Verifizierte Formelauswertung in Computer–Algebra–Systemen und objektorientierten Programmierumgebungen

Inaugural-Dissertation
zur
Erlangung des Grades eines
Doktors der Naturwissenschaften
des Fachbereichs Mathematik der
Bergischen Universität – Gesamthochschule Wuppertal

vorgelegt von
Andreas Steins
(1995)

Inhaltsverzeichnis

0 Einleitung **3**

1 Verifizierte Auswertung arithmetischer Ausdrücke **5**
1.1 Der Algorithmus von Böhm 7
1.2 Eine Verallgemeinerung des Böhm–Algorithmus nach Fischer 10
1.3 Wiederberechnungsverfahren 12
1.3.1 Ein Wiederberechnungsverfahren für dynamische Raster 13

2 Genaue Standardfunktionen in dynamischen Rastern **27**
2.1 Grundlagen 29
2.1.1 Verwendeter Fehlerkalkül und grundlegende Abschätzungen 29
2.1.2 Polynom-Auswertung nach dem Horner–Schema 30
2.2 Standardfunktionen über Potenzreihen 39
2.2.1 Die Exponentialfunktion 40
2.2.2 Der Sinus und Kosinus 43
2.2.3 Der hyperbolische Sinus 48
2.2.4 Der Arcustangens 50
2.2.5 Der Areasinus 54
2.2.6 Der Areakosinus 57
2.2.7 Der Areatangens 61
2.3 Standardfunktionen über Newton-Verfahren 64
2.3.1 Intervall–Newton–Verfahren 64
2.3.2 Die Quadratwurzel 68
2.3.3 Der natürliche Logarithmus 71
2.4 Standardfunktionen über algebraische Identitäten 73
2.4.1 Die allgemeine Potenzfunktion 73
2.4.2 Der hyperbolische Kosinus 75
2.4.3 Der Tangens 76
2.4.4 Der hyperbolische Tangens 77
2.4.5 Der Arcussinus 79
2.4.6 Der Arcuskosinus 81

3 Implementierungen **83**
3.1 Designgrundlagen 83
3.2 REDUCE 84

3.2.1 Die Benutzer–Schnittstelle 84
3.2.2 Implementationsdetails 85
3.3 C++ 94
3.3.1 Die Benutzer–Schnittstelle 94
3.3.2 Eine Klassenhierarchie für Wiederberechnungsverfahren 95
3.3.3 Eine C++–Schnittstelle für arithmetische Datentypen 101

A Beispiel–Applikationen **114**
A.1 Der X11-Funktionsplotter `xmvfplot` 114
A.2 Der X11-Taschenrechner `xmvcalc` 116

Literaturverzeichnis **118**

Kapitel 0

Einleitung

Die vorliegende Arbeit befaßt sich mit der verifizierten Auswertung arithmetischer Ausdrücke. Anstoß dazu war unter anderem die Unzufriedenheit mit den numerischen Fähigkeiten vieler Computer–Algebra–Systeme (kurz CASe). Sie liefern zwar in vielen Fällen exakte symbolische Resultate, können diese aber in der Regel nicht mit vorgebbarer Genauigkeit numerisch auswerten oder gar einschließen. Fast alle Systeme suggerieren dies zwar durch das Vorhandensein einer beliebig langen Gleitpunktarithmetik, die Anzahl signifikanter Stellen läßt sich jedoch meist nur durch fragwürdige Techniken wie sukzessivem Erhöhen der Stellenzahl und Beobachtung der „stabilen“ Ziffern erraten. Falls ausnahmsweise doch eine Möglichkeit zur Ermittlung der Güte eines Resultats vorgesehen ist, basiert sie offensichtlich auf Heuristiken (*Mathematica*), die ohne große Mühe falsifiziert werden können.

Im ersten Kapitel dieser Arbeit wird zunächst ein Überblick über erprobte Verfahren zur Auswertung bestimmter Klassen von arithmetischen Ausdrücken gegeben. Anschließend wird ein Wiederberechnungsverfahren beschrieben, das besonders auf die zuvor dargelegten Gegebenheiten in CASen zugeschnitten ist. Es ist jedoch auch in Umgebungen anwendbar, die zumindest eine adäquate Unterstützung von Langzahlarithmetiken (Operatorkonzept, Klassen) bieten.

Im zweiten, umfangreichsten Kapitel dieser Arbeit wird dann auf die genaue Berechnung von Standardfunktionen in dynamischen Rastern eingegangen. Ihre Verfügbarkeit ist neben der Existenz einer sachgemäß implementierten Arithmetik unverzichtbar für die praktische Realisierung der in Kapitel 1 vorgestellten Verfahren. Die Methoden der für die gesamte (X)SC–Sprachenfamilie grundlegenden Arbeiten von Braune ([Br]) und Krämer ([Kr]) werden dazu auf die besonderen Verhältnisse in dynamischen Rastern angepaßt und erweitert. Auch der Einsatz von Intervall–Newton–Verfahren und die Problematik der Bestimmung von zugehörigen Startintervallen werden diskutiert, da sinnvollerweise nicht alle gängigen Standardfunktionen über Polynomapproximationen berechnet werden sollten.

Das dritte Kapitel schließlich beschreibt zwei im Rahmen dieser Arbeit vorgenommene Implementierungen der in den beiden ersten Kapiteln erarbeiteten Algorithmen. Die erste wurde mit Hilfe von `REDUCE` vorgenommen und zeigt insbesondere, daß die beschriebenen Methoden vollkommen transparent in dieses CAS integrierbar sind.

Aufgrund der inzwischen — nicht zuletzt wegen einer fehlenden graphischen Benutzeroberfläche — geschwundenen Bedeutung von `REDUCE`, wurde eine zweite Implementierung in An-

griff genommen, diesmal allerdings nicht innerhalb eines CASs, sondern als C++–Bibliothek. Die Sprache C++ unterstützt ein Operatorkonzept und erlaubt dank ihres mächtigen Klassenkonzeptes die einfache Implementierung von Langzahlarithmetiken. Da sie als "general purpose language" auch für das "scientific computing" geeignet und außerdem weit verbreitet ist, schien sie als Kandidat für ein solches Projekt eine gute Wahl zu sein.
Die vorgenommene Implementierung geht jedoch weit über die alleinige Realisierung einer Langzahlarithmetik und der zugehörigen Standardfunktionen hinaus. Auch wenn dies bereits zur Kodierung eines Wiederberechnungsverfahrens — mit einer Schnittstelle, die etwa Zeichenketten zur Übergabe eines arithmetischen Ausdrucks verwendet – ausgereicht hätte, so wäre das Resultat doch nicht sehr befriedigend ausgefallen. Die Wiederberechnung sollte vielmehr in einer für den Anwender transparenten Art und Weise integriert werden. Auch sollte es möglich sein, die Arithmetik innerhalb eines Programmes selbst noch zur Laufzeit frei zu wählen. Dies wurde durch Schaffung eines meta–arithmetischen Typs mit dem Namen `ArithmeticDomainElement` (kurz `ADE`) erreicht, der das in `REDUCE` verwendete Konzept arithmetischer Domänen (Bereiche) nachahmt. Über spezielle Methoden erlaubt diese Klasse die Wahl verschiedener Arithmetiken und die Kontrolle von Parametern wie Stellenzahl oder die Vorgabe von Genauigkeitszielen. Sie ist aber dennoch für den Programmierer vollkommen analog zum vordefinierten arithmetischen C++–Typ `double` zu verwenden und kann außerdem mit überschaubarem Aufwand um eigene Arithmetiken und Standardfunktionen erweitert werden.
Im Anhang schließlich werden zwei OSF/Motif–Applikationen vorgestellt, die den Typ `ADE` verwenden und seine besonderen Möglichkeiten zu einem großen Teil ausschöpfen. Der Funktionenplotter `xmvfplot` und der Taschenrechner `xmvcalc` sind beide im besonderen Maße dazu geeignet, die Probleme des numerischen Rechnens auf dem Computer zu veranschaulichen.
Die Quelltexte aller im Rahmen dieser Arbeit erstellten Bibliotheken und Programme sind vom FTP–Server des Fachbereichs Mathematik der Bergischen Universität – Gesamthochschule Wuppertal aus dem Verzeichnis `pub/sources/Steins` abrufbar:

```
% ftp wmfiz1.math.uni-wuppertal.de
login: anonymous
password: eigene e-mail Adresse
ftp> cd pub/sources/Steins
```

Mein besonderer Dank gilt an dieser Stelle Herrn Prof. Dr. G. Heindl, der die Anregung und Möglichkeit zur Beschäftigung mit der Thematik und viele wertvolle Hinweise zur Präsentation der erhaltenen Resultate gegeben hat. Nicht unerwähnt bleiben darf auch Prof. Dr. H.-J. Buhl, mit dem ich viele hilfreiche Diskussionen über Aspekte des Softwaredesigns führen durfte und der außerdem als aktiver Beta–Tester eine Fülle von Anregungen zur Verbesserung der beiden zuvor erwähnten OSF/Motif–Applikationen geliefert hat.

Kapitel 1

Verifizierte Auswertung arithmetischer Ausdrücke

Die verifizierte Auswertung arithmetischer Ausdrücke ist ein wichtiger Teilaspekt der kontrollierten Numerik. Für bestimmte Typen solcher Ausdrücke existieren bereits etablierte Verfahren, über die im folgenden ein kurzer Überblick gegeben werden soll.
Zu den *einfachen* Typen arithmetischer Ausdrücke für die spezielle Algorithmen existieren zählen:

- Skalarprodukt–Ausdrücke ((X)SC–Sprachenfamilie).
- Univariate Polynome sowie Ausdrücke, die ausschließlich Operatoren aus $\{+, -, \cdot\}$ enthalten (Böhm–Algorithmus und verschiedene Modifikationen).

Skalarprodukt–Ausdrücke werden in der (X)SC–Sprachenfamilie ([Kl1], [Kl2]) durch Verwendung eines sogenannten überlangen Akkumulators behandelt (siehe [Ha]). Sie fallen daher in gewisser Weise aus dem hier diskutierten Kontext, da eigentlich kein spezialisierter Algorithmus verwendet wird, sondern lediglich ein modifizierter (meist softwaremäßig emulierter) Koprozessor. Dieser wird, eventuell durch Umstellen der Ausdrücke, solange wie möglich genutzt und erst — unter Berücksichtigung des gewünschten Rundungsmodus — ausgelesen, wenn dies unvermeidbar ist. Die erzielte Genauigkeit beruht einzig und allein auf der Tatsache, daß nur diese einzige finale Rundung vorzunehmen ist.
Das Verfahren von Böhm und seine Modifikationen beruhen im wesentlichen auf einer Darstellung des Problems, die die Anwendung verifizierender Löser für lineare Gleichungssysteme erlaubt. Diese verbessern iterativ eine Einschließung des Defektes einer Näherungslösung. Eine kompakte Darstellung der Verfahrensidee und der theoretischen Grundlagen findet sich z. B. in [Kie], Abschnitt 1.4.5. Dort wird auch auf die Originalarbeiten von Rump ([Ru1], [Ru2]) verwiesen. Durch Verwendung des sog. “staggered correction“ Formats und überlanger Akkumulatoren ist die mit diesen Verfahren erzielbare Genauigkeit im Prinzip nur durch das Erreichen des Unterlaufbereichs beschränkt.
Die erforderlichen Schritte zur Überführung in ein Gleichungssystem sind zwar stets möglich, liefern aber bereits für die gewöhnliche Division und Standardfunktionen nicht auflösbare Nichtlinearitäten.

