äußeren

Impulse definiert ist, und sein orthogonales Komplement, den $D_\perp$-dimensionalen Orthogonalraum. Dies bezeichnet man als PO-Zerlegung und es gilt natürlich $D_\perp + D_\parallel = D$. Der nicht ganzzahlige Anteil von D steckt dabei ausschließlich in $D_\perp$. Alle Schleifenimpulse lassen sich dann schreiben als

$$\boldsymbol{x} = (x_0, \ldots, x_{D_\parallel - 1}, \boldsymbol{x}_\perp)$$

und für den Anteil $\boldsymbol{x}_\perp$ hat Gleichung (1.3) immer noch Gültigkeit, während die verbleibenden Integrale über den Parallelraum als (Riemann'sche) Integrale ganzzahliger Dimension aufgefasst werden dürfen.

Partielle Integration

Die partielle Integration [ChTk 1981] ist ein Verfahren, in dem komplizierte Feynmandiagramme auf Einfachere zurückgeführt werden. Ausgangspunkt ist die Translationsinvarianz des offenen D-dimensionalen Integrales (1.2) in der Form:

$$\int d^D x \, \frac{\partial}{\partial x_\mu} f(\boldsymbol{x}) = 0. \tag{1.4}$$

Durch Ausführen dieser partiellen Ableitung vor der Integration werden mehrere Integrale in Verbindung gesetzt, wodurch das Aufstellen von Rekursionsformeln möglich wird. Durch geeignetes Zusammenfügen dieser Rekursionsformeln können komplizierte Integrale als eine Linearkombination von wenigen sogenannten Masterintegralen dargestellt werden, die nur einmal berechnet werden müssen.

Als ein Beispiel skizzieren wir wie das im Falle der skalaren Master-Zweischleifenfunktion (Abbildung 1.1 links), funktioniert. Wenn die Propagatoren $1/P_i$, $i = \{1 \ldots 5\}$ mit Exponenten $\nu_i \in \underset{\sim}{\nu} = \{\nu_1 \ldots \nu_5\}$ versehen sind, so lautet das Feynman-Integral

$$J(\underset{\sim}{\nu}) = \int d^D k \, d^D l \, \frac{1}{P_1^{\nu_1} P_2^{\nu_2} P_3^{\nu_3} P_4^{\nu_4} P_5^{\nu_5}}$$

oder, mit einer festgelegten Impulswahl,

$$\begin{aligned} J(\underset{\sim}{\nu}) &= \int d^D k \, d^D l \, \frac{1}{\big((k-p)^2 - m_1^2 + i\rho\big)^{\nu_1} \big((l+p)^2 - m_2^2 + i\rho\big)^{\nu_2}} \\ &\quad \times \frac{1}{\big(k^2 - m_3^2 + i\rho\big)^{\nu_3} \big(l^2 - m_4^2 + i\rho\big)^{\nu_4} \big((l+k)^2 - m_5^2 + i\rho\big)^{\nu_5}}, \end{aligned} \tag{1.5}$$

worin p^μ der durchfließende äußere Impuls ist. Durch lineare Transformationen der Scheifenvariablen l und k kann immer ein beliebiger Propagator frei von allen anderen Impulsen gemacht werden; beispielsweise wird durch $l \to l - k$ der Propagator P_5 nur von l abhängig. Wenn wir dann den Operator $(\partial/\partial l^\mu) l^\mu$ auf den Integranden alleine anwenden, so haben wir genau eine Ableitung der Gestalt (1.4) konstruiert, mit

$$f(\boldsymbol{l}) = \frac{l^\mu}{\big((l-k+p)^2 - m_2^2 + i\rho\big)^{\nu_2} \big((l-k)^2 - m_4^2 + i\rho\big)^{\nu_4} \big(l^2 - m_5^2 + i\rho\big)^{\nu_5}}.$$

Abbildung 1.1.: *Schematische Gleichung als Beispiel partieller Integration. Die Punkte auf den Propagatoren deuten an, dass diese zu quadrieren sind. Die linke Seite ist endlich, das heißt, die Ordnungen ε^{-2}, ε^{-1} und ε^{0} der beiden Graphen auf der rechten Seite heben sich gegenseitig weg.*

Die Propagatoren $1/P_1^{\nu_1}$ und $1/P_3^{\nu_3}$ sind schon als unabhängig von der Integrationsvariable l vor das Integral (1.4) gezogen und weggekürzt worden. Die Ableitung kann nun ausgeführt werden. Der Term $\partial/\partial l^\mu l^\mu$ liefert einen Beitrag D, die Ableitungen von $1/P_i^{\nu_i}$ inkrementieren ν_i und bringen Impulse in den Zähler, die wiederum mit l^μ kontrahiert werden. Unter Ausnutzung der Impulserhaltung an den Vertizes und mit geeigneten Erweiterungen kann man auf diese Art die sogenannte Dreiecksregel für $J(\underset{\sim}{\nu})$ herleiten:

$$\begin{aligned}\Big[-\nu_2 \mathbf{2}^+(\mathbf{5}^- - \mathbf{1}^- + m_1^2 - m_5^2 - m_2^2) - \nu_4 \mathbf{4}^+(\mathbf{5}^- - \mathbf{3}^- + m_3^2 - m_5^2 - m_4^2) \\ + D - 2\nu_5 - \nu_2 - \nu_4 + 2\nu_5 m_5^2 \mathbf{5}^+ \Big] J(\underset{\sim}{\nu}) \quad = \quad 0. \end{aligned} \tag{1.6}$$

Die Operatoren $\mathbf{1}^\pm$, $\mathbf{2}^\pm$, ... darin sind Konvention: Sie inkrementieren oder dekrementieren den entsprechenden Index von $J(\underset{\sim}{\nu})$ und sind vor der Ausführung des Integrals anzuwenden. Ein einfacher Fall ergibt sich, wenn alle Massen m_i verschwinden:

$$J(\underset{\sim}{\nu}) = \frac{1}{D - 2\nu_5 - \nu_2 - \nu_4} \Big[\nu_2 \mathbf{2}^+(\mathbf{5}^- - \mathbf{1}^-) + \nu_4 \mathbf{4}^+(\mathbf{5}^- - \mathbf{3}^-) \Big] J(\underset{\sim}{\nu}).$$

Gilt ferner $\nu_1 = \nu_2 = \nu_3 = \nu_4 = \nu_5 = 1$, so erhält man die einfache Beziehung aus Abbildung 1.1, in der das nichttriviale Diagramm der linken Seite als Linearkombination von zwei trivialen Diagrammen ausgedrückt wird.

Dieses Verfahren ist leicht algorithmisierbar und findet erfolgreiche Anwendungen. Technische Probleme darin bereitet die gegenseitige Kürzung von Divergenzen, die verlangt, dass man geschickt Buch führt über die benötigte Ordnung in ε um möglichst keine überflüssigen Terme zu berechnen. Die auftretenden Rekursionsrelationen beinhalten wie die Dreiecksregel (1.6) nur Ringoperationen zwischen den einzelnen Integranden J. Das Verfahren neigt insgesamt jedoch sehr schnell zu einem explosionsartigen Anschwellen in der Zahl der Terme – ein Arbeitsspeicherbedarf von mehreren hundert Gigabyte ist derzeit keine Seltenheit [BCK 2001]. Aus diesen beiden Gründen ist und bleibt FORM der einzige angemessene Traktor zum Bestellen dieses Feldes. Eine fruchtbare Variation über diesem Thema ist auch die Herleitung von Differenzialgleichungssystemen in den Mandelstam-Variablen, die die verbleibenden Integrale erfüllen, um diese damit bisweilen sogar numerisch auszurechnen [Remi 1997].

Entwicklung nach äußeren Impulsen

Auch in der Entwicklung nach äußeren Impulsen werden komplizierte Feynmanintegrale auf Einfachere zurückgeführt, hier jedoch auf solche, die eine leichter zu handhabende äußere Impulsstruktur besitzen, nämlich im Idealfall auf Vakuumgraphen. Um das Verfahren zu skizzieren gehen wir wieder aus von Gleichung (1.5), und beobachten, dass J insgesamt nicht nur

eine Funktion von $\underset{\sim}{\nu}$ und $\underset{\sim}{m} = \{m_1 \dots m_5\}$ sondern auch vom externen Impulsfluss p^μ ist. Da sich aus p^μ aber nur ein einziges Lorentzskalar p^2 bilden lässt, kann J nur hiervon abhängen. Die Entwicklung von skalaren Funktionen $J(\underset{\sim}{\nu}, \underset{\sim}{m}, p^2)$ nach p^2 muss aus Gründen der Lorentz-kovarianz statt mit der einfachen Impulsableitung $\partial/\partial p_\mu$ mit dem d'Alembert-Operator im Impulsraum $\Box_p = \partial^2/\partial p_\mu \partial p^\mu$ geschehen. Jedes reguläre $J(p^2)$ kann so entwickelt werden:

$$J(p^2) = J(0) + \frac{\big(\Box_p J(p^2)\big)|_{p=0}}{2D} p^2 + \frac{\big(\Box_p^2 J(p^2)\big)|_{p=0}}{8D(D+2)} (p^2)^2 + \mathcal{O}\big((p^2)^3\big). \tag{1.7}$$

Wenn man $\Box_p$ auf das Integral (1.5) anwendet, so erhält man wieder inkrementierte und dekrementierte Exponenten ν_i im Nenner. Man kann das Ergebnis wieder mit den Operatoren $\mathbf{1}^\pm$, $\mathbf{2}^\pm$, ... formulieren:

$$\begin{aligned} &\Box_p J(\underset{\sim}{\nu}, \underset{\sim}{m}, p^2) \\ &= 4\big[(\nu_1 + \nu_2 + 1 - D/2)(\nu_1 \mathbf{1}^+ + \nu_2 \mathbf{2}^+) + \nu_1(\nu_1+1) m_1^2 (\mathbf{1}^+)^2 + \nu_2(\nu_2+1) m_2^2 (\mathbf{2}^+)^2 \\ &\quad + \nu_1 \nu_2 \big((m_1^2 + m_2^2 - m_3^2)\mathbf{1}^+\mathbf{2}^+ - \mathbf{1}^+\mathbf{2}^+\mathbf{3}^-\big)\big] J(\underset{\sim}{\nu}, \underset{\sim}{m}, p^2). \end{aligned}$$

Nach Anwendung von $\Box_p^n$ setzt man $p = 0$ und erhält so eine Linearkombination von Vakuumdiagrammen als Entwicklungsparameter in p^2. Diese Vakuumdiagramme können aber häufig ausgerechnet werden. Viele können analytisch mithilfe von Gamma-Funktionen oder hypergeometrischen Funktionen ausgedrückt werden, einige andere können über partielle Integration mit ersteren verknüpft werden – auf jeden Fall sind sie auch im massiven Fall leichter analytisch in den Griff zu bekommen als die Zweibeinfunktionen [DaT 1992].

Auch dieses Verfahren ist einer algorithmischen Behandlung ausgesprochen zugänglich. Es stößt aber an seine Grenzen, wenn die Reihenentwicklung für kleine Impulsquadrate (1.7) beim Auftreten von Imaginärteilen, also an der ersten kinematischen Schwelle, zusammenbricht. Das Verfahren wurde auch erweitert um die Entwicklung für große k^2 oberhalb der höchsten kinematischen Schwelle [DaST 1993]. Im Bereich der Schwellen selbst hilft häufig Padé-Approximation über die Konvergenzprobleme hinweg [BFT 1993]. Es ist auch leicht einzusehen, dass die Entwicklung in äußeren Parametern rapide mit der Anzahl der Parameter an Komplexität gewinnt.

1.3. Kinematische Abhängigkeiten

Alle skalaren Zweischleifen-Funktionen können in der Form

$$J = \int d^D k \int d^D l \, \frac{1}{P_{l,m_1} \, P_{l+k,m_2} \, P_{k,m_3} \cdots}, \tag{1.8}$$

geschrieben werden mit mindestens drei inversen Propagatoren der Form $P_{l,m_i} = (l+p)^2 - m_i^2 + i\rho$. Hierin ist l ein Schleifenimpuls, und p eine Linearkombination externer Viererimpulse. In den nicht ausgeschriebenen Propagatoren können auch allgemeinere Linearkombinationen wie $P_{\alpha l + \beta k, m_i}$ vorkommen. Es wird gleich darauf verzichtet, einen Index i an die einzelnen ρ anzuhängen, da es möglich ist, sie alle gleichzusetzen.[2]

[2] In [Frin 1996, Abschnitt 6.1] wurde für einen ausgewählten Fall durchexerziert was geschieht wenn verschiedene infinitesimale Imaginärteile mit verschiedenen Relationen $\rho_i < \rho_j$ eingesetzt werden – mit dem wenig verblüffenden Ergebnis, dass beide Verfahren äquivalent sind.

Abgesehen von internen Parametern, also Massen, sind die skalaren Zweischleifen-Funktionen wie alle skalaren greenschen Funktionen wahlweise Funktionen von äußeren Impulskomponenten oder invarianten Skalarprodukten derselben. Deren Anzahl ist durch Impulserhaltung und die Dimensionalität des Raumes D gegeben. Allgemein ist eine n-Beinfunktion wegen Impulserhaltung abhängig von $n-1$ ein- bzw. auslaufenden Impulsen. Daraus errechnet sich leicht die Anzahl der Parameter als die Anzahl der unabhängigen Lorentzskalare $p_{i\mu}p_j{}^{\mu}$, die sich daraus bilden lassen. Man erhält in $D=4$ die nebenstehende Tabelle. Die 0-Bein-Funktionen („Vakuumblasen") hängen wie die 1-Bein-Funktionen („Tadpoles") wegen Impulserhaltung nicht von äußeren Parametern ab. Die Zweibein-Funktionen („Selbstenergien") hängen nur von einem äußeren Impuls ab, man kann sie im Ruhsystem des Teilchens beispielsweise durch dessen Ruhmasse beschreiben. Die Dreibeinfunktionen („Vertexfunktionen") hängen von zwei äußeren Impulsen ab, also von $\sharp=3$ Parametern, z.B. den drei Invarianten, die man aus zwei Impulsen bilden kann und die Vierbeinfunktionen entsprechend von sechs Parametern. Bei höheren Greensfunktionen als den Fünfbeinfunktionen erhöht sich die Anzahl der Parameter jeweils um 4, da die hinzukommenden Impulse als Linearkombinationen der schon vorhandenen gebildet werden können, von denen 4 den Raum aufspannen.

Äußere Beine n	Parameter $\sharp(n)$
0,1	0
2	1
3	3
4	6
$n \geq 4$	$4n-10$

Tabelle 1.1.: *Anzahl $\sharp$ der externen Parameter der n-Bein-Funktionen in $D=4$ Dimensionen*

1.4. Die bisher untersuchten Funktionen („Mainz I und Mainz II")

Die Mainzer Methoden zur Berechnung von Feymandiagrammen beschränken sich auf Zweischleifenfunktionen. Sie haben eine gewisse Allgemeinheit in dem Sinne, dass sie nicht vom Vorhandensein einer sehr eingeschränkten Menge von Massenskalen pro Diagramm ausgehen. Dies macht sie im Prinzip vielseitiger einsetzbar als andere Verfahren – zum Beispiel in der elektroschwachen Wechselwirkung des Standardmodells oder auch in der SU(3)×SU(3) chiralen Störungstheorie, wo zumindest die drei Massenskalen aus dem Oktett der pseudoskalaren Mesonen m_π, m_K und m_η zur Ordnung p^6 in Zweischleifen-Diagrammen mit den effektiven Kopplungen der $\mathcal{L}_2$-Lagrangedichte berechnet werden müssen. Diese drei Massen sind verschieden, aber von der gleichen Größenordnung, und dem muss das Verfahren Rechnung tragen.

Vom topologischen Standpunkt aus können wir uns alle Zweischleifenfunktionen mit beliebiger Anzahl externer Beine konstruieren, indem wir diese an die „Mastertopologie" anheften. Abbildung 1.3 zeigt rechts ein paar ausgewählte Beispiele. Einige davon haben einschlägige Namen. So bezeichnet man beispielsweise II a) als „Sunset-Graphen", I b) als „planare Dreibeinfunktion", II b) als „gekreuzte Dreibeinfunktion" und III c) als „Acnode-Graphen". Die zweidimensionale Darstellung verschleiert bisweilen Symmetrieeigenschaften der Topologie. So macht erst die nebenstehende Abbildung die Symmetrieeigenschaften der gekreuzten Dreibeinfunktion II b)

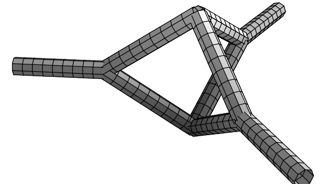


Abbildung 1.2.: *Die gekreuzte Dreibeinfunktion*

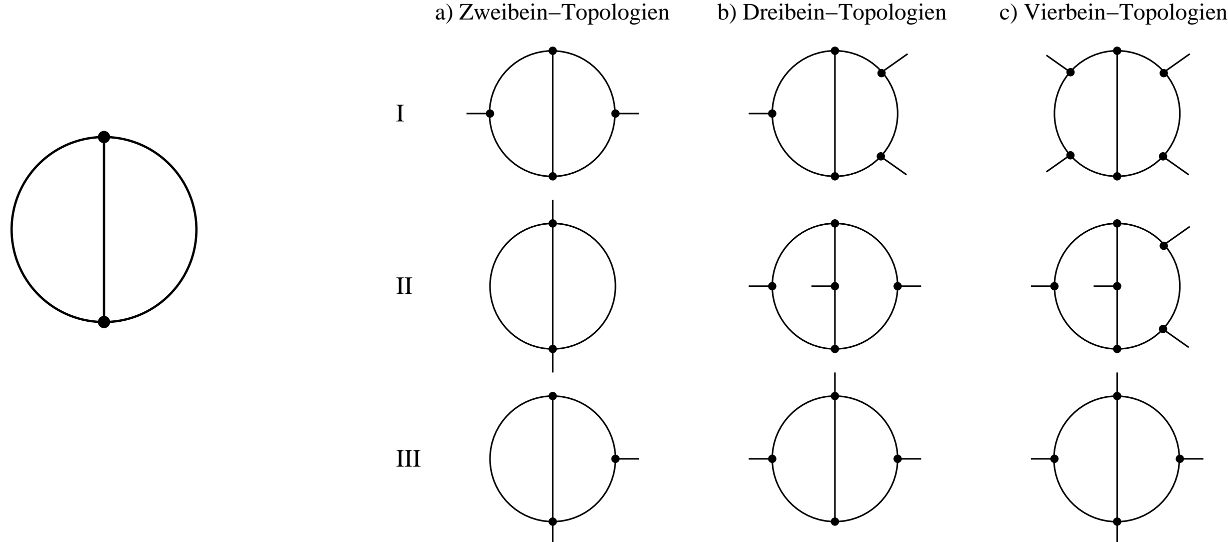


Abbildung 1.3.: *Die Zweischleifen-Mastertopologie und wie daraus systematisch Mehrbeinfunktionen durch Anheften äußerer Beine konstruiert werden können.*

offensichtlich. Ferner ist die Methode des Anheftens äußerer Beine an Mastertopologien unhandlich bei drei oder mehr Schleifen und erschwert das Auffinden der Symmetriefaktoren. Ein systematischeres, graphentheoretisches Vorgehen wie etwa dasjenige von QGRAF [Nogu 1993] ist dieser Ad-hoc-Methode offensichtlich überlegen.

In Abbildung 1.3 fehlen ferner faktorisierende Zweischleifen-Topologien, also solche, die sich als Produkt zweier Einschleifen-Graphen darstellen lassen. Diese lassen sich zwar analytisch ausdrücken; durch den Beitrag ε^{-1} des einen Graphen zum endlichen Beitrag des anderen muss die ε-Entwicklung allerdings um eine Ordnung weiter getrieben werden als dies für einfache Einschleifen-Topologien nötig ist. Die Behandlung innerhalb des alten *xloops* skizzieren [Brue 1997, Fran 1997].

Die Zweibeinfunktionen („Mainz I“)

Die Zweibeinfunktionen auf Zweischleifen-Niveau werden von *xloops* nach einem in [Krei 1991] skizzierten Verfahren berechnet. Es findet Anwendung sowohl bei skalaren als auch bei Tensorintegralen [Krei 1993]. Darin wird von den Schleifenimpulsen l und k zunächst deren Parallelraumkomponente l_0 und k_0 abgespalten und die Orthogonalraumkomponenten $l_\perp$ und $k_\perp$ in $D-1$ Dimensionen sphärisch symmetrisiert. Bis auf einen relativen Winkel ϑ zwischen $l_\perp$ und $k_\perp$ kann man die Winkelintegrationen ausführen. Für die ϑ-Integration geht dies jedoch nur in ganzzahliger Dimension D. Falls das Integral nicht endlich ist werden hierzu geeignete Abzugsterme aufgesucht, die das Integral endlich machen – die divergenten Teile lassen sich analytisch berechnen. Danach kann man $D=4$ setzen und stets sowohl die ϑ-, als auch die $l_\perp$- und $k_\perp$-Integrationen ausführen. Die verbleibenden beiden Integrationen sind mit numerischen Methoden zugänglich [Fran 1997].

Dieses Verfahren ist besonders dann sehr attraktiv, wenn viele verschiedene Massenskalen im Integral vorkommen. Sind alle Massen gleich oder verschwinden sehr viele Massen exakt, dann gibt es überlegenere Methoden, die ohne Zweifach-Integraldarstellung auskommen.

Die Dreibeinfunktionen („Mainz II")

Auf Zweischleifen-Niveau sind hier vor allem die planare und die gekreuzte Vertexfunktion (-◁[und -◁X) interessant. Das verwendete Verfahren [Krei 1992b] beruht wieder darauf, den zweidimensionalen Parallelraum abzutrennen. Die Orthogonalraumvariablen werden darin üblicherweise als $l_\perp = \sqrt{s}$ und $k_\perp = \sqrt{t}$ geschrieben. In $D = 4$ werden sie von zwei Winkeln begleitet, die beide sofort aufintegriert werden können, einer davon trivial. Hiernach werden die inversen Propagatoren in den Variablen l_1 und k_1 linearisiert, indem man die Ersetzungen $l_0 \to l_0 + l_1$ und $k_0 \to k_0 + k_1$ vornimmt – wir werden dieser Linearisierung ab Seite 26 noch mehrmals begegnen. Dies macht den Integranden einer Integration in l_1 und k_1 mithilfe des Cauchy'schen Residuensatzes zugänglich. Als ein Nebeneffekt werden dabei die Integrationsgrenzen in l_0 und k_0 in Abhängigkeit äußerer Impulsvariablen in endliche Dreiecke transformiert (siehe nächster Abschnitt). Sowohl die s- als auch die t-Integration können noch analytisch ausgeführt werden. Die verbleibende Zweifach-Integraldarstellung wird wieder numerisch integriert, wobei das Integrationsgebiet nun durch äußere Impulskomponenten parametrisierte Dreiecke in der l_0-k_0-Ebene sind.

Auch dieses Verfahren wird in Anwesenheit vieler verschiedene Massenskalen am attraktivsten [Kili 1996, Frin 1996, Frin 2000]. Die numerischen Schwierigkeiten selbst im planaren skalaren Fall sind jedoch immer noch Gegenstand einer Untersuchung.

1.5. Beschränkte Integrationsgebiete nach Residuenintegration?

Bei dem Mainzer Verfahren für Dreibeinfunktionen wurden zwei Residuenintegrationen durchgeführt und wir werden in Kapitel 2 sehen, dass bei den Vierbeinfunktionen sogar vier Integrationen mit dem Cauchy'schen Residuensatz erledigt werden können. Beide Male werden die Integrationsgrenzen in anderen Integrationsvariablen stark eingeschränkt. Es wurde behauptet, dass die Gebiete in denen danach die numerische Integration durchgeführt wird, immer endlich sein müssen. Wir werden in diesem Abschnitt sehen, dass dies kein Zufall ist, andererseits aber auch nicht zwingend so sein muss.

Die Residuenintegration wird immer in solchen Variablen durchgeführt, in denen die inversen Propagatoren P_i linearisiert worden sind. Bei dieser Linearisierung werden die linearisierten Variablen stets multipliziert mit einer weiteren Impulsvariable, die wir die „zugeordnete" Variable nennen und mit einer Tilde markieren wollen: Enthält das noch nicht linearisierte P den Term $\tilde{l}^2 - l^2$, so führt die Ersetzung $\tilde{l} \to \tilde{l} + l$ diesen über in $\tilde{l}^2 + 2\,\tilde{l}\,l$. Bei der Dreibeinfunktion entsprach l_0 der zugeordneten Variable von l_1 und k_0 derjenigen von k_1. Die zugeordneten Variablen $\tilde{l}$ und $\tilde{k}$ werden zunächst nicht weiter ausintegriert – ihr Vorzeichen zusammen mit

dem Vorzeichen des Imaginärteils von P bestimmt, ob das Residuum beiträgt oder nicht. Anders herum formuliert verschwindet das Integral, wenn die zugeordnete Variable das falsche Vorzeichen hat.

Um genauer einzusehen, unter welchen Umständen die Gebiete in den zugeordneten Variablen endlich werden, brauchen wir zweierlei: Erstens einen Vorschlag für einen Mechanismus, wie außerhalb endlicher Gebiete die Terme konspirieren, so dass sie sich zu Null addieren, und zweitens eine Art Parametrisierung dieses Mechanismus. Die folgenden Überlegungen basieren auf einer in [FKT 1997, Seite 15] skizzierten Idee.

Der Mechanismus besteht darin, dass bei einer Residuenintegration eines Terms der Form $1/(P_1 P_2 P_3 \cdots)$ immer wieder doppelt vorkommende Terme auftreten: Der Satz A.4 über die Residuensumme sagt gerade, dass die Summe über alle Residuen verschwinden muss. Also sind nicht alle Residuen voneinander unabhängig. Wie dieser Mechanismus nun bei zwei linearisierten Variablen angreift, klärt das folgende Lemma:

Lemma 1.1 *Betrachte ein Integral der Form*

$$J(z_1, z_2, z_3) := \int_{-\infty}^{\infty} dk \int_{-\infty}^{\infty} dl \, \frac{1}{(k - z_1)(l - z_2)(k + l + z_3)} \tag{1.9}$$

mit komplexen Koeffizienten $z_i = x_i + iy_i$. *J ist nur dann von Null verschieden, wenn alle Vorzeichen von y_i gleich sind.*

Wir bemerken zunächst, dass man nach Substitution von $k \to -k - l$ das Integral mit der Vertauschung $z_1 \leftrightarrow z_3$ zurückerhält. Ebenso kann man beliebige andere Vertauschungen herbeiführen: Das Integral J ist symmetrisch unter allen $z_i \leftrightarrow z_j$.

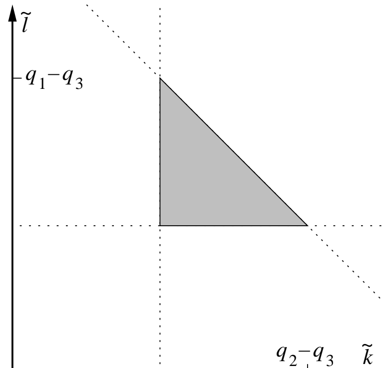
Zum Beweis des Lemmas integriert man J unter Zuhilfenahme von Satz A.4 und erhält:

$$\begin{aligned} J &= \{\theta(y_2) - \theta(-y_3)\} \int_{-\infty}^{\infty} dk \, \frac{2\pi i}{(k - z_1)(k + z_2 + z_3)} \\ &= \{\theta(y_2) - \theta(-y_3)\}\{\theta(y_1) - \theta(-(y_2 + y_3))\} \frac{(2\pi i)^2}{z_1 + z_2 + z_3}. \end{aligned} \tag{1.10}$$

Man überprüft leicht anhand einer Wahrheitstabelle, dass der Vorfaktor aus θ-Funktionen genau dann 1 ist, wenn die Vorzeichen der y_i alle gleich sind und in den übrigen Fällen verschwindet. □

Die inversen Propagatoren P_i sind bei unseren Methoden zur Drei- und Vierbeinfunktion – nach ihrer Linearisierung und Ausklammern von Faktoren linear in den zugeordneten Impulsen $\tilde{k}$, $\tilde{l}$ und Kombinationen äußerer Impulskomponenten q_i – genau von der Form aus (1.9). Die Imaginärteile der z_i lauten dann

$$\begin{aligned} y_1 &= -\rho \Big/ \left(\tilde{k} + q_1\right) \\ y_2 &= -\rho \Big/ \left(\tilde{l} + q_2\right) \\ y_3 &= +\rho \Big/ \left(\tilde{k} + \tilde{l} + q_3\right) \end{aligned}$$



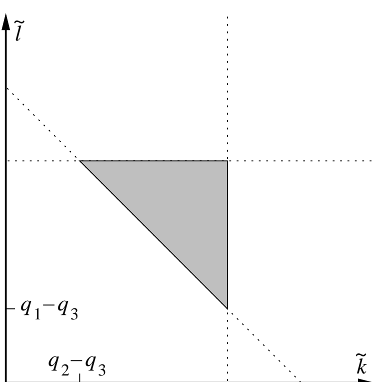


Abbildung 1.4.: *Nach Residuenintegration verbleiben nur endliche Dreiecke in den zugeordneten Impulsen.*

wobei q_i hier abkürzend für beliebige Linearkombinationen externer Impulskomponenten stehen.

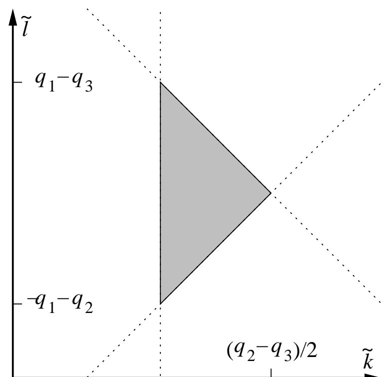
$y_i > 0$	$y_i < 0$
$\tilde{k} < -q_1$	$\tilde{k} > -q_1$
$\tilde{l} < -q_2$	$\tilde{l} > -q_2$
$\tilde{k} + \tilde{l} > -q_3$	$\tilde{k} + \tilde{l} < -q_3$

Tabelle 1.2.: *Die zwei möglichen Kombinationen von Bedingungen*

Da laut Lemma 1.1 für einen Beitrag alle Vorzeichen der Imaginärteile y_i gleich sein müssen, können nur zwei Kombinationen vorkommen: entweder alle positiv oder alle negativ. Man kann dies direkt in Bedingungen an $\tilde{k}$ und $\tilde{l}$ übersetzen und findet so die beiden in der nebenstehenden Tabelle aufgeführten Möglichkeiten. Diese entsprechen *endlichen* Dreiecken, wie sie in Abbildung 1.4 graphisch dargestellt sind. Das Gebiet kann zwar zu einem Punkt entarten, wenn die Diagonale $\tilde{k} + \tilde{l} = -q_3$ den Kreuzungspunkt der Horizontalen $\tilde{l} = -q_2$ mit der Vertikalen $\tilde{k} = -q_1$ schneidet, es kann jedoch nicht ganz verschwinden. Falls im linken Dreieck die Diagonale über den Kreuzungspunkt hinaus verschoben wird erhält man das rechte Dreieck und umgekehrt. Unbeschränkte Gebiete, die z.B. nach mindestens einer Seite hin keine Beschränkung haben, können also nicht auftreten.

Dies bleibt auch dann noch richtig, wenn eine beliebige Anzahl weiterer linearisierter Propagatoren $(k - z_i)^{-1}$ und $(l - z_j)^{-1}$ hinzugefügt werden. Nach geeigneten Partialbruchzerlegungen kann man den Integranden dann immer als Summe von Termen der Art (1.9) schreiben. Die einzelnen Imaginärteile haben ja auch immer die Gestalt $y_i = -\rho/(\tilde{k} + q_i)$ beziehungsweise $y_j = -\rho/(\tilde{l} + q_j)$ mit (reellwertigen) Kombinationen äußerer Impulskomponenten im Nenner. Gekreuzte Topologien hingegen sind notorisch schwieriger und es lohnt sich, sie besonders sorgfältig zu untersuchen. Wenn zwei Propagatoren beide Integrationsvariablen l und k in linearisierter Form enthalten, so kann man durch eine lineare Verschiebung stets dafür sorgen, dass ein inverser Propagator linear in $l + k$ ist, ein anderer linear in $l - k$ und alle weiteren nur entweder l oder k enthalten. Wir interessieren uns also für die θ-Funktionen, die in der Integration von

$$\hat{J}(z_1, z_2, z_3) := \int_{-\infty}^{\infty} dk \int_{-\infty}^{\infty} dl \, \frac{1}{(k - z_1)(l - k - z_2)(k + l + z_3)}$$



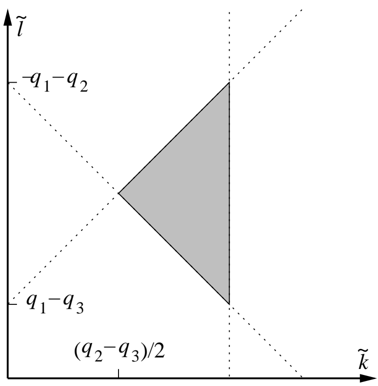


Abbildung 1.5.: *Nach Residuenintegration von gekreuzten Funktionen können die abgebildeten endlichen Gebiete zurückbleiben. Dies muss aber nicht immer so sein; siehe Text.*

erzeugt werden. Wie zuvor im planaren Fall kann man dieses Integral leicht mithilfe von Satz A.4 ausführen:

$$\begin{aligned}\hat{J} &= \{\theta(y_2)-\theta(-y_3)\}\int_{-\infty}^{\infty} dk\,\frac{2\pi i}{(k-z_1)(2k+z_2+z_3)}\\ &= \{\theta(y_2)-\theta(-y_3)\}\{\theta(y_1)-\theta(-(y_2+y_3))\}\frac{(2\pi i)^2}{2z_1+z_2+z_3}.\end{aligned}$$

Die Struktur der θ-Funktionen ist aber im Vergleich mit (1.10) dieselbe geblieben. Daher müssen auch alle Vorzeichen der Imaginärteile wieder gleich sein, um beitragen zu können. Übersetzt man dies auf die zugeordneten Impulse $\tilde{l}$ und $\tilde{k}$, so findet man, dass nur die endlichen Integrationsgebiete aus Abbildung 1.5 übrigbleiben.

Wenn von vornherein alle kausalen ρ gleich gesetzt werden, kann es allerdings dazu kommen, dass der kausale Faktor ρ vor der zweiten Anwendung des Residuensatzes in einem der inversen Propagatoren verschwunden ist. Dann ist ein z_i rein reell und unser Mechanismus bricht zusammen. Wenn $y_i = 0$, dann ist die entsprechende θ-Funktion in (1.10) laut Satz A.6 zu ersetzen durch einen Faktor $1/2$, so dass keine einschränkende Bedingung mehr vorliegt. Die Endlichkeit der Integrationsgebiete in den zugeordneten Impulsvariablen wird dadurch kompromittiert. Bei der gekreuzten Dreibeinfunktion ist dies der Fall, wobei dort aber durch eine geeignete Koordinatentransformation Terme ausserhalb endlicher Dreiecke wieder gegeneinander weggekürzt werden können, so dass die Integrationsgebiete doch wieder endlich werden. Auch bei den Vierbeinfunktionen treten immer wieder solche Fälle auf, die dann durch nichttriviale Kürzungen – allerdings schon auf dem Niveau der θ-Funktionen – verschwinden. Eine solche nichttriviale Kürzung sieht sehr häufig wie folgt aus:

$$\begin{aligned}&\big(\theta(f_1(\tilde{k},\tilde{l}))-\tfrac{1}{2}\big)\big(\theta(f_3(\tilde{k},\tilde{l}))-\tfrac{1}{2}\big)-\big(\theta(f_1(\tilde{k},\tilde{l}))-\tfrac{1}{2}\big)\big(\theta(f_4(\tilde{k},\tilde{l}))-\tfrac{1}{2}\big)\\ &-\big(\theta(f_2(\tilde{k},\tilde{l}))-\tfrac{1}{2}\big)\big(\theta(f_3(\tilde{k},\tilde{l}))-\tfrac{1}{2}\big)+\big(\theta(f_2(\tilde{k},\tilde{l}))-\tfrac{1}{2}\big)\big(\theta(f_4(\tilde{k},\tilde{l}))-\tfrac{1}{2}\big)\\ &= \big(\theta(f_1(\tilde{k},\tilde{l}))-\theta(f_2(\tilde{k},\tilde{l}))\big)\big(\theta(f_3(\tilde{k},\tilde{l}))-\theta(f_4(\tilde{k},\tilde{l}))\big),\end{aligned}$$

was manifest frei ist von allen halbzahligen Gewichten. Bei der gekreuzten Vierbeinfunktion kann es allerdings zu Gebieten kommen, die einen sich ins Unendliche erstreckenden

Streifen in der Ebene der zugeordneten Variablen bilden. Ob diese wie bei der Dreibeinfunktion durch geeignete Transformationen zum Verschwinden gebracht werden können, ist noch unklar. Der oben skizzierte Mechanismus kann hierfür jedenfalls nicht alleine verantwortlich sein.

2. Die skalaren Zweischleifen-Vierbeinfunktionen

Beware of bugs in the above code;
I have only proved it correct, not tried it.
Donald E. Knuth, in einer Notiz an Peter van Emde Boas

Über die Vierbeinfunktionen ist auf Zweischleifen-Ebene bisher recht wenig bekannt. Kürzlich wurde mit einer Feynmanparametrisierung und anschließender Integration der Feynmanparameter mithilfe einer Mellin-Barnes-Darstellung für Summen ein analytisches Ergebnis für sowohl die planare als auch die gekreuzte skalare Box gefunden [Smir 1999, Tau 1999]. Diese Darstellungen gelten aber nur dann, wenn alle internen und externen Teilchen masselos sind, das Diagramm also nur von zwei Parametern abhängt. Das Verfahren konnte auf Tensorintegrale erweitert werden [AGORT 2000, SmiVe 1999] und sogar auf spezielle massive Fälle, allerdings nur mit einer einzigen Massenskala [Smir 2000, Smir 2001].

In der chiralen Störungstheorie ist das als Acnode-Graph bekannte Diagramm von besonderem Interesse, sobald zur Ordnung p^6 Zweischleifenintegrale bestehend aus $\mathcal{L}_2$-Vertizes zu den effektiven Vertizes aus $\mathcal{L}_6$ und den Einschleifenintegralen bestehend aus $\mathcal{L}_4$- und $\mathcal{L}_2$-Vertizes beitragen. Er ist von besonderer Bedeutung in den Prozessen $\gamma\gamma \to \pi^0\pi^0$ und $\eta \to \pi^0\gamma\gamma$, da diese zur Ordnung p^2 verschwinden. Wenn aber wie in SU(2)×SU(2) chiraler Störungstheorie alle internen Teilchen gleiche Massen m_π haben, dann wird die Parametermannigfaltigkeit stark eingeschränkt und die Rechnungen sind weitgehend analytisch durchführbar [Bier 2000]. Das in diesem Kapitel verfolgte Verfahren für skalare Vierbeinfunktionen basiert auf einer Überlegung von Dirk Kreimer [Krei 1994] und kommt ganz ohne Nebenbedingungen in der Wahl der bis zu zwölf Parameter aus. Es ist daher viel allgemeiner als die anderen Verfahren, erfordert aber erhebliche zusätzliche Arbeit, wenn man den numerischen Vergleich anstrebt. Dies liegt daran, dass die anderen Verfahren nur Eckpunkte im Parameterraum abdecken können, bei dem hier beschriebenen genau an diesen Eckpunkten aber numerische Instabilitäten auftreten, die per Hand behoben werden müssen.

2.1. Die Vierbeinfunktion („Mainz III“)

Das Integral

$$J = \int d^D k \int d^D l \, \frac{1}{P_{m_1}(l)\, P_{m_2}(l+k)\, P_{m_3}(k) \cdots}, \qquad (1.8)$$

hängt jetzt von sechs voneinander unabhängigen äußeren kinematischen Variablen ab. Eine mögliche Wahl sind die Mandelstam-Variablen s, t und u zusammen mit den Massen m_i, $i \in \{1 \dots 4\}$ der äußeren Teilchen und der Bedingung $s + t + u = \sum_i m_i^2$. Für die Integrationen wird jedoch auch hier wieder ein spezielles Bezugssystem gewählt werden müssen, so dass sechs äußere Impulskomponenten explizit festzulegen sind. Eine mögliche kinematische Konfiguration des skalaren Acnode-Graphen lautet

$$J_{\triangledown} = \int d^4k \int d^4l \, \frac{1}{P_{m_1}(l)\, P_{m_2}(l+p_1)\, P_{m_3}(k+p_2-p_3)\, P_{m_4}(k-p_3)\, P_{m_5}(k+l)}$$

$$\text{mit} \quad \begin{cases} p_1^\mu = (q_1, q_x, 0, 0) \\ p_2^\mu = (q_2, -q_x, 0, 0) \\ p_3^\mu = (q, p_x, p_y, 0) \, . \end{cases}$$

Die Struktur der Vierbeintopologien lässt sich wie in Abbildung 2.2 klassifizieren. Hierin sind in der ersten Zeile die Topologien mit sieben internen Propagatoren aufgelistet, in den darauffolgenden Zeilen II und III jeweils diejenigen, die durch Streichung eines Propagators aus der Zeile darüber entstehen. Die planare Box I b) aus Abbildung 2.2 ist identisch mit I c) in Abbildung 1.3. Die gekreuzte Box I a) in Abbildung 2.2 findet sich in 1.3 als II c) wieder. Schließlich ist III b) in Abbildung 2.2 der aus 1.3 als III c) bekannte Acnode-Graph. Auch hier wieder verschleiert die zweidimensionale Darstellung Symmetrien, wie nebenstehende dreidimensionale Version des Graphen II a) aus 2.2 deutlich macht. In der letzten Spalte finden sich übrigens die planare und gekreuzte Dreibeinfunktion wieder und auch eine faktorisierende Topologie. Die ab diesem Abschnitt entwickelten Methoden werden sich prinzipiell auf all diejenigen Graphen anwenden lassen, bei denen jede geschlossene Schleife mindestens drei Propagatoren enthält. Das sind gerade die nicht grau hinterlegten in Abbildung 2.2. Über sie ist in allgemeinen Fällen bisher am wenigsten bekannt. Sie sind natürlich für Streuprozesse interessant, aber auch für 3-Teilchen-Zerfälle.

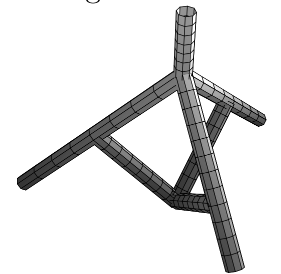


Abbildung 2.1.: *Die Topologie II a)*

Integration der vier „inneren“ Schleifenvariablen

Die in [Krei 1994] vorgeschlagene Methode für Vierbeinfunktionen beruht auf der Idee, vier der Schleifenintegrationen mithilfe des Residuensatzes (Satz A.3) auszuführen. Die Anwendung des Residuensatzes wird sehr vereinfacht, wenn die inversen Propagatoren P_i linear sind in der zu integrierenden Variablen, da dann die Polstellen trivial aufzufinden sind und keine Quadratwurzeln eingeführt werden, die die weitere Integration verkomplizieren würden.

Diese Linearisierung kann wegen der Signatur der Minkowskimetrik in zwei von vier Variablen sofort durchgeführt werden. Wählen wir l_1 und k_1 als die zu linearisierenden Variablen und wenden die Transformation

$$l_0 \longrightarrow l_0 + l_1, \qquad k_0 \longrightarrow k_0 + k_1 \tag{2.1}$$

an, so wird aus einem Propagator P_l

$$\begin{aligned} (l+p)^2 - m^2 + i\rho &= (l_0+p_0)^2 - (l_1+p_1)^2 - (l_2+p_2)^2 - (l_3+p_3)^2 - m^2 + i\rho \\ &\longrightarrow (l_0+p_0)^2 + 2\, l_1\, (l_0+p_0-p_1) - p_1^2 - (l_2+p_2)^2 - (l_3+p_3)^2 - m^2 + i\rho. \end{aligned} \tag{2.2}$$

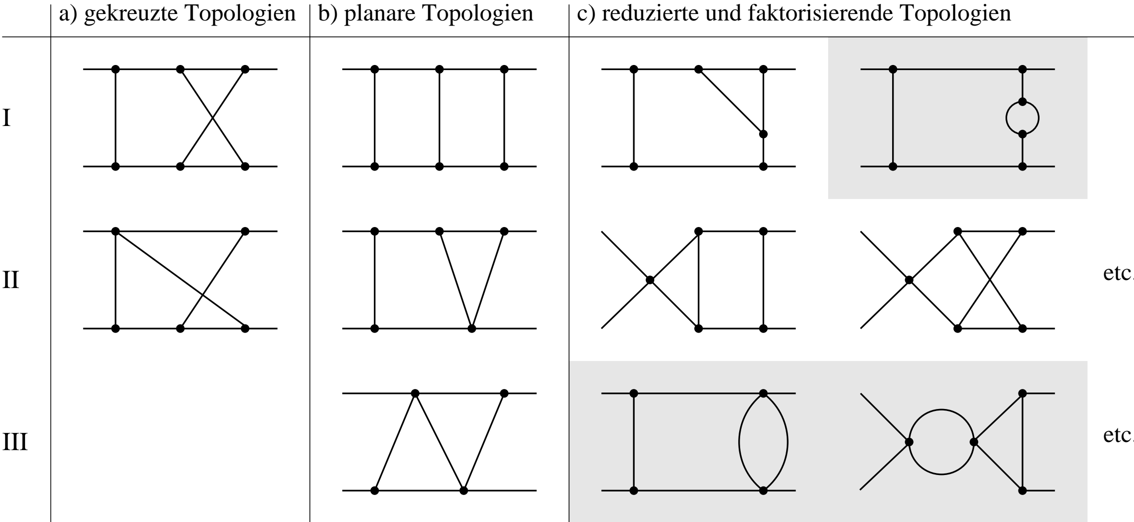


Abbildung 2.2.: *Mögliche skalare Vierbeintopologien. Die grau hinterlegten sind nicht mit in dieser Arbeit behandelten Methoden zugänglich.*

Darin bezeichnen p_i beliebige äußere Impulskomponenten. Diese Transformation darf sofort angewendet werden da die nicht grau hinterlegten skalaren Topologien in Abbildung 2.2 absolut konvergieren. Wir nennen wieder l_1 und k_1 die linearisierten Variablen und l_0 und k_0 die zugeordneten Variablen.

Schließt man den Integrationsweg in der oberen komplexen Halbebene und benutzt den Residuensatz um die l_1- und k_1-Integrationen auszuführen, so erhält man Bedingungen an die Integrationsgebiete der zugeordneten Schleifenvariablen l_0 und k_0. Diese Bedingungen stammen von den Vorzeichen der Imaginärteile der Pole in den linearisierten Variablen, die von den zugeordneten Variablen abhängig sind. Das Residuum trägt nur bei, wenn die Polstelle im Integrationsweg liegt, wenn der Imaginärteil also positiv ist.[1] Diese Bedingungen betreffen lediglich die zugeordneten Variablen l_0- und k_0, da keine weiteren Schleifenvariablen (z.B. l_2, etc.) im l_1-linearen Term in (2.2) auftauchen. Wir werden im nächsten Unterabschnitt auf die Bedingungen zurückkommen und sie klassifizieren.

Da die Integrationen über die Variablen l_2, l_3, k_2 und k_3 immer noch unbeschränkt sind, liegt es nahe, auch bei zweien davon den Residuensatz anzuwenden. Wie erwähnt wird dies erheblich erleichtert, wenn die inversen Propagatoren zuvor linearisiert werden können. Die Linearisierung in (2.1) war möglich wegen der Signatur der Lorentz-Metrik, insbesondere wegen des relativen Vorzeichens zwischen der 0- und der 1-Komponente im Skalarprodukt. Da der Orthogonalraum eindimensional ist, kann er wegen Lorentz-Invarianz o.B.d.A. in eine der Koordinatenachsen gelegt werden, zum Beispiel $l_\perp = l_3$ und $k_\perp = k_3$. Dadurch verschwinden die p_3 in Gleichung (2.2) und es wird möglich im Falle der Vierbeinfunktionen ein weiteres relatives Vorzeichen einzuführen und dann genau wie oben zu linearisieren. Dies ermöglicht der folgende

[1] Schließt man statt dessen den Integrationsweg in der unteren komplexen Halbebene, so gibt es einen Beitrag dann und nur dann, wenn der Imaginärteil negativ ist. Dies schlägt sich gemäß (A.4) aber nur in einem allgemeinen Vorzeichen nieder, welches dem umgekehrten Umlaufsinn entspricht. Dies macht die freie Wahl des Integrationsweges manifest und wir werden im Folgenden immer in der oberen Halbebene schließen.

Satz 2.1 (Kreimer-Rotation) *Seien $P_q = q_\mu q^\mu - m^2 + i\rho$ inverse Propagatoren mit Impulsfluss q^μ (äußere* und *innere Impulse) und j der Lorentzindex einer Orthogonalraumkomponente. Dann gilt:*

$$\begin{aligned}
&\int_{-\infty}^{+\infty} dl_j \int_{-\infty}^{+\infty} dk_j \, \frac{1}{P_l\big(l_j^2\big)\, P_{l+k}\big((l_j+k_j)^2\big)\, P_k\big(k_j^2\big)\cdots} \\
&= \; - \int_{-\infty}^{+\infty} dl_j \int_{-\infty}^{+\infty} dk_j \, \frac{1}{P_l\big(-l_j^2\big)\, P_{l+k}\big(-(l_j+k_j)^2\big)\, P_k\big(-k_j^2\big)\cdots}.
\end{aligned} \tag{2.3}$$

Beweis: Im Orthogonalraum mischen die Schleifenvariablen nicht mit äußeren Impulsen; die inversen Propagatoren sind also von der Form

$$\begin{aligned}
P_l\big(l_j^2\big) &= l_j^2 + (\textit{reelle Zahl}) + i\rho, \\
P_{l+k}\big((l_j+k_j)^2\big) &= (l_j+k_j)^2 + (\textit{reelle Zahl}) + i\rho, \\
P_k\big(k_j^2\big) &= k_j^2 + (\textit{reelle Zahl}) + i\rho.
\end{aligned}$$

Daher liegen die Pole des Integranden in den komplexen l_j, k_j und $(l_j + k_j)$-Ebenen alle im ersten und dritten Quadranten. Da die Integranden für große l_j, k_j und $l_j + k_j$ schnell genug abfallen, liegt es nahe, eine Rotation um $\pi/2$ im Uhrzeigersinn zu unternehmen. Dies läuft gerade auf die gewünschte Änderung des Vorzeichens in (2.3) heraus. Gemischte inverse Propagatoren des Typs P_{l+k} stehen dieser Rotation dabei im Wege, da die Nullstellen in den komplexen k_j- oder l_j-Ebenen alleine betrachtet nicht auf den ersten oder dritten Quadranten beschränkt sind. Um die Rotation ausführen zu können müssen *alle* Pole in l_j, k_j und $l_j + k_j$ also zunächst in dieselben Quadranten verschoben werden. Wegen der Symmetrie des Integranden kann man sich auf den ersten Quadranten in der reellen k_j-l_j-Ebene beschränken und schreiben:

$$\begin{aligned}
&\int_{-\infty}^{+\infty} dl_j \int_{-\infty}^{+\infty} dk_j \, \frac{1}{P_l\big(l_j^2\big)\, P_{l+k}\big((l_j+k_j)^2\big)\, P_k\big(k_j^2\big)\cdots} \\
&= 2 \int_0^{+\infty} dl_j \int_0^{+\infty} dk_j \left(\frac{1}{P_l\big(l_j^2\big)\, P_{l+k}\big((l_j+k_j)^2\big)\, P_k\big(k_j^2\big)\cdots} \right. \\
&\qquad\qquad \left. + \frac{1}{P_l\big(l_j^2\big)\, P_{l+k}\big((l_j-k_j)^2\big)\, P_k\big(k_j^2\big)\cdots} \right)
\end{aligned}$$

Nun reparametrisieren wir diesen Quadranten mit der Substitution $l_j^2 \to u\, v^2$ und $k_j^2 \to (1-u)\, v^2$:

$$\begin{aligned}
&\frac{1}{2} \int_0^1 \frac{du}{\sqrt{u\,(1-u)}} \int_0^{+\infty} v\, dv \left(\frac{1}{P_l\big(u\, v^2\big)\, P_{l+k}\big((\sqrt{u}+\sqrt{1-u})^2\, v^2\big)\, P_k\big((1-u)\, v^2\big)\cdots} \right. \\
&\qquad\qquad \left. + \frac{1}{P_l\big(u\, v^2\big)\, P_{l+k}\big((\sqrt{u}-\sqrt{1-u})^2\, v^2\big)\, P_k\big((1-u)\, v^2\big)\cdots} \right)
\end{aligned} \tag{2.4}$$

Die Faktoren u, $(1-u)$ und $(\sqrt{u} \pm \sqrt{1-u})^2$ sind positiv im Integrationsintervall, so dass die Polstellen in der komplexen v-Ebene alle im ersten und dritten Quadranten liegen. Daher

kann man nun den Integrationsweg der v-Integration um den vierten Quadranten schließen, wobei die Jacobi-Determinante ihr Vorzeichen wechselt:

$$\frac{1}{2}\int_0^1 \frac{du}{\sqrt{u\,(1-u)}}\int_0^{+\infty} -v\,dv \left(\frac{1}{P_l\big(-u\,v^2\big)\,P_{l+k}\big(-(\sqrt{u}+\sqrt{1-u})^2\,v^2\big)\,P_k\big(-(1-u)\,v^2\big)\cdots} + \frac{1}{P_l\big(-u\,v^2\big)\,P_{l+k}\big(-(\sqrt{u}-\sqrt{1-u})^2\,v^2\big)\,P_k\big(-(1-u)\,v^2\big)\cdots}\right).$$

Invertieren der Transformationen, die zu (2.4) geführt haben, liefert die Behauptung des Satzes. □

Bei gekreuzten Funktionen mit zwei gemischten Propagatoren können auch Propagatoren mit P_{l-k} auftreten. Es ist jedoch klar, dass alle Schritte in obigem Beweis auch dann noch Gültigkeit haben. Er lässt sich sogar mit $P'\big((al_j+bk_j)^2\big)$ mit beliebigem festem a und b genauso herleiten.

Ist o.B.d.A. $j=3$, so wirkt dies wie eine Abänderung der üblichen Minkowski-Metrik $(+,-,-,-)$ in eine $(+,-,-,+)$-Metrik. Daher der Name „Rotation“ in Analogie zur „Wick-Rotation“ (siehe z.B. [ItZu 1993]), bei der in ähnlicher Weise die Zeitachse rotiert wird um von einer Minkowski'schen in eine euklidische Metrik zu transformieren. Die hier vorgestellte Transformation ist jedoch grundlegend verschieden von der Wick-Rotation, bei der eine analytische Fortsetzung am Ende der Rechnung gefunden werden muss, um die Greensfunktion für beliebige äußere Impulse zu erhalten. Der Vorzeichenwechsel in (2.3) jedoch ist eine einfache analytische Identität, die in jeder Orthogonalraumvariablen einzeln Gültigkeit besitzt.[2]

Nach Ausführen der Residuenintegration war die ursprüngliche Struktur von (1.8) in l_2, k_2, l_3 und k_3 unverändert geblieben, so dass man nach der Rotation in eine $(+,-,-,+)$-Metrik die Transformationen

$$l_3 \longrightarrow l_3 + l_2, \qquad k_3 \longrightarrow k_3 + k_2 \tag{2.5}$$

ausführen kann um analog zu (2.2) eine Linearisierung in den Variablen l_2 und k_2 zu erhalten. Nach diesen Operationen kann die Residuenintegration in allen inneren Variablen ausgeführt werden. Die dazu notwendige Vertauschung der Integrationsreihenfolge ist wegen der absoluten Konvergenz des Integrals wieder erlaubt.

Wie oben erwähnt bestimmt das Vorzeichen des linearen Koeffizienten (l_1 in (2.2)), ob eine Polstelle innerhalb oder außerhalb des geschlossenen Integrationsweges liegt. Daher erhält man bei jeder Integration einer inneren Variablen eine Summe von Residuen wo jeder Summand mit einer Heaviside'schen θ-Funktion in den äußeren Variablen k_0, l_0, k_3 und l_3 gewichtet wird.

Bei der Residuenintegration in den linearisierten Impulsvariablen sorgen nun zwei Beziehungen dafür, dass die Anzahl der Terme nach vier hintereinander geschalteten Integrationen überschaubar bleibt. Die erste ist eine Konsequenz des Satzes über die Residuensumme Satz A.4, der besagt, dass die Summe der Residuen einer holomorphen Funktion verschwindet. Die Anwendung dieses Satzes auf linearisierte Variablen führt zu Korollar A.5 und ist in Anhang A beschrieben.

[2] Der Unterschied zwischen der Wick-Rotation und Kreimer-Rotation manifestiert sich auch dadurch, dass die Pole der Propagatoren bei letzterer weiterhin nahe an der reellen Achse bleiben, so dass es nicht erlaubt ist, sofort $\rho=0$ zu setzen.

Die zweite Beziehung zwischen den Termen, die bei der Residuenintegration auftritt, ist eine Folge der aufeinanderfolgenden Integration der in l_1 und l_2 linearen inversen Propagatoren P_1, P_2,..., P_n. Solche Paare von Schleifenimpulsen, die in Propagatoren doppelt vorkommen, werden in Folge als Zwillingsvariable bezeichnet werden. Die versprochene Beziehung kann wie folgt ausgedrückt werden:

Lemma 2.2 *Betrachte den Term, den man durch zweimalige Anwendung des Residuensatzes erhält: Zunächst durch die Berechnung des von P_i herrührenden Residuums an der Nullstelle $l_1^{(i)}$ und dann durch Berechnung des von P_j herrührenden Residuums an der Nullstelle $l_2^{(j)}$ (wobei $l_1^{(i)}$ schon eingesetzt ist). Dieser Term unterscheidet sich nur durch ein Vorzeichen von demjenigen Term, den man durch umgekehrte Residuenberechnung erhält: Zunächst durch die Berechnung des von P_j herrührenden Residuums an $l_1^{(j)}$ und dann durch Berechnung des von P_i herrührenden Residuums an $l_2^{(i)}$.*

Zum Beweis schreiben wir die inversen Propagatoren als $P_i(l_1, l_2) = \alpha_i + \beta_{i1} l_1 + \beta_{i2} l_2$ und bemerken, dass man die Polstellen $l_1^{(i)}$, $l_1^{(j)}$, $l_2^{(i)}$ und $l_2^{(j)}$ erhält durch Auflösen des inhomogenen linearen Gleichungssystems

$$\begin{pmatrix} P_i(l_1, l_2) \\ P_j(l_1, l_2) \end{pmatrix} = \begin{pmatrix} \alpha_i \\ \alpha_j \end{pmatrix} + \begin{pmatrix} \beta_{i1} & \beta_{i2} \\ \beta_{j1} & \beta_{j2} \end{pmatrix} \begin{pmatrix} l_1 \\ l_2 \end{pmatrix} \stackrel{!}{=} \begin{pmatrix} 0 \\ 0 \end{pmatrix}. \tag{2.6}$$

unter der Voraussetzung, dass $\det(\beta) \neq 0$. Die beiden Reihenfolgen in der Wahl der Residuen entsprechen direkt den beiden Reihenfolgen, wie man dieses System lösen kann: Entweder man löst die erste Zeile um das Ergebnis in die zweite Zeile einzusetzen oder umgekehrt. Die Lösungen, eingesetzt in die verbleibenden $P(q)$, sind natürlich dieselben. Also gilt $l_1^{(i)} = l_1^{(j)}$ und $l_2^{(i)} = l_2^{(j)}$. Dies beweist die Proportionalität der beiden Residuen. Die Proportionalitätskonstante -1 erhält man, indem man das Residuum nach der $(1, i) \to (2, j)$-Reihenfolge explizit ausrechnet. Es lautet:

$$J_{[i,j]} = \frac{1}{(\beta_{i1}\beta_{j2} - \beta_{j1}\beta_{i2})\, P_3 \cdots P_n|_{l_1^{(i)}, l_2^{(j)}}}.$$

Dies ist aber antisymmetrisch unter der Vertauschung von i und j. □

Man beobachte, dass der Term $(\beta_{i1}\beta_{j2} - \beta_{j1}\beta_{i2})$ im Nenner obigen Residuums eine 2×2-Determinante ist. Dieses Phänomen wird auf den nächsten Seiten verständlich gemacht werden.

Die in Vierbeinfunktionen so ausgewerteten Residuen aus Produkten von inversen Propagatoren $P_k(l_1, l_2)|_{l_1^{(i)}, l_2^{(j)}}$ können algebraisch recht umfangreich werden. Selbst bei Benutzung leistungsfähiger symbolischer Manipulationssysteme können Rechenzeiten in der Größenordnung von Minuten entstehen, nur um polynomiale Vereinfachungen wie das Kürzen von größten gemeinsamen Teilern durchzuführen. Es ist daher erstrebenswert, die Ausdrücke von vorneherein so einfach wie möglich zu konstruieren. Tatsächlich gibt es bei zweifacher Residuenintegration eine interessante Kürzung im auf Zähler und Nenner normalisierten Residuum. Diese Kürzung ist systematisch und lässt sich ausnutzen um einen Bruch zu generieren, in dem keine weiteren Kürzungen vorkommen werden.

Um die versprochene Kürzung zu sehen werden wir etwas auf die in der Beschreibung der Matrix-Klasse (Kapitel 4) erläuterten Eliminationsverfahren vorgreifen müssen. Wir gehen

wie folgt vor: Fasst man das Aufsuchen der Nullstellen der P_i und Einsetzen in die verbleibenden P_k als lineares Eliminationsverfahren auf, so findet man, dass es äquivalent zur Gauß-Elimination ist. Als Entrée wenden wir die Gauß-Elimination auf ein passend konstruiertes Gleichungssystem an und finden die Kürzung als Rechenergebnis. Anschließend werden wir die Gauß-Elimination gegen die divisionsfreie Elimination austauschen. Die Sylvester-Identität (Satz 4.2 auf Seite 107), die von der divisionsfreien- auf die teilerfreie Bareiss-Elimination führt, wird hier ebenso zur Anwendung kommen. Dies ermöglicht uns sogar, eine Verallgemeinerung auf mehr als zwei Variablen einzusehen.

Wir schreiben die in l_1 und l_2 linearen inversen Propagatoren als lineares Gleichungssystem:[3]

$$\begin{pmatrix} P_1(l_1,l_2) \\ P_2(l_1,l_2) \\ P_3(l_1,l_2) \end{pmatrix} = \begin{pmatrix} \beta_{11} & \beta_{12} & \alpha_1 \\ \beta_{21} & \beta_{22} & \alpha_2 \\ \beta_{31} & \beta_{32} & \alpha_3 \end{pmatrix} \begin{pmatrix} l_1 \\ l_2 \\ 1 \end{pmatrix}. \tag{2.7}$$

Das Residuum an der Nullstelle $l_1^{(1)} = -\frac{\beta_{12}l_2+\alpha_1}{\beta_{11}}$ von $P_1(l_1)$ findet man durch Einsetzen

$$\frac{1}{P_1(l_1,l_2)P_2(l_1,l_2)P_3(l_1,l_2)} \longrightarrow \frac{2\pi i}{\beta_{11}P_2^{(1)}(l_2)P_3^{(1)}(l_2)}$$

und dies entspricht dem ersten Schritt der Gauß-Eliminationsvorschrift (4.9), nach dem das Gleichungssystem

$$\begin{pmatrix} P_2^{(1)}(l_2) \\ \\ P_3^{(1)}(l_2) \end{pmatrix} \equiv \begin{pmatrix} \beta'_{22} & \alpha'_2 \\ \\ \beta'_{32} & \alpha'_3 \end{pmatrix} \begin{pmatrix} l_2 \\ \\ 1 \end{pmatrix} = \begin{pmatrix} \beta_{22} - \dfrac{\beta_{21}\beta_{12}}{\beta_{11}} & \alpha_2 - \dfrac{\beta_{21}\alpha_1}{\beta_{11}} \\ \beta_{32} - \dfrac{\beta_{31}\beta_{12}}{\beta_{11}} & \alpha_3 - \dfrac{\beta_{31}\alpha_1}{\beta_{11}} \end{pmatrix} \begin{pmatrix} l_2 \\ \\ 1 \end{pmatrix}.$$

übrigbleibt. Im zweiten Eliminationsschritt wird die Nullstelle $l_2^{(2)} = -\alpha'_2/\beta'_{22} = -(\alpha_2 - \frac{\beta_{21}\alpha_1}{\beta_{11}})/(\beta_{22} - \frac{\beta_{21}\beta_{12}}{\beta_{11}})$ von $P_2^{(1)}$ in l_2 eingesetzt

$$\frac{2\pi i}{\beta_{11}P_2^{(1)}P_3^{(1)}} \longrightarrow \frac{(2\pi i)^2}{\beta_{11}\beta'_{22}P_3^{(2)}} = \frac{(2\pi i)^2}{\beta_{11}\Big(\beta_{22} - \dfrac{\beta_{21}\beta_{12}}{\beta_{11}}\Big)P_3^{(2)}}$$

und das verbleibende (eindimensionale) Gleichungssystem ist

$$P_3^{(2)} = \alpha'_3 - \frac{\beta'_{32}\alpha'_2}{\beta'_{22}} = \Big(\alpha_3 - \frac{\beta_{31}\alpha_1}{\beta_{11}}\Big) - \frac{\Big(\beta_{32} - \dfrac{\beta_{31}\beta_{12}}{\beta_{11}}\Big)\Big(\alpha_2 - \dfrac{\beta_{21}\alpha_1}{\beta_{11}}\Big)}{\Big(\beta_{22} - \dfrac{\beta_{21}\beta_{12}}{\beta_{11}}\Big)}.$$

[3] Weitere P_k können natürlich analog notiert werden. Dies wird sich aber als nicht nötig herausstellen. Nachdem die Nullstellen $l_1^{(1)}$ von $P_1(l_1,l_2)$ und $l_2^{(2)}$ von $P_2(l_1,l_2)|_{l_1^{(1)}}$ aufgefunden worden sind, können wir sie einfach in die verbleibenden P_k einsetzen und anmultiplizieren.

Bringen wir alles auf einen Hauptnenner, multiplizieren aus und kürzen gemeinsame Faktoren aus Zähler und Nenner, so finden wir wieder eine Determinante im Nenner:

$$\frac{(2\pi i)^2}{\beta_{11}\Big(\beta_{22}-\frac{\beta_{21}\beta_{12}}{\beta_{11}}\Big)P_3^{(2)}} = \frac{(2\pi i)^2}{\beta_{11}\beta_{22}\alpha_3+\beta_{12}\alpha_2\beta_{31}+\alpha_1\beta_{21}\beta_{32}-\beta_{31}\beta_{22}\alpha_1-\beta_{21}\beta_{12}\alpha_3-\beta_{11}\beta_{32}\alpha_2}.$$

Hieraus ergibt sich der folgende nützliche Satz:

Satz 2.3 *Betrachte den Term, den man nach zweimaliger Anwendung des Residuensatzes auf* $\frac{1}{P_1(l_1,l_2)\cdots P_n(l_1,l_2)}$, $n \geq 3$ *am von* P_1 *in* l_1 *erzeugten Pol* $l_1^{(1)}$ *und danach am von* P_2 *in* l_2 *erzeugten Pol* $l_2^{(2)}$ *erhält. Mit der Notation aus Gleichung (2.7) lautet dieser Term*

$$\operatorname*{Res}_{l_2=l_2^{(2)}}\operatorname*{Res}_{l_1=l_1^{(1)}}\frac{1}{P_1(l_1,l_2)\cdots P_n(l_1,l_2)} = 1\Bigg/\left(\begin{vmatrix}\beta_{11} & \beta_{12} & \alpha_1\\ \beta_{21} & \beta_{22} & \alpha_2\\ \beta_{31} & \beta_{32} & \alpha_3\end{vmatrix} P_4(l_1^{(1)},l_2^{(2)})\cdots P_n(l_1^{(1)},l_2^{(2)})\right).$$

Bei dem insgesamt wegkürzbaren gemeinsamen Faktor handelt es sich im Übrigen um $\beta_{11}(\beta_{11}\beta_{22}-\beta_{21}\beta_{12})$, also einen durchaus nichttrivialen Term von dritter Ordnung in Impulsvariablen. Der Satz erlaubt daher beträchtliche Vereinfachungen. Man bemerke zudem, dass aus dem Satz auch das Vorzeichen in Lemma 2.2 folgt, da die Determinante antisymmetrisch ist unter Vertauschung zweier Zeilen oder Spalten. Insbesondere entspricht die Vertauschung der ersten und der zweiten Zeilen genau der Vertauschung der Integrationsreihenfolge in Lemma 2.2.

Die Quintessenz der Eliminationsmethode besteht darin, dass nach zweimaliger Gauß-Elimination in der Koeffizientenmatrix der Propagatoren das Residuum rechts unten stehen bleibt. Um das Auftreten der Determinante und die Vereinfachungen besser zu verstehen greifen wir nun etwas vor und wenden anstatt des Gauß- das teilerfreie Eliminationsverfahren von Bareiss (4.11) auf das Gleichungssystem (2.7) an. Analog zu Gleichung (2.1) erhält man nun die Einsetzung

$$\frac{1}{P_1(l_1,l_2)P_2(l_1,l_2)P_3(l_1,l_2)} \longrightarrow \frac{2\pi i}{\beta_{11}\tilde{P}_2^{(1)}(l_2)/\beta_{11}\tilde{P}_3^{(1)}(l_2)/\beta_{11}}$$

worin $\tilde{P}_1^{(1)}$ und $\tilde{P}_2^{(1)}$ nun durch den ersten Eliminationsschritt bestimmt sind:

$$\begin{pmatrix}\tilde{P}_2^{(1)}(l_2)\\ \tilde{P}_3^{(1)}(l_2)\end{pmatrix} \equiv \begin{pmatrix}\tilde{\beta}'_{22} & \tilde{\alpha}'_2\\ \tilde{\beta}'_{32} & \tilde{\alpha}'_3\end{pmatrix}\begin{pmatrix}l_2\\ 1\end{pmatrix} = \begin{pmatrix}\beta_{22}\beta_{11}-\beta_{21}\beta_{12} & \alpha_2\beta_{11}-\beta_{21}\alpha_1\\ \beta_{32}\beta_{11}-\beta_{31}\beta_{12} & \alpha_3\beta_{11}-\beta_{31}\alpha_1\end{pmatrix}\begin{pmatrix}l_2\\ 1\end{pmatrix}.$$

Das Analogon zu (2.1) lautet nun

$$\frac{2\pi i}{\beta_{11}\tilde{P}_2^{(1)}(l_2)/\beta_{11}\tilde{P}_3^{(1)}(l_2)/\beta_{11}} \longrightarrow \frac{(2\pi i)^2}{\beta_{11}\tilde{\beta}'_{22}/\beta_{11}\tilde{P}_3^{(2)}/(\beta_{11}\tilde{\beta}'_{22})} = \frac{(2\pi i)^2}{\beta_{11}(\beta_{22}\beta_{11}-\beta_{21}\beta_{12})/\beta_{11}\tilde{P}_3^{(2)}/\big(\beta_{11}(\beta_{22}\beta_{11}-\beta_{21}\beta_{12})\big)}$$

und $\tilde{P}_3^{(2)}$ folgt aus dem zweiten Eliminationsschritt zu

$$\begin{aligned}\tilde{P}_3^{(2)} &= \tilde{\alpha}'_3\tilde{\beta}'_{22} - \tilde{\beta}'_{32}\tilde{\alpha}'_2 \\ &= (\alpha_3\beta_{11} - \beta_{31}\alpha_1)(\beta_{22}\beta_{11} - \beta_{21}\beta_{12}) - (\beta_{32}\beta_{11} - \beta_{31}\beta_{12})(\alpha_2\beta_{11} - \beta_{21}\alpha_1).\end{aligned}$$

Die Sylvester-Identität (Satz 4.2) verlangt nun, dass dieses $\tilde{\alpha}'_3\tilde{\beta}'_{22} - \tilde{\beta}'_{32}\tilde{\alpha}'_2 = \tilde{P}_3^{(2)}$ von β_{11} ohne Rest geteilt wird und das Ergebnis die Determinante

$$\tilde{P}_3^{(2)}/\beta_{11} = \begin{vmatrix} \beta_{11} & \beta_{12} & \alpha_1 \\ \beta_{21} & \beta_{22} & \alpha_2 \\ \beta_{31} & \beta_{32} & \alpha_3 \end{vmatrix}$$

ist. Dies ist prinzipiell erweiterbar auf größere Systeme derselben Art, wenn der Residuensatz in mehr als zwei Variablen angewendet werden soll.

Der Satz über die Residuensumme zusammen mit Lemma 2.2 machen das Verfahren erst übersichtlich, indem sie die Anzahl der auftretenden Terme limitieren. Im Falle der planaren Boxfunktion beschränkt der Satz über die Residuensumme die Anzahl der Summanden von 144 auf 36 und Lemma 2.2 identifiziert jeweils vier der 36 miteinander, so dass nur 9 übrigbleiben. Satz 2.3 erlaubt uns, den ggT von Zähler und Nenner der verbleibenden Summanden zu isolieren, ohne ihn mit einem ggT-Algorithmus berechnen zu müssen, was zwar nicht unmöglich aber sehr aufwändig wäre (einige Minuten pro Rechnung bei voll expandierten inversen Propagatoren).

Außerdem sollte an dieser Stelle auf die für eine Implementierung wertvolle Tatsache hingewiesen werden, dass eine solchermaßen vereinfachte Darstellung unanfällig ist gegen unechte Divisionen durch Null. Ein Ergebnis, dargestellt als Bruch aus Zähler und Nenner, kann nur dann eine Division durch Null hervorrufen, wenn der Nenner wirklich verschwindet. Verschachtelte Brüche hingegen können in Zähler und Nenner einzelne Divisionen durch Null enthalten, die in normalisierter Form überhaupt nicht auftreten würden, die Normalisierung jedoch vereiteln. Anders ausgedrückt: hebbare Singularitäten werden frühzeitig beseitigt. Das ist dasselbe Muster wie das des weiter unten (im Kasten auf Seite 106) beschriebenen Implementierungsproblemes bei der Gauß-Elimination.

Fünfbeinfunktionen?

Nach der Betrachtung von Zwei-, Drei- und Vierbeinfunktionen stellt sich die Frage, ob das Verfahren prinzipiell auf skalare Funktionen mit noch mehr äußeren Impulsen ausgedehnt werden kann. Wenn die Anzahl der äußeren Beine $n \geq 5$ ist, gibt es keinen Orthogonalraum und auch die Dreierkomponenten der inneren Schleifenimpulse mischen mit äußeren Parametern. Die Tatsache $n \geq 5$ jedoch garantiert, dass die Konvergenz der Integrale nach geeigneter Partialbruchzerlegung noch ausreicht, so dass lineare Verschiebungen erlaubt sind. Auf diese Weise können die Dreierkomponenten von den äußeren Parametern befreit und das Verfahren auf eine Linearkombination von Vierbeinfunktionen zurückgeführt werden.

Das Verfahren in drei Dimensionen?

Man könnte fragen, warum wir bei der Linearisierung stets den quadratischen Anteil des Schleifenimpules l_1 in l_0 und denjenigen von l_2 in l_3 untergebracht haben. Die linearisierende Substitution

$$l_0 \to l_0 + l_1, l_3 \to l_3 + l_2 : \qquad l_0^2 - l_1^2 - l_2^2 + l_3^2 - m^2 + i\rho \;\longrightarrow\; l_0^2 - 2l_0 l_1 - 2l_2 l_3 + l_3^2 - m^2 + i\rho$$

könnte ersetzt werden durch

$$l_0 \to l_0 + l_1 + l_2 : \qquad l_0^2 - l_1^2 - l_2^2 + l_3^2 - m^2 + i\rho \;\longrightarrow\; l_0^2 - 2l_0 l_1 - 2l_2 l_2 - 2l_1 l_2 + l_3^2 - m^2 + i\rho$$

und entsprechend für den k-Schleifenimpuls. Dabei blieben die 3-Komponenten der Schleifenimpulse unangetastet mit dem Vorteil, dass man die Kreimer-Rotation nicht benötigt und das ganze Verfahren so auch in drei Raumzeitdimensionen, also ohne verfügbarem Orthogonalraum, anwendbar wäre. Das Problem bereitet die dabei auftretende Mischung der Impulskomponenten l_1 und l_2. Nehmen wir als Integrationsreihenfolge l_1, l_2 an. Nach der l_1-Integration fällt zwar wie gewohnt ein inverser Propagator weg, dafür taucht nun aber l_2 im Residuum auf. Außerdem hängt das Integrationsgebiet nun von einer Kombination von l_0 und l_2 sowie äußeren Impulsen ab. Die l_2-Abhängigkeit des Gebietes verbietet aber gerade die Anwendung des Residuensatzes bei der nächsten Integration.

2.2. Die einschränkenden Bedingungen

Wir benutzen den Satz über die Residuensumme (Korollar A.5) sowie Lemma 2.2 und Satz 2.3 um eine Masterformel für die vier inneren Integrationen zu gewinnen, die sich für eine programmatische Implementierung eignet. Dann untersuchen wir die dabei anfallenden symbolischen Bedingungen an die verbleibenden vier Integrationen und schlagen Methoden vor, diese in überschaubare Form zu bekommen.

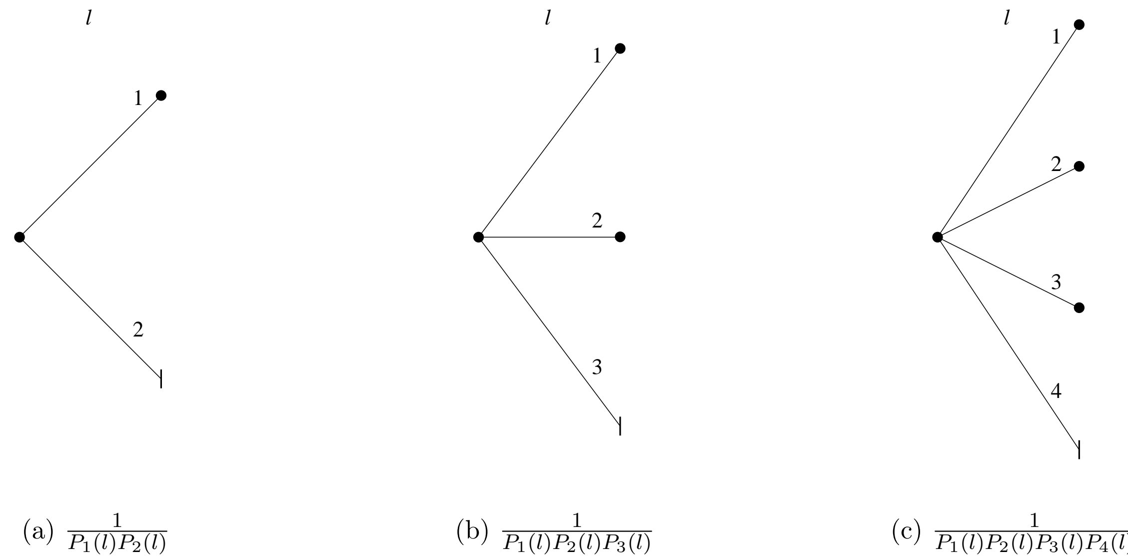
Die einschränkenden Bedingungen können gemäß Korollar A.5 in Form von Heaviside'schen θ-Funktionen geschrieben werden. Als Fingerübung erläutern wir dies anhand von zwei Propagatoren $P_i(l)$ und $P_k(l)$, später werden mehr hinzu kommen. Führt man die Residuenintegration des Integrals

$$\int dl \frac{1}{P_i(l)P_k(l)}$$

aus, wobei $P_i(l)$ und $P_k(l)$ gegeben sind durch das lineare Gleichungssystem

$$\begin{pmatrix} P_i(l) \\ P_k(l) \end{pmatrix} = \begin{pmatrix} \beta_i & \alpha_i \\ \beta_k & \alpha_k \end{pmatrix} \begin{pmatrix} l \\ 1 \end{pmatrix},$$

so sind die Nullstellen von $P_i(l)$ und $P_k(l)$, also $l^{(i)} = -\alpha_i/\beta_i$ und $l^{(k)} = -\alpha_k/\beta_k$, in der komplexen Ebene aufzusuchen. Wir können nun ausnutzen, dass die Koeffizienten der internen Impulse, also die β, keinen Beitrag vom kausalen $i\rho$ haben. Wir werden gleich sehen, dass dies auch nach mehrfacher Residuenintegration noch gültig bleiben wird. Das von $P_i(l)$ herrührende Residuum liefert also nur dann einen Betrag, wenn $\operatorname{Im}(-\alpha_i/\beta_i) = -\operatorname{Im}(\alpha_i)/\beta_i$ positiv ist. Das



(a) $\frac{1}{P_1(l)P_2(l)}$ (b) $\frac{1}{P_1(l)P_2(l)P_3(l)}$ (c) $\frac{1}{P_1(l)P_2(l)P_3(l)P_4(l)}$

Abbildung 2.3.: *Verbleibende Terme nach einer Residuenintegration im linearisierten Schleifenimpuls l für drei verschiedene Integranden.*

volle Ergebnis lautet dann

$$\begin{aligned}\int dl \frac{1}{P_i(l)P_k(l)} &= (2\pi i)\Big(\operatorname*{Res}_{P_i=0} \frac{1}{P_i(l)P_k(l)} + \operatorname*{Res}_{P_k=0} \frac{1}{P_i(l)P_k(l)}\Big) \\ &= (2\pi i)\frac{\theta(-\operatorname{Im}(\alpha_i)/\beta_i) - \theta(-\operatorname{Im}(\alpha_k)/\beta_k)}{\beta_i\alpha_k - \beta_k\alpha_i}. \end{aligned} \tag{2.8}$$

Es ist instruktiv, sich dieses Ergebnis symbolisch als Baum zu verdeutlichen, wie in Abbildung 2.3(a). Ausgehend von einem Produkt inverser Propagatoren (Wurzel) entstehen zwei Summanden (Äste). Der Strich am Ende des 2-Astes deutet an, dass diese Integration nach dem Satz A.4 über die Residuensumme nicht ausgeführt wird sondern sich nur in der zweiten θ-Funktion im Zähler äußert.

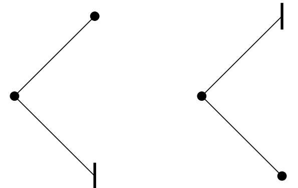


Abbildung 2.4.: *Wahlfreiheit bei* $\int dl\, 1/(P_1(l)P_2(l))$

Dieses Vorgehen zeichnet ein P_n vor allen anderen aus und es ist nicht ganz trivial, welchem der Vorzug zu geben ist. Tatsächlich kann die Integration so auf Wege geleitet werden, die einer weiteren symbolischen Behandlung entweder mehr oder weniger zugänglich sind. Natürlich ist das Endergebnis nach dem Satz über die Residuensumme invariant unter der Auswahl des „abgeschnittenen“ Astes. Für den nebenstehend abgebildeten Fall mag man diese Invarianz noch sofort einsehen, da das Ergebnis manifest invariant unter der Vertauschung der Indizes ist. Im ersten Fall lautet es schließlich

$$\begin{aligned}\int dl \frac{1}{P_1(l)P_2(l)} &\equiv \int dl \frac{1}{(\beta_1 l + \alpha_1)(\beta_2 l + \alpha_2)} \\ = \frac{\theta(-\operatorname{Im}(\alpha_1)/\beta_1) - \theta(-\operatorname{Im}(\alpha_2)/\beta_2)}{\beta_1 P_2|_{l^{(1)}}} &= \frac{\theta(-\operatorname{Im}(\alpha_1)/\beta_1) - \theta(-\operatorname{Im}(\alpha_2)/\beta_2)}{\beta_2\alpha_1 - \beta_1\alpha_2}.\end{aligned}$$

Aber für mehr als zwei Propagatoren ist die Situation ganz anders. Das Ergebnis, welches man mit unserem Verfahren bei drei Propagatoren durch Abschneiden des P_3-Astes erhält

$$\begin{aligned}\int dl \frac{1}{P_1(l)P_2(l)P_3(l)} &\equiv \int dl \frac{1}{(\beta_1 l+\alpha_1)(\beta_2 l+\alpha_2)(\beta_3 l+\alpha_3)} \\ &= \frac{\theta(-\mathrm{Im}(\alpha_1)/\beta_1)-\theta(-\mathrm{Im}(\alpha_3)/\beta_3)}{\beta_1 P_2|_{l^{(1)}}\, P_3|_{l^{(1)}}} + \frac{\theta(-\mathrm{Im}(\alpha_2)/\beta_2)-\theta(-\mathrm{Im}(\alpha_3)/\beta_3)}{\beta_2 P_1|_{l^{(2)}}\, P_3|_{l^{(2)}}} \\ &= \frac{\beta_1\big(\theta(-\mathrm{Im}(\alpha_1)/\beta_1)-\theta(-\mathrm{Im}(\alpha_3)/\beta_3)\big)}{(\beta_1\alpha_2-\beta_2\alpha_1)(\beta_1\alpha_3-\beta_3\alpha_1)} + \frac{\beta_2\big(\theta(-\mathrm{Im}(\alpha_2)/\beta_2)-\theta(-\mathrm{Im}(\alpha_3)/\beta_3)\big)}{(\beta_2\alpha_1-\beta_1\alpha_2)(\beta_2\alpha_3-\beta_3\alpha_2)}\end{aligned}$$

ist nur durch eine Konspiration der θ-Funktionen im Zähler identisch mit dem Ergebnis, welches man durch Abschneiden des P_2-Astes erhält:

$$\begin{aligned}&\int dl \frac{1}{P_1(l)P_2(l)P_3(l)} \\ &= \frac{\beta_1\big(\theta(-\mathrm{Im}(\alpha_1)/\beta_1)-\theta(-\mathrm{Im}(\alpha_2)/\beta_2)\big)}{(\beta_1\alpha_2-\beta_2\alpha_1)(\beta_1\alpha_3-\beta_3\alpha_1)} + \frac{\beta_3\big(\theta(-\mathrm{Im}(\alpha_3)/\beta_3)-\theta(-\mathrm{Im}(\alpha_2)/\beta_2)\big)}{(\beta_3\alpha_1-\beta_1\alpha_3)(\beta_3\alpha_2-\beta_2\alpha_3)}.\end{aligned}$$

In der Praxis kann man die Wahlfreiheit des „abgeschnittenen" Integrationsastes ausnutzen, um das symbolische Ergebnis zu vereinfachen.

Völlig analog zu Gleichung (2.8) kann man nun darangehen, die Residuenintegration in mehr als zwei inversen linearen Propagatoren zu betrachten. Für drei durch

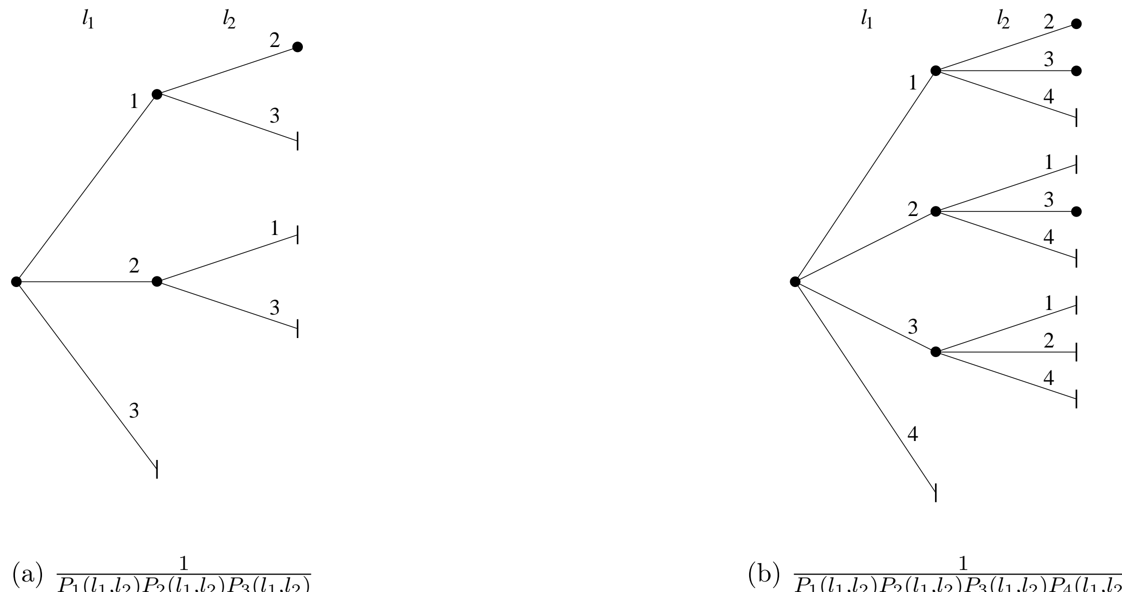
$$\begin{pmatrix} P_i(l) \\ P_j(l) \\ P_k(l) \end{pmatrix} = \begin{pmatrix} \beta_i & \alpha_i \\ \beta_j & \alpha_j \\ \beta_k & \alpha_k \end{pmatrix} \begin{pmatrix} l \\ 1 \end{pmatrix}$$

bestimmte $P(l)$ kommt gegenüber (2.8) zunächst einmal ein additiver Term vom zusätzlichen P, ausgewertet an der Nullstelle $l^{(i)}$, hinzu. Außerdem tritt ein additiver Faktor auf, den man durch die Vertauschung der Indizes i und j erhält:

$$\begin{aligned}&\int dl \frac{1}{P_i(l)P_j(l)P_k(l)} \\ &= (2\pi i)\Big(\operatorname*{Res}_{P_i=0} \frac{1}{P_i(l)P_j(l)P_k(l)} + \operatorname*{Res}_{P_j=0} \frac{1}{P_i(l)P_j(l)P_k(l)} + \operatorname*{Res}_{P_k=0} \frac{1}{P_i(l)P_j(l)P_k(l)}\Big) \\ &= (2\pi i)\Big(\frac{\theta(-\mathrm{Im}(\alpha_i)/\beta_i)-\theta(-\mathrm{Im}(\alpha_k)/\beta_k)}{(\beta_i\alpha_j-\beta_j\alpha_i)P_k(l^{(i)})} + \frac{\theta(-\mathrm{Im}(\alpha_j)/\beta_j)-\theta(-\mathrm{Im}(\alpha_k)/\beta_k)}{(\beta_j\alpha_i-\beta_i\alpha_j)P_k(l^{(j)})}\Big). \qquad (2.9)\end{aligned}$$

Im Prinzip haben wir hier eine Partialbruchzerlegung vorgenommen – man vergleiche die Gleichungen (2.9) und (A.5). Die zwei Terme entsprechen den zwei Punkten an den Ästen in Abbildung 2.3(b), die zweite θ-Funktion in den Zählern des Ergebnisses kommen wieder vom abgeschnittenen Zweig.

Die verbleibenden Integrationsvariablen stecken alle in α_i, α_j und α_k. Diese werden im Nenner gemischt mit β_i und β_k, also mit reellen Impulsvariablen – Terme der Form $\alpha_i\alpha_k$ treten dabei nicht auf. Folglich sind die βs im nächsten Integrationsschritt wieder reell. Berücksichtigen wir noch, dass dies auch im ersten Integrationsschritt so war, da die P_n ja gemäß Gleichung (2.2) gebildet worden sind, so sehen wir das folgende beruhigende Lemma ein:



(a) $\frac{1}{P_1(l_1,l_2)P_2(l_1,l_2)P_3(l_1,l_2)}$ (b) $\frac{1}{P_1(l_1,l_2)P_2(l_1,l_2)P_3(l_1,l_2)P_4(l_1,l_2)}$

Abbildung 2.5.: *Verbleibende Terme nach zwei Residuenintegrationen in linearisierten Zwillingsvariablen l_1 und l_2. Die abgeschnittenen Äste müssen nicht mehr berechnet werden.*

Lemma 2.4 *Bei unserem Verfahren bleiben die Koeffizienten der Schleifenimpulse stets reell. Anders ausgedrückt: in Gleichung (2.7) gilt $\beta_{nm} \in \mathbb{R}$, $\alpha_n \in \mathbb{C}$.*

Dieses Lemma rechtfertigt rückwirkend die Schreibweise der Argumente der θ-Funktionen in Gleichungen (2.8) und (2.9). Außerdem stellt es eine Invariante des Verfahrens dar und eignet sich damit vorzüglich zur Fehlerdetektion in der Implementierung.

Wir werden nun eine zweite Integration explizit ausführen, da die einschränkenden Bedingungen bei der Integration von Zwillingsvariablen noch explizit konstruiert werden müssen. Dazu gehen wir wieder von der Darstellung der inversen Propagatoren als linearem Gleichungssystem in den Zwillingsvariablen l_1 und l_2 wie in (2.7) aus:

$$\begin{pmatrix} P_i(l_1,l_2) \\ P_j(l_1,l_2) \\ P_k(l_1,l_2) \end{pmatrix} = \begin{pmatrix} \beta_{i1} & \beta_{i2} & \alpha_i \\ \beta_{j1} & \beta_{j2} & \alpha_j \\ \beta_{k1} & \beta_{k2} & \alpha_k \end{pmatrix} \begin{pmatrix} l_1 \\ l_2 \\ 1 \end{pmatrix}.$$

Berechnet werden soll das Integral

$$\int dl_1 dl_2 \frac{1}{P_i(l_1,l_2)P_j(l_1,l_2)P_k(l_1,l_2)},$$

wobei wir den Nenner des Ergebnisses schon aus Satz 2.3 kennen. Wir müssen nun an der von P_i, P_j und P_k produzierten Polstelle das Residuum berechnen. Nachdem die l_1-Integration wie in (2.8) ausgeführt worden ist, wurden die Nullstellen der P_n in l_1

$$l_1^{(i)} = -\frac{\beta_{i2}l_2 + \alpha_i}{\beta_{i1}}, \qquad l_1^{(j)} = -\frac{\beta_{j2}l_2 + \alpha_j}{\beta_{j1}}$$

implizit in den verbleibenden Impulsen eingesetzt:

$$
\begin{aligned}
P_j(l_2)\big|_{l_1^{(i)}} &= \frac{1}{\beta_{i1}}\big(\beta_{i1}\beta_{j2} - \beta_{i2}\beta_{j1}\big)l_2 + \frac{1}{\beta_{i1}}\big(\beta_{i1}\alpha_j - \beta_{j1}\alpha_i\big) \\
P_k(l_2)\big|_{l_1^{(i)}} &= \frac{1}{\beta_{i1}}\big(\beta_{i1}\beta_{k2} - \beta_{i2}\beta_{k1}\big)l_2 + \frac{1}{\beta_{i1}}\big(\beta_{i1}\alpha_k - \beta_{k1}\alpha_i\big) \\
P_i(l_2)\big|_{l_1^{(j)}} &= \frac{1}{\beta_{j1}}\big(\beta_{j1}\beta_{i2} - \beta_{j2}\beta_{i1}\big)l_2 + \frac{1}{\beta_{j1}}\big(\beta_{j1}\alpha_i - \beta_{i1}\alpha_j\big) \\
P_k(l_2)\big|_{l_1^{(j)}} &= \frac{1}{\beta_{j1}}\big(\beta_{j1}\beta_{k2} - \beta_{j2}\beta_{k1}\big)l_2 + \frac{1}{\beta_{j1}}\big(\beta_{j1}\alpha_k - \beta_{k1}\alpha_j\big)
\end{aligned}
\tag{2.10}
$$

Wir suchen wieder die Nullstellen in der komplexen l_2-Ebene auf und erhalten als Ergebnis

$$
\begin{aligned}
&\int dl_1 dl_2 \frac{1}{P_i(l_1,l_2)P_j(l_1,l_2)P_k(l_1,l_2)} = \\
&\Bigg(\Big\{\theta\Big(\frac{\mathrm{Im}(\alpha_i)}{\beta_{i1}}\Big) - \theta\Big(\frac{\mathrm{Im}(\alpha_k)}{\beta_{k1}}\Big)\Big\}\Big\{\theta\Big(\frac{\beta_{i1}\,\mathrm{Im}(\alpha_j) - \beta_{j1}\,\mathrm{Im}(\alpha_i)}{\beta_{i1}\beta_{j2} - \beta_{i2}\beta_{j1}}\Big) - \theta\Big(\frac{\beta_{i1}\,\mathrm{Im}(\alpha_k) - \beta_{k1}\,\mathrm{Im}(\alpha_i)}{\beta_{i1}\beta_{k2} - \beta_{i2}\beta_{k1}}\Big)\Big\} \\
&- \Big\{\theta\Big(\frac{\mathrm{Im}(\alpha_j)}{\beta_{j1}}\Big) - \theta\Big(\frac{\mathrm{Im}(\alpha_k)}{\beta_{k1}}\Big)\Big\}\Big\{\theta\Big(\frac{\beta_{j1}\,\mathrm{Im}(\alpha_i) - \beta_{i1}\,\mathrm{Im}(\alpha_j)}{\beta_{j1}\beta_{i2} - \beta_{j2}\beta_{i1}}\Big) - \theta\Big(\frac{\beta_{j1}\,\mathrm{Im}(\alpha_k) - \beta_{k1}\,\mathrm{Im}(\alpha_j)}{\beta_{j1}\beta_{k2} - \beta_{j2}\beta_{k1}}\Big)\Big\}\Bigg) \\
&\times (2\pi i)^2 \Bigg/ \begin{vmatrix} \beta_{i1} & \beta_{i2} & \alpha_i \\ \beta_{j1} & \beta_{j2} & \alpha_j \\ \beta_{k1} & \beta_{k2} & \alpha_k \end{vmatrix}.
\end{aligned}
\tag{2.11}
$$

Mit $i = 1$, $j = 2$ und $k = 3$ entsprechen die beiden Summanden genau den Ästen 1-2 und 2-1 in Abbildung 2.5(a). Falls im ursprünglichen Integral weitere Propagatoren $P_l(l_1,l_2)\ldots P_n(l_1,l_2)$ vorkamen, so sind diese in (2.11) eingesetzt zu verstehen. Sie müssen lediglich an den Nullstellen

$$
l_1^{(i,j)} \equiv l_1^{(j,i)} = \frac{\alpha_j\beta_{i2} - \alpha_i\beta_{j2}}{\beta_{j2}\beta_{i1} - \beta_{j1}\beta_{i2}} \tag{2.12}
$$

$$
l_2^{(i,j)} \equiv l_2^{(j,i)} = \frac{\alpha_i\beta_{j1} - \alpha_j\beta_{i1}}{\beta_{j2}\beta_{i1} - \beta_{j1}\beta_{i2}} \tag{2.13}
$$

ausgewertet werden, analog zu Satz 2.3 von Seite 32. Für vier Propagatoren erhält man beispielsweise das System von Abbildung 2.5(b).

Es sollte noch angemerkt werden, dass eine Implementierung dieses Verfahrens in einem symbolischen System eine solche Einsetzung in zum Beispiel $P_n(l_1,l_2) = \beta_{n1}l_1 + \beta_{n2}l_2 + \alpha_n$ nicht naiv vornehmen sollte. Sonst wird nämlich der $\Rightarrow$ Integritätsbereich der Polynome über den α und β verlassen und die entstehenden Rechnungen im Quotientenkörper sind ungleich aufwändiger. Stattdessen können Zähler und Nenner der entstehenden Terme getrennt verwaltet werden. Dabei hilft, dass die Nenner in (2.12) und (2.13) identisch sind. Wenn man diesen Nenner $\beta_{j2}\beta_{i1} - \beta_{j1}\beta_{i2}$ zum Zähler des Ergebnisses multipliziert, so kann man die Ersetzung

im Nenner des Ergebnisses wieder als Determinante schreiben:

$$\begin{aligned}(\beta_{j2}\beta_{i1}-\beta_{j1}\beta_{i2})\cdot P_n|_{l_1^{(i,j)},l_2^{(i,j)}} &= \beta_{n1}(\alpha_j\beta_{i2}-\alpha_i\beta_{j2})+\beta_{n2}(\alpha_i\beta_{j1}-\alpha_j\beta_{i1})+\alpha_n(\beta_{j2}\beta_{i1}-\beta_{j1}\beta_{i2})\\ &= \begin{vmatrix}\beta_{i1} & \beta_{i2} & \alpha_i\\ \beta_{j1} & \beta_{j2} & \alpha_j\\ \beta_{n1} & \beta_{n2} & \alpha_n\end{vmatrix}.\end{aligned}$$

Wir sind nun in der Lage, die versprochene Masterformel geschlossen anzugeben.

Satz 2.5 (Masterformel für Residuenintegration in Zwillingsvariablen) *Seien n inverse Propagatoren gegeben durch*

$$\begin{pmatrix}P_1(l_1,l_2)\\ P_2(l_1,l_2)\\ \vdots\\ P_n(l_1,l_2)\end{pmatrix}=\begin{pmatrix}\beta_{11} & \beta_{12} & \alpha_1\\ \beta_{21} & \beta_{22} & \alpha_2\\ \vdots & \vdots & \vdots\\ \beta_{n1} & \beta_{n2} & \alpha_n\end{pmatrix}\begin{pmatrix}l_1\\ l_2\\ 1\end{pmatrix}$$

und seien die darin auftretenden β_{ij} alle reell. Wir schreiben abkürzend für die benötigten 3×3-Determinanten

$$B_{ijk}:=\begin{vmatrix}\beta_{i1} & \beta_{i2} & \alpha_i\\ \beta_{j1} & \beta_{j2} & \alpha_j\\ \beta_{k1} & \beta_{k2} & \alpha_k\end{vmatrix}.$$

Wenn das Integral $\int dl_1 dl_2 \prod_{i=1}^n \frac{1}{P_i(l_1,l_2)}$ existiert und die B_{ijk} für alle paarweise verschiedenen i,j,k nicht Null sind, dann gilt:

$$\begin{aligned}&\int dl_1 dl_2 \prod_{i=1}^n \frac{1}{P_i(l_1,l_2)} =\\ &\sum_{i=1}^{n-2}\sum_{j=i+1}^{n-1}\Bigg(\left\{\theta\Big(\frac{\mathrm{Im}(\alpha_i)}{\beta_{i1}}\Big)-\theta\Big(\frac{\mathrm{Im}(\alpha_n)}{\beta_{n1}}\Big)\right\}\\ &\qquad\times\left\{\theta\Big(\frac{\beta_{i1}\,\mathrm{Im}(\alpha_j)-\beta_{j1}\,\mathrm{Im}(\alpha_i)}{\beta_{i1}\beta_{j2}-\beta_{i2}\beta_{j1}}\Big)-\theta\Big(\frac{\beta_{i1}\,\mathrm{Im}(\alpha_n)-\beta_{n1}\,\mathrm{Im}(\alpha_i)}{\beta_{i1}\beta_{n2}-\beta_{i2}\beta_{n1}}\Big)\right\}\\ &\qquad-\left\{\theta\Big(\frac{\mathrm{Im}(\alpha_j)}{\beta_{j1}}\Big)-\theta\Big(\frac{\mathrm{Im}(\alpha_n)}{\beta_{n1}}\Big)\right\}\\ &\qquad\times\left\{\theta\Big(\frac{\beta_{j1}\,\mathrm{Im}(\alpha_i)-\beta_{i1}\,\mathrm{Im}(\alpha_j)}{\beta_{j1}\beta_{i2}-\beta_{j2}\beta_{i1}}\Big)-\theta\Big(\frac{\beta_{j1}\,\mathrm{Im}(\alpha_n)-\beta_{n1}\,\mathrm{Im}(\alpha_j)}{\beta_{j1}\beta_{n2}-\beta_{j2}\beta_{n1}}\Big)\right\}\Bigg)\\ &\qquad\times\frac{(2\pi i)^2\,(\beta_{j2}\beta_{i1}-\beta_{j1}\beta_{i2})^{n-3}}{B_{ijn}\prod_{l\neq i,j,n}B_{ijl}}.\end{aligned} \tag{2.14}$$

Dieses Zwischenergebnis ist für eine Implementierung eminent geeignet. Es muss nur eine Darstellung für die θ-Funktionen geschrieben werden, die rationalen Koeffizienten bleiben übersichtlich und wie auf Seite 33 erläutert in der Anzahl beschränkt. Zweckmäßigerweise

können die Faktoren $(2\pi i)^2$ sowie ganzzahlige Faktoren in $P_i(l_1^{(j)}, l_2^{(k)})$ herausdividiert und getrennt abgespeichert werden,[4] wenn man eine passende Datenstruktur wählt, die Zähler und Nenner in voneinander getrennten Feldern verwaltet.

Mögliche Entartungen

Eine ständig auftretende Entartung in diesem Verfahren beruht darauf, dass wir unseren inversen Propagatoren P immer identische kausale Imaginärteile $i\rho$ mitgegeben haben. Wir hätten diese auch verschieden wählen können, was aber neben der Einführung zusätzlicher symbolischer Variablen eine aufwändige Fallunterscheidung zur Folge hätte. In den θ-Funktionen würden dann nämlich immer Terme der Form $\rho_i - \rho_j$ vorkommen von denen das Vorzeichen zu bestimmen wäre. Ist aber $\rho_i = \rho_j$, so kommt es bei der Einsetzung der Polstellen während einer Residuenintegration durch Differenzbildung bisweilen dazu, dass ein Imaginärteil völlig verschwindet. Dann wird eine darauffolgende Residuenintegration entlang der reellen Achse durchgeführt mit einem Pol direkt auf dem Integrationsweg. Betrachtet man dieses Integral als Hauptwertintegral, so kann man die Integration aber mit dem Satz A.6 durchführen. In allen obigen Gleichungen ist dann einfach $\theta(0)$ durch $1/2$ zu ersetzen.

Bei dem Auftreten von Determinanten im Nenner stellt sich natürlich immer die Frage, ob diese nicht vielleicht verschwinden. Zunächst einmal gibt es den Fall, dass in einer Matrix zwei Zeilen identisch sind:

$$\begin{pmatrix} P_i(l_1,l_2) \\ P_j(l_1,l_2) \\ P_k(l_1,l_2) \end{pmatrix} = \begin{pmatrix} \beta_{i1} & \beta_{i2} & \alpha_i \\ \beta_{i1} & \beta_{i2} & \alpha_i \\ \beta_{k1} & \beta_{k2} & \alpha_k \end{pmatrix} \begin{pmatrix} l_1 \\ l_2 \\ 1 \end{pmatrix}.$$

Dies kann in Feynmandiagrammen passieren, wenn zwischen zwei identischen skalaren Propagatoren ein äußeres Teilchen anknüpft ohne Viererimpuls zu übertragen. Das bedeutet natürlich $P_i = P_j$ oder aber einen quadrierten Propagator P_i in der ursprünglichen Amplitude, was jedoch nach Gleichung (A.9) noch mit einer Residuenintegration handhabbar ist. (Topologien mit der Schleifeneinsetzung ⟩○⟨, wie etwa I d) in Abbildung 2.2, sind natürlich auch betroffen. Diese können aber mit Methoden aus dem Standardrepertoire behandelt werden, etwa als Dispersionsintegral.)

Problematischer wird es, wenn eine Teilentartung auftritt, wie zum Beispiel

$$\begin{pmatrix} P_i(l_1,l_2) \\ P_j(l_1,l_2) \\ P_k(l_1,l_2) \end{pmatrix} = \begin{pmatrix} \beta_{i1} & \beta_{i2} & \alpha_i \\ \beta_{i1} & \beta_{i2} & \alpha_j \\ \beta_{k1} & \beta_{k2} & \alpha_k \end{pmatrix} \begin{pmatrix} l_1 \\ l_2 \\ 1 \end{pmatrix},$$

also $\beta_{j1} = \beta_{i1}$ und $\beta_{j2} = \beta_{i2}$, aber $\alpha_i \neq \alpha_j$, um den schon behandelten Fall quadrierter Propagatoren auszuschließen. Man beachte, dass die Determinante dann nicht einmal verschwindet. Ein Feynmandiagramm mit externen Teilchen ohne Impulsübertrag aber verschiedenen internen Massen entspricht diesem Fall. Dabei geht zum Beispiel nach der l_1-Integration ein

[4] In GiNaC bietet es sich an, hierfür `integer_content()` zu missbrauchen. Es ist aber nur für expandierte multivariate Polynome über $\mathbb{Z}$ definiert. Aufgrund der Imaginärteile wird dem Aufruf daher eine Zerlegung in Real- und Imaginärteil mittels eines Funktors vorangesetzt.

Propagator verloren, da nun $P_2(l_1^{(1)})$ und $P_1(l_1^{(2)})$ keine l_2-Abhängigkeit mehr enthalten. Verzichten wir auf die doppelte Integration in Zwillingsvariablen und führen die l_1-Integration durch, so erhalten wir:

$$\begin{aligned}\int dl_1 & \frac{1}{P_i(l_1,l_2)P_j(l_1,l_2)P_k(l_1,l_2)} \\ = \quad & (2\pi i)\Big(\frac{\beta_{i1}\{\theta(-\mathrm{Im}(\alpha_i)/\beta_{i1}) - \theta(-\mathrm{Im}(\alpha_k)/\beta_{k1})\}}{(\alpha_j-\alpha_i)((\beta_{i1}\beta_{k2}-\beta_{k1}\beta_{i2})l_2 + (\alpha_k\beta_{i1}-\alpha_i\beta_{k1}))} \\ & \quad + \frac{\beta_{i1}\{\theta(-\mathrm{Im}(\alpha_j)/\beta_{i1}) - \theta(-\mathrm{Im}(\alpha_k)/\beta_{k1})\}}{(\alpha_i-\alpha_j)((\beta_{i1}\beta_{k2}-\beta_{k1}\beta_{i2})l_2 + (\alpha_k\beta_{i1}-\alpha_j\beta_{k1}))}\Big).\end{aligned}$$

Die nachfolgende Integration über l_2 ist hier nicht mehr ausführbar, es sei denn es waren ursprünglich noch mehr $P(l_1,l_2)$ vorhanden, die die Konvergenz wieder herstellen. Die obige Summe sieht nur auf den ersten Blick integrabel aus, wenn man sie auf den Hauptnenner bringt. Die Differenz der dort auftretenden $P_3(l_1^{(1)})$ und $P_3(l_1^{(2)})$ im Zähler wird dann zwar auch frei von l_2. Die θ-Funktionen im Zähler vereiteln dieses Vorhaben jedoch leider, so dass dies jeweils nur in Teilgebieten möglich ist.

2.3. Umformung der einschränkenden Bedingungen

Die einschränkenden Bedingungen, also die θ-Funktionen in Gleichung (2.14), können nicht als algebraische Ausdrücke im allgemeinsten Sinne (beispielsweise in der GiNaC-Klasse `ex`) abgespeichert werden, da wir sie besonders untersuchen und den anonymen ⇒ Evaluator daher umgehen müssen. Lediglich ihre Argumente als rationale Funktionen sind allgemeine algebraische Ausdrücke. Stattdessen enthält jeder additive Term in der Amplitude nach der Residuenintegration ein eigens dafür vorgesehenes Objekt einer Klasse `constraint`, welches die Logik implementiert. Die Klasse `constraint` stellt semantisch gesehen selbst eine Summe aus Produkten von Differenzen von θ-Funktionen dar. Da insgesamt vier Residuenintegrationen ausgeführt werden von denen jede eine Differenz der Form $\theta(-\mathrm{Im}(\alpha_i)/\beta_i) - \theta(-\mathrm{Im}(\alpha_n)/\beta_n)$ mit sich bringt, haben die Produkte maximal vier Terme. Von diesen Produkten wiederum gibt es am Ende jeweils vier Stück, da über zwei Paare von Zwillingsvariablen integriert wird, von denen laut Lemma 2.2 jeweils zwei additive Terme sich nur durch ein Vorzeichen unterscheiden.

Diese rigide Form legt eine Implementierung nahe, in der Addition und Multiplikation von Hand den speziellen Bedürfnissen angepasst werden. Hierfür wurden ein paar Klassen definiert, die in ineinander verschachtelter Form benutzt werden. Von innen nach außen sind dies `constraint_atom`, das eine Differenz aus zwei θ-Funktionen darstellt. Ein `constraint_atom` kann im uninitialisierten Status sein, in dem sie semantisch als Eins zu lesen ist, da noch keine einschränkenden Bedingungen aufgetreten sind. Die Methode `constraint_atom::empty()` gestattet es, diesen Zustand abzufragen. Die Containerklasse `constraint_product` ist ein Quadrupel von `constraint_atom`. Und die Containerklasse `constraint` schließlich ein Quadrupel von `constraint_product`. Man muss nur die bei deren Konstruktion benötigten Rechenregeln für Addition und Multiplikation explizit programmieren.

Zunächst einmal können positive Faktoren aus den Argumenten der θ-Funktionen eliminiert werden:

$$\theta(n \cdot P) \longrightarrow \theta(P). \tag{2.15}$$

Für ganzzahlige $n > 0$ und voll expandierte Polynome über $\mathbb{Z}$ leistet eine Division durch den Rückgabewert der GiNaC-Funktion `integer_content()` den gewünschten Effekt.

Falls die Argumente der θ-Funktionen quadratisch vorkommen, so kann man sie eliminieren, da sie immer aus Imaginärteilen aufgebaut werden und daher per Konstruktion rein reell sind. Allgemeiner kann man dies mithilfe der quadratfreien Faktorisierung erreichen, die wir nun kurz skizzieren. Enthält ein Polynom

$$P(x) = P_i(x) P_{ii}^2(x) P_{iii}^3(x) P_{iv}^4(x) \cdots \tag{2.16}$$

einen Faktor n-fach, so enthält die Ableitung diesen Faktor $n-1$-fach:

$$\begin{aligned} P'(x) \;=\; & \big(P_i'(x)P_{ii}(x)P_{iii}(x)P_{iv}(x)\cdots \\ & + 2P_i(x)P_{ii}'(x)P_{iii}(x)P_{iv}(x)\cdots \\ & + 3P_i(x)P_{ii}(x)P_{iii}'(x)P_{iv}(x)\cdots \\ & + 4P_i(x)P_{ii}(x)P_{iii}(x)P_{iv}'(x)\cdots \\ & + \cdots\big)\, P_{ii}(x)P_{iii}^2(x)P_{iv}^3(x)\cdots . \end{aligned}$$

Durch iterative Bestimmung des ggT eines Polynoms und seiner Ableitung kann so leicht eine Faktorisierung der Art (2.16) erreicht werden, wobei die einzelnen $P_n(x)$ noch nicht voll faktorisiert sein müssen – das Polynom $x^2 - 1$ kann mit dieser Methode beispielsweise nicht faktorisiert werden, da der ggT von $x^2 - 1$ und $2x$ 1 ist. Im multivariaten Fall findet man zunächst eine beliebige Variable x auf und sucht die Faktoren in der Ableitung nach x. Man wiederholt das Verfahren dann in jedem der aufgefundenen Faktoren und in einer nächsten Variablen so lange, bis keine freien Variablen mehr aufgefunden werden. Durch diese rekursive Anwendung wird die quadratfreie Faktorisierung in der Praxis besser, je mehr Variable vorhanden sind und je niedriger der führende Exponent ist. Diese Beobachtung wird gleich in Satz 2.6 präzisiert werden.

Diese quadratfreie Faktorisierung erlaubt eine programmatische Vereinfachung der Argumente der θ-Funktionen:

$$\theta(P_i \cdot P_{ii}^2 \cdot P_{iii}^3 \cdots) \longrightarrow \theta(P_i \cdot P_{iii} \cdots). \tag{2.17}$$

Diese Umformung ist offensichtlich richtig, da $\text{sign}(P_{ii}^2) = 1$ und mithin $\text{sign}(P_{iii}^3) = \text{sign}(P_{iii})$, etc.

Hat man nun eine θ-Funktion mit einem Produkt als Argument, so kann man diese als Summe von θ-Funktionen umschreiben. $\theta(P_i P_{ii})$ ist genau dann 1, wenn P_i und P_{ii} entweder beide positiv oder beide negativ sind:

$$\theta(P_i P_{ii}) = \theta(P_i)\theta(P_{ii}) + \theta(-P_i)\theta(-P_{ii}).$$

Zusätzlich gilt

$$\theta(-x) = 1 - \theta(x), \tag{2.18}$$

womit man dies weiter umschreiben kann zu:

$$\begin{aligned}\theta(P_iP_{ii}) &= \theta(P_i)\theta(P_{ii}) + (1-\theta(P_i))(1-\theta(P_{ii}))\\ &= 2\theta(P_i)\theta(P_{ii}) - \theta(P_i) - \theta(P_{ii}) + 1.\end{aligned}$$

Falls hierin P_{ii} wieder in ein Produkt faktorisieren sollte, kann man das Verfahren wiederholen und elegant als rekursive Prozedur programmieren. Die ersten paar Ergebnisse lauten:

$$\left.\begin{aligned}\theta(P_iP_{ii}) &= 2\theta(P_i)\theta(P_{ii}) - \theta(P_i) - \theta(P_{ii}) + 1\\ \theta(P_iP_{ii}P_{iii}) &= 4\theta(P_i)\theta(P_{ii})\theta(P_{iii})\\ &\quad - 2\theta(P_i)\theta(P_{iii}) - 2\theta(P_i)\theta(P_{ii}) - 2\theta(P_{ii})\theta(P_{iii})\\ &\quad + \theta(P_i) + \theta(P_{ii}) + \theta(P_{iii})\\ \theta(P_iP_{ii}P_{iii}P_{iv}) &= 8\theta(P_i)\theta(P_{ii})\theta(P_{iii})\theta(P_{iv})\\ &\quad - 4\theta(P_i)\theta(P_{ii})\theta(P_{iii}) - 4\theta(P_i)\theta(P_{ii})\theta(P_{iv})\\ &\quad - 4\theta(P_i)\theta(P_{iii})\theta(P_{iv}) - 4\theta(P_{ii})\theta(P_{iv})\theta(P_{iii})\\ &\quad + 2\theta(P_i)\theta(P_{ii}) + 2\theta(P_i)\theta(P_{iii}) + 2\theta(P_i)\theta(P_{iv})\\ &\quad + 2\theta(P_{ii})\theta(P_{iii}) + 2\theta(P_{ii})\theta(P_{iv}) + 2\theta(P_{iii})\theta(P_{iv})\\ &\quad - \theta(P_i) - \theta(P_{ii}) - \theta(P_{iii}) - \theta(P_{iv}) + 1\\ &\vdots\end{aligned}\right\} \qquad (2.19)$$

Die mit dieser Prozedur gewonnenen Polynome aus θ-Funktionen können Quadrate enthalten. Man kann sie jedoch alle eliminieren, da die θ-Funktion idempotent ist. Es gilt die Vereinfachungsregel der Tautologie:

$$\theta(P)^n = \theta(P). \qquad (2.20)$$

Im Fall $P = 0$ wäre sie falsch, da $\theta(0) = 1/2$. Falls P manifest verschwindet, ist die Vereinfachung $\theta(P) \to 1/2$ aber schon ausgeführt worden und die Regel kommt gar nicht erst zur Anwendung. Falls P nicht manifest verschwindet, sondern nur in Grenzfällen wenn P zu einem späteren Zeitpunkt an bestimmten kinematischen Punkten symbolisch ausgewertet wird, dann liegt genau eine Gebietsgrenze vor. Diese stellt aber eine Menge vom Maß 0 dar und ist daher für die verbleibenden Integrationen irrelevant – was die Anwendung der Idempotenzregel auch wieder sicher macht.

In den beiden Umformungsregeln (2.18) und (2.20) ist übrigens auch der logische Widerspruch automatisch beinhaltet:

$$\begin{aligned}\theta(P)\theta(-P) &\overset{(2.18)}{\longrightarrow} \theta(P)(1-\theta(P)) = \theta(P) - \theta(P)^2\\ &\overset{(2.20)}{\longrightarrow} \theta(P) - \theta(P) = 0.\end{aligned}$$

Falls sich beim Leser bis hierhin der Verdacht eingeschlichen haben sollte, dass es sich bei all dem um eine boolsche Prädikatenlogik handelt, so muss er an dieser Stelle ausgeräumt werden. Die Argumente der θ-Funktionen entsprechen zwar Prädikaten in dem Sinne, dass $\theta(P) = 1 \Leftrightarrow P > 0$, und das Produkt $\theta(P_i)\theta(P_{ii})$ kann darin als logisches „Und“ ($\wedge$) geschrieben werden, aber die Summe $\theta(P_i) + \theta(P_{ii})$ ist kein Äquivalent zum logischen „Oder“ ($\vee$), da sie für $P_i > 0$ und $P_{ii} > 0$ den Wert 2 annehmen kann. Unter anderem folgt daraus, dass wir

auch kein Äquivalent zu den de Morgan'schen Regeln $\neg a \wedge \neg b \Leftrightarrow \neg(a \vee b)$ und $\neg a \vee \neg b \Leftrightarrow \neg(a \wedge b)$ unter unseren Instrumenten zur Termumschreibung vorfinden.

Sollte nach diesen Vereinfachungen das Polynom von θ-Funktionen faktorisieren, so kann man diese Faktorisierung auf jeden Fall mit dem Algorithmus für quadratfreie Faktorisierung auffinden:

Satz 2.6 *Gegeben sei ein Polynom $P(\underset{\sim}{t})$ über dem Ring der ganzen Zahlen in n freien Variablen $\underset{\sim}{t} = \{t_1, t_2, \ldots, t_n\}$. Wenn alle $t_j \in \underset{\sim}{t}$ die Idempotenz $t_j = t_j^2$ erfüllen, dann ist die quadratfreie Faktorisierung von $P(\underset{\sim}{t})$ nach Anwendung der Idempotenz identisch mit der vollen Faktorisierung.*

Zum Beweis gehen wir von der vollen Faktorisierung $P_i(\underset{\sim}{t})P_{ii}(\underset{\sim}{t})P_{iii}(\underset{\sim}{t})\cdots$ von $P(\underset{\sim}{t})$ aus und zeigen, dass der Algorithmus der quadratfreien Faktorisierung tatsächlich alle Faktoren auffinden kann. Es ist klar, dass bei Polynomen über den ganzen Zahlen eine beliebige Variable $t \in \underset{\sim}{t}$ nur in einem der Faktoren $P_i(\underset{\sim}{t}), P_{ii}(\underset{\sim}{t}), \ldots$ vorkommen kann, weil sonst in der ausmultiplizierten Form mindestens t^2 auftreten würde (da der Ring der ganzen Zahlen die Charakteristik 0 hat). Wir zerlegen $\underset{\sim}{t}$ in $\underset{\sim}{t} = t \cup \underset{\sim}{\tau}$ und nehmen o.B.d.A. an, dass t in P_i vorkommt. Dann ist

$$P(\underset{\sim}{t}) = (p(\underset{\sim}{\tau})t + q(\underset{\sim}{\tau}))P_{ii}(\underset{\sim}{\tau})P_{iii}(\underset{\sim}{\tau})\cdots$$

mit zwei noch unbekannten Polynomen $p(\underset{\sim}{\tau})$ und $q(\underset{\sim}{\tau})$. Für die Ableitung von $P(\underset{\sim}{t})$ nach t gilt

$$\tfrac{d}{dt}P(\underset{\sim}{t}) = p(\underset{\sim}{\tau})P_{ii}(\underset{\sim}{\tau})P_{iii}(\underset{\sim}{\tau})\cdots$$

und für den ggT

$$\mathrm{ggT}\big(P(\underset{\sim}{t}), \tfrac{d}{dt}P(\underset{\sim}{t})\big) = \mathrm{ggT}\big(p(\underset{\sim}{\tau}), q(\underset{\sim}{\tau})\big)\, P_{ii}(\underset{\sim}{\tau})P_{iii}(\underset{\sim}{\tau})\cdots$$

Wenn $\mathrm{ggT}(p(\underset{\sim}{\tau}), q(\underset{\sim}{\tau})) = 1$, dann ist $P_i(\underset{\sim}{t})$ gefunden. Die Variable t kommt im Rest nicht mehr vor und daher kann es keine weiteren Faktoren in t mehr geben. Andererseits hat $P_i(\underset{\sim}{t}) = (p(\underset{\sim}{\tau})t + q(\underset{\sim}{\tau}))$ dann und nur dann noch weitere Faktoren, wenn $\mathrm{ggT}(p(\underset{\sim}{\tau}), q(\underset{\sim}{\tau})) \neq 1$ ist. Diese werden dann aber durch Anwendung des Verfahrens in den verbleibenden Variablen $\underset{\sim}{\tau}$ gefunden. Durch Iteration über alle $t \in \underset{\sim}{t}$ werden so alle Faktoren sukzessive gefunden – unabhängig von der gewählten Reihenfolge der t_j. □

Dieser Satz ist zwar sehr hilfreich beim Aufsuchen einer solchen Faktorisierung – er kommt jedoch mit einem kleinen Wermutstropfen einher: Es wird davon ausgegangen, dass das zu faktorisierende Polynom in expandierter Form und nach Anwendung der Idempotenz $t_i^2 = t_i$ vorliegt. Die Anwendung der Idempotenz führt zwar zu einer Vereinfachung in dem Sinne, dass im expandierten Darstellungsbaum des Polynoms keine neuen Äste entstehen sondern nur welche verschwinden; sie kann eine faktorisierende Struktur jedoch auch zerstören. Aus $(t_1 + t_2)(t_1 + t_3)$ wird zum Beispiel $t_1 + t_1t_3 + t_2t_1 + t_2t_3$ und das faktorisiert überhaupt nicht mehr. Andererseits ist die faktorisierende Struktur immerhin invariant unter Transformationen der Form $\theta(x) \to 1 - \theta(-x)$, da $\theta(-x)$ beziehungsweise $\theta(x)$ in genau einem der Faktoren vorkommt: Schreiben wir $t := \theta(x)$ und $t' := \theta(-x)$, so findet die quadratfreie Faktorisierung $p(\underset{\sim}{\tau}) - p(\underset{\sim}{\tau})t' + q(\underset{\sim}{\tau})$ genauso als Faktor auf wie das ursprüngliche $p(\underset{\sim}{\tau})t + q(\underset{\sim}{\tau})$.

Nun erzwingt die Anwesenheit von äußeren Impulskomponenten in den θ-Funktionen leider die Erstellung eines immensen Entscheidungsbaumes. An jedem Zweig würden dann Polynome aus äußeren Impulskomponenten nach ihrem Vorzeichen unterschieden. Selbst wenn man diesen (exponentiellen) Aufwand nicht scheut, ist es nach dem derzeitigen Stand von Inferenzmaschinen für Entscheidungen in symbolischen Gleichungssystemen recht zweifelhaft, ob die richtigen Tautologien und Widersprüche in dem System von θ-Funktionen für eine Vereinfachung gefunden werden können [WeGo 1991]. Aus diesem Grunde muss ab nun ein pragmatischer Standpunkt eingenommen werden: die äußeren Impulskomponenten sollen alle numerisch vorliegen.

Unter dieser Voraussetzung kann eine weitere nützliche Vereinfachung vorgenommen werden. Wir haben bisher die Vorzeichen der Argumente der θ-Funktionen beliebig nach Gleichung (2.18) gewählt. Sehr häufig ist aber eine Wahl nützlicher als die andere. Wenn nämlich $P_i - P_{ii}$ keine unbekannten Symbole mehr enthält, also numerisch als reelle Zahl vorliegt, dann wird durch $\theta(P_i) - \theta(P_{ii})$ ein Streifen im Raum der in P_i vorkommenden zugeordneten Schleifenvariablen definiert. Wenn hingegen $P_i + P_{ii}$ numerisch ist, dann definiert $\theta(P_i) - \theta(-P_{ii})$ einen Streifen im Raum der zugeordneten Variablen. Dann wird aber je nach Vorzeichen von $P_i + P_{ii}$ beziehungsweise $P_i - P_{ii}$ durch $\theta(P_i)\theta(-P_{ii})$ beziehungsweise $\theta(-P_i)\theta(P_{ii})$ jeweils dasselbe Gebiet definiert. Wir können nun testweise (2.18) anwenden um solche einfacheren Faktoren zu finden. Dies alles ist noch kein strategisches Vorgehen und insbesondere kein Algorithmus. Ein solcher muss unbedingt noch gefunden werden. Im Folgenden wird die Anwendung dieser Instrumente zur Termumschreibung zur Veranschaulichung der Problematik durchgeführt.

Beispiele für Umformungen einschränkender Bedingungen

Wir zeigen nun an einem einfachen aber realistischen Beispiel, wie die bisher beschriebenen Umformungen in der Lage sind, die einschränkenden Bedingungen nahezu automatisch in eine geeignete Gestalt zu bringen, so dass die durch sie definierten Gebiete direkt abgelesen werden können. Allerdings ist es noch nicht gelungen, die oben in diesem Abschnitt aufgezählten Werkzeuge so zu dirigieren, dass dieser Prozess völlig automatisiert abläuft. Dies ist endgültig aber notwendig, da die Rechnungen per Hand viel zu zeitraubend und fehleranfällig sind.

Wir beschränken uns der Übersichtlichkeit zuliebe auf zwei zugeordnete Variablen $\tilde{k}$ und $\tilde{l}$, welche nach der Residuenintegration in k und l durch Bedingungen eingeschränkt werden. Ein häufiges Muster nach zweimaliger Anwendung von Gleichung (2.8) ist beispielsweise das folgende Produkt von θ-Funktionen:

$$\Theta := \big\{\theta(-\tilde{l}) - \theta(-\tilde{k}-\tilde{l})\big\}\big\{\theta(-\tilde{k}-p) - \theta(\tilde{k}\,\tilde{l}\,(\tilde{k}+\tilde{l}))\big\} \tag{2.21}$$

mit einer externen Impulskomponente p, für die o.B.d.A. $p > 0$ gelten mag. In [Krec 1997a] steckt es in Gleichungen (2.4) und (2.5) und wurde per Hand wie folgt analysiert. Der erste Term $\{\theta(-\tilde{l}) - \theta(-\tilde{k}-\tilde{l})\}$ schält aus der $\tilde{k}$-$\tilde{l}$-Ebene zwei Dreiecke heraus, eines davon mit positivem und eines mit negativem Gewicht (Abbildung 2.6 links). Eines davon ist im zweiten, das andere im vierten Quadranten, also ist das Produkt $\tilde{k}\,\tilde{l}$ stets negativ. Dies kann nun in den zweiten Term $\{\theta(-\tilde{k}-p) - \theta(\tilde{k}\,\tilde{l}\,(\tilde{k}+\tilde{l}))\}$ eingesetzt werden, um ihn als $\{\theta(-\tilde{k}-p) - \theta(-\tilde{k}-\tilde{l})\}$ zu

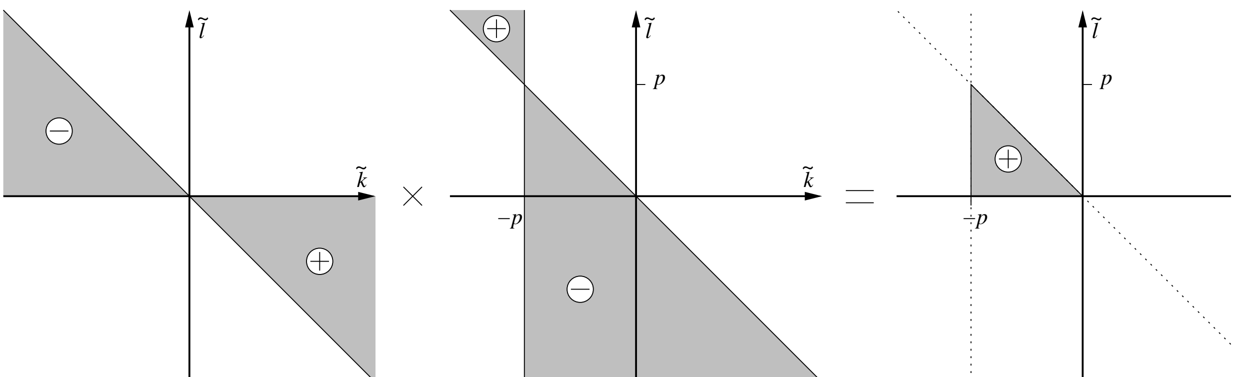


Abbildung 2.6.: *Graphische Darstellung des faktorisierten Polynoms in θ-Funktionen (2.21).*

schreiben. Dies schält dann die beiden Dreiecke aus Abbildung 2.6 mitte heraus. Das Produkt ist das endliche Dreieck von Abbildung 2.6 rechts, begrenzt durch $\tilde{l} = -\tilde{k}$, $\tilde{k} = -p$ und $\tilde{l} = 0$. Mit den Umformungen aus diesem Abschnitt kann man dieses Ergebnis auch reproduzieren. Es wird sich herausstellen, dass es dabei eine für das automatische Identifizieren von Gebieten wesentlich besser geeignete Form annimmt. Ausgehend von Gleichung (2.21) identifizieren wir die vier vorkommenden Polynome

$$\begin{aligned} P_i &= \tilde{k} \\ P_{ii} &= -\tilde{k} - p \\ P_{iii} &= \tilde{l} \\ P_{iv} &= -\tilde{k} - \tilde{l}. \end{aligned}$$

Damit hat Θ die Gestalt $\{\theta(-P_{iii}) - \theta(P_{iv})\}\{\theta(P_{ii}) - \theta(-P_i P_{iii} P_{iv})\}$. Mit der Zerlegung (2.19) für $\theta(-P_i P_{iii} P_{iv})$ erhält man

$$\begin{aligned} \Theta = &\big\{1 - \theta(P_{iii}) - \theta(P_{iv})\big\} \\ &\big\{-1 + \theta(P_i) + \theta(P_{ii}) + \theta(P_{iii}) + \theta(P_{iv}) \\ &- 2\theta(P_i)\theta(P_{iii}) - 2\theta(P_i)\theta(P_{iv}) - 2\theta(P_{iii})\theta(P_{iv}) + 4\theta(P_i)\theta(P_{iii})\theta(P_{iv})\big\}. \end{aligned}$$

Durch Ausmultiplikation dieses Produktes entsteht ein etwas unübersichtliches Polynom aus θ-Funktionen:

$$\begin{aligned} \Theta = &-1 + \theta(P_i) + \theta(P_{ii}) + 2\,\theta(P_{iii}) + 2\,\theta(P_{iv}) \\ &- \theta(P_{iii})^2 - \theta(P_{iv})^2 - \theta(P_{ii})\theta(P_{iii}) - \theta(P_{ii})\theta(P_{iv}) \\ &- 3\,\theta(P_i)\theta(P_{iii}) - 3\,\theta(P_i)\theta(P_{iv}) - 4\,\theta(P_{iii})\theta(P_{iv}) \\ &+ 2\,\theta(P_i)\theta(P_{iii})^2 + 2\,\theta(P_i)\theta(P_{iv})^2 + 2\,\theta(P_{iii})^2\theta(P_{iv}) + 2\,\theta(P_{iii})\theta(P_{iv})^2 \\ &+ 8\,\theta(P_i)\theta(P_{iii})\theta(P_{iv}) - 4\,\theta(P_i)\theta(P_{iii})^2\theta(P_{iv}) - 4\,\theta(P_i)\theta(P_{iii})\theta(P_{iv})^2, \end{aligned}$$

was nach Ausnutzen der Idempotenz und Zusammenfassen von gleichen Termen zusammenschrumpft zu:

$$\begin{aligned} \Theta = &-1 + \theta(P_i) + \theta(P_{ii}) + \theta(P_{iii}) + \theta(P_{iv}) \\ &- \theta(P_i)\theta(P_{iii}) - \theta(P_i)\theta(P_{iv}) - \theta(P_{ii})\theta(P_{iii}) - \theta(P_{ii})\theta(P_{iv}). \end{aligned} \tag{2.22}$$

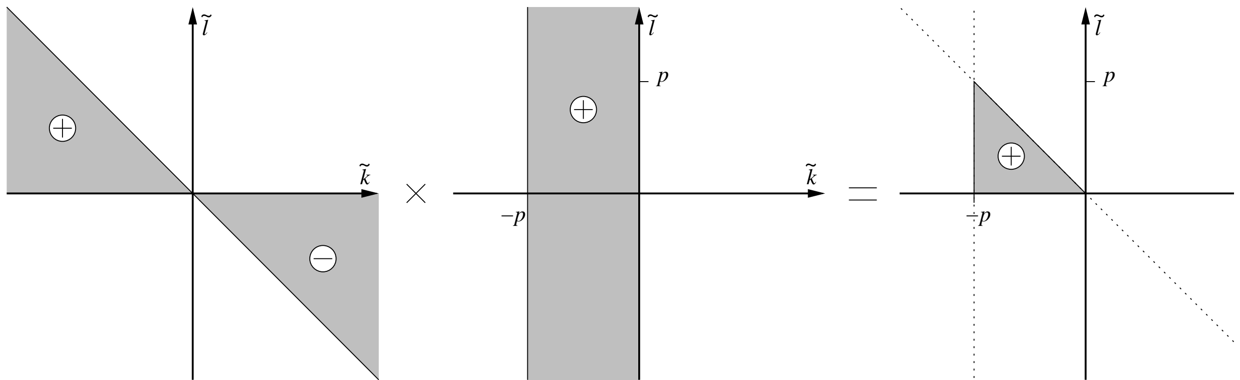


Abbildung 2.7.: *Alternative, (2.23) entsprechende Darstellung des Polynoms in θ-Funktionen aus Abbildung 2.6.*

Dieses Ergebnis faktorisiert in $\Theta = \{\theta(P_{iii}) + \theta(P_{iv}) - 1\}\{1 - \theta(P_i) - \theta(P_{ii})\}$, was eigentlich erstaunlich ist, da es gegenüber (2.21) eine Vertauschung beschreibt: Es ist ausgeschrieben

$$\Theta = \big\{\theta(\tilde{l}) + \theta(-\tilde{k}-\tilde{l}) - 1\big\}\big\{1 - \theta(\tilde{k}) - \theta(-\tilde{k}-p)\big\} \tag{2.23}$$

und die beiden Anteile lassen sich wie in Abbildung 2.7 veranschaulichen. Der linke Anteil ist derselbe geblieben, aber der rechte ist in einen durch $\tilde{k} = -p$ und $\tilde{k} = 0$ beschränkten Streifen übergegangen. Es ist dies eine Umformung, die nicht möglich und auch nicht richtig gewesen wäre ohne Wechselwirkung mit dem linken Gebiet. Das Endergebnis ist wieder das Dreieck aus Abbildung 2.6, genau wie es sein muss. Außerdem ist für das Erkennen des mittleren Gebietes jetzt keine Information aus dem linken Gebiet mehr notwendig – für Abbildung 2.6 mussten wir noch $\tilde{k}\,\tilde{l} < 0$ aus dem linken Gebiet ablesen um das mittlere zu erkennen.

Wir sind aber noch nicht ganz fertig. Die Polynome P_i und P_{ii} unterscheiden sich außer durch ein Vorzeichen nur durch die numerisch bekannte Konstante p. Unter Ausnutzung von (2.18) können wir die Vorzeichen von P_i und P_{ii} beliebig manipulieren um wahlweise eines von vier Produkten zu erzeugen. Weil wir $p > 0$ angenommen haben, ist ihre geometrische Interpretation auch bekannt:

$$\begin{aligned}
\theta(\tilde{k}+p)\,\theta(-\tilde{k}) &= \quad = \theta(\tilde{k}+p)\,\theta(-\tilde{k}) \\
\theta(\tilde{k}+p)\,\theta(\tilde{k}) &= \quad = \theta(\tilde{k}) \\
\theta(-\tilde{k}-p)\,\theta(-\tilde{k}) &= \quad = \theta(-\tilde{k}-p) \\
\theta(-\tilde{k}-p)\,\theta(\tilde{k}) &= \quad = 0.
\end{aligned}$$

Die zweite und dritte Möglichkeit beschreiben kein endliches Gebiet und die Letzte beschreibt gar kein Gebiet. Wenn die resultierenden Gebiete also beschränkt sind, dann birgt eine Transformation in die erste Vorzeichenwahl die Möglichkeit, die Gebiete in faktorisierter Form aufzufinden. Unsere Vorzeichenwahl ist aber gerade die vierte. Sie wird also testweise durch

die Transformation

$$\begin{aligned} \theta(\tilde{k}) &\longrightarrow 1-\theta(-\tilde{k}) \\ \theta(-\tilde{k}-p) &\longrightarrow 1-\theta(\tilde{k}+p) \end{aligned}$$

modifiziert werden müssen. Damit wird aus (2.22)

$$\Theta = \theta(\tilde{k}+p)\theta(-\tilde{k})\theta(\tilde{l}) + \theta(\tilde{k}+p)\theta(-\tilde{k})\theta(-\tilde{k}-\tilde{l}) - \theta(\tilde{k}+p)\theta(-\tilde{k}).$$

Man kann dies entweder mit der quadratfreien Faktorisierung oder per Hand mit einer doppelten Ersetzung

$$\begin{aligned} \theta(\tilde{k}) &\longrightarrow \tau + \theta(-\tilde{k}-p) \\ \Theta &\longrightarrow \mathrm{coeff}(\Theta,\tau^0) + \mathrm{coeff}(\Theta,\tau^1)\theta(\tilde{k}+p)\theta(-\tilde{k}) \end{aligned}$$

faktorisieren und erhält so

$$\Theta = \big\{\theta(\tilde{l}) + \theta(-\tilde{k}-\tilde{l}) - 1\big\}\theta(-\tilde{k})\theta(\tilde{k}+p).$$

Eine ähnliche Abbildung auf endlich dargestellte θ-Funktionen im Term $\{\theta(\tilde{l}) + \theta(-\tilde{k}-\tilde{l}) - 1\}$ erfordert nun kombinatorisches Zusatzwissen, welches von Hand beigesteuert werden muss, da $P_{iii} + P_{iv}$ nun nicht numerisch ist sondern noch von $\tilde{k}$ abhängt. Damit erhält man das attraktive Ergebnis

$$\Theta = \theta(\tilde{l})\,\theta(-\tilde{k}-\tilde{l})\,\theta(-\tilde{k})\,\theta(\tilde{k}+p),$$

wovon das Dreieck aus Abbildung 2.6 rechts sofort abgelesen werden kann.

Mit einem analogen Vorgehen ist es insbesondere auch möglich, die auftretenden Gebiete der planaren Box ⊞ in der Parametrisierung

$$\begin{aligned} J_{\boxplus} &= \int d^Dk \int d^Dl \, \frac{1}{P(l,m_1)\,P(l-p_1,m_2)\,P(l+p_2,m_3)\,P(k+p_1-p_3,m_4)} \\ &\qquad \times \frac{1}{P(k+p_1,m_5)\,P(k-p_2,m_6)\,P(k+l,m_7)} \\ \text{mit} &\quad \begin{cases} p_1^\mu = (q_1, q_x, 0, 0) \\ p_2^\mu = (q_2, -q_x, 0, 0) \\ p_3^\mu = (q, p_x, p_y, 0) \end{cases} \end{aligned} \tag{2.24}$$

zu finden. Dies funktioniert jedoch noch nicht vollautomatisch. Sie faktorisieren in k_3-l_3- und k_0-l_0-Gebiete, wobei die k_3-l_3-Gebiete die vertrauten Dreiecke (beschränkt durch p_y) beschreiben, während die k_0-l_0-Gebiete wie in Abbildung 2.9 zu liegen kommen. Dies bestätigt eine früher durchgeführte Bestimmung der Gebiete, die noch ohne die Masterformel (2.14) auskommen musste und in der die Gebiete per Hand abgelesen wurden [KKS 1998].

Ein numerischer Vergleich der planaren Box mit [Smir 1999] steht allerdings noch in weiter Ferne. Für ihn müssten die Integrationsgebiete der Vierfachdarstellung erst zuverlässig (sprich: automatisch) in numerisch zugängliche Einheitsgebiete umgewandelt werden. Darauf wird im nächsten Abschnitt etwas eingegangen.

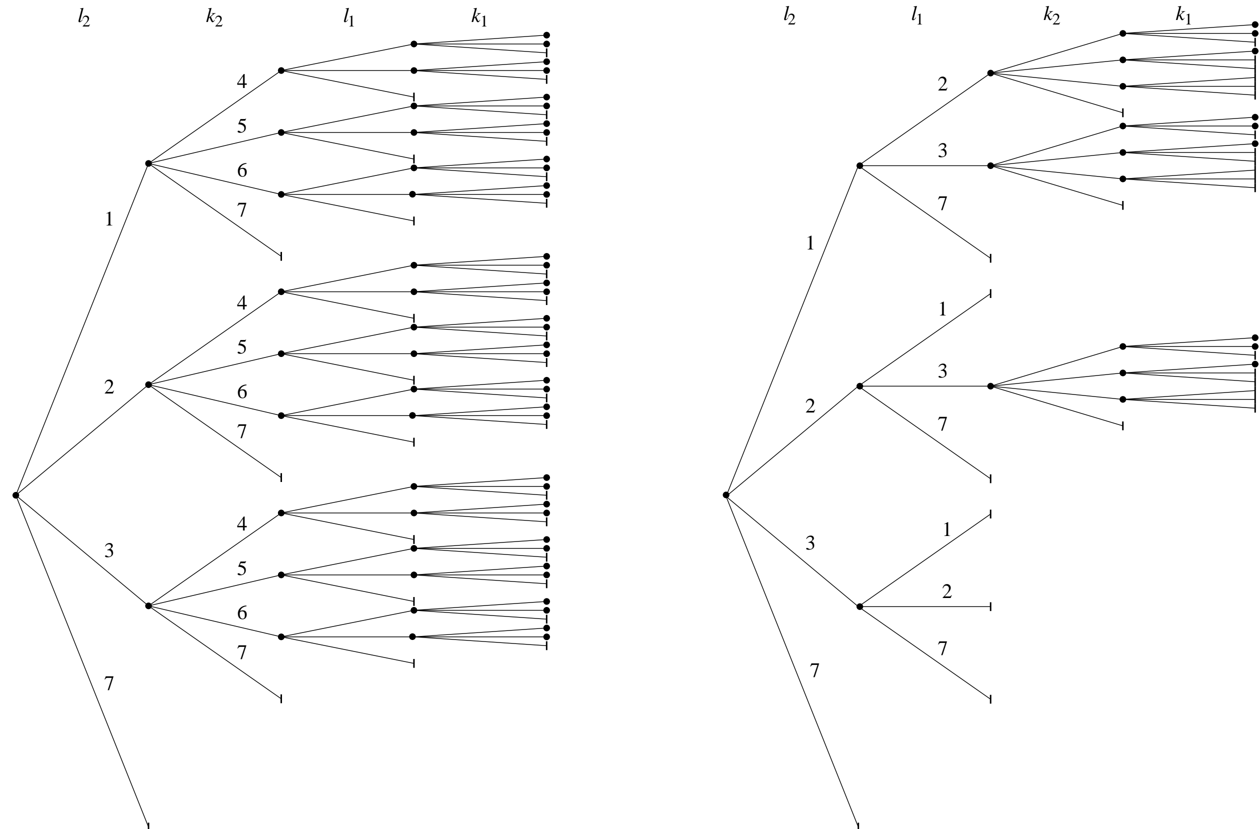


Abbildung 2.8.: *Verbleibende Terme nach zwei verschiedenen Integrationsmethoden zur planaren Box in der Darstellung (2.24): links nur unter Ausnutzung des Satzes über die Residuensumme und rechts unter Ausnutzung der vollen Integrationsregel für Zwillingsvariablen. Die Integrationsreihenfolge ist nicht die gleiche: links kann sie beliebig gewählt werden, rechts müssen l_1 und l_2 sowie k_1 und k_2 hintereinander ausgeführt werden. Dies ist in diesem Fall für das Ergebnis aber nicht von Relevanz.*

2.4. Ausblick: Das weitere Vorgehen

Die so aufgefundenen Gebiete in den zugeordneten Variablen k_3, l_3, k_0 und l_0 sind entweder Dreiecke oder Vierecke, die mit geeigneten linearen Transformationen in den zugeordneten Variablen in Einheitsgebiete überführt werden können. Beispielsweise kann das Dreieck aus Abbildung 2.7 rechts durch die Ersetzung $\tilde{k}/p = \lambda\mu - 1$ und $\tilde{l}/p = \lambda(1-\mu)$ in ein Einheitsquadrat $(0\ldots1)\times(0\ldots1)$ in der λ-μ-Ebene überführt werden (mit der Jacobi-Determinanten $p^2\lambda$). Für Vierecke gilt dies genauso, es müssen stets die Eckpunkte der Gebiete durch Lösen zweidimensionaler symbolischer linearer Gleichungssysteme aufgefunden werden, danach kann durch eine Kombination aus Verschiebung und Scherung die verbleibende Integration in einem Einheitsgebiet dargestellt werden, wobei der Integrand weiterhin eine rationale Funktion (in den neuen Integrationsvariablen) bleibt. Hierfür muss aber zunächst die Erkennung der Gebiete automatisiert werden.

Die Integranden sind im Allgemeinen in den nach dieser Methode anfallenden Gebieten unterhalb der Schwellen einer numerischen Integration mit Werkzeugen aus Anhang B zugänglich. Ob dies oberhalb der Schwellen geht, muss aber bezweifelt werden. In den vorliegenden Inte-

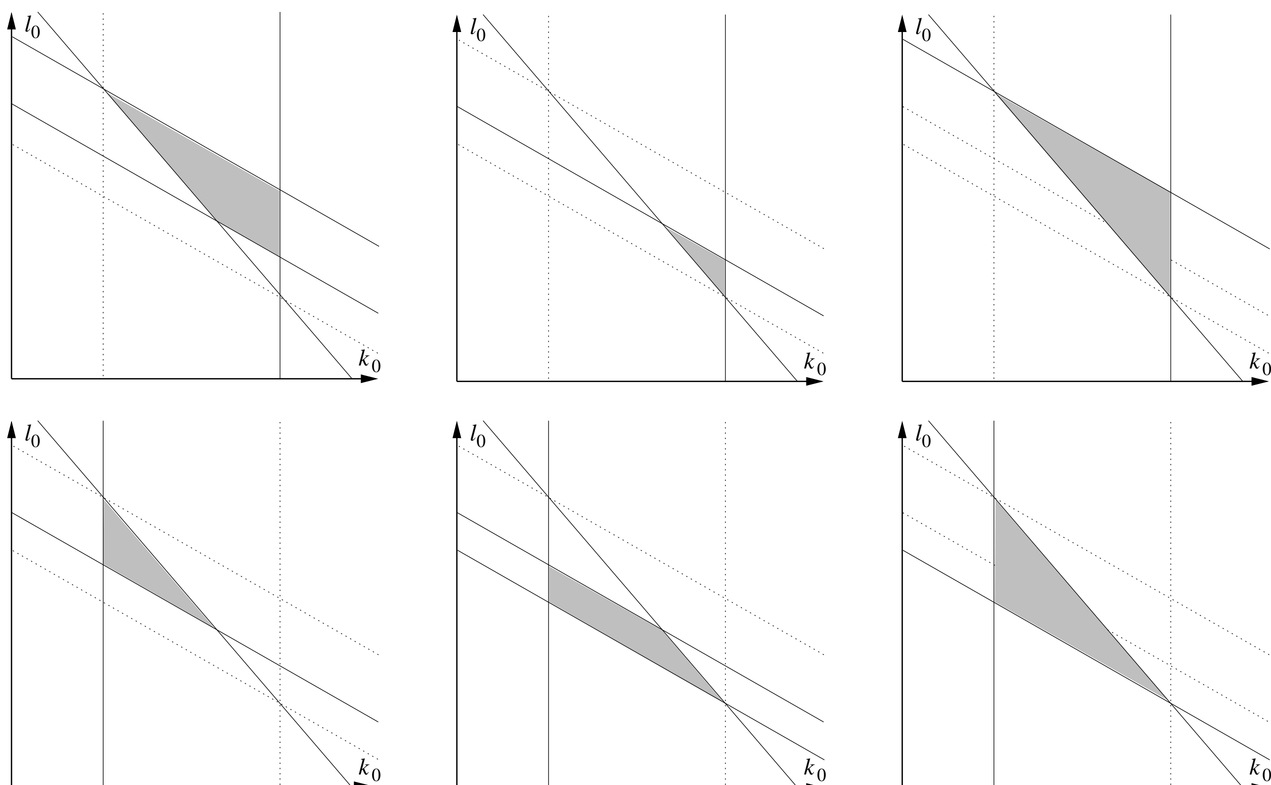


Abbildung 2.9.: *Graphische Darstellung der verbleibenden k_0-l_0 Gebiete bei der planaren Box. Die Parameter der die Gebietsgrenzen beschreibenden Geraden sind außer von äußeren Impulskomponenten noch von k_3 und l_3 abhängig. Von neun Gebieten sind drei verschwunden, da sie zu Widersprüchen in den θ-Funktionen führten. Die k_3-l_3-Gebiete sind einfacher, sie bestehen aus einfachen Dreiecken wie in Abbildung 2.6 bzw. 2.7 und sind nur von äußeren Impulskomponenten parametrisiert.*

gralen wird der Imaginärteil durch die Sokhotsky-Plemelj-Relationen

$$\lim_{\rho\to 0}\int dx \frac{f(x)}{x+x_0\pm i\rho} = \text{P.V.}\int dx \frac{f(x)}{x+x_0} \mp i\pi \int dx\, f(x)\delta(x+x_0) \tag{2.25}$$

erzeugt, wobei die Integrationsgrenzen noch einzusetzen sind. Man erhält sie durch Auftrennen des Integranden in Real- und Imaginärteil, wobei der Imaginärteil eine Grenzwertdarstellung der δ-Distribution ist. Das darin vorkommende Hauptwertintegral (siehe auch Anhang A) ist aber für ein numerisches Programm erfahrungsgemäß wenig zugänglich, da adaptive Methoden zwar bei endlichen Integralen passabel arbeiten, bei solchen, die nicht endlich sind und erst durch die Hauptwertvorschrift endlich gemacht werden, aber kaum konvergieren können. In kinematischen Spezialfällen war es möglich bis zu zwei weitere Integrationen analytisch durchzuführen [KKS 1998]. Dadurch wurde das Integral auch im Realteil oberhalb der Schnitte regularisiert und dort für numerische Verfahren zugänglich gemacht. Die Ergebnisse stimmen mit einer anderen Rechnung [PoTa 1996] überein. Dass dies überhaupt möglich war, lag jedoch an Nebenbedingungen im Parameterraum der Funktion, die die Anzahl der freien Parameter auf drei begrenzten.

Im Allgemeinen muss für eine weitere Integration ein systematisches Verfahren wie etwa das von Horowitz [Horo 1971, DST 1988] verwendet werden. Dieser Algorithmus kann im Prinzip

zuverlässig die Stammfunktionen durch Logarithmen und inverse trigonometrische Funktionen ausdrücken. Allerdings setzt die Implementierung dieses Verfahrens (insbesondere das Einsetzen der Integrationsgrenzen) Kenntnisse über das Vorzeichen von $i\rho$ voraus. Es stellt sich aber heraus, dass der Vorfaktor von $i\rho$ ein Polynom nicht nur in äußeren Impulsvariablen und den inneren Massen, sondern auch in den verbliebenen inneren Schleifenimpulsen ist. Dies wird Aussagen über die Vorzeichen vermutlich unmöglich machen. Unterhalb der ersten kinematischen Schwelle treten alle diese Probleme jedoch noch nicht auf.