Für beliebige arithmetische Ausdrücke, die insbesondere Standardfunktionen enthalten dürfen, stehen im wesentlichen die folgenden Verfahren zur Verfügung:

- Eine Verallgemeinerung des Böhm–Algorithmus nach Fischer et al ([Fi1]).
- Wiederberechnungsverfahren nach Richmann ([Ri]) u. a.

1.1 Der Algorithmus von Böhm

Die Auswertung eines univariaten Polynoms

$$P(x) = \sum_{k=0}^{N} a_k x^k$$

an der Stelle x_0 nach Horner kann nach Einführung von Hilfsvariablen $r_N, r_{N-1}, \dots, r_0$ mit

$$\begin{aligned} r_N &:= a_N \\ r_{N-1} &:= r_N \cdot x_0 + a_{N-1} \\ &\vdots \\ r_0 &:= r_1 \cdot x_0 + a_0 = P(x) \end{aligned}$$

wie folgt als lineares Gleichungssystem formuliert werden:

$$\begin{pmatrix} 1 & & & & 0 \\ -x_0 & 1 & & & \\ & \ddots & \ddots & & \\ & & -x_0 & 1 & \\ 0 & & & -x_0 & 1 \end{pmatrix} \cdot \begin{pmatrix} r_N \\ r_{N-1} \\ \vdots \\ r_1 \\ r_0 \end{pmatrix} = \begin{pmatrix} a_N \\ a_{N-1} \\ \vdots \\ a_1 \\ a_0 \end{pmatrix}$$

$$\iff L \cdot r = a \quad .$$

Offensichtlich liefert die exakte Lösung $\hat{r}$ dieses Gleichungssystems insbesondere den Wert des Polynoms P an der Stelle x_0. Es gilt: $P(x_0) = \hat{r}_0$.
Der Böhm–Algorithmus startet mit einer Näherungslösung $\tilde{r} = (\tilde{r}_N, \dots, \tilde{r}_0)$, die üblicherweise mit Hilfe des Horner–Schemas und normaler Gleitpunktarithmetik berechnet wird. Im Anschluß wird mit Hilfe eines überlangen Akkumulators eine Einschließung D des Defektes $a - L \cdot \tilde{r}$ berechnet und das System $L \cdot Y = D$ unter Verwendung von Intervall-Arithmetik durch Vorwärtssubstitution gelöst. Bis zu diesem Punkt ist das Verfahren im Prinzip eine Intervall–Version der bekannten Nachiteration. Die dort übliche Berechnung des Residuums mit doppelter Genauigkeit wird hier lediglich durch die Verwendung eines überlangen Akkumulators ersetzt, was, da nur eine abschließende intervallmäßige Rundung erfolgt, zu einer sehr engen Einschließung des Defektes führt.
$\tilde{r} + Y$ ist jetzt eine Darstellung der Einschließung der Lösung im sog. “staggered correction“ Format (siehe [Lo1]). Zur Erhöhung der erzielbaren Genauigkeit wird dabei die Summation nicht explizit durchgeführt, sondern die Korrekturterme lediglich in einem Vektor abgespeichert. Sollte die obige Einschließung den Genauigkeitsanforderungen nicht genügen, wird die Nachiteration, jetzt allerdings mit der neuen Näherung $\tilde{r} + \text{mid}(Y)$ wiederholt. Auch diese Addition wird nicht explizit durchgeführt, der neue Defekt wird vielmehr in der Form $a - L \cdot \tilde{r} - L \cdot \text{mid}(Y)$ berechnet. Dies ist wiederum ein sog. Skalarprodukt–Ausdruck, der mit Hilfe eines überlangen Akkumulators sehr eng eingeschlossen werden kann.
Diese Vorgehen wird dann bis zum Erreichen einer vorgegebenen Genauigkeit oder einer Höchstzahl von Iterationen wiederholt.

Das gesamte Verfahren läßt sich auf multivariate Polynome verallgemeinern, wie Lohner in [Lo2] gezeigt hat. Die wesentliche Idee besteht darin, als Koeffizienten bereits Werte im "staggered correction" Format zuzulassen und das Ergebnis ebenfalls in diesem Format darzustellen. Polynome in mehreren Veränderlichen können dann durch Schachtelung des so modifizierten Verfahrens ausgewertet werden.
Die zweite Abbruchbedingung des Algorithmus deutet bereits an, daß dieser Algorithmus keinesfalls garantieren kann, eine Einschließung der gewünschten Güte zu liefern. Er scheitert insbesondere, falls die Einschließungen der Defekte die Null enthalten *und* ihre Unter– und Obergrenzen den gleichen Betrag besitzen.

Eine Verallgemeinerung auf einfache arithmetische Ausdrücke

Die Grundidee des Böhm–Algorithmus läßt sich auf bestimmte einfache arithmetische Ausdrücke übertragen. Es sind dies solche, in denen nur die Grundoperationen Addition, Subtraktion und Multiplikation vorkommen.
Zur algorithmischen Formulierung seien die auftretenden Konstanten mit $a_0, \ldots, a_N$ durchnummeriert. Die Auswertung des Ausdrucks entspricht der Abarbeitung einer (i. a. nicht eindeutig bestimmten) Sequenz

$$\begin{array}{rcl} r_M & := & a_{i_M} \quad , \quad i_M \in \{0, \ldots, N\} \\ & \vdots & \\ r_k & := & \left\{ \begin{array}{rl} a_{i_k} & , \quad i_k \in \{0, \ldots, N\} \\ a_{i_k} *_k r_{j_k} & , \quad i_k \in \{0, \ldots, N\}, j_k \in \{k+1, \ldots, M\} \\ r_{i_k} *_k a_{j_k} & , \quad i_k \in \{0, \ldots, N\}, j_k \in \{k+1, \ldots, M\} \\ r_{i_k} *_k r_{j_k} & , \quad i_k, j_k \in \{k+1, \ldots, M\} \end{array} \right. \qquad , \quad *_k \in \{+, -, \cdot\} \\ & \vdots & \\ r_0 & := & \ldots \quad . \end{array}$$

Sofern keine Zeilen der Art

$$r_k := r_{i_k} \cdot r_{j_k} \quad , \quad i_k, j_k \in \{k+1, \ldots, M\} \quad , \quad k \in \{1, \ldots, M\}$$

auftreten, läßt sich dies ebenfalls auf die Lösung eines linearen Gleichungssystems zurückführen, dessen Koeffizientenmatrix untere Dreiecksgestalt besitzt. Wieder enthält die Komponente $\hat{r}_0$ des Lösungsvektors $\hat{r}$ den Wert des betrachteten Ausdrucks.

Beispiele: (vgl. [Kie], Abschnitt 1.4.6) Der auszuwertende Ausdruck sei gegeben durch: $(a_0 + a_1) \cdot a_2 - a_3 \cdot a_4$. Eine mögliche Sequenz zu seiner Auswertung ist

$$\begin{array}{rcl} r_5 & := & a_0 \\ r_4 & := & r_5 + a_1 \\ r_3 & := & r_4 \cdot a_2 \\ r_2 & := & a_3 \end{array}$$

$$\begin{aligned} r_1 &:= r_2 \cdot a_4 \\ r_0 &:= r_3 - r_1 \quad . \end{aligned}$$

Das zugehörige Gleichungssystem hat die Gestalt

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & -a_2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -a_4 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} r_5 \\ r_4 \\ r_3 \\ r_2 \\ r_1 \\ r_0 \end{pmatrix} = \begin{pmatrix} a_0 \\ a_1 \\ 0 \\ a_3 \\ 0 \\ 0 \end{pmatrix} \quad .$$

Ein einfaches Beispiel für einen Ausdruck, der sich diesem Zugang nicht unmittelbar erschließt ist: $(a_0 + a_1) \cdot (a_2 + a_3)$. Eine naheliegende Sequenz zu dessen Berechnung ist

$$\begin{aligned} r_4 &:= a_0 \\ r_3 &:= r_4 + a_1 \\ r_2 &:= a_2 \\ r_1 &:= r_2 + a_3 \\ r_0 &:= r_3 \cdot r_1 \quad . \end{aligned}$$

Sie läßt sich allerdings nicht in Matrix–Schreibweise bringen, da in der letzten Zeile eine Nichtlinearität auftritt. Diese läßt sich jedoch unter Ausnutzung der Distributivität der Multiplikation sukzessive eliminieren. Es gilt:

$$\begin{array}{rclcl} & & r_0 & = & r_3 * r_1 \\ r_1 = r_2 + a_3 & \Rightarrow & r_0 & = & r_3 \cdot (r_2 + a_3) \\ & & & = & r_3 \cdot r_2 + r_3 \cdot a_3 \\ r_2 = a_2 & \Rightarrow & r_0 & = & r_3 \cdot a_2 + r_3 \cdot a_3 \quad . \end{array}$$

Jetzt sind alle kritischen Terme beseitigt und nach Einführung von zwei zusätlichen Variablen

$$\begin{aligned} r_6 &:= r_3 \cdot a_2 \\ r_5 &:= r_3 \cdot a_3 \end{aligned}$$

ergibt sich schließlich $r_0 = r_6 + r_5$, was zu folgendem Gleichungssystem führt:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & -a_2 & 0 & 0 & 1 & 0 & 0 \\ 0 & -a_3 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & -1 & 1 \end{pmatrix} \cdot \begin{pmatrix} r_4 \\ r_3 \\ r_2 \\ r_1 \\ r_6 \\ r_5 \\ r_0 \end{pmatrix} = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad .$$

Bemerkung: Die Reihenfolge der Variablen $r_0, \ldots, r_6$ wurde so gewählt, daß die untere Dreiecksgestalt erhalten bleibt.

Eine entsprechende Vorbehandlung ist immer möglich, muß aber gegebenfalls iteriert bis zum Verschwinden aller Nichtlinearitäten durchgeführt werden. Hierbei besteht allerdings die Gefahr, daß redundante Variablen verbleiben, die die Dimension des Systems unnötig vergrößern. In unserem Beispiel sind dies r_1 und r_2.
Mit der Division läßt sich nicht auf diese Art und Weise verfahren. Es ist wohl möglich, Dividend und Divisor getrennt nach obigem Schema zu behandeln. Da die nachfolgende Division dann allerdings zwischen möglicherweise bereits fehlerbehafteten Größen stattfindet, ist kein maximal genaues Resultat mehr zu erwarten.