Der Wert der hier dargestellten Ergebnisse besteht darin, dass der in [Krei 1994] aufgezeigte Weg zur Berechnung von Vierbeinfunktionen unter Zulassung der gesamten Parametermannigfaltigkeit auf ein berechenbares Maß reduziert wurde. Dies gelingt mit dem Satz über die Residuensumme und der Sylvester-Identität. Ohne diese beiden Hilfsmittel überschreiten die symbolischen Anforderungen auf absehbare Zeit die Kapazität von verfügbaren Rechnern. Die auftretenden Beschränkungen an die verbleibenden vier Integrationen sind zwar noch nicht vollständig automatisiert, jedoch liegt das Werkzeug hierfür nun bereit. Insbesondere konnte gezeigt werden, dass die quadratfreie Faktorisierung für alle notwendigen Operationen ausreichend ist.

Teil II.

Computeralgebra für Schleifenrechnungen

3. GiNaC: Motivation und Design

These problems [of system building] *arise from the desire to build a nearly-autonomous system for mathematical problem representation and solution: the intent is for the system to make it unnecessary for the user to provide detailed programming at the level of data representation of basic mathematical concepts.*
Richard J. Fateman [Fate 1990]

Dieses Kapitel ist der Untersuchung der Grundlagen und praktischer Aspekte der Computeralgebra gewidmet. Es sollen einige allgemeine Funktions- und Designprinzipien angeschnitten werden und die Gründe für das Entstehen von GiNaC dargelegt und strukturelle Implementationsmuster begründet werden. Die Details der Implementierung werden, soweit sie in dieser Arbeit berührt wurden, im nächsten Kapitel beschrieben.

Wenn Computeralgebra zwar kein neues Forschungsgebiet ist[1], so ist es jedenfalls ein immer noch sehr aktives. Auf kommerzieller Seite wird der Computeralgebrasystem- (CAS-[2]) Markt beherrscht von einer kleinen Handvoll Systeme ohne nennenswerte offene Alternativen und geforscht wird – leider – zu einem nicht unbeachtlichen Teil hinter verschlossenen Firmentüren. Ich hoffe, dass es gelingt, in diesem Kapitel aufzuzeigen, warum wir die Abkehr von diesen Systemen im Rahmen des *xloops*-Projektes für notwendig hielten und warum dies der richtige Ansatz ist.

3.1. Die Motivation für GiNaC

Die Probleme bei der Anwendung von handelsüblichen Computeralgebrasystemen fallen in zwei Gruppen: erstens ist für den Benutzer kaum nachvollziehbar, welche Algorithmen und Umformungen vertrauenswürdig sind und welche nicht. Zweitens erfüllen sie in linguistischer Hinsicht nicht den gewachsenen Anforderungen moderner Programmierung. Die Vertrauenswürdigkeit von Algorithmen und Umformungen kann prinzipiell nur einschätzen, wer sie nachvollzogen und verstanden hat oder auf einen Beweis vertraut. Programmierfehler in der Implementation sind hiermit nicht gemeint – selbst bei Einsicht der Quellen sind Implementierungen nur in seltenen Fällen verifizierbar – vielmehr solche „harmlosen“ Umformungen wie $\sqrt{x^2} \to x$,

[1] Charles Babbage hatte schon 1836 die Idee, als er in sein Notitzbuch schrieb: „*This day I had for the first time a general but very indistinct conception of the possibility of making an engine work out algebraic developments—I mean without any reference to the value of the letters.*“ (Zitiert nach [Larc 1999])

[2] Ein besserer Name ist eigentlich das seltener gebrauchte Akronym SAC für *Symbolic and Algebraic Computation.*

die nur bei eingeschränktem Wertebereich gültig sind. Eine Unterscheidung zwischen solchen Algorithmen, die vertrauenswürdig sind und solchen die es nicht sind, ist daher eine Gratwanderung die nur mithilfe der Implementatoren durchführbar ist. C. Bellarin und L. Paulson [BePa 1998] geben ein Beispiel, wo eine Auswahl der von Sumit [Bron 1996b] angebotenen Algorithmen getroffen wird, die vertrauenswürdig genug für das automatisierte Beweisen von Theoremen aus der Codierungstheorie erscheinen – allerdings ohne diese Auswahl anhand einer Negativliste mitsamt Gegenbeispielen zu untermauern.

Anfang der 90er Jahre war kaum ein erhältliches System vor solchen Umformungen wie der obigen gefeit [Stou 1991]. Maple bietet dem Benutzer noch heute die Option `symbolic` um sie trotzdem durchzuführen:

```
1  > simplify(sqrt(x^2));
2                        csgn(x) x
3  > simplify(sqrt(x^2),symbolic);
4                             x
```

Nicht alle Benutzer sind sich aber über die Bedeutung des Schlüsselwortes `symbolic` bewusst und verwenden es um Ausdrücke so weit wie möglich zu „verkleinern". Idealerweise enthält ein System nur völlig unbedenkliche Algorithmen. Dies kollidiert aber zumindest bei kommerziellen Systemen mit Marktanforderungen und so sind diese Systeme eine Ansammlung von Algorithmen, von denen viele *ad hoc* und nicht ganz vertrauenswürdig sind.

Die zweite Problemklasse bilden die linguistischen Einschränkungen und Fallen, die den Programmierer in jedem derzeitigen System verfolgen. Hierfür scheint das Bewusstsein auf Seiten der Implementatoren weitaus weniger ausgeprägt, so dass kaum auf Besserung gehofft werden kann. Der Rest dieses Abschnittes stellt einige dieser Probleme anhand von Maple vor. Die Fokussierung auf Maple entspricht lediglich meiner persönlichen Anwendererfahrung und sollte nicht als Parteiergreifung oder Freibrief für ein anderes System ausgelegt werden.[3]

Ein besonders heimtückischer Fehler in MapleV ist die Verletzung des ⇒ Scopes, des Gültigkeitsbereichs lokaler Variablen. Eine Reihe solcher Scopeverletzungen wurden systematisch in [West 1999] (im treffend „*Mathematics versus Computer Science*" genannten Abschnitt) aufgespürt, indem zunächst einer Variablen ein Wert zugewiesen wurde und sie danach als Laufvariable in Summen, Produkten, Integralen oder Taylor-Reihen benutzt wurde. Alle getesteten CASe mit Ausnahme von Derive hatten darin Schwierigkeiten, globale und lokale Variablen auseinander zu halten. Hier seien noch zwei Beispiele aufgeführt, die noch etwas subtiler sind. Das erste betrifft nur MapleVR4. Es handelt sich um den Versuch, die Reihenentwicklung des Dilogarithmus um den Ursprung zu berechnen:

```
1  i:=1:
2  series(polylog(2,x),x):
3  Error, (in sum) summation variable previously assigned,
4                  second argument evaluates to, 1 = 1 .. 6
```

Hierin wird `i` nicht einmal als Laufvariable vom Benutzer angegeben. Die Funktion `series/polylog` ruft tatsächlich `sum` mit Laufvariable `i` auf, ohne `i` zuvor als lokal zu deklarieren. Dass die Implementatoren selbst darüber stolpern zeigt uns, dass die von Wester

[3] Dies steht in amüsantem Widerspruch zu dem in der Einleitung zu [Stro 1994] genannten Effekt: „*Flaws in the well-known language are deemed minor and simple workarounds are presented, whereas similar flaws in other languages are simply unknown to the people doing the comparison or deemed unsatisfactory.*"

aufgedeckten Probleme keineswegs rein akademisch sind. Es macht auch eindrücklich klar, dass Regressionstests in Systemen ohne lexikalischem Scope zusätzlich durch die Abhängigkeit externer Variablenbindungen erschwert werden.

Noch weitere Probleme dieser Art tauchen auf, wenn Variablennamen mit der `cat()`-Funktion bzw. deren Äquivalent, dem `.`-Operator, zusammengesetzt werden. Das folgende Beispiel zeigt das Problem:

```
 1  Q1 := k:
 2  Q2 := l:
 3
 4  testfun := proc(x,y)
 5      local Q1,Q2;
 6      Q.1 := foo;     # Modifikation globaler Variable!
 7      Q2 := bar;      # Ok: lokale Variable
 8      lprint(Q1, Q2);
 9      RETURN([Q1,Q2]);
10  end:
```

Der Unterschied der beiden Zuweisungen ist nicht offensichtlich. Obwohl sowohl `Q1` als auch `Q2` innerhalb der Prozedur `testfun` expressis verbis lokal definiert wurden, also die globale Definition überschrieben werden sollte, wird in der ersten Zuweisung tatsächlich die globale Variable `Q1` modifiziert:

```
1  > testfun(Q1,Q2);
2  Q1   bar
3                        [Q1, bar]
```

Das Beispiel ist besonders gefährlich aus zwei Gründen. Erstens sind solche Fehler schwer zu finden. In dieser Arbeit wurde das Problem entdeckt, weil ich versuchte nachzuvollziehen, wie ein ein halbes Jahr zuvor geschriebenes Programm funktioniert und es stets zerbrach, wenn man die Zusammensetzung weglässt. Existierender Code ist daher voll von solchen Fehlern. Zweitens ist der Fehler genau aus diesem Grund in Maple schwer zu beheben. Tut man es, so ist gewiss, dass viele Programme ihren Dienst verweigern. *xloops* ist ein solches Beispiel: es sind Stellen bekannt, in denen von diesem Problem „Gebrauch gemacht" wird und auch solche, wo es schon zu Verwirrung und Fehlern führte.

Wohlgemeinte didaktische Sperren sollen vermutlich die Verwendung ungeeigneter Datenstrukturen in manchen Systemen vermeiden. Als Liste bezeichnet man in der Informatik gemeinhin eine Ansammlung von Objekten die miteinander linear (möglicherweise bidirektional) verkettet sind. Die Zugriffszeit auf das n-te Element in einer Liste wächst also linear mit n an, weshalb nur dann von Listen Gebrauch gemacht werden sollte, wenn schnelles Hinzufügen/Entfernen von Elementen an zufälliger Position verlangt wird. Kurioserweise ist Maple nicht davon abzubringen, den Benutzer ab Listen der Länge 100 zu Vorsicht bezüglich der Zugriffszeit zu erziehen:

```
1  > L := [seq(i,i=0..100)]:
2  > L[77] := 0;
3  Error, assigning to a long list, please use arrays
```

Meines Erachtens nach sollte ein System außer der unvermeidlichen Geschwindigkeitsstrafe dem Programmierer bzw. Anwender keine willkürlichen Überraschungen bieten, die unter Umständen ein Umstrukturieren ganzer Programme erzwingen (was bisweilen sogar unmöglich ist, wenn eine interne Routine Listen benutzt).[4]

Sprachumstellungen von Version zu Version tragen ein übriges dazu bei, die Langlebigkeit eines Programmes in Grenzen zu halten. Einige ausgewählte Beispiele aus den letzten Versionen von Maple:

- Von MapleVR4 nach MapleVR5 wurde das Zeichen um den zuletzt evaluierten Ausdruck abzurufen von `"` nach `%` umgestellt. Vermutliche Intention: Das doppelte Anführungszeichen sollte zur Begrenzung von Strings zur Verfügung stehen; andere Systeme (z.B. Mathematica) benutzen schon immer `%`.
- Von MapleV nach Maple6 wurde der Operator zum Zusammenfügen von Namen von `.` nach `||` umbenannt (ohne jedoch die von ihm verursachten Verwirrungen im Scope zu beseitigen).
- Vor Maple6 wurden `do`-Schleifen mit `od` abgeschlossen. Seit Maple6 heißt es `end do`. `od` steht noch zur Verfügung, soll aber demnächst abgeschafft werden. Analoges gilt für `if` und `fi`. Vermutliche Intention: Verkleinerung der Schlüsselwortmenge.

Selbst wenn es möglich sein sollte, die Portierung auf die jeweils aktuelle Version sofort vorzunehmen (entwder per Hand oder mithilfe der mitgelieferten Übersetzungswerkzeuge), so stieß man bisher immer noch auf Überraschungen, die in einem professionellen Umfeld mit einem angemessenen Maß an Regressionstests nicht auftreten dürften. Die jüngste Version Maple7 fällt beispielsweise dadurch auf, dass sie 2000!/1999! zu 1 vereinfacht, weil Zähler und Nenner irrtümlicherweise als syntaktisch identisch identifiziert werden – keine frühere Version hatte damit Probleme. Obwohl Maple zweifelsfrei eines der derzeit leistungsstärksten symbolischen Systeme ist, wird in Abwesenheit von „Bugfix"-Releases die Herstellerabhängigkeit innerhalb eines größeren Projektes hierbei leider schnell unverantwortlich.

Systemanforderungen

Es folgt eine Auflistung der Eigenschaften, die ein System aufweisen muss, um für die Softwareimplementierung der Mainzer Methoden (und – mit wenigen Abstrichen – auch anderer Ansätze) geeignet zu sein.

- **Unexpandierte Darstellung multivariater Polynome:**
  Die anvisierten Berechnungen erfordern häufig das effiziente Hantieren mit in natürlicher Weise vorfaktorisierten Polynomen $(p^2 - m_1^2)((p-k)^2 - m_2^2)\cdots$ wie sie aus inversen Propagatoren entstehen. Systeme, die alle äquivalenten multivariaten Polynome zunächst in eine expandierte kanonische Form überführen indem sie sie ausmultiplizieren und sortieren (wie FERMAT [Lewi 1997], unter manchen Bedingungen [Koe 1999] auch REDUCE, AXIOM [JeSu 1992] und sogar FORM), verschleiern die Propagatorstruktur. In [Koe 1999] wird ein weiterer Vorteil der unexpandierten Darstellung am Beispiel

[4] Erfahrene Maple-Anwender empfehlen in obigem Beispiel immer `L:=subsop(77=0,L);` zu schreiben, da dies auch bei langen Listen funktioniert. Man beachte jedoch, dass damit die gesamte Liste kopiert wird.

der Tschebyscheff-Polynome geschildert: Mit einem rekursiven „Divide and Conquer“-Verfahren lassen sich diese Polynome sehr schnell in unexpandierter Form berechnen, so ist z.B. $T_8(x) = 2(2(2x^2-1)^2-1)^2-1$. Für viele Anwendungen ist diese Form der Darstellung schon völlig ausreichend, so z.B. für die numerische Auswertung entweder in Gleitkomma- oder exakter rationaler Arithmetik – vorausgesetzt es lässt sich zeigen, dass die Darstellung gut konditioniert ist.

- **Keine versteckten Grenzen:**
  Alle Klassen darstellbarer Objekte sollten nur durch verfügbaren Speicher und Rechenzeit beschränkt sein. Eine unaufhebbare Beschränkung auf $2^{16}-1$ Summanden in MapleV ist nicht vertretbar. Manche Systeme bürden dem Benutzer das Einstellen von Puffergrößen auf, was zwar die Bedienung nicht gerade erleichtert aber noch akzeptabel ist, wenn alle Pufferfunktionen vollkommen dokumentiert sind. (Leider ist auch dies nicht selbstverständlich: in Form2 stößt man rasch auf die Beschränkung auf ganze Zahlen mit einer Integerlänge kleiner als 400 Byte.) Versteckte Grenzen müssen nicht immer die Speicherverwaltung betreffen; sie können sich auch im Laufzeitverhalten äußern. Maple zum Beispiel wird im Laufe der Benutzung immer langsamer. Dies liegt daran, dass Nachschlagetabellen für Rechenergebnisse ausufern und die Zugriffszeiten darin nicht von $\mathcal{O}(\log n)$ sind sondern eher linear.
- **Offenheit und Erweiterbarkeit:**
  Die in den letzten Jahren in der Arbeitsgruppe ThEP durchgeführten Schleifenrechnungen sind über ein minimales Maß an Interoperabilität und punktuellen – häufig lediglich numerischen – Vergleichen nicht hinausgekommen. Sie bleiben damit deutlich hinter den gesteckten und im *xloops*-Projekt stets beschriebenen Zielen zurück. Da man nicht erwarten kann, das Endstadium der Entwicklung einer symbolischen Maschine zu erreichen, bedeutet dies übertragen, dass auf ein offenes und von folgenden Studentengenerationen erweiterbares Design geachtet werden muss. Ein objektorientierter Ansatz und weitgehend orthogonales Design können hier hilfreich sein.
- **Persistenz:**
  Algebraische Ausdrücke als Ergebnisse einer langen Rechnung sollten auf Festplatte geschrieben und zu einem späteren Zeitpunkt wieder eingelesen werden können. Üblicherweise geschieht dies bei symbolischen Systemen durch Ausgabe der Ausdrücke in der Form wie sie am Bildschirm dargestellt werden und Einlesen in genau dieser Form. Da dies in C++ so nicht möglich ist, wurde in [Baue 2000] für GiNaC ein Objektpersistenzmodell entwickelt.

Des weiteren sollte das System unbedingt portabel sein, um nicht in eine erneute Abhängigkeit, diesmal von einem Compilerhersteller, zurückzufallen.

Eine kursorische Geschichte der Sprache C++

C++ ist, wie die meisten Programmiersprachen, das Ergebnis eines langwierigen und teilweise wechselhaften Entwicklungsprozesses (bei den Ausnahmen handelt es sich um Experimentier- und Lernsprachen). Wie der Name impliziert handelt es sich von wenigen – pathologischen – Ausnahmen abgesehen um eine Übermenge der Sprache C (in der der Operator `++` die

davorstehende Variable inkrementiert). In diesem Abschnitt soll etwas Licht auf die Kette von Vorläufern geworfen werden. Es werden sich dabei einige Parallelen zur Motivation für GiNaC aufzeigen.[5]

Von C erbt C++ die traditionelle prozedurale Denkweise. Nun geht diese jedoch keinesfalls auf C zurück, sondern die Ursprünge sind selbst jenseits Cs Muttersprache B und Großmuttersprache BCPL [Rich 1967] zu suchen, nämlich in FORTRAN und Algol60. Der Entwicklungsschritt von BCPL nach B fand 1969 in den Bell Telephone Laboratories statt, als K. Thompson auf der Suche nach einer Sprache für das neuentwickelte Betriebssystem Unix war. Nach einem gescheiterten Versuch mit FORTRAN hatte er sich entschlossen, eine eigene Sprache zu entwickeln. Die Maschine, auf der er seine Experimente durchführte, war eine DEC PDP-7 mit 8192 Wörtern der Länge 18 Bit. Solche Hardware erklärt, warum weder BCPL noch B typisierte Sprachen waren. Zwar gab es schon einige der noch heute üblichen Spezifikatoren wie `auto` und `static`, aber die Notwendigkeit einer Typisierung war einfach noch nicht vorhanden – konnte man in einem 18-Bit Wort doch noch bequem eine brauchbare Gleitkommazahl unterbringen. Schrieb man beispielsweise in BCPL

```
1  let a = 15;     // Variable initialisieren in BCPL
2  let v = vec 4; // Quadrupel in BCPL
```

oder in B

```
1  auto a = 15;    /* Variable initialisieren in B */
2  auto v[4];      /* Quadrupel in B */
```

so wurde jeweils ein Array `v` aus 4 zusammenhängenden Wörtern für spätere Benutzung auf dem ⇒ Stack reserviert. Die Array-Semantik von C, in der `x[`n`]` äquivalent ist zu einer Dereferenzierung `*(x+`n`)`, geht direkt auf BCPL zurück. Die Syntax ist ein Erbe von B.[6]

Diese Situation änderte sich mit dem Erscheinen neuer Hardware. Die PDP-11, die 1970 bei Bell installiert wurde, war eine Byte-orientierte Maschine. D. Ritchie erweiterte B 1971 daher zunächst um die Typen `int` und `char` und später um `float`. Um Variablen zu deklarieren *musste* man ab nun den Typ spezifizieren:

```
1  int a = 15;      /* Variable initialisieren in C */
2  int v[4];        /* Integer-Quadrupel in C */
```

Weitere Neuerungen betrafen Records, in C `struct` genannt, sowie eine flexiblere Zeigersemantik wie z.B. Deklarationen vom Typ `int (*f)();` für Pointer auf Funktionen, die `int` zurückliefern sollen. Ein besonderes Verdienst von C ist es, Klarheit bei den booleschen Operatoren geschaffen zu haben: So war das bitweise Und (`and` bzw. `&`) und das bitweise Oder (`or` bzw. `|`) in BCPL und B überladen mit dem logischen Und und dem logischen Oder, falls es innerhalb von `if`-Abfragen auftauchte. In diesem Falle handelte es aber nicht um Überladung eines Operators in verschiedenen Kontexten und daher auch nicht um guten Stil. Die Bedeutungen sind unterschiedlich und können im selben Kontext vorkommen, wie ein in C gebräuchliches Maskenidiom deutlich macht:

[5] Die historischen Daten in diesem Abschnitt stammen aus [Ritc 1993], [Raym 1998] und [Stro 1994].

[6] Leider ist die Syntax der Dereferenzierung etwas verunglückt: in verschachtelten Konstruktionen wäre ein Suffix-Operator natürlicher zu lesen als ein Präfix-Operator. Außerdem führt die Wahl des Multiplikationszeichens `*` bei Anfängern immer wieder zu unnötigen Verwirrungen.

```
1 if (a & 0x0f) {
2     /* Programmblock 1 */
3 } else {
4     /* Programmblock 2 */
5 }
```

Im Falle a=0xf0 würde die Interpretation als logisches Und den ersten Programmblock ausführen, die Interpretation als bitweises Und den zweiten. Zur Unterscheidung wurden daher für die logischen Operatoren die Symbole && und || eingeführt, und festgesetzt, dass die Symbole & und | stets für bitweise Operatoren stehen.

In C war von Anfang an die Definition benutzereigener Datenstrukturen (Records) vorgesehen. Das Schlüsselwort struct kann ohne weiteres als Keim eines Objektmodelles angesehen werden. Aufzählungstypen hingegen (enum) kamen erst spät hinzu und führen wegen ihrer Beschränkung auf maschinendarstellbare ganze Zahlen bis heute eher ein Schattendasein. Aufgrund seines Ursprungs als Sprache für die Implementierung des Betriebssystems Unix erlaubt das Speichermodell von C die flexible Platzierung von Daten an drei verschiedenen Orten: automatisch auf dem Stack, dynamisch auf dem Heap und drittens statisch, also an einer festen Adresse.

C wurde 1990 von der ISO standardisiert [ISO 1990], wonach die verschiedenen Dialekte langsam zu konvergieren begannen. Eine stark erweiterte zweite Auflage des Standards folgte neun Jahre später [ISO 1999].

C++ war ursprünglich eine Spracherweiterung von C, die sehr früh (frühe 80er Jahre) die Sprache um Klassen und Objekte erweiterte wie sie viele Programmierer an Simula zu schätzen gelernt hatten.[7] Eine Klasse ist ein Zusammenschluss von Daten, der – anders als die in C schon vorhandene struct – auch Funktionsanweisungen („Methoden") beinhalten kann. Das mit dieser Spracherweiterung einhergehende häufigere Benennen von Typen machte eine stärkere Typisierung als die in C übliche notwendig. So ist es in C++ beispielsweise nicht mehr erlaubt durch void f() eine Funktion mit einer unspezifizierten Argumentenliste zu deklarieren.

Wie in jedem Objektmodell können Klassen um Funktionalität erweitert werden, indem man eine neue Klasse von ihr ableitet. Die abgeleitete Klasse „erbt" die Datenfelder und die Funktionalität der Elternklasse und kann insbesondere Methoden auch überschreiben, sofern die Elternklasse diese schon als virtuell deklariert hatte. Da ein Zeiger auf eine Basisklasse auch für eine abgeleitete Klasse stehen kann, wird der ⇒ dispatch in C++ so bewerkstelligt, dass jedem Objekt einer Klasse mit virtuellen Funktionen ein Zeiger auf eine Tabelle mit Zeigern auf die gültigen Funktionen mitgegeben wird.

Neu in C++ gegenüber C ist auch die Überladung von Funktionen nach ihren Argumenten. Eine Funktion void f(int) wird vom Compiler unterschieden von einer Funktion void f(double).[8] Dies dient hauptsächlich der Lesbarkeit von Programmen, da es immer gebräuchlicher wurde, die Argumentenliste im Funktionsnamen zu enkodieren (wie die transzendenten C-Funktionen in Anhang A oder im C-Quelltext der PARI-Bibliothek). Die entstandenen Mehrdeutigkeiten auf Linker-Ebene wurden dadurch gelöst, dass die Argumentenlisten nun

[7] Der erste Name war konsequenterweise „C with Classes".

[8] Eine Überladung nach den Rückgabewerten wird in C++ nicht unterstützt da es dem Programmierer frei steht den Rückgabewert zu ignorieren oder in einen anderen Typ zu „casten" – dies steht einer Auflösung der Mehrdeutigkeit durch den Compiler im Wege und ist eine von C geerbte Altlast.

vom Compiler in den Funktionsnamen enkodiert werden (⇒ name mangling), und zwar meist transparent für den Programmierer. Ursprünglich wurden solchermaßen überladene Funktionen durch das Schlüsselwort `overload` gekennzeichnet, um unbemerkt eingeführte Mehrdeutigkeiten zu vermeiden. Diese anfängliche Unsicherheit im Umgang mit Überladung wich schnell einer breiten Akzeptanz und die nächste logische Folgerung war, die von Algol68 bekannten überladenen Operatoren einzuführen. Diese erlauben es, arithmetische Operationen eigener Klassen intuitiv in Infix-Notation `i+j` zu schreiben, anstatt `plus(i,j)`. Die Klasse `complex` war lange das Paradebeispiel. Referenzen – auch ein Erbe von Algol68 – waren eine notwendige Folge von überladenen Operatoren: große Objekte übergibt man idealerweise als Zeiger anstatt als Kopie, in unserem Beispiel würde man also `plus(&i,&j)` schreiben, was aber durch keinen überladenen Operator als `&i+&j` ausgedrückt werden kann, da dies in C die Bedeutung der Addition auf Adressen hat. Das Problem wurde gelöst, indem man nun ausschließlich bei der Deklaration einer Funktion spezifiziert, ob eine Variable als Wert oder als Referenz übergeben wird – anstatt bei der Deklaration und beim Aufruf.[9]

Untypisierte Sprachen erlauben typunabhängige Programmierung. Dies kann durchaus sinnvoll sein, wenn zum Beispiel ein Algorithmus zum Sortieren unabhängig ist von dem, was sortiert werden soll, solange darauf nur eine Ordnungsrelation definiert ist. In C hat es sich zum Beispiel eingebürgert, solche Aufgaben entweder mit Makros zu erledigen oder nur noch mit Zeigern auf die zu verwalteten Objekte zu arbeiten. Im Falle des Sortierens wurde letzteres sogar in Form der Bibliotheksfunktion `qsort()` in [ATT 1989] standardisiert. Die starke Typisierung zwang zu einer Alternative und zur Einführung des womöglich leistungsstärksten Bestandteils von C++, den Templates. Mit ihnen kann eine völlige Trennung von typunabhängigen Algorithmen und darauf operierenden Datentypen erreicht werden. Mit der Standard Template Library (STL) wurde eine Referenzimplementierung mit den am häufigsten gebrauchten Containerklassen in den Sprachstandard aufgenommen. Sie nimmt dem Programmierer die mühsame Arbeit der Implementierung von Vektoren, Listen und assoziativen Containern seiner eigenen Datentypen ab und lässt ihn sich auf das Wesentliche konzentrieren.

Nichtlokale Fehlerbehandlungsmechanismen (sogenannte „*exceptions*") erlauben uns, das Versagen einer Routine außerhalb der unmittelbar aufrufenden Funktion korrekt zu handhaben und eventuell einen anderen Programmpfad einzuschlagen. Sie waren schon lange bekannt und geschätzt (beispielsweise in Algol68) und wurden Anfang der 1990er Jahre in Form der derzeit bekannten `try {...} catch(...) {...}` -Blöcke in C++ integriert.

Ein objektorientierter Ansatz ist der Offenheit und Erweiterbarkeit des Programmes sehr zuträglich. Er erlaubt es, Programme weitgehend entlang orthogonaler Richtlinien zu organisieren. Die Datenstrukturen können darin in gewissem Grade auf die zugrundeliegenden mathematischen Strukturen abgebildet werden. Vererbung hilft, das Anwachsen der Programmkomplexität zu zügeln: Hat eine Ansammlung von Software beispielsweise n Datenstrukturen, die mittels m Schnittstellen miteinander interoperieren müssen, so wächst die Komplexität traditionell etwa mit dem Produkt $n \cdot m$. Bringt eine Basisklasse aber schon etwas Funktionalität mit, so dass abgeleitete Klassen diese wiederverwenden können, so wächst die Komplexität in geringerem Maße, im günstigsten Falle vielleicht wie $n \log(m)$.

[9] Referenzen sind gewissermaßen zu spät eingeführt worden. Sonst wäre `this` in jedem Objekt sicherlich kein Zeiger, sondern eine Referenz.

Es gibt natürlich auch häufig ins Feld geführte Probleme der Sprache. In dieser Arbeit relevant wurden zum Beispiel die Nichtgenormtheit der Schnitte in der komplexen Ebene und die Nichtexistenz von ganzzahligen Datentypen mit genau bekannter Größe. Für beide Probleme ist mittelfristig jedoch Abhilfe in Sicht: die Revision des C-Standards [ISO 1999] definiert die Schnitte in kompatibler Art und Weise (siehe Anhang A) und führt in der Header-Datei `<stdint.h>` ganzzahlige Datentypen mit 8, 16, 32 und 64 Bit ein. Ein häufiges Problem sind Binärinkompatibilitäten zwischen verschiedenen Versionen einer benutzten Bibliothek – nicht nur als Folge der stärkeren Typisierung: Die Größe einer Klasse kann sich unvermittelt ändern, wenn eine Basisklasse ihre Größe ändert, oder die Größe der `vtable` kann sich ändern durch Einfügung neuer virtueller Methoden, mit unvorhersehbaren Konsequenzen zur Laufzeit. Die üblichen Lösungswege sind das Auffüllen von abgeleiteten Klassen mit Füllbytes, die später entfernt werden können („Padding“), mehrere Versionen der dynamischen Bibliothek im System zu verteilen (berüchtigt als „DLL-Hölle“) oder sie statisch gleich in die Applikation zu linken. In einem frei verfügbaren System können diese architekturbedingten Probleme allerdings vernachlässigt werden, da einer Neuübersetzung der gesamten Bibliothek nichts im Wege steht. Etwas enttäuschender ist meiner Auffassung nach die Tatsache, dass C++ dem Programmierer im Vergleich zu C keine neuen Hilfsmittel für Funktionen mit variabler Argumentenzahl an die Hand gibt. In symbolischen Algorithmen wäre dies bisweilen begehrenswert – die in Abschnitt 4.5 beschriebenen Pseudofunktionen werden mit dem Problem auf ihre Art umgehen müssen. Im etwas größeren Bild ist die gesamte Abhängigkeit vom C-Linker für C++ ein unerschöpflicher Quell kleiner Probleme: Das Schlüsselwort `export` ist aus diesem Grunde bis heute unimplementiert geblieben und die auf Seite 119 beschriebene Abhängigkeit von Linker-Charakteristiken wäre trivial zu beheben, wird aber nicht spezifiziert, da man den Linker nicht als Teil des Sprachumfanges verstehen möchte.

Das Hauptargument für die Benutzung von C++ in Projekten, die länger leben sollen als der typische Revisionsabstand einer propietären Softwarebasis ist der internationale Standard, in diesem Falle [ISO 1998]. Er schützt das Projekt vor Willkürlichkeiten, die die Benutzbarkeit von Code zerbrechen, dessen Autor nicht mehr zum Portieren auf die neue Umgebung zur Verfügung steht.[10]

Schlussfolgerung

Eine Einbettung des symbolischen/algebraischen Teils einer Berechnung in C++ bietet sich an und verspricht folgende Vorteile:

[10] Hiervon gibt es menschlich bedingte Ausnahmen. Die in dieser Arbeit aufgetauchte Debatte um einen GCC-Bugreport, nachzulesen im GNATS Audit Trail `gcc/1565` unter `http://gcc.gnu.org/` zeigt, dass bisweilen lediglich die Autoren des Standards in der Lage sind, dessen Wortlaut zu verstehen. Es ging hierbei um die Frage, ob

```
1 #define NIL(xxx)  xxx
2 #define G_0(arg) NIL(G_1) (arg)
3 #define G_1(arg) NIL(arg)
4 G_0(42)
```

vom C/C++-Präprozessor zu `42` oder zu `NIL(42)` expandiert werden soll. Diese Frage konnte nur aus der Erinnerung eines der Standard-Autoren über die beabsichtigte Semantik geklärt werden.

- **Effizienz:**
  Durch Kompilation zu Maschinencode wird zeitkritischer Overhead vermieden. Dies mag bei symbolischen Rechnungen wenig Vorteile bringen, macht sich aber insbesondere bei der Integration mit nicht-symbolischen Rechnungen bemerkbar, wenn beispielsweise kleine Schleifen mit effizienten Integerargumenten durchlaufen werden können statt mit arithmetischen Typen beliebiger Genauigkeit.
- **Strukturierte Sprachelemente:**
  Schleifenkonstrukte etc. sind in der Sprache enthalten und müssen nicht erst von Hand implementiert werden.[11]
- **Portabilität:**
  Effiziente Compiler sind weit verbreitet, ein gewisses Maß an Herstellerunabhängigkeit ist daher erzielbar.
- **Typsicherheit:**
  Schon zur Kompilierzeit wird abgesichert vor Operationen mit inkompatiblen Operanden.
- **Algebraische Syntax:**
  Die elementaren Operationen `+`, `-`, `*`, `=`, `==`, etc. können überladen und in Infix-Notation intuitiv verwendet werden. Dies ermöglicht lesbarere Programme als Sprachen ohne Operatorüberladung. Wir können beispielsweise `x+y` schreiben, wo wir in Lisp `(+ x y)`, in C `add(x,y,&z)` und in Java `x.add(y)` schreiben müssten.
- **Integrationsfähigkeit:**
  Einer Schätzung zufolge [PrWe 1999] wird REDUCE in 50% der Rechenzeit lediglich verwendet um FORTRAN- oder C-Code für die numerische Weiterverarbeitung zu erzeugen. Dieser Prozess kann zwar nur zum Teil eliminiert werden, aber mit den Problemen der Zweisprachigkeit wird der Benutzer gar nicht erst konfrontiert. (Siehe auch Kasten auf Seite 69.)
- **Langlebigkeit:**
  C++ ist keine Modesprache, sondern die *lingua franca* wissenschaftlichen Rechnens in der Hochenergiephysik. Dies gilt in der Experimentalphysik noch weitaus mehr als in der Theorie.

3.2. Das Design von GiNaC

GiNaC soll als Bibliothek implementiert sein und mit symbolischen Ausdrücken (Symbolen, Summen, Produkten, ...) direkt in dieser Sprache umgehen können. Ähnlich wie die Anweisung `l=x+y;` für den Typ `int` eine Zuweisung des Ergebnisses der Summe von `x` und `y` an eine Variable `l` darstellt, sollen Anweisungen mit symbolischen Objekten geschrieben werden

[11] Die Implementierung in Computeralgebrasystemen kann durchaus unbefriedigend sein. MapleV beispielsweise kennt zwar `while...do` aber nicht `do...while`. Dies kann bei einigen Algorithmen zu sehr unnatürlichen und verdrehten Implementierungen führen. Ein gutes Beispiel ist der Vergleich der in [GCL 1992] abgedruckten Fassung des Yun'schen Algorithmus zur quadratfreien Faktorisierung mit der in GiNaC implementierten Fassung. Letztere kommt dank `do...while` ohne eine Duplizierung des Schleifenblockes aus und ist leichter verständlich.

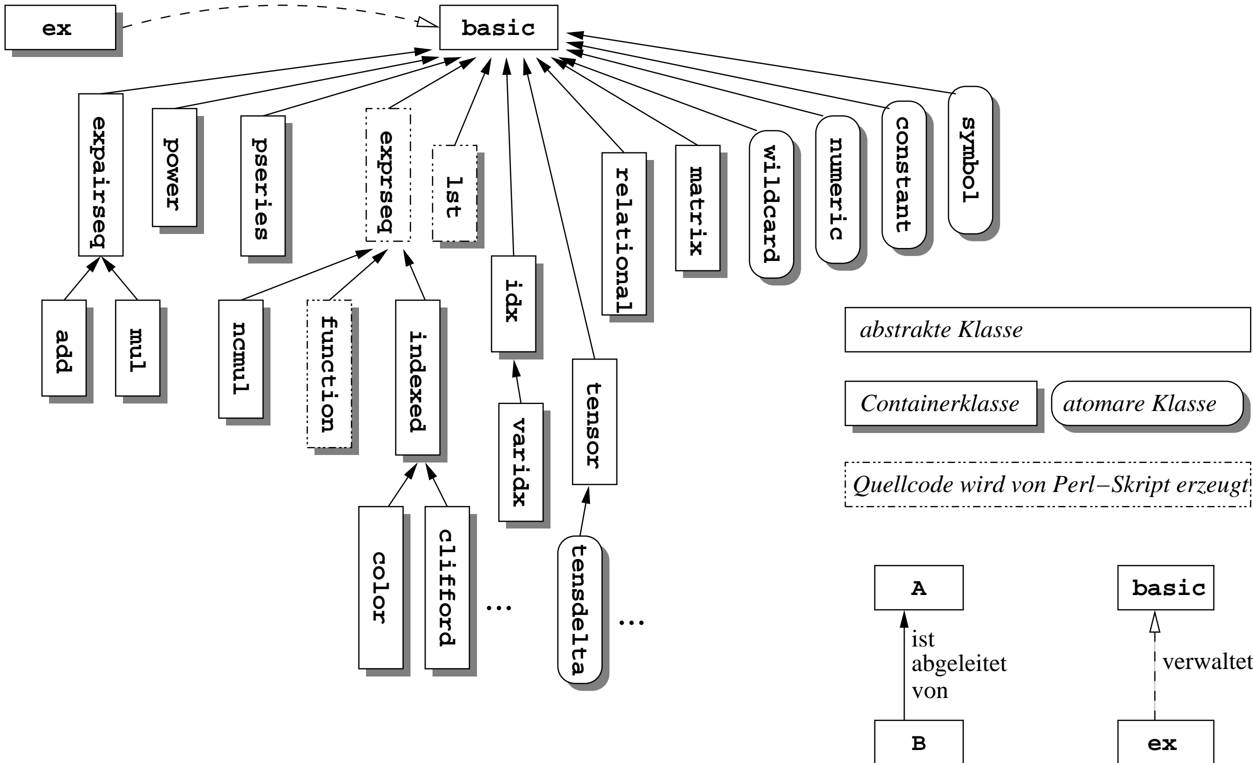


Abbildung 3.1.: *Übersicht über die Klassenhierarchie von GiNaC 1.0 und einige Beziehungen zwischen einzelnen Klassen.*

können. Sind `x` und `y` beispielsweise aus dem Körper der gebrochenrationalen Zahlen, so ist das Ergebnis klarerweise wieder in diesem Körper. Sind sie dagegen Symbole ohne Bindung, so ist das Ergebnis eine unevaluierbare Summe. Hatte `y` zuvor die Variablenbindung −`x`, so ist das Ergebnis aber einfach die Null. Aus diesem einfachen Beispiel folgt, dass es wenig sinnvoll ist, die Operatoren für Addition und Subtraktion so zu überladen, dass sie immer ein Summenobjekt zurückliefern.

Diese automatische Umformung wird von einem eingebauten anonymen ⇒ Evaluator vorgenommen und sein Rückgabewert ist die Klasse aller darstellbaren Ausdrücke, genannt `ex`, kurz für „*expression*“. Klasse `ex` ist ein sogenannter „Wrapper“, eine Hüllklasse (in anderen Zusammenhängen auch „Proxy“ genannt, so z.B. in der Terminologie von [GHJV 1995]). Sie ist extrem leichtgewichtig, da sie das eigentliche algebraische Objekt indirekt mittels eines Zeigers darauf auf dem Heap als Referenz verwaltet. Um etwas präziser zu sein: mit „extrem leichtgewichtig“ ist gemeint, dass `ex` außer einem Zeiger nichts enthält (auch keinen `vptr`), also `sizeof(ex)`=`sizeof(void*)`. Die Verwaltung als Zeiger dient zweierlei Zwecken: Erstens haben damit alle Ausdrücke die gleiche Größe, was eine Voraussetzung für das Erzeugen von STL-Containern ist. Zweitens kann nur per Zeiger oder Referenz die zu einem Objekt passende virtuelle Methode aufgerufen werden, wenn `ex` einen Methodenaufruf weiterdelegiert.

Eine Referenzzählung trägt dafür Sorge, dass Objekte genau so lange leben wie sie in Gebrauch sind. Einerseits ruft die Sprachimplementierung beim Verlassen eines Blockes automatisch die Destruktoren der im Block deklarierten Objekte auf. Andererseits können nicht alle Objekte gelöscht werden, wenn wiederum andere Ausdrücke auf sie Bezug nehmen. Dass dies tatsäch-

lich der Fall ist, dafür sorgen der Copy-Konstruktor und der Zuweisungsoperator der Klasse ex. Ist ein darstellbares Objekt auf dem Stack, so wird zunächst einmal eine Kopie davon dynamisch auf dem Heap erzeugt und in allen weiteren Wrapper-Objekten nur noch ein Zeiger darauf verwaltet:

```
1 symbol x("x");
2 ex f = 2*x;      // legt Kopie von x auf dem Heap an
3 ex g = f;        // kopiert Zeiger, erhöht Referenzzähler
```

Die aus der Sprache Perl [WCS 1996] bekannten Skalare verhalten sich unter Referenzierung so ähnlich (auf den subtilen Unterschied wird auf Seite 71 eingegangen werden). Python ist eine weitere populäre Programmiersprache, die völlig auf Referenzzählung beruht.

Mechanismen der Referenzzählung

Der Klasse ex obliegt die alleinige Zuständigkeit für das korrekte Verwalten der von basic abgeleiteten algebraischen Objekte auf dem Heap. Die zuständigen Methoden und ihre Funktionsweise werden im Folgenden skizziert.

Erzeugen: Erzeuge Basisobjekt auf dem Heap, verwalte einen Zeiger darauf. Nicht nur Summen und Produkte sondern auch einzelne Symbole und Zahlen sind Objekte – die Objektauflösung in GiNaC ist also extrem feinkörnig. Wenn solche Objekte auch noch sehr häufig vorkommen, kann dies zu einer gewissen Speicherverschwendung führen. Diesem Problem versucht man häufig mit dem sogenannten „Flyweight"-Muster beizukommen [GHJV 1995]. So könnte der ex-Konstruktor für kleine ganze Zahlen eine Flyweight-Fabrik aufrufen, die zum Beispiel Zeiger auf schon erzeugte ganze Zahlen in einem assoziativen Array speichert und nur noch den Referenzzähler erhöht, falls die Zahl schon vorhanden ist.

Kopieren: Der Copy-Konstruktor `ex::ex(const ex &)` kopiert den Zeiger auf das Basisobjekt und inkrementiert dessen Referenzzähler. Der Copy-Konstruktor ist also nichttrivial und die nichtlokale Manipulation des von ex verwalteten Objektes macht seine Ausführungskomplexität zwar unabhängig von der Größe des verwalteten Objektes, aber dennoch etwas langsam. Aus diesem Grund empfiehlt es sich, Objekte vom Typ ex in Parametern stets als Referenz zu übergeben – obwohl sie sehr klein sind (`sizeof(ex)`=`sizeof(void*)`) und als Faustregel kleine Objekte in C++ sonst typischerweise als Wert übergeben werden sollten.

Zerstören: Der Destruktor `ex::~ex()` dekrementiert zunächst den Referenzzähler des verwalteten basic-Objektes, und falls dieser auf Null gesunken ist, ruft er dessen delete-Operator auf (welcher normalerweise nicht explizit überladen ist, sondern nur implizit als Destruktoraufruf von `virtual basic::~basic()` definiert ist).

Zuweisen: Die Methode `ex::operator=(const ex &)` implementiert die Zuweisung als sukzessives Zerstören (falls der dekrementierte Referenzzähler des verwalteten und von basic abgeleiteten Objektes `this->bp` auf Null gesunken ist) und anschließendes Kopieren (durch Zuweisung des basic-Zeigers und Inkrementieren dessen Referenzzählers).

Vergleichen: Die Methode `.compare()` etabliert eine kanonische Äquivalenzrelation auf den GiNaC-Objekten: Zwei gleiche Objekte sollen unter a`.compare(`b`)` 0 zurückliefern (Reflexivität), ansonsten ± 1, wobei a`.compare(`b`)`$= -b$`.compare(`a`)` (Symmetrie) und a`.compare(`b`)`$=\pm 1 \wedge b$`.compare(`c`)`$=\pm 1 \Rightarrow a$`.compare(`c`)`$=\pm 1$ (Transitivität) gelten sollen. Hierbei wird in vier Schritten vorgegangen: Beim Vergleich zweier `ex` miteinander bietet es sich an, zunächst nur die Pointer auf die eingehüllte Klasse zu vergleichen: sind diese identisch, so kann sofort 0 zurückgegeben werden. Als Nächstes werden die jedem Objekt von einer ⇒ Hashfunktion zugeordneten Hashwerte $\in \{0 \ldots 2^{32}-1\}$ verglichen, wobei die Äquivalenzrelation vermöge des Vergleiches natürlicher Zahlen etabliert wird. Im Falle einer Hashkollision werden die RTTI-Schlüssel (run-time type information) der entsprechenden Klassen verglichen, und falls auch diese identisch sind, wird auf eine von der entsprechenden Klasse bereitgestellte Methode `.compare_same_type()` zurückgegriffen, die die Eigenschaften der Äquivalenzrelation explizit definieren muss.

Das Erzeugen von Ausdrücken kann ein Problem für die Effizienz der Bibliothek sein. Die meisten Funktionen in GiNaC müssen, da der Rückgabewert während des Kompilierens noch nicht bekannt ist, ein allgemeines Objekt der Klasse `ex` zurückgeben. Da die Rückgabe jedoch nicht auf dem Heap sondern auf dem Stack erfolgt, muss bei der Entgegenahme des Ergebnisses in der aufrufenden Funktion das Objekt erst in den Heap kopiert – also dynamisch alloziert – werden. Das Objekt wird also zweimal kopiert. Besser wäre es, wenn die aufgerufene Funktion das von `basic` abgeleitete Objekt selbst schon dynamisch auf dem Heap erzeugt. Damit der Konstruktor `ex::ex(const basic &)` aber nicht trotzdem noch einmal kopiert, muss er die Möglichkeit haben festzustellen, ob das Objekt schon dynamisch alloziert ist. Zu diesem Zwecke markiert die aufgerufene Funktion das Objekt als auf dem Heap liegend, indem es selbst das Flag `status_flags::dynallocated` setzt. Der gewöhnliche Benutzer braucht sich hierum nicht zu kümmern. Es reicht normalerweise völlig aus, das Objekt einfach per `return mein_objekt;` zurückzugeben. Performanzkritische Methoden innerhalb der Bibliothek sollten jedoch das Idiom `return (new mein_objekt)->setflag(status_flags::dynallocated);` benutzen.[12] Eine Rückgabe per `return (new mein_objekt);` darf niemals erfolgen. Sie erzeugt ein Speicherleck.

Eine Option: Fusion redundanter Ausdrücke

Im Prinzip ist es möglich, beim Vergleich zweier Hüllobjekte vom Typ `ex` miteinander eine Objektabreicherung durch defensive Fusion redundanter Ausdrücke vorzunehmen und somit Speicher zu sparen.[13] Falls die Zeiger verschieden sind, die Objekte sich aber dennoch als

[12] Ein verführerischer Gedanke kommt bei diesem Schema unweigerlich immer wieder auf. Das Bit namens `status_flags::dynallocated` wird doch immer dann gesetzt, wenn ein Objekt explizit mittels `new` auf dem Heap angelegt wird. Das Setzen dieses Bits könnte also automatisch von einem überladenen `operator new` erledigt werden. Die Implementierung dieser Idee muss jedoch scheitern. Der `operator new` kann in C++ lediglich den Speicher *vorbereiten*. Danach wird der Konstruktoraufruf ausgeführt und initialisiert den Speicher. Eventuell von `new` gesetzte Werte werden dabei zwangsläufig überschrieben. Für eine erschöpfende Diskussion warum ein automatisches Setzen dieses Bits in C++ auch mit anderen Tricks niemals sowohl korrekt als auch portabel funktionieren kann siehe [Meye 1996, *Kapitel 27*].

[13] Wir nennen die Fusion *defensiv* um sie zu unterscheiden von einer aggressiven Möglichkeit wie einer Durchmusterung aller allozierten Ausdrücke. Dies wäre jedoch mindestens ein $\mathcal{O}(\log N)$-Prozess und ist daher

gleich herausstellen, kann eines der Objekte vernichtet werden und in den Hüllobjekten ein Zeiger umgelenkt und ein Referenzzähler erhöht werden. Die Idee lässt sich an einem kleinen Programmausschnitt darstellen:

```
1  symbol x("x");
2  ex e1 = x+x;
3  ex e2 = 3*x-x;
4  if (e1.compare(e2) == 0) {
5      // ...
6  }
```

Wir markieren die darin konstruierten Objekte vom Typ `ex` als Kreise und die von `basic` abgeleiteten Objekte mit Rauten zusammen mit dem Stand ihrer Referenzzähler:

2	`ex e1 = x+x;`	`e1` wird erzeugt und soll `x+x` verwalten, was auf dem Heap sofort zu `2*x` kanonisiert wird.
3	`ex e2 = 3*x-x;`	`e2` wird erzeugt und soll `3*x-x` verwalten, was auf dem Heap zu `2*x` kanonisiert wird. Noch ist es aber ein anderes `2*x`-Objekt als in Zeile *2*.
4 *5*	`if (e1.compare(e2)==0) {` `    //...`	`e1` wird mit `e2` verglichen. Die Gleichheit wird bestätigt und die von `e1` und `e2` referenzierten Objekte werden „fusioniert“. Da einer der Referenzzähler auf Null sinkt, kann sein `2*x`-Objekt vernichtet werden.

In dem Fall, dass zwei Objekte mit Referenzzähler > 1 verglichen werden, kann nicht prinzipiell ausgeschlossen werden, dass die Zeiger ungeschickt umgelegt werden. Lediglich die Tendenz, immer das Objekt mit dem ohnehin schon höheren Referenzzähler zu bevorzugen, ist an dieser Stelle formulierbar. Zu diesem Zwecke wird in `ex::compare()` aus der Zeile

```
1      return bp->compare(*other.bp);
```

der Anweisungsblock

```
1      const int cmpval = bp->compare(*other.bp);
2      if (cmpval==0) {
3          if (bp->refcount<=other.bp->refcount) {
4              if (--bp->refcount==0)
5                  delete bp;
6              bp = other.bp;
7          } else {
8              if (--other.bp->refcount==0)
9                  delete other.bp;
10             other.bp = bp;
11         }
12         ++bp->refcount;
13     }
14     return cmpval;
```

völlig unpraktikabel. Mathematica unterstützt mit der Funktion `Share[]` diese aggressive Fusion, verfügt aber – nach einer privaten Mitteilung von Henry Cejtin – nicht über die defensive Variante. Auch sonst scheint die defensive Fusion bisher nicht in der Literatur beschrieben worden zu sein.

Optimierte Codegenerierung vs. Compileroptimierung

Die meisten Computeralgebrasysteme haben die Fähigkeit, Programmcode für numerische Evaluation in Maschinengenauigkeit zu erzeugen. Die freien Symbole müssen bei der Ausführung natürlich durch Werte aus einer Stützpunktmenge ersetzt werden. Viele Systeme können den generierten Code zudem schon optimieren. Da GiNaC keine ausgefeilten Methoden hierfür hat, stellt sich die Frage ob der erzeugte Code überhaupt konkurrenzfähig ist. Die Optimierung, die Maple beim Aufruf `C(f,optimized)` *durchführt, besteht im Wesentlichen aus der Elimination aller redundanten Subausdrücke in f, geht also insbesondere über die hier vorgestellte defensive Fusion zur Laufzeit hinaus. Betrachten wir das erste Beispiel aus der Maple-Dokumentation $f(x) := 1 - 2x + 3x^2 - 2x^3 + x^4$. Es wird unoptimiert zu* `1.0-2.0*x+3.0*x*x-2.0*x*x*x+pow(x,4.0)` *transformiert, wobei sich schon der Aufruf von* `pow(float, float)` *äußerst negativ auf die Performanz auswirkt. Im optimierten Modus werden zunächst zwei temporäre Variablen* `t1=x*x` *und* `t3=t1*t1` *erzeugt und dann $f(x)$ mittels* `1.0-2.0*x+3.0*t1-2.0*t1*x+t3` *berechnet. Die Elimination gemeinsamer Unterausdrücke ist unter dem Namen CSE* (engl: *common subexpression elimination) aber eine geläufige Compileroptimierung. Das Problem beim ursprünglichen Ausdruck für $f(x)$ ist, dass der Compiler bei der syntaktischen Suche nach Unterausdrücken keine Umklammerung vornehmen darf. Die Sprachstandards gebieten aus Gründen der numerischen Stabilität die strenge Einhaltung der Assoziativitätsreihenfolge, für C in [ISO 1999,* Abschnitt 5.1.2.3*] und für C++ in [ISO 1998,* Abschnitt 1.9*]. In [Baue 2000] wurde jedoch ein einfaches Schema entworfen, in dem durch explizite Klammerung dem Compiler geholfen werden kann: Der obige Ausdruck wird darin umgeschrieben zu* `1.0-2.0*x+3.0*(x*x)-2.0*x*(x*x)+(x*x)*(x*x)`*, so dass der Compiler den Ausdruck* `x*x` *wiederverwerten kann. In diesem Falle wird von jedem modernen Compiler automatisch derselbe Code generiert wie bei der von Maple optimierten Routine. Es sei angemerkt, dass die obige Art der Umklammerung mathematisch äquivalent ist zur auf Seite 112 beschriebenen schnellen Exponentiation. Die Optimierung wie sie beispielsweise von MapleV durchgeführt wird ist also eigentlich eine Art Pleonasmus. Eine abschließende Beurteilung bezüglich Effizienz und Stabilität dieser beiden Ansätze und auch externer Werkzeuge wie Ctadel [Enge 1998] steht noch aus. Die in [PrWe 1999] vorgenommene analysiert jedenfalls nicht das Endprodukt in Form von Maschinensprache und berücksichtigt nicht das Wechselspiel zwischen den verschiedenen Optimierungsschritten.*

und die Variable `ex::bp` muss als `mutable` deklariert werden, da die Methode als `const` deklariert worden ist und somit verspricht, `ex::bp` nicht zu verändern. Analog zu `ex::compare()` sollte dann eigentlich noch `ex::is_equal()` abgeändert werden. Letzteres scheint aber in der Praxis immer zu deutlich langsamerem Code zu führen.

Redundante Ausdrücke entstehen in symbolischen Umformungen tatsächlich recht häufig und das Fusionieren kann bei gewissen Umformungen ein enormes Einsparungspotenzial haben. Da `ex::compare()` aber eine Methode ist, die sich sehr empfindlich auf das Laufzeitverhalten auswirkt und der obige Code sich in der Praxis bisher nur als minimal beschleunigend und speichersparend (beides ca. 15% im Rahmen von [BKK 2001]) ausgewirkt hat, wird dieser Abreicherungsmechanismus derzeit nicht benutzt.[14] Außerdem bieten die genauen Implikationen

[14] Dass dieses Fusionieren Speicher spart ist sofort einsichtig. Eine Messung hat ergeben, dass bei typischen Anwendungen im `cmpval==0`-Fall in mehr als 50% aller Fälle Objekte fusioniert werden können. Dass es

dieser Technik für sich genommen wahrscheinlich ein reichhaltiges informatisches Untersuchungsfeld. Generell gibt es zwei mögliche Probleme dabei:

- Der Einfluss auf die Semantik von das Objekt modifizierenden Methoden muss genau untersucht werden. Womöglich sollten alle Objekte vom Typ `ex` implizit als `const` verstanden werden, um semantische Verwirrungen auszuschließen. Dies kann syntaktisch zum Beispiel dadurch erzwungen werden dass alle Rückgabewerte und alle Methoden als `const` deklariert werden.
- Allgemein ist jeglicher Programmcode, der Zeiger auf von `basic` abgeleitete Objekte (zum Beispiel `const symbol *dummy = &ex_to<symbol>(e1);`) anlegt, um sie später wieder zu verwenden, anfällig gegen Interferenzen mit dem Fusionieren. Das referenzierte Objekt könnte nämlich in der Zwischenzeit vom Heap gelöscht worden sein. Um potenzielle Probleme zu vermeiden dürfen solche Zeiger ohne ein für die Referenzzählung zuständiges Hüllobjekt der Klasse `ex` nicht benutzt werden. GiNaC erlaubt aber gerade das Definieren von diesen Objekten durch den Benutzer – es hat keine vollständig ausgebildete „*Bridge*“ zwischen `ex` und den von `ex` verwalteten Klassen. Streng genommen könnten wir diese Sicherheit in der Sprache C++ niemals garantieren – der Benutzer kann im Zweifelsfalle mit `reinterpret_cast` jeden Schutz umgehen.

Referenzzählung und zirkuläre Verweise

Implementierungen von Referenzzählungsmechanismen sind häufig der Kritik ausgesetzt, dass sie nicht völlig sicher gegen Speicherlecks sind. Ein Vergleich mit anderen Referenzzählern tut daher an dieser Stelle gut. Es handelt sich meist um zirkuläre Referenzen, die nicht mehr entflochten werden können. In Perl [WCS 1996] beispielsweise erzeugt die folgende Prozedur ein Speicherleck:

```
1 sub memleak
2 {
3     # Initialisiere zwei Skalare
4     my $e1 = "x";
5     my $e2 = "y";
6     # Referenziere sie gegenseitig
7     $e1 = \$e2;
8     $e2 = \$e1;
9 }
```

In den Zeilen *4* und *5* werden skalare Variablen (hier: Strings) angelegt, deren beider Referenzzähler auf eins steht. In Zeile *7* wird durch die Anweisung, `$e1` als Referenz auf `$e2` zu verstehen, der Referenzzähler von `$e2` erhöht (und dabei dessen vorheriger Wert `"x"` vernichtet). Ebenso wird in Zeile *8* der Referenzzähler von `$e1` auf zwei erhöht. Beim Verlassen des Funktionsblockes werden die Referenzzähler von `$e1` und `$e2` um eins erniedrigt und stehen somit wieder auf 1. Danach können sie aber nicht mehr angesprochen werden. Es ist eine

sich aber überhaupt beschleunigend auswirkt liegt wiederum an der Methode `ex::compare()` selbst: Beim Vergleich zweier gleicher Objekte greift bei geteilten Ausdrücken die Abkürzung über den Zeigervergleich, bei ungeteilten müssen alle vier Schritte durchlaufen werden.

zirkuläre Struktur entstanden, die unmöglich wieder aufgebrochen werden kann, da `$e1` und `$e2` nicht mehr im Scope, also nicht mehr verfügbar sind.

Vergleichen wir dies mit einer scheinbar analogen Situation in GiNaC:

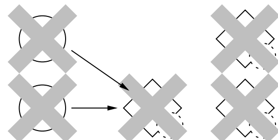
```
1  void no_memleak(void)
2  {
3      // Initialisiere zwei skalare Ausdrücke
4      symbol x("x"), y("y");
5      ex e1 = x;
6      ex e2 = y;
7      // Referenziere sie gegenseitig
8      e1 = e2;
9      e2 = e1;
10 }
```

Dies führt in GiNaC nicht zu einem Speicherleck. Letztendlich liegt das daran, dass die Referenzzählung anders als bei Perl nicht bei den Schlüsseln (dort: skalare Variablen, hier: Objekten der Klasse `ex`), sondern bei den von ihnen referenzierten Werten (dort: Strings, hier: Objekte der Klasse `symbol`) stattfindet. Um dies einzusehen ist es notwendig, die Referenzzählung manuell nachzuvollziehen. Wieder notieren kreisförmige Objekte die Klasse `ex` und rautenförmige Objekte eine von `basic` abgeleitete Klasse mit dem aktuellen Wert ihres Referenzzählers.

4	`symbol x("x"), y("y");`	Zwei Objekte der Klasse `symbol` werden auf dem Stack erzeugt.
5	`ex e1 = x;`	`e1` wird erzeugt und soll `x` verwalten, aber `x` ist noch nicht dynamisch alloziert. Also wird zunächst eine Kopie davon auf dem Heap angelegt.
6	`ex e2 = y;`	Ebenso wird `e2` erzeugt und eine Kopie von `y` auf dem Heap angelegt um von `e2` verwaltet zu werden.
8	`e1 = e2;`	Da `e1` nun nicht mehr `x` verwaltet, sinkt der Referenzzähler des dynamisch allozierten `x` auf 0 weshalb es von `ex::operator=` vernichtet wird.
9	`e2 = e1;`	`ex::operator=` erhöht den Referenzzähler des dynamisch allozierten `y` und erniedrigt ihn gleich wieder, da die von `e1` und `e2` verwalteten Objekte identisch sind. Er fällt nicht auf 0, daher wird nichts vernichtet. Die Anweisung ist also eine Null-Operation.

```
10 }
```

Die Variablen x, y, e1 und e2 fallen aus dem Scope, ihre Destruktoren werden aufgerufen. Die Destruktoren von e1 und e2 erniedrigen den Referenzzähler des dynamisch allozierten y jeweils um eins und der Letzte knipst das Licht aus indem er dieses y vernichtet.



Wir können anmerken, dass die Sicherheit vor unbeabsichtigten Speicherlecks auch bei Variationen der obigen Routine erhalten bleibt – immer vorausgesetzt natürlich, dass der Programmierer nicht explizit ein solches durch unsachgemäßen Umgang mit new konstruiert. Beispielsweise ist auch die folgende Routine sicher:

```
1  void no_memleak(void)
2  {
3      // Initialisiere zwei skalare Ausdrücke
4      ex e1(symbol("x"));
5      ex e2(symbol("y"));
6      // Referenziere sie gegenseitig
7      e1 = e2;
8      e2 = e1;
9  }
```

Strenger formuliert könnte ein Speicherleck durch zirkuläre Referenzen dann auftreten, wenn von basic abgeleitete Objekte gegenseitig aufeinander verweisen. Das ist aber nicht ohne weiteres möglich, da diese Klassen keine Referenzen beinhalten. Auch Containerklassen wie add oder mul enthalten nur Objekte vom Typ ex als Elemente. Die Trennung von algebraischen Objekten und der sie verwaltenden Klasse ex gewährleistet die Speichersicherheit.

Dieses Prinzip scheint auch in anderen auf Referenzzählung basierten Computeralgebrasystemen zur Anwendung zu kommen. Nach einer privaten Mitteilung von Henry Cejtin ist es auch in Mathematica für den Benutzer prinzipiell unmöglich, zirkuläre Ausdrücke zu erzeugen.

Darstellungsbäume

Da jeder symbolische Ausdruck selbst Unterausdruck in einem übergeordneten Ausdruck sein kann, muss ihre Darstellung eine Baumstruktur sein. Das unexpandierte multivariate Polynom $2d^3(4a + 6b - 3 - b)$ kann beispielsweise in der in Abbildung 3.2 skizzierten Datenstruktur repräsentiert werden. Dies ist aber aus Gründen der Effizienz nicht begehrenswert. Alle Computeralgebrasysteme führen daher Termumschreibungsregeln in einem sogenannten anonymen Evaluator durch. Er fasst Terme zusammen und führt sie in eine effizientere Darstellung über. In der Abbildung 3.2 wird er in der Klammer $6b$ und $-b$ zu $5b$ addieren, auch wenn sie nicht beieinander stehen. Das Ergebnis ist der Darstellungsbaum aus Abbildung 3.3. Die vorgenommene Vereinfachung entspricht trivialerweise auch einer Abreicherung von Objekten, kommt die Variable b doch in $5b$ einmal weniger vor als in $6b - b$. Sie betreffen aber jeweils nur einen Ast in der Verzweigung. In unserem Beispiel war das nur die Summe $4a + 6b - 3 - b$.

Die Darstellung in Abbildung 3.3 ist aber immer noch nicht besonders effizient. Die Bestandteile von Polynomen sind immer irgendwelche symbolische Terme und numerische Koeffizienten und während der anonymen Evaluation werden Koeffizienten, die zu syntaktisch identischen

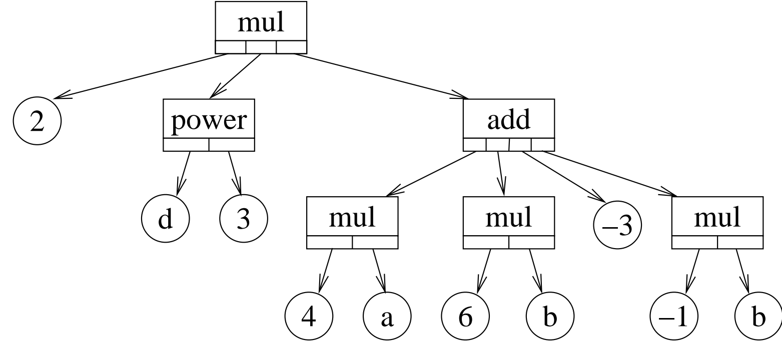


Abbildung 3.2.: *Mögliche unevaluierte Darstellung von* $2d^3(4a + 6b - 3 - b)$.

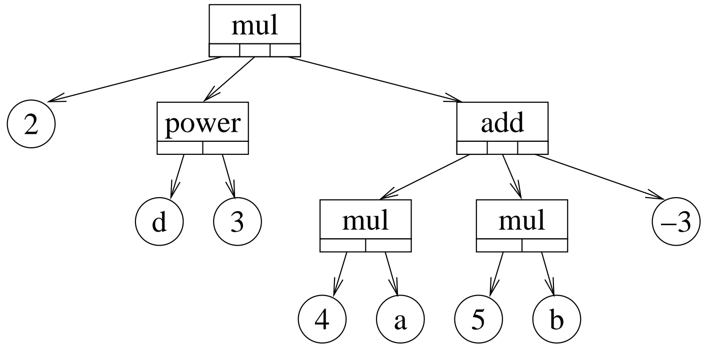
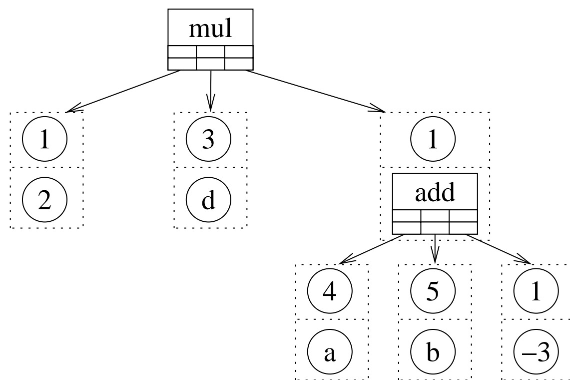


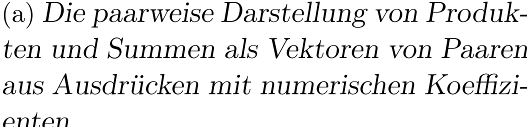
Abbildung 3.3.: *Naive (und ineffiziente) Darstellung des evaluierten Polynoms* $2d^3(4a + 5b - 3)$.

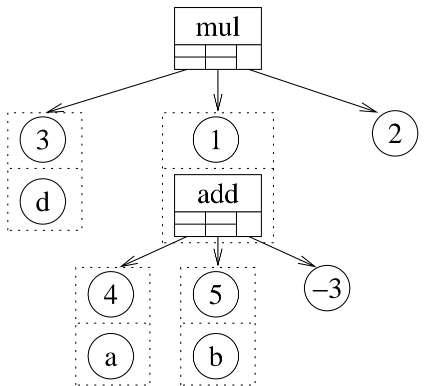
symbolischen Termen bestehen, zusammengefasst. Besser geeignet ist die distributive paarweise Darstellung aus Abbildung 3.4(a), bei der immer ein symbolischer Term mit einem Koeffizienten zusammengefasst wird. Da viele Evaluationsregeln für Summen und Produkte strukturell identisch sind, werden beide von einer Basisklasse `expairseq` abgeleitet (vergleiche Abbildung 3.1). Die Semantik der Koeffizienten ist für Summen diejenige multiplikativer Faktoren, während sie für Produkte diejenige von Exponenten ist: Der Regel $2a + a \rightarrow 3a$ bei Summen entspricht $a^2 a \rightarrow a^3$ bei Produkten. Eine beträchtliche Teilmenge der Evaluationsregeln braucht so nur einmal implementiert zu werden.

Außerdem gehört sowohl zu Summen als auch Produkten im Allgemeinen ein rein numerischer Koeffizient, wie die 5 in $2a + 3b + 5$ oder $5a^2b^3$. Dieser wird gesondert, also nicht paarweise, schon in `expairseq` untergebracht. Das Ergebnis, der Darstellungsbaum in Abbildung 3.4(b), erscheint zwar zunächst etwas unübersichtlicher, die Erfahrung zeigt aber, dass sich die Implementierung aller polynomialen Methoden (`expand`, etc.) nicht unbeträchtlich vereinfacht gegenüber den symmetrischeren und weniger effizienten Darstellungen.

Nur solche Vereinfachungen, die in Komplexität $< n^2$ ausgeführt werden können, werden von GiNaC automatisch im anonymen Evaluator durchgeführt. Bei multivariaten Polynomen heißt dies zum Beispiel, dass das Distributivgesetz Anwendung findet bei Multiplikation mit numerischen Faktoren, jedoch nicht bei Multiplikationen mit symbolischen Skalaren. Maple's und MuPAD's Evaluationsmaschinen verhalten sich sehr ähnlich. Diese Wahl ist natürlich willkürlich und kann bei der praktischen Implementierung zu überraschenden Problemen führen, die bei einer strengeren kanonischen Darstellung nicht auftreten würden. Die Probleme,




(a) *Die paarweise Darstellung von Produkten und Summen als Vektoren von Paaren aus Ausdrücken mit numerischen Koeffizienten*



(b) *Paarweise Darstellung mit Sonderstellung isolierter numerischer Koeffizienten (realisiert in GiNaC)*

Abbildung 3.4.: *Realistische Darstellungen von* $2d^3(4a+5b-3)$. *In der paarweisen Darstellung sind isolierte numerische Koeffizienten (wie die* -3 *in* $4a+5b-3$*) stets ganze Ausdrücke mit numerischem Koeffizienten* 1.

die in den Kästen auf Seiten 75, 106 und 109 beschrieben werden, sind letztlich alle darauf zurückzuführen.

Methodenfortpflanzung

Ganz im Geiste der objektorientierten Programmierung sind die Methoden auf Klassen in der Regel rekursiv definiert, solange sie in ein Schema passen, das den Baum entweder top-down oder bottom-up durchschreitet. Containerobjekte $C(x_0,\ldots,x_{n-1})$ geben die entsprechende Methode f zunächst an ihre Kinder weiter, wenden ihre eigene Implementierung von f auf die Ergebnisse $f(x_0)\ldots f(x_{n-1})$ an und geben das Gesamtergebnis dann zurück. Top-down (auch *preorder traversal*) unterscheidet sich natürlich nur dadurch von bottom-up (*postorder traversal*), dass in ersterer Strategie f erst auf das Objekt selbst angewendet wird und dann auf die Kinder, während diese Reihenfolge in der zweiten Strategie umgekehrt ist. Die Ziffern in Abbildung 3.5 kennzeichnen die so zustande kommenden Reihenfolgen.

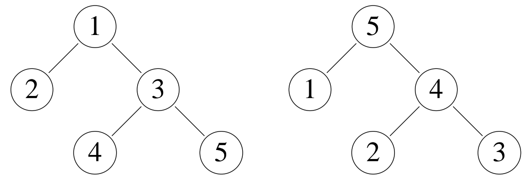


Abbildung 3.5.: *Baumdurchschreitung: Preorder und Postorder*

Man denke zum Beispiel an die Differentiation. Ein Objekt der Containerklasse `add` differenziert erst seine Kinder und gibt dann die Summe der Ergebnisse zurück, ein Objekt der Containerklasse `mul` muss zusätzlich noch die Produktregel implementieren. Eine vollständige Auflistung der implementierten Regeln findet sich in [Baue 2000]. Obwohl das Verfahren glasklar erscheint, verursachte es dennoch reichlich Kopfzerbrechen, bis eine ausreichende Effizienz erreicht war. Die Subtilität liegt hier auf einer Wechselwirkung mit GiNaC's Hashwert-unterstütztem syntaktischem Vergleichen von Ausdrücken und der internen Darstellungsweise von Produkten und Summen (siehe Kasten auf Seite 75).

Effizientes Differenzieren

Vor Version 0.6.3 waren die Ausdrücke, die GiNaC beim Differenzieren erzeugte, häufig zu umständlich – höhere Ableitungen wurden mitunter hoffnungslos ineffizient. Die Ursache war eine Wechselwirkung der Ableitungsregeln mit dem hashwertunterstützten syntaktischen Vergleichen von Ausdrücken. So werden $2x(1+x)$ *und* $x(2+2x)$ *nicht automatisch als äquivalent erkannt, da ihnen verschiedene Darstellungen zukommen. Die Ursache sieht man am leichtesten ein, wenn man rationale Funktionen mehrfach ableitet. Man differenziert beispielsweise* P'/P, $P \in \mathbb{Q}[x]$ *zweimal:*

$$\begin{aligned} \left(\frac{P'}{P}\right)' &= \frac{P''}{P} - \frac{(P')^2}{P^2} \\ \left(\frac{P'}{P}\right)'' &= \frac{P'''}{P} - \frac{P''P'}{P^2} - 2\frac{P'P''}{P^2} + 2\frac{(P')^3}{P^3} = \frac{P'''}{P} - 3\frac{P''P'}{P^2} + 2\frac{(P')^3}{P^3}. \end{aligned}$$

Der letzte Schritt kann von einer Maschine nur dann ausgeführt werden, wenn sie die Gleichheit der beiden Terme $\frac{P''P'}{P^2}$ *und* $\frac{P'P''}{P^2}$ *erkennt. Die Möglichkeit hierfür kann jedoch bei der Anwendung der Produktregel schon in der ersten Ableitung verbaut werden. Setzt man zum Beispiel* $P = x + x^3$, *so erzeugt die Ableitung von* P'/P *nach* x

$$\left(\frac{P'}{P}\right)' = 6\frac{x}{x+x^3} - \frac{(1+3x^2)^2}{(x+x^3)^2}.$$

Leiten wir den zweiten Term nochmals ab, so erzeugt die Ableitung des Zählers $2(1+3x^2)6x$, *wenn sie als ein* `mul`*-Objekt innerer Ableitung mal Argument* $(1+3x^2)$ *mit um Eins erniedrigtem Exponenten mit dem Exponenten als Überalles-Koeffizienten dargestellt wird. Stellt man sie stattdessen als innere Ableitung mal äußerer Ableitung dar, so erhält man* $6(2+6x^2)x$, *da die Regel* $2(1+3x^2) \to (2+6x^2)$ *für* `mul`*-Objekte automatisch angewendet wird. Dies ist zwar algebraisch korrekt, führt jedoch dazu, dass der entstehende Term nicht mehr automatisch zusammengefasst wird mit dem Term* $-6\frac{x(1+3x^2)}{(x+x^3)^2}$, *der aus der Ableitung des ersten Summanden entsteht. Weiteres Differenzieren des Ergebnisses verschlimmert die Situation rasch. Man verifiziert auch leicht, dass selbst eine direkte Implementierung der Leibniz-Regel für höhere Ableitungen von Produkten* $\frac{d^n}{dx^n}(PQ) = \sum_{i=0}^{n}\binom{n}{i}\frac{d^i}{dx^i}P\frac{d^{n-i}}{dx^{n-i}}Q$ *keine Abhilfe schafft. Betroffene Ausdrücke kommen in Schleifenrechnungen nicht selten vor, das Muster* P'/P *ist dort typisch – außerdem ist es natürlich gleich* $\log(P)'$.

Das Ausmultiplizieren mittels Aufruf von `.expand()` ist ein typisches Beispiel, wie Methoden sich rekursiv durch einen Baum fortpflanzen. Die Performanz dieser häufig gebrauchten Methode ist empfindlich von der internen Darstellung abhängig. Wir wollen GiNaC mit ein paar anderen symbolischen Maschinen vergleichen und benutzen hierzu den als Denny-Fliegner-Test bekannten Benchmark.[15] Er besteht aus drei Schritten:

- Sei e die (expandierte) Summe der n Symbole $\{a_0, \dots a_{n-1}\}$ quadriert: $e \leftarrow (\sum_{i=0}^{n-1} a_i)^2$. Der Ausdruck e besteht also aus $n(n+1)/2$ Termen.
- In e substituiere $a_0 \leftarrow -\sum_{i=2}^{n-1} a_i$. Es werden also in n Termen der Summe jeweils ein Symbol ersetzt durch Ausdrücke bestehend aus $n-2$ Summanden.

[15] Er ist benannt nach Denny Fliegner, der ihn während der Parallelisierung von Form zur Konsistenzüberprüfung benutzte.

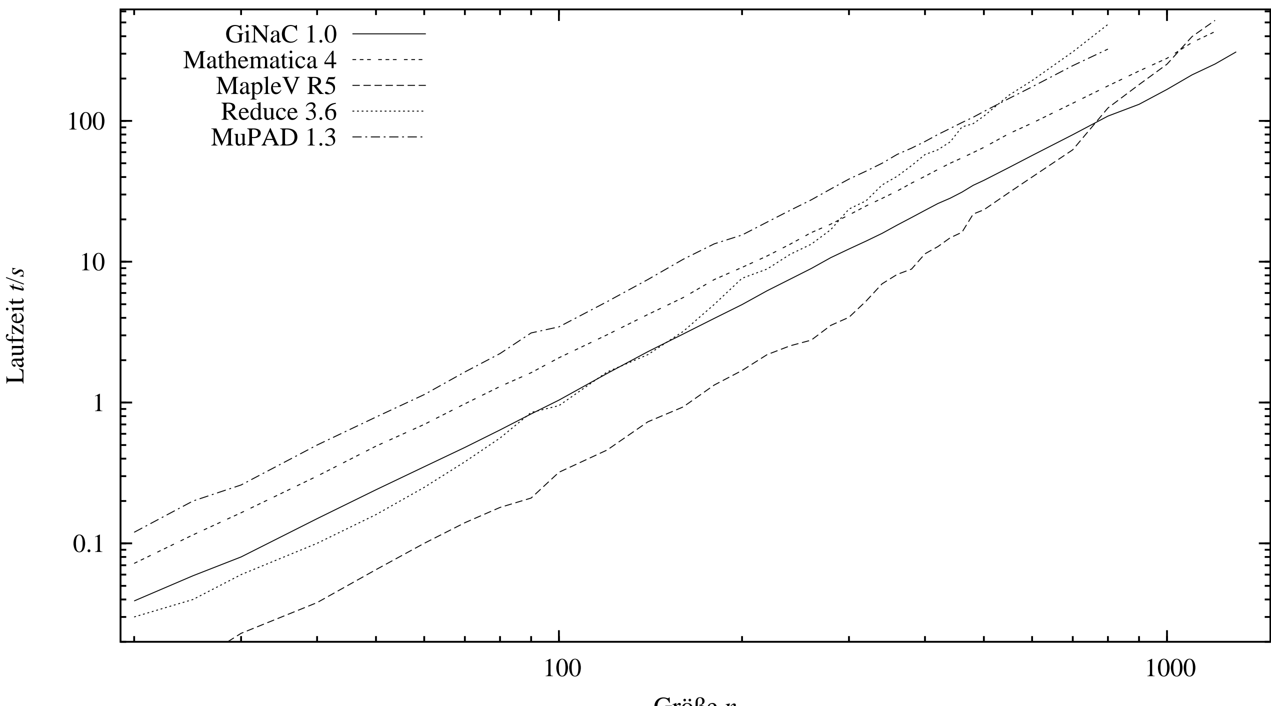


Abbildung 3.6.: *Laufzeiten verschiedener Symboliksysteme für Denny Fliegners Konsistenztest auf einer Alpha-Architektur.*

- Expandiere e. Das Ergebnis ist dann a_1^2.

Abbildung 3.6 zeigt, dass die Systeme Mathematica, MuPAD [Fuch 1997] und GiNaC das natürliche n^2-Skalierungungsverhalten aufweisen, wobei GiNaC das schnellste System ist. Die zwei Systeme Reduce und Maple hingegen sind überraschend schnell bei kleinen Problemen, werden jedoch zunehmend langsamer, wenn man das Problem vergrößert. Dies ist eine direkte Konsequenz der unterschiedlichen Speicherverwaltung: Maple und Reduce benutzen einen Garbage-Collector während Mathematica, MuPAD und GiNaC eine Referenzzählung implementieren, was häufige Zeiger-Dereferenzierungen erfordert (ein aufwändiger Prozess, wenn die gesuchten Daten nicht mehr in einem der Caches der Architektur liegen). Außer dem Skalierungsverhalten lässt sich an den Endpunkten der Kurven auch noch die Effizienz der Speicherverwaltung aus der Abbildung ablesen: Jedem System wurden für die Berechnung maximal 1GB Speicher zur Verfügung gestellt und die Kurve endet bei der damit erreichbaren Zahl n von Termen in e. Einige Systeme verschwenden diesen deutlich großzügiger als andere.

4. GiNaC: Implementierung

«Je ne sers à rien, dit Bruno avec résignation. Je suis incapable d'élever des porcs. Je n'ai aucune notion sur la fabrication des saucisses, des fourchettes ou des téléphones portables. Tout ces objets qui m'entourent, que j'utilise ou que je dévore, je suis incapable de les produire; je ne suis même pas capable de comprendre leur processus de production. [. . .] mes compétences techniques personnelles sont largement inférieures à celles de l'homme de Néanderthal.»
Michel Houellebecq: «Les particules élémentaires»

In diesem Kapitel werden Aspekte der Implementierung von GiNaC beschrieben. Dabei wird das Hauptaugenmerk auf solchen Aspekten liegen, die im Rahmen dieser Arbeit behandelt worden sind sowie auf der Motivation von Designentscheidungen die nicht an anderer Stelle schon dargestellt worden sind.

4.1. Die wichtigsten Klassen

Alle von `basic` abgeleiteten Klassen aus Abbildung 3.1 nehmen automatisch an der Speicherverwaltung teil, sobald sie von einem Objekt der Klasse `ex` verwaltet werden. Wenn es sich um Containerklassen handelt, enthalten sie weitere Subobjekte vom Typ `ex`. Die Klassen implementieren jedoch selbst auch automatische Umformungen in einem ⇒ Evaluator. Hierfür überschreibt in GiNaC jede Klasse eine Methode namens `eval`. In den folgenden Abschnitten werden die einzelnen Klassen beschrieben und diese automatischen Umformungen aufgezählt.

Vereinfachungen und kanonische Form

Die Menge der Vereinfachungen, die in einem CAS automatisch vom Evaluator durchgeführt werden dürfen, muss sorgfältig auf innere Konsistenz abgeklopft werden. Im Falle von Funktionen und ihren Inversen ist zum Beispiel wohlbekannt, dass die Regel $e^{\ln x} \to x$ unbedenklich ist, die Regel $\ln e^x \to x$ jedoch nicht für beliebige x anwendbar ist, da die Exponentialfunktion als Abbildung nicht injektiv ist (dies wird bisweilen Aslaksen-Test genannt, nach [Asla 1996]). Eine automatische Vereinfachung darf also nicht durchgeführt werden, da dies zu Widersprüchen führen würde, wie das Beispiel $\ln e^{3\pi i} = \ln(-1) = \pi i$ verdeutlicht.[1] Alternativ dürfte

[1] Reduce [Hear 1995] erlaubt sich, dies dennoch zu vereinfachen. Dies führt jedoch an anderer Stelle zu Überraschungen, wie im folgenden Dialog:

man die Vereinfachung durchführen, wenn man dafür in Kauf nimmt, die Entwindungszahl $\mathcal{K}$ oder die η-Funktion (siehe Anhang A) in die rechte Seite mit aufzunehmen. Es ist jedoch noch kein Algorithmus bekannt, der diesen Zusatzterm im weiteren Verlauf der Rechnung im Allgemeinen wieder zu eliminieren vermag [CDJLW 2001].

Ähnlich wie in Mathematica oder Maple stellen wir uns jedes freie Symbol in einer Rechnung als unbekanntes Element aus $\mathbb{C}$ vor – schließlich kann es zu einem späteren Zeitpunkt durch eine komplexe Zahl ersetzt werden. Denkbar wäre an dieser Stelle, auch unfreie Symbole zuzulassen, indem man sie spezifiziert als Element eines Bereiches, zum Beispiel als reell. Daraus aufgebaute Ausdrücke würden dann abgefragt werden können, ob sie aus diesem Bereich sind (Polynome sind Element des größten Bereiches der in ihren Symbolen vorkommt). Dies ist mit Bereichen noch realisierbar, bricht aber zusammen, wenn Eigenschaften wie Positivität eines Symbols spezifiziert werden sollen: Sind $x, y > 0$, so ist die Frage ob $x-y > 0$ ist, unentscheidbar und bedarf daher einer ternären Logik oder zweiten Abfrage mit negiertem Prädikat (also $x - y < 0$). Inferenzmaschinen für die Entscheidungsfindung solcher Prädikate sind jedoch notorisch von exponentieller Komplexität und immer noch ein aktiver Wissenschaftszweig für sich – genug Gründe warum wir uns nicht damit abgeben wollen und können.

4.2. Kanonisierung von Produkten: die Klassen ‚mul' und ‚ncmul'

Produkte von kommutierenden symbolischen Objekten werden in der Klasse `mul` dargestellt. Nichtkommutative Produkte werden in der Klasse `ncmul` gesondert sortiert. Wenden wir uns zunächst den kommutativen Produkten zu. Wie schon in Abbildung 3.4(b) auf Seite 74 gezeigt, besteht die Darstellung genau wie bei Summen aus Paaren von symbolischen Ausdrücken und Zahlen. Wo bei der Klasse `add` die Zahlen multiplikative Faktoren darstellten, sind es bei der Klasse `mul` Exponenten, in jedem Fall haben sie additive Semantik. Zusätzlich gibt es jeweils noch ein zusätzliches numerisches Argument, in Abbildung 3.4(b) rechts dargestellt.

Der anonyme Evaluator der Klasse `mul` darf im Gegensatz zum Konstruktor allgemeine symbolische Ausdrücke zurückgeben und zum Beispiel im Produkt $x^1 \cdot x^{-1}$ die Faktoren mit gleichen symbolischen Termen zusammenfassen zum nicht-Produkt 1. Er bedient sich der folgenden

```
1 1: log(exp(x));
2 x
3 2: log(exp(3*Pi*i));   % log(-1.0000)
4 log(-1)
5 3: log(exp(47/5*i));   % log(≃-0.9997+0.0248*i)
6 (47*i)/5
```

Das System Yacas [Pink 2000] erlaubt sich dies sogar konsequent bei allen trigonometrischen Funktionen:

```
1 In( 0 ) = Ln(Exp(x));
2 Out( 0 ) = x;
3 In( 1 ) = ArcSin(Sin(x));
4 Out( 1 ) = x;
```

Regelmenge, wobei c_i für numerische Objekte und x_i für nichtnumerische stehen soll:

$$
\begin{aligned}
\texttt{mul}(\cdot c) &\rightarrow c \\
\texttt{mul}(x_1 \cdot 1) &\rightarrow x_1 \\
\texttt{mul}(\ldots \cdot x_n^{c_n} \cdot 0) &\rightarrow 0 \\
\texttt{mul}(\ldots \cdot c_1^1 \cdot \ldots \cdot c) &\rightarrow \texttt{mul}(\ldots \cdot (c_1 c)) \\
\texttt{mul}(\ldots \cdot x_1^{c_1} \cdot 1 \cdot x_2^{c_2} \cdot \ldots) &\rightarrow \texttt{mul}(\ldots \cdot x_1^{c_1} \cdot x_2^{c_2} \cdot \ldots) \\
\texttt{mul}(\texttt{add}(\ldots + c_1{\cdot}x_1 + c_2{\cdot}x_2 + \ldots) \cdot c) &\rightarrow \texttt{add}(\ldots + (c\,c_1){\cdot}x_1 + (c\,c_2){\cdot}x_2 + \ldots)
\end{aligned}
$$

Hierin tauchen Zahlen immer in der Form c_i^1, also mit dem Exponenten 1 auf. Dafür sorgt der anonyme Evaluator der Klasse `power` (siehe nächster Abschnitt). Man bemerke, dass diese Regelmenge im letzten Schritt das Distributivgesetz beinhaltet, wenn einer der Koeffizienten numerisch ist. Die Anzahl der berechneten Terme bleibt dann nämlich konstant, während sie im symbolischen Fall mit dem Produkt der Anzahl der Ausgangsterme ansteigt – hierfür muss explizit `expand` aufgerufen werden.

Nichtkommutative Produkte sind einfache Vektoren von nichtkommutativen Objekten – es gibt keine Exponenten repräsentierende numerischen Koeffizienten. Da die einzelnen Objekte darin verschiedenen Algebren angehören können, schreiben wir zur Unterscheidung $x_i, x_j \in \mathcal{A}_x$ und $y_i, y_j \in \mathcal{A}_y$. Die Regelmenge des anonymen Evaluators lautet:

$$
\begin{aligned}
\texttt{ncmul}() &\rightarrow 1 \\
\texttt{ncmul}(x_1) &\rightarrow x_1 \\
\texttt{ncmul}(\ldots \cdot c_1 \cdot \ldots \cdot c_2 \cdot \ldots) &\rightarrow \texttt{mul}(\texttt{ncmul}(\ldots) \cdot (c_1 c_2)) \\
\texttt{ncmul}(\ldots \cdot x_1 \cdot y_1 \cdot \ldots \cdot x_2 \cdot y_2 \cdot \ldots) &\rightarrow \texttt{mul}(\texttt{ncmul}(x_1 \cdot x_2 \cdot \ldots) \cdot \texttt{ncmul}(y_1 \cdot y_2 \cdot \ldots)) \\
\texttt{ncmul}(\ldots \cdot x_1 \cdot \ldots \cdot \texttt{ncmul}(x_2 \cdot x_3 \cdot \ldots)) &\rightarrow \texttt{ncmul}(x_1 \cdot x_2 \cdot x_3 \cdot \ldots)
\end{aligned}
$$

Numerische und andere kommutative Objekte werden also aus dem `ncmul`-Objekt herausgezogen und in einem übergeordneten `mul`-Objekt untergebracht. All dies geschieht völlig transparent für den Programmierer. Im Gegensatz zu den meisten anderen Computeralgebrasystemen werden nichtkommutative Produkte mit dem überladenen `*`-Operator aus nichtkommutativen Objekten aufgebaut.[2] Dies entspricht dem mathematischen Bild, dass die Eigenschaft, nicht zu kommutieren, eine Eigenschaft der Objekte der Algebra ist, während die aus Maple und Reduce bekannte Schreibweise mit dem `&*`-Operator dem Bild entspricht, dass die Nichtkommutativität eine Eigenschaft des Produktes in dieser Algebra ist. Beide Ansichten sind freilich äquivalent, die Benutzung eines einzigen Operators ist jedoch sehr attraktiv, beseitigt sie doch eine häufige Fehlerquelle.

4.3. Vereinfachungen in der Klasse ‚power'

Das Aufstellen von Regeln zur automatischen Vereinfachung von Potenzobjekten stellt sich als besonders schwierig heraus. Im Gegensatz zu Summen und Produkten von einfachen Symbolen

[2] [ISO 1998, *Abschnitt 1.9.15*] spezifiziert, dass ein Compiler überladene Operatoren niemals als kommutativ annehmen darf.

ist es ist nicht möglich, eine kanonische Form anzugeben, die effizient den Gleichheitstest solcher Objekte ermöglicht. Zudem gibt es Klassen von Potenzobjekten, bei denen, selbst wenn man auf eine kanonische Form verzichtet, ein Gleichheitstest mit algorithmischen Mitteln gar nicht möglich ist. Es gibt drei Klassen von algebraischen Zahlen: einfache Wurzeln ($\sqrt{2}$), verschachtelte Wurzeln ($\sqrt{1+\sqrt{2}}$) und solche, die sich nicht als Wurzeln schreiben lassen, sondern lediglich als Nullstellen algebraischer Gleichungen – wie die Lösung von $x^5-x+1=0$. Da Letztere nicht mehr von Objekten der Klasse `power` dargestellt werden können, betrachten wir kurz die Zweite. Eine kanonische Form zu finden, in die sich alle algebraischen Zahlen dieser Klasse umformen lassen, ist zwar nicht prinzipiell unmöglich in dem Sinne, dass das Problem zumindest noch wohldefiniert ist, erfordert aber im allgemeinen Fall bisher nicht bekannte algorithmische Hilfsmittel [JeRi 1999, Land 2002]. Eine nahe liegende kanonische Form könnte darin bestehen, dass man verschachtelte Wurzeln stets umschreibt in Summen aus einfachen Wurzeln – aus $\sqrt{9+\sqrt{32}}$ macht man z.B. $1+\sqrt{8}$. Falls der Grad der Verschachtelung zwei nicht überschreitet und es sich lediglich um Quadratwurzeln handelt, gibt es Algorithmen, die diese Vereinfachung bewältigen [DST 1988] (welche jedoch in keinem Computeralgebrasystem für Erzeugung kanonischer Formen verwendet zu werden scheinen). Für Objekte mit höheren als Quadratwurzeln, wie die linke Seite der auf S. Ramanujan zurückgehenden Identität

$$\sqrt[3]{\sqrt[5]{32/5}-\sqrt[5]{27/5}} = \sqrt[5]{1/25}+\sqrt[5]{3/25}-\sqrt[5]{9/25}, \tag{4.1}$$

scheinen jedoch gar keine Algorithmen bekannt zu sein um sie (falls möglich) in Summen einfacher Wurzeln wie die rechte Seite umzuwandeln. Es bleibt also die Frage, ob für Objekte der ersten Klasse von algebraischen Zahlen, also unverschachtelte Wurzeln, eine kanonische Form gefunden werden kann. Die Antwort lautet ja, aber der Aufwand wächst zu stark, um eine Implementierung sinnvoll erscheinen zu lassen. Wie man schnell einsieht wären einfache Wurzeln zwar zerlegbar in Produkte aus einfachen Wurzeln von Primzahlen $\sqrt[3]{15}\to\sqrt[3]{3}\sqrt[3]{5}$, welche dann nach Größe sortiert werden könnten, dies liefe aber offensichtlich auf Primfaktorzerlegung eventuell großer ganzer Zahlen hinaus, was unangemessenen Rechenaufwand erfordert. Wenn wir jetzt noch konstatieren, dass bisher nur auf numerische Argumente der innersten Wurzeln eingegangen worden ist und die Komplikationen im symbolischen Fall noch gar nicht erwähnt worden sind, glaubt man vielleicht, dass für Wurzeln überhaupt keine kanonische Form praktikabel ist.

Für die Darstellung der Termumschreibungsregeln der Klasse `power` beginnen wir mit der

Definition 4.1 (Potenz) *Als Definition setzen wir hier voraus*

$$x^a \equiv e^{a\ln x} \tag{4.2}$$

wo der Logarithmus den Schnitt (siehe Anhang A) entlang der negativen reellen Achse habe:

$$\ln x \equiv \ln|x| + i\arg x, \qquad \arg x \in (-\pi,\pi]$$

Wir notieren, dass hieraus erst die Regel $x^a x^b \to x^{a+b}$ zum Zusammenfassen in der Klasse `mul` folgt: für beliebige komplexe x und y gilt schliesslich $e^x e^y = e^{x+y}$, woraus natürlich wegen 4.2 auch

$$x^a x^b \equiv e^{a\ln x}e^{b\ln x} = e^{(a+b)\ln x} \equiv x^{a+b}$$

folgt $\forall x, a, b \in \mathbb{C}$. Diese Regel wird schon in `mul::eval()` für numerische Exponenten angewendet, für symbolische bleibt die linke Seite vom Evaluator ganz unangetastet.

Da algebraische Korrektheit die Generalvoraussetzung bei allen Umformungen ist, aber Vereinfachungspotenzial ausgeschöpft werden soll wo immer möglich, werden die Regeln für den anonymen Evaluator (implementiert in der Methode `power::eval()`) überraschend komplex. Wir zählen im Folgenden diese Vereinfachungen mit ihrem jeweiligen Gültigkeitsbereich auf und beweisen sie.

$$\begin{aligned}
\texttt{power(}x\texttt{\^{}1)} &\rightarrow x \\
\texttt{power(}x\texttt{\^{}0)} &\rightarrow 1 \\
\texttt{power(1\^{}}x\texttt{)} &\rightarrow 1 \\
\texttt{power(}c_1\texttt{\^{}}c_2\texttt{)} &\rightarrow {c_1}^{c_2}
\end{aligned}$$

Diese trivialen Vereinfachungsregeln folgen sofort aus der Definition. Die letzte Regel ist so zu lesen, dass die Exponentiation numerisch ausgeführt wird, sofern dies exakt möglich ist, etwa in $2^3 \rightarrow 8$ oder $27^{1/3} \rightarrow 3$ – auf die Ausnahme 0^a für numerisches a werden wir weiter unten auf Seite 83 noch einmal zurückkommen. Diese Regel wird in der Praxis so modifiziert, dass `power::eval()` dafür sorgt, dass der Wert des gebrochenrationalen Anteils des Exponenten zwischen 0 und 1 zu liegen kommt, falls die Exponentiation nicht exakt ausgeführt werden kann, also z.B. $7^{-3/2} \rightarrow 7^{-2}\,7^{1/2}$. Es sei angemerkt, dass Identitäten mit verschachtelten Wurzeln wie Gleichung (4.1) oben mit diesem Mittel zwar nicht herleitbar sind, aber durch Exponentiation zumindest schon in den Bereich der exakten automatischen Verifizierbarkeit rücken.[3]

$$\text{Für } c \in \mathbb{Z}: \quad \texttt{power(mul(}\cdots x_1 \cdot x_2 \cdots\texttt{)\^{}}c\texttt{)} \rightarrow \texttt{mul(}\cdots\texttt{power(}x_1\texttt{\^{}}c\texttt{)} \cdot \texttt{power(}x_2\texttt{\^{}}c\texttt{)}\cdots\texttt{)}$$

Um diese Regel einzusehen führen wir eine Fallunterscheidung durch. Für $c = 0$ ist es trivial. Sei daher zunächst $c > 0$:

$$(x_1x_2)^c = \underbrace{(x_1x_2)\cdots(x_1x_2)}_{c\times} = \underbrace{x_1\cdots x_1}_{c\times}\underbrace{x_2\cdots x_2}_{c\times} = {x_1}^c{x_2}^c.$$

Ist hingegen $c < 0$, so führt man auf den Fall $c > 0$ zurück:

$$(x_1x_2)^c = \frac{1}{(x_1x_2)^{|c|}} = \frac{1}{x_1^{|c|}x_2^{|c|}} = x_1^{-|c|}x_2^{-|c|} = x_1^c x_2^c.$$

Falls der Exponent nicht ganzzahlig ist, so kann unter Umständen eine ähnliche Regel angewendet werden:

$$\begin{aligned}
\text{Für } c_1 > 0: \quad & \texttt{power(mul(}x \cdot c_1\texttt{)\^{}}c_2\texttt{)} \rightarrow \texttt{mul(power(}x\texttt{\^{}}c_2\texttt{)} \cdot {c_1}^{c_2}\texttt{))} \\
\text{Für } c_1 < 0: \quad & \texttt{power(mul(}x \cdot c_1\texttt{)\^{}}c_2\texttt{)} \rightarrow \texttt{mul(power(}-x\texttt{\^{}}c_2\texttt{)} \cdot {c_1}^{c_2}\texttt{))}
\end{aligned}$$

[3] Tatsächlich gehört eine Variation dieser Gleichung zur Suite der Regressionstests von GiNaC.

Ist $c_1 \in \mathbb{R}$ positiv und $x, c_2 \in \mathbb{C}$ beliebig (x symbolisch, da sonst `mul::eval()` schon multipliziert hätte), so darf diese Regel auch angewendet werden, da

$$(c_1 x)^{c_2} = e^{c_2 \ln(c_1 x)} = e^{c_2(\ln|c_1 x| + i \arg(c_1 x))},$$

worin wegen

$$\ln|c_1 x| = \ln|c_1| + \ln|x| = \ln c_1 + \ln|x| \quad \text{und} \quad \arg(c_1 x) = \arg x$$

vereinfacht werden darf zu

$$(c_1 x)^{c_2} = e^{c_2(\ln c_1 + \ln|x| + i \arg x)} = e^{c_2(\ln c_1 + \ln x)} = e^{c_2 \ln c_1} e^{c_2 \ln x} = {c_1}^{c_2} x^{c_2}.$$

Für negative c_1 ist dies nicht richtig, da zum Beispiel im Falle $c_1 = -1$, $c_2 = \frac{1}{2}$ und $x = e^{i\pi/4}$ $(c_1 x)^{c_2} = (e^{-i3\pi/4})^{\frac{1}{2}} = e^{-i3\pi/8}$, aber ${c_1}^{c_2} x^{c_2} = e^{i\pi/2} e^{i\pi/8} = e^{i5\pi/8}$.

Für $c_2 \in \mathbb{Z}$ oder $-1 < c_1 \leq 1$: `power(power(`x`^`c_1`)^`c_2`)` $\rightarrow$ `power(`x`^(`$c_1\, c_2$`))`

Wieder seien $x, c_1 \in \mathbb{C}$ beliebig und wir führen eine Fallunterscheidung durch: Sei $c_2 > 0$, dann ist

$$(x^{c_1})^{c_2} = \underbrace{(x^{c_1}) \cdots (x^{c_1})}_{c_2 \times} = \underbrace{e^{c_1 \ln x} \cdots e^{c_1 \ln x}}_{c_2 \times} = e^{\overbrace{c_1 \ln x + \cdots + c_1 \ln x}^{c_2 \times}} = e^{c_1\, c_2 \ln x} = x^{c_1\, c_2}.$$

Im Falle $c_2 < 0$ führt man es durch Setzen von $c_2 = -|c_2|$ wieder auf den ersten Fall zurück:

$$(x^{c_1})^{c_2} = \frac{1}{(x^{c_1})^{|c_2|}} = \frac{1}{x^{c_1\, |c_2|}} = x^{-c_1\, |c_2|} = x^{c_1\, c_2}.$$

Im Falle $-1 < c_1 \leq 1$ und $x, c_2 \in \mathbb{C}$ beliebig ist die Regel ebenfalls anwendbar. Beweis:

$$x^{c_1} = e^{c_1 \ln|x| + i c_1 \arg(x)}$$

Falls $c_1 \in \mathbb{R}$, ist $|x^{c_1}| = e^{c_1 \ln|x|}$ und $\arg(x^{c_1}) - c_1 \arg(x) = 2k\pi$. Wenn nun $-1 < c_1 \leq 1$, dann ist $-\pi < c_1 \arg(x) \leq \pi$, und folglich $k = 0$, also

$$\arg(x^{c_1}) = c_1 \arg(x)$$

Man beachte, dass dies für $c_1 = -1$ im Allgemeinen nicht mehr richtig ist, da die rechte Seite $-\arg(x)$ dann auch $-\pi$ sein kann. Also ist

$$\begin{aligned}
\ln(x^{c_1}) &= \ln|x^{c_1}| + i \arg(x^{c_1}) \\
&= \ln(e^{c_1 \ln|x|}) + i c_1 \arg(x) \\
&= c_1 \ln|x| + i c_1 \arg(x) \text{ (weil } c_1 \ln|x| \in \mathbb{R}) \\
&= c_1 \ln x.
\end{aligned}$$

Daher gilt

$$(x^{c_1})^{c_2} = e^{c_2 \ln x^{c_1}} = e^{c_2\, c_1 \ln x} = x^{c_1\, c_2}.$$

Ausnahmen

Die Ausnahme 0^a für numerisches a ist nicht ganz unumstritten. Klar ist der Fall noch für reelle $a \neq 0$, wo für $a > 0$ das Ergebnis 0 sein soll, für $a < 0$ ein Überlauffehler auftritt. Für $a = 0$ wird häufig $0^0 \equiv 1$ gesetzt (z.B. in [Stee 1990], Abschnitt 12.5.3, die Spezifikation der Lisp-Funktion `expt`, mit der schlichten Bemerkung „*By definition,* $0^0 = 1$."). Tatsächlich ist die Kontroverse um 0^0 jahrhundertealt, wurde jedoch zunächst aus dem Blickwinkel der Analysis geführt. So ist beispielsweise $0^0 := \lim_{\varepsilon\to 0} \varepsilon^\varepsilon \equiv \lim_{\varepsilon\to 0} e^{\varepsilon \ln \varepsilon} \to e^0 = 1$. Dies ist aber ebenso willkürlich wie $0^0 := \lim_{\varepsilon\to 0} 0^\varepsilon = 0$ oder eine andere Definition. Aufschlussreicher ist es, algebraisch oder kombinatorisch an das Problem heranzugehen. So wird in [GKP 1989] darauf hingewiesen, dass die binomische Formel $(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$ für $x \neq 0$ und $y = 0$ ihre Gültigkeit verliert, wenn man $0^0 \neq 1$ setzt. Diese Argumentation zielt aber an den meisten CAS-Implementierungen vorbei, da dort die Regeln $x + 0 \to x$ und $x^0 \to 1$ ausgeführt werden, bevor in der binomischen Formel 0^0 auftauchen kann. Es wäre doch sehr naiv anzunehmen, dass die Eingabe $(x + 0)^2$ ein CAS an den Abgrund der Inkonsistenz treibt. Kombinatoriker wiederum argumentieren, dass man die Anzahl der Abbildungen der leeren Menge auf die leere Menge gerne als 0^0 schreiben würde und dies daher 1 sein muss.

Da wir aber in einem symbolischen Programm normalerweise beim Einsetzen der Regel nicht wissen, ob gerade Exponent oder Basis als Erstes gegen 0 gegangen ist, betrachtet GiNaC das Auftreten von 0^0 vorsichtshalber generell als Fehler. Für alle anderen komplexen Exponenten a ist der Fall jedoch wieder klar. Ist $a = ci$ rein imaginär, so formt man um $0^{ci} = e^{ci \ln 0}$, und sieht, dass das Ergebnis zwar auf dem Einheitskreis liegt, die Phase jedoch nicht definiert ist. Für andere komplexe Exponenten kann man sich dann auf den Fall $a = i \pm 1$ beschränken

$$0^{i\pm 1} = 0^i \, 0^{\pm 1} = \begin{cases} 0^i \, 0^1 & = 0 \\ 0^i \, 0^{-1} & = e^{i\varphi}\infty \end{cases}$$

womit klar ist, dass 0^a nur in der rechten komplexen Halbebene des Exponenten definiert ist. Dies stimmt immerhin mit [Stee 1990] und Mathematica überein, nicht jedoch mit MuPAD, Maple und Reduce. Die letzteren beiden scheinen die Regel $0^a \to 0$ vor den numerischen Sonderfall vorzuziehen. (Form hingegen kennt keine komplexen Zahlen und wendet die Regel sogar für reelle, negative a an; Version 1 und 2 kommen für reine Symbole vollends durcheinander und vereinfachen $0^a \to a^3$.)

4.4. Die Numerik-Klasse

Das bei symbolischen Algorithmen häufige Phänomen des Anwachsens von Zwischenergebnissen (engl: *intermediate expression swell*) macht eine solide Handhabung von exakten Zahlen, also ganzen Zahlen aus dem Integritätsbereich $\mathbb{Z}$ oder dem darüber gebildeten Quotientenkörper der rationalen Zahlen $\mathbb{Q}$ unabdingbar. Eine Beschränkung auf `ints` von Maschinengröße (wie etwa in der Bibliothek Magnum [Roth 1995]) oder auf Gleitkommazahlen mit fest eingestellter Mantisse (wie etwa 29 Dezimalstellen in Schoonschip [VeWi 1993]) ist nicht akzeptabel. Die meisten Computeralgebrasysteme verfügen daher über flexible Datentypen für ganze und rationale Zahlen, die lediglich durch den vorhandenen Hauptspeicher beschränkt sind. Eine

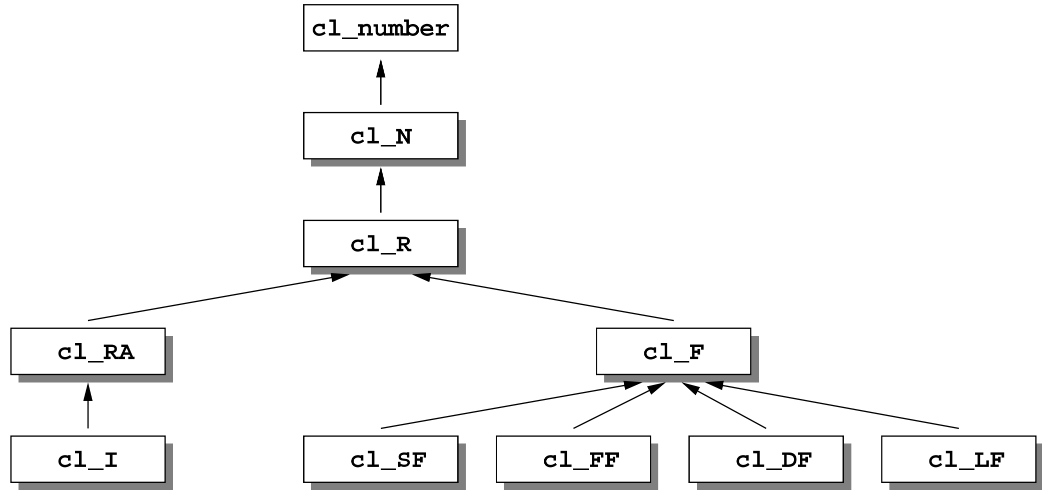


Abbildung 4.1.: *Übersicht über die Klassenhierarchie von CLN. Nur die Basisklasse* `cl_number` *wird von der GiNaC-Adapterklasse* `numeric` *verwaltet. Intern kommen die folgenden davon abgeleiteten Klassen vor:* `cl_N` *(komplexe Zahlen),* `cl_R` *(reelle Zahlen),* `cl_RA` *(gebrochenrationale Zahlen),* `cl_I` *(ganze Zahlen),* `cl_LF` *(Gleitkommazahlen beliebiger Genauigkeit).*

Ausnahme bildet FORM [Verm 1991] 2, welches in weitgehender Abwesenheit eines Speichermanagements einen (undokumentierten) Puffer von genau 400 Byte für jeweils eine ganze Zahl zur Verfügung zu stellen scheint.

GiNaC benutzt die Bibliothek CLN [HaKr 2000] für die Manipulation aller numerischen Objekte. Da die Benutzerschnittstelle von CLN ganz anders als diejenige von GiNaC gestaltet ist, bildet eine „Adapter“-Klasse namens `numeric` die Schnittstelle von CLN auf diejenige von GiNaC ab. Das Design von CLN selbst entspricht dem Muster der sogenannten „*Bridge*“: die Abstraktion (Abbildung 4.1) ist vollständig getrennt von der Implementation, von der der Benutzer nichts wissen muss und die sich sogar ändern kann ohne die Kompatibilität zu gefährden. Die nach außen sichtbare Seite ist die in Abbildung 4.1 dargestellte Klassenhierarchie. Man beachte, dass sie vom Gesichtspunkt der objektorientierten Programmierung Kopf zu stehen scheint: So ist die Klasse der ganzen Zahlen `cl_I` von der Klasse der rationalen Zahlen `cl_RA` abgeleitet, obwohl die Darstellung einer rationalen Zahl zwei ganze Zahlen beinhaltet. Die nach außen sichtbaren Klassen sind aber nur Schnittstellen und enthalten nur Zeiger auf die eigentliche Implementierung, ganz analog zur Klasse `ex` in GiNaC. Diese Klassenabstraktion in C++ kann daher völlig konsistent mit den Einbettungen der mathematischen Ringstrukturen sein, in der eine ganze Zahl beispielsweise auch eine rationale Zahl ist.

CLN war ursprünglich (in den frühen 1990er Jahren) eine reine C-Bibliothek und wurde später in einen konsistenten C++-Rahmen eingebettet. Sie kann als Referenzimplementierung für die Handhabung großer Zahlen gelten. Während Addition zweier ganzer Zahlen eine Routine mit linearer Ordnung in der Länge N der Argumente ist, ist die Multiplikation wesentlich aufwändiger. Die übliche Schulbuch-Multiplikation ist von der Ordnung $\mathcal{O}(N^2)$, aber es existieren Verbesserungen, wie etwa ein überraschend einfaches Verfahren von Karatsuba [KaOf 1962] mit der Ordnung $\mathcal{O}(N^{\log_2 3}) \simeq \mathcal{O}(N^{1{,}58})$ oder das wesentlich kompliziertere Verfahren von Schönhage und Strassen [SchSt 1971], welches die asymptotisch ideale Ordnung $\mathcal{O}(N \log N \log\log N)$ aufweist. Diese fortgeschrittenen Verfahren haben zwar eine attraktive asymptotische Ord-

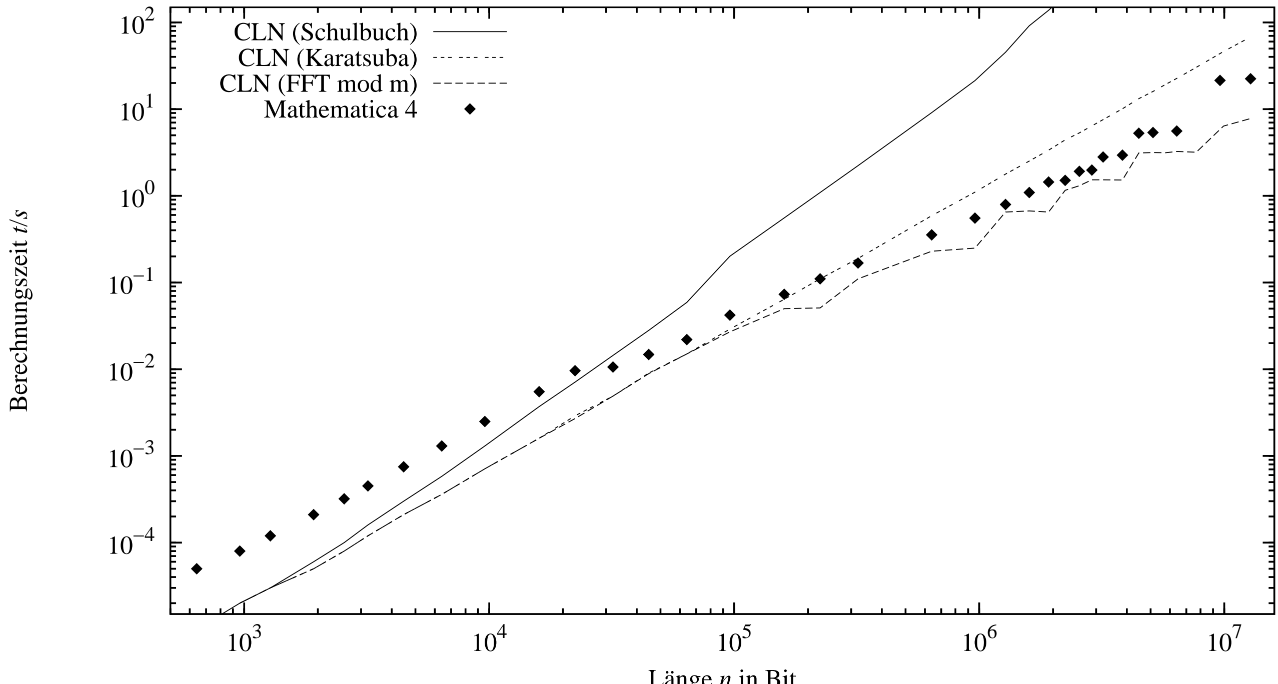


Abbildung 4.2.: *Gemessene Laufzeiten der Multiplikation zweier gleich großer ganzer Zahlen in CLN für verschiedene dort implementierte Algorithmen. Der Vergleich mit Mathematica zeigt, dass auch dieses System asymptotisch ideale Multiplikationsalgorithmen implementiert.*

nung, die Konstante davor macht sie aber für kleine Zahlen unbrauchbar. In der Praxis müssen also die Punkte, an denen der eine Algorithmus den anderen ablöst, bestimmt, und automatisch die passende Methode gewählt werden. Abbildung 4.2 zeigt die Laufzeiten in CLN für drei verschiedene Verfahren im Vergleich zu Mathematica. Man erkennt deutlich, dass auch bei Mathematica beträchtlicher Aufwand in eine schnelle Multiplikation investiert worden ist. Das asymptotische Verhalten und die Stufen in der Laufzeit als Funktion der Größe der Operanden am oberen Ende sind ein Indiz für eine FFT-basierte Methode, die ja meistens mit 2^n-Blöcken arbeitet.[4]

Eine schnelle Multiplikationsroutine kann für unsere symbolischen Rechnungen von direkter Relevanz sein. So greifen Divisionsroutinen und Algorithmen zur ggT-Bestimmung in $\mathbb{Z}$ darauf zurück. Die in GiNaC implementierte heuristische ggT-Bestimmung multivariater Polynome aus $\mathbb{Z}[X]$ macht Probedivisionen in $\mathbb{Z}$ [Baue 2000].

Die Fähigkeiten verschiedener Bibliotheken, bei der Multiplikation an die optimale asymptotische Geschwindigkeit heranzukommen, wurden eingehend in [Bern 2002] untersucht (graphisch aufbereitet in Abbildung 4.3). Die dort gemessenen Laufzeiten sind für CLN noch suboptimal, da die Bibliothek ohne Unterstützung der Assemblerroutinen aus der MPN-Schicht der GNU Multi Precision Bibliothek GMP übersetzt wurde. Es muss auch beachtet werden, dass einige Systeme vom Benutzer eine explizite Speicherallozierung für das Rechenergebnis verlangen, was CLN (für den Benutzer) in transparenter Weise erledigt. Der Overhead hierfür wird so

[4] Bei Operandengrößen $\lesssim$ 5000 Bit wurde im Falle von Mathematica die Zeit für eine leere Messschleife berücksichtigt. Für die Langsamkeit dort habe ich keine plausible Erklärung.

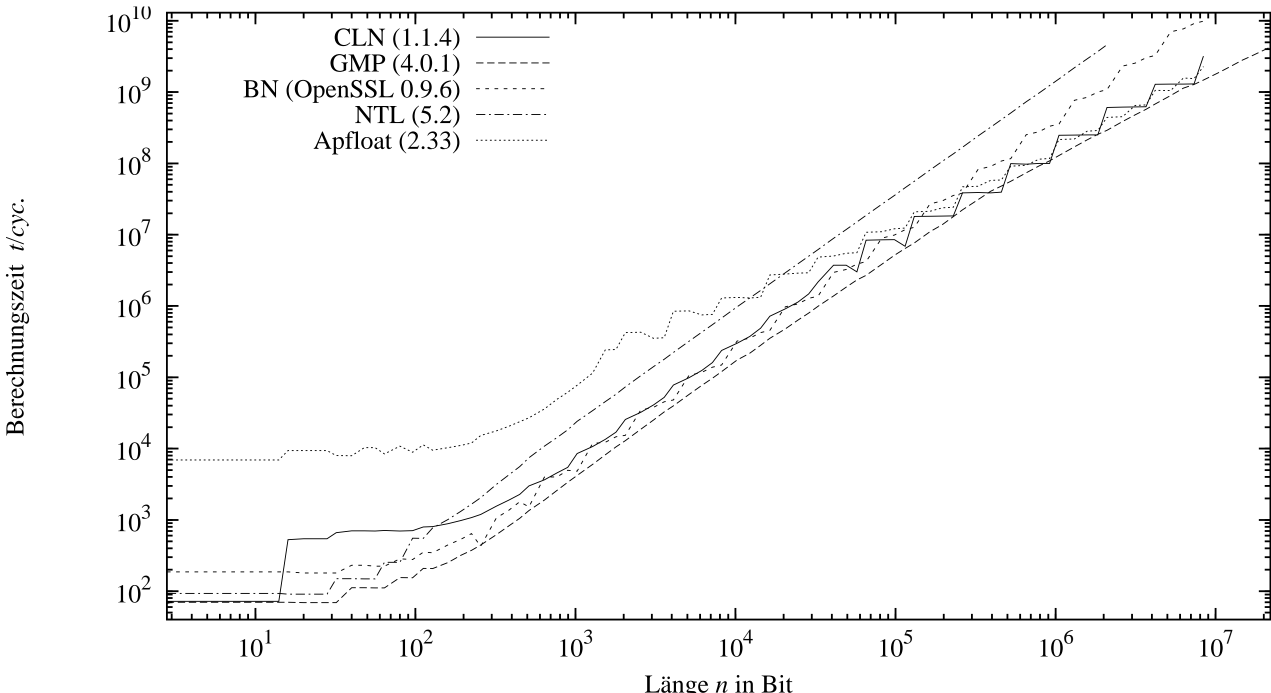


Abbildung 4.3.: *Gemessene Laufzeiten in Taktzyklen der Multiplikation zweier gleich großer ganzer Zahlen in CLN und verschiedenen anderen Softwarepaketen, nach Daten gemessen in [Bern 2002]. Als Testsystem diente ein mit 900MHz getakteter AMD Athlon unter FreeBSD.*

aber zur Multiplikation hinzugeschlagen. Insgesamt stellt sich CLN im Vergleich als durchaus geeignet heraus um als Basis für ein symbolisches System zu dienen.

CLN eignet sich auch für numerische Berechnungen in Gleitkommazahlen mit beliebiger Genauigkeit, falls die übliche Maschinendarstellung mit 53-Bit-Mantisse (C-Typ `double`, entsprechend FORTRAN-Typ `real*8`) partout nicht mehr ausreichend ist. Die Genauigkeit kann dynamisch festgesetzt werden – eine Neukompilierung des Programmes wie bei manchen anderen Paketen ist nicht erforderlich. Sämtliche einfach transzendenten Funktionen sind schon vorhanden, während doppelt transzendente wie der Dilogarithmus[5] noch fehlen. Die implementierten Funktionen sind bemerkenswert schnell. Dies liegt an einem „binary splitting" [HaPa 1998] genannten Verfahren, in dem die Aufsummierung einer Reihenentwicklung so umgeordnet wird, dass ein Teil der Komplexität in die Multiplikation großer Zahlen aus $\mathbb{Q}$ verlagert wird. Diese kann aber schneller als $\mathcal{O}(N^2)$ bewerkstelligt werden und kürzt so die Reihensummierung ab. Erst im letzten Schritt wird dann wieder in die Fließkommadarstellung zurückumgewandelt.

Nun gibt es außer CLN noch weitere Bibliotheken für beliebige Genauigkeit, die als „*foundation class*" für GiNaCs Zahlen in Frage kämen. Victor Shoups NTL (Number Theory Library) [Shou 2000], David Baileys MPFun, Arjen Lenstras LIP, Mikko Tommilas Apfloat [Tomm 2001] und nicht zuletzt die GMP-Bibliothek selbst. Es gibt eine Reihe von wichtigen Eigenschaften, die außer CLN jedoch keines dieser Pakete bietet und die CLN für Computeralgebra prädestinieren:

Kleine ganze Zahlen sind unmittelbar:
Zwar kommen Koeffizienten $\gtrsim 2^{32}$ bei der Manipulation von Polynomen über $\mathbb{Z}$ nicht

[5] Dieser ist für beliebige Genauigkeit derzeit provisorisch in GiNaC implementiert.

selten vor, die meisten Koeffizienten bleiben jedoch klein. Es ist nun aber Verschwendung, auf dem Heap Objekte anzulegen, die lediglich eine maschinendarstellbare ganze Zahl repräsentieren, wenn man bedenkt, dass eine solche Zeigerdereferenzierung auf allen modernen Architekturen 5-10 Taktzyklen dauern kann und die Speicherallozierung als solche über den Systemaufruf `malloc()` selbst noch einmal zwischen 40 und 200 Zyklen, je nach Betriebssystem. In CLN ist die Darstellung der Basisklasse `cl_number` daher eine C-`union`, die entweder einen Zeiger auf den Anfang des eigentlichen Objekts darstellt oder eine unmittelbare ganze Zahl. Die für diese Unterscheidung notwendige Logik kann ohne zusätzliche Funktionsaufrufe implementiert werden (dies geschieht in der Header-Datei `cln/object.h`), wenn man ausnutzt, dass die Anfangsadressen vom System allozierter Speicherbereiche (das „*alignment*“) nicht beliebig sind, sondern je nach System Vielfache von zwei, vier oder acht sind. In diesen Adressen sind die am wenigsten signifikanten Bits immer Null und damit redundant. Sie werden daher zur Markierung unmittelbarer Zahlen herangezogen. Der Sprung bei $n = 16$ in Abbildung 4.3 erklärt sich aus dem dadurch eingeführten Overhead, da die Bitlänge des Ergebnisses der Multiplikation etwa die Summe der Bitlängen der Operanden ist.

Algebraische Körpereinbettungen werden honoriert:
Die Einbettung der natürlichen Zahlen in die rationalen Zahlen und der reellen Zahlen in die komplexe Zahlenebene wird berücksichtigt. Konstruiert man zum Beispiel eine rationale Zahl aus ganzzahligem Zähler und Nenner, so wird automatisch der größte gemeinsame Teiler gekürzt. Ist hiernach der Nenner 1, so wird das Ergebnis nicht als rationale Zahl dargestellt, sondern sofort in eine ganze Zahl verwandelt. Analog werden komplexe Zahlen, wenn der Imaginärteil exakt (also nicht als Gleitkommazahl) verschwindet, in reelle Zahlen umgewandelt. Es werden also genau diejenigen von CASen gewohnten Umformungen auf Zahlentypen durchgeführt, die gemeinhin als Typen-Retraktion bekannt sind.

Transparente Speicherverwaltung:
Die Speicherverwaltung von CLN basiert, wie diejenige von GiNaC, auf Referenzzählung. Sie ist daher auch nicht unterbrechbar und für den Programmierer völlig transparent. Beliebig viele korrekt implementierte Speicherverwaltungen auf Basis von Referenzzählung können in einem einzigen Programm koexistieren ohne miteinander zu interferieren. Dies mag zwar trivial klingen, ist aber für manche andere Systeme tatsächlich ein Problem. MuPAD zum Beispiel benutzt die Numerik von PARI [Coh 2000], einer in reinem C geschriebenen Bibliothek, bei der man sorgfältig Stackgrößen definieren muss. Es kommt vor, dass der Benutzer mit unabgefangenen Pari-Fehlermeldungen konfrontiert wird – worauf das Weiterarbeiten hoffnungslos wird. Im Falle von Magma [BCP 1997] gibt es Interferenzen in der Speicherverwaltung, diesmal mit dem System KANT V4 [Pohs 1996], die bis zu Systemabstürzen führen können.

GiNaC ist ferner bemüht, die Anzahl individueller Objekte, die dieselbe Zahl repräsentieren, gering zu halten. Der Grund ist weniger Speicherersparnis als ein Geschwindigkeitsvorteil: Die Methode `ex::compare()` kann anhand der Zeiger sofort Gleichheit feststellen und 0 zurückliefern (siehe auch Fußnote auf Seite 69). Idealerweise lässt man hierzu die Objekte vom Typ `numeric` von einer Flyweight-Fabrik erzeugen, zumindest für die häufig gebrauchten ganzen Zahlen mit Betrag kleiner als ein vorgegebener Schwellenwert. Dies stößt jedoch auf ein Hindernis bei der Implementierung: Eine solche Fabrik soll anstelle immer neuer Objekte Zei-

ger auf bereits erzeugte Objekte zurückliefern. Sie kann nur in Konstruktoren der Klasse `ex` eingebaut werden. Dem Benutzer steht aber immer die Möglichkeit offen, selbst ein Objekt vom Typ `numeric` auf dem Stack oder dem Heap zu erzeugen und dieses dann von einem `ex` referenzieren zu lassen, wodurch die Flyweight-Fabrik umgangen wird. Letztendlich sind die Probleme dieselben wie die auf Seite 70 bei der Fusion genannten: die „*Bridge*" abstrahiert die Klasse `ex` nicht vollständig von den von `basic` abgeleiteten Klassen weg.

4.5. Pseudofunktionen

Die Klasse `function` implementiert Pseudofunktionen. Pseudofunktionen unterscheiden sich von gewöhnlichen Funktionen dadurch, dass sie keine Abbildungen, etwa $f : \mathbb{C} \to \mathbb{C}$ implementieren, sondern als symbolische Ausdrücke unevaluiert stehen bleiben dürfen, wenn keine Vereinfachungen bekannt sind. So wird für ein Symbol `x` aus `sin(x)` wieder `sin(x)`, andererseits aus `sin(0)` 0. Die Pseudofunktionen müssen also selbst Objekte einer von `basic` abgeleiteten Klasse sein, die von `ex` verwaltet werden können. In dem Fall, dass eine Evaluierung nicht möglich ist, muss von der entsprechenden `eval()`-Funktion das in der Basisklasse `basic` dafür vorgesehene Bit `evaluated` gesetzt werden, um nicht eine Endlosrekursion zu erzeugen.

Von den verschiedenen Möglichkeiten der Implementierung wurde eine flache Struktur ausgewählt, in der alle Pseudofunktionen Objekte der Klasse `function` sind. Es erschien als zu aufwändig für jede mathematische Funktion eine von einer Basisklasse abgeleitete Klasse einzuführen. Der Implementator der Funktion müsste dann intime Kenntnisse von GiNaC haben um die Methoden `.destroy()`, `.copy()`, `.compare_same_type()` für das Speichermanagement sowie drei weitere Hilfsmethoden zum Archivieren zu schreiben. Zudem „erben" mathematische Funktionen nur selten Eigenschaften voneinander im Sinne der objektorientierten Programmierung, so dass man davon wenig Gebrauch machen könnte.

Pseudofunktionen werden zwecks ⇒ Kanonisierung unterschieden durch ihre Argumentenliste sowie eine Seriennummer, die die Art der Funktion spezifiziert: Die Objekte `sin(x)` und `cos(x)` unterscheiden sich lediglich durch diese Seriennummer. Niedergeschrieben in C++-Programmen werden sie als handele es sich um gewöhnliche Funktionen: die Hilfsfunktion `function sin(const ex&);` konstruiert das entsprechende `function`-Objekt in für den Programmierer transparenter Weise.[6]

Den Objekten mit einer bestimmten Seriennummer werden Hilfsfunktionen zugeordnet wie Ausgabeformate, Evaluationsalgorithmen, Differentiationsregeln und Reihenentwicklungsregeln. Das geschieht wie in Abbildung 4.4 skizziert mithilfe eines Registers, in denen Print-Namen für die Ausgabe sowie Zeiger auf die kompilierten Hilfsfunktionen gespeichert werden. Man kann sich diese Hilfsfunktionen als *event handler* vorstellen. Beim Aufruf von Methoden wie `.eval()` oder `.evalf()` wird die entsprechende Funktion anhand der Seriennummer nachgeschlagen und ausgeführt. Dieses Schema ist nicht gerade elegant, entspricht es doch in etwa einer selbstgemachten `vtable`. Es ist ein Kompromiss, der einen ausreichend flexiblen Umgang

[6] Die Option, Pseudofunktionen durch Funktoren erzeugen zu lassen, wurde auch untersucht. Als Objekte kollidieren solcherlei Funktoren aber sehr leicht mit anderweitig deklarierten Funktionen: Es ist nicht möglich innerhalb desselben Namespaces eine Funktion und ein Objekt namens `sin` zu haben.

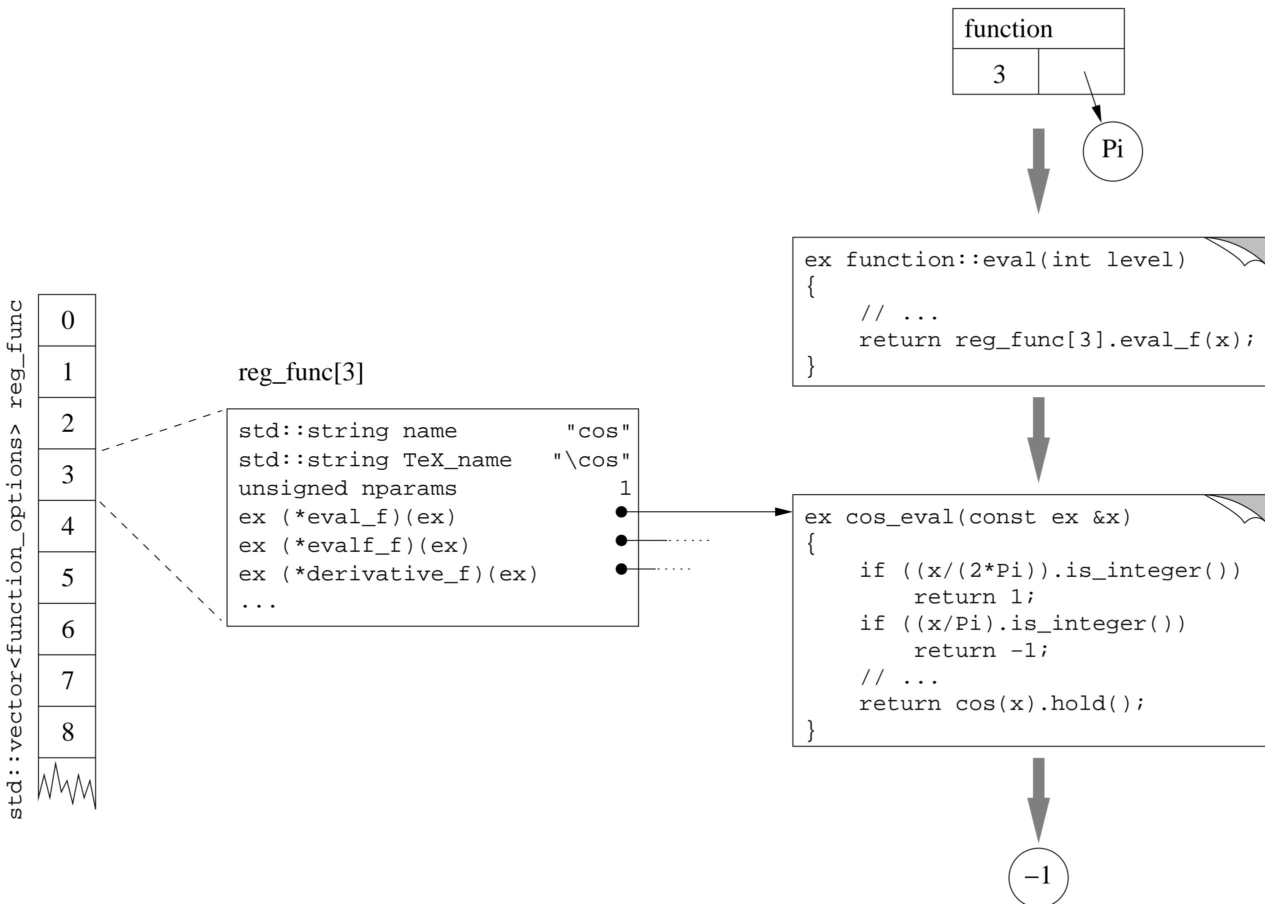


Abbildung 4.4.: *Methodenaufruf bei Pseudofunktionen (leicht vereinfacht) im Falle der automatischen Evaluation von* $\cos(\pi) \to -1$. *Die Methode* `function::eval()` *schlägt das zur Seriennummer 3 gehörende* `function_options` *Objekt in einer statischen Tabelle nach. Darin findet es den Zeiger auf die zum Cosinus gehörende Evaluierungsfunktion.*

mit symbolischen mathematischen Funktionen erlaubt und die Anzahl der pro Funktion zu schreibenden Zeilen auf einem Minimum hält.

4.6. Laurentreihen: die Klasse ‚pseries'

Nichtabbrechende Taylor- und – allgemeiner – Laurent-Reihen werden in einer eigenen Klasse namens `pseries` („*power series*") repräsentiert. Es gibt zwei grundsätzliche Ansätze, solche Reihen im Computer darzustellen. Der naheliegendste ist, die Koeffizienten bis zu einer vorgegebenen Ordnung zu entwickeln und abzuspeichern. Wenn die Folge durch Ableitung des Ausgangsausdrucks[7] entstanden ist, so schreiben wir für die n-te Ableitung ausgewertet an der Stelle a anstatt $f^{(n)}(x)|_{x=a}$ kurz $f^{(n)}(a)$. Die Taylor-Reihe hat dann die Darstellung

$$f(x)|_{x=a} = f(a) + f^{(1)}(a) + \tfrac{1}{2!}f^{(2)}(a) + \mathcal{O}(x^3)$$

[7] Wir werden noch sehen, dass nicht alle auftretenden Taylor-Reihen durch Ableitung erzeugt werden können.

wobei für den Abbruch $\mathcal{O}(x^3)$ eine geeignete Darstellung gefunden werden muss (in GiNaC ist es eine dafür vorgesehene Pseudofunktion). Ein alternativer Ansatz der Darstellung besteht darin, die Reihe nur formal als „zu Taylor-Entwickeln“ zu markieren und die Berechnung der Koeffizienten erst dann durchzuführen, wenn sie von einem anderen Teil des Programmes gebraucht werden. Diese Technik bezeichnet man als *lazy evaluation*. Sie ist besonders geeignet, wenn nicht von vornherein feststeht, bis zu welcher Ordnung eine Reihe entwickelt werden soll, und sie hat Geschwindigkeitsvorteile, weil niemals mehr berechnet wird als unbedingt nötig ist. Allerdings verkompliziert sie Operationen zwischen mehreren solchen Reihen, wenn man nicht eine Verschwendung von Speicherplatz in Kauf nehmen will. Um dies einzusehen betrachten wir die Multiplikation zweier Reihen $f(x)|_{x=a}$ und $g(x)|_{x=a}$. Nehmen wir an, beide Reihen sind bis zur Ordnung $n = 2$ vorentwickelt, die Koeffizienten $f(a)$, $g(a)$, $f^{(1)}(a)$, $g^{(1)}(a)$, $f^{(2)}(a)$ sowie $g^{(2)}(a)$ also berechnet. Ferner müssen jeweils ein uneingesetztes $f^{(2)}(x)$ und $g^{(2)}(x)$ aufgehoben werden, um die Reihen später bei Bedarf weiterentwickeln zu können. Das Produkt $(f \cdot g)(x)|_{x=a}$ ist leicht berechnet

$$\begin{aligned}(f \cdot g)(x)|_{x=a} \;=\;& f(a)g(a) + \\ & \big(f^{(1)}(a)g(a) + f(a)g^{(1)}(a)\big) + \\ & \tfrac{1}{2!}\big(f^{(2)}(a)g(a) + 2f^{(1)}(a)g^{(1)}(a) + f(a)g^{(2)}(a)\big) + \mathcal{O}(x^3) \qquad (4.3)\end{aligned}$$

und in einem neuen Reihenobjekt gespeichert. Allerdings sollte jetzt auch wieder der Term $(f \cdot g)^{(2)}(x)$ mit der Reihe gespeichert werden, um später neue Terme nachgenerieren zu können. Dies ist aber nicht mehr möglich, da er $\frac{1}{2!}(f^{(2)}(x)g(x) + 2f^{(1)}(x)g^{(1)}(x) + f(x)g^{(2)}(x))$ lautet, aber die Terme $f(x)$, $g(x)$, $f^{(1)}(x)$ und $g^{(1)}(x)$ gar nicht mehr vorliegen. Lediglich $f^{(2)}(x)$ und $g^{(2)}(x)$ wurden aufgehoben. Terme der Produktreihe können also nicht mehr nachgeneriert werden. Als Ausweg könnte man statt des letzten abgeleiteten Terms den unabgeleiteten Ausgangsausdruck aufheben. Die Nachgenerierung von Termen ist dann aber ebenso aufwändig wie die gesamte Neuberechnung der Reihe. Alternativ könnte man alle Terme ohne die Einsetzung $|_{x=a}$ aufheben. Dies ist jedoch problematisch, da diese Terme sehr häufig groß sind und erst durch die Einsetzung kollabieren. Eine Vervielfachung des von `pseries`-Objekten belegten Speichers erscheint kaum angemessen.

Dass überhaupt eine eigene Klasse benötigt wird anstatt mit Polynomen zu rechnen liegt daran, dass man dadurch, dass eine Reihe in x bei einer Ordnung n abbrechend dargestellt wird, effektiv modulo x^n rechnet: das Inverse einer Reihe $a_0 + a_1x + a_2x^2 + \mathcal{O}(x^3)$ ist wieder eine Reihe $c_0 + c_1x + c_2x^2 + \mathcal{O}(x^3)$ anstatt ein Element aus einem Quotientenkörper über Polynomen. Dies ermöglicht eine Anzahl beachtlicher Vereinfachungen. So ist zum Beispiel die Exponentiation von Reihen weniger aufwändig als das Potenzieren von Polynomen, wenn man ein von Leonhard Euler gefundenes Verfahren implementiert [Knu 1998, GCL 1992]. Sei $A(x) = a_0 + a_1x + a_2x^2 \ldots$ eine Taylor-Reihe. Wir wollen $C(x) = A(x)^p = c_0 + c_1x + c_2x^2 \ldots$ berechnen. Wir leiten hierzu $C(x) = A(x)^p$ nach x ab, multiplizieren beide Seiten mit $A(x)$ und erhalten so

$$\begin{aligned}C'(x)A(x) &= p\,A(x)^{p-1}A'(x)A(x) \\ &= p\,C(x)A'(x).\end{aligned}$$

Durch Ausmultiplizieren und Koeffizientenvergleich gelangt man so zu der Rekursionsformel für die c_i

$$c_i = \big(i\,p\,a_i\,c_0 + ((i-1)p-1)\,a_{i-1}\,c_1 + \cdots + (p-(i-1))\,a_1\,c_{i-1}\big)\big/(a_0\,i), \qquad (4.4)$$

die sich lösen lässt mit dem Startwert $c_0 = {a_0}^p$. Sie lässt sich auch erweitern auf den Fall, dass der führende Koeffizient von $A(x)$ nicht konstant ist. Hierzu multipliziert man $A(x)$ mit x^m und wiederholt die Ableitung der Rekursionsformel. Der führende Koeffizient von $C(x)$ ist dann ${a_0}^p x^{m\,p}$ und man findet, dass (4.4) immer noch gültig ist.

Dies ist nicht nur effizient, sondern auch verallgemeinerbar. Lediglich dass p unabhängig von x ist, wurde ausgenutzt: diese Rekursionsformel bewahrt genauso für gebrochenrationale Exponenten p ihre Gültigkeit oder für transzendente Exponenten wie π und sie lässt sich sogar auf Puiseux-Reihen mit gebrochenrationalen Exponenten des Entwicklungsparameters verallgemeinern (siehe Kasten auf Seite 92). Prinzipiell ist sie sogar für symbolische Exponenten p anwendbar. Man vergleiche dies mit den beschränkten Möglichkeiten bei der Potenzierung von univariaten Polynomen $P(x)^p$.

Betrachten wir Reihenentwicklung als Beispiel der auf Seite 74 diskutierten Methodenfortpflanzung. Reihenentwicklung kann sehr elegant programmiert werden als Methode, die bottom-up durch den Darstellungsbaum läuft.[8] GiNaC implementiert Reihenentwicklung für alle eingebauten Funktionen.[9] Die zugrundeliegende Datenstruktur (Klasse `pseries`) wurde schon in [Baue 2000] beschrieben, ebenso die Rechenoperationen wie Addition, Multiplikation etc. darauf. Hier sollen ein paar häufige Probleme mit Spezialfällen genannt und Lösungen diskutiert werden.

Für Ableitungen reicht es wie gesagt aus, den Darstellungsbaum von der Wurzel beginnend zu durchschreiten. Ist das auch für Reihenentwicklung der Fall, die ja in der Regel auf Ableitungen zurückgreift? Abgesehen von verallgemeinerten Reihen wie Puiseux-Reihen, die im Rahmen von GiNaC nicht benötigt werden, findet man drei Arten von Ausnahmen, in denen dies nicht geht:

1) Die Ableitungen können aus irgendeinem Grunde am Entwicklungspunkt nicht ausgewertet werden (z.B. beim Dilogarithmus ${\rm Li_2}'(0)$)

2) Es muss die Entwicklung an einem Verzweigungspunkt oder auf einem Schnitt berechnet werden (z.B. Logarithmus $\log(0)$)

3) Es liegt ein Pol vor (z.B. Gamma-Funktion $\Gamma(-n), n \in \{0, 1, 2, \dots\}$)

Außerhalb dieser Fälle wird einfach auf Taylor-Reihenentwicklung zurückgegriffen (was natürlich die Kenntnis der Ableitungen voraussetzt). Dies geschieht, indem eine Exception an die aufrufende Routine zurückgeworfen wird, welche die Implementierung der Taylor-Reihe via Ableitung in `basic::series()` aufruft. Die eingeschlagenen Lösungswege für die drei Klassen

[8] Dennoch hatten bis auf Maple alle der in [West 1995] getesteten CAS erhebliche Schwierigkeiten mit der Taylorentwicklung $e^{-x}\sin(x) \simeq x - x^2 + \frac{1}{3}x^3 - \frac{1}{30}x^5 + \mathcal{O}(x^6)$ an der Stelle $x = 0$, was für GiNaC nie ein Problem darstellte. Da wir keinen Einblick in die Sourcen jener Systeme haben, kann man nur soviel vermuten, dass dort entweder völlig andere Ansätze verwendet wurden, oder dass Produkte von Folgen unimplementiert waren.

[9] Einzige Ausnahme sind verallgemeinerte Reihen, in denen die Koeffizienten von x abhängen, aber subpolynomial anwachsen, die bisher noch nicht implementiert worden sind.

Eine Option: Puiseux-Reihen

Die derzeit von GiNaC darstellbaren Reihen haben alle ganzzahlige Exponenten, es können stets nur Taylor- oder Laurent-Reihen dargestellt werden. Die Exponenten sind zwar in ihrer Implementierung vom allgemeinem Typ `ex`*, aber nur ganze Zahlen kommen vor. Es handelt sich hierbei um eine Invariante der Klasse* `pseries`*. Puiseux-Reihen wie*

$$\log(1+\sqrt{x})\big|_{x=0} = x^{\frac{1}{2}} - \tfrac{1}{2}x + \tfrac{1}{3}x^{\frac{3}{2}} - \tfrac{1}{4}x^2 + \tfrac{1}{5}x^{\frac{5}{2}} + \mathcal{O}(x^3)$$

wurden bisher nicht implementiert in der Annahme, dass sie in Schleifenrechnungen nicht vorkommen, sind doch die in dimensionaler Regularisierung auftretenden Reihen in ε stets Laurent-Reihen. Diese Hoffnung war wohl etwas zu optimistisch. Dennoch konnten bisher auftretende Fälle transformiert werden auf Laurentreihen – durch $x \to x^2$ um in unserem Beispiel zu bleiben. Falls dies einmal nicht mehr ausreichen sollte, muss `pseries` *neu gestaltet werden. Insbesondere der Datentyp, der mit der Methode* `.degree()` *den Grad und mit* `.ldegree()` *den Grad des führenden Koeffizienten zurückgibt, muss von* `int` *zu einem Typ geändert werden, welcher rationale Zahlen darstellen kann. Addition und Multiplikation von Puiseux-Reihen sind weitgehend analog zu derjenigen für Laurent-Reihen. Wir skizzieren nun, wie sich auch skalare Exponentiation verallgemeinern lässt.*

Sei $A(x) = a_0 + a_1x^{q_1} + a_2x^{q_2} + \cdots + \mathcal{O}(x^n)$ und $C(x) = A(x)^p = c_0 + c_1x^{r_1} + c_2x^{r_2} + \cdots + \mathcal{O}(x^{n\,p})$ mit $q_i, r_i \in \mathbb{Q}$. Wir berechnen zunächst das kleinste gemeinsame Vielfache der Nenner aller q_i, $Q = \mathrm{kgV}(q_1, q_2, \dots)$. Dann ersetzen wir $x \to x' = x^Q$:

$$C(x^Q) \equiv A(x^Q)^p$$

Damit haben wir das Problem auf die gewöhnliche skalare Exponentiation zurückgeführt, denn die Exponenten von x' in $A(x')$ sind alle ganzzahlig. Durch Anwendung des Algorithmus für skalare Exponentiation findet man die gesuchten Koeffizienten c_i von $C(x')$ und kann die Rückersetzung $x' \to x = x^{1/Q}$ vornehmen um die gesuchte Reihe $C(x)$ zu erzeugen. Auch dieses Verfahren verallgemeinert in offensichtlicher Weise auf den Fall, dass der führende Koeffizient von $A(x)$ nicht konstant ist.

von Ausnahmen werden im Folgenden anhand der genannten Beispiele vorgestellt. In Zukunft notwendig werdende Implementierungen dieser Art sollten damit keine grundsätzlichen Probleme mehr bereiten.

Fall **1**) Ableitungen können nicht ausgewertet werden:
Die Reihenentwicklung des Dilogarithmus um den Nullpunkt ist bekanntlich

$$\mathrm{Li}_2(x)|_{x=0} = x + \frac{1}{4}x^2 + \frac{1}{9}x^3 + \frac{1}{16}x^4 + \frac{1}{25}x^5 + \mathcal{O}(x^6),$$

woraus man abliest, dass die n-te Ableitung am Ursprung durch

$${\mathrm{Li}_2}^{(n)}(x)|_{x=0} = \frac{n!}{n^2}$$

gegeben sein muss. Rechnet man es jedoch stur aus, so findet man

$${\mathrm{Li}_2}'(x)|_{x=0} = -\frac{\log(1-x)}{x}\bigg|_{x=0},$$

was eine Division durch Null nach sich zieht, wenn man $x \equiv 0$ einsetzt. Zukünftige Versionen von GiNaC könnten das Problem eventuell lösen, indem sie Grenzwerte benutzen – der vorliegende Fall ist schon mit der Regel von l'Hôpital lösbar. Die derzeitige Notlösung besteht in Abwesenheit von Grenzwerten aus einem vielleicht etwas grob anmutenden Trick. Wir wissen ja, was die Reihenentwicklung ist, also konstruieren wir einfach ‚per Hand' eine solche Folge und geben sie zurück. Da aber nicht immer im Argument direkt entwickelt wird, sondern im Allgemeinen eine Entwicklung der Form $\mathrm{Li}_2(f(x))|_{x=p}$ mit $f(p) = 0$ verlangt wird, kaskadieren wir die Reihenentwicklung. Zunächst wird $\mathrm{Li}_2(s)|_{s=0} = s + \frac{1}{4}s^2 + \frac{1}{9}s^3 + \ldots$ in einem Hilfssymbol s erzeugt, dann die Taylor-Reihe des Arguments der Funktion $f(x)|_{x=p}$ für s substituiert. Fasst man Terme gleicher Ordnung zusammen, so ergibt dies die gewünschte Taylorreihe.

```
1  static ex Li2_series(const ex & x,           // argument of Li2
2                       const relational & rel, // expansion variable and point
3                       int ord)                // order of expansion
4  {
5      const ex x_at_pt = x.subs(rel);
6      if (!x_pt.is_zero())
7          throw do_taylor();  // caught by function::series()
8      if (x_pt.is_zero()) {
9          const symbol s;
10         ex ser;
11         // construct manually the primitive expansion
12         for (int i=1; i<order; ++i)
13             ser += pow(s,i)/pow(i,2);
14         // substitute the argument's series expansion
15         ser = ser.subs(s==x.series(rel,order));
16         // maybe that series is terminating, so add a proper order term
17         epvector nseq;
18         nseq.push_back(expair(Order(1), order));
19         ser += pseries(rel, nseq);
20         // reexpansion will collapse the series again
21         return ser.series(rel.order);
22     }
23     // missing: treat other pathological cases...
24 }
```

Die Methode hat einen Haken: Zwar lassen sich damit korrekt vorgeschaltete Funktionen taylorentwickeln, wie z.B. $\mathrm{Li}_2(\sin(x))_{x=0} = x + \frac{1}{4}x^2 - \frac{1}{18}x^3 + \mathcal{O}(x^4)$, nicht jedoch nachgeschaltete, wie $\sin(\mathrm{Li}_2(x))_{x=0}$. Dann nämlich tritt das Problem, welches wir gerade gelöst haben, wieder auf: Die Ableitung von $\sin(\mathrm{Li}_2(x))$ kann nicht an der Stelle $x = 0$ ausgewertet werden, da der Sinus nichts vom Dilogarithmus weiß. Sollte dies einmal ein echtes Problem darstellen, so müssen entweder Grenzwerte implementiert oder das Design der Klasse `pseries` erheblich verändert werden.

Fall **2**) Ein Verzweigungspunkt/Schnitt liegt vor:
Auf diesen wichtigen Fall muss wieder durch Einsetzen des Auswertepunktes in das Argument getestet werden. Meist kann man die Verzweigungspunkte auf solche anderer Funktionen zurückführen (siehe Kasten), etwa auf denjenigen des Logarithmus, der dann nicht weiter in Form von Laurentreihen ausgewertet wird. Dies geht bei allen transzendenten und hyperbolischen Funktionen, auch bei doppelt transzendenten wie zum Beispiel beim Dilogarithmus

Verzweigungspunkte = Problempunkte?

Viele Computeralgebrasysteme erlauben die Laurententwicklung von Funktionen an Verzweigungspunkten. Hierbei ist jedoch wichtig, dass die Lage des Schnittes korrekt wiedergegeben wird, wobei Korrektheit anhand der Konsistenz mit der numerischen Evaluation zu verstehen ist. Über die kanonische Lage der Schnitte einiger häufiger Funktionen informiert Tabelle A.1 auf Seite 149. Im Falle der trigonometrischen Funktionen kann die Entwicklung – wie in jener Tabelle angegeben – zurückgeführt werden auf die inerte Entwicklung des natürlichen Logarithmus am Verzweigungspunkt: $\log(x)|_{x=0} = \log(x)$*. Eigenartigerweise werden einige Systeme hier aber inkonsistent. Als Beispiel kann man* $\operatorname{atanh}(x)$ *um* $x = 1$ *entwickeln:*

$$\operatorname{atanh}(x)_{x=1} = \tfrac{1}{2}\big(\log(2) - \log(1-x)\big) + \tfrac{1}{4}\big(x-1\big) + \mathcal{O}\big((x-1)^2\big)$$

Hierin wird der Schnitt bestimmt von $\log(1-x)$*, verläuft also entlang der positiven reellen Achse beginnend bei* $x = 1$ *in Übereinstimmung mit Tabelle A.1. Man kann leicht verifizieren, dass sich dies mit den in diesem Abschnitt vorgestellten Methoden von selbst ergibt, wenn man* $\log(ax)|_{x=0}$ *nicht automatisch zu* $\log(a) + \log(x)$ *entwickelt. Das ist ja auch in der Tat falsch für nichtpositive Werte von* a*. Diese leichtfertige Entwicklung von* $\log(ax)|_{x=0}$ *wird aber beispielsweise von MapleV und Mathematica durchgeführt. Bei allen transzendenten Funktionen führt das zu Problemen: Im Falle von Mathematica sind die Schnitte entweder um* $\pi/2$ *in der komplexen Ebene gedreht oder es treten zusätzliche Konstanten wie* $i\pi = \log(-1)$ *im Ergebnis auf. Bei MapleV passiert dies nicht, jedoch nur aufgrund expliziter Korrektur in allen einzelnen Funktionen, die für die Reihenentwicklung zuständig sind. Dies ist der Grund dafür, dass in GiNaC seit Version 0.7.1 die Entwicklung von* $\log(ax)|_{x=0}$ *völlig inert ist.*

(siehe Anhang A). Dessen Verzweigungspunkt bei $x = 1$ hat beispielsweise die Entwicklung

$$\mathrm{Li}_2(x)|_{x=1} = \tfrac{1}{6}\pi^2 + \big(1 - i\pi - \log(x-1)\big)(x-1) + \mathcal{O}\big((x-1)^2\big),$$

die speziell in `Li2_series()` kodiert werden muss. Auf Schnitten abseits von Verzweigungspunkten kann entweder die stetige analytische Fortsetzung gewählt oder aber der Schnitt durch eine θ-Funktion ausgedrückt werden. Der Benutzer entscheidet hierüber mit einem zusätzlichen Parameter beim Aufruf der Reihenentwicklung. Die stetige Fortsetzung kann wie im gewöhnlichen Fall mittels Ableitung berechnet werden. Die Darstellung des Schnittes (siehe Tabelle A.1) muss wieder separat kodiert werden, um beispielsweise die Darstellung

$$\log(x)_{x=-1} = i\pi(1 - 2\theta(ix)) - (x + 1) - \tfrac{1}{2}(x + 1)^2 + \mathcal{O}\big((x + 1)^3\big)$$

zu erzeugen. Anstelle der θ-Funktion wird in GiNaC allerdings stets deren komplexe Fortsetzung `csgn()` mit Unstetigkeit auf der imaginären Achse herangezogen. Diese Sonderbehandlung an Schnitten muss für jede Pseudofunktion separat geschehen.