1.2 Eine Verallgemeinerung des Böhm–Algorithmus nach Fischer

Ein beliebiger arithmetischer Ausdruck wird bei Fischer als Abbildung $f : D_1 \times \cdots \times D_m \to \boldsymbol{R}$ mit $D_i \subseteq R$ für $i = 1, \ldots, m$ aufgefaßt. Seine sukzessive Auswertung kann durch Einführung zusätzlicher Variablen z_k für die Zwischenresultate auch durch ein i. a. nichtlineares Gleichungssystem beschrieben werden:

$$
\begin{array}{rcl}
z_1 & := & f_1(x_1, \ldots, x_m) \\
 & \vdots & \\
z_i & := & f_i(x_1, \ldots, x_m, z_1, \ldots, z_{i-1}) \\
 & \vdots & \\
z_n & := & f_n(x_1, \ldots, x_m, z_1, \ldots, z_{n-1}) \quad .
\end{array}
$$

Die $f_k (k = 1, \ldots, n)$ repräsentieren Operationen, die nur ein oder zwei Argumente betreffen. Die exakte Lösung $\hat{z}$ liefert dann in ihrer n–ten Komponente $\hat{z}_n$ den Wert des Ausdrucks $f(x_1, \ldots, x_m)$.
Es soll ein Verfahren konstruiert werden, das den Fehler $\Delta z := \hat{z} - \tilde{z}$ einer Näherungslösung $\tilde{z}$ unter Verwendung eines überlangen Akkumulators einschließt. Ähnlich wie beim Böhm–Algorithmus kann dieser Vorgang anschließend solange iteriert werden, bis eine vorgegebene Fehlerschranke erreicht wird. Dabei wird der aktuelle Wert der berechneten Näherung jeweils mit Hilfe des Mittelpunktes der berechneten Einschließung des Fehlers korrigiert. Dies erfolgt zur Erhöhung der erzielbaren Genauigkeit ebenfalls wieder durch Verwendung des „staggered correction“ Formats und nicht durch explizite Berechnung.
Es stellt sich heraus, daß die folgende implizite Form des obigen Systems zur Herleitung eines solchen Verfahrens besser geeignet ist. Sie erlaubt die Elimination von Divisionen durch Multiplikation beider Seiten mit dem entsprechenden Divisor und damit für rationale Ausdrücke eine durchgehende Verwendung eines überlangen Akkumulators bei der Berechnung

der Residuen. Sie lautet:

$$\begin{aligned} g_1(z_1) &= 0 \\ &\vdots \\ g_i(z_1,\dots,z_i) &= 0 \\ &\vdots \\ g_n(z_1,\dots,z_n) &= 0 \end{aligned}$$

mit $g_k : D_1 \times \cdots \times D_k \to \boldsymbol{R}$, $D_i \subseteq \boldsymbol{R}$, $k = 1,\dots,n$ und $i = 1,\dots,k$. Die Variablen $x_1,\dots,x_m$ werden dabei als Konstanten aufgefaßt und deshalb unterdrückt. Setzt man nun die stetig partielle Differenzbarkeit der g_k bzgl. der z_i für $i = 1,\dots,k$ voraus, ergibt sich durch iterierte Anwendung des Mittelwertsatzes der Differentialrechnung für $k = 1,\dots,n$

$$\begin{aligned} g_k(\hat{z}_1,\dots,\hat{z}_k) &= g_k(\tilde{z}_1,\hat{z}_2,\dots,\hat{z}_k) + \partial_1 g_k(\xi_1,\hat{z}_2,\dots,\hat{z}_k)\cdot\Delta z_1 \\ &= g_k(\tilde{z}_1,\tilde{z}_2,\hat{z}_3,\dots,\hat{z}_k) + \partial_1 g_k(\xi_1,\hat{z}_2,\dots,\hat{z}_k)\cdot\Delta z_1 \\ &\quad + \partial_2 g_k(\tilde{z}_1,\xi_2,\hat{z}_3,\dots,\hat{z}_k)\cdot\Delta z_2 \\ &\vdots \\ &= g_k(\tilde{z}_1,\dots,\tilde{z}_k) + \sum_{i=1}^{k}\partial_i g_k(\tilde{z}_1,\dots,\tilde{z}_{i-1},\xi_i,\hat{z}_{i+1},\dots,\hat{z}_k)\cdot\Delta z_i \end{aligned}$$

mit $\xi_i \in \operatorname{conv}\{\tilde{z}_i,\hat{z}_i\}$. Da $\hat{z}$ die exakte Lösung des impliziten Systems ist, folgt daraus

$$\begin{aligned} \Delta z_k &= -\left(g_k(\tilde{z}_1,\dots,\tilde{z}_k) + \sum_{i=1}^{k-1}\partial_i g_k(\tilde{z}_1,\dots,\tilde{z}_{i-1},\xi_i,\hat{z}_{i+1},\dots,\hat{z}_k)\cdot\Delta z_i\right) \\ &\quad /\partial_k g_k(\tilde{z}_1,\dots,\tilde{z}_{k-1},\xi_k) \quad , \end{aligned}$$

falls die Bedingung $\partial_k g_k(\tilde{z}_1,\dots,\tilde{z}_{k-1},\xi_k) \neq 0$ erfüllt ist. Da dies bei den im Kontext der Auswertung arithmetischer Ausdrücke auftretenden g_k stets der Fall ist (die g_k sind sogar linear bzgl. z_k), lassen sich hieraus die Δz_k induktiv mit

$$\Delta z_1 = -g_1(\tilde{z}_1)\,/\partial_1 g_1(\xi_1)$$

beginnend bestimmen.
Zur intervallmäßigen Auswertung ersetzt man ξ_i jeweils durch $\operatorname{conv}\{\tilde{z}_i,\tilde{z}_i+[\Delta z_i]\}$ und die für $i = 1,\dots,k-1$ nicht bekannten Werte $\hat{z}_i$ wiederum durch $\tilde{z}_i + [\Delta z_i]$. Dies führt mit der Abkürzung

$$g_k^{(i)}(\tilde{z},\tilde{z}+[\Delta z]) := \partial_i g_k(\tilde{z}_1,\dots,\tilde{z}_{i-1},\operatorname{conv}\{\tilde{z}_i,\tilde{z}_i+[\Delta z_i]\},\tilde{z}_{i+1}+[\Delta z_{i+1}],\dots,\tilde{z}_{k-1}+[\Delta z_{k-1}],\hat{z}_k)$$

auf die Inklusionsformel

$$\begin{aligned} \Delta z_k &\in [\Delta z_k] \\ &:= -\Big(g_k(\tilde{z}_1,\dots,\tilde{z}_k) + \sum_{i=1}^{k-1} g_k^{(i)}(\tilde{z},\tilde{z}+[\Delta z])\cdot[\Delta z_i]\Big)\Big/\partial_k g_k(\tilde{z}_1,\dots,\tilde{z}_{k-1},\xi_k) \end{aligned}$$

für die k–te Komponente des Fehlervektors Δz. Diese komplexe Vorschrift vereinfacht sich für die Grundrechenarten und Standardfunktionen erheblich. Spezielle Inklusionsformeln hierfür

finden sich in [Fi1] ebenso wie Hinweise zur Modifikation dieser Inklusionsformeln für eine im staggered correction Format vorliegende Näherung $\tilde{z}$.
Der auf diesen Vorüberlegungen basierende Algorithmus startet mit einer durch konventionelles Auswerten erhaltenen Näherung $\tilde{z}$ und gleicht im wesentlichen dem Böhm–Algorithmus, wobei jedoch der Fehler direkt mit Hilfe der oben hergeleiteten Inklusionsformel eingeschlossen wird.
Eine Erweiterung dieser Strategie zur Behandlung von Intervallargumenten wird in [Fi2] angegeben. Ausnahmefälle wie Überlauf oder Division durch Null werden hingegen bereits in [Fi1] kurz diskutiert.
Wegen seiner engen Verwandtschaft zum Böhm–Algorithmus hat dieses Verfahren allerdings die gleichen Schwächen. Es kann wie dieser beim Auftreten von Fehlereinschließungen, die die Null als Mittelpunkt enthalten, keine Verbesserungen einer Näherung mehr erreichen.

1.3 Wiederberechnungsverfahren

Die mit allen bisher zur Auswertung arithmetischer Ausdrücke vorgestellten Verfahren erzielbare Genauigkeit resultiert im wesentlichen aus der geschickten Verwendung eines überlangen Akkumulators und Speicherung der erhaltenen Korrekturterme im “staggered correction“ Format. Diese Tatsache bindet die Algorithmen eng an die (X)SC–Sprachenfamilie. In anderen Programmierumgebungen wäre zunächst eine Nachbildung dieser Werkzeuge durch geeignete Software erforderlich, um die bisher vorgestellten Algorithmen unverändert zu übernehmen.
Da Computer–Algebra–Systeme in der Regel Gleitpunktarithmetiken mit dynamischer Mantissenlänge zur Verfügung stellen wären Verfahren wünschenswert, die hiervon profitieren können. Die Grundidee aller Wiederberechnungsverfahren besteht darin, den gegebenen Ausdruck zunächst einer gewöhnlichen (Intervall–)Auswertung zu unterziehen, dabei aber mit Hilfe geeigneter Datenstrukturen über alle Zwischenergebnisse und die zugehörigen Operationen Buch zu führen. Sollte das Resultat nicht die erforderliche Genauigkeit aufweisen wird die Berechnung — in Teilen oder global — mit erhöhter Stellenzahl wiederholt. Die erste Auswertung erfolgt in der Regel mit einer gewissen Anzahl von Schutzziffern, um den Wiederberechnungsschritt nach Möglichkeit zu vermeiden. Die im ersten Durchlauf gespeicherten Zwischenresultate sind zur Ermittlung von Konditionszahlen notwendig, mit deren Hilfe die im zweiten Schritt einzuhaltenden Fehlertoleranzen bestimmt werden.
Auf die detaillierte Darstellung von Verfahren anderer Autoren ([Ri], [Fi3], [Schu]) wird an dieser Stelle verzichtet. Schumacher formuliert u. a. den von Richman entwickelten Algorithmus in seiner Sprechweise ([Schu], Abschnitt 3.2.1) neu, wohingegen Fischer die Einbindung schneller automatischer Differentiationstechniken behandelt ([Fi3], Kapitel 5).
Das hier vorgestellte Verfahren ist speziell auf die Integration in Computer–Algebra–Systeme sowie Programmierumgebungen, die ein Operatorkonzept zusammen mit einer geeigneten Langzahl–Arithmetik unterstützen, zugeschnitten.