Fall **3**) Eine Polstelle liegt vor:

An Polstellen werden üblicherweise Rekursionsformeln, Reflexionsformeln und so weiter angewendet. Das Problem wird somit in eine Form gebracht, in der eine rationale Funktion in eine Laurent-Reihe entwickelt wird. Die Gamma-Funktion beispielsweise gehorcht der Rekursionsformel

$$\Gamma(x) = \frac{1}{x}\Gamma(x+1). \tag{4.5}$$

Wird die Entwicklung am Nullpunkt verlangt, so kann man statt $\Gamma(\varepsilon)|_{\varepsilon=0}$ natürlich $\frac{1}{\varepsilon}\Gamma(1+\varepsilon)|_{\varepsilon=0}$ entwickeln. Allgemein lässt sich so jeder Pol bei $-m$ zunächst „wegschieben":

$$\Gamma(x)|_{x=-m} = \frac{\Gamma(x+m+1)}{x(x+1)\cdots(x+m)}\Big|_{x=-m}. \quad (4.6)$$

Eine Routine, die in GiNaC die Laurententwicklung einer bestimmten mathematischen Funktion übernimmt, muss nur dann implementiert werden, wenn gewöhnliche Taylorentwicklung versagt. Dies kann freilich zwei verschiedene Gründe haben: 1) die Ableitungen sind nicht bekannt und 2) es existieren Polstellen irgendwo in der komplexen Ebene. Gehen wir von dem Fall aus, dass die Ableitungen alle bekannt sind (oder auch nur formal definiert), so ist also für die Gamma-Funktion[10] eine Rekursionsformel zu implementieren, die wie in (4.6) auf den Taylorfall zurückgreift. Dieser braucht dann nicht mehr programmiert zu werden, denn einfache Taylorentwicklung ist schon in `basic::series()` vorhanden. Man verwirklicht das in C++ am elegantesten, indem man eine Exception zurückwirft an die aufrufende Routine:

```
1  static ex tgamma_series(const ex & x,          // argument of tgamma
2                          const relational & rel, // expansion variable and point
3                          int ord)                // order of expansion
4  {
5      const ex x_at_pt = x.subs(r);
6      if (!x_at_pt.info(info_flags::integer) || x_at_pt.info(info_flags::positive))
7          throw do_taylor();   // caught by function::series()
8      // if we got here we have to care for a simple pole at -m:
9      numeric m = -ex_to_numeric(x_at_pt);
10     ex ser_denom = 1;
11     for (numeric p; p<=m; ++p)
12         ser_denom *= x+p;
13     return (tgamma(x+m+1)/ser_denom).series(s, pt, ord+1);
14 }
```

Das Programmbeispiel zeigt noch eine kleine Subtilität, die leicht zu Verwirrung führt. Was man wirklich berechnen will ist normalerweise nicht die Entwicklung von $\Gamma(x)|_{x=z}$ sondern $\Gamma(f(x))|_{x=z}$. Daher muss die Variable `x_at_pt` eingeführt werden (Zeile *5*) und *diese* darauf getestet werden, ob sie in $\{0, -1, -2, \dots\}$ liegt. Danach wird der Nenner aus (4.6) konstruiert (Zeilen *11-12*), dann die Reihe gebildet und zurückgegeben (Zeile *13*). Man beachte, dass dieses Vorgehen auch einige triviale Fälle mitbehandelt, ohne dass dies explizit ausgedrückt werden muss: Ist beispielsweise `x` unabhängig von der Entwicklungsvariablen, so wird die Exception `do_taylor()` ebenfalls geworfen und nur der nullte Term in der Taylorentwicklung, also `tgamma(x)` ohne Ordnungsterm, zurückgegeben.

Gerade die Entwicklung der Gamma-Funktion in eine Laurentreihe, wie sie in der dimensionalen Regularisierung benötigt wird, macht die Implementierung einer ganzen Sammlung von Zusatzfunktionen erforderlich. Da dies in gewissem Maße ein abgeschlossenes Modul in GiNaC

[10] Die Gamma-Funktion $\Gamma(x)$ heißt in GiNaC `tgamma(x)` in Übereinstimmung mit [ISO 1999]. Der etwas umständliche Name wurde gewählt, um sie von `lgamma(x)`, dem natürlichen Logarithmus der Gamma-Funktion, zu unterscheiden. Die in anderen Standards wie [ATT 1986] definierte Funktion `gamma(x)` berechnete ebenfalls $\ln(\Gamma(x))$, was wohl der Grund dafür ist, dass sich diese Syntax nicht durchgesetzt hat und wir sie in GiNaC nicht verwenden.

darstellt, wird die Abhängigkeitskaskade im Folgenden skizziert. Die Polentwicklung wurde ja gerade auf die Taylorentwicklung am Punkt $x = 1$ zurückgeführt, was bedeutet, dass sämtliche Ableitungen an diesem Punkt bekannt sein müssen. Man definiert die Digamma-Funktion ψ als Ableitung des Logarithmus der Gamma-Funktion

$$\psi(x) = \frac{d}{dx}\log(\Gamma(x)) = \frac{\Gamma'(x)}{\Gamma(x)}$$

und die n-te Polygamma-Funktion ψ_n als n-te Ableitung der Digamma-Funktion für $x \in \mathbb{C}$:

$$\psi_n(x) = \frac{d^n}{dx^n}\psi(x).$$

(Es ist also $\psi_0(x) \equiv \psi(x)$.) Am Punkt $x = 1$ werden die Funktionswerte

$$\begin{aligned} \psi(1) &= -\gamma \\ \psi_n(1) &= (-)^{n+1} n!\zeta(n+1) \end{aligned}$$

angenommen [AbS 1972, GrRy 1994] wo $\zeta(x)$ die Riemann'sche ζ-Funktion ist, die also zumindest an den Punkten $\{1, 2, 3, \ldots\}$ ausgewertet werden sollte. Für ungeradzahlige Argumente ist dies nicht möglich, für geradzahlige kann man jedoch [IyKa 1980]

$$\zeta(m) = \frac{|B_m|}{(m)!}\pi^m 2^{m-1} \qquad \forall\, m \in \{2, 4, 6, \ldots\} \tag{4.7}$$

berechnen. Den letzten Schritt in dieser Kette stellt die Implementierung der Bernoulli-Zahlen B_m dar, die zum Beispiel rekursiv berechnet werden können:

$$B_0 = 1, \qquad B_m = \frac{-1}{m+1}\sum_{k=0}^{m-1}\binom{m+1}{k}B_k. \tag{4.8}$$

Die n-te Bernoulli-Zahl hängt also von allen vorherigen Bernoulli-Zahlen ab, was ihre Berechnung sehr aufwändig macht.[11]

[11] Bernoulli-Zahlen treten in den Entwicklungen aller inversen trigonometrischen Funktionen auf. Dort genügt jedoch eine Gleitkomma-Berechnung, wofür stabile Algorithmen bekannt sind. Hat man zum Beispiel – wie im Falle von CLN – effiziente Riemann'sche Zeta-Funktionen numerisch implementiert, so reicht es offensichtlich, den Spieß umzudrehen und Gleichung (4.7) zu invertieren.
Zur exakten Auswertung sei angemerkt, dass die Summe (4.8) einer „Divide and Conquer"-Strategie zugänglich ist, womit in der Praxis noch B_{50000} berechenbar ist. Da jedoch zusätzlich zu B_n häufig auch $B_m, m < n$ benötigt werden, ist ein Verfahren, welches eine vollständige Remember-Tabelle implementiert, vorzuziehen. Tatsächlich weisen alle gängigen CASe das Laufzeitverhalten von Remember-Tabellen auf. Am effizientesten ist z.B. der Umweg über Tangens-Polynome, welche durch die Rekursionsformel $T_n(x) = (1 + x^2)\,T_{n-1}(x)'$ mit $T_0(x) = x$ definiert sind. Die $n-1$-te Tangens-Zahl $T_{n-1} \equiv T_{n-1}(0)$ ist mit der Bernoulli-Zahl durch $T_{n-1} = (-)^{\frac{n}{2}-1}\frac{2^{2n}-2^n}{n}B_n$ verknüpft [GKP 1989]. Das n-te Tangens-Polynom ist von der Ordnung $n+1$ und besitzt lediglich ganzzahlige Koeffizienten, was einer effizienten iterativen Berechnung sehr entgegenkommt. GiNaC implementiert noch einen dritten Zugang, der zwar um die Hälfte langsamer ist, jedoch ohne das Abspeichern der Tangens-Polynome für nachfolgende Berechnungen auskommt. Es ist eine Variation des „Divide and Conquer"-Verfahrens, äquivalent zur Implementierung der Pari-Funktion `bernvec()`.

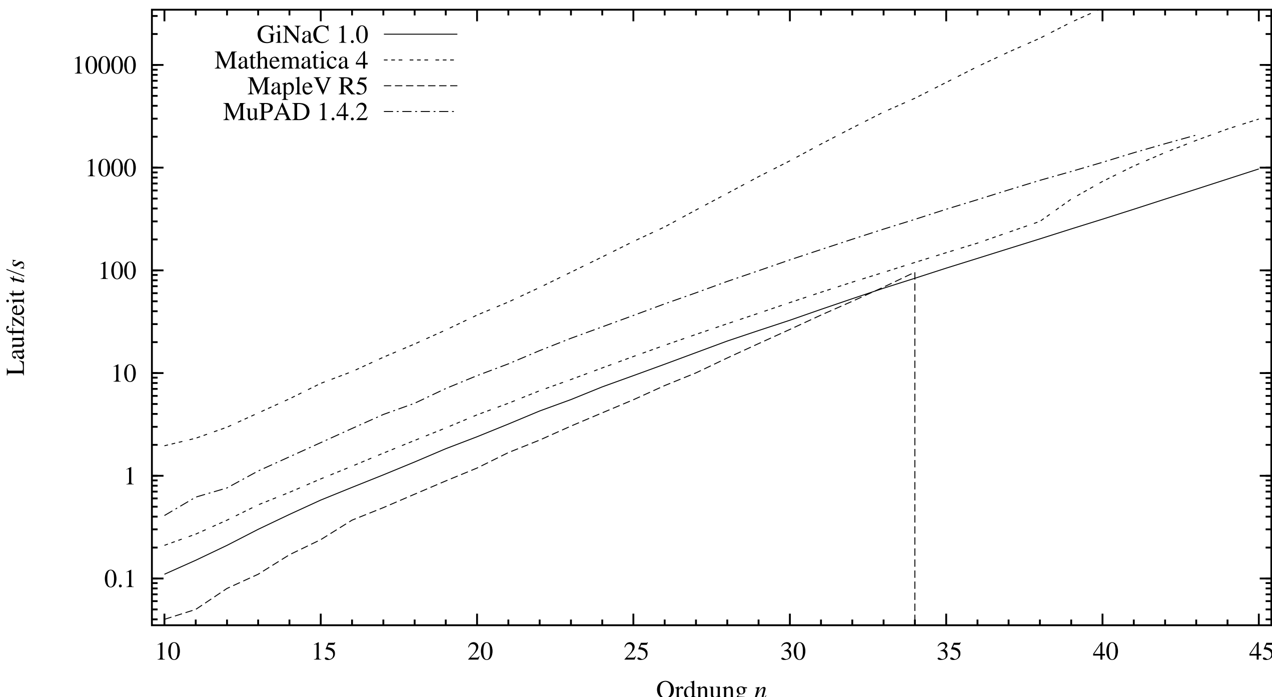


Abbildung 4.5.: *Laufzeiten für die symbolische Laurententwicklung von $\Gamma(x)|_{x=0}$ auf einer IA32-Architektur.*

Nun reicht die Auswertung von $\psi_n(1)$ zwar aus für die Laurententwicklung an den Polstellen der Gamma-Funktion, hinterlässt jedoch ein etwas unabgeschlossenes Bild: Zum einen ergibt die Taylorentwicklung von $\Gamma(2+\varepsilon)|_{\varepsilon=0}$ lediglich $1+\psi(2)\varepsilon+\frac{1}{2}(\psi_1(2)+\psi(2)^2)\varepsilon^2+\mathcal{O}(\varepsilon^3)$ und so weiter, da zwar $\Gamma(n)=(n-1)!$ evaluiert werden kann, jedoch über die Evaluation der Polygamma-Funktionen noch nichts gesagt ist. Zum anderen wird auch die Laurententwicklung der ψ-Funktionen an ihren Polstellen bisweilen benötigt und bedarf einer Implementierung. Beide Lücken können geschlossen werden, indem mithilfe der Rekursionsformeln

$$\begin{aligned}\psi(x+1) &= \psi(x)+x^{-1}\\ \psi_n(x+1) &= \psi_n(x)+(-1)^n n! x^{-n-1}\end{aligned}$$

die Funktionswerte für $x \in \mathbb{N}^+$ und die Pole der Ordnung 1 bzw. $n+1$ bei den übrigen $x \in \mathbb{Z}$ zurückgeführt werden auf Evaluationen am Punkt $x = 1$ – ganz analog zum Vorgehen in (4.6).

Es ist bemerkenswert, dass diese jegliches Optimierungspotenzial ignorierende Vorgehensweise tatsächlich schon zu einem Modul führt, dass sich durchaus mit Konkurrenzprodukten messen kann (Abbildung 4.5). Hier wurden auf einem mit 450MHz getakteten Intel P-III unter Linux die Laufzeiten für Laurententwicklung der Gammafunktion bis zu hohen Ordnungen gemessen. Maples supereffizienter Kernel bricht bei der Ordnung 35 mit der Fehlermeldung `object too large` ab, da dann offenbar Zwischenergebnisse mit Summen aus 2^{16}–1 Termen entstehen. Für den Vergleichslauf musste die angeforderte Ordnung im Falle von Mathematica und MuPAD um eins inkrementiert werden, um dieselben höchsten Koeffizienten zu erhalten wie Maple und GiNaC. Ferner wurden alle Systeme gezwungen, ihr Ergebnis bis zu Koeffizienten zu vereinfachen, die nur von $\zeta(m)$, π und der Euler-Konstanten γ abhängen. Im Fall Mathematica musste dafür noch ein zusätzliches `FunctionExpand[]` eingeschaltet werden um tatsächlich

ein Ergebnis der Form

$$\Gamma(x) = \frac{1}{x} - \gamma + \Big(\frac{\pi^2}{12} + \frac{\gamma^2}{2}\Big)x - \Big(\frac{\pi^2\gamma}{12} + \frac{\gamma^3}{6} + \frac{\zeta(3)}{3}\Big)x^2 + \mathcal{O}(x^3)$$

zu erhalten (ansonsten liefert Mathematica unausgewertete Polygamma-Funktionen $\psi_m(1)$ zurück). Gerade dieses `FunctionExpand[]` aber ist für den Großteil der Rechenzeit verantwortlich (obere Mathematica-Kurve in Abbildung 4.5). Lässt man es weg so ist Mathematica nur noch geringfügig langsamer als GiNaC (untere Kurve). Das Knie in dieser Kurve beginnend bei $n = 38$ wird uns weiter unten auf Seite 116 noch einmal begegnen.

Die in dimensionaler Regularisierung ebenfalls manchmal benötigte analytische Evaluation der hier betrachteten Funktionen für alle halbzahligen Argumente wird idealerweise zurückgeführt auf ganzzahlige Argumente. Hierzu werden nur die Verdopplungsformeln

$$\begin{aligned}
\Gamma(2x) &= (2\pi)^{-1/2}2^{2x-1/2}\Gamma(x)\Gamma(x+\tfrac{1}{2}) \\
\psi(2x) &= \log(2) + \big(\psi(x) + \psi(x+\tfrac{1}{2})\big)/2 \\
\psi_n(2x) &= \big(\psi_n(x) + \psi_n(x+\tfrac{1}{2})\big)/2^{n+1}
\end{aligned}$$

aufgelöst nach den Funktionen mit Argument $x + \frac{1}{2}$; alle anderen darin vorkommenden Funktionen sind nun bekannt.

Zusammenfassend lässt sich feststellen, dass die Fortpflanzung einer auf einem Containerobjekt aufgerufenen Methode auf seine Kinder in vielen Fällen ein übersichtliches Verfahren darstellt um mathematische Operationen auf komplexen Ausdrücken zu implementieren. Im Beispiel der Laurententwicklung der Gamma-Funktion läuft die eleganteste Programmierung zwar darauf hinaus, dass auch die Digamma-, die Polygamma-, die ζ-Funktion und die Bernoulli-Zahlen effizient programmiert werden. Das Ergebnis ist jedoch ein natürlicher, leicht zu wartender und modularer Code, der abgeschlossen ist in dem Sinne, dass er alles zu leisten vermag, was für dimensionale Regularisierung erforderlich ist.

4.7. Die Matrix-Klasse

Die Fähigkeit, mit zweidimensionalen Arrays beliebiger Elemente umzugehen, ist in jedem der gängigen CAS implementiert. Bei der Darstellung von Matrizen in Computern unterscheidet man vier verschiedene Ansätze, aus denen eine Auswahl zu treffen ist:

1. Explizite dichte Darstellung:
   Eine $n \times m$-Matrix wird spezifiziert durch vollständige Angabe aller ihrer Einträge. Die Darstellung ist üblicherweise ein Vektor (manchmal auch eine Liste).

   a) Numerische Matrizen: Die Einträge sind üblicherweise alle vom gleichen eingebauten Typ wie `float` oder `double`. Darauf optimierte Algorithmen werden von einigen rein numerischen Paketen wie beispielsweise MATLAB angeboten.

   b) Symbolische Matrizen: Die Einträge sind nicht von einem eingebauten numerischen Typ sondern entweder symbolisch oder aus einem Ring oder einer rationalen Erweiterung eines Ringes, etwa $\mathbb{Q}$.

Die aus dem numerischen Fall wohlbekannten Algorithmen lassen sich nur sehr eingeschränkt auf den symbolischen Fall übertragen. Dieser Abschnitt wird darlegen, warum das so ist und wie in GiNaC der Versuch unternommen wird, mit vertretbarem Aufwand dennoch möglichst effizient zu sein.

2. Explizite dünn besetzte Darstellung:
   Eine $n \times m$-Matrix wird spezifiziert durch Angabe aller interessanter Elemente, die anderen werden typischerweise als verschwindend angenommen. Dies ist besonders in der Numerik von Interesse, wenn Matrizenoperationen aufwändig werden und eine Multiplikation schneller als etwa $\mathcal{O}(n^3)$ verlangt wird. Als Darstellung bietet sich in C++ der STL-Container `map` an, dessen Zugriffszeit von logarithmischer Ordnung ist – allerdings bedingt durch die Implementierung als RB-Baum mit einem Overhead von drei Zeigern und einer booleschen Variablen, also bis zu 28 Bytes plus Füllbytes, nur für sehr große dünn besetzte Matrizen in Erwägung zu ziehen.

3. Implizite atomare Darstellung:
   Jede Matrix wird dargestellt durch genau ein Symbol. Das CAS implementiert im Wesentlichen die Arithmetik über einem nichtkommutativen Ring. Es war eine der Haupttriebfedern hinter GiNaC, Objekte wie Farb-, Dirac,- oder Isospinmatrizen gerade nicht explizit als zusammengesetzte Objekte sondern implizit darzustellen über die Eigenschaften der von ihnen erzeugten Algebren. Dies geschieht jedoch nicht in der Matrix-Klasse sondern ist Aufgabe der Klassen `ncmul` und `indexed`.

4. Implizite elementbasierte Darstellung:
   Bei diesem recht neuen Ansatz, der erstmals in [Fate 2001] vorgeschlagen wurde, können Matrizen ohne Wissen über ihre Dimension dargestellt und manipuliert werden. Zum Beispiel kann die $n \times n$ Hilbert-Matrix dargestellt werden durch Spezifizierung ihrer Elemente: $h_{ij} = 1/(i+j+1)$, $i, j \in [0 \ldots n-1]$. Unter bestimmten Voraussetzungen können solche Matrizen addiert, multipliziert und sogar ihre Determinanten berechnet werden. Diese *lazy evaluation*-Technik ist jedoch hauptsächlich zum automatisierten Beweisen von Theoremen interessant und am besten in Sprachen mit echter Unterstützung von Lambda-Kalkül (wie Scheme oder Lisp) zugänglich.

In *xloops* müssen häufig lineare Transformationen ausgeführt werden wie zum Beispiel beim Lorentztransformieren von Impulsen, oder es müssen kleine lineare Gleichungssysteme gelöst werden wie etwa beim Zusammenfassen von Integrationsgebieten oder beim Integrieren rationaler Funktionen mit der Horowitzschen Methode [Horo 1971]. Da die auftretenden Matrizen üblicherweise klein sind, wurde die dichte explizite Darstellung gewählt. Weiter unten werden wir beschreiben wie durch sorgfältige Implementierung der darauf agierenden Algorithmen auch diese Darstellung bisweilen verbessert werden kann, um zum Beispiel das $\mathcal{O}(n^3)$-Verhalten der Matrixmultiplikation für dünn besetzte aber dicht dargestellte Matrizen abzukürzen.

Das effiziente Implementieren von Paketen für Lineare Algebra ist mittlerweile ein eigener kleiner Industriezweig – angefangen bei den Numerical Recipes reicht das Spektrum über LINPACK, LAPACK und BLAS, bis zu kommerziellen Paketen wie IMSL und NAG. Diese Pakete sind jedoch für sehr große *numerische* lineare Systeme entworfen worden und versuchen

daher, nicht nur die Zahl der Rechenoperationen sondern auch den verwendeten Speicherplatz zu minimieren, sowie durch Rundungsfehler hervorgerufene Entartungen zu entdecken und soweit möglich zu verhindern. Im Falle symbolischer Matrizenoperationen kommen die meisten der dort verwendeten Tricks nicht in Frage, erstens aufgrund des enormen Anwachsens der Zwischenausdrücke, und zweitens weil zwischen Symbolen keine Ordnungsrelation besteht. Das Anwachsen der Zwischenausdrücke ist darauf zurückzuführen, dass arithmetische Operationen wie Addition, Multiplikation etc. eben nicht *ausgeführt*, sondern nur *niedergeschrieben* werden können. So ist die Determinante der generischen 3×3-Matrix $\tilde{M}_{3\times 3}$

$$\det \tilde{M}_{3\times 3} \equiv \det \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} = aei - afh - bdi + bfg + cdh - ceg$$

bereits ein Additionsobjekt mit 6 Multiplikationsobjekten als Einträgen. Schon die Determinante einer generischen 5×5-Matrix würde mit ihren 120 Elementen hier eine halbe Seite füllen. Die Determinante der generischen symbolischen Matrix ist also ebenso rechenintensiv wie die Permanente $p = \sum_{\sigma\in S_n} \prod_{i=0}^{n-1} a_{i\,\sigma_i}$, die schon für numerische Matrizen NP-vollständig ist. Aufgrund der fehlenden Ordnungsrelation zwischen Symbolen kann man ferner auch nur insofern von einem Pivotelement sprechen, als dass das Element nicht verschwindet. Diese häufig übersehene Tatsache könnte man als „Symbolikfalle“ bezeichnen. Sie besagt, dass der Komplexitätsgrad ein und desselben Algorithmus angewandt auf symbolische Objekte ein höherer sein kann als angewandt auf numerische Objekte. Dies alles schließt jedoch nicht aus, dass gewisse Algorithmen, die in der numerischen Linearen Algebra zur Anwendung kommen, auf symbolischen Systemen nicht auch anwendbar sind. So ist zum Beispiel die gewöhnliche Matrizenmultiplikation augenscheinlich ein Prozess der Komplexität $\mathcal{O}(n^3)$, jedoch gibt es tatsächlich Algorithmen mit einer besseren asymptotischen Komplexität (wie derjenige von Strassen [Stra 1969] mit $\mathcal{O}(n^{\log_2 7} \simeq n^{2{,}81})$ oder von Coppersmith und Winograd [CoWi 1990] mit $\mathcal{O}(n^{2.376})$), die im Prinzip auch auf symbolische Pakete übertragbar wären. Da diese jedoch die asymptotische Komplexität ist und aufgrund der resultierenden großen Zwischenergebnisse ohnehin zu überlegen ist, warum man dicht besetzte symbolische Matrizen der Größe $n \gtrsim 100$ multiplizieren muss, ist die Anwendung solch fortgeschrittener Algorithmen in symbolischen Paketen zweifelhaft. Tatsächlich scheint kein symbolisches Paket zu existieren, welches von ihnen Gebrauch macht [DST 1988]. Daraus kann man nur lernen, dass es sich durchaus lohnt, die bekannten primitiven Algorithmen vor einer Implementierung einmal genau auf ihre Effizienz hin zu untersuchen. Dieser Abschnitt wird mit ein paar zwar nicht neuen, aber dennoch nicht selbstverständlichen Ergebnissen dieser Analyse enden. Die Erfahrung hat gezeigt, dass dieses Gebiet ein ungeheures Optimierungspotenzial enthält, welches zumindest teilweise ausgenutzt werden muss, um nicht später mit astronomischen Laufzeiten konfrontiert zu werden.

Darstellung der Matrix-Klasse

Zunächst einmal erläutern wir die Darstellung von Matrizen in GiNaC. Für die dichte Darstellung bieten sich Vektoren an. Manche Lisp-basierte Systeme (wie Maxima und Macsyma) benutzen die in Lisp nahe liegenden Listen, was oberflächlich betrachtet äquivalent zur Benutzung von Vektoren ist. Dies ist jedoch ungeschickt: da die Zugriffszeit in einer Liste

nicht konstant ist, führt es zu Überraschungen bei der Implementierung von Matrixmultiplikation, die selbst im numerischen Fall dann wesentlich schlechter als $\mathcal{O}(n^3)$ sein kann. Zwar können zweidimensionale Container in der STL durch Iteration generiert werden (z.B durch die Template-Instanzierung vector<vector<ex> > m;), jedoch täuscht dies eine zweidimensionale Struktur nur vor, und verschachtelte Templates bereiteten bei manchen Compilern Schwierigkeiten. Da die Repräsentation in einem objektorientierten Programmierparadigma ohnehin vom Anwender weggekapselt wird, wurde eine Darstellung gewählt, in der die Zeilen einer Matrix hintereinander in einem Vektor abgespeichert werden und zwei Hilfsvariablen über die Dimension der Matrix informieren. Die zweidimensionale Indizierung muss dann von jeder einzelnen Methode konsistent auf die eindimensionale abgebildet werden. Eine Methode zur Transponierung des Matrixobjektes schreibt sich dann in GiNaC wie folgt:

```
1  const class matrix
2  matrix::transpose(void) const
3  {
4      std::vector<ex> trans(col*row);
5
6      for (unsigned r=0; r<col; ++r) {
7          for (unsigned c=0; c<row; ++c) {
8              trans[r*row+c] = m[c*col+r];
9          }
10     }
11     return matrix(col,row,trans);
12 }
```

Hierbei bezeichnen row und col die Dimensionen (Zeilen und Spalten) und m ist die Darstellung der Matrix im Objekt – also ein Vektor aus ex-Elementen der Länge row*col. Man beachte, dass die Indizierung des STL-Vektors über eckige Klammern geschieht statt über die .at()-Methode, das heißt, dass range-checking aus Gründen der Effizienz und der Hybris des Programmierers innerhalb der Methoden nicht verwendet werden.

Die Frage ist erlaubt, warum eine dichte Darstellung gewählt worden ist statt einer dünnen (sparse) oder beiden Darstellungen mit einer Wahlmöglichkeit für den Anwender. Eine dünne Matrix ließe sich beispielsweise ideal durch einen assoziativen Array wie zum Beispiel den STL-Container map<int,ex> beziehungsweise map<pair<int,int>,ex> darstellen. Die Zugriffskomplexität in diesem Container ist zwar nicht explizit im Sprachstandard spezifiziert, jedoch wird üblicherweise implizit verstanden, dass eine map als sortierter RB-Baum implementiert wird. Daher können Suchoperationen binär in $\mathcal{O}(\log(n))$ durchgeführt werden, Einfügungen sind jedoch von $\mathcal{O}(n)$.

Zunächst einmal ist das Mischen dieser beiden Darstellungen notorisch schwierig. Addiert man eine dicht besetzte Matrix D mit einer dünn besetzten S, so erwartet man als Ergebnis natürlich eine dichte Matrix und das System sollte auch eine solche Darstellung für $D + S$ wählen. Multipliziert man eine dicht besetzte Matrix D mit einer dünn besetzten S, so ist der Fall schon weniger klar und das Ergebnis DS kann sowohl dicht besetzt sein (z.B. D beliebig, $S = \mathbb{1}$) als auch dünn (z.B. $j = 0 \Leftrightarrow D_{ij} = 0$, $i > 0 \Leftrightarrow S_{ij} = 0$, also $DS = \mathbb{0}$). Ferner haben Eliminationsverfahren die Eigenschaft, dünn besetzte Matrizen in dicht besetzte Dreiecksmatrizen zu überführen. Mischen dieser beiden Darstellungen erfordert also ständige Entscheidungen, entweder des Programmierers oder auf Konstruktorebene.

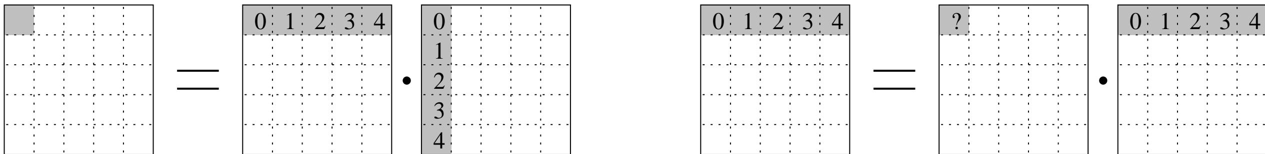


Abbildung 4.6.: *Schleifenumordnung bei der Matrix-Multiplikation. Links die naive Implementierung. Rechts eine Anordnung, die eine Abkürzung der innersten Schleife zulässt, wenn das mit „?“ markierte Element exakt verschwindet.*

Glücklicherweise lassen sich selbst in der hier gewählten dichten Darstellung bei einigen Standardalgorithmen Verbesserungen anbringen, die einen Teil der Vorteile der dünnen Darstellung mitbringen. Man betrachte beispielsweise die Matrixmultiplikation, wie gesagt ein Prozess von $\mathcal{O}(n^3)$, zumindest bei dichten Matrizen. Die Komplexität[12] der folgenden Implementierung geht für dünne Matrizen asymptotisch gegen $\mathcal{O}(n^2)$:

```
1  const matrix
2  matrix::mul(const matrix & other) const
3  {
4      std::vector<ex> prod(row*other.col);
5
6      for (unsigned r1=0; r1<row; ++r1) {
7          for (unsigned c=0; c<col; ++c) {
8              if (m[r1*col+c]==0)
9                  continue;                 // Überspringe nächste Schleife
10             for (unsigned r2=0; r2<other.col; ++r2)
11                 prod[r1*other.col+r2] += m[r1*col+c] * other.m[c*other.col+r2];
12         }
13     }
14     return matrix(row, other.col, prod);   // Ctor aus Darstellung
15 }
```

Hierin wurden die beiden inneren Schleifen gegenüber der Schulbuch-Methode umgeordnet, so dass die Einträge in der Ergebnismatrix sukzessive aufakkumuliert anstatt Eintrag für Eintrag vollständig berechnet werden. Dies ermöglicht die Abkürzung in Zeile *9*, falls ein Element der `*this`-Matrix (der linken Matrix) verschwindet.[13]

[12] Wir definieren den Begriff Komplexität hier etwas unorthodox: Normalerweise bezeichnet man damit die Anzahl der Körperoperationen, also die Summe der Additionen, Subtraktionen, Multiplikationen und Divisionen. Hier verstehen wir darunter die Anzahl der aus dem Speicher zu holenden Objekte. Das ist bei symbolischen Rechnungen realistischer, da der Aufwand der Körperoperationen hier im Gegensatz zu Gleitkommarechnungen nicht nach oben beschränkt ist.

[13] Ohne die Abfrage, ob das Element aus der linken Matrix verschwindet, wird dieselbe Umordnung der beiden inneren Schleifen übrigens auch bei der Multiplikation großer Matrizen aus Gleitkommazahlen angewendet. Sie vermeidet nämlich ein n-maliges vollständiges Auslesen der rechten Matrix. Stattdessen wird jede Zeile erst n mal ausgelesen, bevor zur Nächsten übergegangen wird. Dadurch verbleiben die Daten eher in einem schnellen CPU-nahen Speicher, anstatt aus einem langsamen CPU-fernen Speicher geholt werden zu müssen („Cache-Affinität“). Bei der Programmierung von Vektorrechnern gehört diese Unterscheidung zum kleinen Einmaleins: das normale Verfahren wird dort als `ijk`-Methode bezeichnet, das oben vorgestellte als `ikj`-Methode (Siehe zum Beispiel [CoTr 1995]).

Implementierte Eliminationsverfahren

Betrachten wir nun die Effizienz, mit der dicht besetzte symbolische Matrizen invertiert werden können, bzw. deren Determinante berechnet wird. Es wird sich dabei herausstellen, dass dies weniger trivial ist als es zunächst den Anschein hat. Die naheliegendste Methode ist, die Definition der Determinante über die Permutationsgruppe S_n

$$\det \tilde{M}_{n\times n} = \sum_{\sigma\in S_n} \mathrm{sign}(\sigma) m_{1,\sigma_1} \dots m_{n,\sigma_n}$$

auch zu ihrer Bestimmung heranzuziehen. Für die generische 3×3-Matrix erhält man

$$\det \tilde{M}_{3\times 3} = aei - afh - bdi + bfg + cdh - ceg,$$

aber da dieser Fall exotisch ist und selten n^2 freie Symbole auftreten, können normalerweise in diesem Ergebnis Terme zusammengefasst werden.

Bei der als Laplace-Entwicklung bekannten Entwicklung nach einer ausgewählten Zeile oder Spalte können solche Zusammenfassungen häufig frühzeitig ausgenutzt werden. Untersuchen wir aber die Komplexität zunächst ganz allgemein, so stellen wir noch ein weiteres Verbesserungspotenzial fest, falls die Dimension größer als drei ist. Die Laplace-Entwicklung der generischen 3×3-Matrix nach der ersten Spalte liefert:

$$\det \tilde{M}_{3\times 3} \equiv \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - d \begin{vmatrix} b & c \\ h & i \end{vmatrix} + g \begin{vmatrix} b & c \\ e & f \end{vmatrix}.$$

Bei 4×4-Matrizen führt sie aber dazu, dass alle 2×2-Minoren doppelt berechnet werden: Diejenige unten rechts bestehend aus den Elementen $(2,2)$, $(3,3)$, $(2,3)$ und $(3,2)$ kommt beispielsweise jeweils in der Minore für die Elemente $(0,0)$ und $(1,0)$ vor (Abbildung 4.7). Allgemein wird jede $k\times k$-Minore in einer $n\times n$-Matrix $(n{-}k)!$ mal berechnet werden müssen, wobei $1 < k < n-1$. Die Anzahl der auszuführenden Multiplikationen beträgt jeweils k, diejenige der Additionen $k-1$. Nun gibt es $\binom{n}{k}$ solcher Minoren, die Anzahl der Operationen beträgt also

$$K_{\mathrm{Laplace}} = (2+1)(n-2)!\binom{n}{2} + (3+2)(n-3)!\binom{n}{3} + \cdots + (n+n{-}1),$$

was schon für moderate n schnell gegen $K_{\mathrm{Laplace}} \simeq e\,n!$ konvergiert. Die nahe liegende Verbesserung besteht im Abspeichern der Zwischenergebnisse, damit jede Minore nur einmal berechnet wird. Die Anzahl der Operationen beträgt dann nur noch

$$K'_{\mathrm{Laplace}} = (2+1)\binom{n}{2} + (3+2)\binom{n}{3} + \cdots + (n+n{-}1) = (n-1)(2^n-1)$$

und der Speicheraufwand maximal $\simeq 2\binom{n}{n/2}$ Ausdrücke. (Im Schritt $\lfloor n/2\rfloor$ müssen aus $\binom{n}{\lfloor n/2\rfloor}$ abgespeicherten Minoren $\binom{n}{\lfloor n/2\rfloor+1}$ neue gebildet werden.) Dieses Verfahren ist in der Methode `matrix::determinant_minor()` implementiert. Obwohl es vom Komplexitätsgesichtspunkt nicht polynomial sondern exponentiell ist (siehe Tabelle 4.2), handelt es sich häufig um das

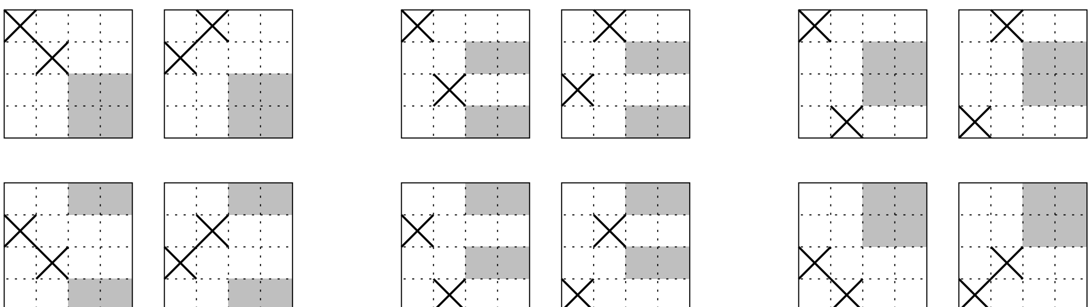


Abbildung 4.7.: *Überflüssige Berechnung von 2×2-Minoren bei Laplace-Entwicklung einer 4×4-Matrix ohne Zwischenspeichern. Die grauen Felder markieren die Minoren.*

schnellste Verfahren – beispielsweise bei dichten Matrizen mit multivariaten Polynomen als Einträgen [GeJo 1976].

Der in [ShSt 1998] für symbolische Matrizen explizit empfohlene Algorithmus von Leverrier (bisweilen auch als Algorithmus von Fadeev beschrieben) geht wie folgt vor. Sei M die $n \times n$ Ausgangsmatrix. Man setzt $B_1 = M$ und $c_1 = \mathrm{Sp}(M)$. Dann berechnet man

$$B_i = M(B_{i-1} - c_{i-1}\mathbb{1}), \qquad c_i = \mathrm{Sp}\,(B_i)/i, \qquad i = 2, \ldots, n.$$

Die c_i sind genau die Koeffizienten des charakteristischen Polynoms von M und $(-)^{n+1}c_n$ mithin die Determinante.

Naiv in GiNaC implementiert liefert das Verfahren jedoch für die generische symbolische 3×3-Matrix den unförmigen Ausdruck

$$\begin{aligned}\det \tilde{M}_{3\times3} \;=\; & \tfrac{1}{3}\big(hc+ba+b(-a-i)\big)d + \tfrac{1}{3}\big(g(-e-i)+gi+hd\big)c + \tfrac{1}{3}f\big(hi+h(-a-i)+gb\big) \\ & + \tfrac{1}{3}i\big(\tfrac{1}{2}(-a-e)i-\tfrac{1}{2}e(-a-i)-bd-\tfrac{1}{2}a(-e-i)\big) + \tfrac{1}{3}h\big(fe+dc+(-a-e)f\big) \\ & + \tfrac{1}{3}\big(gf+d(-e-i)+ed\big)b + \tfrac{1}{3}\big(-\tfrac{1}{2}(-a-e)i-\tfrac{1}{2}e(-a-i)+\tfrac{1}{2}a(-e-i)-hf\big)a \\ & + \tfrac{1}{3}\big(-\tfrac{1}{2}(-a-e)i-gc+\tfrac{1}{2}e(-a-i)-\tfrac{1}{2}a(-e-i)\big)e + \tfrac{1}{3}g\big(fb+(-a-e)c+ac\big),\end{aligned}$$

der auf jeden Fall expandiert werden muss, wie man schon an den artifiziellen Brüchen erkennt – das Ergebnis muss ja ein Polynom über den ganzen Zahlen sein.

Tatsächlich ist das Ergebnis in Leverrier's Verfahren selbst nicht ganz so unhandlich, wie es aussieht, da häufig auftretende Unterausdrücke in einem System wie GiNaC nur Referenzen auf einmal abgelegte Ausdrücke darstellen. Erst beim Ausmultiplizieren mit `expand()` verursacht es erheblichen Aufwand. Der eigentliche Rechenaufwand beim Bilden von symbolischen Determinanten besteht also in der Notwendigkeit der Kanonisierung der auftretenden Ausdrücke. Die nebenstehende Tabelle gibt einen Eindruck vom explosionsartigen Anschwellen der Komplexität. In ihr sind exemplarische Laufzeiten jeweils für die Generierung und die Kanonisierung von Determinanten dicht besetzter symbolischer $n \times n$-Matrizen in GiNaC aufgetragen. Daraus geht eindeutig hervor, dass der Algorithmus für symbolische Matrizen nicht geeignet ist. Er kann zwar deutlich verbessert werden, indem man bei jedem Schritt bei der Spurbildung und der Multiplikation ausmultipliziert, bleibt aber dennoch weit hinter

n	t_{gen}	t_{expand}
4	<0.01s	0.2s
5	0.01s	0.94s
6	0.02s	30.4s
7	0.04s	1202s

Tabelle 4.1.: *Laufzeiten zur Generierung und Ausmultiplikation von symbolischen Determinanten mit dem Leverrier-Verfahren.*

z.B. der Laplace-Entwicklung zurück. Im numerischen Fall ist die Gauß- bzw. die Bareiss-Entwicklung geeigneter. Der Algorithmus von Leverrier ist in GiNaC daher lediglich für die Berechnung von charakteristischen Polynomen implementiert und auch dann nur, wenn alle Einträge der Matrix Zahlen sind. Da insgesamt n Matrix-Multiplikationen auszuführen sind, ist seine Komplexität dann von der Ordnung $\mathcal{O}(n^3)$ bis $\mathcal{O}(n^4)$, je nachdem ob die Matrix dünn oder dicht besetzt ist.

Im folgenden wenden wir uns drei klassischen Eliminationsverfahren zu, die die Ausgangsmatrix in eine (obere) Dreiecksmatrix überführen. Der Algorithmus der Gauß-Elimination lautet:

$$\begin{aligned} m_{i,j}^{(0)} &= m_{i,j} \\ m_{i,j}^{(k+1)} &= m_{i,j}^{(k)} - \frac{m_{i,k}^{(k)} m_{k,j}^{(k)}}{m_{k,k}^{(k)}}. \end{aligned} \tag{4.9}$$

Hierin, wie auch bei allen folgenden Eliminationsschemata, laufen die Indizes $0 \le k < n-1$, $k \le i < n$, $k < j < n$. Den Divisor $m_{k,k}^{(k)}$ bezeichnet man als Pivotelement.

Im numerischen Falle kann die Komplexität leicht berechnet werden. Wir beschränken uns hier auf den Fall $m = n$: In jedem Eliminationsschritt werden 3 elementare Operationen in der $n' \times n'$ Submatrix rechts unten ausgeführt, wobei $n' = n - 1 \ldots 1$. Die Anzahl der Rechenschritte beträgt also

$$3\big((n-1)^2 + (n-2)^2 + \cdots + 2^2 + 1^2\big) = n^3 - \frac{3}{2}n^2 + \frac{1}{2}n.$$

Die Determinante ist danach das Produkt der n Diagonalelemente. Damit ist die Gauß-Spalte in Tabelle 4.2 erklärt.

Anhand eines Beispiels wird deutlich, welche Transformationen die Elemente der Matrix bei den Eliminationsschritten durchlaufen:

$$\begin{pmatrix} 6 & -3 & -4 & 9 \\ -7 & -5 & 6 & -9 \\ -8 & -2 & 2 & -1 \\ 4 & -7 & 1 & 8 \end{pmatrix} \longrightarrow \begin{pmatrix} 6 & -3 & -4 & 9 \\ 0 & -17/2 & 4/3 & 3/2 \\ 0 & 0 & -218/51 & 169/17 \\ 0 & 0 & 0 & 1705/218 \end{pmatrix}.$$

Obwohl alle Elemente ursprünglich aus einem Integritätsbereich waren (in diesem Falle aus $\mathbb{Z}$), wird dieser schon im ersten der drei Schritte verlassen. Ist das Ziel des Eliminationsverfahrens die Invertierung von Matrizen, so ist dies zu erwarten. Ist es aber die Berechnung der Determinanten, die ja selbst als Polynom Element desselben Integritätsbereiches ist, so wäre es wünschenswert, alle Zwischenschritte in demselben Integritätsbereich durchzuführen. Dies gilt besonders für Polynome, wo der Aufwand für das Berechnen des ggT sehr groß sein kann. Dient die Elimination der Lösung eines Gleichungssystemes, so liegen die Lösungen zwar im Quotientenkörper, aber die Notwendigkeit zu Dividieren kann bis zum schrittweisen Auflösen verschoben werden. Die divisionsfreie Elimination (engl. *division free elimination*) umgeht dieses Problem. Die Eliminationsvorschrift lautet bei diesem Algorithmus:

$$\begin{aligned} m_{i,j}^{(0)} &= m_{i,j} \\ m_{i,j}^{(k+1)} &= m_{i,j}^{(k)} m_{k,k}^{(k)} - m_{i,k}^{(k)} m_{k,j}^{(k)}. \end{aligned} \tag{4.10}$$

Ein Implementationsproblem bei Gauß-Elimination

$$m^{(0)} \equiv m = \begin{pmatrix} x-1 & x^2-x & 1 \\ 1 & x & 1 \\ 0 & x & 0 \end{pmatrix}$$

Wenden wir den ersten Gauß-Eliminationsschritt an (k=0 in (4.9)) und konstruieren alle auftretenden Terme gemäß unseren Regeln:

$$m^{(1)} = \begin{pmatrix} x-1 & x^2-x & 1 \\ 1 & x-\frac{x^2-x}{x-1} & 1-\frac{1}{x-1} \\ 0 & x & 0 \end{pmatrix}$$

Im nächsten und letzten Schritt wird durch das Pivotelement $m^{(1)}_{1,1}$ geteilt. Es ist aber $m^{(1)}_{1,1} = x - \frac{x^2-x}{x-1}$, was verschwindet. Die defensive Vereinfachung hat das Pivotelement nicht zu 0 vereinfacht, der naive Test auf 0 versagt und das Endergebnis enthält das Element $m^{(2)}_{2,2} = x(1-\frac{1}{x-1})\big/(x-\frac{x^2-x}{x-1})$ mit einer versteckten Division durch Null. Die Gauß-Elimination verlässt den Integritätsbereich $\mathbb{Z}[x]$ und daher hilft auch kein Ausmultiplizieren in (4.9). Hier muss in jedem Schritt diejenige Vereinfachung explizit aufgerufen werden, die im entsprechenden Quotientenkörper, hier also $\mathbb{Q}[x]$, auf Null testen kann, also `normal()`.

Anhand unseres Beispieles sieht man, dass hier zwar keine Quotienten gebildet, die Zwischenausdrücke jedoch sehr groß werden. Man sieht leicht ein, dass im Falle von Polynomen des Grades m das Element (n,n) rechts unten nach der letzten Elimination den Grad $2^n m$ hat:

$$\begin{pmatrix} 6 & -3 & -4 & 9 \\ -7 & -5 & 6 & -9 \\ -8 & -2 & 2 & -1 \\ 4 & -7 & 1 & 8 \end{pmatrix} \longrightarrow \begin{pmatrix} 6 & -3 & -4 & 9 \\ 0 & -51 & 8 & 9 \\ 0 & 0 & 1308 & -3042 \\ 0 & 0 & 0 & -3130380 \end{pmatrix}.$$

Teilerfreie Elimination (engl: *fraction free elimination*) behebt dieses Problem der divisionsfreien Elimination durch die Beobachtung, dass das Pivotelement des letzten Eliminationsschrittes alle Elemente der eliminierten Matrix teilt. In obigem Beispiel gilt in der dritten Zeile $6|1308$ und $6|3042$ und in der vierten $51|86955$, wo 86955 das rechte untere Element nach dem vorletzten Eliminationsschritt ist ($6^2 \cdot 86955 = 3130380$). Die Eliminationsvorschrift lautet dann:

$$\begin{aligned} m^{(0)}_{i,j} &= m_{i,j} \\ m^{(-1)}_{-1,-1} &= 1 \\ m^{(k+1)}_{i,j} &= \frac{m^{(k)}_{i,j} m^{(k)}_{k,k} - m^{(k)}_{i,k} m^{(k)}_{k,j}}{m^{(k-1)}_{k-1,k-1}} \end{aligned} \qquad (4.11)$$

n	Permutations-gruppe	Laplace-Entwicklung ohne Zwischen-speichern	Laplace-Entwicklung mit Zwischen-speichern	Algorithmus von Leverrier	Eliminationsverfahren Gauß	Eliminationsverfahren Bareiss
1	0	0	0	0	0	0
2	3	3	3	16	4	3
3	17	14	14	108	17	16
4	95	63	45	384	45	47
5	599	324	124	1000	94	104
6	4319	1955	315	2160	170	195
7	35279	13698	762	4116	279	328
8	322559	109599	1785	7168	427	511
9	3265919	986408	4088	11664	620	752
10	36287999	9864099	9207	18000	864	1059
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\gg 1$	$\sim n\,n!$	$\sim e\,n!$	$\sim 2^n\,n$	$\sim 2n^4$	$\sim n^3$	$\sim \frac{4}{3}n^3$

Tabelle 4.2.: *Anzahl der elementaren Rechenoperationen (Additionen, Multiplikationen und Divisionen) zur Berechung der Determinanten einer Matrix in verschiedenen Algorithmen unter der Annahme, dass die Rechenoperationen ausgeführt werden können.*

Angewendet auf unser bekanntes Beispiel ergibt sich nach drei Eliminationsschritten

$$\begin{pmatrix} 6 & -3 & -4 & 9 \\ -7 & -5 & 6 & -9 \\ -8 & -2 & 2 & -1 \\ 4 & -7 & 1 & 8 \end{pmatrix} \longrightarrow \begin{pmatrix} 6 & -3 & -4 & 9 \\ 0 & -51 & 8 & 9 \\ 0 & 0 & 218 & -507 \\ 0 & 0 & 0 & 1705 \end{pmatrix}.$$

Durch die in jedem Schritt durchgeführte Division sind die Elemente der Matrix in erträglicher Größe geblieben: Für Polynome des Grades m in der Ausgangsmatrix hat das Element (n,n) nach der letzten Elimination nun den Grad nm. Es ist anzumerken, dass keine weiteren Kürzungen möglich sind, da nach dem letzten Schritt im Element (n,n) bereits die Determinante der ursprünglichen Matrix steht.

Der Beweis, dass diese Division immer exakt aufgeht, erfolgt mithilfe der Sylvester-Identität. Hierzu definieren wir die Matrix $M^{(k)} \equiv (m_{ij}^{(k)})$ aus Subdeterminanten von $M^{(0)}$:

$$\begin{aligned} \left(m_{ij}^{(0)}\right) &:= m^{(0)} \\ m_{ij}^{(k)} &:= \begin{vmatrix} m_{0,0} & \cdots & m_{0,k-1} & m_{0,j} \\ \vdots & \ddots & \vdots & \vdots \\ m_{k-1,0} & \cdots & m_{k-1,k-1} & m_{k-1,j} \\ m_{i,0} & \cdots & m_{i,k-1} & m_{i,j} \end{vmatrix}. \end{aligned} \tag{4.12}$$

Die so definierte Matrix aus Subdeterminanten hat die Dimension $(n-k)\times(n-k)$ (vergleiche Abbildung 4.8 links). Für die Indizes gilt $k \leq i, j < n$.

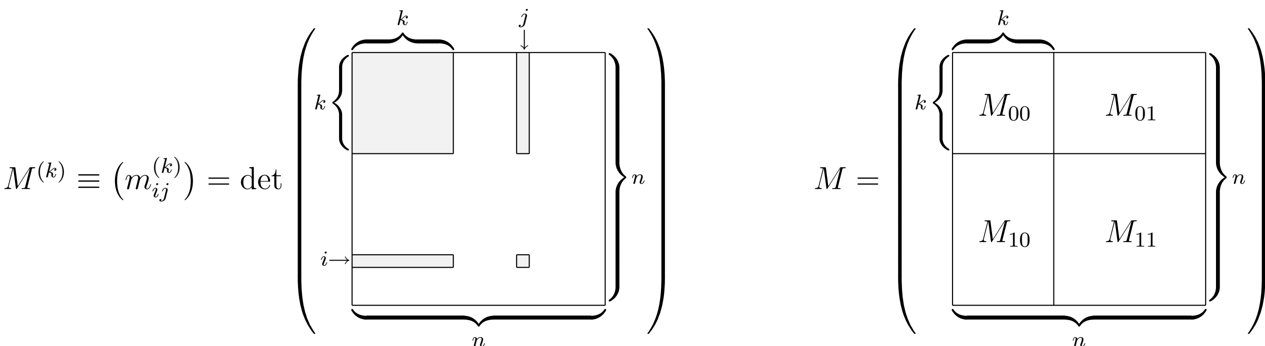


Abbildung 4.8.: *Die beim Beweis der Sylvester-Identität vorkommenden Matrixpartitionierungen.*

Satz 4.2 (Sylvester-Identität) *Für alle in Gleichung (4.12) definierten* $m_{ij}^{(k)}$ *gilt*

$$|M|(a_{k,k}^{(k-1)})^{n-k-1} = \begin{vmatrix} m_{k,k}^{(k)} & \cdots & m_{k,n-1}^{(k)} \\ \vdots & \ddots & \vdots \\ m_{n-1,k}^{(k)} & \cdots & m_{n-1,n-1}^{(k)} \end{vmatrix}.$$

Zum Beweis partitionieren wir die Matrix M wie in Abbildung 4.8 rechts angedeutet in vier Submatrizen und splitten sie in zwei Anteile:

$$M = \begin{pmatrix} M_{00} & M_{01} \\ M_{10} & M_{11} \end{pmatrix} = \begin{pmatrix} M_{00} & \mathbb{0} \\ M_{10} & \mathbb{1} \end{pmatrix} \cdot \begin{pmatrix} \mathbb{1} & M_{00}^{-1} M_{01} \\ \mathbb{0} & M_{11} - M_{10} M_{00}^{-1} M_{01} \end{pmatrix},$$

wobei wir M_{00} als nichtsingulär annehmen. Dann ist die Determinante

$$|M| = |M_{00}||M_{11} - M_{10}M_{00}^{-1}M_{01}|.$$

Da die zweite Determinante auf der rechten Seite von der Ordnung $n-k$ ist, können wir $|M_{00}|$ durch Multiplikation mit $|M_{00}|^{n-k-1}$ hineinziehen und erhalten

$$|M||M_{00}|^{n-k-1} = \big||M_{00}|(M_{11} - M_{10}M_{00}^{-1}M_{01})\big|.$$

Hierin erkennt man $|M_{00}|M_{00}^{-1} = M_{00}^{adj}$, so dass sich die rechte Seite nach der Cramer'schen Regel auflösen lässt:

$$\begin{aligned} \big||M_{00}|(M_{11} - M_{10}M_{00}^{-1}M_{01})\big| &= \Big||M_{00}|(m_{i,j} - \sum_{r,s=0}^{k-1} m_{i,r}(M_{00}^{-1})_{r,s} m_{s,j})\Big| \\ &= \Big||M_{00}|m_{i,j} - \sum_{r,s=0}^{k-1} m_{i,r}(M_{00}^{adj})_{s,r} m_{s,j}\Big| \\ &= \big|m_{i,j}^{(k)}\big|, \qquad k \leq i,j < n. \end{aligned}$$

Die letzte Gleichung folgt durch Laplace-Entwicklung der untersten Zeile von (4.12) von rechts nach links. Also haben wir $|M||M_{00}|^{n-k-1} = |m_{i,j}^{(k)}|$. Das ist aber genau die Behauptung des Satzes. □

Ein Implementationsproblem teilerfreier Elimination

Bei der Implementierung der teilerfreien Elimination (Gleichung (4.11)) kommt es zu einem subtilen Problem mit automatisch durchgeführten Vereinfachungsregeln der Klasse `mul`*. Man kann es am besten an einem Beispiel einsehen. Sei*

$$m^{(0)} \equiv m = \begin{pmatrix} (a+b)c & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & f/c^2 \end{pmatrix}.$$

Anwenden des ersten Eliminationsschrittes liefert:

$$m^{(1)} = \begin{pmatrix} (a+b)c & 0 & 0 & 0 \\ 0 & (a+b)c & 0 & 0 \\ 0 & 0 & (a+b)c & 0 \\ 0 & 0 & 0 & (a+b)f/c \end{pmatrix}.$$

Im Element unten rechts bahnt sich bereits das Verhängnis an, denn dort wurde aus $(a+b)cf/c^2$ ein c weggekürzt. Diese sehr frühe Kürzung von Ausdrücken, deren Gleichheit durch syntaktischen Vergleich festgestellt werden kann, findet schon im Konstruktor der Klasse `mul` *statt und ist daher auch nicht durch ein* `.hold()` *zu verhindern. Der nächste Eliminationsschritt würde zu $m^{(2)} = m^{(1)}$ führen, da das Pivotelement $m^{(1)}_{1,1}$ gleich dem Divisor $m^{(0)}_{0,0}$ ist. Trennt man alle Matrixelemente nach Zähler Z und Nenner N auf, so ist der Zähler des Divisors nun $Z(m^{(0)}_{0,0}) = (a+b)c$, der Nenner $N(m^{(0)}_{0,0}) = 1$. Die Division von $Z(m^{(1)}_{3,3}) = (a+b)f$ durch $Z(m^{(0)}_{0,0})$ geht nun jedoch nicht mehr auf. Der Divisor ist nun also kein Teiler mehr und der Algorithmus bricht zusammen.*

Angewendet auf das divisionsfreie Eliminiationsschema liefert die Sylvester-Identität

$$m^{(k)}_{i,j} = \begin{vmatrix} m^{(k-1)}_{k,k} & m^{(k-1)}_{k,j} \\ m^{(k-1)}_{i,k} & m^{(k-1)}_{i,j} \end{vmatrix} \Big/ m^{(k-2)}_{k-1,k-1},$$

was die Teilerfreiheit zeigt, da $m^{(k)}_{i,j}$ per Definition kein Bruch ist.

Die Division in Gleichung (4.11) geht zwar für $k > 2$ immer auf, dies funktioniert jedoch in der Praxis nicht mehr, sobald vorher eine Kürzung durchgeführt worden ist (siehe Kasten auf Seite 109). Daher wird in Quotientenkörpern eine explizite Variante des teilerfreien Eliminationsschemas benötigt, welches Zähler und Nenner getrennt verwaltet:

$$\begin{aligned} m^{(0)}_{i,j} &= m_{i,j} \\ Z(m^{(-1)}_{-1,-1}) &= 1 \\ N(m^{(-1)}_{-1,-1}) &= 1 \\ Z(m^{(k+1)}_{i,j}) &= \frac{Z(m^{(k)}_{i,j})Z(m^{(k)}_{k,k})N(m^{(k)}_{i,k})N(m^{(k)}_{k,j}) - Z(m^{(k)}_{i,k})Z(m^{(k)}_{k,j})N(m^{(k)}_{i,j})N(m^{(k)}_{k,k})}{Z(m^{(k-1)}_{k-1,k-1})} \qquad (4.13) \\ N(m^{(k+1)}_{i,j}) &= \frac{N(m^{(k)}_{i,j})N(m^{(k)}_{k,k})N(m^{(k)}_{i,k})N(m^{(k)}_{k,j})}{N(m^{(k-1)}_{k-1,k-1})}. \end{aligned}$$

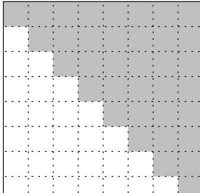
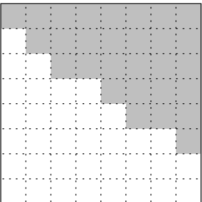



Abbildung 4.9.: *Obere Dreiecksmatrix vs. obere Staffelmatrix.*

Es ist völlig äquivalent zur teilerfreien Elimination und stellt nur eine Umformulierung derselben dar. Der Verwaltungsaufwand ist vernachlässigbar, jedenfalls gegenüber der Alternative, in jedem Schritt den ggT zu berechnen und Polynomdivision durchzuführen.

Diese drei Eliminationsverfahren sind in GiNaC (derzeit als private Methoden der Klasse `matrix`) implementiert. Die Schleifen sind in allen Fällen dabei um Logik erweitert, die die Handhabung unterdeterminierter Gleichungssysteme erlaubt. Hierzu wird nicht in eine Dreiecksmatrix sondern in eine Staffelmatrix (engl: *echelon matrix*) transformiert, in der jede Zeile mehr führende Nullen hat als die vorherige, nicht unbedingt aber genau eine führende Null mehr (Vergleiche Abbildung 4.9). Die Notwendigkeit dafür entsteht bei dem Versuch unterdeterminierte lineare Gleichungssysteme aufzulösen. Dies soll noch möglich sein, wobei sich die Routine dann für die Formulierung der Lösung vom Benutzer mitgelieferter freier Variablen bedienen muss (siehe nächste Seite).

Bei der Implementierung muss man sich auch Gedanken darüber machen, inwieweit das Ergebnis „vereinfacht“ werden sollte. Mathematica lässt das Ergebnis einfach stehen:

```
1  In[1]:= Det[a/(a-b),1,b/(a-b),1]
2
3              a       b
4  Out[1]= ----- - -----
5          a - b   a - b
6
7  In[2]:= Together[%]
8
9  Out[2]= 1
```

MapleV ruft dagegen explizit `normal()` auf, was äquivalent ist zu Mathematicas `Together[]`:[14]

```
1  > with(linalg):
2  Warning, new definition for norm
3  Warning, new definition for trace
4  > det(matrix(2,2,[[a/(a-b),1],[b/(a-b),1]]));
5                                1
```

Ersteres ist zweifellos unbefriedigend, da es nicht der Tatsache Rechnung trägt, dass wir uns von vornherein nicht in einem Integritätsbereich befinden, sondern in einem Quotientenkörper. Zweiteres ist besser, könnte aber verschwenderisch sein, falls Maple's `normal()` im Falle

[14] Streng genommen macht `Together[]` mehr als `normal()`, denn es berücksichtigt algebraische Körpererweiterungen. Es entspricht eher Maples `radnormal()`, welches auch korrekt $\frac{x^2+2ix-1}{x+i} \to x+i$ vereinfacht. Dies ist aber für die Betrachtungen hier ohne Belang.

nichtrationaler Funktionen tatsächlich etwas Nichttriviales macht. Wir rufen in GiNaC explizit `normal()` auf, falls mindestens eines der Elemente der Ausgangsmatrix aus einem Quotientenkörper war.

Die drei beschriebenen Eliminationsverfahren und ihre möglichen Aufrufer sind so orthogonal zueinander implementiert wie nur möglich. Die aufrufenden Funktionen müssen heuristisch die Entscheidung über das zu verwendende Verfahren treffen, falls der Benutzer keine Hinweise gegeben hat. Als Aufrufer kommen in Frage:

`matrix::solve()` Löst ein lineares Gleichungssystem $AB = C$, wobei A eine beliebige $m\times n$-Matrix ist, C eine beliebige $m \times p$-Matrix und B eine $n \times p$-Matrix mit Symbolen. Der Aufruf erfolgt nach dem Schema A`.solve(`B`,`C`)` und liefert die Lösung als Rückgabewert. Anders als bei Computeralgebrasystemen üblich zwingt GiNaC den Benutzer zur Eingabe einer Symbolmatrix B. Dies hat den Vorteil, dass auch bei unterbestimmten Gleichungssystemen keine Symbole mit (für den Benutzer) unvorhersagbaren Namen erzeugt werden müssen. Ein triviales Beispiel hierfür ist das Gleichungssystem

$$\begin{pmatrix} 1 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} c \end{pmatrix}$$

mit der Lösung

$$\begin{pmatrix} c+y \\ y \end{pmatrix}.$$

In MapleV kann man dagegen direkt `linsolve(`A`,`C`)` lösen lassen, bekommt dann aber ein Ergebnis in c und dem neuen Symbol `_t[1][1]`, was für eine automatische Weiterverarbeitung selten brauchbar ist.

`matrix::inverse()` Hier wird zwecks Reduzierung von überflüssigem Programmcode über den Umweg `matrix::solve()` auf die Eliminationsverfahren zugegriffen: man löst einfach das Gleichungssystem $AB = \mathbb{1}$ nach B auf. Man beachte, dass dies nicht weniger optimal ist als andere Invertierungsroutinen, da das Eliminationsverfahren nur einmal auf die augmentierte Matrix $(A|\mathbb{1})$ angewendet wird. Heuristiken für die Auswahl des geeigneten Eliminationsverfahrens brauchen daher auch nur einmal in `matrix::solve()` implementiert zu werden. Dadurch ist `matrix::inverse()` inklusive Fehlerbehandlung nur ca. 20 Zeilen lang.

`matrix::determinant()` Wie wir gesehen haben, liefert die teilerfreie Eliminationsvorschrift (4.11) im rechten unteren Element der eliminierten Matrix die Determinante. Bei der Gauß- und der divisionsfreien Elimination kann die Determinante leicht aus den Diagonalelementen der entstandenen Dreiecksmatrix berechnet werden. Das ist jedoch meist insgesamt aufwändiger. Lediglich im rein numerischen Fall wird die Gauß-Routine bevorzugt, da sie dann äquivalent zur Jordan-Elimination ist. Die Heuristik wird jedoch in den meisten Fällen kein Eliminationsverfahren wählen sondern die auf Seite 103 beschriebene, verbesserte Laplace-Entwicklung.

Weitere Besonderheiten der Matrix-Klasse

Ganzzahlige Potenzen nichtkommutativer Objekte in GiNaC werden normalerweise sofort von `power::eval()` ausmultipliziert, wie zum Beispiel in der Regel $\gamma_0^2 \to \gamma_0\gamma_0$ für Dirac-Matrizen. (Erst in zwei darauffolgenden Schritten wird der Ausdruck zu $\eta_{00}\mathbb{1}$ und schließlich $\mathbb{1}$ evaluiert.) Dies scheint für alle nichtkommutativen Objekte sinnvoll zu sein, sofern sie in atomarer Darstellung vorliegen und Regeln für die Evaluation von Produkten bekannt sind. Dieses Umschreiben in lineare Produkte macht die Objekte direkt einer größeren Anzahl von Vereinfachungen zugänglich als dies mit Evaluationsregeln in `power::eval()` möglich wäre. Man kann dies leicht am Beispiel $(\gamma_0\gamma_1\gamma_0)^2$ nachvollziehen.

Für Potenzen von Matrizen machen wir hier jedoch eine Ausnahme: Sie nicht in `ncmul`-Objekte umzuwandeln macht sie einer schnellen Exponentiationsroutine (siehe [Knu 1998], Abschnitt 4.6.3) zugänglich. Dabei werden einmal berechnete Zwischenergebnisse aufgehoben, wie in $A^4 = A^2 \cdot A^2$, wozu die Binärdarstellung des Exponenten herangezogen wird. Für die Berechnung von A^p müssen dann genau $\lfloor \log_2 p \rfloor + \nu(p) < 2\log_2 p$ Matrixmultiplikationen ausgeführt werden, wobei $\nu(p)$ die Anzahl der Einsen in der Binärdarstellung von p zählt. Der Algorithmus in Pseudocode ausgedrückt lautet dann:[15]

```
1  C ← 𝟙;
2  while (p ≠ 1)
3      if (p odd)
4          C ← C · A;
5      p ← ⌊p/2⌋;
6      A ← A · A;
7  return A · C;
```

Wenn aber `power::eval()` nicht für die Exponentiation zuständig sein soll, stellt sich die Frage, wann diese ausgeführt wird. Da alle typischen Matrixoperationen die Komplexität $\mathcal{O}(n^\beta)$ mit $\beta \geq 2$ haben, macht sich GiNaC den pragmatischen aber nicht ganz orthogonalen Ansatz zu eigen, einen separaten benannten Evaluator `evalm()` einzuführen. Dieser erst addiert und multipliziert Matrizen und führt die Exponentiation zu einer ganzen (nicht notwendigerweise positiven) Zahl aus.

Dieser Weg ist wenig originell, ist er doch identisch mit dem von MapleV eingeschlagenen.[16] Er hat sich aber in der Praxis bewährt, sofern der Benutzer weiß, dass Matrizen vom anonymen Evaluator unangerührt bleiben.

[15] Ersetzt man in den Zeilen *4*, *6* und *7* die Multiplikation durch Addition, so erhält man übrigens den als „Russische Bauernmultiplikation“ bekannten Algorithmus zur Berechnung des Produktes einer beliebigen Zahl a mit einer natürlichen Zahl p.

[16] Man kann sogar leicht herausfinden, dass Maple seit MapleVR4 die schnelle Exponentiation für Matrizen benutzt, indem man die Probematrix

$$A^4 \equiv \begin{pmatrix} a & 1 \\ 0 & 1 \end{pmatrix}^4$$

einmal mit dem unexpandierten Ergebnis von $((A \cdot A) \cdot A) \cdot A$ und dann mit dem Ergebnis von $(A \cdot A) \cdot (A \cdot A)$ vergleicht. Ersteres liefert $(A^4)_{0,1} = a^3 + a^2 + a + 1$, letzteres $(A^4)_{0,1} = a^2(a+1) + a + 1$.

5. Kritische Analyse des GiNaC-Ansatzes

However, the existence of the guild of mathematicians disburdens us from a vain trial in building a complete, universal, computing system.
Trudy Weibel, Gaston Gonnet [WeGo 1991]

5.1. Effizienz

Eine allgemeine vergleichende Effizienzanalyse von Computeralgebrasystemen ist ein aussichtsloses Unterfangen – lediglich punktuelle Untersuchungen bestimmter Fähigkeiten sind sinnvoll und Ergebnisse sind stets mit Vorsicht zu genießen. Dies gilt für den Vergleich der vielfältigen symbolischen Fähigkeiten noch mehr als für das Aufstellen von Benchmarks.[1]

Einerseits steht natürlich immer die Möglichkeit offen, durch eine vollständige Umstrukturierung der Darstellung aller Klassen etwas völlig Neuartiges und vielleicht Performanteres zu schaffen. Andererseits kann man auch durch sorgfältige Feinabstimmung der inneren Abläufe allen Operationen schon auf die Sprünge helfen. Abbildung 5.1 zeigt die Laufzeiten in einer Variation von Fliegners Konsistenztest, in der in der expandierten Summe $(\sum_{i=0}^{49} a_i)^3$ die Ersetzung $a_0 \leftarrow -\sum_{i=2}^{49} a_i$ vorgenommen und das Ergebnis ausmultipliziert wird. Der Test wurde gewählt, weil sich die verwendeten Algorithmen in der entsprechenden Zeitspanne nicht und auch die Implementierung nur unwesentlich geändert haben. Die groben Optimierungsarbeiten wie das Übergeben von Referenzen statt Objekten wurden alle schon vor Version 0.4 vorgenommen. Tabelle 5.1 listet die Eckpunkte, die zu den Verbesserungen von insgesamt etwa 50% beigetragen haben dürften. Es hat den Anschein, dass solcherlei Feinabstimmungen weitgehend ausgereizt sind.

Es gibt einige bekanntere Kollektionen, in denen die Fähigkeiten von Computeralgebrasystemen verglichen werden. In [West 1995, West 1999] werden jedoch hauptsächlich sehr fortgeschrittene Fähigkeiten wie zum Beispiel die symbolische Integration getestet – lediglich in der Sparte Laurentreihenentwicklung und ggTs könnten wir hier Vergleiche anstellen. Ein auch für uns interessanter Test wurde in [LeWe 1999] vorgestellt. Dort wurden einige extrem große Ausdrücke erzeugt und Umformungen vorgenommen, die zum größten Teil in GiNaC implementierte Algorithmen voraussetzen. Diese Algorithmen gehen weit über die von reinen

[1] Die zwei recht sorgfältigen Vergleiche mathematischer Fähigkeiten einiger ausgewählter CAS [West 1995] und [West 1999] sind weithin anerkannt und die Systemhersteller trimmen ihre Produkte regelrecht auf das Bestehen dieser Tests.

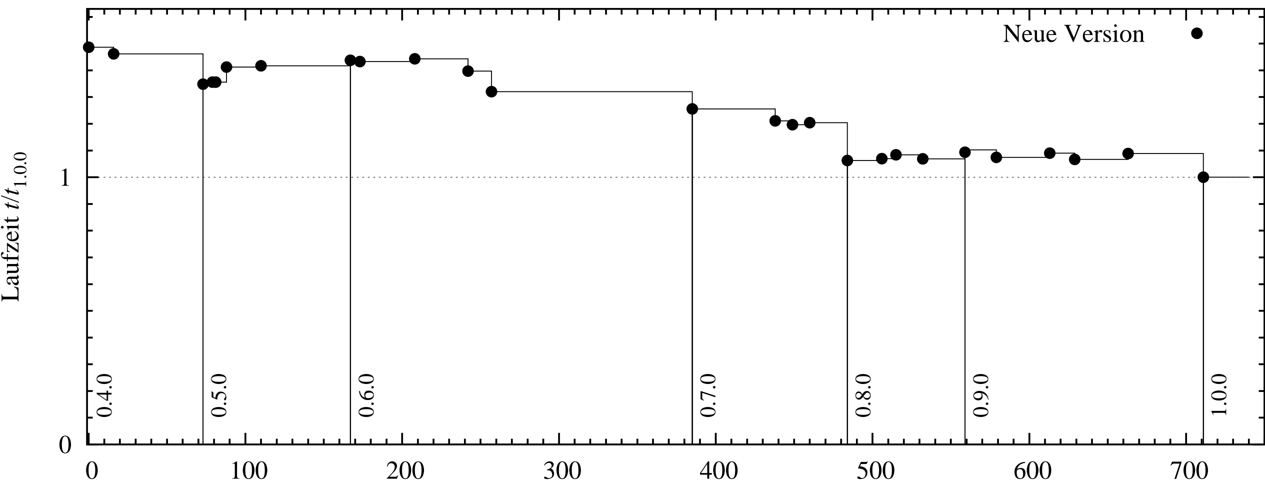


Abbildung 5.1.: *Entwicklung der Effizienz von GiNaC als Funktion der Tage nach Version 0.4.0.*

Version	Datum	Änderung
0.6.4	10.8.2000	Die Konstruktoren von `numeric` markieren das Objekt nun gleich als `expanded` in den `status_flags`, wodurch `ex::expand()` bei jeder Zahl einen virtuellen Funktionsaufruf einsparen kann.
0.7.0	15.12.2000	Die Klasse `numeric` enthält direkt ein CLN-Objekt anstatt eines Zeigers darauf. Dies spart eine Indirektion.
0.7.3	28.2.2001	Die Koeffizienten in einer `expairseq` fließen in die Berechnung der Hashwerte mit ein. Dadurch verringert sich zwar die Anzahl der Hashkollisionen, es entsteht aber zusätzlicher Aufwand bei der Berechnung.
0.8.0	24.3.2001	Da der Operator `*` nun auch für nichtkommutative Produkte stehen soll, wurden alle Operatoren intern überarbeitet. Dies führte zu optimalerem Inlining.
1.0.0	6.11.2001	Flyweights wurden statisch, also ohne zusätzlichen Funktionsaufruf.

Tabelle 5.1.: *Einige Eckpunkte in der zeitlichen Entwicklung der Effizienz aus Abbildung 5.1.*

Sortierern wie Form zur Verfügung gestellten hinaus, sind aber nicht so anspruchsvoll wie diejenigen, die man beispielsweise in Maple findet. Somit ist diese Kollektion ein idealer Prüfstein für GiNaC. Die Ergebnisse sind in Tabelle 5.2 gelistet. Es stellte sich heraus, dass der Test offensichtlich maßgeschneidert ist um das System Fermat [Lewi 1997] in ein möglichst gutes Licht zu rücken, dessen Autor Robert Lewis auch die Tests mitentworfen hat. Dies erklärt die Dominanz von manifesten Matrixeliminationstests – 6 von 34 – sowie von Matrixmanipulationen wie die Smith-Normalform, die intern auf eine Elimination herauslaufen – 21 von 34. Es erklärt vor allem die völlige Abwesenheit von Differentiation, Reihenentwicklung und numerischer Funktionsevaluation, da Fermat diese nicht beherrscht. Zudem stellte sich bei der Wiederholung der Tests heraus, dass die meisten Systeme (nicht nur in den hier verwendeten neueren Versionen) besser abschneiden als in [LeWe 1999] angegeben. Dies gilt insbesondere für Singular [GPS 2000].