1.3.1 Ein Wiederberechnungsverfahren für dynamische Raster

Als arithmetischen Ausdruck wollen wir im folgenden eine Abbildung $f : \boldsymbol{R}^m \supseteq D \to \boldsymbol{R}$ bezeichnen. Dabei wird zusätzlich gefordert, daß es ein $n \in \boldsymbol{N}$ und eine Sequenz

$$\begin{aligned} x_{m+1} &:= g_1(x_1, \ldots, x_m) \\ &\vdots \\ x_{m+i} &:= g_i(x_1, \ldots, x_m, x_{m+1}, \ldots, x_{m+i-1}) \\ &\vdots \\ y := x_{m+n} &:= g_n(x_1, \ldots, x_m, x_{m+1}, \ldots, x_{m+n-1}) \end{aligned}$$

mit $g_i : \boldsymbol{R}^{m+i-1} \supseteq D_i \to \boldsymbol{R}$ und $y = f(x_1, \ldots, x_m)$ für $(x_1, \ldots, x_m) \in D$ gibt.
Die Zahl n sowie die Funktionen g_i sind hierbei natürlich nicht notwendig eindeutig bestimmt, da es in der Regel verschiedene Möglichkeiten zur Auswertung eines gegebenen arithmetischen Ausdrucks gibt. Die Abbildungen g_i $(i = 1, \ldots, n)$ repräsentieren die auf einem Rechner verfügbaren elementaren Funktionen und Grundrechenarten, die zusätzlich eingeführten Variablen $x_{m+1}, \ldots, x_{m+n}$ die bei der Berechnung anfallenden Zwischenergebnisse bzw. den letztendlich ermittelten Wert des arithmetischen Ausdrucks. Eine formale Unterscheidung von Zwischenergebnissen und Eingangsdaten ist nicht sinnvoll, da in Computer–Algebra–Systemen auch symbolische Konstanten wie e und π in arithmetischen Ausdrücken auftreten können.

Bemerkung: Der obige Formalismus zur Beschreibung der Auswertung eines arithmetischen Ausdrucks erlaubt es, mehrfach auftretende Teilausdrücke zu berücksichtigen. Wir wollen jedoch zur Vereinfachung annehmen, daß für $i = 1, \ldots, n + m - 1$ jedes x_i in höchstens einer der Funktionen g_j $(j = i + 1, \ldots, n)$ tatsächlich vorkommt. Der Auswertung liegt also im Prinzip ein geeigneter "parse tree" zugrunde. Ohne diese Annahme könnten die entsprechenden Teilausdrücke ggf. mehrfach mit unterschiedlichen Fehlerschranken zur Wiederberechnung anstehen. In der Praxis kann diese Einschränkung bei dem hier entwickelten Intervall–Verfahren entfallen, solange sichergestellt ist, daß eine Wiederberechnung nie zu einer echten Obermenge und damit zu einer Verschlechterung eines bestehenden Resultats führt.

Nach einer intervallmäßigen Auswertung der gewählten Sequenz

$$\begin{aligned} [x_{m+1}] &:= g_1([x_1], \ldots, [x_m]) \\ &\vdots \\ [x_{m+i}] &:= g_i([x_1], \ldots, [x_m], [x_{m+1}], \ldots, [x_{m+i-1}]) \\ &\vdots \\ [y] := [x_{m+n}] &:= g_n([x_1], \ldots, [x_m], [x_{m+1}], \ldots, [x_{m+n-1}]) \quad , \end{aligned}$$

deren Durchführbarkeit natürlich gewährleistet sein muß, wird in der Regel aber nur $\{y\} \subset [y]$ und nicht $\{y\} = [y]$ gelten.

Bemerkung: Die Notation $[x]$ für ein $x \in \boldsymbol{R}$ soll andeuten, daß $[x]$ ein Intervall aus $I\boldsymbol{R}$ ist, welches x enthält.

Gesucht wird deshalb ein Algorithmus der sicherstellt, daß eine vorgegebene positive absolute oder relative Fehlerschranke Δ bzw. ε nicht überschritten wird. Nach Durchführung des Verfahrens soll also

$$\operatorname{diam}([y]) \leq \Delta \tag{1.1}$$

bzw.

$$\frac{diam([y])}{\inf(|[y]|)} \leq \varepsilon \quad \text{für } 0 \notin [y] \tag{1.2}$$

gelten. Dabei bezeichnet $\operatorname{diam}(X) := \sup(X) - \inf(X)$ den Durchmesser eines Intervalls $X \in I\boldsymbol{R}$ und $|X| := \{|x| | x \in X\}$ die *intervallwertige* Betragsfunktion.
Dies kann zum Beispiel durch eine geschickte Anpassung der verwendeten Gleitpunktraster bei einer *wiederholten Berechnung* erreicht werden, wobei entweder die komplette Sequenz in einem globalen feineren Raster oder aber einzelne Teile in verschiedenen, geeignet gewählten Rastern zu durchlaufen sind.
Der im folgenden hergeleitete Algorithmus verfolgt die zweite Strategie und ist daher insbesondere für Programmierumgebungen geeignet, die eine Langzahlarithmetik mit beliebigen Mantissenlängen zur Verfügung stellen (`REDUCE`, *Mathematica*) oder das Überladen von Operatoren und damit die einfache Implementierung einer solchen Arithmetik unterstützen (C++, `PASCAL (X)SC`).
Zur Formulierung des Algorithmus ist es erforderlich, Kriterien anzugeben, die es erlauben aus einer vorgegebenen Fehlerschranke für y sukzessive solche für g_i $(i = n, \ldots, 1)$ bzw. x_i $(i = m+n-1, \ldots, 1)$ herzuleiten. Dazu werden wir zunächst eine Analyse der Fehlerfortpflanzung für eine gewöhnliche Gleitpunktauswertung

$$\begin{aligned}
x_{m+1} + \Delta x_{m+1} = \tilde{x}_{m+1} &:= \tilde{g}_1(\tilde{x}_1, \ldots, \tilde{x}_m) \\
&\vdots \\
x_{m+i} + \Delta x_{m+i} = \tilde{x}_{m+i} &:= \tilde{g}_i(\tilde{x}_1, \ldots, \tilde{x}_m, \tilde{x}_{m+1}, \ldots, \tilde{x}_{m+i-1}) \\
&\vdots \\
\tilde{y} := x_{m+n} + \Delta x_{m+n} = \tilde{x}_{m+n} &:= \tilde{g}_n(\tilde{x}_1, \ldots, \tilde{x}_m, \tilde{x}_{m+1}, \ldots, \tilde{x}_{m+n-1})
\end{aligned}$$

vornehmen. Die dabei erhaltenen Fehlerschranken sind auf eine erneute Intervall–Auswertung zu übertragen, da sie so bestimmt werden, daß sie für alle $\tilde{x}_i \in [x_i]$ $(i = 1, \ldots, m+n)$ Gültigkeit besitzen.
Die (1.1) entsprechenden Genauigkeitsanforderungen lauten

$$|\tilde{y} - y| = |\Delta x_{m+n}| \leq \Delta \tag{1.3}$$

bzw.

$$\left|\frac{\tilde{y} - y}{y}\right| =: \left|\varepsilon_{x_{m+n}}\right| \leq \varepsilon \quad \text{für } y \neq 0 \quad . \tag{1.4}$$

Setzt man für die Funktionen g_i stetig partielle Differenzierbarkeit voraus erhält man durch iterierte Anwendung des Mittelwertsatzes der Differentialrechnung unter Verwendung der abkürzenden Schreibweise

$$x^{(i)} := (x_1, \dots, x_{m+i}) \quad \text{bzw.} \quad \tilde{x}^{(i)} := (\tilde{x}_1, \dots, \tilde{x}_{m+i})$$

aus (1.3)

$$\begin{aligned}
|\tilde{y} - y| &= \left| \tilde{g}_n(\tilde{x}^{(n-1)}) - g_n(x^{(n-1)}) \right| \\
&= \left| \Delta g_n(\tilde{x}^{(n-1)}) + g_n(\tilde{x}^{(n-1)}) - g_n(x^{(n-1)}) \right| \quad \text{mit } \Delta g_n := \tilde{g}_n - g_n \\
&= \left| \Delta g_n(\tilde{x}^{(n-1)}) + \sum_{j=1}^{m+n-1} \Delta x_j \partial_j g_n(x_1, \dots, x_{j-1}, \xi_j, \tilde{x}_{j+1}, \dots, \tilde{x}_{m+n-1}) \right| \qquad (1.5) \\
& \quad \text{mit } \xi_j \in \operatorname{conv}\{x_j, \tilde{x}_j\} \\
&\leq \Delta \quad . \qquad (1.6)
\end{aligned}$$

Um mit den obigen Abschätzungen in der Praxis arbeiten zu können, ist es erforderlich, obere Schranken für die Fehler Δx_i $(i = 1, \dots, m+n)$ und Δg_i $(i = 1, \dots, n)$ angeben zu können. Wir werden dazu naheliegenderweise die bei der intervallmäßigen Auswertung erhaltenen Einschließungen heranziehen.
Mit der entsprechenden Abkürzung

$$\left[x^{(i)}\right] := ([x_1], \dots, [x_{m+i}])$$

können, da nach Voraussetzung für $(i = 1, \dots, m+n)$ jeweils $\tilde{x}_i \in [x_i]$ gilt, alle in (1.5) auftretenden Größen wie folgt abgeschätzt werden:

$$\begin{aligned}
\left| \Delta g_n(\tilde{x}^{(n-1)}) \right| &\leq \operatorname{diam}([x_{m+n}]) \qquad (1.7) \\
|\Delta x_j| &\leq \operatorname{diam}([x_j]) \qquad (1.8) \\
|\partial_j g_n(x_1, \dots, x_{j-1}, \xi_j, \tilde{x}_{j+1}, \dots, \tilde{x}_{m+n-1})| &\leq \sup\left(\left| \partial_j g_n \left(\left[x^{(n-1)} \right] \right) \right| \right) \quad . \qquad (1.9)
\end{aligned}$$

(1.7) ist allerdings eine grobe Überschätzung des Auswertefehlers und sollte deshalb nach Möglichkeit vermieden werden.
Zur Berechnung der Einschließungen der partiellen Ableitungen kann z. B. auf die Technik der automatischen Differentiation (vgl. [Fi3]) zurückgegriffen werden. Es zeigt sich jedoch, daß diese Information für die gängigen Standardfunktionen und Operatoren meist leicht aus den bekannten Einschließungen von Ergebnis und Argument(en) abgeleitet werden kann. Da sie tatsächlich nur benötigt wird, falls eine Wiederberechnung unvermeidbar ist, erscheint dieses Vorgehen aus Optimierungserwägungen zweckmäßiger. Da wir außerdem an späterer Stelle noch eine Modifikation des verwendeten Kalküls angeben werden, die auch die Behandlung einiger nicht überall differenzierbarer Funktionen (Absolutbetrag, Quadratwurzel, Arkussinus und -kosinus und Areakosinus) erlaubt, würde das Anwendungsspektrum des Wiederberechnungsverfahrens unnötig eingeschränkt.