Speichereffizienz ist ein Thema, welches von Computer-Algebra-Benchmarks selten bis gar nicht angesprochen wird.[2] Bei der Handhabung großer symbolischer Ausdrücke erfährt man

[2] „*We think that the user of a computer algebra system is mostly interested in good timings. The memory*

Benchmark	System:	GiNaC	MapleV	MuPAD	Pari-GP	Singular
	Version:	**1.0.0**	**R5**	**2.0.0**	**2.0.19β**	**2.0.1**
	Erscheinungsdatum:	11/2001	9/1997	5/2001	3/2000	6/2001
A:	Teile Fakultäten $\left.\frac{(1000+i)!}{(900+i)!}\right\|_{i=1}^{100}$	0.20	6.66	1.13	0.37	0.21
B:	$\sum_{i=1}^{1000} 1/i$	0.019	0.08	0.10	0.041	0.51
C:	ggT(große Zahlen)	0.25	10.2	3.01	1.65	0.31
D:	$\sum_{i=1}^{10} iyt^i/(y+it)^i$	0.68	0.13	2.20	0.20	0.11
E:	$\sum_{i=1}^{10} iyt^i/(y+\|5-i\|t)^i$	0.56	0.05	2.20	0.11	2.30
F:	ggT(bivariate Polynome)	0.07	0.08	1.36	0.057	0.09
G:	ggT(trivariate Polynome)	2.01	2.89	5.43	99.5	0.27
H:	det(Rang 80 Hilbert-Matrix)	9.12	33.5	19.5	3.97	15.9
I:	Invertiere Rang 40 Hilbert-Matrix	2.73	6.41	5.10	0.62	0.35
J:	Verifiziere I	1.52	2.28	1.90	0.22	0.02
K:	Invertiere Rang 70 Hilbert-Matrix	17.8	92.0	32.2	5.90	1.84
L:	Verifiziere K	8.69	21.6	10.6	1.57	0.058
M_1:	det(symbolische 26×26 Matrix)	0.36	0.40	0.51	0.016	0.003
M_2:	det(symbolische 101×101 Matrix)	1517.3	*GU*	*CR*	*CR*	28.2
N:	$\sum$ rationaler Funktionen vereinfachen	704.4	*GU*	*CR*	*CR*	*CR*
O_1:	Drei Rang 15 Determinanten (Mittel)	39.5	*GU*	*CR*	*CR*	53.8
O_2:	ggT der Ergebnisse aus O_1	*CR*	*UN*	*UN*	*UN*	*CR*
P:	det(Rang 101, dünn besetzte Matrix)	1.10	12.6	4.3	0.09	0.023
P':	det(Rang 101, weniger dünn besetzt)	5.61	13.3	11.3	0.38	2.85
Q:	charpoly(P)	103.1	1429.7	2751.2	0.15	0.14
Q':	charpoly(P')	209.8	1497.3	2796.1	*CR*	3.82

Tabelle 5.2.: *Vergleich symbolischer Pakete nach [LeWe 1999]. Alle Laufzeiten sind in Sekunden angegeben, als Testsystem diente ein Intel P-III 450MHz mit 384MB RAM unter Linux. Abkürzungen: GU („gave up“, wie bei Maple's* `object too large`*), CR („crashed“, meist nicht genug Speicher), UN („unable“, ein Voraussetzungstest konnte nicht durchgeführt werden).*

jedoch, dass hier gewaltige Unterschiede zwischen den Systemen bestehen, die für ein gegebenes Problem das eine oder andere System bisweilen unbrauchbar machen können. So bricht zum Beispiel der Test von Denny Fliegner aus Abbildung 3.6 für MuPAD deutlich früher ein als für die anderen Systeme und kann mit Form auf der Testmaschine sogar noch bis $n \simeq 1700$ getrieben werden statt bis $n \simeq 1200$.

Dank der konsequenten Referenzzählung kann GiNaC beim Speicherverbrauch unter Umständen um Größenordnungen besser abschneiden als andere Systeme. Anhand von Ableitungen kann man dies schön verdeutlichen. Als Beispiel berechnen wir die Euler-Zahlen mithilfe ihrer Definitionsformel und messen den Speicherbedarf. Als Euler-Zahlen E_n bezeichnet man die Entwicklungskoeffizienten des inversen Cosinus Hyperbolicus

$$\frac{1}{\cosh(x)} \equiv \sum_{n=0}^{\infty} E_n \frac{t^n}{n!}.$$

management is not of such a large interest to him besides the fact that large memory usage might influence the timings or may even crash the system.“ Wolfram Koepf in [Koe 1999].

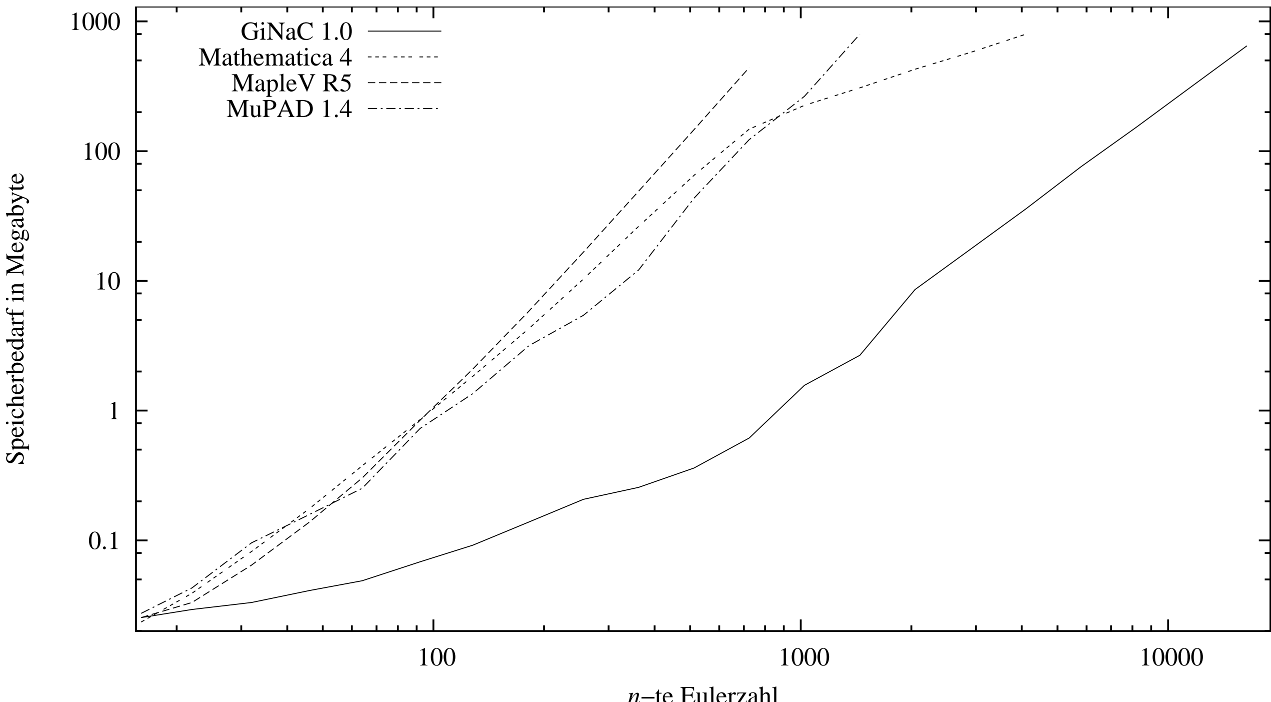


Abbildung 5.2.: *Speicherbedarf verschiedener CA-Systeme für die Berechnung von Eulerzahlen. Alle Messungen wurden auf der Architektur Intel-x86 durchgeführt.*

Das Ergebnis zeigt Abbildung 5.2. (Reduce konnte leider nicht zum Test antreten. Das Problem ist in Reduce zwar trivial formulierbar, jedoch versagt dieses System früh und meldet `CopyFromStack error`, `Binding stack overflow` und schließlich `Segmentation Fault`.) Der abgebildete Speicherbedarf in Megabytes ist derjenige vom System für die Lösung des Problems allozierte – also der Gesamtbedarf abzüglich des Speicherbedarfs um das System leer zu starten. Der Knick im von Mathematica allozierten Speicherbedarf steht möglicherweise in Zusammenhang mit dem Knie in Abbildung 4.5. Er geht mit einer dramatischen Verlangsamung der Rechengeschwindigkeit einher und könnte eine Änderung der (undokumentierten) internen Darstellung von Ausdrücken andeuten.[3] Es wäre interessant zu wissen, ob sich die Kurven von Mathematica und GiNaC jenseits von 10GB tatsächlich schneiden. (Dies kann natürlich aufgrund des auf 32 Bit beschränkten Adressraumes der x86-Architektur grundsätzlich nicht beantwortet werden.)

Abbildung 5.3 stellt dar, wie aus dem Darstellungsbaum ein gerichteter azyklischer Graph wird, indem mehrfach auftretende Ausdrücke bei der Berechnung von E_2 wiederverwertet werden und nur deren Referenzzähler erhöht wird. Ein Paket zur automatischen Berechnung von Antipoden als Renormierungs-Counterterme profitiert nicht unerheblich von solchen Einsparungen [BKK 2001].

An Programmbibliotheken werden verständlicherweise recht hohe Ansprüche an Speichersicherheit gestellt in dem Sinne, dass jeder allozierte Speicherbereich auch wieder freigegeben

[3] Angedeutet im Abschnitt „New in Version 4“ der begleitenden Dokumentation [Wolf 1999]: „*Internal packed array technology to make repetitive operations on large numerical datasets radically more efficient in speed and memory.*“ Dass eine solche Umorganisation auch Geschwindigkeitsvorteile bringen soll, ist jedoch größtenteils Wunschdenken.

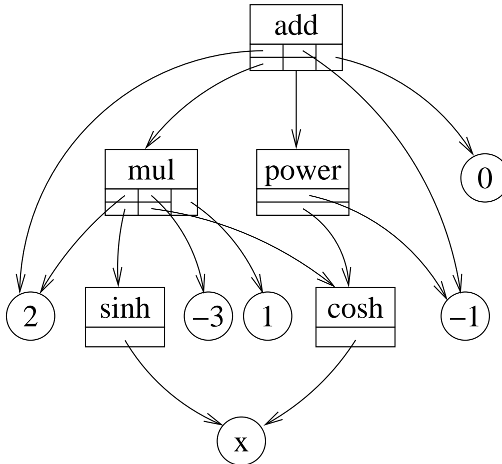


Abbildung 5.3.: *GiNaCs Darstellung von* $\frac{d^2}{dx^2}(1/\cosh(x)) = 2\sinh(x)^2/\cosh(x)^3 - 1/\cosh(x)$ *wie er bei der Berechnung der Euler-Zahl* E_2 *in Abbildung 5.2 auftritt.*

werden soll. Ein objektorientiertes Design, in dem `free` / `delete` systematisch von den Destruktoren aufgerufen wird, erleichtert es, die Übersicht zu wahren. Außerdem können Werkzeuge zum Aufspüren von Speicherlecks wertvolle Dienste leisten und sollten regelmäßig zur Anwendung kommen. Andererseits hat es sich aber auch als sehr schwierig herausgestellt, selbst große Speicherlecks überhaupt rechtzeitig zu bemerken. Die Beobachtung der Größe eines Programmes im Speicher ist *kein* verlässliches Maß. Die Abbildung 5.4 zeigt, wie der Speicherverbrauch eines Programmes, welches die quadratfreie Faktorisierung bivariater Zufallspolynome $\mathbb{Q}[x, y]$ berechnet, immer größer werden kann, obwohl eigentlich alle Destruktoren aufgerufen worden sein sollten. Die auf Seite 42 erläuterte quadratfreie Faktorisierung wurde gewählt, weil in ihr ein großer Querschnitt von Funktionen zur Anwendung kommt – wie die Berechnung von multivariaten ggTs und zum Vergleich die Ausmultiplikation großer Polynome. Die Grafik zeigt eindringlich, wie das Muster der Speicherentwicklung des mit GCC-2.95.2 kompilierten Programms alle Merkmale eines Speicherlecks aufzuweisen scheint, während derselbe Programmlauf in einem mit einer GCC-3.0 Vorabversion kompilierten Programm sich konstant einpegelt und nur noch die Merkmale leichter Heapfragmentierung aufweist. Der Grund hierfür ist außerhalb von GiNaC zu suchen. Die Implementierung der Container in der Standard Template Library zwischen diesen beiden Compilern unterscheidet sich erheblich: im älteren Compiler geben die Container ihren Speicher niemals frei, um ihn zu einem späteren Aufruf wiederverwerten zu können ohne ihn aufwändig mit `malloc` / `new` erst anfordern zu müssen. Dieses Verhalten wird seit 1995 immer wieder umworben [SGI 1995] und wird in weiten Kreisen immer noch für erstrebenswert gehalten. Kurz vor Veröffentlichung des GCC-3.0 wurde es wiederbelebt [FSF 2001, *Chapter 17: Library Introduction*]. Um mit solchen Compilern Speicherlecks aufzuspüren muss der Standardallokator der STL explizit umgestellt werden.[4]

[4] Dies kann theoretisch beim Kompilieren der Bibliothek mit der Präprozessordirektive `__USE_MALLOC` geschehen. Es hat sich aber als zuverlässiger erwiesen, dies vor dem gesamten Bootstrap-Prozess des Compilers schon in der Quelldatei `libstdc++-v3/include/bits/c++config` per `#define` vorzunehmen, da man sonst leicht unbeabsichtigt gegen die C++ Regel der „einmaligen Definition“ verstößt.

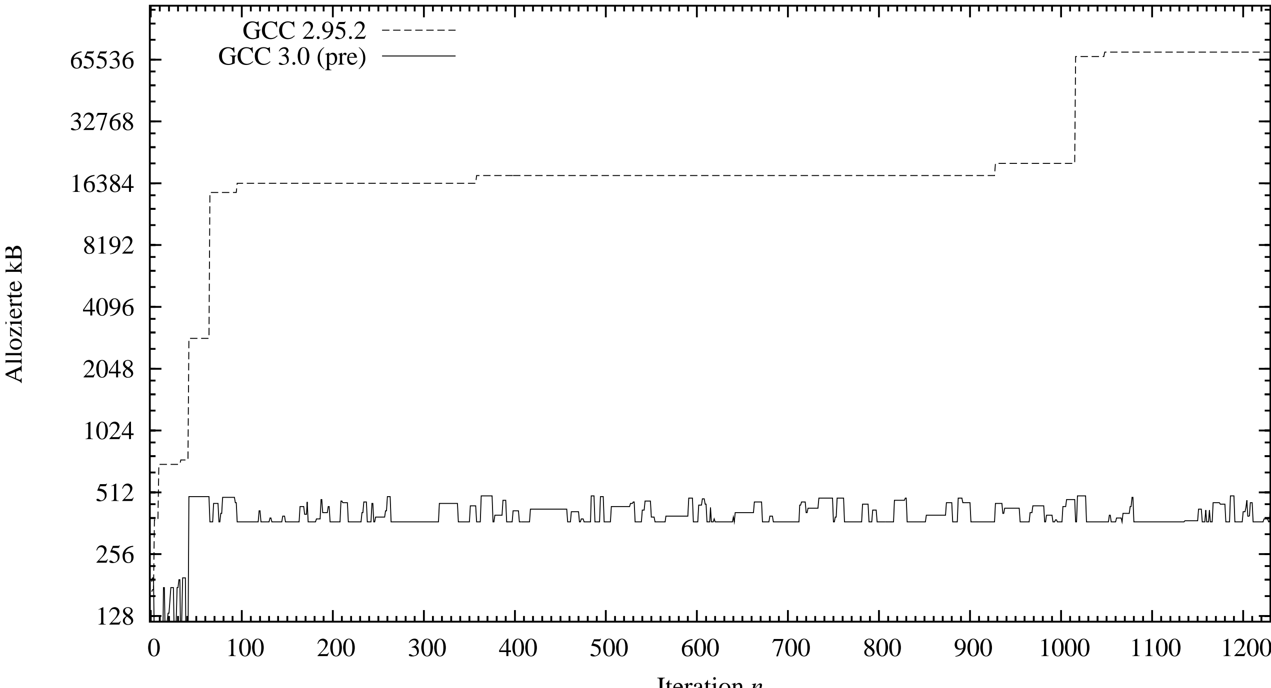


Abbildung 5.4.: *STL Template Speicherallozierung: Abhängig von der Implementierung wird ein Teil des gebrauchten Speichers auch nach dem Aufruf der Destruktoren der STL Templates nicht wieder freigegeben, sondern für spätere Verwendung aufbewahrt. Gemessen wurde der auf dem Heap belegte Speicher nach 1200 Aufrufen von* `sqrfree()` *in GiNaC 0.8.2.*

5.2. Handhabbarkeit

Das Schreiben symbolischer Programme in C++ verlangt dem Programmierer zweifelsfrei mehr Denkarbeit und Kenntnisse der zugrundeliegenden Datenstrukturen ab als in einer eigens dafür geschaffenen Sprache wie Maple. Ob dies nun als negativ oder positiv empfunden wird, ist subjektiv und muss dahingestellt bleiben. Stattdessen erörtern wir in diesem und dem nächsten Abschnitt exemplarisch ein paar besser taxierbare Gesichtspunkte.

Portabilität

Die Frage nach der Abhängigkeit von GiNaC von einem spezifischen Compiler ist berechtigt. Sobald die Entscheidung für CLN als Klassenbibliothek gefallen war, hatten wir uns zwar nicht auf Unix-Plattformen, jedoch auf den C++-Compiler aus der GNU Compiler Collection (GCC) festgelegt. Dies liegt an der Tatsache, dass ein C++Programm – oder eine Bibliothek – normalerweise in verschiedene Kompilationseinheiten zerlegt wird.[5] Wenn in diesen Einheiten nun statische Objekte zu initialisieren sind, so werden deren Konstruktoren beim Aufruf des Programmes in einer kaum kontrollierbaren Reihenfolge aufgerufen. Die Sprache garantiert nur, dass statische Objekte innerhalb eines Moduls in der Reihenfolge ihres Auftretens initialisiert werden, die Reihenfolge der Module hängt aber von Linker-Charakteristiken ab. Wenn ein Konstruktor nun aber ein anderes statisches Objekt benutzt und dieses noch nicht initialisiert

[5] CLN besteht aus ca. 850 kleinen Programmeinheiten, GiNaC aus ca. 40 mittelgroßen.

worden ist, so kommt es zu undefiniertem Verhalten. Ein idealer Linker würde einen gerichteten Abhängigkeitsgraphen erstellen und die Objekte in der korrekten Reihenfolge initialisieren.[6] In CLN Version 1.1 gibt es 39 solcher globalen Objekte. Ihre Initialisierungsreihenfolge wird mit einem halbautomatischen Trick garantiert, der Eigenschaften der GNU Binärschnittstelle ausnutzt um während der Initialisierung zwischen den einzelnen Modulen umherzuspringen und daher nicht portabel ist. Um CLN – und damit auch GiNaC und *xloops* – auf einen anderen Compiler zu portieren, muss CLN als statische Bibliothek übersetzt werden um sie unabhängig von der Initialisierung durch den dynamischen Linker zu machen. Zusätzlich müssen die Inline-Assemblerroutinen mit dem Präprozessorsymbol `NO_ASM` ausgeschaltet werden, sofern der Zielcompiler die GNU-Syntax nicht unterstützt.

Im August 2000 konnte die Portabilität von GiNaC erstmals bewiesen werden. Als Testsystem diente der C++-Compiler des Herstellers ‚Portland Group' in der Version 3.1-3. Es gelang mit minimalem Aufwand, CLN damit als statische Bibliothek zu übersetzen. Das Ergebnis war unbefriedigend langsam und algebraisch bisweilen inkorrekt, was vermutlich auf Compilerfehler zurückzuführen ist. Damit eignet sich dieses Gerät leider nur als penibler Syntaxprüfer. Als solcher konnte er jedoch GiNaC in der Version 0.6.4 problemlos übersetzen und sogar die in Tabelle 5.2 aufgelisteten Benchmarks durchführen modulo einiger Rechenfehler, die wohl auf die Kompilierfehler bei der Übersetzung von CLN zurückzuführen sind. Die Laufzeiten waren um das zweifache bis achtfache langsamer als diejenigen des GCC-Kompilates, womit sich dieser kommerzielle Compiler für die Weiterentwicklung in mittlerer Zukunft als uninteressant erwiesen haben dürfte.[7]

Im Juli 2001 wurde die Portabilität abermals getestet, diesmal anhand des KAI C++-Compilers von Intel, Version 4.0f3. Dieser Compiler ist eigentlich nur ein Code-Generator, der C++-Programme intern in optimierte C-Programme umwandelt und diese dann vom nativen Systemcompiler in Binärdateien übersetzen lässt. Die erfolgreiche Übersetzung von CLN und GiNaC setzte einige manuelle Eingriffe in die Quellcodes voraus, um ein paar leicht zu identifizierenden Compilerfehlern aus dem Wege zu gehen. Die Laufzeit war nur unwesentlich geringer als diejenige des GCC-Kompilates. Rechenfehler wurden nicht beobachtet.

Kurz darauf konnte die Portabilität auf die Version 5.0beta des Referenzcompilers für x86-Plattformen vom Hersteller Intel bestätigt werden. Die Benchmarks aus Tabelle 5.2 liefen damit wider Erwarten langsamer als mit dem GCC-Kompilat – im Durchschnitt um etwa 50%.

Zusammenfassend lässt sich sagen, dass eine Portierung des Systems CLN/GiNaC auf einen neuen Compiler durchaus durchführbar ist, was angesichts der insgesamt 140 000 Programmzeilen schon als bemerkenswert gelten dürfte. Da der Arbeitsaufwand hierfür jedoch nicht unerheblich ist und andere Compiler bislang keinerlei Vorteile gegenüber dem freien GCC erkennen ließen, sollte eine solche Portierung wohlüberlegt werden.

[6] Der GNU Linker arbeitet sie in der Reihenfolge ab, in der sie beim Zusammenlinken des Programmes auf der Kommandozeile spezifiziert wurden. Es ist fraglich, ob diese Konvention geschickt ist: Nur die Tatsache, dass es eine solche Konvention gibt, wird die spätere Umstellung auf eine automatisch nach Abhängigkeiten sortierte Reihenfolge enorm erschweren.

[7] Falls es noch einmal versucht werden sollte: der Compiler von Portland Group konvertiert nicht korrekt von `signed char` nach `unsigned char`, wenn der Schalter `-Msignextend` nicht gesetzt ist. Diese Konversion ist Voraussetzung für CLN.

Rapid Prototyping

Der für compilierte Sprachen typische Zyklus Editieren–Kompilieren–Linken–Ausführen kann während aktiver Programmentwicklung rasch zu langwierig werden. Aufgrund der aufwändigen und teils tief verschachtelten Standard-Headers und fehlenden Unterstützung für vorkompilierte Header in einigen weit verbreiteten Compilern betrifft dies C++ noch mehr als andere kompilierte Sprachen wie etwa C.

Als Lösungsversuch wurde eine Schnittstelle zum C/C++-Interpreter Cint implementiert. Cint ist unter anderem das Herzstück des am CERN entwickelten objektorientierten Datenanalysepaketes ROOT, welches sich unter Experimentalphysikern großer Beliebtheit erfreut. Es kommt in der Auswertung von LHC-Daten zum Einsatz und wird daher noch mindestens 10 Jahre weiter gepflegt und erweitert werden müssen.

Hier sei eine Beispielsitzung mit GiNaC-cint demonstriert. Beginnen wir mit ein paar Einzeilern zu Berechnung einer bekannten Taylor-Reihe:

```
1  $ ginaccint
2  Welcome to ginaccint V1.0.2 (GiNaC V1.0.3, Cint V5.15.24)
3    __,  _______  GiNaC: (C) 1999-2001 Johannes Gutenberg University Mainz,
4   (__) *       | Germany.  Cint C/C++ interpreter: (C) 1995-2001 Masaharu
5    ._) i N a C | Goto and Agilent Technologies, Japan.  This is free software
6  <-------------' with ABSOLUTELY NO WARRANTY.  For details, type `.warranty'
7  Type `.help' for help.
8
9  >> symbol v("v"), c("c");
10 >> ex gamma = 1/sqrt(1 - pow(v/c,2));
11 >> ex gamma_nr = gamma.series(v==0,8);
12 >> cout << pow(gamma_nr,-2) << endl;
13 (1+(1/2*c^(-2))*v^2+(3/8*c^(-4))*v^4+(5/16*c^(-6))*v^6+Order(v^8))^(-2)
14 >> cout << pow(gamma_nr,-2).series(v==0,8) << endl;
15 1+(-c^(-2))*v^2+Order(v^8)
```

Programmschleifen können in der Regel genau wie in einem normalen C++-Programm notiert werden

```
16 >> for (int i=0; i<20; i+=2) {
17  >     cout << bernoulli(i) << ", ";
18  > }
19 1, 1/6, -1/30, 1/42, -1/30, 5/66, -691/2730, 7/6, -3617/510, 43867/798,
```

während aufgrund einer Cint-Einschränkung Funktionen ein wenig zusätzlichen Aufwand in Form eines aktiven Kommentares erfordern:

```
20 >> //ginaccint.function
21 next expression can be a function definition
22 >> const ex EulerNumber(const unsigned n)
23  > {
24  >     const symbol xi;
25  >     const ex generator = pow(cosh(xi),-1);
26  >     return generator.diff(xi,n).subs(xi==0);
27  > }
28 creating file /tmp/ginac26197caa
29 >> EulerNumber(42);
30 Out1 = -10364622733519612119397957304745185976310201
```

Eine von interaktiven Computeralgebrasystemen inspirierte Erweiterung ist die Möglichkeit, auf die zuletzt evaluierten Ausdrücke (vom Typ `ex`) mittels der `Out`n Variablen zurückzugreifen:

```
31 >> Out1/762131;
32 Out2 = -1359952912756417481954933903061965197
```

Selbst semantisch anspruchsvolle C++-Konstrukte wie folgende Implementierung des Denny Fliegner'schen Konsistenztests sind möglich:

```
33 >> #include <sstream>
34 >> vector<symbol> a;
35 >> ex bigsum = 0;
36 >> for (int i=0; i<6; ++i) {
37  >     ostringstream buf;
38  >     buf << "a" << i << ends;
39  >     a.push_back(symbol(buf.str()));
40  >     bigsum += a[i];
41  > }
42 >> ex sbtrct = -bigsum + a[0] + a[1];
43 >> cout << pow(bigsum,2).expand().subs(a[0]==sbtrct).expand() << endl;
44 a1^2
45 >> quit;
```

Da Cint zwar dank Unterstützung durch die GNU-Readline-Bibliothek komfortabel zu bedienen, aber dennoch recht anfällig gegen Fehleingaben ist, kann die skizzierte interaktive Bedienungsart etwas umständlich sein. Elegant lassen sich aber auch kleine Skripte schreiben, die direkt aufgerufen werden können. Ein Skript, welches Test E aus Tabelle 5.2 implementiert, kann wie folgt aussehen:

```
1 #! /usr/bin/ginaccint --silent
2 symbol y("y"), t("t");
3 ex s;
4 for (int i=1; i<=10; ++i) {
5     s += i*y*pow(t,i)/pow(y + abs(5-i)*t,i);
6 }
7 cout << s.normal() << endl;
8 quit;
```

Der `--silent` Schalter unterdrückt hierbei das Eingabeecho.

Doch Cint ist kein Ersatz für einen Compiler. Er besitzt eine ganze Reihe von Einschränkungen, die ihn in der Summe nur für kleinere Aufgaben brauchbar machen. Einige davon sind durch die Natur der Interpretation bedingt, andere stellen Implementierungslücken dar, auf deren Schließung man hoffen kann und wiederum andere erscheinen auch nach längerer Betrachtung recht willkürlich. Ein paar Beispiele: Cint braucht manchmal etwas „Nachhilfe“ bei Blockbegrenzungen mit geschweiften Klammern. C++ (ebenso wie C) erlaubt es, bei Blöcken, die nur aus einer einzigen Anweisung bestehen, diese in manchen Fällen wegzulassen, wie in folgendem Beispiel:

```
1 if (condition)
2     printf("yes");
3 else
4     printf("no");
```

Dies ist jedoch prinzipiell ungeeignet für einen interaktiven Interpreter, da der Benutzer normalerweise wünscht, dass die Eingabe nach dem abgeschlossenen `if (condition) printf("yes");` sofort abgearbeitet wird. Eine nachfolgende `else`-Anweisung kann somit nicht mehr als solche rechtzeitig erkannt und bearbeitet werden. Das ist nur eine der vielen Spracheinschränkungen, die ein C++-Interpreter notgedrungen haben muss. Cint bringt leider noch eine ganze Reihe weiterer Einschränkungen mit sich. So entspricht die Variablenbindung zum Beispiel nicht dem Block-Scope, wie es von C++ gefordert wird, sondern einem Funktions-Scope. Insbesondere ist das `i` in `for (int i=0; ; )` auch nach dem Ende des `for`-Blocks noch gültig. Schlimmer noch ist, dass Cint teilweise den Regeln des dynamischen Scopes folgt. Zwar werden Variablenbindungen von aufrufenden Funktionen nicht in aufgerufenen Funktionen exponiert. Da jedoch alle interaktiven Variablendeklarationen globalen Charakter haben, können Funktionen bei ihrer Definition durchaus auf erst später vor ihrem Aufruf deklarierte Variablen zugreifen. Eine Reihe weiterer Konventionen können zu Abweichungen zwischen Cint und echten Compilern führen und den Neuling verwirren – hier sei nur noch auf die etwas bizarre Tatsache hingewiesen, dass Cint `**` als Exponentiationsoperator in Fortran-Syntax überlädt.

Modularität

GiNaC erbt von C++ alle Infrastruktur um die Aufteilung von Programmen in übersichtliche Module zu ermöglichen – mit offensichtlichen und nützlichen Vorteilen. Ein häufiges Missverständnis betrifft jedoch Symbole, die in mehreren Modulen gemeinsam verwendet werden. Die Klasse `symbol` identifiziert Symbole nicht anhand ihres Strings. Dieser dient nur zu Ausgabezwecken. Stattdessen führt jedes Symbol eine Seriennummer mit sich, und eine statische Variable in der Klasse führt Buch über die als Nächstes von einem Konstruktor zuzuweisende Seriennummer. Wenn nun in einer Header-Datei `a.h` ein `symbol x("x");` deklariert wird und sowohl `a.cc` als auch `b.cc` die Deklaration aus diesem Header benutzen, so führt dies im ausführbaren Programm zu zwei verschiedenen Symbol-Objekten mit dem gleichen Namen `x`, da sie ja zweimal initialisiert worden sind. Falls irgendwann Ausdrücke von einem Modul in das andere Modul übergeben werden, kann es im schlimmsten Fall zu nicht vereinfachten Objekten der Form `x-x` kommen oder zu der etwas überraschenden Ersetzung `x.subs(x==0)` → `x`. Man kann dieses Problem umgehen, indem man `x` in `a.h` als `extern` deklariert und in genau einer Übersetzungseinheit, z.B. in `a.cc` als `static` definiert.

Ein eleganterer Weg, sich vor dem Problem zu schützen besteht darin, die Symbole nicht statisch zu initialisieren sondern von einer Fabrik erzeugen zu lassen. Der Header `a.h` enthält dann anstelle der Definition

```
1  const symbol x("x");
```

das Idiom zum Aufruf der Fabrik

```
1  const symbol x = symbol_factory("x");
```

Die Fabrik selbst entspricht dem Muster der „Flyweight-Factory“ aus [GHJV 1995] und kann mit dem assoziativen Array `std::map<T1,T2>` aus der STL in wenigen Zeilen implementiert werden:

```
1  const symbol symbol_factory(const string &s)
2  {
3      static std::map<string, symbol> directory;
4      std::map<string, symbol>::iterator i = directory.find(s);
5      if (i!=directory.end())
6          return i->second;
7      return directory.insert(map<string, symbol>::value_type(s, symbol(s)))
8                      .first->second;
9  }
```

Die Routine speichert Strings und Symbole in `directory` und liefert das vorhandene Symbol zurück, wenn der zugehörige String schon abgespeichert ist. Ansonsten erzeugt es ein neues Symbol zu dem String, speichert beide und liefert das Symbol zurück. Die Zugriffszeit ist vermöge der Implementierung von `map` als balanciertem RB-Baum von der Ordnung $\mathcal{O}(\log(N))$.

5.3. Erweiterbarkeit

Eine interessante Frage ist, wie man ein System wie GiNaC um Formen regelbasierten Wissens erweitert. Im Sinne von Systemen die – wie Mathematica – völlig auf regelbasiertem Wissen und Mustererkennung (*Pattern Matching*) beruhen ist dies nicht denkbar. Aber: ist es überhaupt notwendig und erstrebenswert? Mustererkennung sollte eigentlich nur dann eingesetzt werden, wenn algorithmische Methoden nicht vorhanden sind.

Baumrekursion

Üblicherweise stellt man sich eine Baumrekursion als etwas vor, was an der Wurzel beginnend jeden Knoten (je nach seiner Art) über die Kinder iteriert und das Ergebnis nach einer vorgegebenen Regel verarbeitet. Die Art des zurückgegebenen Knotens muss nicht unbedingt die ursprüngliche sein – man denke beispielsweise an die Differentiation, bei der Summen zwar auf Summen abgebildet werden ($(f+g)' = f' + g'$), Produkte aber nicht auf Produkte ($(fg)' = f'g + fg'$). Dieses Bild ist jedoch nicht immer ausreichend um einen Term zu bearbeiten; bisweilen sind ausgefeiltere Strategien erforderlich.

Versuchen wir uns etwas heranzutasten, indem wir untersuchen, wie man einige häufig gebrauchte „Vereinfachungen“ zwischen trigonometrischen Funktionen einbauen könnte. Wir beschränken uns hier exemplarisch auf den Sinus und Cosinus. Beinhaltet ein Ausdruck sowohl Pseudofunktionen vom Typ `sin` als auch solche vom Typ `cos`, so kann man mithilfe der Relation $\cos(x) = \sqrt{1-\sin(x)^2}$ alle Pseudofunktionen vom Typ `cos` eliminieren. Da dies für alle möglichen Argumente des Cosinus passieren kann, muss dies mit der Klasse `wildcard` geschehen wie in folgendem Beispiel:

```
1      ex e = pow(sin(x),2)+pow(cos(x),2);
2      e = e.subs(cos(wild())==sqrt(1-pow(sin(wild()),2)));
```

Darin wird `e` zu `1` vereinfacht. Es ist anzumerken, dass die Methode `.subs()` darin rein syntaktisch arbeitet.

Allgemein muss man sich also eine Transformationsstrategie zurechtlegen. In obigem Beispiel passt die Strategie der Subsitution $\cos(x) \to \sqrt{1-\sin(x)^2}$ noch in das einfache Top-Down Modell der Evaluation. Dies muss nicht immer so sein – siehe [Fate 1999] für eine Sammlung von Beispielen, in denen das Top-Down Modell versagt.

Enthält ein Ausdruck zum Beispiel $\cos(x)$, $\cos(2x)$, $\cos(3x)$ etc. und möchte man diese alle durch $\cos(x)$ ausdrücken, so kann man aus $\sin(x+y) = \sin(x)\cos(y) + \sin(y)\cos(x)$ und $\cos(x+y) = \cos(x)\cos(y) - \sin(x)\sin(y)$ die Regeln

$$\begin{aligned} \sin(nx) &\longrightarrow \sin((n-1)x)\cos(x) + \sin(x)\cos((n-1)x) \\ \cos(nx) &\longrightarrow \cos((n-1)x)\cos(x) - \sin(x)\sin((n-1)x) \\ \sin(2x) &\longrightarrow 2\cos(x)\sin(x) \\ \cos(2x) &\longrightarrow 2\cos(x)^2 - 1 \end{aligned}$$

ableiten und rekursiv anwenden.[8] Beim Durchschreiten des Baumes sollen dann alle Vielfache des Cosinus- oder Sinus-Argumentes `x` nach diesen Regeln reduziert werden. Da der Algorithmus nach dem minimalen Argument `x` parametrisiert ist, bietet sich die Implementierung mithilfe eines Funktors (auch bekannt als funktionsartigs Objekt oder *function object*) an, der in seiner Darstellung `x` enthält. Die GiNaC-Klassen sind mit der `.map()`-Methode ausgestattet um die Baumrekursion zu erleichtern. Hierfür muss der Funktor von `map_function` abgeleitet worden sein.[9] Die Deklaration kann also wie folgt aussehen:

```
1  class sin_cos_multiple_angle_reducer : public map_function {
2      ex arg;
3  public:
4      sin_cos_multiple_angle_reducer(const ex& x) : arg(x) {}
5      ex operator()(const ex&);
6  };
```

Die Definition des Funktor-Operators `operator()` wird dann die Regeln zum Reduzieren der Argumente der in einem Ausdruck auftretenden `sin`- und `cos`-Pseudofunktionen implementieren:

```
7  ex sin_cos_multiple_angle_reducer::operator()(const ex& expr)
8  {
9      if (is_ex_the_function(expr, cos)) {
10         const ex trialdiv = normal(expr.op(0)/arg);
11         if (is_a<numeric>(trialdiv)) {
12             // Regeln: cos(n*x)==cos((n-1)*x)*cos(x)-sin((n-1)*x)*sin(x)
13             //         cos(2*n*x)==2*cos(n*x)^2-1
14             //         sin(-n*x)==-sin(n*x),  cos(-n*x)==cos(n*x)
15             const numeric n = ex_to<numeric>(trialdiv);
16             if (n.is_integer()) {
17                 if (n.is_even())
18                     return (2*pow(cos(n/2*arg),2)-1).map(*this);
```

[8] Eigentlich würden die ersten beiden Regeln ausreichen. Sie setzen aber n Rekursionsschritte voraus mit einer Verdoppelung des Ausdrucksbaumes bei jedem Schritt und werden daher rasch ineffizient. Die beiden Gleichungen des doppelten Winkels kürzen die Rekursion ab und reduzieren die Komplexität von $\mathcal{O}(n^2)$ auf $\mathcal{O}(n)$.

[9] Diese Konstruktion entspricht exakt dem „Visitor"-Muster aus [GHJV 1995].

```
19                 else
20                     return (-csgn(n-1)*sin(abs(n-1)*arg)*sin(arg)
21                             +cos(abs(n-1)*arg)*cos(arg)).map(*this);
22             }
23         }
24         return expr;
25     }
26     if (is_ex_the_function(expr, sin)) {
27         const ex trialdiv = normal(expr.op(0)/arg);
28         if (is_a<numeric>(trialdiv)) {
29             // Regeln: sin(n*x)==cos((n-1)*x)*sin(x)+sin((n-1)*x)*cos(x)
30             //         sin(2*n*x)==2*cos(n*x)*sin(n*x)
31             //         sin(-n*x)==-sin(n*x),  cos(-n*x)==cos(n*x)
32             const numeric n = ex_to<numeric>(trialdiv);
33             if (n.is_integer()) {
34                 if (n.is_even())
35                     return (2*cos(n/2*arg)*sin(n/2*arg)).map(*this);
36                 else
37                     return (+csgn(n-1)*sin(abs(n-1)*arg)*cos(arg)
38                             +cos(abs(n-1)*arg)*sin(arg)).map(*this);
39             }
40         }
41         return expr;
42     }
43     return expr.map(*this);
44 }
```

Gegeben sei nun der Ausdruck $A := \cos(3x) + 3\cos(x)$. Der obige Funktor kann nun wie folgt benutzt werden:

```
1     ex A = cos(3*x)+3*cos(x);
2     sin_cos_multiple_angle_reducer f(x);
3     A = f(A);
```

wonach A zu $2\cos(x)^3 - 2\sin(x)^2\cos(x) + 2\cos(x)$ vereinfacht worden ist. Anwenden der syntaktischen Substitution $\sin(x)^2 \longrightarrow 1 - \cos(x)^2$ und Ausmultiplizieren vereinfacht A weiter zu $4\cos(x)^3$.

Der Funktor `sin_cos_multiple_angle_reducer` hat einen Nachteil: Das minimale gemeinsame Argument – in unserem Falle `x` – muss von Hand im Konstruktor spezifiziert werden. Im Idealfall würde es den Kandidaten für eine solche Reduktion selbst herausfinden. Die Lösung besteht darin, einen zweiten Funktor zu schreiben, dessen `operator()` aus einem Ausdruck die Liste der Kandidaten extrahiert, die dann elementweise abgearbeitet werden kann. Wir beginnen mit der Definition des Funktors `sin_cos_multiple_argument_finder`. Sie enthält eine Liste `lst reduced` für alle aufgefundenen Kandidaten, eine Helfermethode `.choose_candidate(const ex&, const ex&)`, die die Vielfachheit feststellt und gegeben nx und x, $n \in \mathbb{N}$ sich für x entscheidet sowie eine Helfermethode `.reduce()`, welche Redundanzen aus der Liste der Kandidaten entfernt:

```
1 class sin_cos_multiple_argument_finder {
2     lst reduced;
3     static const ex choose_candidate(const ex&, const ex&);
4     void reduce(void);
```

```
5  public:
6      ex operator()(const ex&);
7  };
```

Beginnen wir mit der Implementierung des `operator()`. Er soll sein Argument rekursiv durchschreiten und dabei die Argumente von `cos`- und `sin`-Pseudofunktionen in der Liste `reduced` aufsammeln. Für diese Durchschreitung können wir nicht `.map(*this)` verwenden wie in `sin_cos_multiple_angle_reducer::operator()`, da diese strukurerhaltend ist und Summen auf Summen sowie Produkte auf Produkte abbildet. Stattdessen können wir aber den `operator()` explizit aufrufen:

```
8  ex sin_cos_multiple_argument_finder::operator()(const ex& expr)
9  {
10     if (is_ex_the_function(expr, cos) || is_ex_the_function(expr, sin)) {
11         return lst(expr.op(0));
12     } else {
13         ex retval;
14         for (unsigned i=0; i<expr.nops(); ++i) {
15             retval = this->operator()(expr.op(i));
16             for (unsigned i=0; i<retval.nops(); ++i)
17                 reduced.append(retval.op(i));
18         }
19     }
20     reduce();
21     return reduced;
22 }
```

Vor der Rückgabe der Liste `reduced` werden mit einem Aufruf von `.reduce()` die redundanten Elemente daraus entfernt. Eigentlich soll `reduced` keine Liste sondern eher eine Menge sein, in der kein Element doppelt vorkommt. Wir können die Mengeneigenschaft des STL-Containers `std::set` für diese Reduktion zuhilfe nehmen:

```
23 void sin_cos_multiple_argument_finder::reduce(void)
24 {
25     set<ex, ex_is_less> uniq;
26     for (unsigned i=0; i<reduced.nops(); ++i)
27         uniq.insert(reduced.op(i));
28     reduced = lst();  // Lösche diese Liste, um Platz zu schaffen.
29     for (set<ex,ex_is_less>::iterator i=uniq.begin(); i!=uniq.end(); ++i) {
30         ex candidate = *i;
31         for (set<ex,ex_is_less>::iterator j=uniq.begin(); j!=uniq.end(); ++j) {
32             ex ratio = normal((*j)/candidate);
33             if (is_a<numeric>(ratio) && ratio.info(info_flags::real))
34                 candidate = choose_candidate(*j, candidate);
35         }
36         reduced.append(candidate);
37     }
38 }
```

Der letzte fehlende Baustein ist die darin aufgerufene statische Methode zur Auswahl des Kandidaten, `.choose_candidate(const ex&, const ex&)`. Da sie nur aufgerufen wird, wenn festgestellt worden ist, dass ihre Argumente Vielfache voneinander sind, kann sie als Voraussetzung annehmen, dass für ihre Argumente A und B die Brüche $A/\operatorname{ggT}(A,B)$ und $B/\operatorname{ggT}(A,B)$

nicht symbolisch sind. Die nachfolgende Implementierung vermag auch festzustellen, dass x und $\frac{3}{2}x$ Vielfache von $\frac{1}{2}x$ sind:

```
39 const ex sin_cos_multiple_argument_finder::choose_candidate(const ex& A, const ex& B)
40 {
41     const ex gcd_AB = gcd(A, B);
42     const numeric a = ex_to<numeric>(normal(A/gcd_AB));
43     const numeric b = ex_to<numeric>(normal(B/gcd_AB));
44     const numeric denoms_lcm = lcm(a.denom(),b.denom());
45     return gcd_AB * gcd(a*denoms_lcm, b*denoms_lcm) / denoms_lcm;
46 }
```

Gegeben sei nun $A := 2\sin(1)^2+\sin(5x)\cos(2x)-\sin(2x)\cos(5x)+\sin(x)-4\sin(x)\cos(x)^2+\cos(2)-1$. Wir können nun zunächst eine Liste der Reduktionskandidaten anlegen und danach den alten `sin_cos_multiple_angle_reducer` darauf wirken lassen:

```
1     sin_cos_multiple_argument_finder f1;
2     ex arglst = f1(A);
3     for (int i=0; i<arglst.nops(); ++i) {
4         sin_cos_multiple_angle_reducer f2(arglst.op(i));
5         A = f2(A);
6     }
```

Nach syntaktischer Substitution von $\cos(z) \longrightarrow \sqrt{1-\sin(z)^2}$ und Ausmultiplizierung des Ergebnisses vereinfacht `A` zu `0`.

Diese beiden Beispiele sollen gezeigt haben, wie Funktoren als parametrisierte Funktionen benutzt werden können um aus einem beliebigen algebraischen Ausdruck einen Zustand zu extrahieren und Termumformungen vornehmen zu können. Hierzu ist kein Eingriff in die Klassenhierarchie notwendig und die Funktoren können so arrangiert werden, dass sie einen Darstellungsbaum in beliebiger Weise durchschreiten – nicht unbedingt top-down wie dies die Methodendelegation in einer Klassenhierarchie erzwingt. Sind die Funktoren einmal kompiliert, so ist all dies auch zur Laufzeit denkbar und so kann eine Sammlung davon einen raffinierten Ersatz für benannte ⇒ Simplifier liefern.

5.4. Schlussfolgerungen und Ausblick

Bei der Implementierung symbolischer Algorithmen zwingt GiNaC den Benutzer zum Nachdenken über die verwendeten Datenstrukturen und anzuwendenden Algorithmen. In dieser Hinsicht ist es mit FORM vergleichbar, dem Lastesel der theoretischen Hochenergiephysik.[10] Im Gegensatz zu FORM wurden jedoch keine Anstrengungen unternommen, den verfügbaren Arbeitsspeicher mittels Swap-Dateien über den physikalisch Vorhandenen auszudehnen – GiNaC ist hier dem VM-Subsystem des zugrundeliegenden Betriebssystems ausgeliefert. Dafür stellt es dort einige algebraische Fähigkeiten zur Verfügung wo FORM sich auf die Ringoperationen `+`, `-` und `*` beschränkt.

[10] J. A. M. Vermaseren pflegt das Motto „*The user should do the thinking, the computer the computing*“ und auf G. J. v. Oldenborgh scheint das Gleichnis zurückzugehen MACSYMA, Maple und Mathematica als „*swiss army knife*“ zu bezeichnen, während FORM ein „*chef knife*“ sei [Olde 1995].

Übersicht GiNaC-basierter Projekte

Eine kleine Anzahl weiterer Projekte sind von GiNaC inspiriert worden. Hier folgt ein kurzer aktueller Schnappschuss der „Spin-Offs" soweit sie mir bekannt sind:

giac *von Bernard Parisse. Bei diesem ambitionierten Projekt handelt es sich um eine Sammlung kleiner ausführbarer Programme, jedes einzelne zuständig für eine symbolische Operation:* `factor`, `series`, *etc. Ursprünglich waren sie Frontends zur GiNaC-Bibliothek, neuere Versionen scheinen jedoch nur noch auf GMP zu basieren. Ziel ist eine innerhalb der Unix-Shell mit Pipes verknüpfte Benutzung. Siehe* `ftp://fourier.ujf-grenoble.fr/pub/hp48/giac.tgz`. *Seit kurzem gibt es dort sogar ein Frontend für das X-Window System.*

gTybalt *von Stefan Weinzierl ist eine Art Meta-Framework, welches innerhalb einer von Cint gesteuerten Umgebung die graphischen Fähigkeiten von ROOT, die TEX-Ausgabe von TeXmacs und die symbolischen Fähigkeiten von GiNaC vereinigt. Siehe* `http://www.fis.unipr.it/~stefanw/gtybalt.html`. *Es ist analog zu der Schnittstelle von TeXmacs zu Maxima, MuPAD und Reduce [Groz 2001]. In diesem Rahmen wurde auch ein Programm zur symbolischen Reihenentwicklung einiger spezieller höherertranszendenten Funktionen, wie sie in Schleifenrechnungen benötigt werden, entwickelt [Wei 2002].*

Octave-Modul *von Ben Sapp. Octave ist eine (an MATLAB angelehnte) interaktive Sprache zum Lösen von numerischen linearen und nichtlinearen Problemen. Ein Octave-Modul zur symbolischen Erweiterung basiert auf GiNaC. Siehe* `http://bsoctave.sourceforge.net/`.

pyginac *von Pearu Peterson reicht in Python (eine interpretierte, interaktive und objektorientierte Sprache, in etwa vergleichbar mit Perl, Scheme oder Java) formulierte symbolische Ausdrücke an die kompilierte GiNaC-Bibliothek weiter. Siehe* `http://cens.ioc.ee/projects/pyginac/`.

Purrs *ist eine an der Universität Parma von Roberto Bagnara und Mitarbeitern entwickelte Software zur automatisierten Analyse der Komplexität von Programmen. Der Teil der Aufgabe, für den GiNaC herangezogen wird, betrifft das automatisierte Lösen von symbolischen Rekursionsrelationen. Siehe* `http://www.cs.unipr.it/purrs/`.

Ob das Programmieren in C++, das häufig als „schwer" empfunden wird, letztlich einer Akzeptanz von Seiten der Physiker im Wege steht, muss dahingestellt bleiben. Im Gegensatz zu herkömmlichen Computeralgebrasystemen trennt GiNaC zwischen Variablen (Klasse `ex`) und Symbolen (Klasse `symbol`). Zuweisungen von Ausdrücken an Symbole finden nicht statt, stattdessen können Variable mit der Methode `.subs()` ersetzt werden.[11] Diese Trennung hat sich in der Praxis bisher nie als hinderlich herausgestellt – im Gegenteil, sie eliminert das Fehlerpotenzial, das sich aus vergessenen Variablenbindungen ergibt.

Einige Projekte basieren schon jetzt auf GiNaC (siehe Kasten). Persönlich hoffe ich, dass der Stil der Programmierung sich vom rein Prozeduralen loslösen wird und *xloops* in Zukunft immer mehr aus mehreren kleinen Modulen mit wohldefinierten Funktionalitäten besteht,

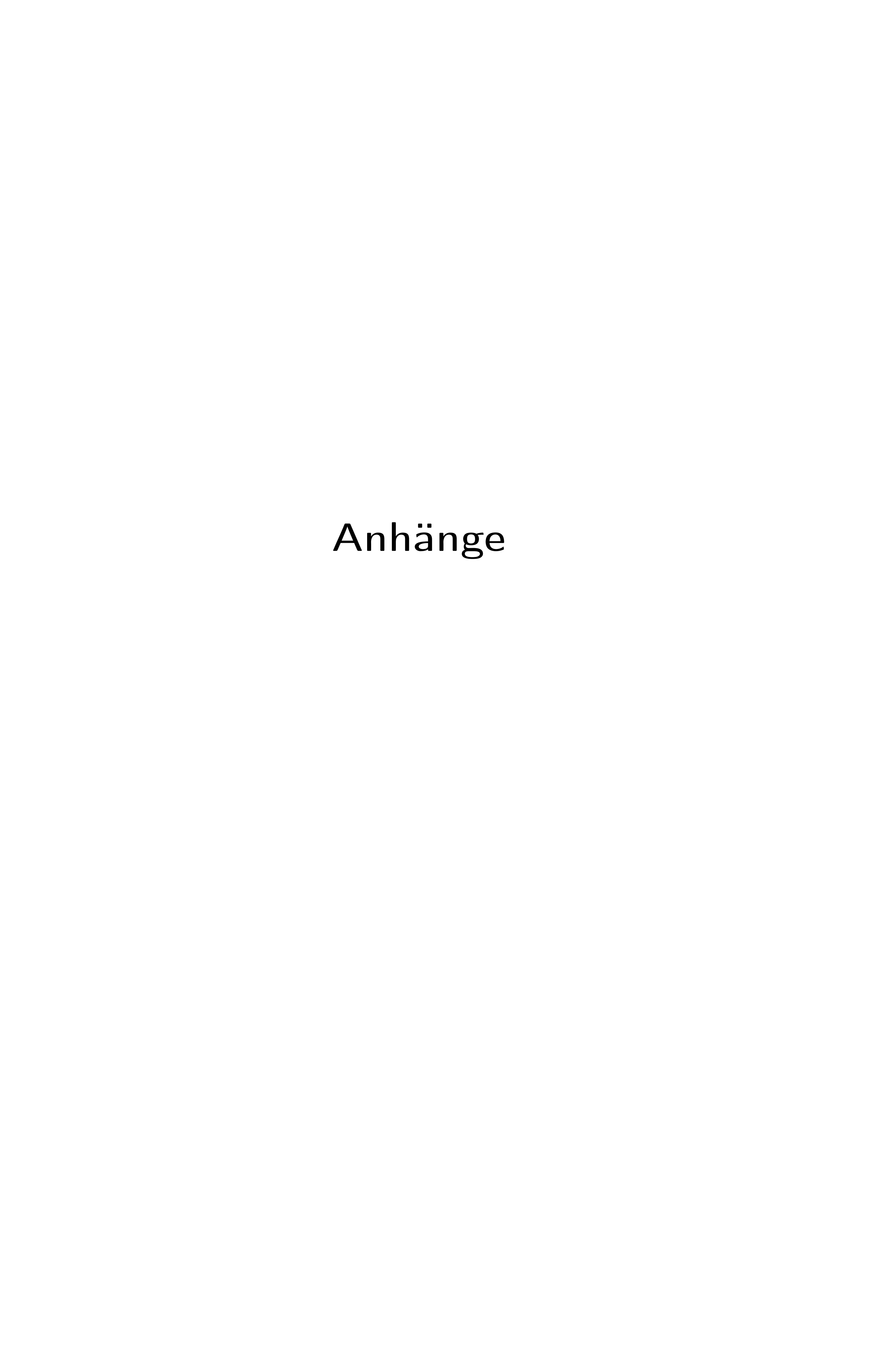
[11] Benutzern von FORM ist diese Denkweise vertraut: Die `id`-Anweisung ist auch weniger Zuweisung als Ersetzung und die Unterscheidung zwischen `Local` und `symbol` entspricht exakt unserer zwischen `ex` und `symbol`.

unter Ausnutzung eleganter sprachlicher Konstrukte wie Funktor-Klassen sowie der Standard Template Library. Ich bin überzeugt davon, dass die Wartbarkeit von *xloops* zunimmt, je weniger monolithisch es wird.

Algebraischer Ansatz?

Personen mit rigoroser mathematischer Ausbildung fordern von Computeralgebrasystemen immer wieder eine präzise Abbildung algebraischer Strukturen auf darstellende Datenstrukturen. Das System Axiom [JeSu 1992] versucht diesem Ziel nahe zu kommen. Vorsicht ist jedoch angebracht, wenn verschiedene Strukturen zu irgendeinem Zeitpunkt aufeinandertreffen. Wie unklar das Ergebnis dann schnell wird zeigt ein einfaches Beispiel. Sei $P = x+2 \in \mathbb{Z}[x]$ ein Polynom mit ganzzahligen Koeffizienten in x. $P/3$ kann dann als $\frac{x+2}{3}$ aufgefasst werden, also als Quotient zweier ganzzahliger Polynome in x. Oder – kaum weniger plausibel – als $P/3 = \frac{1}{3}x+\frac{2}{3}$, als Element des Ringes $\mathbb{Q}[x]$ der Polynome in x mit gebrochenrationalen Koeffizienten. Für die automatische Weiterverarbeitung kann dieser Unterschied von erheblicher Bedeutung sein. Ein anderes Beispiel, in dem ein rigoroser algebraischer Ansatz schnell unpragmatisch wird, ist die vermeintliche Vereinfachung von $(x^{101}-1)/(x-1)$ zu $x^{100}+x^{99}+\cdots+x+1$. Siehe [Dave 2000] für einen Übersichtsartikel zu den Vor- und Nachteilen eines solchen Ansatzes.

Etwas ferner von den derzeit absehbaren Bedürfnissen, aber dennoch vorstellbar, sind: Für die Weiterentwicklung und Implementierung gewisser Algorithmen (Faktorisierung, symbolische Integration) in GiNaC kann eventuell eine Klasse zur Darstellung von algebraischen Zahlen, die sich nicht in einfachen Wurzeln ausdrücken lassen (siehe Seite 80) unabdingbar werden. Eine solche Klasse `root` könnte analog zu `power` direkt von `basic` abgeleitet sein und ein Polynom in einer Standardvariablen ρ enthalten. Der Versuch, $x^5 - x + 1 = 0$ nach x aufzulösen, könnte dann ein Objekt `root(ρ^5-ρ+1)` erzeugen, welches bei einer Weiterverarbeitung eventuell entsprechend vereinfacht werden kann, z.B. `root(ρ^5-ρ+1)^5` → `root(ρ^5-ρ+1)-1`. Für eine eventuelle Implementierung symbolischer Integration (Risch-Algorithmus) müssen zunächst geeignete Darstellungen des Integranden in einem Differenzialkörper gefunden werden. Hierzu müssten transzendente Erweiterungen des rationalen Funktionenkörpers erlaubt werden um so etwa den Tangens t durch seine Ableitung $Dt = t^2 + 1$ darzustellen. Eine Einführung bietet [Bron 1996a]. Ob dies ohne grundlegende Änderungen vonstatten gehen kann, sei dahingestellt. Jedenfalls scheint dies für die derzeitige Praxis der Schleifenrechnungen nicht erforderlich zu sein.



Anhänge

A. Hilfsmittel aus der komplexen Analysis

A few weeks of developing and testing
can save a whole afternoon in the library
anonymous

Im Folgenden notieren wir für komplexe Variablen häufig $z = x + iy$ und $\zeta = \xi + i\eta$. Bisweilen werden wir für komplexe Funktionen $f(\zeta)$ auch Real- und Imaginärteil ausschreiben als $f = u + iv$ und Ableitungen abkürzend mit $f_\zeta \equiv \frac{\partial f}{\partial \zeta}$ notieren.

A.1. Der Cauchy'sche Residuensatz in einer Veränderlichen

Satz A.1 (Cauchy'scher Integralsatz) *Sei $f : \Omega \to \mathbb{C}$ analytisch in einem einfach zusammenhängenden Gebiet $\Omega \subseteq \mathbb{C}$. Sei $\mathscr{C}$ eine einfach geschlossene, positiv orientierte Jordan-Kurve in Ω. Dann gilt:*

$$\oint_{\mathscr{C}} f(\zeta) d\zeta = 0.$$

Dieser Satz lässt sich leicht ganz „zu Fuß“ beweisen, indem man die Bahnkurve $\mathscr{C}$ parametrisiert ($\oint_{\mathscr{C}} f(\zeta)d\zeta = \int_0^1 f(\zeta(\tau))\frac{\partial \zeta}{\partial \tau} d\tau$) und bemerkt, dass die Vektorfelder $(u, -v)$ und (v, u) wegen den Cauchy-Riemann'schen Differenzialgleichungen die Integrabilitätsbedingung erfüllen. □

Triviale Folgerungen aus diesem fundamentalen Satz sind die Wegunabhängigkeit von Kurvenintegralen analytischer Funktionen sowie der Deformationssatz, demzufolge ein Integral entlang einer geschlossene Kurve $\mathscr{C}$ über einen Integranden, der analytisch ist bis auf einen Punkt z in $\mathscr{C}$, ersetzt werden kann durch ein Integral entlang eines Kreises um z innerhalb von $\mathscr{C}$. Es sei an dieser Stelle noch angemerkt, dass im Spezialfall kreisförmiger Bahnkurven der Satz A.1 in den Mittelwertsatz von Gauß für harmonische Funktionen übergeht, da Real- und Imaginärteil von $f(\zeta)$ vermöge der Cauchy-Riemann'schen Differenzialgleichungen harmonische Funktionen sind

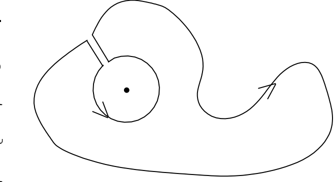


Abbildung A.1.: *Zum Deformationssatz*

$$u_{\xi\xi} = -v_{\eta\xi} = -v_{\xi\eta} = -u_{\eta\eta}$$
$$v_{\xi\xi} = -u_{\xi\eta} = -u_{\eta\xi} = -v_{\eta\eta}$$

und als durch ζ parametrisierte Fläche betrachtet daher Minimalflächen mit verschwindender Krümmung darstellen.

Satz A.1 ist auch Ausgangspunkt für das

Lemma A.2 (Cauchy'sche Integralformel) *Sei $f(\zeta)$ analytisch in einem einfach zusammenhängenden Gebiet Ω und $\mathscr{C}$ eine einfach geschlossene, positiv orientierte Jordan-Kurve, z ein Punkt in $\mathscr{C}$. Dann gilt:*

$$\oint_{\mathscr{C}} \frac{f(\zeta)}{\zeta - z} d\zeta = 2\pi i f(z).$$

Deformation der Kurve $\mathscr{C}$ in einen Kreis $S_\varepsilon(z)$ mit Radius ε um z liefert

$$\oint_{\mathscr{C}} \frac{f(\zeta)}{\zeta - z} d\zeta = \oint_{S_\varepsilon(z)} \frac{f(z)}{\zeta - z} d\zeta + \oint_{S_\varepsilon(z)} \frac{f(\zeta) - f(z)}{\zeta - z} d\zeta,$$

wobei der erste Term wegen der Parametrisierung $\oint_{S_\varepsilon(z)} \frac{1}{\zeta - z} d\zeta = \int_0^{2\pi} \frac{\varepsilon i e^{i\tau}}{\varepsilon e^{i\tau}} d\tau$ identisch mit $2\pi i f(z)$ ist und der zweite Term aufgrund der Stetigkeit von f verschwindet. □

Satz A.3 (Cauchy'scher Residuensatz) *Sei $f(\zeta)$ analytisch bis auf isolierte Singularitäten in einem einfach zusammenhängenden Gebiet Ω, $\mathscr{C}$ eine einfach geschlossene, positiv orientierte Jordan-Kurve, die nicht durch eine der Singularitäten führt und in deren Innerem $n < \infty$ Singularitäten bei $\zeta = z_i$, $i \in I$ liegen. Dann gilt:*

$$\boxed{\oint_{\mathscr{C}} f(\zeta) d\zeta = 2\pi i \sum_{i \in I} \operatorname*{Res}_{\zeta = z_i} f(\zeta).}$$

Zum Beweis entwickle man die Funktion f um die Singularitäten in eine Laurentreihe $f(\zeta) = \sum_{n=-\infty}^{\infty} a_n^{(i)} (\zeta - z_i)^n$ und bezeichne mit $p^{(i)}(\zeta) = \sum_{n=-1}^{-\infty} a_n^{(i)} (\zeta - z_i)^n$ den Hauptteil bei z_i. Dann ist $g(\zeta) := f(\zeta) - \sum_i p^{(i)}(\zeta)$ analytisch in Ω; also gilt $\oint_{\mathscr{C}} g(\zeta) d\zeta = 0$ und es genügt, die Summe der Hauptteile $p^{(i)}$ zu integrieren:

$$\oint_{\mathscr{C}} f(\zeta) d\zeta = \oint_{\mathscr{C}} \sum_{i \in I} p^{(i)}(\zeta) d\zeta = \sum_{i \in I} \oint_{\mathscr{C}} p^{(i)}(\zeta) d\zeta.$$

Die Vertauschung von Integration und Summation ist möglich wegen der Endlichkeit der Indexmenge I. Nun kann man jeden Hauptteil $p^{(i)}$ getrennt integrieren, wobei in $\oint_{\mathscr{C}} \sum_{n=-1}^{-\infty} a_n^{(i)}$ lediglich die Terme mit $n = -1$ entsprechend Lemma A.2 beitragen und die übrigen verschwinden, was man wieder anhand einer Parametrisierung einsieht: $\oint_{S_\varepsilon(z)} \frac{1}{(\zeta - z)^n} d\zeta = \int_0^{2\pi} \frac{\varepsilon i e^{i\tau}}{\varepsilon e^{in\tau}} d\tau = 0$ falls $n \neq 1$. □

Satz A.4 (über die Residuensumme) *Ist $f(\zeta)$ in der geschlossenen Ebene bis auf isolierte Stellen eindeutig und holomorph, so verschwindet die Summe der Residuen von $f(\zeta)$.*

Der Beweis folgt [BeSo 1965]: Sei $\mathscr{C}$ irgendein einfach geschlossener endlicher Weg, auf dem keine Singularitäten liegen. Man kann $\frac{1}{2\pi i}\oint_{\mathscr{C}} f(\zeta)\,d\zeta$ auf zwei Weisen ausrechnen: indem man $f(\zeta)$ im Inneren oder im Äußeren betrachtet. Wählt man nun $\mathscr{C}$ so, dass im Inneren keine Singularitäten von $f(\zeta)$ liegen (was nach Vssg. möglich ist), und durchläuft man $\mathscr{C}$ derart, dass das Äußere zur Linken liegt, so folgt die Behauptung des Satzes:[1]

$$0 = \frac{1}{2\pi i}\oint_{\mathscr{C}} f(\zeta)\,d\zeta = \sum_i \operatorname*{Res}_{\zeta=z_i} f(\zeta).$$

□

Man kann also in der Summe ein (beliebiges) der resultierenden Residuen ausdrücken durch die negative Summe aller übrigen. Angewendet auf Integrale vom Typ $\int_{-\infty}^{+\infty} d\zeta \prod_{i=1}^{n} \frac{1}{P_i(\zeta)}$ mit $P_i(\zeta) = \zeta - z_i$, $z_i \in \mathbb{C}$ bleiben also nur noch die θ-Funktionen übrig. Dies ist die Aussage des folgenden Satzes.

Korollar A.5 (zum Satz über die Residuensumme) *Sei $P_j(\zeta) = \zeta - z_j$ mit $z_j = x_j + i\,y_j$, $x_j,\, y_j \in \mathbb{R}$ und $y_j \neq 0\ \forall j$, wobei $j \in \{1 \dots n\}$, $n \geq 2$. Außerdem seien alle z_j paarweise verschieden. Dann ist*

$$\frac{1}{2\pi i}\int_{-\infty}^{+\infty} d\zeta\, \frac{1}{P_1(\zeta)\,P_2(\zeta)} = \frac{\theta(y_1) - \theta(y_2)}{P_2(z_1)} \tag{A.1}$$

$$\frac{1}{2\pi i}\int_{-\infty}^{+\infty} d\zeta\, \frac{1}{P_1(\zeta)\,P_2(\zeta)\,P_3(\zeta)} = \frac{\theta(y_1) - \theta(y_3)}{P_2(z_1)\,P_3(z_1)} + \frac{\theta(y_2) - \theta(y_3)}{P_1(z_2)\,P_3(z_2)} \tag{A.2}$$

$$\vdots$$

$$\frac{1}{2\pi i}\int_{-\infty}^{+\infty} d\zeta \prod_{j=1}^{n} \frac{1}{P_j(\zeta)} = \sum_{j=1}^{n-1} \frac{\theta(y_j) - \theta(y_n)}{\prod_{i\neq j} P_i(z_j)}. \tag{A.3}$$

Anmerkung: Die obigen Formeln entsprechen einem in der oberen Halbebene geschlossenen Integrationsweg. Ersetzt man in (A.3) $\theta(y_j) = 1 - \theta(-y_j)$, so erhält man

$$\sum_{j=1}^{n-1} \frac{\theta(y_j) - \theta(y_n)}{\prod_{i\neq j} P_i(z_j)} = -\sum_{j=1}^{n-1} \frac{\theta(-y_j) - \theta(-y_n)}{\prod_{i\neq j} P_i(z_j)}. \tag{A.4}$$

Die rechte Seite kann als die entsprechende Gleichung für einen in der unteren Halbebene geschlossenen Integrationsweg interpretiert werden: Die Vorzeichen der Argumente der θ-Funktionen spiegeln die Teilmenge der Polstellen in dieser Halbebene und das allgemeine Vorzeichen den Integrationsweg mit negativem Umlaufsinn wieder. Der Satz über die Residuensumme (bzw. die daraus abgeleitete Gleichung (A.3)) macht also die Freiheit in der Wahl des Integrationsweges manifest.

[1] Mögliche Residuen im Unendlichen müssen mitberücksichtigt werden, da der Satz sich auf die *geschlossene* Ebene bezieht.

Anhand von Korollar A.5 lässt sich auch einsehen, auf welche Weise Residuensatz und Partialbruchzerlegung bei rationalen Funktionen miteinander zusammenhängen.[2] Nach dem Fundamentalsatz der Algebra lässt sich der Nenner einer rationalen Funktion immer in der Form $1/\prod_{j=1}^{n} P_j(\zeta)$ schreiben. Klammert man $1/P_n(\zeta)$ aus und zerlegt die übrigen sukzessive als Partialbruch, so bleiben $n-1$ Summanden, deren Nennerpolynom quadratisch in ζ ist, und daher noch schnell genug abfällt, um integrierbar zu sein:

$$\frac{1}{P_1(\zeta)P_2(\zeta)P_3(\zeta)} = \left(\frac{1}{P_1(\zeta)\,(z_1-z_2)} + \frac{1}{P_2(\zeta)\,(z_2-z_1)}\right)\frac{1}{P_3(\zeta)} \tag{A.5}$$

$$\frac{1}{P_1(\zeta)P_2(\zeta)P_3(\zeta)P_4(\zeta)} = \left(\frac{1}{P_1(\zeta)\,(z_1-z_2)\,(z_1-z_3)} + \frac{1}{P_2(\zeta)\,(z_2-z_1)\,(z_2-z_3)} + \frac{1}{P_3(\zeta)\,(z_3-z_1)\,(z_3-z_2)}\right)\frac{1}{P_4(\zeta)} \tag{A.6}$$

$$\vdots$$

$$\prod_{j=1}^{n}\frac{1}{P_j(\zeta)} = \left(\sum_{j=1}^{n-1}\frac{1}{P_j(\zeta)}\prod_{i\neq j}\frac{1}{(z_j-z_i)}\right)\frac{1}{P_n(\zeta)}. \tag{A.7}$$

Nun kann man in der Summe gliedweise die Integration über die reelle Achse ausführen und dabei unter Benutzung von (A.1) die Gleichung (A.3) erzeugen, da $P_i(z_j) = z_j - z_i$ ist.

Falls $z_i = z_j$ für mindestens ein Paar $i \neq j$ gilt dieser Zusammenhang übrigens noch genauso. Dann liegt bei $\zeta = z_j$ ein k-facher Pol vor und für die Berechnung des Residuums kann man die Gleichung $\mathrm{Res}_{\zeta=z_j}\frac{1}{P_1(\zeta)\cdots P_n(\zeta)} = \frac{1}{(k-1)!}\left(\left(\frac{d}{d\zeta}\right)^{k-1}(\zeta-z_j)^k\frac{1}{P_1(\zeta)\cdots P_n(\zeta)}\right)\Big|_{\zeta=z_j}$ heranziehen. Ist beispielsweise ein doppelter Pol bei z_n, so findet man in Analogie zu Gleichungen (A.1)-(A.3)

$$\frac{1}{2\pi i}\int_{-\infty}^{+\infty} d\zeta\,\frac{1}{P_1(\zeta)\,P_2(\zeta)^2} = \frac{\theta(y_1)-\theta(y_2)}{P_2(z_1)^2} \tag{A.8}$$

$$\vdots$$

$$\frac{1}{2\pi i}\int_{-\infty}^{+\infty} d\zeta\frac{1}{P_n(\zeta)}\prod_{j=1}^{n}\frac{1}{P_j(\zeta)} = \sum_{j=1}^{n-1}\frac{\theta(y_j)-\theta(y_n)}{P_n(z_j)\prod_{i\neq j}P_i(z_j)}, \tag{A.9}$$

wobei schon der Satz A.4 über die Residuensumme ausgenutzt worden ist. Andererseits bricht natürlich die naive Partialbruchzerlegung nach dem Muster $\frac{1}{P_1P_1} = \frac{1}{P_1(P_1-P_1)} + \frac{1}{P_1(P_1-P_1)}$ zusammen, $P_n(\zeta)^2$ darf also nur als ganzes behandelt werden.[3] Jedenfalls ermöglicht die folgende

[2] Eine anregende historische Zusammenfassung dieser im 19ten Jahrhundert von Sylvester untersuchten Parallelen geben Bhatnagar in [Bhat 1996] und Knuth [Knu 1997, *Abschnitt 1.2.3, Übung 33*].

[3] Die notwendige und hinreichende Bedingung dafür, dass eine Partialbruchzerlegung von $1/PQ$ durchgeführt werden kann, ist, dass P und Q koprim sind. Man sieht das sofort ein, wenn man Partialbruchzerlegung als Umkehrung der Addition von Brüchen auffasst, da $1/PQ$ ja zerlegt wird in $a/P + b/Q$. Sind P und Q nicht koprim, so ist der Nenner von $a/P + b/Q$, der ja das $\mathrm{kgV}(P, Q)$ ist, ungleich PQ und umgekehrt. Der programmatische Weg zur Berechnung von a und b besteht daher darin, sie als Kofaktoren der Bézout-Identität $aQ + bP = 1$ zu verstehen und mit dem beispielsweise in [Baue 2000] beschriebenen und implementierten erweiterten euklidischen Algorithmus zu bestimmen.

Partialbruchzerlegung

$$\begin{aligned}\frac{1}{P_1(\zeta)P_2(\zeta)P_3(\zeta)^2} &= \left(\frac{1}{P_1(\zeta)\,(z_1-z_2)}+\frac{1}{P_2(\zeta)\,(z_2-z_1)}\right)\frac{1}{P_3(\zeta)^2}\\ &\vdots\\ \frac{1}{P_n(\zeta)}\prod_{j=1}^{n}\frac{1}{P_j(\zeta)} &= \left(\sum_{j=1}^{n-1}\frac{1}{P_j(\zeta)P_n(\zeta)}\prod_{i\neq j}\frac{1}{(z_j-z_i)}\right)\frac{1}{P_n(\zeta)},\end{aligned}$$

die Gleichung (A.9) aus (A.8) direkt zu gewinnen.