In der Praxis liegen ohnehin meist unäre und binäre Operatoren vor, d. h. die partiellen Ableitungen $\partial_j g_n(x_1, \ldots, x_{j-1}, \xi_j, \tilde{x}_{j+1}, \ldots, \tilde{x}_{m+n-1})$ verschwinden bis auf einen oder zwei Terme. Manchmal werden Addition und Multiplikation auch als n-äre Operatoren zugelassen (`REDUCE`). Diese Fälle müssen gesondert behandelt werden.
Bei den nachfolgenden Betrachtungen werden wir aus Gründen der einfacheren Darstellung auf die Indizierung verzichten. Sie sind sinngemäß auf die g_i $(i = 1, \ldots, n)$ zu übertragen.

Unäre Operatoren

Wenden wir uns zunächst dem Fall eines unären Operators

$$y := g(x) \quad ,$$

typischerweise einer Standardfunktion von $\boldsymbol{R}$ nach $\boldsymbol{R}$ zu. (1.5) vereinfacht sich dann zu

$$|\Delta g(\tilde{x}) + \Delta x \cdot g'(\xi)| \leq \Delta \tag{1.10}$$

für $\xi \in \mathrm{conv}\{x, \tilde{x}\}$ oder unter Verwendung des Resultats einer intervallmäßigen Auswertung

$$[y] := g([x])$$

(1.9) ausnutzend:

$$|\Delta g(\tilde{x})| + |\Delta x| \cdot \sup(|g'([x])|) \leq \Delta \quad . \tag{1.11}$$

Da $|\Delta x| \leq \mathrm{diam}([x])$ gilt, kann, falls

$$\mathrm{diam}([x]) \cdot \sup(|g'([x])|) < \Delta \tag{1.12}$$

erfüllt ist, an dieser Stelle direkt

$$|\Delta g(\tilde{x})| \leq \Delta - \mathrm{diam}([x]) \cdot \sup(|g'([x])|) \tag{1.13}$$

gefordert werden. Für den noch zu formulierenden Algorithmus bedeutet dies, daß das Argument des Operators bereits ausreichend genau vorliegt und nur die eigentliche Auswertung mit höherer Genauigkeit zu wiederholen ist.
Andernfalls müssen sowohl die Berechnung des Arguments als auch des Ergebnisses der Operation erneut vorgenommen werden. Zur Bestimmung der dabei maximal tolerierbaren Fehler aus (1.11) sind verschiedene Strategien denkbar.
Die naheliegendste besteht darin, zu fordern, daß sich die Wirkung beider Fehler die Waage hält, d. h.

$$\max\{|\Delta x| \cdot \sup(|g'([x])|), |\Delta g(\tilde{x})|\} \leq \frac{\Delta}{2} \tag{1.14}$$

anzustreben.
Man könnte ebenfalls erwägen, zu verlangen, daß die Fehler selbst die gleiche Größenordnung besitzen, was auf die Schranken

$$|\Delta g(\tilde{x})|, |\Delta x| \leq \frac{\Delta}{1 + \sup(|g'([x])|)}$$

führt. Dies scheint zunächst vorteilhaft, da nur eine Berechnung für beide Werte ausreicht. Allerdings resultiert aus dieser Vorschrift eine ständige Verschärfung der Fehlerschranken bei der nochmaligen Abarbeitung der Sequenz. Da der Nenner der rechten Seite betragsmäßig immer größer als 1 ist, gilt insbesondere stets

$$|\Delta x| < \Delta \quad .$$

Wegen dieses negativen Trends bei der Propagation der Fehlerschranken haben wir im folgenden auf (1.14) zurückgegriffen.
In der Praxis werden bei der Berechnung von arithmetischen Ausdrücken relative Fehler bevorzugt, weil sie Informationen über die Anzahl der signifikanten Stellen eines ermittelten Ergebnisses liefern können. Da mit ihnen aber nur durchgängig gearbeitet werden kann, wenn keine der berechneten Einschließungen die Null enthält, wurde zunächst ein Kriterium für absolute Fehler hergeleitet. Der Übergang zu relativen Fehlern geschieht in kanonischer Weise.
Falls $0 \notin [y]$ gilt und eine Schranke ε für den Betrag des relativen Fehlers von y vorgegeben ist, kann das zuvor beschriebene Vorgehen für $\Delta = \varepsilon \cdot \inf(|[y]|)$ angewendet werden.
Falls $0 \notin [x]$ bzw. $0 \notin [y]$ gilt, können die hergeleiteten Schranken für $|\Delta x|$ und $|\Delta g(\tilde{x})|$ über die Beziehungen

$$|\Delta x| \leq |\varepsilon_x| \cdot \sup(|[x]|) \quad \text{bzw.} \qquad |\Delta g(\tilde{x})| \leq |\varepsilon_{g(\tilde{x})}| \cdot \sup(|[y]|)$$

in Obergrenzen für die tolerierbaren relativen Fehler von x bzw. $g(\tilde{x})$ umgerechnet werden.

Binäre Operatoren

Zu den in der Praxis vorkommenden binären Operatoren gehören Addition, Subtraktion, Multiplikation, Division und Potenzieren. Wie bereits an früherer Stelle bemerkt, wollen wir Addition und Multiplikation in voller Allgemeinheit diskutieren, d. h. sie als n–äre Operatoren zulassen. Die allgemeine Potenzfunktion wird in der Regel mit Hilfe der Identität $x_1^{x_2} = e^{\log(x_1) \cdot x_2}$ berechnet. Statt der Sequenz

$$y := g_1(x_1, x_2) := x_1^{x_2}$$

wird also zur Berechnung von $x_1^{x_2}$ tatsächlich die Befehlsfolge

$$\begin{aligned} x_3 &:= g_1(x_1, x_2) := \log(x_1) \\ x_4 &:= g_2(x_1, x_2, x_3) := x_3 \cdot x_2 \\ y := x_5 &:= g_3(x_1, x_2, x_3, x_4) := \exp(x4) \end{aligned}$$

abgearbeitet. Dementsprechend finden die zuvor hergeleiteten Fehlerschranken für unäre Operatoren sowie die noch zu betrachtenden für die Multiplikation Anwendung. Falls die Potenzfunktion jedoch für kleine ganzzahlige Exponenten effizienter durch sukzessive Multiplikation implementiert werden soll, zeigt Lemma (2.1.2) einen möglichen Ausweg auf.
Wenden wir uns also zunächst der Subtraktion zu. Für

$$y := g(x_1, x_2) := x_1 - x_2$$

vereinfacht sich (1.5) zu

$$|\varepsilon \cdot (\tilde{x}_1 - \tilde{x}_2) + \Delta x_1 - \Delta x_2| \leq \Delta \quad ,$$

wobei $|\varepsilon| \leq \varepsilon_-$ gilt und ε_- vom verwendeten Raster sowie dem gewählten Rundungsmodus abhängt.

Bemerkung: Die Gültigkeit einer solchen Fehlerschranke ε_- für die verwendete Arithmetik kann vorausgesetzt werden, da in dynamischen Rastern kein Unter- oder Überlauf auftritt.

Mit Hilfe des Resultats

$$[y] := g([x_1], [x_2]) := [x_1] - [x_2]$$

einer intervallmäßigen Auswertung kann bei vorgegebener absoluter Fehlerschranke Δ analog zu (1.12), falls

$$\text{diam}([x_1]) + \text{diam}([x_2]) < \Delta \tag{1.15}$$

erfüllt ist, die Wiederberechnung der Argumente unterbleiben und mit Hilfe der hinreichenden Bedingung:

$$\varepsilon_- \cdot \sup(|[y]|) \leq \Delta - \text{diam}([x_1]) - \text{diam}([x_2])$$

das für die erneute Berechnung der Differenz zu verwendende Raster geeignet gewählt werden. Ist dies nicht möglich, kann etwa

$$\varepsilon_- \cdot \sup(|[y]|) \leq \frac{\Delta}{3} \qquad \text{und} \qquad \max\{|\Delta x_1|, |\Delta x_2|\} \leq \frac{1}{3} \cdot \Delta \tag{1.16}$$

gefordert werden.
Unter der Voraussetzung $0 \notin [y]$ können die vorausgegangenen Betrachtungen sinngemäß auf relative Fehlerschranken übertragen werden. Diese müssen lediglich — wie bei den unären Operatoren vorgeführt — durch Multiplikation mit $\inf(|[y]|)$ in eine absolute Schranke umgerechnet werden.
Falls $[x_i]$, $i \in \{1, 2\}$ die Null nicht enthält, kann gemäß

$$|\Delta x_i| \leq |\varepsilon_{x_i}| \cdot \sup(|[x_i]|)$$

sofort eine relative Fehlerschranke für den entsprechenden Operanden hergeleitet werden.

Für die Division

$$y := g(x_1, x_2) := x_1/x_2 \qquad \text{mit } x_2 \neq 0$$

ergibt sich für $\tilde{x}_2 \neq 0$ aus (1.5)

$$\left|\varepsilon \cdot \frac{\tilde{x}_1}{\tilde{x}_2} + \Delta x_1 \cdot \frac{1}{\tilde{x}_2} - \Delta x_2 \frac{x_1}{\xi^2}\right| \leq \Delta$$

für ein $\xi \in \text{conv}\{x_2, \tilde{x}_2\}$, wobei auch hier — wie bei der Subraktion — die Existenz einer Fehlerschranke $\varepsilon_/$ mit $|\varepsilon| \leq \varepsilon_/$ vorausgesetzt wird. Ein Vergleich mit dem Resultat der

klassischen Fehleranalyse

$$\begin{aligned}\Delta y &= (1+\varepsilon)\cdot\frac{x_1+\Delta_{x_1}}{x_2\cdot(1+\varepsilon_{x_2})}-\frac{x_1}{x_2}\\ &= \varepsilon\cdot\frac{\tilde{x}_1}{\tilde{x}_2}+\frac{x_1+\Delta_{x_1}}{x_2\cdot(1+\varepsilon_{x_2})}-\frac{x_1}{x_2}\\ &= \varepsilon\cdot\frac{\tilde{x}_1}{\tilde{x}_2}+\frac{x_1}{x_2\cdot(1+\varepsilon_{x_2})}+\Delta x_1\cdot\frac{1}{\tilde{x}_2}-\frac{x_1}{x_2}\\ &= \varepsilon\cdot\frac{\tilde{x}_1}{\tilde{x}_2}+\frac{x_1}{x_2}\cdot\left(1-\frac{\varepsilon_{x_2}}{1+\varepsilon_{x_2}}\right)+\Delta x_1\cdot\frac{1}{\tilde{x}_2}-\frac{x_1}{x_2}\end{aligned}$$

erlaubt sogar die Elimination von ξ. Es ergibt sich:

$$\left|\varepsilon\cdot\frac{\tilde{x}_1}{\tilde{x}_2}+\Delta x_1\cdot\frac{1}{\tilde{x}_2}-\Delta x_2\cdot\frac{x_1}{x_2\tilde{x}_2}\right|\le\Delta\quad .$$

Wie bei der Subtraktion kann mit Hilfe des Resultats

$$[y]:=g\left([x_1],[x_2]\right):=[x_1]\,/\,[x_2]\qquad\text{mit } 0\notin[x_2]$$

einer intervallmäßigen Auswertung analog zu (1.12), falls

$$\frac{\operatorname{diam}([x_1])+\operatorname{diam}([x_2])\cdot\sup(|[y]|)}{\inf(|[x_2]|)}<\Delta$$

erfüllt ist, die Wiederberechnung der Argumente unterbleiben und mit Hilfe der hinreichenden Bedingung

$$\varepsilon_/\cdot\sup(|[y]|)\le\Delta-\frac{\operatorname{diam}([x_1])+\operatorname{diam}([x_2])\cdot\sup(|[y]|)}{\inf(|[x_2]|)}$$

das für die erneute Berechnung des Quotienten zu verwendende Raster geeignet gewählt werden. Wie bei der Subtraktion kann andernfalls alternativ

$$\varepsilon_/\cdot\sup(|[x_3]|)\le\frac{\Delta}{3}\quad\text{und}\quad\max\{|\Delta x_1|,|\Delta x_2|\cdot\sup(|[y]|)\}\le\frac{1}{3}\cdot\Delta\cdot\inf(|[x_2]|)\qquad(1.17)$$

gefordert werden.
Relative Fehlerschranken sind wie der Subtraktion zu behandeln.

N–äre Operatoren

Für die Addition

$$y:=g(x_1,\ldots,x_N):=\sum_{i=1}^{N}x_i$$

ergibt (1.5)

$$\left|\varepsilon\cdot\sum_{i=1}^{N}\tilde{x}_i+\sum_{i=1}^{N}\Delta x_i\right|\le\Delta\quad\text{mit}\quad|\varepsilon|\le\varepsilon_\Sigma.$$

Diese Darstellung setzt allerdings voraus, daß die Addition der $\tilde{x}_i$ $(i = 1, \ldots, N)$ selbst exakt durchgeführt wird und nur *eine* Rundung ins Ausgaberaster erfolgt. Mit Hilfe des Resultats einer intervallmäßigen Auswertung

$$[y] := g([x_1], \ldots, [x_N]) := \sum_{i=1}^{N} [x_i]$$

kann, falls

$$\sum_{i=1}^{N} \operatorname{diam}([x_i]) < \Delta$$

erfüllt ist, die Wiederberechnung *aller* Summanden vermieden werden. Dazu muß lediglich das Ausgaberaster mit Hilfe der hinreichenden Bedingung

$$\varepsilon_\Sigma \cdot \sup(|[y]|) \leq \Delta - \sum_{i=1}^{N} \operatorname{diam}([x_i])$$

geeignet gewählt werden. Andernfalls kann etwa

$$\varepsilon_\Sigma \cdot \sup(|[y]|) \leq \frac{\Delta}{N+1} \quad \text{und} \quad \max_{i=1,\ldots,N} \{|\Delta x_i|\} \leq \frac{\Delta}{N+1} \tag{1.18}$$

gefordert werden. Sollte die exakte Durchführung der Summation nicht möglich sein, kann z. B. Lemma (2.1.3) herangezogen werden, um ein geeignetes Raster für die Durchführung der Additionen zu wählen.
Der zweite Teil von (1.18) ist zwar relativ einfach zu handhaben, liefert jedoch viel zu scharfe Schranken, falls viele Summanden exakt oder zumindest „sehr gut eingeschlossen" vorliegen und nur wenige stark gestört sind. Das folgende Vorgehen erlaubt es, diesen Umstand auszunutzen und weniger strenge Grenzen zu erhalten.
Dazu sei zunächst o. B. d. A. angenommen, daß $\operatorname{diam}([x_i]) \leq \operatorname{diam}([x_{i+1}])$ für $i = 1, \ldots, N-1$ gilt, die Argumente also in aufsteigender Reihenfolge bezüglich ihrer Durchmesser angeordnet sind. Nun wird das maximale $N_0 \in \{1, \ldots, N\}$ mit der Eigenschaft

$$\sum_{i=1}^{N_0} \operatorname{diam}([x_i]) \leq \frac{N_0}{N+1} \cdot \Delta \tag{1.19}$$

bestimmt und anschließend alle Summanden x_j mit $j > N_0$ erneut berechnet, wobei jetzt allerdings etwas schwächer

$$|\Delta x_j| \leq \frac{1}{N - N_0} \left(\frac{N}{N+1} \cdot \Delta - \sum_{i=1}^{N_0} \operatorname{diam}([x_i]) \right)$$

gefordert wird. Existiert ein solches N_0 nicht, bleibt der zweite Teil von (1.18) gültig. Relative Fehlerschranken sind wie der Subtraktion zu behandeln.

Für die Multiplikation

$$y := g(x_1, \ldots, x_N) := \prod_{i=1}^{N} x_i$$

ergibt (1.5)

$$\left|\varepsilon \cdot \prod_{i=1}^{N} \tilde{x}_i + \sum_{i=1}^{N} \Delta x_i \cdot x_1 \cdot \ldots \cdot x_{i-1} \cdot \tilde{x}_{i+1} \cdot \ldots \cdot \tilde{x}_N\right| \le \Delta \quad \text{mit} \quad |\varepsilon| \le \varepsilon_\Pi \quad .$$

Auch diese Darstellung ist wie bei der Addition nur zulässig, wenn die Multiplikation der $\tilde{x}_i$ $(i = 1, \ldots, N)$ exakt durchgeführt wird und nur *eine* Rundung ins Ausgaberaster erfolgt. Mit Hilfe des Resultats einer intervallmäßigen Auswertung

$$[y] := g([x_1], \ldots, [x_N]) := \prod_{i=1}^{N} [x_i]$$

kann, falls

$$\sum_{i=1}^{N} \operatorname{diam}([x_i]) \cdot \prod_{\substack{j=1 \\ j \ne i}}^{N} \sup(|[x_j]|) < \Delta \tag{1.20}$$

erfüllt ist, die Wiederberechnung *aller* Faktoren vermieden werden. (1.20) ist im Fall $N > 2$ für praktische Belange wenig geeignet, da die partiellen Produkte in der Regel nicht verfügbar sind und aufwendig berechnet werden müßten. Die leicht abgeschwächte Bedingung

$$\sup(|[y]|) \cdot \sum_{i=1}^{N} \frac{\operatorname{diam}([x_i])}{\sup(|[x_i]|)} < \Delta \tag{1.21}$$

hingegen ist leichter zu überprüfen, auch wenn auf den ersten Blick die Division durch die $\sup(|[x_i]|)$ nicht immer möglich scheint. Sollte jedoch ein $i_0 \in \{1, \ldots, N\}$ mit $\sup(|[x_{i_0}]|) = 0$ existieren, kann vernünftigerweise $[y] = 0$ angenommen werden, was eine Wiederberechnung ohnehin ad absurdum führt.
Das Ausgaberaster kann nun mit Hilfe der hinreichenden Bedingung

$$\varepsilon_\Pi \cdot \sup(|[y]|) \le \Delta - \sup(|[y]|) \cdot \sum_{i=1}^{N} \frac{\operatorname{diam}([x_i])}{\sup(|[x_i]|)}$$

geeignet gewählt werden. Andernfalls kann etwa

$$\varepsilon_\Pi \cdot \sup(|[y]|) \le \frac{\Delta}{N+1} \quad \text{und} \quad \max_{i=1,\ldots,N} \left\{ \frac{\Delta x_i}{\sup(|[x_i]|)} \right\} \le \frac{1}{N+1} \cdot \frac{\Delta}{\sup(|[y]|)} \tag{1.22}$$

gefordert werden. Auch in diesem Fall kann, falls die Berechnung des Produkts nicht exakt durchführbar ist, ε_Π jetzt allerdings mit Hilfe von Lemma (2.1.1) ermittelt werden.
Wie bei der Addition „verschenkt" der zweite Teil von (1.22) eventuell bereits vorhandene Genauigkeit der Faktoren $x_1, \ldots, x_N$. Um dem entgegenzuwirken, sei zunächst o. B. d. A. angenommen, daß $\frac{\operatorname{diam}([x_i])}{\sup(|[x_i]|)} \le \frac{\operatorname{diam}([x_{i+1}])}{\sup(|[x_{i+1}]|)}$ für $i = 1, \ldots, N-1$ gilt, die Argumente also in aufsteigender Reihenfolge bezüglich gewisser relativer Fehlergrößen angeordnet sind. Nun wird wie bei der Addition das maximale $N_0 \in \{1, \ldots, N\}$ mit der Eigenschaft

$$\sum_{i=1}^{N_0} \frac{\operatorname{diam}([x_i])}{\sup(|[x_i]|)} \le \frac{N_0}{N+1} \cdot \frac{\Delta}{\sup(|[y]|)} \tag{1.23}$$

bestimmt und anschließend alle Faktoren x_j mit $j > N_0$ erneut berechnet, wobei jetzt allerdings etwas schwächer

$$|\Delta x_j| \leq \frac{1}{N - N_0} \left(\frac{N}{N+1} \cdot \frac{\Delta}{\sup(|[y]|)} - \sum_{i=1}^{N_0} \frac{\operatorname{diam}([x_i])}{\sup(|[x_i]|)} \right)$$

gefordert wird. Existiert ein solches N_0 nicht, bleibt der zweite Teil von (1.22) gültig. Relative Fehlerschranken sind wie der Subtraktion zu behandeln.