A.2. Hauptwertintegrale

Wenn das Integral $\int_{\mathscr{C}} f(\zeta)d\zeta$ über eine Polstelle von $f(\zeta)$ führt, so ist das Riemann'sche Integral nicht definiert. Existiert aber der Grenzwert $\lim_{\varepsilon\to 0}\int_{\mathscr{C}\setminus\mathscr{C}^{(\varepsilon)}} f(\zeta)d\zeta$ für eine symmetrische ε-Umgebung $\mathscr{C}^{(\varepsilon)}$ um z_i auf dem Integrationsweg $\mathscr{C}$, so kann man das Integral darüber definieren. Man nennt dies das Hauptwertintegral und wir schreiben es wie folgt (P.V. für engl. *principal value*):

$$\text{P.V.}\int f(\zeta)d\zeta.$$

Mithilfe dieses Integrals kann man den Cauchy'schen Residuensatz retten. Hierzu untersucht man einen Pol bei z_i auf dem geschlossenen Integrationsweg $\mathscr{C}$. Zerlegt man $\mathscr{C}$ wie in Abbildung A.2 skizziert in zwei Anteile $\mathscr{C}_{\varepsilon,z_i}$ und γ_{ε,z_i}, so kann man das Hauptwertintegral schreiben als Grenzübergang:

$$\text{P.V.}\oint_{\mathscr{C}} f(\zeta)d\zeta = \lim_{\varepsilon\to 0}\left(\oint_{\mathscr{C}_{\varepsilon,z_i}} f(\zeta)d\zeta - \int_{\gamma_{\varepsilon,z_i}} f(\zeta)d\zeta\right).$$

Das erste Integral ist unabhängig von ε und lässt sich mit dem gewöhnlichen Residuensatz berechnen. Das zweite Integral wird für $\varepsilon\to 0$ durch den Hauptteil $p^{(i)}(\zeta)$ von $f(\zeta)$ an $\zeta = z_i$ bestimmt:

$$\lim_{\varepsilon\to 0}\int_{\gamma_{\varepsilon,z_i}} f(\zeta)d\zeta = \lim_{\varepsilon\to 0}\int_{\gamma_{\varepsilon,z_i}} p^{(i)}(\zeta)d\zeta \equiv \lim_{\varepsilon\to 0}\sum_{n=-1}^{-\infty}\int_{\gamma_{\varepsilon,z_i}} a_n^{(i)}(\zeta - z_i)^n d\zeta.$$

Man kann es berechnen, indem man γ_{ε,z_i} als $\gamma_{\varepsilon,z_i} : \zeta = z_i + \varepsilon e^{-i(\tau-\tau_0)}$, $\tau = [0,\pi]$ parametrisiert mit einem geeignet gewählten τ_0. Dann ist

$$\int_{\gamma_{\varepsilon,z_i}} a_n^{(i)}(\zeta - z_i)^n d\zeta = a_n^{(i)}\int_0^{\pi}\left(\varepsilon e^{-i(\tau-\tau_0)}\right)^n\left(-i\varepsilon e^{-i(\tau-\tau_0)}\right)d\tau$$

wofür drei Fälle zu unterscheiden sind:

a) Der Beitrag $n = -1$ liefert gerade das Residuum, aber nur mit einem halben Gewicht gegenüber dem gewöhnlichen Residuensatz:

$$\int_{\gamma_{\varepsilon,z_i}} a_{-1}^{(i)}(\zeta - z_i)^{-1}d\zeta = a_{-1}^{(i)}\int_0^{\pi} -i d\tau = -i\pi a_{-1}^{(i)} = -\pi i \operatorname*{Res}_{\zeta=z_i} f(\zeta).$$

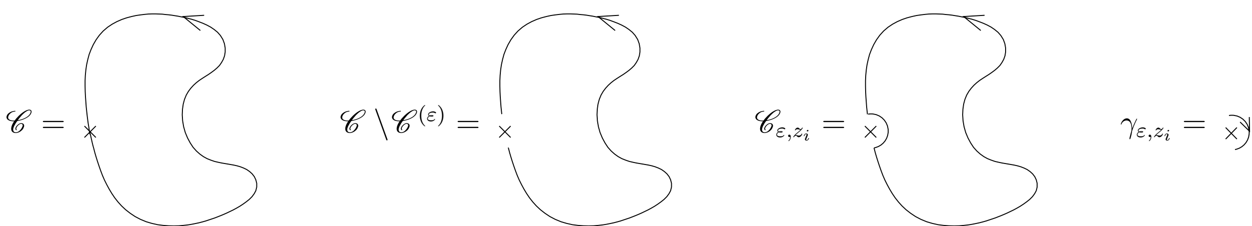


Abbildung A.2.: *Aufteilung der Integrationswege bei der Hauptwertintegration zum Beweis von Satz A.6.*

b) Die geraden Beiträge $n = 2m$ liefern einen Beitrag, falls die Koeffizienten $a_{2m}^{(i)}$ nicht ohnehin verschwinden:

$$\int_{\gamma_{\varepsilon,z_i}} a_{2m}^{(i)} (\zeta - z_i)^{2m} d\zeta = -i a_{2m}^{(i)} \int_0^\pi \varepsilon^{2m+1} e^{-i(2m+1)(\tau-\tau_0)} d\tau = -\frac{2\varepsilon^{2m+1} e^{-i(2m+1)\tau_0}}{2m+1} a_{2m}^{(i)}.$$

Da $n = 2m$ negativ war, ist dieser Anteil nicht wohldefiniert.

c) Die ungeraden Beiträge $n = 2m + 1$ mit $m < -2$ verschwinden:

$$\int_{\gamma_{\varepsilon,z_i}} a_{2m+1}^{(i)} (\zeta - z_i)^{2m+1} d\zeta = -i a_{2m+1}^{(i)} \int_0^\pi \varepsilon^{2m+2} e^{-i(2m+2)(\tau-\tau_0)} d\tau = 0.$$

Falls also die Laurententwicklung von $f(\zeta)$ an $\zeta = z_i$ nur ungerade negative Potenzen enthält, ist das Hauptwertintegral definiert. Hätten wir den Pol in der Zerlegung des Integrationsweges mit in das geschlossene Gebiet $\mathscr{C}_{\varepsilon,z_i}$ aufgenommen, statt ihn auszuschließen, hätten wir dasselbe Ergebnis bekommen, da das Teilstück γ_{ε,z_i} dann im mathematisch positiven Sinne zu parametrisieren gewesen wäre. Wir können nun den folgenden Satz formulieren:

Satz A.6 (Residuensatz mit Polen auf dem Integrationsweg) *Sei $f(\zeta)$ analytisch bis auf isolierte Singularitäten in einem einfach zusammenhängenden Gebiet Ω, $\mathscr{C}$ eine einfach geschlossene, positiv orientierte Jordan-Kurve, die bei $\zeta = z_i$, $i \in I$ durch endlich viele Singularitäten von $f(\zeta)$ führt. Wenn die Hauptteile $\mathrm{p}^{(i)}(\zeta)$ von $f(\zeta)$ an den Polen auf den Integrationswegen nur ungerade Anteile haben und im Inneren von $\mathscr{C}$ nur $n < \infty$ Singularitäten bei $\zeta = z_j$, $j \in J$ liegen, dann gilt:*

$$\boxed{\text{P.V.}\oint_{\mathscr{C}} f(\zeta) d\zeta = 2\pi i \sum_{j \in J} \operatorname*{Res}_{\zeta = z_j} f(\zeta) + \pi i \sum_{i \in I} \operatorname*{Res}_{\zeta = z_i} f(\zeta).}$$

A.3. Schnitte, Umkehrungen elementarer Funktionen und all das

Viele irrationale und transzendente Funktionen sind in der komplexen Ebene nicht eindeutig definierbar. Der Wertebereich, den die Wurzelfunktion annehmen kann, ist beispielsweise „zweiblättrig“. In solchen Fällen muss eine Festlegung getroffen werden, welche Werte die implementierte Funktion zurückliefern soll. Da es im Allgemeinen nicht möglich ist, dabei die

Analytizität (oder auch nur die Stetigkeit) des Bildes der Funktion zu erhalten, müssen sog. „Schnitte" in der komplexen Ebene festgelegt werden, die die Unstetigkeit im Bild definieren.

Dieser Abschnitt ist den Schnitten wichtiger Funktionen gewidmet. Es sollen die Topologien der Riemannflächen anschaulich dargestellt sowie Konventionen von Schnitten, die diese Topologien scheinbar stören, miteinander verglichen werden. Dies ist von Bedeutung für ein CAS, da Änderungen solcher Konventionen zu subtilen Fehlern führen können. Unter Berücksichtigung bestehender Standards wird hier die Konvention in GiNaC zementiert. Auf unproblematische Funktionen wie die trigonometrischen und hyperbolischen braucht hier nicht eingegangen zu werden, da sie einblättrig sind, also keinen Schnitt haben. Bei den problematischen Funktionen mit Schnitt werden wir feststellen, dass der Versuch diesen zu standardisieren in C++ nur ansatzweise unternommen wurde. Wir werden sehen, dass die aktuelle Revision von C diese Lücken schließt und einen Vergleich anstellen mit dem Quasi-Standard [Stee 1990] von Common Lisp, da dieser traditionell als CAS-Leitfaden herangezogen wird. Dabei wird sich glücklicherweise herausstellen, dass diese drei Regelwerke nicht in einem Widerspruch zueinander ausgelegt werden müssen.

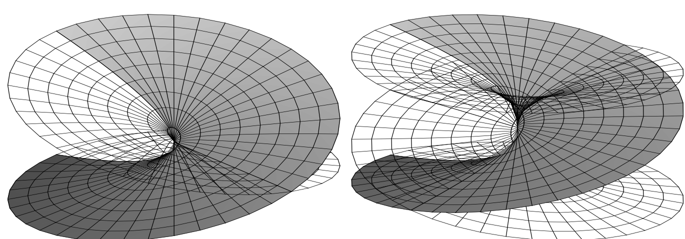


Abbildung A.3.: *Die Blätter von* $\sqrt{z}$ *und* $\sqrt[3]{z}$

Betrachten wir zunächst die Wurzelfunktion $\sqrt{z}$ als Umkehrung von z^2. Setzt man den „Keim" $(\sqrt{\ }, z_0 = 1)$ von der Stelle $z_0 = 1$ über die obere Halbebene analytisch fort zu $z = -1$, so erhält man $\sqrt{-1} = i$. Eine analytische Fortsetzung entlang eines Pfades in der unteren Halbebene führt dagegen zu $\sqrt{-1} = -i$. Schließt man den Kreis zum Ausgangspunkt z_0, so findet man $\sqrt{1} = -1$. Schließt man ihn noch einmal, so kommt man zum Keim $(\sqrt{\ }, z_0 = 1)$ zurück. Diese Mehrdeutigkeit lässt sich in der Riemannfläche veranschaulichen (Abbildung A.3, links). Hierbei ist zu beachten, dass die beiden „Blätter" an der scheinbaren Durchdringungslinie nicht aufeinanderfallen. Das Gedankenbild Riemann'scher Flächen ist vielmehr Folgendes: die Funktionen werden zwar als eindeutig betrachtet, aber dafür werden ihre Argumente als auf einer mehrblättrigen Mannigfaltigkeit liegend angesehen. Für die k-te Wurzel gilt dann analog zur Quadratwurzel, dass man k-mal schließen muss (Abbildung A.3, rechts). Die in diesem Abschnitt abgebildeten Riemannflächen sind genau genommen gar keine, sondern die Plots der Blätter der Funktionen. Mit der Topologie der Riemannflächen stimmen sie jedoch überein. Während dort die Mehrdeutigkeit im Argument der Funktion $f(z_k)$ liegt, ist sie hier in der Funktion $f_k(z)$ selbst. Dies ist von einem algebraischen Gesichtspunkt praktikabler, da wir leichter die Funktionswerte als die Argumente indizieren können [Jeff 2001].

Der C++-Standard [ISO 1998] definiert den Schnitt der komplexen Wurzelfunktion in Abschnitt 26.2.8:

```
template<class T> complex<T> sqrt (const complex<T>& x);
```

Notes: the branch cuts are along the negative real axis.

Returns: the complex square root of x, in the range of the right half-plane. If the argument is a negative real number, the value returned lies on the positive imaginary axis.

Dies ist in voller Übereinstimmung mit [Stee 1990], Abschnitt 12.5.3:

> `sqrt`: The branch cut for square root lies along the negative real axis, continuous with quadrant II. The range consists of the right half-plane, including the non-negative imaginary axis and excluding the negative imaginary axis.

Transzendente Funktionen

Der natürliche Logarithmus als Umkehrung der Exponentialfunktion ist offensichtlich unendlichdeutig. Wenn z' mit z durch $z' = ze^{2n\pi i}$ verknüpft ist, so ist klarerweise $\log(z') = \log(z) + 2n\pi i$, da der Imaginärteil des Logarithmus gerade die Phase des Argumentes ist: $\log(z) = \log(re^{i\varphi}) = \log(e^{\log r + i\varphi}) = \log r + i\varphi$. Die Unendlichdeutigkeit lässt sich jedoch auch über die Integraldarstellung $\log(z) = \int_1^z \frac{1}{\zeta} d\zeta$ einsehen in einer Weise, wie sie leicht verallgemeinerbar ist auf Funktionen, die wie z.B. der Dilogarithmus über eine Integraldarstellung definiert sind. Hierzu trennt man den Integrationsweg von 1 bis z' auf in einen Teil von 1 bis z und einen weiteren von z bis z'. Beim zweiten Teil ist darauf zu achten, dass die Null n mal umlaufen wird:

$$\log(z') = \int_1^z \frac{1}{\zeta} d\zeta + \int_z^{z'} \frac{1}{\zeta} d\zeta = \log(z) + n \oint_{S_r(0)} \frac{1}{\zeta} d\zeta.$$

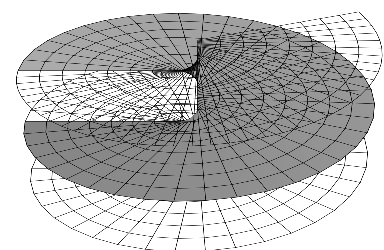


Abbildung A.4.: *Der Imaginärteil von* $\log(z)$

Der Zusatzterm ist wegen der Cauchy'schen Integralformel gerade wieder $2n\pi i$. Da also für ein beliebiges $z \in \mathbb{C}$ die Logarithmusfunktion nur modulo $2n\pi i$ eindeutig ist, ergeben sich für den Imaginärteil die unendlich vielen Blätter der einfachen Helix, so wie es in nebenstehender Zeichnung qualitativ angedeutet ist. Der Realteil der Logarithmusfunktion $\log(z)$ ist dabei stets eindeutig und gegeben durch den Logarithmus des Betrages des Argumentes: $\mathrm{Re}(\log z) = \log |z|$.

Der C++-Standard setzt den Schnitt des natürlichen Logarithmus unmissverständlich fest:

```
template<class T> complex<T> log (const complex<T>& x);
```

> **Notes:** the branch cuts are along the negative real axis.
> **Returns:** the complex natural (base e) logarithm of x, in the range of a strip mathematically unbounded along the real axis and in the interval [-i times pi, i times pi] along the imaginary axis. When x is a negative real number, `imag(log(x))` is pi.

Bis auf den unbedeutenden Fehler bezüglich der linken Intervallgrenze des Wertebereiches für den Imaginärteil finden wir hier auch wieder eine Übereinstimmung mit dem Lisp-Standard:

> `log`: The branch cut for the logarithm function of one argument (natural logarithm) lies along the negative real axis, continuous with quadrant II. The domain excludes the origin. For a complex number z, $\log z$ is defined to be
>
> $$\log z = (\log |z|) + i(\mathrm{phase}\, z)$$

> Therefore the range of the one-argument logarithm function is that strip of the complex plane containing numbers with imaginary parts between $-pi$ (exclusive) and pi (inclusive).

Der Schnitt gebietet Vorsicht bei Umformungen von Summen von Logarithmen. Über die dabei auftretenden Imaginärteile lässt sich mithilfe der in [t'HoVe 1979] eingeführten η-Funktion buchführen:

$$\log(z_1 z_2) = \ln(z_1) + \log(z_2) + \eta(z1, z2).$$

Dies kann als definierende Gleichung für $\eta(z1, z2)$ verstanden werden. Man findet die η-Funktion in modernerer Literatur auch häufig unter dem Namen „Entwindungszahl" $\mathcal{K}$ wieder, die mit der η-Funktion durch $2\pi i\mathcal{K}(\log(z1) + \log(z2)) = -\eta(z1, z2)$ verknüpft ist [CDJLW 2001]. Solange z_1, z_2 und $z_1 z_2$ alle nicht auf dem Schnitt des Logarithmus liegen, sie also nicht reell und negativ sind, lässt sich die η-Funktion durch Heaviside'sche Sprungfunktionen wie folgt ausdrücken:

$$\begin{aligned} \eta(z_1, z_2) \quad = \quad & 2i\pi\big[\theta(-\mathrm{Im}\, z_1)\,\theta(-\mathrm{Im}\, z_2)\,\theta(\mathrm{Im}(z_1 z_2)) \\ & - \theta(\mathrm{Im}\, z_1)\,\theta(\mathrm{Im}\, z_2)\,\theta(-\mathrm{Im}(z_1 z_2))\big], \end{aligned}$$

wobei die θ-Funktionen hier wieder mit $\theta(0) = \frac{1}{2}$ definiert sind. Vorsicht: Diese Gleichung ist nur im unkonventionellen Fall eines log-Schnittes entlang der negativen reellen Achse stetig zum dritten Quadranten richtig. Sie ist aber dennoch für eine Implementierung geeignet: in der hier getroffenen Vereinbarung müssen lediglich noch zusätzliche Faktoren von $i\pi$ korrigiert werden, falls z_1, z_2, $z_1 z_2$ oder Kombinationen davon reell und negativ sind. Für die vollständige Gleichung sei auf die Implementierung in GiNaC (`eta_eval()`) verwiesen.

Eine weitere wichtige transzendente Funktion ist der Arcustangens. Er tritt wie der natürliche Logarithmus bei der Integration rationaler Funktionen auf. Er lässt sich allerdings auf den natürlichen Logarithmus zurückführen. Durch Invertierung der Definitionsgleichung

$$z \equiv \tan(\mathrm{atan}(z)) \equiv \frac{\sin(\mathrm{atan}(z))}{\cos(\mathrm{atan}(z))} = -i\,\frac{e^{2i\,\mathrm{atan}(z)} - 1}{e^{2i\,\mathrm{atan}(z)} + 1}$$

findet man die Beziehung

$$e^{2i\,\mathrm{atan}(z)} = \frac{1 + iz}{1 - iz}.$$

Nimmt man von beiden Seiten den Logarithmus und beachtet, dass $\log(e^z) \equiv z \mod 2\pi i$, so sieht man, dass der Realteil nur modulo π eindeutig ist:

$$\mathrm{atan}(z) = \frac{1}{2i}\log\Big(\frac{1 + iz}{1 - iz}\Big) \mod \pi. \qquad \text{(A.10)}$$

Mit dieser Definition ist auch eine natürliche Wahl des Schnittes vorgeschlagen, indem man einfach das $\mod \pi$ streicht: Man könnte ihn dorthin legen, wo das Argument des Logarithmus reell und negativ ist, also beginnend bei den Verzweigungspunkten $\pm i$ entlang $z = \pm ir$, $r = (1 \ldots \infty)$. Setzt man den Arcustangens ober- und unterhalb des Schnittes fort, so findet man für den Realteil eine unendlichblättrige Topologie, wie sie in Abbildung A.5 links angedeutet ist. Wir werden diesem Vorschlag nur bis auf die Funktionswerte auf dem Schnitt selbst folgen.

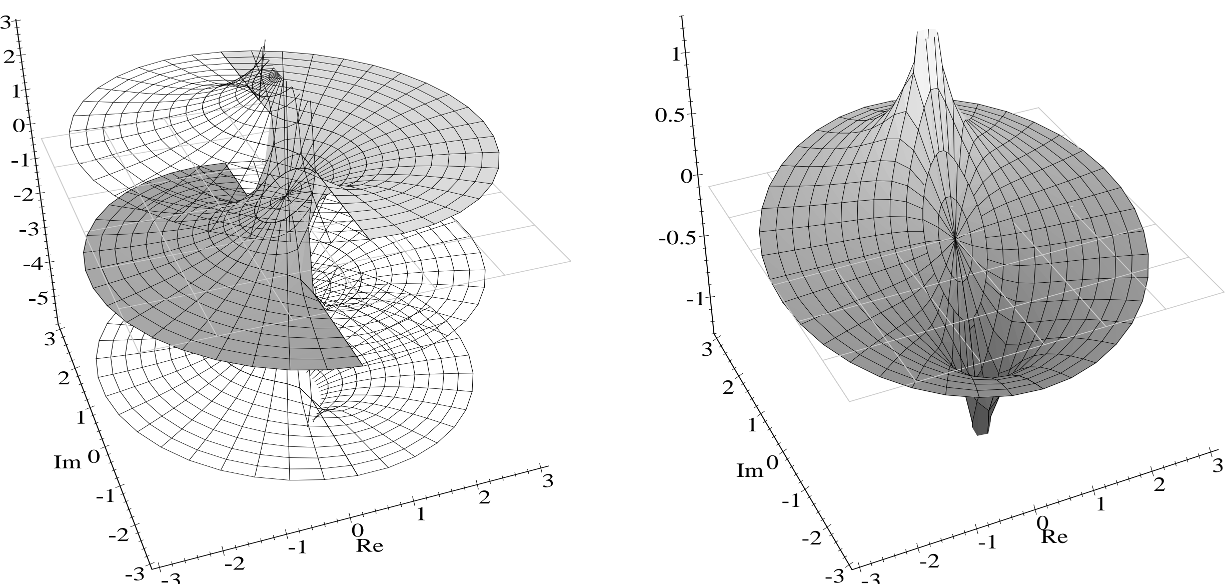


Abbildung A.5.: *Real- und Imaginärteil der Arcustangensfunktion (Reelle Achse nach rechts hinten). Der Hauptzweig ist massiv dargestellt.*

Unglückseligerweise hat das ISO/IEC JTC 1 SC 22 [4] versäumt, inverse trigonometrische und inverse hyperbolische Funktionen im C++-Standard [ISO 1998] zu spezifizieren – lediglich trigonometrische und hyperbolische Funktionen werden aufgeführt. Mit der Revision des C-Standards [ISO 1999] hielten jedoch komplexe Funktionen in C Einzug und man kann erwarten, dass kommende Revisionen von [ISO 1998] zumindest semantisch nicht dagegen verstoßen werden. In C bekommen sie dort das Präfix `c` vorangestellt und gelten für den Typ `double` als Argument und Rückgabewert, beziehungsweise mit dem Suffix `f` für den Typ `float` und dem Suffix `l` für den Typ `long double`. Man hat also beispielsweise die Prototypen `double complex catan(double complex z);` und `float complex catanf(float complex z);`. Abschnitt 7.3.5.3 schreibt vor:

> **Description**
> The `catan` functions compute the complex arc tangent of `z`, with branch cuts outside the interval [-i, +i] along the imaginary axis.
> **Returns**
> The `catan` functions return the complex arc tangent value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [$-\pi/2$,$+\pi/2$] along the real axis.

Diese Spezifizierung wird vervollständigt durch die Erläuterung in 7.3.3.2, wie Schnitte zu lesen sind:

> [...] implementations shall map a cut so the function is continuous as the cut is approached coming around the finite endpoint of the cut in a counter clock-

[4] Abkürzung für: International Organization for Standardization / International Electrotechnical Commission Joint Technical Committee 1, (*Information technology*) SubCommitee 22 (*Programming languages, their environments and system software interfaces*).

> wise direction. (Branch cuts for the functions specified here have just one finite endpoint.)

Für den Arcustangens bedeutet dies insbesondere, dass der Schnitt auf der positiven imaginären Achse stetig mit dem ersten Quadranten ist, während der Schnitt auf der negativen imaginären Achse stetig an den dritten Quadranten anschließt. Man kann davon ausgehen, dass künftige Revisionen des C++-Standards sich an die im C-Standard spezifizierte Semantik halten werden. Zum Vergleich sei noch der relevante Absatz in [Stee 1990] zitiert:

> `atan`: X3J13 voted in January 1989 (COMPLEX-ATAN-BRANCH-CUT) to replace the formula $\operatorname{atan}(z) = -i\log((1+iz)\sqrt{1/(1+z^2)})$ with the formula
>
> $$\operatorname{atan} z = \frac{\log(1+iz) - \log(1-iz)}{2i}$$
>
> This is equivalent to the formula
>
> $$\operatorname{atan} z = \frac{\operatorname{atanh} iz}{i}$$
>
> recommended by Kahan [Kah 1987]. It causes the upper branch cut to be continuous with quadrant I rather than quadrant II, and the lower branch cut to be continuous with quadrant III. [...]
> The branch cut for the arc tangent function is in two pieces: one along the positive imaginary axis above i (exclusive), continuous with quadrant I, and one along the negative imaginary axis below $-i$ (exclusive), continuous with quadrant III. The points i and $-i$ are excluded from the domain. The range is that strip of the complex plane containing numbers whose real part is between $-\pi/2$ and $\pi/2$. A number with real part equal to $-\pi/2$ is in the range if and only if its imaginary part is strictly negative; a number with real part equal to $\pi/2$ is in the range if and only if its imaginary part is strictly positive. Thus the range of the arc tangent function is *not* identical to that of the arc sine function.

Dies ist gekürzt um diejenigen Besonderheiten, die durch die Unterscheidung zwischen $+0$ und -0 entstehen. Systeme, die eine solche Unterscheidung zulassen, werden in [Kah 1987] favorisiert und in [Stee 1990] optional berücksichtigt. Sie lassen die Formulierung von $\sqrt{-4+0i} = 2i$ und $\sqrt{-4-0i} = -2i$ zu und können Verwechslungen bei Schnitten vermeiden. Obwohl diese Unterscheidung attraktiv ist, löst sie nicht die Probleme, mit denen ein Implementator eines CAS konfrontiert ist: „Was ist gemeint, wenn ein Benutzer $\sqrt{-4}$ eingibt?“. Daher, und weil CLN dies nicht unterstützt, sehen wir hier und im Folgenden von der Unterscheidung zwischen $+0$ und -0 ab.

Die vorgeschlagene Gleichung sieht unserer Gleichung (A.10) schon sehr ähnlich, welche jedoch nur modulo π eindeutig ist und daher noch alle Freiheiten zur Wahl des Schnittes zulässt. Tatsächlich ist für Argumente $z = -ir, r > 1$

$$\begin{aligned}\frac{1}{2i}\log\Big(\frac{1+iz}{1-iz}\Big) &= \frac{1}{2i}\log\Big(\underbrace{\frac{1+r}{1-r}}_{<0}\Big) = \frac{1}{2i}\Big(\log\Big(\underbrace{\frac{1+r}{r-1}}_{>0}\Big) + \log(-1)\Big)\\ &= \frac{1}{2i}\Big(\big(\log(1+r) - \log(r-1)\big) + \log(-1)\Big),\end{aligned}$$

aber

$$\frac{1}{2i}\big(\log(1+iz)-\log(1-iz)\big) = \frac{1}{2i}\big(\log(\underbrace{1+r}_{>0})-\log(\underbrace{1-r}_{<0})\big)$$
$$= \frac{1}{2i}\Big(\big(\log(1+r)-\log(r-1)\big)-\log(-1)\Big).$$

Die beiden Definitionen unterscheiden sich also auf dem Schnitt in der unteren Halbebene um $\frac{1}{2i}2\log(-1)=\pi$. Dort wollen wir der lispschen Definition folgen, die sich wieder als kompatibel mit der C-Definition herausstellt.

Analog zum Arcustangens sollen nun noch die übrigen inversen trigonometrischen Funktionen und die Lage ihrer Schnitte untersucht werden.

Für den Arcussinus können wir wieder wie beim Arcustangens vorgehen. Durch Invertierung der Definitionsgleichung

$$z \equiv \sin(\operatorname{asin}(z)) \equiv \frac{1}{2i}\big(e^{i\operatorname{asin}(z)} - e^{i\operatorname{asin}(z)}\big)$$

findet man die quadratische Gleichung

$$0 = e^{2i\operatorname{asin}(z)} - 2ize^{i\operatorname{asin}(z)} - 1.$$

Löst man sie und nimmt von beiden Seiten den Logarithmus, wobei wieder $\log(e^z) \equiv z \mod 2\pi i$ beachtet werden muss, so wird manifest, dass der Imaginärteil nun zweideutig ist, ebenso wie der Realteil der allerdings zusätzlich noch um modulo 2π uneindeutig ist:

$$\operatorname{asin}(z) = -i\log\big(iz \pm \sqrt{1-z^2}\big) \mod 2\pi. \tag{A.11}$$

Das Ergebnis ist die Topologie in Abbildung A.6. Nur das positive Vorzeichen entspricht einer im Ursprung stetigen Funktion und ist daher zu bevorzugen.

Auch diese Funktion wird in C++ nicht spezifiziert. Der C-Standard hingegen sagt lediglich:

> **Description**
> The `casin` functions compute the complex arc sine of `z`, with branch cuts outside the interval [-1, +1] along the real axis.
> **Returns**
> The `casin` functions return the complex arc sine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval $[-\pi/2,+\pi/2]$ along the real axis.

Mit der Konvention über die Stetigkeit auf dem Schnitt stimmt dies mit der lispschen Definition ([Stee 1990], 12.5.3) überein:

> `asin`: The following definition for arc sine determines the range and branch cuts:
>
> $$\operatorname{asin}(z) = -i\log\big(iz + \sqrt{1-z^2}\big)$$

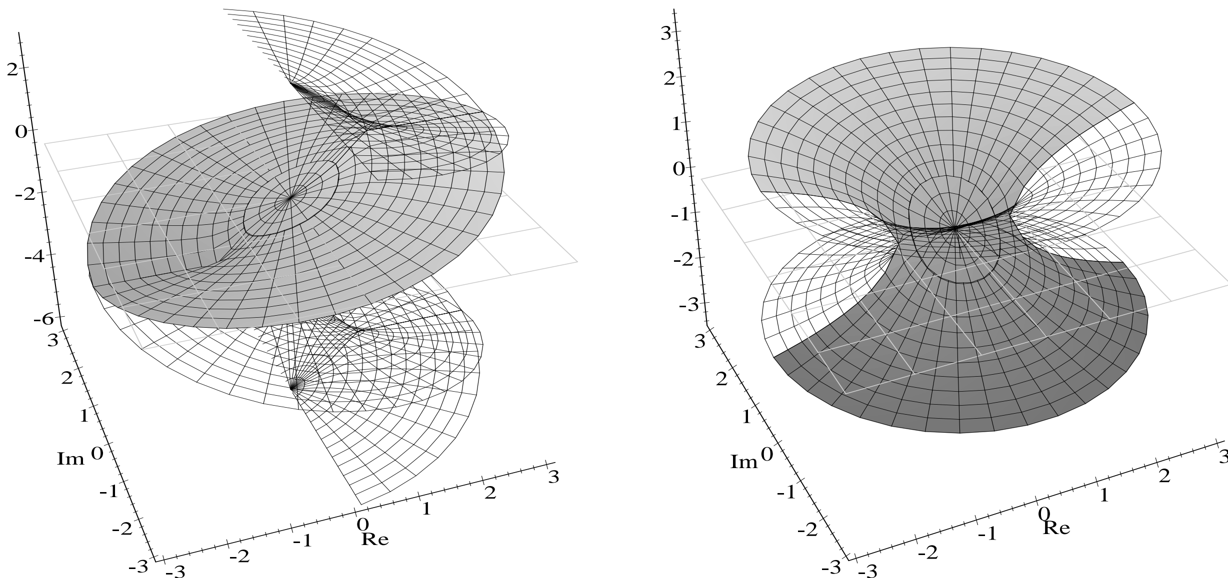


Abbildung A.6.: *Real- und Imaginärteil der Arcussinusfunktion (Reelle Achse nach rechts hinten). Der Hauptzweig ist massiv dargestellt.*

> This is equivalent to the formula
>
> $$\text{asin}(z) = \frac{\text{asinh}(iz)}{i}$$
>
> recommended by Kahan [Kah 1987].
> The branch cut for the arc sine function is in two pieces: one along the negative real axis to the left of -1 (inclusive), continuous with quadrant II, and one along the positive real axis to the right of 1 (inclusive), continuous with quadrant IV. The range is that strip of the complex plane containing numbers whose real part is between $-\pi/2$ and $\pi/2$. A number with real part equal to $-\pi/2$ is in the range if and only if its imaginary part is non-negative; a number with real part equal to $\pi/2$ is in the range if and only if its imaginary part is non-positive.

Der Arcuscosinus kann wieder genauso behandelt werden. Andererseits ist natürlich $\text{acos}(z) = \frac{\pi}{2} - \text{asin}(z)$ und diese Definition kann auch für den Schnitt herangezogen werden. Die Wertemenge ist also im Realteil beschränkt auf das Intervall $[0, \pi]$. Wie aus dem Vergleich der Abbildungen A.7 und A.6 hervorgeht, ist die Topologie natürlich dieselbe wie diejenige des Arcussinus, lediglich der gewählte Schnitt ist aufgrund seiner Symmetrie etwas gewöhnungsbedürftig.

Ebenso wie die hyperbolischen Funktionen aus den trigonometrischen Funktionen durch Multiplikation des Argumentes mit i hervorgehen, gehen auch die inversen hyperbolischen Funktionen aus den inversen trigonometrischen Funktionen hervor. Sie werden daher hier nicht

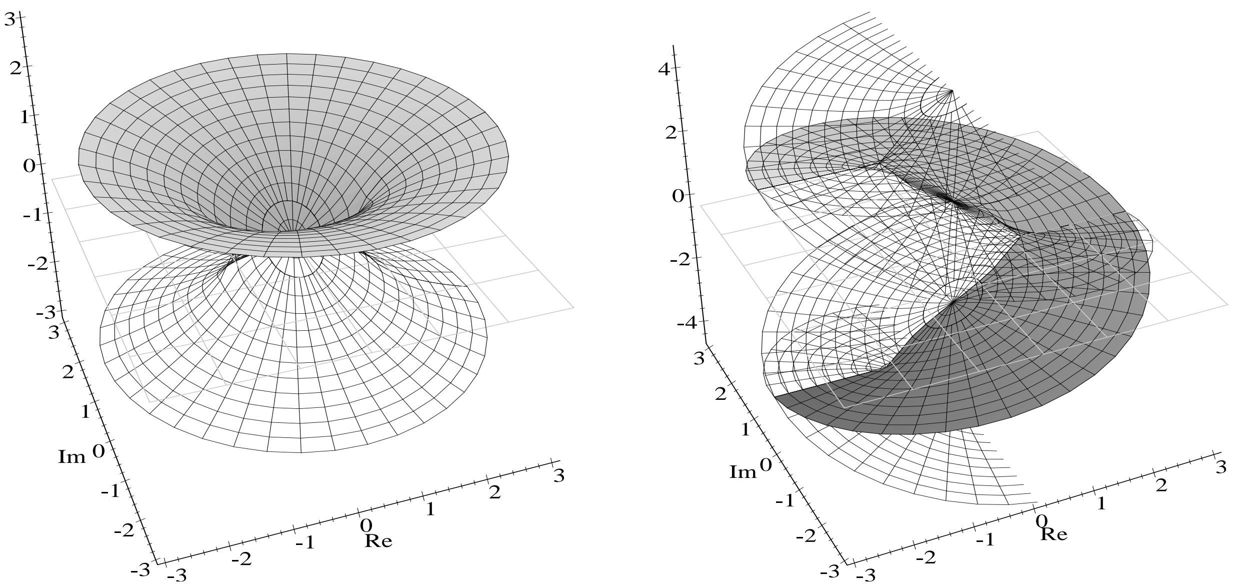


Abbildung A.7.: *Real- und Imaginärteil des Arcuscosinushyperbolicus (Reelle Achse nach rechts hinten). Der Hauptzweig ist massiv dargestellt.*

weiter behandelt. Die folgenden Identitäten sind exakt, auch auf eventuellen Schnitten:

$$
\begin{aligned}
\sinh(z) &= -i\sin(iz) & \sin(z) &= -i\sinh(iz)\\
\cosh(z) &= \cos(iz) & \cos(z) &= \cosh(iz)\\
\tanh(z) &= -i\tan(iz) & \tan(z) &= -i\tanh(iz)\\
\operatorname{asinh}(z) &= -i\operatorname{asin}(iz) & \operatorname{asin}(z) &= -i\operatorname{asinh}(iz)\\
\operatorname{atanh}(z) &= -i\operatorname{atan}(iz) & \operatorname{atan}(z) &= -i\operatorname{atanh}(iz).
\end{aligned}
$$

Zusammenfassend lässt sich sagen, dass zwischen den einzelnen Regelwerken keine Widersprüche auftreten, so dass wir uns nicht fragen müssen, welcher Konvention wir uns verpflichtet fühlen sollten. Traditionell folgen CAS-Hersteller gerne dem Lisp-Standard und so nimmt es nicht Wunder, dass sowohl Mathematica als auch Maple die Schnitte genau so implementieren wie hier beschrieben. Vorsicht ist jedoch geboten beim Vergleich mit Reduce und MuPAD – diese Systeme bieten in der Implementierung ihrer Schnitte eine Reihe Überraschungen.

Doppelt transzendente Funktionen

Doppelt transzendente Funktionen treten bei der Integration von transzendenten Funktionen auf, oder allgemeiner bei der Integration von Produkten aus einer rationalen Funktion und einer transzendenten Funktion.

Prominentestes Beispiel ist der Dilogarithmus Li_2.[5] Er ist definiert durch

$$\mathrm{Li}_2(z) := -\int_0^z \frac{\log(1-\zeta)}{\zeta}\,d\zeta.$$

Der Verzweigungspunkt ist mit dieser Definition bei $z = 1$. Dort beginnt der Schnitt und spiegelt denjenigen der Logarithmusfunktion wieder: Da wir jenen entlang der negativen reellen

[5] Auch Spence- oder Jonquière-Funktion genannt.

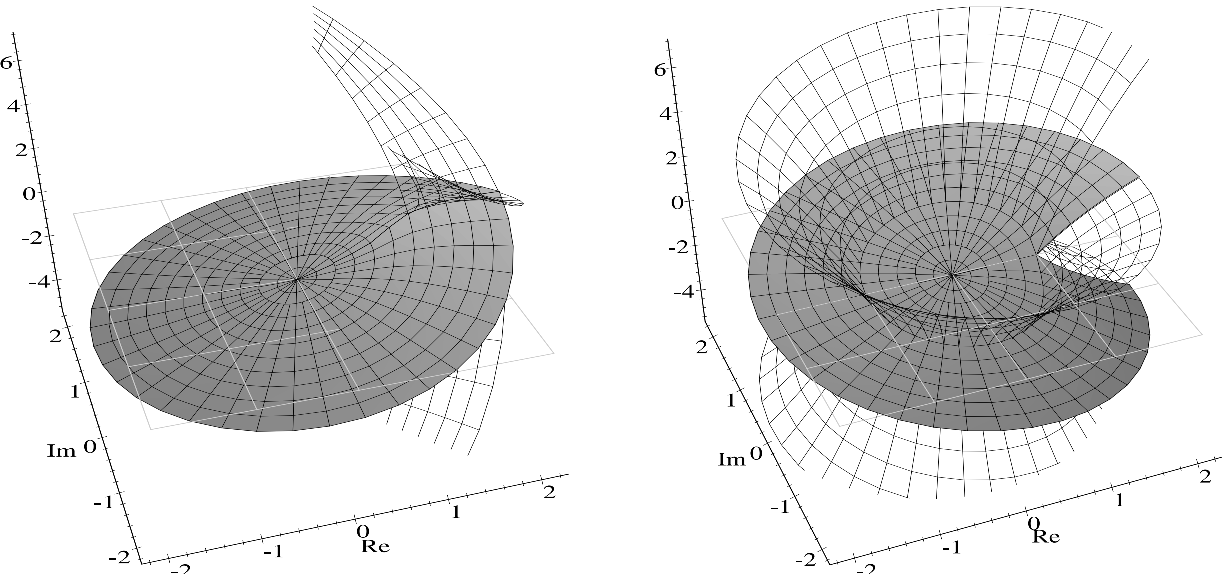


Abbildung A.9.: *Real- und Imaginärteil der Dilogarithmusfunktion (Reelle Achse nach rechts hinten). Außerhalb des Einheitskreises „erbt" der Dilogarithmus die Unendlichdeutigkeit vom Logarithmus.*

Achse vereinbart haben, liegt der Schnitt des Dilogarithmus da, wo $\log(1–z)$ seinen Schnitt hat, also beginnend bei 1 entlang der positiven reellen Achse und stetig zum vierten Quadranten.

Um etwas über die Mehrdeutigkeit von Real- bzw. Imaginärteil der Dilogarithmusfunktion zu lernen, gehen wir vor wie bei der Logarithmusfunktion. Sei wieder z' mit z durch $z' = ze^{2n\pi i}$ verknüpft. Dann ist

$$\mathrm{Li}_2(z') = \mathrm{Li}_2(z) - n \oint_{S_r(0)} \frac{\log(1-\zeta)}{\zeta} d\zeta.$$

Das Wegintegral verschwindet im Falle $r < 1$ aufgrund des Cauchy'schen Integralsatzes, da der Integrand analytisch innerhalb $S_r(0)$ ist. Falls $r \geq 1$, wird jedoch über den Schnitt von $\log(1-\zeta)$ integriert und man muss die auftretende Phase berücksichtigen. Da das Integral über den neben skizzierten Integrationsweg aber verschwindet und die Divergenz bei $\zeta = 1$ schwächer als ζ^{-1} ist, kann man das Integral über den Kreisbogen $S_r(0)$ ersetzen durch ein Integral oberhalb des Schnittes von $\zeta = r$ bis 1 und unterhalb des Schnittes von $\zeta = 1$ bis r. Setzt man dann noch die Differenz der Logarithmusfunktion ober- und unterhalb ihres Schnittes ($2\pi i$) ein, so erhält man für das gesuchte Wegintegral

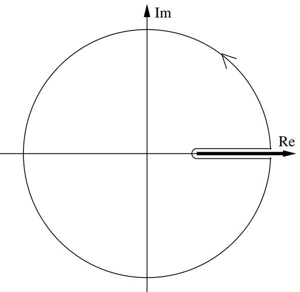


Abbildung A.8.: *Integrationsweg zum Schnitt von* $\mathrm{Li}_2(z)$

$$\begin{aligned}\oint_{S_r(0)} \frac{\log(1-\zeta)}{\zeta} d\zeta &= \int_r^1 \frac{\log(1-(\zeta+i\varepsilon))}{\zeta} d\zeta + \int_1^r \frac{\log(1-(\zeta-i\varepsilon))}{\zeta} d\zeta \\ &= \int_r^1 \frac{\log(1-(\zeta+i\varepsilon))}{\zeta} d\zeta + \int_1^r \frac{\log(1-(\zeta+i\varepsilon))+2\pi i}{\zeta} d\zeta \\ &= \int_1^r \frac{2\pi i}{\zeta} d\zeta \;=\; 2\pi i \log(r).\end{aligned}$$

Dies ist zunächst rein imaginär. Der Imaginärteil des Dilogarithmus „erbt“ die Unendlichdeutigkeit außerhalb des Einheitskreises $|z| = 1$ vom Logarithmus. Wenn wir nun noch die Uneindeutigkeit des Imaginärteils des Logarithmus in obigem Resultat selbst berücksichtigen, erhalten wir auch eine Unendlichdeutigkeit des Realteils des Dilogarithmus außerhalb des Einheitskreises, die die Analytizität am Scheitel entlang des Schnittes wieder herstellt. Abbildung A.9 skizziert das Ergebnis.

Höhere Polylogarithmen fallen an bei der mehrfachen Integration des Dilogarithmus (siehe [Lew 1981]). Allgemein ist

$$\mathrm{Li}_m(z) := \int_0^z \frac{\mathrm{Li}_{m-1}(\zeta)}{\zeta} d\zeta,$$

womit der Schnitt dort zu liegen kommt, wo $\mathrm{Li}_{m-1}(\zeta)$ den Schnitt aufweist, also stets entlang der postiven reellen Achse, beginnend bei 1 und stetig mit der oberen komplexen Halbebene. Die Unstetigkeit auf dem Schnitt kann in völliger Analogie zum Dilogarithmus berechnet werden. Man findet für $z' = ze^{2n\pi i}$

$$\mathrm{Li}_m(z') = \mathrm{Li}_m(z) + \frac{2n\pi i}{(m-1)!}(\log z)^{m-1}.$$

$f(z)$ Definition	Definitionsbereich Wertebereich	Schnitt	Unstetigkeit am Schnitt
$\log(z)$	$\mathbb{C}\setminus 0$ $\{\zeta\in\mathbb{C} : -\pi < \mathrm{Im}(\zeta) \le \pi\}$		$2\pi i$
$\sqrt{z}$ $e^{\log(z)/2}$	$\mathbb{C}$ $\mathbb{C}$		$2i\sqrt{\lvert z\rvert}$
$\mathrm{asin}(z)$ $-i\log(iz+\sqrt{1-z^2})$	$\mathbb{C}$ $\{\zeta\in\mathbb{C} : -\frac{\pi}{2} \le \mathrm{Re}(\zeta) \le \frac{\pi}{2}\}$		$i\log\left(\frac{z-\sqrt{z^2-1}}{z+\sqrt{z^2-1}}\right)$
$\mathrm{acos}(z)$ $-i\log(z+i\sqrt{1-z^2})$	$\mathbb{C}$ $\{\zeta\in\mathbb{C} : 0 \le \mathrm{Re}(\zeta) \le \pi\}$		$i\log\left(\frac{z-\sqrt{z^2-1}}{z+\sqrt{z^2-1}}\right)$
$\mathrm{atan}(z)$ $\frac{\log(1+iz)-\log(1-iz)}{2i}$	$\mathbb{C}\setminus\{i,-i\}$ $\{\zeta\in\mathbb{C} : -\frac{\pi}{2} \le \mathrm{Re}(\zeta) \le \frac{\pi}{2}\}$		π
$\mathrm{asinh}(z)$ $\log(z+\sqrt{1+z^2})$	$\mathbb{C}$ $\{\zeta\in\mathbb{C} : -\frac{\pi}{2} \le \mathrm{Im}(\zeta) \le \frac{\pi}{2}\}$		$\log\left(\frac{\lvert z\rvert-\sqrt{\lvert z\rvert^2-1}}{\lvert z\rvert+\sqrt{\lvert z\rvert^2-1}}\right)$
$\mathrm{acosh}(z)$ $2\log(\sqrt{(z+1)/2}+\sqrt{(z-1)/2})$	$\mathbb{C}$ $\{\zeta\in\mathbb{C} : -\pi < \mathrm{Im}(\zeta) \le \pi,$ $\mathrm{Re}(\zeta) \ge 0\}$		$2\pi i$ für $z \le -1$ $2i\log\left(\sqrt{(z+1)/2}+\sqrt{(z-1)/2}\right)$ für $z \in [-1\ldots 1]$
$\mathrm{atanh}(z)$ $\frac{\log(1+z)-\log(1-z)}{2}$	$\mathbb{C}\setminus\{-1,1\}$ $\{\zeta\in\mathbb{C} : -\frac{\pi}{2} \le \mathrm{Im}(\zeta) \le \frac{\pi}{2}\}$		πi
$\mathrm{Li}_2(z)$ $-\int_0^z \frac{\log(1-\zeta)}{\zeta}d\zeta$	$\mathbb{C}$ $\mathbb{C}$		$2\pi i\log z$
$\mathrm{Li}_n(z)$ $-\int_0^z \frac{\mathrm{Li}_{n-1}(1-\zeta)}{\zeta}d\zeta$	$\mathbb{C}$ $\mathbb{C}$		$\frac{2\pi i}{(n-1)!}(\log z)^{n-1}$

Tabelle A.1.: *Auflistung der Schnitte in der komplexen Ebene*

B. ‚pvegas': parallele MC-Integration

Real supercomputing consists of converting
CPU-bound problems to IO-bound ones
anonymous

In diesem Anhang soll kurz auf den in dieser Arbeit verwendeten Monte-Carlo- (MC-) Integrationsalgorithmus eingegangen werden, sowie auf die parallelisierte Version desselben. Zum einen gibt es deutliche algorithmische Veränderungen gegenüber der in [Krec 1997a] vorgestellten Version, die teilweise schon in [Krec 1997b] beschrieben wurden, zum anderen wurden nach dem Erfahrungsrückfluss zahlreicher Anwender noch weitere Veränderungen vorgenommen und eine verlässliche Bewertung der Skalierungseigenschaften erst ermöglicht.

B.1. Vegas

Als MC-Integrationsalgorithmus approximiert `vegas` [Lepa 1978] ein Integral durch Auswertung des Integranden an einer Stützpunktmenge im D-dimensionalen Integrationsgebiet Ω:

$$S^{(1)} := \frac{|\Omega|}{n} \sum_{i=1}^{n} f(\boldsymbol{x}_i) \to \int_{\Omega} f(\boldsymbol{x})\, d\boldsymbol{x}, \tag{B.1}$$

welches wie $1/\sqrt{n}$ konvergiert, unabhängig von der Dimensionalität des Integrationsvolumens.[1] Zwei wohlbekannte Verbesserungen an dem Konvergenzverhalten sind das sogenannte „*stratified sampling*" und das „*importance sampling*". Bei ersterem wird die zufällige Stützpunktmenge ersetzt durch eine sorgfältig präparierte, die das Integrationsvolumen gleichmäßiger ausfüllt – im Idealfall nähert sich die Konvergenz dabei $1/n$.[2] Beim *importance sampling* wird die Stützpunktmenge dort verfeinert, wo der Integrand ein interessantes Verhalten aufweist, also entweder $f(\boldsymbol{x})$, $|\boldsymbol{\nabla} f(\boldsymbol{x})|$ oder beide groß sind – man spricht daher auch von einem „adaptiven" Verfahren.

[1] Dies steht im Gegensatz zu iterierten eindimensionalen Integrationsverfahren (iterierte Gauß-Quadratur, etc.): Ist dort die Konvergenz im eindimensionalen Fall n^{-h}, so verschlechtert sie sich durch das Iterieren über die D Dimensionen zu $n^{-h/D}$ – eine einfache Konsequenz Gauß'scher Fehlerfortpflanzung. Da $h \simeq 1$, sind MC-Integrationen solchen Verfahren im Falle $D > 2$ vorzuziehen.

[2] Ein Vergleich mit der belebten Natur drängt sich geradezu auf: die natürliche Verteilung einzelner Bäume in einem Nadelwald folgt erfahrungsgemäß keiner naiven zweidimensionalen Zufallsfolge. Da freistehende nachwachsende Bäume bessere Überlebenschancen haben als überschattete Sprösslinge, streut die Gesamtverteilung besser – was auf winterlichen Luftaufnahmen sofort ins Auge springt.

stratified sampling:
Verteilt die Stützpunkte gleichmäßiger als echte Zufallsfolgen.

importance sampling:
Passt die Stützpunktmenge dem Verlauf der jeweiligen Funktion an.

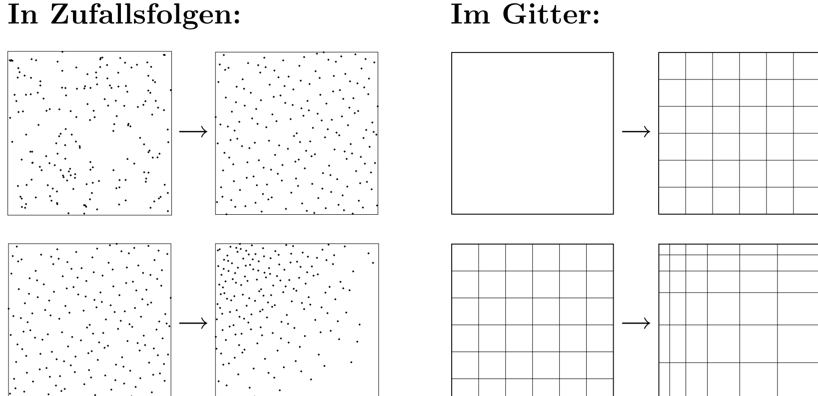


Abbildung B.1.: *Sampling-Methoden:* „stratified sampling“ *und* „importance sampling“. *Die Zufallsverteilungen der mittleren Spalte wurden erzeugt mit: einem linearen Kongruenzgenerator (links oben), einer Sobol'-Reihe [PTVF 1992] (rechts oben und links unten), mit der $\sqrt{\ }$-Funktion transformierte Sobol'-Reihe (rechts unten).*

Da die Dimensionalität des Integrals für `vegas` ein Parameter sein soll, werden beide Methoden verwirklicht, indem ein orthogonales Gitter über dem Integrationsgebiet eingeführt wird (siehe Abbildung B.1). Die gleiche Anzahl von Zufallsstützpunkten in jedem Subvolumen garantiert, dass sie im Gesamtvolumen gleichmäßiger gestreut sind – und implementiert so *stratified sampling*. Iterativ werden die Gitterlinien dann dem Integranden angepasst, was *importance sampling* entspricht. Die Einschränkung bei dieser Methode ist die Orthogonalität des Gitters: es lässt *importance sampling* nur für Funktionen zu, die sich gewissermaßen an die vorgegebenen Koordinaten halten, im Idealfall faktorisieren. Tatsächlich ist *importance sampling* äquivalent zu einer Transformation des Integrals

$$\int_{\Omega} f(\boldsymbol{x})\, d\boldsymbol{x} \;=\; \int_{P^{-1}(\Omega)} f(\boldsymbol{P}(\boldsymbol{y})) \left| \frac{\partial \boldsymbol{P}}{\partial \boldsymbol{y}} \right| d\boldsymbol{y}$$

mit der Randbedingung, dass die Ränder des Integrationsgebietes $\partial\Omega$ unverändert bleiben. Dann entspricht dem orthogonalen Gitter eine faktorisierende Transformation $\boldsymbol{P}(\boldsymbol{y})$, die jede Koordinate von $\boldsymbol{y}$ einzeln transformiert. Ein solches Gitter wird naturgemäß nicht in der Lage sein, sich iterativ an eine Funktion anzupassen, die ihr Maximum z.B. entlang der Diagonalen von Ω hat.

B.2. Parallelisierung

Eine Parallelisierung von `vegas` profitiert sicherlich davon, dass der Aufruf des Integranden an verschiedenen Stützpunkten unabhängig voneinander ist.[3] Man könnte also `vegas` mehrfach parallel aufrufen und jedem der p Prozessoren einen Anteil n/p aus den n Punkten der Stützpunktmenge zuteilen. Nennen wir dies **Ansatz I** und notieren es in einem Stück Pseudo-Code, worin die Programmstruktur durch Einrückung dargestellt wird:

[3] Dies muss in der Praxis sicher gestellt werden: unvorsichtig geschriebene Makros, Schreibzugriffe auf statische Variablen und common-Blöcke können die *thread-Sicherheit* zerstören.

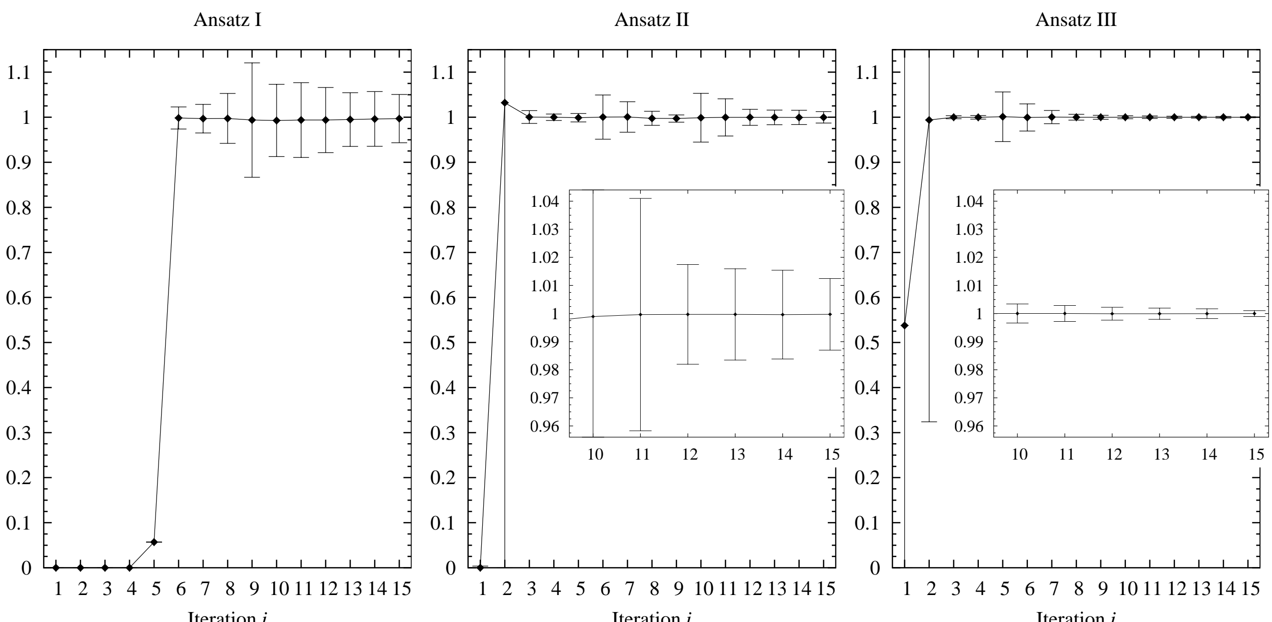


Abbildung B.2.: *Konvergenzvergleich dreier verschiedener Parallelisierungsansätze für $p = 16$ Prozessoren. Die exakte Lösung des Integrals ist auf 1 normiert. Die Fehlerbalken sind zur Visualisierung in allen Diagrammen um einen Faktor 50 vergrößert.*

```
1  for i←1 to p do in parallel
2      for all iterations
3          call vegas(n/p)
4  average results
```

Der Schritt „`average results`“ kann beispielsweise Durchschnittsbildung der Endergebnisse sowie der Fehler beinhalten. Da die Fehler im allgemeinen verschieden groß sein werden, sollten die Ergebnisse mit den Inversen der Fehler relativ zueinander gewichtet werden. **Ansatz I** krankt jedoch an der stochastischen Natur von MC-Integration: Die einzelnen Prozessoren werden sich unterschiedlich gut an den Integranden anpassen, manche unter Umständen nicht einmal dessen „interessante“ Stellen finden, somit falsche Ergebnisse liefern und die Fehler unterschätzen.

Eine naheliegende Verbesserung ist, nach jeder Iteration die einzelnen Prozessoren sich über das Ergebnis ihres Adaptationsprozesses austauschen zu lassen. Dies würde im Falle von `vegas` über eine Synchronisation der Gitterdaten erfolgen, nennen wir es **Ansatz II**:

```
1  for all iterations
2      for i←1 to p do in parallel
3          call vegas(n/p)
4      synchronize grids
5  average results
```

Man bemerke jedoch, dass hierzu die beiden Schleifen vertauscht werden mussten, was zu einem etwas unbalancierten Laufzeitverhalten führen kann, wenn z.B. einer der Prozessoren zwischenzeitlich mit einer anderen Aufgabe beschäftigt ist und am Ende der inneren Schleife

alle anderen auf ihn warten müssen. Beide Verfahren (man kann sie als makroparallel bezeichnen) kranken jedoch noch immer an einem Problem, was mit dem von `vegas` aufgebauten Gitter zusammenhängt: Ruft man ein unparallelisiertes `vegas` mit der Anzahl von Punkten auf, die vorher auf p Prozessoren verteilt wurden (also „`call vegas(n)`"), so kann der Algorithmus sich besser dem Verlauf des Integranden anpassen – denn es herrscht ja die Zwangsbedingung, dass in jedes Subvolumen die gleiche Anzahl von Stützpunkten fallen muss. Es ist also erstrebenswert, eine Parallelisierung zu finden, die die numerische Struktur des sequentiellen Algorithmus nicht zerstört. In `vegas` bietet sich hierzu eine zentrale Schleife an, die über die einzelnen Subvolumina des Gitters iteriert. Der **Ansatz III** ist in `pvegas` implementiert und ein Aufruf lautet, da die Parallelisierung nun intern erledigt werden muss, lediglich:

```
1  call pvegas(n)
```

Abbildung B.2 verdeutlicht die Überlegenheit eines solchen mikroparallelen Verfahrens gegenüber den ersten beiden Ansätzen anhand einer scharfen Gaußfunktion in einem fünfdimensionalen Gebiet Ω. Im ersten Graphen liefert ein Testlauf in den ersten fünf Iterationen völlig falsche Ergebnisse, da einige der 16 Prozessoren ihr Gitter noch nicht an den Integranden angepasst hatten und ihre Fehlerabschätzung so gering war ($\approx 10^{-32}$), dass die Ergebnisse der anderen unterdrückt wurden. Der zweite und dritte Graph zeigen, wie **Ansatz II** dieses Problem wie erwartet löst, die Konvergenz bei den späteren Iterationen jedoch deutlich hinter derjenigen vom mikroparallelen **Ansatz III** zurückbleibt.

Die Anzahl der Subvolumina ist in der Regel allerdings sehr groß[4] und eine völlige Parallelisierung dieser Schleife würde demnach zu einer zu feinkörnigen Problemaufspaltung führen um von einer realistischen Kommunikationshardware bewältigt werden zu können. `pvegas` spaltet die Schleife daher auf in zwei. Hierzu wird zunächst das D-dimensionale Volumen zerlegt in einen „Parallelraum" und einen „Orthogonalraum" mit den jeweiligen Dimensionen $D_\parallel$ und $D_\perp \equiv D - D_\parallel$. Die Iteration über die Parallelraum-Subvolumina wird aufgeteilt auf die p Prozessoren, während jeder von ihnen die Iteration über die dazu gehörenden Orthogonalraum-Subvolumina sequentiell durchführt. Die Aufteilung in $D_\parallel$ und $D_\perp$ kann vom Benutzer den jeweiligen Verhältnissen (Integrand, Genauigkeitsziel, Maschine) angepasst werden, jedoch ist $D_\parallel = \lfloor D/2 \rfloor$ und $D_\perp = \lceil D/2 \rceil$ ein guter Startwert und daher Standardeinstellung. Ein Blick in die Schleifen von `pvegas` zeigt stark vereinfacht Folgendes:

```
1  for all iterations
2      for par←1 to D∥ do in parallel
3          for per←D∥+1 to D do
4              for all x in samplepoints_within_hypercube
5                  accumulate results of call f(x)
6      compute result
```

Für die parallele Implementierung kommt so in natürlicher Weise ein Master-Slave-Modell in Frage, in dem ein Master-Prozess über den Parallelraum iteriert und die Worker über den Orthogonalraum.

[4] Sie ist höchstens gleich $n/2$ da man mindestens 2 Stützpunkte pro Subvolumen braucht, um Varianz definieren zu können. In der Praxis ist sie jedoch selten kleiner.

B.3. Nebenintegrale

Das Aufsummieren von sogenannten Nebenintegralen zusätzlich zum Hauptintegral ist eine häufig an Vegas gestellte Aufgabe. Traditionell werden dafür innerhalb des Integranden f zusätzliche Akkumulatoren aufsummiert, gewichtet mit einer von Vegas bereitgestellten Variablen w, die natürlich auch eine Funktion des Stützpunktes ist:

$$I^{[j]} = \sum_{i=1}^{N} w_i f^{[j]}(\boldsymbol{x}_i).$$

Damit dieses Vorgehen überhaupt sinnvoll ist, müssen natürlich zwei Voraussetzungen erfüllt sein:

- Erstens sollten die Nebenintegranden eine gewisse Ähnlichkeit mit dem Hauptintegranden aufweisen, da das Gitter nur für einen Integranden optimiert werden kann.
- Zweitens sollten sie leicht zu berechnen sein (zumindest sobald der Hauptintegrand vorliegt), da sonst der Rechenaufwand identisch ist mit dem wiederholten Aufruf der Integrationsroutine.

Das traditionelle Verfahren lässt jedoch zu wünschen übrig: die in der Integrandenfunktion aufsummierten Akkumulatoren müssen „per Hand“ nach dem Aufruf der Integrationsroutine ausgelesen werden und vor einem nächsten Aufruf auf Null zurückgesetzt werden. In einer threadbasierten Parallelisierung bricht es völlig zusammen, solange der Benutzer die Akkumulatoren nicht durch einen Mutex (bzw. Semaphore, Critical Section, etc.) absichert, was ihm nicht zugemutet werden kann und obendrein das Laufzeitverhalten empfindlich stören kann. Es musste daher aufgegeben werden. Stattdessen soll die Funktion einen Vektor von Ergebnissen zurückliefern: aus `double f(double x[], double wgt)` wird `void f(double x[], double f[])`, die Akkumulation wird von `pvegas` bewerkstelligt. Hierbei ist `x[]` der Stützpunkt und `f[]` der Ergebnisvektor. Mit diesem Schritt muss endgültig das gesamte Interface von `pvegas` geändert werden. So werden statt dem einen Integral `double *tgral` nun mehrere Integrale `double tgral[]` zurückgeliefert, das gleiche gilt für den Fehler und die Varianz zwischen einzelnen Iterationen.[5] Da C keine Form von Range-checking kennt und `pvegas` nun einen Vektor von Akkumulatoren verwaltet, muss die Anzahl der Nebenfunktionen mit übergeben werden.[6] Damit lautet der vollständige C-Prototyp nun (vergleiche Tabelle B.2):

```
1  void vegas(double regn[], int ndim, void (*fxn)(double x[], double f[]),
2             int init, unsigned long ncall, int itmx, int nprn,
3             int fcns, int pdim, int wrks,
4             double tgral[], double sd[], double chi2a[]);
```

[5] Tatsächlich berührt diese Änderung den Benutzer noch nicht, da in C ja $*v \equiv v$`[0]`.

[6] Man beachte, dass bei `init` < 2 die Ergebnisse der Rechnung ohnehin weggeworfen werden. Dies ermöglicht es, etwas Verwaltungsaufwand zu sparen, indem man erst im letzten Aufruf `init` $= 1$ die Anzahl der Nebenfunktionen einsetzt und sie vorher auf 1 stehen lässt.

B.4. Parallele Zufallszahlen

Die Stützpunktmenge, an der der Integrand berechnet wird, soll möglichst „zufällig" sein. Da deterministische Maschinen aber keine echten Zufallszahlen im informationstheoretischen Sinn erzeugen kann, greift man auf Pseudozufallszahlenfolgen zurück. Für einen Stützpunkt können dann D aufeinanderfolgende Zahlen als Komponenten in einem D-Vektor benutzt werden.

In einer parallelen Umgebung gibt es zwei verschiedene praktikable Ansätze:

- Ein einzelner Zufallszahlengenerator berechnet eine Folge die dann ohne Überlapp an die einzelnen Prozessoren verteilt wird.
- Jeder Prozessor hat seinen eigenen Zufallszahlengenerator und irgendein Konstruktionsprinzip trägt dafür Sorge, dass keine Korrelationen das Ergebnis verfälschen.

Korrelationen können nicht nur im zweiten Fall auftreten. Schlechte Zufallszahlengeneratoren können auch im ersten oder im nichtparallelen Fall zu Artefakten führen. Knuth [Knu 1998] listet eine Reihe von Tests um die Qualität eines Zufallszahlengenerators zu überprüfen, doch leider bestehen in der Praxis selbst einige der schlechtesten Generatoren noch alle vorgefertigten Tests. Man kann aber vermuten, dass `vegas` (und MC-Integratoren ganz allgemein) recht unanfällig gegen Artefakte in den Folgen sind im Vergleich zu anderen MC-Routinen. Das liegt nicht zuletzt an der Randbedingung, dass in unserem Falle die Stützpunkte nur innerhalb eines der Hyperwürfel im `vegas`-Gitter fallen dürfen.

Ein Blick auf Amdahls Gesetz zeigt uns, dass der zweite Ansatz mit p unabhängigen Generatoren der Attraktivere ist. Als Amdahls Gesetz bezeichnet man die Beziehung zwischen der Beschleunigung S (für engl: *speedup*), der Anzahl der CPUs p und dem Anteil des Programmes α, der parallel ausgeführt wird:

$$S = \big((1-\alpha) + p/\alpha\big)^{-1}.$$

Voneinander unabhängige Zufallszahlengeneratoren erhöhen α, was wiederum der Beschleunigung S zugute kommt.

Die einfachsten auf jedem System installierten Zufallszahlengeneratoren sind lineare Kongruenzgeneratoren. Auf einen Startwert wird eine lineare Transformation angewendet, das Ergebnis in einem Restklassenring modulo einer geeigneten Zahl betrachtet und als nächster Startwert verwendet:

$$X_{i+1} := (aX_i + b) \mod m. \tag{B.2}$$

X_j muss dann vom Benutzer in ein geeignetes Intervall aus $\mathbb{R}$ abgebildet werden. Hierin ist m üblicherweise die größte maschinendarstellbare ganze Zahl, etwa 2^{32}, und a und b müssen daran angepasst werden um möglichst die volle Periodizität m zu bekommen. Eine solche Periodizität reicht jedoch häufig nicht aus, insbesondere bei Einsatz paralleler Maschinen kommt es rasch zur Zahlenknappheit. Für `pvegas` bevorzugen wir daher eine Variation über dem Thema Schieberegistergeneratoren.[7] Ein Schieberegistergenerator ist zunächst einmal nur bitweise definiert. Er generiert binäre Pseudozufallsfolgen, indem er paarweise Bits aus einer gegebenen binären Liste mit dem exklusiven Oder verknüpft:

$$x_k := x_{k-P} \oplus x_{k-P+Q} \qquad (k \geq P).$$

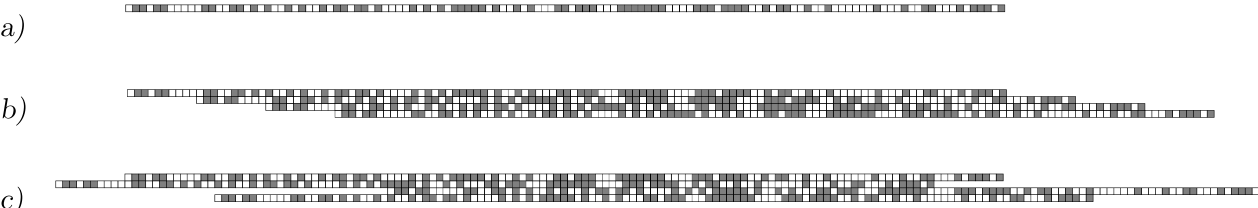


Abbildung B.3.: *(a) die 127 Bit der vollständigen Schieberegister-Reihe zum primitiven Polynom* $x^7 + x^3 + 1 \mod 2$ *und zwei verschiedene Wege, daraus Zufallszahlenfolgen aus Wörtern à vier Bit zu bilden: (b) Tausworthe-Reihe aus vier um* $O = 9$ *versetzten Schieberegister-Reihen und (c) Kirkpatrick-Stoll-Reihe initialisiert mit* $\{4, 11, 11, 4, 1, 7, 14\}$={ , , , , , , }.

Die festen Zahlen P und Q sind dabei so gewählt, dass das Trinom $1+x^P+x^Q$ primitiv modulo zwei ist.[8] Da immer nur die letzten P Elemente der Liste in die weitere Erzeugung von Bits einfließen, also auch nur die letzten P Elemente gespeichert werden müssen, lässt sich dies sehr leicht effizient implementieren. Man kann zeigen, dass die so konstruierte Reihe eine Periodizität von 2^P-1 hat. Alle Kombinationen von P aufeinanderfolgenden Bits kommen darin genau einmal vor – mit den P aufeinanderfolgenden Nullen als einziger Ausnahme. Bisher haben wir nur eine Folge von Bits wie in Abbildung B.3a grafisch angedeutet. Tausworthes Idee bestand nun darin, auf einer Maschine mit Worten zu b-Bit eine Zufallszahlenfolge zu erstellen, indem man sich b solche Bitfolgen untereinandergeschrieben denkt und die Zufallswörter spaltenweise daraus abliest. Die Generierung kann nun sehr effizient wortweise geschehen, indem man das in jeder Maschine vorhandene bitweise exklusive Oder für ganze Wörter anwendet. Die Anweisungen sind dann sehr maschinenfreundlich formulierbar. Sei `w` der Array, in dem die letzten P b-Bit-Werte des Generators gespeichert werden, und `k` ein Index, der zu Beginn des Algorithmus auf dem Anfangselement steht. In Pseudocode ausgedrückt lautet die Vorschrift dann:

```
1  ++k mod P
2  j = (k + Q) mod P
3  w[k] = w[k] ⊕ w[j]
4  ergebnis ← w[k]
```

Damit nicht alle Wörter nur aus entweder 1 oder 0 bestehen, werden die Reihen gegeneinander verschoben. Alte Generatoren dieser Art verschieben die Reihen immer um einen festen Betrag O zueinander wie in Abbildung B.3b angedeutet, indem die Anfangssequenz für jedes Bit um O gegenüber dem vorherigen Bit weitergespult wird. Die entstehende Reihe heißt Tausworthe-Reihe. Es sei darauf hingewiesen, dass für jeden Wert von O eine neue Reihe aus Wörtern zu b-Bit entsteht mit Periode 2^P, obwohl die einzelne Schieberegister-Reihe immer die gleiche bleibt. Üblich für P sind Werte zwischen 100 und 1 000, also Reihenlängen zwischen 10^{30} und 10^{300}. Das Vorspulen geschieht aus Gründen der Praktikabilität aber nur um etwa 1 000 bis 10 000 Werte. Dies galt bald als ungenügend, um Korrelationen sicher auszuschließen. In [Déak 1990] wurde daher ein Verfahren entwickelt um dieses Vorspulen anstatt in linearer in logarithmischer Zeit bewerkstelligen zu können; für eine Implementierung in C siehe [Krec 1997a].

[7] Der in der Festkörperphysik häufig verwendete R250 ist zum Beispiel auch ein Schieberegistergenerator.

[8] Eine umfassende Liste als geeignet bekannter Paare P und Q ist im Quellcode von `pvegas` dokumentiert.

Hersteller:	Architektur:	CPU:	MHz:	Betriebssystem:	$p_{\max}$:	Modell:
Convex	SPP-1200	PA-7200	120	SPP-UX 4.2	46	CPS
HP	X-class	PA-8000	180	SPP-UX 5.2	46	CPS/ Posix
Cray	T3D	EV4	150	Unicos Max 1.3.0.3	256	MPI
Siemens-Scali-Dolphin	Solaris-NoW	Pentium-II	300	SunOS 5.6	31	MPI
DEC	AlphaServer 8400 („Turbo-Laser")	EV5	300	Digital Unix 4.0	8	Posix
SGI	Origin 200	R10000	180	IRIX 6.4	4	Posix
Sun	E3000	UltraSparc	250	SunOS 5.5.1	4	Posix
Intel		IA-64	733	GNU/Linux 2.4	4	Posix
(Eigenbau)	Linux-NoW	AMD K6	233	GNU/Linux 2.0	5	MPI

Tabelle B.1.: *pvegas ist auf allen derzeit gängigen Parallelrechnern lauffähig. $p_{\max}$ bezieht sich auf die maximale Prozessoranzahl, die für Laufzeittests zur Verfügung standen.*

Doch selbst dieses Initialisieren in sub-linearer Zeit stellte sich in der Praxis rasch als unbefriedigend heraus, kann es doch noch viele Sekunden in Anspruch nehmen. Erstrebenswert wäre zudem eine Initialisierung nicht mit konstant verschobenen Bitfeldern sondern mit völlig zufällig gegeneinander verschobenen wie in Abbildung B.3c angedeutet. Dies kann aber sehr leicht in konstanter Zeit implementiert werden, wenn man bereit ist auf das Wissen um den Betrag dieser Verschiebung gegeneinander zu verzichten. Da die Schieberegister-Reihe alle Kombinationen von P aufeinanderfolgenden Bits bis auf eine genau einmal durchläuft, ist die Initialisierung der Reihe mit Zufallszahlen äquivalent zum Einspringen in die Reihe an einem nicht bekannten Punkt.[9] In der Praxis wird man die P benötigten Startworte aus den ersten P Iterationen eines linearen Kongruenzgenerators (B.2) entnehmen. Das Verfahren ist hervorragend für eine Parallelisierung geeignet, da man nur p Startfelder der Länge P nacheinander initialisieren muss und diese dann unabhängig voneinander arbeiten lassen kann. Genau wie bei der Tausworthe-Reihe, bei der für jedes O eine neue Reihe gebildet wurde, erhält man so in der Regel tatsächlich p verschiedene Reihen, jede einzelne mit der Periode 2^P.

Dieses Vorgehen ist übrigens dank der Länge der Schieberegister-Reihe auch ohne zusätzliche Überprüfung der Initialisierung hinreichend sicher. Für das in pvegas voreingestellte Paar $P = 1279$, $Q = 418$ beträgt die Periodenlänge 2^{1279}. Die Wahrscheinlichkeit dafür, dass pro Lauf in einem Wort à 32-Bit eines der Bits versehentlich überall mit 0 initialisiert wird, ist $< 10^{-375}$. Die Wahrscheinlichkeit, dass in einer Maschine wie ASCI Option White innerhalb eines Jahrhunderts irgendwo zwei Bitfolgen überlappen werden, wenn die ganze Maschine ausschließlich mit dem Erzeugen der Zufallsfolge beschäftigt wird, ist immer noch $< 10^{-350}$.

[9] Dieses Verfahren wurde unabhängig voneinander mehrfach entdeckt und erstmals in [KiSt 1981] beschrieben.

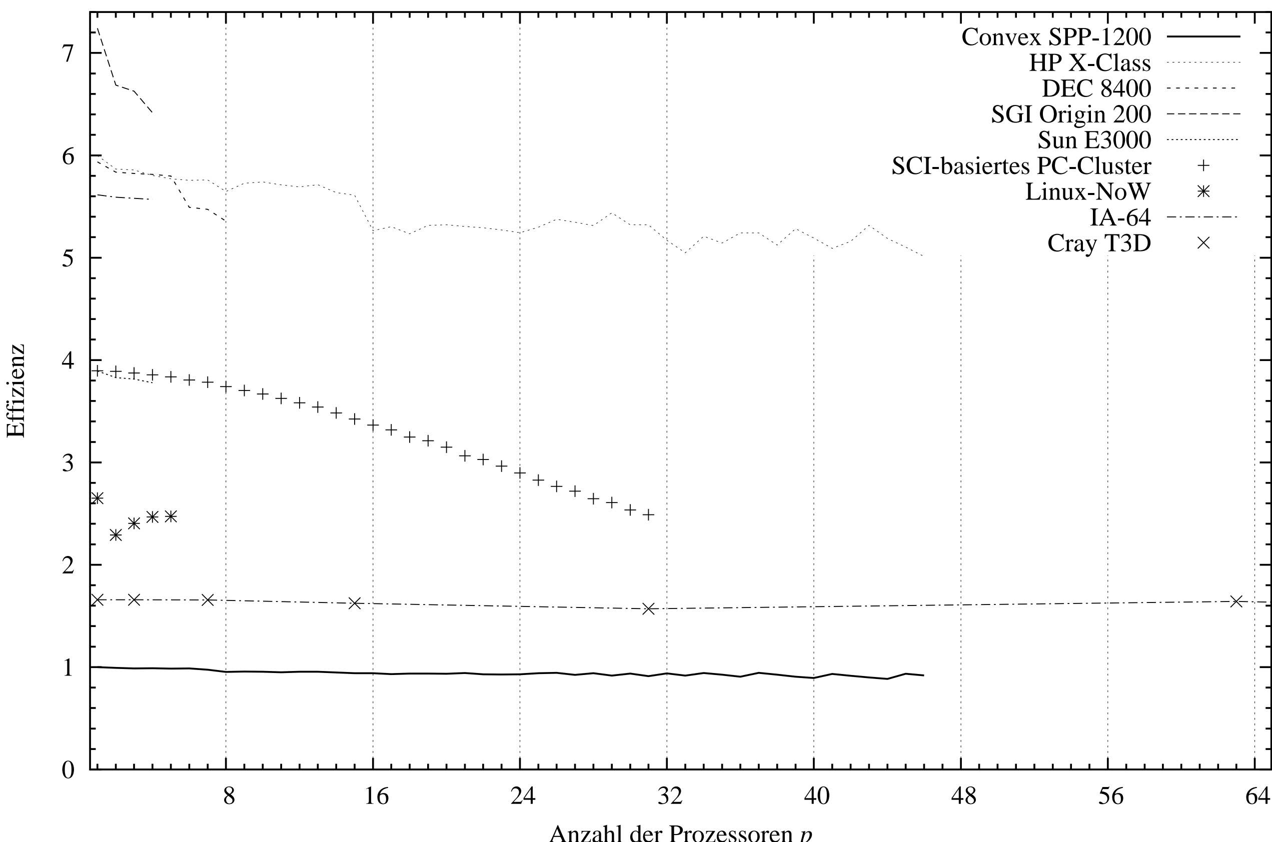


Abbildung B.4.: *Effizienz von* `pvegas` t_0^{SPP}/nt_n. *Das Optimum ist erreicht bei konstanter Effizienz. Alle Werte sind normiert auf die Convex SPP-1200 um den Graphen zu entzerren und einen Vergleich zwischen den verschiedenen Prozessortypen zu ermöglichen.*

B.5. Praktische Erfahrungen und Perspektiven

Die ursprüngliche Implementierung von `pvegas` für eine SMP-Umgebung mit geteiltem Speicher benutzt die standardisierten Posix-Threads und die dazugehörigen Mechanismen zum Verriegeln globaler Variablen. Alternativ dazu kann es auch in einer Umgebung mit CPS-Threads benutzt werden; die dafür notwendigen Änderungen beschränken sich auf syntaktische Feinheiten und können leicht vom C-Präprozessor erledigt werden. Maschinen mit verteiltem Speicher erforderten eine vollständige Neuimplementierung des Algorithmus. Diese ist aber nicht nur auf Supercomputern lauffähig, sondern auch auf sehr preisgünstigen Parallelrechnern aus vernetzten Workstations (NoW: *Network of Workstations*).