Modifikation des Kalküls für nicht überall stetig partiell differenzierbare Funktionen

Der bisher verwendete, aus (1.5) abgeleitete Kalkül zur Berechnung von Fehlerschranken, ist leider nicht für alle gängigen Standardfunktionen anwendbar, selbst wenn die intervallmäßige Auswertung des zugrunde liegenden Ausdrucks durchführbar ist. Es treten zwei Klassen von Ausnahmen auf.
Zu der einen gehört der Absolutbetrag, der an der Stelle 0 nicht differenzierbar ist. Er genügt dort aber zumindest noch einer lokalen Lipschitz–Bedingung. Die Verallgemeinerung des Kalküls ist in diesem Fall offensichtlich, es ist lediglich die obere Schranke für den Betrag der partiellen Ableitung durch die Lipschitz–Konstante zu ersetzen.
Anders stellt sich die Situation bei einigen Funktionen dar, die an den Rändern ihrer Definitionsbereiche unbeschränkte Ableitungen besitzen. Hierunter fallen von den im nachfolgenden Kapitel behandelten die Umkehrfunktionen von Sinus, Kosinus und hyperbolischem Kosinus sowie die Quadratwurzel. Eine (1.5) vergleichbare Darstellung kann in allen diesen Fällen jedoch ebenfalls erzielt werden. Ausgangpunkt ist dazu die folgende, aus (1.3) abgeleitete Darstellung:

$$\begin{aligned} |\tilde{y} - y| &= |\tilde{g}(\tilde{x}) - g(x)| \\ &= |\Delta g(\tilde{x}) + g(\tilde{x}) - g(x)| \\ &\leq |\Delta g(\tilde{x})| + |g(\tilde{x}) - g(x)| \\ &\leq \Delta \quad . \end{aligned}$$

Der Term $|g(\tilde{x}) - g(x)|$ muß jetzt in jedem einzelnen Fall gesondert abgeschätzt werden.
Wir wollen dies hier exemplarisch für den Arcussinus durchführen. Eine obere Schranke für $|\partial_x \arcsin(\xi)|, \xi \in \operatorname{conv}\{x, \tilde{x}\}$ wie zur Anwendung von (1.5) erforderlich, kann nicht angegeben werden, wenn $\pm 1 \in \{x, \tilde{x}\}$ gilt. Im Fall $\tilde{x} = -1$ ist — bei hälftiger Aufteilung des Beitrages zum Gesamtfehler Δ — folgende Ungleichung nach $\Delta x = \tilde{x} - x$ aufzulösen:

$$|\arcsin(\tilde{x}) - \arcsin(x)| = \frac{\pi}{2} + \arcsin(-1 + \Delta x) \leq \frac{\Delta}{2} \quad .$$

Anwendung des Kosinus ergibt

$$\cos(\frac{\pi}{2} + \arcsin(-1 + \Delta x)) \geq \cos(\frac{\Delta}{2}) \quad ,$$

da der Kosinus im in Frage kommenden Intervall $[0, \pi]$ streng monoton fallend ist. Nach Anwendung des Additionstheorems für den Kosinus erhält man

$$\cos(\frac{\pi}{2})\cos(\arcsin(-1+\Delta x)) - \sin(\frac{\pi}{2})\sin(\arcsin(-1+\Delta x)) \geq \cos(\frac{\Delta}{2})$$

und daraus nach einfacher Umformung

$$\Delta x \leq 1 - \cos(\frac{\Delta}{2}) = 2\sin^2(\frac{\Delta}{4}) \quad .$$

Das gleiche Resultat ergibt sich für die verbleibenden Fälle $x = -1$ und $1 \in \{x, \tilde{x}\}$. Die Argumentation für den Arcuskosinus verläuft vollkommen analog und führt zum gleichen Ergebnis.
Im Fall des Areakosinus und $1 \in \{x, \tilde{x}\}$ ergibt sich

$$\Delta x \leq \cosh(\frac{\Delta}{2}) - 1 = 2\sinh^2(\frac{\Delta}{4})$$

und für die Quadratwurzel und $0 \in \{x, \tilde{x}\}$

$$\Delta x \leq \frac{\Delta^2}{4} \quad .$$

Bemerkung: Ohne den Übergang zum halben Winkel wäre die numerische Bestimmung konkreter Werte für obere Schranken in den beiden ersten Fällen sehr aufwendig, da die Gefahr der Auslöschung besteht.

Ein Algorithmus zur hochgenauen Auswertung arithmetischer Ausdrücke

Vor der endgültigen Formulierung soll hier zunächst eine genaue Spezifikation des Algorithmus angegeben werden. Dabei wird von der Verfügbarkeit einer Arithmetik zur Basis b mit dynamischen Mantissenlängen und unbeschränktem Exponentenbereich ausgegangen. Zu verwendeten Begriffen und Notationen sei an dieser Stelle auf die einleitenden Ausführungen in Kapitel 2 verwiesen. Die Grundrechenarten, Standardfunktionen und eventuell zugelassene transzendente Konstanten (e, π, ...) müssen mit vorgebbarer Genauigkeit berechenbar sein. Ist dies alles der Fall, soll das Verfahren folgendes leisten.
Falls die intervallmäßige Auswertung des gegebenen Ausdrucks mit k–stelliger Arithmetik möglich ist, gilt nach einem eventuell erforderlichen Wiederberechnungsschritt:

1. $[x_{m+n}] \in IS(b, k, -\infty, \infty)$.
2. $\text{diam}([x_{m+n}]) \leq b^{-k}$, falls $0 \in [x_{m+n}]$ gilt.
3. $[x_{m+n}]$ enthält maximal 3 Elemente des Rasters $S(b, k, -\infty, \infty)$, falls $0 \notin [x_{m+n}]$ gilt.

Die folgende, unter Verwendung einer noch zu definierenden Funktion `recompute`$([x_i], k)$ formulierte Vorschrift leistet das Gewünschte:

1. **Initialisierung:**
   Berechne $[x_1], \ldots, [x_{m+n}]$ unter Verwendung k–stelliger Intervall–Arithmetik.

2. **Wiederberechnung1:**
   Falls $0 \in [x_{m+n}]$ und $\mathrm{diam}([x_{m+n}]) > b^{-k}$ gilt, rufe `recompute`$([x_{m+n}], k)$ auf.

3. **Wiederberechnung2:**
   Falls $0 \notin [x_{m+n}]$ und $\#([x_{m+n}] \cap S(b, k, -\infty, \infty)) > 3$ gilt, rufe `recompute`$([x_{m+n}], k)$ auf.

4. **Resultat:**
   $[x_{m+n}]$ erfüllt die obigen Spezifikationen.

Bemerkungen:

– Die erstmalige Berechnung erfolgt häufig unter Verwendung zusätzlicher Schutzziffern.

– Der Aufruf von `recompute`$([x_{m+n}], k)$ erfolgt nach obiger Vorschrift eventuell zweimal. Dieser Fall tritt auf, falls nach der ersten Wiederberechnung die Null aus $[x_{m+n}]$ eliminiert wurde. Ohne eine erneute Wiederberechnung würde $[x_{m+n}]$ die Bedingungen

$$0 \notin [x_{m+n}] \qquad \text{und} \qquad \mathrm{diam}([x_{m+n}]) \leq b^{-k}$$

erfüllen, ohne daß notwendigerweise auch

$$\#([x_{m+n}] \cap S(b, k, -\infty, \infty)) \leq 3$$

gilt.

Die Funktion `recompute`$([x_i], \ell)$ erfüllt eine gegenüber der des Wiederberechnungsverfahrens leicht abgewandelte Spezifikation. Nach ihrer Abarbeitung gilt:

1. $[x_i] \in IS(b, \ell, -\infty, \infty)$.

2. $\mathrm{diam}([x_i]) \leq b^{-\ell}$, falls beim Aufruf(!) $0 \in [x_i]$ galt.

3. $[x_i]$ enthält maximal 3 Elemente des Rasters $S(b, \ell, -\infty, \infty)$, falls beim Aufruf(!) $0 \notin [x_i]$ galt.

Bemerkung: Der dritte Punkt der Spezifikation impliziert, daß die obere Schranke der Beträge der relativen Fehler $|\varepsilon_{x_i}|$ für beliebige $x_i \in [x_i]$ im Bereich $[2 \cdot b^{-\ell}, 2 \cdot b^{1-\ell}]$ liegt, also um eine Größenordnung bezüglich der Basis variieren kann. Diese „Lücke“ muß bei der Kalkulation der Fehlerschranken berücksichtigt werden! In den im Rahmen dieser Arbeit vorgenommenen Implementierungen wurde aus Gründen der Konsistenz mit dem zweiten Punkt der Spezifikation stets etwas schärfer $|\varepsilon_{x_i}| \leq b^{-\ell}$ gefordert.

Die Funktion `recompute`$([x_i], \ell)$ verwendet Rekursion und arbeitet nach folgendem Schema:

1. **Rekursionsende:**
   Falls $i \in \{1, \ldots, m\}$ gilt, x_i also zu den Eingangsdaten gehört, gebe eine der Spezifikation von
   `recompute`$(.,.)$ entsprechende ℓ–stellige Einschließung zurück.

2. **Vermeidung:**
   Falls das Argument bzw. die Operanden genau genug vorliegen, gehe zu (4).

3. **Rekursion:**
   Bestimme die erforderlichen Genauigkeiten für Argument bzw. Operanden und rufe `recompute`(.,.) zu ihrer Wiederberechnung auf.

4. **Resultat:**
   Wiederhole die eigentliche Auswertung der Funktion bzw. des Operators mit der ermittelten Genauigkeit und gebe das Resultat auf ℓ Stellen intervallmäßig gerundet zurück.

Bemerkung: Die Bestimmung der erforderlichen Genauigkeiten erfolgt nach den zuvor hergeleiteten Formeln, die hier der Kürze wegen nicht erneut angeführt werden. Die dabei erhaltenen Ergebnisse sind lediglich auf entsprechende negative Potenzen der Basis b abzurunden, d. h. im Falle eines resultierenden absoluten Fehlers Δ ist $\min\{k \in \boldsymbol{N} \mid b^{-k} \leq \Delta\}$ und für einen relativen Fehler ε entsprechend $\min\{k \in \boldsymbol{N} \mid 2 \cdot b^{1-k} \leq \varepsilon\}$ als zweites Argument für den rekursiven Aufruf von `recompute`(.,.) zu verwenden.

Beispiel: Zitat aus [Wo], Seite 52: „Es stellt sich heraus, daß die Zahl $e^{\pi\sqrt{163}}$ sehr nahe bei einer ganzen Zahl liegt. Um zu überprüfen, ob das Ergebnis tatsächlich keine ganze Zahl ist, müssen Sie die Rechnung mit ausreichender numerischer Präzision ausführen."

```
IN[4] := N[Exp[Pi Sqrt[163]], 50]                                    17
OUT[4] := 2.6253 74126 40768 74399 99999 99999 25007 25971 98185 689 10
```

Die Entscheidung, ob dies ein Beweis für die Behauptung $e^{\pi\sqrt{163}} \notin \boldsymbol{Z}$ ist, sei dem geneigten Leser überlassen. Wir werden nun das vorgestellte Wiederberechnungsverfahren anwenden, um diese Frage durch Berechnung einer garantierten Einschließung zweifelsfrei zu beantworten.
Die Sequenz

$$\begin{aligned}
x_1 &:= 163 \\
x_2 &:= \pi \\
x_3 &:= g_1(x_1, x_2) := \sqrt{x_1} \\
x_4 &:= g_2(x_1, x_2, x_3) := x_2 \cdot x_3 \\
x_5 &:= g_3(x_1, x_2, x_3, x_4) := \exp(x_4)
\end{aligned}$$

wird zur Auswertung des Ausdrucks herangezogen. Ihre Abarbeitung mit 31–stelliger BCD–Intervallarithmetik und der entsprechenden Einschließung $[3.1415926535897932384626433832^{80}_{79}]$ von π liefert folgende Resultate:

$$\begin{aligned}
[x_3] &= \left[12.767145334803704661710952009 7^{9}_{8}\right] \\
[x_4] &= \left[40.109169991132519755350083622^{94}_{89}\right] \\
[x_5] &= [262537412640768743.9999999999955, 262537412640768744.0000000000087] \quad .
\end{aligned}$$