Dank des leichtgewichtigen Zufallszahlengenerators und der Möglichkeit, einige Parameter wie $D_\|$ oder die Anzahl der Funktionsaufrufe zu adjustieren, konnte `pvegas` bisher stets nahe an die optimale parallele Skalierung heranreichen. Abbildungen B.4 und B.5 zeigen die Skalierungseigenschaft sowie die Laufzeit für ein Testproblem in $D = 5$ und $D_\| = 2$.

Das wesentliche Handicap am Vegas-Algorithmus selbst stellt die Beschränkung auf rechtwinklige Gitter und damit einhergehend auf Hyperwürfel als Integrationsgebiete dar. An dieser Stelle kann jedoch verhaltener Optimismus auf bessere Verfahren geäußert werden. Zwar existieren fertige adaptive Programmpakete für numerische Quadratur, die im zweidimensionalen Fall

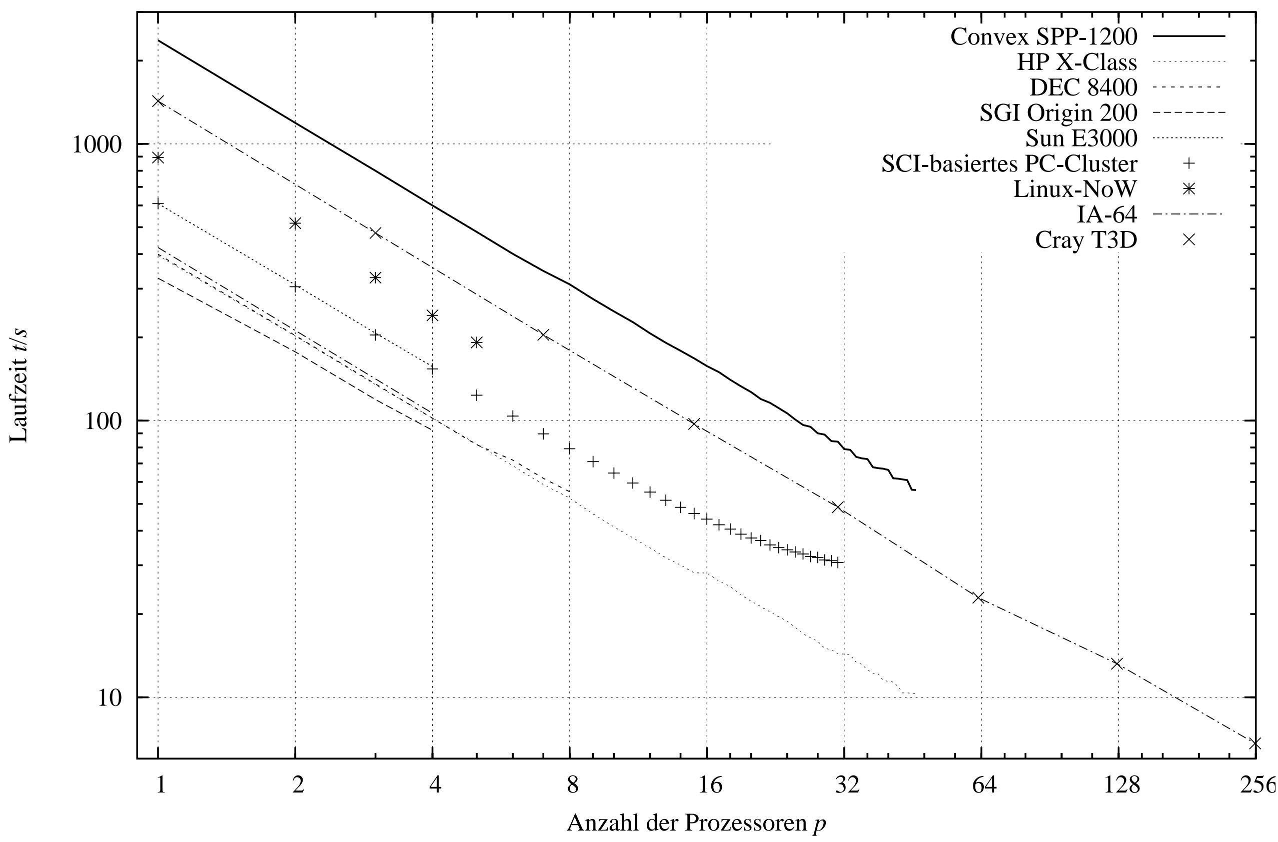


Abbildung B.5.: *Laufzeiten von* `pvegas`*.*

eine auf Dreiecken beruhende Stützpunktmenge benutzen (z.B. [DoRo 1984]), in mehr als zwei Dimensionen ist jedoch kaum Software vorhanden. Die Ursache liegt darin begründet, dass die Verallgemeinerung einer zweidimensionalen Triangulation nichttrivial ist. Für das Aufteilen eines n-dimensionalen Hyperwürfels in n-dimensionale Simplizes gibt es kein Verfahren, welches die attraktiven Eigenschaften einer rechtwinkligen Aufteilung besitzt, nämlich erstens überschaubare Subvolumina liefert und zweitens leicht berechenbar ist. Konkret bedeutet das: Es entstehen bei n-dimensionaler Triangulation mit zunehmendem n immer längere Simplizes, die dann nicht mehr das Verhalten des Integranden wiederspiegeln können, also ungeeignet für Adaptation sind. Der mittlere Tetraeder in Abb B.6 ist schon nicht mehr kongruent zu den vier Pyramiden am Rand. Selbst bei moderaten Dimensionen ist wenig bekannt über optimale Triangulationen. Bei $n = 3$ ist die aus der Methode der Finiten Elemente bekannte Delaunay-Triangulation mit 5 Simplizes noch beweisbar diejenige, bei der die Simplizes am „rundesten" ausfallen. Verlangt man die Triangulation eines höheren n-Würfels in eine minimale Anzahl von n-Simplizes, so hört unser Wissen derzeit bei $n = 7$ mit 1493 Simplizes auf [Smit 1998], darüber existieren nur sehr grobe obere und untere Grenzen. All dies ist freilich kein Argument gegen die Existenz eines Algorithmus zur Erzeugung hinreichend guter adaptiver Stützpunktmengen auf Basis von Triangulation – sie sollten jedoch die damit verbundenen Schwierigkeiten in ein realistisches Licht stellen. Integratoren

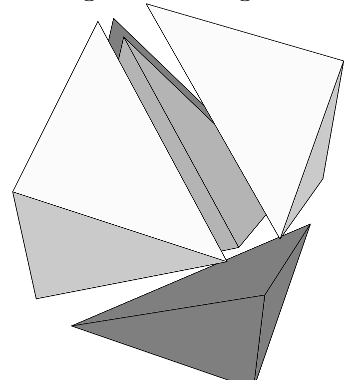
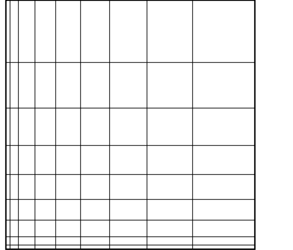
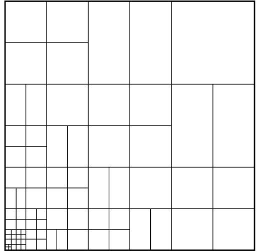
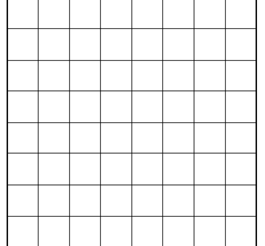
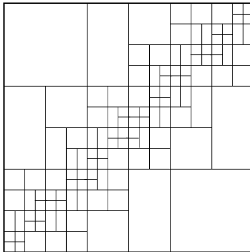


Abbildung B.6.: *Delaunay-Triangulation des 3-Würfels*

(a) *Divergenz in der Ecke unten links*

(b) *Rücken entlang der Diagonalen*

Abbildung B.7.: *Vergleich der erzeugten Gitterstruktur von Vegas (jeweils links) und ParInt (jeweils rechts) für zwei mit Monte-Carlo-Methoden schwer zugängliche Integranden.*

wie FOAM [Jada 1999] basieren dennoch auf einem durch Triangulation unterteilten Volumen, verzichten aber aus den genannten Gründen auf Optimierungen der Simplexform und Simplexanzahl. Dies führt schon in zwei Dimensionen zu einigen sehr prolongierten Dreiecken, eine Tendenz die sich bei höherdimensionalen Simplizes verstärkt. Skepsis ist angebracht, ob diese splitterförmigen Simplizes prinzipiell eine gute Adaptation an den Integranden erlauben können.

Vielversprechender als Triangulationen sind Unterteilungen des Integrationsgebietes in Hyperwürfel genau wie in Vegas, wobei die Struktur der Unterteilungen jedoch nicht in einem faktorisierenden D-dimensionalen Array von Trennlinien sondern in einer etwas komplexeren Unterteilungsbaumstruktur wie in nebenstehender Abbildung B.8 gespeichert wird. Ein solcher Algorithmus wurde in [DGBEG 1996] beschrieben – die Implementierung `ParInt` wird intensiv gepflegt und weiterentwickelt. Abbildung B.7(a) zeigt, wie in dem häufigen Fall eines Poles am Rand eines Integrationsgebietes beide Unterteilungen vernünftige Ergebnisse liefern. In Abbildung B.7(b) hingegen ist der Extremfall für `vegas` skizziert: der Integrand weist einen Rücken entlang der Diagonalen auf. Da das `vegas`-Gitter sich nur an den faktorisierenden Beitrag des Integranden adaptieren kann, bleibt jegliches *importance sampling* aus. Das von `ParInt` verwendete Gitter hat keine Probleme damit, sich an diesen Integranden anzupassen. `ParInt` hat einige Ähnlichkeiten mit `pvegas` und könnte sich daher als Ersatz eigenen. So ist das Integrationsgebiet prinzipiell ein Hyperwürfel in D Dimensionen. Es kann Nebenintegrale aufakkumulieren (hier *vector functions* genannt), was aber auch nur dann sinnvoll ist, wenn die zusätzlichen Funktionen eine Ähnlichkeit mit der Hauptfunktion haben. Außerdem existiert eine parallele Version, die an der Western Michigan University auf einem eigens dafür eingerichteten NoW aus 32 Linux-Rechnern eingesetzt wird – ein Punkt, der nicht ganz nebensächlich ist, denn er garantiert die unproblematische Verfügbarkeit für *xloops*, ohne dass der Portieraufwand überhand nimmt. Diese Parallelisierung

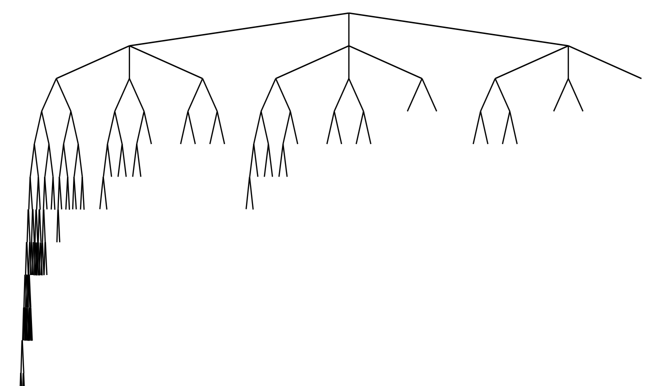


Abbildung B.8.: *Der zu dem Gitter aus Abbildung B.7(a) rechts gehörende Unterteilungsbaum.*

folgt wie `pvegas` dem Master-Slave-Modell, in dem ein Master-Prozessor die Arbeit zwischen mehreren Slave-Prozessoren balanciert. Dieses Ausbalancieren ist notwendigerweise ungemein komplexer als in unserer eleganten Aufteilung in Parallel- und Orthogonalraum. Dies erkennt man schon daran, dass der Master-Prozessor nicht nur Arbeit verteilen, sondern sie sogar einem überfrachteten Slave-Prozessor wieder abnehmen und auf einen nicht ausgelasteten Slave geben kann.

Es ist durchaus vorstellbar, dass angesichts solcher fortgeschrittenen flexiblen Integratoren für `vegas` bald nur noch ein Platz im Museum erfolgreicher Algorithmen verbleibt.

Argument:	NRC-Äquivalent:	Bedeutung:
`double regn[]`	`double regn[]`	`2ndim`-Vektor, die zwei Eckkoordinaten des `ndim`-dimensionalen Hyperwürfels: Elemente $0\dots$`ndim`-1 bezeichnen eine Ecke, `ndim`$\dots$`2ndim`-1 die andere
`int ndim`	`int ndim`	Dimension des Integranden / des Integrationsgebietes
`void (*fxn)(double x[], double f[])`	`double (*fxn)(double x[], double wgt)`	Zeiger auf Integrandenfunktion `x[]`: Koordinatenvektor des Stützpunktes `f[]`: zurückgegebene Funktionswerte, entspricht Rückgabewert in NRC `wgt`: obsolet, gibt es nur in NRC
`int init`	`int init`	Initialisierung: 0 bedeutet „Kaltstart", 1 Aufbauen auf bisherigem Gitter, 2 Aufbauen auf bisherigem Gitter und Ergebnissen
`unsigned long ncall`	`unsigned long ncall`	Anzahl der Stützpunkte pro Iteration
`int itmx`	`int itmx`	Anzahl der Iterationen in diesem Lauf
`int nprn`	`int nprn`	Details in der Ausgabe, in `pvegas` als Bitmaske vereinbart
`int fcns`	–	Anzahl der Nebenfunktionsakkumulatoren, siehe Argument `f[]` von `(*fxn)()`
`int pdim`	–	Parallelraumdimension $D_\parallel$ (0 vereinbart Automatik, also $D_\parallel = \lfloor D/2 \rfloor$)
`int wrks`	–	Parallelisierungsgrad, also Anzahl der threads oder zu benutzenden CPUs
`double tgral[]`	`double *tgral`	Zeiger auf Ergebnis, bzw. Vektor auf die Ergebnisse, falls `fcns` > 1
`double sd[]`	`double *sd`	Zeiger auf Fehler, bzw. Vektor auf die Fehler, falls `fcns` > 1
`double chi2a[]`	`double *chi2a`	Zeiger auf Varianz, bzw. Vektor auf die Varianzen, falls `fcns` > 1

Tabelle B.2.: *Argumente im `pvegas` Prototyp, ihre Entsprechung in der Numerical-Recipes Funktion `vegas` und ihre Bedeutung.*

Glossar

Dispatch: In der objektorientierten Programmierung wird ein Methodenaufruf meist durch zwei Kriterien spezifiziert: Erstens statisch durch den Namen der aufgerufenen Methode (meist augmentiert um deren Parameterliste, ⇒ Name Mangling) und zweitens dynamisch durch den Typ des Empfängerobjektes. Ob `basic::method(void)` oder `numeric::method(void)` aufgerufen wird, kann zur Laufzeit entschieden werden, je nachdem von welchem Typ das Objekt ist, auf dem `.method()` aufgerufen wird. Dies wird auch als späte Bindung (*late binding*) bezeichnet. Da C++ eine kompilierte Sprache ist, müssen aus technischen Gründen zwei Voraussetzungen erfüllt sein: `.method()` muss in der Basisklasse als `virtual` definiert sein um in der Tabelle der virtuellen Funktionen (der `vtable` der Klasse) aufgenommen zu werden und das aufgerufene Objekt darf natürlich nicht direkt sein, sondern muss als Zeiger `*` oder Referenz `&` vorliegen, da sonst eine Basisklasse nicht für eine abgeleitete Klasse einstehen kann. Jedes Objekt einer Klasse mit virtuellen Funktionen enthält zusätzlich zu seinen nichtstatischen Variablen einen `vptr` genannten Zeiger auf die klassenspezifische `vtable`.

Da nur der Typ des Empfängerobjektes dynamisch den Methodenaufruf bestimmt, wird das Objektmodell von C++ als *single-dispatch*-Modell bezeichnet. Die wenigsten Sprachen implementieren ein *multiple-dispatch*-Modell, in dem die Typen mehrerer Empfängerobjekte dynamisch die aufgerufene Funktion bestimmen. Das einzige halbwegs gebräuchliche solche System ist CLOS, das Common Lisp Object Model.

Evaluator: Ein Subsystem innerhalb eines CAS, das für *lokale* Transformationen von Termen zuständig ist. Lokalität bezieht sich hierbei auf Nachbarschaft im Darstellungsbaumes und bedeutet beispielsweise dass $(n+1)!/n!$ nicht zu $n+1$ evaluiert wird, da dies aus Sicht des Bruches eine nichtlokale Untersuchung von Zähler und Nenner voraussetzt, ein Bruch selbst aber keine Kenntnisse der Eigenschaften der Fakultät hat. Dies steht im Gegensatz zu $ab/a \to b$, was lediglich Vergleiche auf Identität erfordert, also lokale Operationen bezüglich des Bruches. Ein Evaluator kann einen Darstellungsbaum von unten nach oben (also von den Blättern zur Wurzel) oder umgekehrt arbeiten oder sogar mehrfach abarbeiten. Fixpunktevaluatoren beispielsweise arbeiten den Darstellungsbaum ab, bis Idempotenz erreicht ist („bis er sich nicht mehr ändert"), was sie den ⇒ Simplifiern ähnlich macht. Die Lokalität ist es, die den Evaluator vom Simplifier unterscheidet. Es muss darauf hingewiesen werden, dass die Definition von Lokalität nicht ganz eindeutig ist: Ist x^2 in dem Ausdruck $x^2 + 1$ ein Term in einer Summe oder eher ein Monom in einem Polynom in x? Im ersten Fall wäre x als nichtlokal anzusehen, im zweiten als lokal. Die meisten Systeme scheinen den zweiten Standpunkt einzunehmen.

Eine genaue Definition eines Evaluators wird auch dadurch erschwert, dass man in der Literatur bisweilen unter Evaluation nur diejenigen Transformationen versteht, die das System *selbsttätig* ausführt. Wir unterscheiden in dieser Arbeit stattdessen zwischen zwei verschiedenen Arten von Evaluatoren:

Anonymer Evaluator:
Man versteht ihn konstruktiv als dasjenige Subsystem innerhalb eines CAS, welches eine Eingabe auf eine der internen Weiterverarbeitung besonders zugängliche (manchmal von Schaltern abhängige) Form abbildet. ⇒ Kanonisierung von Ausdrücken zwecks syntaktischem Vergleich ist traditionell die Hauptaufgabe des anonymen Evaluators. Ein geeignetes Beispiel dafür, wie selbsttätig dies ausgeführt wird, lässt sich am besten anhand eines gängigen Missverständnisses verdeutlichen: In Mathematica führt der Versuch x/x unter der Nebenbedingung $x \equiv 0$ mit dem Befehl `Simplify[x/x, x==0]` zu vereinfachen zu dem Ergebnis 1, obwohl das System eigentlich in der Lage wäre das Ergebnis darzustellen: es lautet in Mathematica `Indeterminate`. Die Funktion `Simplify` hat jedoch gar keine Chance, dies zu finden, da der anonyme Evaluator x/x zu 1 vereinfacht, *bevor* `Simplify` aufgerufen wird. Ein anonymer Evaluator hat also die heikle Aufgabe, erste Vereinfachungen vorzunehmen, ohne jedoch die algebraische Korrektheit zu gefährden. In GiNaC ist – wie in den meisten anderen CAS – stillschweigend die Vereinbarung getroffen, dass Verstöße gegen die algebraische Korrektheit erlaubt sind, solange sie nur auf Mengen vom Maß 0 vorkommen: $x/x = 1$ gilt in $\mathbb{C} \setminus 0$ und gilt daher als erlaubt.

Benannter Evaluator:
Subsysteme, die lokal eine Umformungsregel anwenden, sofern diese vom Benutzer angefordert wird. In Maple und in GiNaC ersetzt `evalf` („`f`“ steht für *float*) beispielsweise alle Konstanten durch ihre numerischen Werte, in beiden Fällen abhängig von einem globalen Schalter: der Genauigkeit in Dezimalstellen.

Hashfunktion: Eine Hashfunktion H ist eine Abbildung, die aus einem beliebigen Eingabewert m (auch einem beliebig langem) eine Ausgabe festen Formats (also vorgegebener Bit-Länge), den Hashwert h, erzeugt: $H : m \to h = H(m)$. Hashfunktionen sind also niemals injektiv. Sie dienen dem effizienten Vergleich großer Datenstrukturen (hier: Darstellungsbäume). Man kann sie sich als „Fingerabdruck“ vorstellen: Stimmen die Hashwerte h_0 und h_1 zweier Eingabewerte m_0 und m_1 nicht überein, so sind die verglichenen Datenstrukturen unterschiedlich. Stimmen die Hashfunktionen überein, so sind die Datenstrukturen möglicherweise gleich und es lohnt sich, sie näher auf Gleichheit zu untersuchen. Beim Erzeugen von Hashwerten geht aufgrund des vorgegebenen Ausgabeformates immer Information verloren, daher kann es vorkommen, dass zu unterschiedlichen Eingabewerten gleiche Hashwerte berechnet werden (die sogenannte „Hashkollision“). Praktische Anforderungen an die Hashfunktionen von Darstellungsbäumen sind: $H(m)$ muss effizient zu berechnen sein und Kollisionen müssen selten vorkommen. In GiNaC wird die Hashfunktion nicht nur zum effizienten Vergleichen, sondern auch zum Erstellen einer Ordnungsrelation auf Ausdrücken benutzt. Zum Berechnen der Hashwerte werden Darstellungsbäume von oben nach unten durchlaufen und der Hashwert eines Ausdrucks als eine Funktion aus den gespeicherten Hashwerten der ihn aufbau-

enden Unterausdrücke sowie des Typs des Ausdruckes (des `tinfo_keys`) ermittelt. Dies begrenzt die Berechnungstiefe für einen Hashwert in der Regel auf eine einzige Ebene.

Heap: Speicherbereich zur dynamischen Allozierung zur Laufzeit. Im Gegensatz zum ⇒ Stack ist der Heap nicht prinzipiell in der Größe beschränkt. Die Verantwortung über die Verwaltung des Heaps obliegt dem Programm. Es muss dafür sorgen, dass auf dem Heap allozierter aber nicht mehr benötigter Speicher freigegeben wird.

Integritätsbereich: (auf engl: *integral domain*) Ein kommutativer, nullteilerfreier Ring. Eine Menge also, auf der Summe und Produkt definiert sind, für die Assoziativ-, Kommutativ- und Distributivgesetz gelten. Nullteilerfrei bedeutet, dass aus $a \cdot b = 0$ stets folgt, dass mindestens a oder b verschwindet. Existiert auch noch das neutrale Element der Multiplikation, so spricht man von einem *Integritätsbereich mit Eins*, was im Kontext dieser Arbeit immer der Fall ist – weshalb mit *Integritätsbereich* immer ein solcher mit Eins gemeint ist. Die Menge $\mathbb{Z}$ der ganzen Zahlen bildet einen Integritätsbereich, ebenso die Polynome darüber $Z[x]$. Integritätsbereiche sind wichtig, weil sich auf ihnen der Begriff *Teilbarkeit* definieren lässt.

Kanonisierung: Die Umwandlung der eingegebenen Form in eine der internen Weiterverarbeitung durch ein CAS besonders zugänglichen Form. Dies dient in erster Linie dem effizienten syntaktischen Vergleichen symbolischer Ausdrücke. Symbolische Summen werden beispielsweise nach einer deterministischen (aber nicht unbedingt vom Benutzer nachvollziehbaren) Weise sortiert und können so in maximal linearer Zeit verglichen werden. In den meisten Systemen (auch in GiNaC) ist dies die vom ⇒ Evaluator ausgeführte Tätigkeit. Die Kanonisierung ist auch Voraussetzung für den Vergleich unterstützt durch eine ⇒ Hashfunktion.

Name Mangling: Anders als C, welches keine überladenen Funktionen kennt, unterscheidet C++ zwischen verschiedenen Signaturen von Funktionen, also zwischen verschiedenen Parameterlisten: den Prototypen `int f(int, class a)` und `int f(int, double)` können zwei verschiedenen Implementierungen angehören. Hierzu muss der Linker zwischen den Signaturen unterscheiden können ohne die Semantik von C++ interpretieren zu müssen. Dies kann durch eine injektive Abbildung von den Signaturen auf nach bestimmten Regeln formatierte Funktionsnamen beliebiger Länge (sog. „mangled names“) geschehen. Beispielsweise bildet der C++-Compiler aus GCC-3.0 den Prototyp `int f(int, double)` ab auf `_Z1fid` und `int f(int, class a)` auf `_Z1fi1a`. Das Name Mangling unterscheidet sich von Compiler zu Compiler, um Laufzeitfehlern mit Linkerfehlern zuvor zu kommen. Leider unterscheidet es sich auch zwischen verschiedenen Compilerversionen und stellt somit die häufigste Ursache für Kompatibilitätsprobleme unter C++ dar. In obigem Beispiel bildete der C++-Compiler aus GCC-2.95.2 `int f(int, class a)` noch auf `f__FiG1a` ab.

NP-Vollständigkeit: In der Komplexitätstheorie unterscheidet man zwei wesentliche Klassen deterministischer Probleme: P und NP. Probleme aus der Klasse P können in polynomialer Zeit und mit polynomialem Speicherplatzbedarf gelöst werden, während solche aus NP nicht in polynomialer Zeit oder mit polynomialem Speicherplatzbedarf lösbar sind. Stattdessen ist die benötigte Zeit superpolynomial, also $\mathcal{O}(e^t)$ oder auch $\mathcal{O}(e^{\sqrt{t}\log(t)})$.

Man vermutet, dass dies zwei prinzipiell verschiedene Problemklassen sind, also dass NP≠P, was aber bislang unbewiesen ist. NP steht für „nichtdeterministisch polynomial", die genaue Definition davon ist hier aber irrelevant. NP-vollständige Probleme sind solche, die in polynomialer Zeit in andere Probleme aus NP transformiert werden können. Zum Beispiel ist die Berechnung der Determinante einer $n \times n$ Matrix $d \equiv \sum_{\sigma \in S_n} \mathrm{sign}(\sigma) \prod_{i=0}^{n-1} a_{i\,\sigma_i}$ offensichtlich aus P (da sie mit Eliminationsverfahren berechnet werden kann), während die Berechnung der Permanente $p \equiv \sum_{\sigma \in S_n} \prod_{i=0}^{n-1} a_{i\,\sigma_i}$ NP-vollständig ist, also derselben Komplexitätsklasse wie das Problem des Handlungsreisenden oder der Hamiltonschen Pfade angehört. Bei symbolischen Rechnungen kann diese Unterscheidung leicht irreführend sein, da die Darstellung der Operanden eines Algorithmus nicht konstant ist, im Gegensatz zum Beispiel zu Gleitkommazahlen mit fester Mantissengröße.

Persistenz: Das Schreiben von Objektstrukturen auf Festplatte zum Zweck des späteren Wiedereinlesens. Handelt es sich wie bei GiNaCs algebraischen Ausdrücken um per Zeiger verknüpfte Bäume oder gar gerichtete azyklische Graphen, so müssen diese zunächst *serialisiert* werden. Da Persistenz von Objekten sich in verschiedensten Zusammenhängen immer wieder als notwendig erweist, gibt es Forderungen nach einer Standardisierung. Da derzeit jedoch noch keine konkreten Vorschläge existieren, muss sie in jeder Klassenhierarchie neu programmiert werden.

Scope: (engl. für: *Weite, Rahmen*, vergleiche auch die englische Kurzform für *periscope*, im übertragenen Sinne „soweit man sehen kann") Programmbereich, in dem eine Variablenbindung Gültigkeit hat – also ein Modul, eine Funktion, eine Klasse oder ein sonstiger Block. Man unterscheidet zwischen den zwei gegensätzlichen Ansätzen:

Dynamisches Scope:
Auf eine Variable kann nicht nur in dem Block zugegriffen werden, in dem sie deklariert wurde, sondern auch in darin aufgerufenen Funktionen. Dies wird im ⇒ Stack implementiert, indem er von oben nach unten nach der letzten gültigen Bindung durchsucht wird.

Lexikalisches Scope:
Eine Variable ist tatsächlich nur in dem Block erreichbar, in dem sie deklariert wurde. Soll sie in aufgerufenen Funktionen sichtbar sein (auch wenn es sich um rekursive Aufrufe derselben Funktion handelt), so muss sie als Parameter übergeben werden.

Traditionell findet dynamisches Scope in den meisten interpretierten Sprachen Verwendung, also insbesondere in alten Lisp-Implementierungen sowie in fast allen Computeralgebrasystemen. Compilierte Sprachen wie C++ zwingen den Programmierer üblicherweise zu lexikalischem Scope. Manche Sprachen unterstützen beide Bindungen (Perl zum Beispiel bietet den Modifikator `my` für lexikalisches Scope und `local` für dynamisches.). Dynamisches Scope wird heute allgemein als verwirrend angesehen, weshalb zum Beispiel MuPAD im Frühjahr 2001 für die zweite Version ihres CAS von dynamischem auf lexikalisches Scope umgestellt hat – trotz der mit der Umstrukturierung der gesamten Bibliothek verbundenen Probleme.

Stack: Speicherbereich zum automatischen Erzeugen von Variablen innerhalb von Funktionen, zum Übergeben von Parametern an diese sowie zum Übergeben des Rückgabewertes an die aufrufende Funktion. Verwaltet wird er als FILO („*first in last out*“), weshalb ein Stack beliebige Objekte mit dem minimalen Speicherbedarf verwaltet – allerdings mit linearer Zugriffskomplexität. Die Größe des Stacks wird von den meisten Betriebssystemen sehr beschränkt, größere Objekte müssen auf dem ⇒ Heap alloziert werden.

Simplifier: Der Simplifier, also der Vereinfacher, führt in einem CAS in der Regel *nichtlokale* Transformationen aus, was ihn vom ⇒ Evaluator unterscheidet. Ein Simplifier kann also $(n+1)!/n! \to n+1$ vereinfachen. Dies erscheint zwar noch als intuitiv, nach welchem Maß aber ein Ergebnis einfacher ist als ein anderes, lässt sich überhaupt nicht definieren. Ebenso wie von einem Fixpunktevaluator erwartet man von einem Simplifier in der Regel Idempotenz: `simp(simp(`*Ausdruck*`))=simp(`*Ausdruck*`)`. Ein Simplifier kann eventuell auch unter Nebenbedingungen arbeiten, wie zum Beispiel der Annahme eines bestimmten Bereiches. Wie im Falle des Evaluators lässt sich zwischen anonymen und benannten Varianten unterscheiden. Der anonyme Simplifier wird traditionell mit `simplify(`*Ausdruck*`)` aufgerufen, während benannte Simplifier wohldefinierte Termumformungen innerhalb eines Bereiches vornehmen – das Herauskürzen von ggT von Zähler und Nenner in einem Quotientenkörper mittels `normal(`*Ausdruck*`)` ist solch ein Beispiel. Anonyme Simplifier dagegen führen selten wohldefinierte Transformationen durch und in Abwesenheit eines Maßes für die Komplexität eines Ausdrucks können die auf einem Ausdruck durchgeführten Transformationen und damit auch die Ergebnisse variieren – beispielsweise als Funktion der Versionsnummer eines CAS. Wenn mit den Ergebnissen weitergearbeitet werden soll, macht dies den Aufruf eines anonymen Simplifiers innerhalb eines komplexen Programmes zu einem im besten Falle fragwürdigen Unterfangen.

Schlagwortverzeichnis

A

Ableitungsregeln . . . 74, 75
Abreicherung . . . 67, 72
Acnode . . . 18, 25, 26
Adapter . . . 84
adaptives Verfahren . . . 151
`add` Klasse . . . 74
Algol . . . 60, 62
AMD . . . 158
Amdahls Gesetz . . . 156
Apfloat . . . 86
Arcuscosinus . . . 145
Arcussinus . . . 144
Arcustangens . . . *141*
Array . . . 60
  assoziativer . . . 66, 101, 122
ASCI . . . 158
Aslaksen-Test . . . 77
Axiom . . . 58, 129

B

B . . . 60
Bézout-Identität . . . 136
Baumebene . . . 1
Baumrekursion . . . 74, *123–129*
BCPL . . . 60
Benchmark . . . 75, 115
Bernoulli-Zahlen . . . 96
binary splitting . . . 86
Blatt . . . 139
bottom-up . . . 74, 91
Bridge . . . 70, 84, 88

C

C . . . 1, 3, 59, 64, 86
C++ . . . *59–63*
Cache . . . 76
Cache-Affinität . . . 102
Cauchy'sche Integralformel . . . *134*
Cauchy'scher Integralsatz . . . *133*
Cauchy'scher Residuensatz . . . 20, *134*
Cauchy-Riemann'sche DGL . . . 133
Cern . . . 120
Charakteristik . . . 44
charakteristisches Polynom . . . 104, 115
chirale Störungstheorie . . . 18, 25
Cint . . . *120–122*
CLN . . . *83–88*, 119
Codegenerierung . . . 64, 69
CompHEP . . . 4
Containerklasse . . . 41, 72, 74, 77
Convex . . . 158
Cramer'sche Regel . . . 108
Cray . . . 158
CSE . . . 69
Ctadel . . . 69

D

d'Alembert-Operator . . . 17
Darstellung
  distributive . . . 73
Darstellungsbaum . . . *72–74*, 91, 127, 166
DEC . . . 158
Deformationssatz . . . 133
`degree()` . . . 92
Delaunay-Triangulation . . . 160
Delegation . . . 65, 127
de Morgan'sche Regeln . . . 44
Derive . . . 56
Design Muster
  Adapter . . . 84
  Bridge . . . 70, 84, 88

Delegation 65
Flyweight 66, 87, 122
Proxy 65
Visitor 124
Determinante 32, 100f, 111
Diana 3
Differentiation 74
Effizienz und 75
Digamma-Funktion 96
Dilogarithmus 92ff, *146*
Dispatch *165*
Distributivgesetz 79
Dreibeinfunktion
gekreuzte 18, 23
planare 13, 18
Dreiecksmatrix 105, 110
Dreiecksregel 16
dynamisches Scope 168

E

Elimination
Bareiss . *siehe* Elimination, teilerfreie
divisionsfreie 31, 105
Gauß 31, 105, 106
teilerfreie 31, 106, 109
Entwindungszahl 78, 141
η-Funktion 78, 141
euklidischer Algorithmus 136
Euler-Algorithmus 90
Euler-Zahlen 115, 121
`eval()` 81, 82, 88, 112
`evalf()` 88, 166
`evalm()` 112
Evaluator *165*
anonymer . . 41, 65, 72, 78, 112, *166*
benannter 112, *166*
event handler 88
`ex` Klasse 41, *65*, 128
Exception 62, 91, 95
`expairseq` Klasse 73
`expand()` 75, 104
Exponentiation
schnelle 69, 112
von Reihen 90
`extern` 122

F

Fünfbeinfunktion 18, *33*
Fabrik 87, 122
Fadeev-Algorithmus . *siehe* Leverrier-Algo.
Faktorisierung
quadratfreie *42*, 64, 117
Fermat 58, 114
Feynmanparametrisierung 13f, 25
Fixpunktevaluator 165, 169
Fliegners Test 75, 113, 115, 121
Flyweight 66, 87, 122
Foam 161
Form . . 1, 3, 16, 75, 83, 114, 115, 127
Fortran 3, 60, 86
Fortsetzung
analytische 94, 139
foundation class 86
`function` Klasse *88–89*
funktionsartiges Objekt . . *siehe* Funktor
Funktor 40, 88, 124
Fusion (von Referenzen) 67

G

Gamma-Funktion 17, 94
Garbage-Collector 76
GCC 118
ggT 30, 42, 85, 110, 115
giac 128
GiNaC *55–112*
GMP 85, 86, 128
größter gemeinsamer Teiler . . *siehe* ggT
Grace 2, 6
Graphengenerator 5
GRC 6
gTybalt 128

H

Hashfunktion 67, *166*
Hashwert 67, 75, *166*
Hauptteil 134, 137
Hauptwertintegral *137–138*
Heap 61, 65, *167*
Fragmentierung 117
Heaviside-Funktion . . . *siehe* θ-Funktion

Heuristik . . . 111
Hilbert-Matrix . . . 99, 115
HP . . . 158
Hyperwürfel . . . 156ff

I

IBM 7094 . . . 1
Idempotenz . . . 43ff, 165, 169
Impulsentwicklung . . . 16
`indexed` Klasse . . . 99
Inferenzmaschine . . . 45, 78
`integer_content()` . . . 40, 42
Integrabilitätsbedingung . . . 133
Integritätsbereich . . . 38, 83, 110, *167*
Intel . . . 119
Invariante . . . 92
Invertierung
  von Matrizen . . . 105, 111
  von Reihen . . . 90

J

Jacobi-Determinante . . . 14, 29
Java . . . 64, 128
Jonquière-Funktion . . . 146

K

$\mathcal{K}$-Funktion . . . *siehe* Entwindungszahl
KAI . . . 119
kanonische Form . . . 77
Kanonisierung . . . 88, 104, *167*
Kant . . . 87
Kausalität . . . 11
Keim . . . 139
kgV . . . 92, 136
Klassenhierarchie . . . 65, 84
kleinstes gmeinsames Vielfaches *siehe* kgV
Komplexität . . . 102, 124
Kongruenzgenerator . . . 156
Körpereinbettung . . . 87
Körperoperation . . . 102
Kovarianz . . . 17
Kreimer-Rotation . . . *28*

L

Lagrangedichte . . . 11, 13, 18
Lambda-Kalkül . . . 99
Laplace-Entwicklung . . . 103
Laurent-Reihe . . . 14, 89ff
lazy evaluation . . . 90, 99
`ldegree()` . . . 92
Leibniz-Regel . . . 75
Leverrier-Algorithmus . . . 104, 107
lexikalisches Scope . . . 168
`lgamma()` . . . 95
l'Hôpital . . . 93
Linker . . . 118
LIP . . . 86
Lisp . . . 1, 64, 99
Liste . . . 57
Logarithmus . . . 80, 94, 140
Lokalität
  von Umformungen . . . 165, 169
`lst` Klasse . . . 125

M

m4 . . . 4
Macsyma . . . 100, 127
Magma . . . 87
Makro . . . 3, 62
Mandelstam-Variablen . . . 16, 26
Mantisse . . . 86
`map()` . . . 124
Maple . . . 69, 79, 91, 94, 110, 127
Masse-Energie-Beziehung . . . 11
Master-Slave . . . 154, 162
Mastertopologie . . . 18
Mathematica1, 68, 72, 76, 85, 94, 110, 123, 127
Matlab . . . 98, 128
`matrix` Klasse . . . *98–112*
Maxima . . . 100, 128
Methode . . . 61
Methodenfortpflanzung . . . *74–76*, 91
Modularität . . . 122
Monte Carlo . . . 5, 151ff
Motorola 68000 . . . 1
MPFun . . . 86

MPI . . . 158
MPN . . . 85
`mul` Klasse . . . 74, *78–79*
Multiplikation
  Karatsuba . . . 84
  Schönhage-Strassen . . . 84
MuPAD . . . 76, 128, 168
Mustererkennung . . . 123
`mutable` . . . 69
Mutex . . . 155

N

name mangling . . . 62, *167*
`ncmul` Klasse . . . *78–79*, 99
Nebenintegral . . . 155
NoW . . . 159, 161
NP-vollständig . . . 100, 167
NTL . . . 6, 86
`numeric` Klasse . . . *83–88*

O

Objektorientierung . . . 62
Octave . . . 128
Operatorüberladung . . . 64
Optimierung . . . 69
Orthogonalität . . . 5, 59, 62, 111
Orthogonalraum . . . 15, 154
Overhead . . . 64, 85, 87, 99

P

Padding . . . 63
Padé-Approximation . . . 17
Parallelraum . . . 14, 154
PARI . . . 6, 61, 96
`ParInt` . . . 161
Partialbruchzerlegung . . . 22, 136
partielle Integration . . . 15–16
pattern matching . . . 123
Perl . . . 65f, 70, 128, 168
Permanente . . . 100, 168
Permutationsgruppe . . . 103
Persistenz . . . 59, 168
Pivotelement . . . 105
PO-Zerlegung . . . 15
Polygamma-Funktion . . . 96
Polylogarithmus . . . 148
Portabilität . . . 64, *118*
Portland Group . . . 119
`power` Klasse . . . *79–83*
Prädikatenlogik . . . 43
Präprozessor . . . 159
Primfaktorzerlegung . . . 80
Proxy . . . 65
`pseries` Klasse . . . *89–98*
Pseudofunktion . . . 63, *88–89*, 123
`psi()` . . . 96
ψ-Funktion . *siehe* Polygamma-Funktion
Puffer . . . 59, 84
Puiseux-Reihe . . . *92*
PURRS . . . 128
pyginac . . . 128
Python . . . 66, 128

Q

QED . . . 11
QGRAF . . . 5, 19
quadratfreie Faktorisierung . *42*, 64, 117
Qual der Wahl . . . 35
Quotientenkörper . . 38, 83, 90, 110, *169*

R

range-checking . . . 101
rapid prototyping . . . 120
RB-Baum . . . 99, 101, 123
Record . . . 61
REDUCE . . . 1, 3, 58, 77, 79, 128
Referenz . . . 65
  zirkuläre . . . 70
Referenzzählung . . . *66–72*, 76, 87, 115
Regressionstest . . . 2f, 57f, 81
Regularisierung
  dimensionale . . . 5, 14, 92, 95, 98
Reihe
  Kirkpatrick-Stoll . . . 157
  Schieberegister . . . 157
  Sobol' . . . 152
  Tausworthe . . . 157
Rekursion . . . 59, 96, 124

unendliche 88
Residuensatz 20, *134*
Residuensumme
Satz über 21, 29, 33, 35, *134*
Restklassenring 156
Retraktion 87
Riemannfläche 139
Ringoperation 16, 127
ROOT 120, 128
RTTI 67
Russische Bauernmultiplikation . . . 112

S

sampling
importance 151ff
stratified 151ff
Scheme 99, 128
Schieberegister 156
Schleifenumordnung 102
Schnitt *138–148*
Schoonschip 1, 83
Schwelle 17
Scope 56, 71, 122, *168*
dynamisches *168*
lexikalisches 57, 168
Semaphore 155
SGI 158
Simplex 160
Simplifier 127, *169*
Simula 61
Singular 6, 115
Smith-Normalform 114
SMP (CAS) 1
SMP (*symmetric multiprocessing*) . . 159
Sokhotsky-Plemelj-Relationen 50
späte Bindung 165
sparse matrix 101
Speicherleck 67f, 117f
Spence-Funktion 146
Spur 104
Stack 60, 61, *168*
Staffelmatrix 110
Standardmodell 2, 11
`static` 60, 122
`status_flags`
`dynallocated` 67
`expanded` 114
STL 62, 101, 117, 122, 126, 129
Stützpunktmenge 151ff
`.subs()` 123, 128
Substitution
syntaktische 125
Sumit 56
Sun 158
Sylvester-Identität 31ff, *107*
`symbol` Klasse 128
Symmetriefaktor 19

T

Taylor-Reihe 56, 89ff
Tcl/Tk 2f
Template 62, 101, 118
ternäre Logik 78
TeXmacs 128
`tgamma()` 95
θ-Funktion 41ff, 94, 135, 141
`this` 62
Threads 159
TM 4
top-down 74, 124, 127
Träger 11
transzendente Funktion . 86, 94, *138–148*
Triangulation 160
Typsicherheit 64

U

Überladung 60, *61*, 64, 122
Unix 60, 61
Unterausdruck 69
Unterteilungsbaum 161

V

Variable
linearisierte *20*, 27
zugeordnete *20*, 27, 45
Verdopplungsformel 98
Vergleich
syntaktischer 75, 109, 166, 167
Verzweigungspunkt *94*, 141, 146

`vptr` 65, *165*
`vtable` 88, *165*

W

Wegintegral 147
Wick-Rotation 13, 29
`wildcard` Klasse 123
Wrapper 65

X

xloops 2ff, 128
XML 6

Y

Yacas 78
Yun'scher Algorithmus 64

Z

ζ-Funktion 96
Zufallszahlen 156–158
Zwillingsvariablen *30, 39*

Literaturverzeichnis

Zum Aufbau dieses Literaturverzeichnisses sei entschuldigend angemerkt, dass es mit dem spürbaren Niedergang der Bedeutung gedruckter (und referierter!) Zeitschriften immer schwieriger wird, eine „kanonische“ Form einzuhalten. Mehr und mehr Autoren gehen dazu über, auch anspruchsvolle Arbeiten höchstens noch in elektronischen Preprint-Foren (sog. „Newsletters“) zu publizieren. Häufig sogar nicht einmal das, stattdessen weisen sie nur auf ausdruckbare Papers hin, die auf ihren Webseiten stehen. Man steht nun vor der Wahl, entweder gar nicht zu zitieren – was auf Vorenthaltung einiger wichtiger Quellen hinausliefe – oder zu improvisieren. Ich habe mich bewusst für letztere Alternative entschieden, stets in der Hoffnung, dass die Informationen mit der Zeit nicht unauffindbar werden.

[AbS 1972] Milton Abramowitz, Irene A. Stegun (Hrg.): *Handbook of Mathematical Functions With Formulas, Graphs and Mathematical Tables*; Dover, New York

[AGORT 2000] Charalampos Anastasiou, Thomas Gehrmann, Carlo Oleari, Ettore Remiddi, Jan B. Tausk: *The Tensor Reduction and Master Integrals of the Two-Loop Massless Crossed Box With Light-Like Legs*; Nucl. Phys. **B**580, 577-601; arXiv:hep-ph/0003261

[Asla 1996] Helmer Aslaksen: *Multiple-valued Complex Functions and Computer Algebra*; SIGSAM Bulletin, **30/2**, 1996, 12-20

[ATT 1986] AT&T Bell Laboratories: *System V Interface Definition, Issue 2*;

[ATT 1989] AT&T Bell Laboratories: *System V Interface Definition, Issue 3*;

[Bar 1968] Erwin H. Bareiss: *Sylvester's Identity and Multistep Integer-preserving Gaussian Elimination*; Math. Comput. **22/103**, 565-578

[Bern 2002] Daniel J. Bernstein: *Integer multiplication benchmarks*; `http://cr.yp.to/speed/mult.html`

[Baue 2000] Christian Bauer: *Der xloops-Algorithmus zur Berechnung von Feynman-Graphen in C++*; Diplomarbeit, Mainz

[BCK 2001] Pavel Baikov, Konstantin G. Chetyrkin, Johann H. Kühn: *The Cross Section of e^+e^- Annihilation Into Hadrons of Order $\alpha_s^4 n_f^2$ in Perturbative QCD*; Phys. Rev. Lett. **88**/1, 012001; arXiv:hep-ph/0108197

[BDIPS 1994] Edward E. Boos, Mikhail N. Dubinin, Viacheslav A. Ilyin, Alexander E. Pukhov, Victor I. Savrin: *CompHEP - Specialized Package for Automatic Calculations of Elementary Particle Decays and Collisions*; SNUTP 94-116; INP MSU-94-36/358; arXiv:hep-ph/9503280

[Bèla 1999] Geneviève Bélanger, Fawzi Boudjema, Jumpei Fujimoto, Tadashi Ishikawa, Toshiaki Kaneko, Kiyoshi Kato, Vincent Lafage, N. Nakazawa, Yoshimisu Shimizu: *Implementation of the Non-Linear Gauge Into GRACE*; Proc. AIHENP-99, Heraklion, Griechenland

[BePa 1998] Clemens Bellarin, Lawrence C. Paulson: *Reasoning about Coding Theory: The Benefits We Get From Computer Algebra*; Proc. AISC-98, 55-66

[BeSo 1965] Heinrich Behnke, Friedrich Sommer: *Theorie der analytischen Funktionen einer komplexen Veränderlichen* (dritte Auflage: 1965); Springer, Berlin

[BFK 1995] Lars Brücher, Johannes Franzkowski, Dirk Kreimer: *A New Method for Computing One-Loop Integrals*; Comp. Phys. Comm. **85**, 153-165; arXiv:hep-ph/9401252

[BFK 1998] Lars Brücher, Johannes Franzkowski, Dirk Kreimer: *XLoops: Automated Feynman Diagram Calculation*; Comp. Phys. Comm. **115**, 140-160

[BFK 2001a] Christian Bauer, Alexander Frink, Richard Kreckel: *The GiNaC Framework for Symbolic Computation Within the C++ Programming Language*; Proc. CALCULEMUS-2000 Symposium, St. Andrews, Schottland

[BFK 2002a] Christian Bauer, Alexander Frink, Richard Kreckel: *Introduction to the GiNaC Framework for Symbolic Computation Within the C++ Programming Language*; Journal of Symbolic Computation, **33**, 1-12; arXiv:cs-sc/0004015

[BFK 2002b] Christian Bauer, Alexander Frink, Richard Kreckel: *GiNaC*; in: Johannes Grabmeier, Erich Kaltofen, Volker Weispfenning (Hrsg.): *Computer Algebra Handbook*; Springer, Heidelberg

[BFT 1993] David J. Broadhurst, Jochem Fleischer, Olev V. Tarasov: *Two-Loop Two-Point Functions With Masses: Asymptotic Expansions and Taylor Series, in Any Dimension*; Z. Phys., **C 60** 287-302; arXiv:hep-ph/9304303

[Bhat 1996] Gaurav Bhatnagar: *A Short Proof of an Identity of Sylvester*; Internat. J. Math. & Math. Sci. **22/2**, 431-435

[Bier 2000] Kay Bieri: *NNLO Calculations in $\gamma\gamma \to \pi\pi$* ; Diplomarbeit, Bern; siehe URL: `http://www-itp.unibe.ch/thesis/bieri/diplom.ps`

[BKK 2001] Isabella Bierenbaum, Richard Kreckel, Dirk Kreimer: *On the Invariance of Residues of Feynman Graphs*; arXiv:hep-th/0111192

[BCP 1997] Wieb Bosma, John Cannon, Catherine Playoust: *The Magma Algebra System I: The User Language*; Journal of Symbolic Computation, **24**, 235-265

[Bron 1996a] Manuel Bronstein: *Symbolic Integration I*; Springer, Heidelberg

[Bron 1996b] Manuel Bronstein: Σ^{IT} – *A Strongly-Typed Embeddable Computer Algebra Library*; Proc. DISCO-96, Karlsruhe, 1128, Springer

[Brue 1997] Lars Brücher: *Automatische Berechnung von Strahlungskorrekturen in perturbativen Quantenfeldtheorien*; Dissertation, Mainz; siehe URL: `http://wwwthep.physik.uni-mainz.de/Publications/theses/dis-bruecher.ps.gz`

[Cart 1963] Henri Cartan: *Théorie Élémentaire des Fonctions Analytiques d'une ou plusieurs Variables Complexes*; Hermann, Paris

[CDJLW 2001] Robert M. Corless, James H. Davenport, David J. Jeffrey, Gurjeet Litt, Stephen M. Watt: *Reasoning About the Elementary Functions of Complex Analysis*; in: *Lecture Notes in Computer Science* **1930**, 115-126, Springer, Berlin

[ChTk 1981] Konstantin G. Chetyrkin, Fyodor V. Tkachov: *Integration by Parts: The Algorithm to Calculate β-Functions in 4 Loops*; Nucl. Phys. **B**192, 159-204

[CKK 1994] Andrzej Czarnecki, Ulrich Kilian, Dirk Kreimer: *New Representation of Two-Loop Propagator and Vertex Functions*; Nucl. Phys. **B**433, 259-275; arXiv:hep-ph/9405423

[Coh 2000] Christian Batut, Karim Belabas, Dominique Bernardi, Henri Cohen, Michel Olivier: *User's Guide to Pari-GP*; (Version 2.0.19), siehe URL: `ftp://megrez.math.u-bordeaux.fr/pub/pari/`

[CoTr 1995] Michel Cosnard, Denis Trystram: *Parallel Algorithms and Architectures*; Thomson, London

[CoWi 1990] Donald Coppersmith, Shmuel Winograd: *Matrix Multiplication via Arithmetic Progressions*; Journal of Symbolic Computation, **9/1**, 251-280

[DaST 1993] Andrey I. Davydychev, Vladimir A. Smirnov, Jan B. Tausk: *Large Momentum Expansion of Two-Loop Self-Energy Diagrams With Arbitrary Masses*; arXiv:hep-ph/9307371

[DaT 1992] Andrey I. Davydychev, Jan B. Tausk: *Two-Loop Self-Energy Diagrams With Different Masses and the Momentum Expansion*; Nucl. Phys. **B**397, 123-142

[Dave 2000] James H. Davenport: *Abstract Data Types in Computer Algebra*; in: *Lecture Notes in Computer Science* **1893**, 21-35, Springer, Berlin

[Déak 1990] Istvan Deák: *Uniform Random Number Generators for Parallel Computers*; Parallel Computing, **15**, 155-164

[DGBEG 1996] Elise de Doncker, Ajay Gupta, Jay Ball, Patricia Ealy, Alan Genz: *ParInt: A Software Package for Parallel Integration*; Proc. of the 10th ACM International Conference on Supercomputing, 149-156

[DoRo 1984] Elise de Doncker, Ian Robinson: *TRIEX: Integration Over a TRIangle Using Nonlinear EXtrapolation*; ACM Transactions on Mathematical Software, **10/1**, 17-22

[DST 1988] James H. Davenport, Yvon Siret, Evelyne Tournier: *Computer Algebra—Systems and Algorithms for Algebraic Computation*; Academic Press Ltd., London

[Enge 1998] Robert A. van Engelen: *Ctadel: A Generator of Efficient Numerical Codes*; Dissertation, Leiden; siehe URL: `http://www.cs.fsu.edu/~engelen/thesis.ps.gz`

[FaHa 1996] Richard J. Fateman, Mark Hayden: *Speeding up Lisp-Based Symbolic Mathematics*; SIGSAM Bulletin, **30/1**, 1996, 25-30

[Fate 1990] Richard J. Fateman: *Advances and Trends in the Design and Construction of Algebraic Manipulation Systems*; Proc. ISSAC-90, Tokyo

[Fate 1999] Richard J. Fateman: *Symbolic Mathematics System Evaluators*; in: Michael J. Wester (Hrsg.): *Computer Algebra Systems – A Practical Guide*; Wiley, Chichester

[Fate 2001] Richard J. Fateman: *Manipulation of Matrices Symbolically*; Unveröffentlicht, `http://www.cs.berkeley.edu/~fateman/temp/symmat.pdf`

[FKT 1997] Alexander Frink, Jürgen G. Körner, Jan B. Tausk: *Massive Two-Loop Integrals and Higgs Physics*; arXiv:hep-ph/9709490

[FlTe 1999a] Mikhail Tentyukov, Jochem Fleischer: *A Feynman Diagram Analyser DIANA*; Comp. Phys. Comm. **132**, 124-141; arXiv:hep-ph/9904258

[FlTe 1999b] Mikhail Tentyukov, Jochem Fleischer: *DIANA, A Program for Feynman Diagram Evaluation*; arXiv:hep-ph/9905560

[FlTe 2000] Jochem Fleischer, Mikhail Tentyukov: *A Feynman Diagram Analyser DIANA – Graphic Facilities*; arXiv:hep-ph/0012189

[Fran 1997] Johannes Franzkowski: *Virtuelle Strahlungskorrekturen im Standardmodell der Elementarteilchenphysik*; Dissertation, Mainz; siehe URL: `http://wwwthep.physik.uni-mainz.de/Publications/theses/dis-franzkowski.ps.gz`

[Frin 1996] Alexander Frink: *Massive Zwei-Loop Vertexfunktionen*; Diplomarbeit, Mainz; siehe URL: `http://wwwthep.physik.uni-mainz.de/Publications/theses/dip-frink.ps.gz`

[Frin 2000] Alexander Frink: *Computer-algebraische und analytische Methoden zur Berechnung von Vertexfunktionen im Standardmodell*; Dissertation, Mainz; siehe URL: `http://wwwthep.physik.uni-mainz.de/Publications/theses/dis-frink.ps.gz`

[FSF 2001] Free Software Foundation: *libstdc++-v3 Documentation*; siehe URL: `http://gcc.gnu.org/onlinedocs/libstdc++/documentation.html`

[Fuch 1997] Benno Fuchssteiner et alii (MuPAD group): *MuPAD User's Manual, MuPAD*; (Version 1.4) Wiley, Chichester, siehe URL: `http://www.mupad.de/`

[GCL 1992] Keith O. Geddes, Stephen R. Czapor, George Labahn: *Algorithms for Computer Algebra*; Kluwer, Norwell, Massachusetts

[GeJo 1976] W. Morven Gentleman, Stephen C. Johnson: *Analysis of Algorithms, A Case Study: Determinants of Matrices With Polynomial Entries*; ACM Transactions on Mathematical Software, **2**/**3**, 232-241

[GHJV 1995] Erich Gamma, Richard Helms, Ralph Johnson, John Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*; Addison-Wesley, Reading, Massachusetts

[GKP 1989] Ronald L. Graham, Donald E. Knuth, Oren Patashnik: *Concrete Mathematics*; Addison-Wesley, Reading, Massachusetts

[GPS 2000] Gert-Martin Greuel, Gerhard Pfister, Hans Schönemann: *Singular 1.3.7. A Computer Algebra System for Polynomial Computations*; Zentrum für Computer Algebra an der Universität Kaiserslautern, URL: `http://www.singular.uni-kl.de/`

[Groz 2001] Andrey Grozin: *TeXmacs Interfaces to Maxima, MuPAD and REDUCE*; arXiv:cs.SC/0107036

[GrRy 1994] Izrail' Solomonovich Gradshteĭn, Iosif Moiseevich Ryzhik; *Table of Integrals, Series, and Products*; (fifth edition); Academic Press, London

[HaKr 2000] Bruno Haible, Richard Kreckel: *CLN, a Class Library for Numbers*; (Version 1.1), siehe URL: `http://www.ginac.de/CLN/`

[HaPa 1998] Bruno Haible, Thomas Papanikolaou: *Fast Multiprecision Evaluation of Series of Rational Numbers*; in: Joe P. Buhler (Hrsg.): *Lecture Notes in Computer Science*; **1423**, Springer, Heidelberg

[HaSt 1998] Robert Harlander, Matthias Steinhauser: *Automatic Computation of Feynman Diagrams*; TTP98-41; arXiv:hep-ph/9812357

[Hear 1995] Anthony C. Hearn: *REDUCE User's Manual Version 3.6*; RAND, Santa Monica, siehe URL: `http://www.zib.de/Symbolik/reduce/`

[Hoar 1981] Charles A. R. Hoare: *Turing Lecture "The Emperor's Old Clothes"*; Comm. ACM **24(2)** 75-83

[Horo 1971] Ellis Horowitz: *Algorithms for Partial Fraction Decomposition and Rational Function Integration*; Proc. Second Symposium on Symbolic and Algebraic Manipulation, ACM Inc., 1971, 441-457

[ItZu 1993] Claude Itzykson, Jean-Bernard Zuber: *Quantum Field Theory*; World Scientific Lecture Notes in Physics

[ISO 1990] ISO/IEC 9899:1990: *Programming Languages—C*; American National Standards Institute, 1990

[ISO 1998] ISO/IEC 14882:1998(E): *Programming Languages—C++*; American National Standards Institute, 1998

[ISO 1999] ISO/IEC 9899:1999: *Programming Languages—C*; American National Standards Institute, 1999

[IyKa 1980] Shôkichi Iyanaga, Yukiyosi Kawada (Hrg.): *Encyclopedic Dictionary of Mathematics*; MIT Press, Boston, Massachusetts, 1980

[IKKKST 1993] Tadashi Ishikawa, Toshiaki Kaneko, Kiyoshi Kato, Setsuya Kawabata, Yoshimisu Shimizu, H. Tanaka: *GRACE Manual*; KEK Report 92-19; Comp. Phys. Comm. **92**, 127-152.

[Jada 1999] Stanislaw Jadach, *FOAM: Multi-Dimensional General Purpose Monte Carlo Generator With Self-Adapting Simplical Grid*; Comp. Phys. Comm. **130**, 244-259; physics/9910004

[Jeff 2001] David J. Jeffrey: *The Multi-Valued Nature of Inverse Functions*; Preprint, erhältlich unter URL: `http://www.apmaths.uwo.ca/~djeffrey/Offprints/inverses.ps`

[JeRi 1999] David J. Jeffrey, Albert D. Rich: *Simplifying Square Roots of Square Roots by Denesting*; in: Michael J. Wester (Hrsg.): *Computer Algebra Systems – A Practical Guide*; Wiley, Chichester

[JeSu 1992] Richard D. Jenks, Robert S. Sutor: *AXIOM: The Scientific Computation System*; The Numerical Algorithms Group / Springer, New York

[Kah 1987] William Kahan: *Branch Cuts for Complex Elementary Functions; or, Much Ado About Nothing's Sign Bit*; In Iserles, A., and Powell, M. (Hrsg.), *The State of the Art in Numerical Analysis*; Clarendon Press, 165-211

[Kane 1995] Toshiaki Kaneko: *A Feynman-Graph Generator for Any Order of Coupling Constants*; Comp. Phys. Comm. **92**, 127-152.

[KaOf 1962] Anatolij A. Karatsuba, Y. Ofman: *Multiplication of Multidigit Numbers by Automatic Computers*; Doklady Akad. Nauk SSSR **145**, 293-294. Übersetzung in: Soviet Physics Doklady 7, 595-596, 1963

[KeRi 1988] Brian W. Kernighan, Dennis M. Ritchie: *The C Programming Language* (second edition); Prentice Hall, Englewood Cliffs, New Jersey

[Kili 1996] Ulrich Kilian: *Massive Zweischleifen-Integrale im Standardmodell*; Dissertation, Mainz

[KiSt 1981] Scott Kirkpatrick, Erich P. Stoll: *A Very Fast Shift-Register Sequence Random Number Generator*; J. Comput. Phys. **40**, 517-526

[KKS 1998] Richard Kreckel, Dirk Kreimer, Karl Schilcher: *First Results With a New Method for Calculating 2-Loop Box-Functions*; Eur. Phys. J. **C**, 693-699, 1998; arXiv:hep-ph/9804333

[Knu 1997] Donald E. Knuth: *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*; (third edition) Addison-Wesley, Reading, Massachusetts

[Knu 1998] Donald E. Knuth: *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*; (third edition) Addison-Wesley, Reading, Massachusetts

[Koe 1999] Wolfram Koepf: *Efficient Computation of Chebyschev Polynomials in Computer Algebra*; in: Michael J. Wester (Hrsg.): *Computer Algebra Systems – A Practical Guide*; Wiley, Chichester

[Krec 1997a] Richard Kreckel: *Irreduzible Zwei-Loop-Beiträge zu den Prozessen* $\gamma\gamma \to \pi\pi$ *und* $\eta \to \pi\gamma\gamma$; Diplomarbeit, Mainz; siehe URL: `http://wwwthep.physik.uni-mainz.de/Publications/theses/dip-kreckel.ps.gz`

[Krec 1997b] Richard Kreckel: *Parallelization of Adaptive MC Integrators*; Comp. Phys. Comm. **106**, 258-266; arXiv:physics/9710028

[Krec 2000] Richard Kreckel: *Large Scale Symbolic Programming With GiNaC*; Proc. ACAT-2000, Fermilab, Illinois

[Krec 1998] Richard Kreckel: *Parallelization of Adaptive MC Integrators—Recent pvegas Developments*; arXiv:physics/9812011

[Krei 1991] Dirk Kreimer: *The Master Two Loop Two Point Function: The General Case*; Phys. Lett. **B273**, 277-281

[Krei 1992a] Dirk Kreimer: *2-Loop Integrals in the Standard Model*; Phys. Atom. Nucl. 56 (1993) 1546-1552; arXiv:hep-ph/9212254

[Krei 1992b] Dirk Kreimer: *The Two-Loop Three-Point Functions: General Massive Cases*; Phys. Lett. **B292** 341-347

[Krei 1993] Dirk Kreimer: *Tensor Integrals for Two Loop Standard Model Calculations*; Mod. Phys. Lett. **A9**, 1105-1120; arXiv:hep-ph/9312223

[Krei 1994] Dirk Kreimer: *A Short Note on Two-Loop Box Functions*; Phys. Lett. **B347**, 1995, 107; arXiv:hep-ph/9407234

[Land 2002] Susan Landau: *Computation With Algebraic Numbers*; in: Johannes Grabmeier, Erich Kaltofen, Volker Weispfenning (Hrsg.): *Computer Algebra Handbook*; Springer, Heidelberg

[Larc 1999] Peter J. Larcombe: *On Lovelace, Babbage and the Origins of Computer Algebra*; in: Michael J. Wester (Hrsg.): *Computer Algebra Systems – A Practical Guide*; Wiley, Chichester

[Lepa 1978] G. Peter Lepage: *A New Algorithm for Adaptive Multidimensional Integration*; J. Comput. Phys. **27**, 192-203

[Lew 1981] Leonard Lewin: *Polylogarithms and Associated Functions*; North Holland, Amsterdam

[LeWe 1999] Robert H. Lewis, Michael Wester: *Comparison of Polynomial-Oriented Computer Algebra Systems*; SIGSAM Bulletin **33/4**, 5-13; erhältlich unter URL: `http://www.fordham.edu/~lewis/cacomp.html`

[Lewi 1997] Robert H. Lewis: *Fermat: A Computer Algebra System for Polynomial and Matrix Computation*; siehe URL: `http://ww.bway.net/~lewis/`

[MacK 1996] David MacKenzie: *Autoconf—Creating Automatic Configuration Scripts*, (edition 2.12); Free Software Foundation, Boston, Massachusetts, 1996

[MaT 1998] David MacKenzie, Tom Tromey: *GNU Automake*, (version 1.3); Free Software Foundation, Boston, Massachusetts

[Meye 1996] Scott Meyers: *More Effective C++*; Addison-Wesley, Reading, Massachusetts

[MOTV 1999] Gordon Matzigkeit, Alexandre Oliva, Thomas Tanner, Gary V. Vaughan: *GNU Libtool*, (version 1.3.3); Free Software Foundation, Boston, Massachusetts, 1999

[Nogu 1993] Paulo Nogueira: *Automatic Feynman Graph Generation*; J. Comput. Phys. **105**, 279-289.

[Olde 1995] Geert Jan van Oldenborgh: *An Introduction to FORM*; INLO-PUB-5/95, erhältlich unter URL: `http://www.lorentz.leidenuniv.nl/form/form/formcourse.ps.gz`

[Pink 2000] Ayal Z. Pinkus: *Yacas—Yet Another Computer Algebra System*; siehe URL: `http://www.xs4all.nl/~apinkus/yacas.html`

[PeSc 1995] Michael E. Peskin, Daniel V. Schroeder: *Introduction to Quantum Field Theory*; Addison-Wesley, Reading, Massachusetts

[Pohs 1996] Mario Daberkow, Claus Fieker, Jürgen Klüners, Michael Pohst, Katherine Roegner, Martin Schörnig, Klaus Wildanger: *Kant V4*; Journal of Symbolic Computation, **24**, 267-283

[PoTa 1996] Peter Post, Jan B. Tausk: *The Sunset Diagram in SU(3) Chiral Perturbation Theory*; Mod. Phys. Lett. **A11**, 2115-2128; arXiv:hep-ph/9604270

[PTVF 1992] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery: *Numerical Recipes in C*; Cambridge University Press, Cambridge, 1988; (zweite Auflage: 1992)

[PrWe 1999] John K. Prentice, Michael Wester: *Code Generation Using Computer Algebra Systems*; in: Michael J. Wester (Hrsg.): *Computer Algebra Systems – A Practical Guide*; Wiley, Chichester

[Pukh 1999] Alexander E. Pukhov, Edward E. Boos, Mikhail N. Dubinin, Victor P. Edneral, Viacheslav A. Ilyin, Dmitri N. Kovalenko, Alexander Kryukov, Victor I. Savrin *CompHEP – A Package for Evaluation of Feynman Diagrams and Integration Over Multi-Patricle Phase Space*; arXiv:hep-ph/9908288

[Raym 1998] Eric S. Raymond: *A Brief History of Hackerdom*; published in *Open Sources*, O'Reilly, Sebastopol, California, 1998; siehe URL: `http://www.tuxedo.org/~esr/writings/hacker-history/`

[Remi 1997] Ettore Remiddi: *Differential Equations for Feynman Graph Amplitudes*; Nuovo Cim. **A**110, 1435-1452; arXiv:hep-th/9711188

[Rich 1967] Martin Richards: *The BCPL Reference Manual*; MIT Project MAC Memorandum M-352, 1967; siehe URL: `http://www.cs.bell-labs.com/~dmr/`

[Ritc 1993] Dennis M. Ritchie: *The Development of the C Language*; Vortrag gehalten auf der zweiten Konferenz „History of Programming Languages", Cambridge, Massachusetts, 1993; siehe URL: `http://www.cs.bell-labs.com/~dmr/`

[Roth 1995] Wolfgang Roth: *Faktorisierung von Polynomen über endlichen Körpern und ganzen Zahlen*; Diplomarbeit an der Fakultät für Mathematik und Informatik, Mannheim, 1995

[Ryde 1985] Lewis H. Ryder: *Quantum Field Theory*; Cambridge University Press, Cambridge

[SaMu 1982] Teteaki Sasaki, Hirokazu Murao: *Efficient Gaussian Elimination Method for Symbolic Determinants and Linear Systems*; ACM Transactions on Mathematical Software, **8**, 1982, 277-289

[SchSt 1971] Arnold Schönhage, Volker Strassen: *Schnelle Multiplikation grosser Zahlen*; Computing **7**, 281-292

[SGI 1995] *SGI STL Allocator Design*; siehe URL: `http://www.sgi.com/tech/stl/alloc.html`

[Shou 2000] Victor Shoup: *NTL - Number Theory Library - Version 5.0*; siehe URL: `http://shoup.net/ntl/`

[ShSt 1998] Tan Kiat Shi, Willi-Hans Steeb: *Symbolic C++—An Introduction to Computer Algebra Using Object-Oriented Programming*; Springer, Singapore, 1998

[Smir 1999] Vladimir A. Smirnov: *Analytical Result for Dimensionally Regularized Massless On-Shell Double Box*; Phys. Lett. **B460**, 397-404; arXiv:hep-ph/9905323

[Smir 2000] Vladimir A. Smirnov: *Analytical Result for Dimensionally Regularized Massless Master Non-Planar Double Box With One Leg Off Shell*; Phys. Lett. **B500**, 330-337; arXiv:hep-ph/0011056

[Smir 2001] Vladimir A. Smirnov: *Analytical Result for Dimensionally Regularized Massive On-Shell Planar Double Box*; Phys. Lett. **B524**, 129-136; arXiv:hep-ph/0111160

[Smit 1998] Warren D. Smith: *A Lower Bound for the Simplexity of the N-Cube via Hyperbolic Volumes*; Preprint, siehe URL: `http://www.neci.nj.nec.com/homepage/wds/journalpubs.html`

[SmiVe 1999] Vladimir A. Smirnov, Oleg L. Veretin: *Analytical Result for Dimensionally Regularized Massless On-Shell Double Boxes With Arbitrary Indices and Numerators*; arXiv:hep-ph/9907385

[Stee 1990] Guy L. Steele: *Common Lisp the Language*; Digital Press, Woburn, Massachusetts, 1990

[Stou 1991] David R. Stoutemyer: *Crimes and Misdemeanors in the Computer Algebra Trade*; Notices AMS, **38**-7, 778-785

[Stra 1969] Volker Strassen: *Gaussian Elimination is not Optimal*; Numer. Math. **13**, 1969, 354-356

[Stro 1997] Bjarne Stroustrup: *The C++ Programming Language*; Addison-Wesley, Reading, Massachusetts, 1986; (zweite Auflage: 1991, dritte Auflage: 1997)

[Stro 1994] Bjarne Stroustrup: *The Design and Evolution of C++*; Addison Wesley, Reading, Massachusetts

[Taus 1965] Robert C. Tausworthe: *Random Numbers Generated by Linear Recurrence Modulo Two*; Math. Comput. **19**, 201-209

[Tau 1999] Jan B. Tausk: *Non-Planar Massless Two-Loop Feynman Diagrams With Four On-Shell Legs*; Phys. Lett. **B469**, 225-234; arXiv:hep-ph/9909506

[t'Ho 1971a] Gerardus 't Hooft: *Renormalization of massless Yang-Mills fields*; Nucl. Phys. **B**33, 173

[t'Ho 1971b] Gerardus 't Hooft: *Renormalizable Lagrangians for massive Yang-Mills fields*; Nucl. Phys. **B**35, 167-188

[t'HoVe 1979] Gerardus 't Hooft, Martinus J. G. Veltman: *Scalar One-Loop Integrals*; Nucl. Phys. **B**153, 365-401

[Tomm 2001] Mikko Tommila: *Apfloat: A C++ High Performance Arbitrary Precision Arithmetic Package*; siehe URL: `http://www.jjj.de/mtommila/apfloat/`

[Verm 1991] Jos A. M. Vermaseren: *Symbolic Manipulation With FORM, Version 2—Tutorial and Reference Manual*; Computer Algebra Nederland, Amsterdam, 1991

[Verm 2000] Jos A. M. Vermaseren: *New Features of FORM*; arXiv:math-ph/0010025

[VeWi 1993] Martinus J. G. Veltman, David N. Williams: *Schoonschip 91*; University of Michigan preprint UM-TH-93-18; siehe URL: `http://feynman.physics.lsa.umich.edu/~williams/preprints/schip91.pdf`

[WCS 1996] Larry Wall, Tom Christiansen, Randal Schwartz: *Programming Perl*; O'Reilly, Sebastopol

[WeGo 1991] Trudy Weibel, Gaston H. Gonnet: *An Algebra of Properties*; Gelbe Berichte **157**, Departement Informatik, ETH Zürich

[Wei 2002] Stefan Weinzierl: *Symbolic Expansion of Transcendental Functions*; arXiv:math-ph/0201011

[West 1995] Michael Wester: *A Review of CAS Mathematical Capabilities*; (mehrere aktualisierte aufeinanderfolgende Veröffentlichungen, u.a. in „Computer Algebra Nederland Nieuwsbrief“ **13** 41-48, 1994) siehe URL: `http://math.unm.edu/~wester/cas/`

[West 1999] Michael Wester: *A Critique of the Mathematical Abilities of CA Systems*; in: Michael J. Wester (Hrsg.): *Computer Algebra Systems – A Practical Guide*; Wiley, Chichester

[Wolf 1999] Stephen Wolfram: *Mathematica*; (fourth edition); Addison-Wesley, Reading, Massachusetts

Danke schön. . . ;-)

Produktionscrew Staudingerweg

Produktionsleitung	Karl Schilcher
Regie	Dirk Kreimer
Montage	Christian Bauer
Requisite	Alexander Frink
Materialassistenz	Hubert Spiesberger
Produktionsassistenz	Jürgen Körner
Beleuchtung	Dirk Kreimer, Martin Reuter, Bas Tausk
Pilze	Christian Pöselt
Unix	Lars Brücher, Johannes Plass
Tee	Johannes Franzkowski, Christian Pöselt
Sushi & Sashimi	Christian Bauer
Spam	Oliver Lauscher
Ablenkung	Astrid Bauer, Isabella Bierenbaum, Do Hoang Son, Alexander Holfter, Mathias Mertens, Mario Paschke, Isolde Savelsberg, Oliver Welzel
Orthografie	Dieter Kreckel, Christian Bauer, Hubert Spiesberger
ETAP-Kontaktbüro	Thomas Feser, Marc Hellwig

Produktionscrew Altstadt

Trost, Unterstützung	Henrike Franz
Ausdauer	Dieter & Hildegunde Kreckel
Miles and more	Henrike Franz, Martina & Tom
Undisziplinierte Debatten	„Philosophengemeinschaft 11b“
Loyalität	Gromit

Beiträge aus aller Welt

Beratung (Design-Fragen)	Bruno Haible
Beratung (LaTeX 2ε-Fragen)	Jörg Knappen
Bier	Michael Hofmann
Skeptizismus, Verschwörungstheorien	Richard J. Fateman
Vorschläge (pvegas)	Nikolas Kauer
Pizza	UFO (Unix For Others)
	Thomas Barella, Bernd Ulmann

Bei der Erstellung dieser Arbeit
wurden keine Studenten verletzt,
geschädigt oder getötet.

Lebenslauf

Persönliche Daten

Name:	Richard B. Kreckel
geboren:	am 10. Februar 1969 in Bingen
Staatsangehörigkeit:	Deutsch
Anschrift:	Eisgrubweg 11B 55116 Mainz

Ausbildung

1975-1979	Grundschule Bad Soden am Taunus
1979-1981	Stefan George Gymnasium, Bingen
1981-1985	Goethe-Schule, Buenos Aires, Argentinien
1985-1989	Stefan George Gymnasium, Bingen
1989	Abitur
1989-1990	Grundwehrdienst
WS 1990/1991	Aufnahme des Physikstudiums an der Universität Mainz
Mai 1992	Vordiplom in Physik
SS 1992 – WS 1995/1996	Hauptstudium
August 1993 – Juli 1994	Studienaufenthalt in Seattle, USA (DAAD-Austauschprogramm)
1996	Diplomand bei Prof. Dr. K. Schilcher im Institut für Physik
Mai 1997	Diplom in Physik

Forschungsaufenthalte / Tagungen

Februar-April 1998	TRACS, Edinburgh Parallel Computing Centre
23. September 1999	„TRACS Users Meeting“ (TUG99), Barcelona
12.-16. April 1999	„VI Advanced Computing Conference in Physics Research“ (AIHENP99), Heraklion
25.-28. Juni 2000	„6th IMACS Conference on Applications of Computer Algebra“ (ACA2000), St. Petersburg
16.-20. Oktober 2000	„VII Workshop on Advanced Computing and Analysis Techniques in Physics Research“ (ACAT 2000), Fermilab
27.-30. November 2001	„Workshop on Computer Particle Physics“ (CPP2001), Tokyo