Dieses Resultat erfüllt nicht die Spezifikation des Verfahrens für eine 31–stellige hochgenaue Auswertung und ermöglicht insbesondere auch keine Antwort auf unsere Fragestellung. Es muß also eine Wiederberechnung erfolgen und dazu sind zunächst die erforderlichen Genauigkeiten Δ_{g_3} und Δ_{x_4} zu bestimmen. Der Test gemäß (1.12) versagt, es ist also (1.14) anzuwenden. Nach dem Übergang zu relativen Fehlerschranken ε_{g_3} bzw. ε_{x_4} erhält man

$$\begin{array}{rcll} |\varepsilon_{g_3}| & \leq & 0.4999999999999999999999999999748 \cdot 10^{-31} & \text{und} \\ |\varepsilon_{x_4}| & \leq & 0.1246597723439656833727217914825 \cdot 10^{-32} & . \end{array}$$

Da

$$\min \left\{ k \in \boldsymbol{N} \ |2 \cdot 10^{1-k} \leq 0.4999999999999999999999999999748 \cdot 10^{-31} \right\} = 33$$

bzw.

$$\min \left\{ k \in \boldsymbol{N} \ |2 \cdot 10^{1-k} \leq 0.1246597723439656833727217914825 \cdot 10^{-32} \right\} = 35$$

gilt, ist also zunächst $\texttt{recompute}([x_4], 35)$ aufzurufen und anschließend g_3 auf 33 Stellen genau auszuwerten.
In Abweichung von (1.22) gehen wir bei der Wiederberechnung von x_4 davon aus, daß die Multiplikation exakt ausgeführt wird. Mit klassischer Fehlerrechnung verifiziert man leicht, daß dies zu 2 zusätzlichen signifikanten Ziffern für x_2 bzw. x_3 führt, also $\texttt{recompute}([x_2], 37)$ bzw. $\texttt{recompute}([x_3], 37)$ auszuführen sind.
Der erste Aufruf führt zur genauen Berechnung

$$[x_2] = \left[3.14159265358979323846264338327950288_4^5 \right]$$

von π auf 37 Stellen.
Da x_1 exakt vorliegt und damit (1.12) erfüllt ist, kann Δ_{g_1} nach (1.13) bestimmt werden, was direkt auf $\Delta_{g_1} \leq 10^{-37}$ führt. Der Übergang auf eine relative Fehlerschranke

$$|\varepsilon_{g_1}| \leq 10^{-37} \cdot \frac{\inf(|[x_3]|)}{\sup(|[x_3]|)}$$

zeigt, daß 2 zusätzliche Ziffern erforderlich sind.
Die Wiederberechnung der kompletten Sequenz ergibt schließlich:

$$\begin{array}{rcl} [x_3] & = & \left[12.767145334803704661710952009780892 3_4^5 \right] \\ [x_4] & = & \left[40.10916999113251975535008362290414_0^1 \right] \\ [x_5] & = & \left[262537412640768743.999999999999_2^3 \right] \quad . \end{array}$$

Es gilt also in der Tat $e^{\pi\sqrt{163}} \notin \boldsymbol{Z}$.

Kapitel 2

Genaue Standardfunktionen in dynamischen Rastern

Wir wollen in diesem Kapitel generell nur Gleitpunktraster betrachten, in denen weder Unter- noch Überlauf auftreten. Diese sind lediglich durch die Basis b der Zahldarstellung und die Anzahl der verfügbaren Mantissenstellen k charakterisiert und können in der üblichen Notation als $S(b, k, -\infty, \infty)$ mit $b \in \boldsymbol{N}$, $b > 1$ und $k \in \boldsymbol{N} \setminus \{0\}$ geschrieben werden. Wir werden abkürzend $S(b, k)$ zur Bezeichnung verwenden.
Unter einer auf k Stellen hochgenau implementierten Standardfunktion wollen wir im weiteren Verlauf dieser Arbeit folgendes verstehen.

Definition 2.0.1 *Eine intervallwertige Implementierung $\tilde{f} : D \supset \tilde{D} \to IS(b, k)$ einer reellen (Standard–)*
Funktion $f : \boldsymbol{R} \supseteq D \to \boldsymbol{R}$ heißt hochgenau auf k Stellen, falls für alle $x \in \tilde{D}$ mit $f(x) \neq 0$ folgende Bedingungen erfüllt sind:

1. $\inf(\tilde{f}(x)) \leq f(x) \leq \sup(\tilde{f}(x))$.

2. *$\tilde{f}(x)$ enthält höchstens drei Punkte des Rasters $S(b, k)$.*

Enthält $\tilde{f}(x)$ für alle $x \in \tilde{D}$ maximal zwei Elemente spricht man von maximaler Genauigkeit.

Bemerkungen:

- Die maximal genaue Implementierung einer Funktion ist i.a. nicht möglich, wie schon die Betrachtung der einfachen Funktion $f(x) = (\frac{1}{3}) \cdot x$ an der Stelle $x = 3$ im Raster $S(10, k)$, $k \in \boldsymbol{N} \setminus \{0\}$ zeigt (vgl. [Kr], S. 13).

- Für Raster mit endlichem Exponentenbereich ist obige Definition bereits ohne die Einschränkung $f(x) \neq 0$ sinnvoll und üblich.

- Im Hinblick auf die Verallgemeinerung auf dynamische Raster wurde in der obigen Definition der Definitionsbereich $\tilde{D}$ von $\tilde{f}$ bewußt nicht genauer spezifiziert. Im Falle eines festen Rasters wird in der Regel $\tilde{D} = D \cap S(b, k)$ gelten.

Die Ausdehnung auf den Fall $f(x) = 0$ ist nicht kanonisch. In der Praxis sollte hier eine geeignete obere Schranke für die Breite des Ergebnisintervalls $\tilde{f}(x)$ gefordert werden. Im Rahmen dieser Arbeit wurde der Wert b^{-k} gewählt. Die Beweggründe dazu werden aus der folgenden Überlegung ersichtlich.
Die oberen Schranken für die Absolutwerte des relativen Fehlers $\varepsilon_{\tilde{f}}$ liegen bei einer auf k Stellen maximal genauen Funktionsauswertung im Intervall $[b^{-k}, b^{1-k}]$, im hochgenauen Fall im Intervall $[2 \cdot b^{-k}, 2 \cdot b^{1-k}]$. Eine optimale Implementierung sollte also von einer Fehlerschranke von b^{-k} ausgehen, damit für eine möglichst große Teilmenge des Definitionsbereichs $\tilde{D}$ ein maximal genaues Ergebnis aus $IS(b, k)$ erzielt wird.
Für die zugehörigen Intervallversionen (d. h. für Argumente $X \in I\tilde{D}$) übertragen wir diese Terminologie sinngemäß jeweils auf Unter- und Obergrenze. Im übrigen gehen wir auf sie *nur* explizit ein, falls sie sich nicht aufgrund des Monotonieverhaltens der betrachteten Standardfunktion in trivialer Weise aus den Werten der zugehörigen Punktroutinen ergeben. Dies trifft in dieser Arbeit nur für Sinus und Kosinus, die allgemeine Potenzfunktion und den hyperbolischen Kosinus zu.
Für dynamische Raster (d. h. im Prinzip beliebige Mantissenlängen) kann die Definition von Hoch- und Maximalgenauigkeit verallgemeinert werden, indem man formal k als zusätzliches Funktionsargument aufnimmt und fordert, daß für alle $k \in \boldsymbol{N} \setminus \{0\}$ die Funktionen $\tilde{f}(., k)$ k–stellig hoch- bzw. maximalgenau sind.

Definition 2.0.2 *Für eine reelle (Standard–)Funktion* $f : \boldsymbol{R} \supseteq D \to \boldsymbol{R}$ *und ein dynamisches Raster*

$$S(b) := \bigcup_{k \in \boldsymbol{N} \setminus \{0\}} S(b, k)$$

heißt eine Implementierung $\tilde{f} : \tilde{D} \times \boldsymbol{N} \setminus \{0\} \to IS(b)$ *hoch- bzw. maximalgenau, falls für alle* $k \in \boldsymbol{N} \setminus \{0\}$ *und die Intervallrundungen* $rd_k : IS(b) \to IS(b, k)$ *sowie die Einschränkungen* $\tilde{f}_k : \tilde{D} \times \{k\} \to IS(b)$ *die Verknüpfungen* $rd_k \circ \tilde{f}_k$ *k-stellig hoch- bzw. maximalgenau sind.*

Die Auswertung von Standardfunktionen erfolgt in der Regel nach einem der drei folgenden Verfahren:

– direkt über ein geeignet gewähltes Approximationspolynom, wobei eventuell eine Argumentreduktion vor- und eine Rücktransformation des Ergebnisses nachzuschalten sind.

– mit Hilfe des Newton-Verfahrens.

– durch Rückführung auf andere Standardfunktionen unter Verwendung geeigneter algebraischer Identitäten (siehe etwa [ASt]).

Es gibt Standardfunktionen wie zum Beispiel den natürlichen Logarithmus, die mehrere Alternativen zulassen, und solche, die Mischformen aus der ersten und dritten Möglichkeit erfordern. Approximationen durch Kettenbrüche werden wir in dieser Arbeit nicht betrachten, da sie bestenfalls bei der Näherung einiger Konstanten wie π, e oder $\log(10)$ Verwendung finden und keine a priori Fehlerabschätzungen erlauben. Für a posteriori Abschätzungen sei an dieser Stelle auf klassische Werke ([Kh], [Pe]) verwiesen.

2.1 Grundlagen

2.1.1 Verwendeter Fehlerkalkül und grundlegende Abschätzungen

In dieser Arbeit wird der von Braune (vgl. [Br], Kap. 2.3) und Krämer [Kr] eingeführte, aus einer Spezialisierung des Landau-Symbols entstandene, Fehlerkalkül verwendet. Er erlaubt die einfache Handhabung von relativen Fehlern gleicher Größenordnung, die im wesentlichen durch die Basis b und die Mantissenlänge ℓ eines Rasters $S(b,\ell)$ sowie den Rundungsmodus festgelegt werden. Wir wollen diesen Kalkül hier jedoch in Abweichung von Braune und Krämer in einer Intervallschreibweise verwenden, die unserer Auffassung nach leichter zugänglich ist und die Notation dennoch nicht unnötig kompliziert.

Definition 2.1.1 *Das relative Fehlerintervall für ein Raster $S(b,\ell)$ ist definiert durch $E_\ell := \left[-b^{1-\ell}, b^{1-\ell}\right]$.*

Bemerkung: Unabhängig vom verwendeten Rundungsmodus gilt für jede Operation $* \in \{+,-,\cdot,/\}$ im Raster $S(b,\ell)$ (angedeutet durch den Index ℓ) und alle $x, y \in S(b,\ell)$

$$x *_\ell y \in (x * y) \cdot (1 + E_\ell)$$

Für die so definierten Fehlerintervalle und beliebige reelle Zahlen x, y und z gelten folgende, mit Hilfe der Rechenregeln für Intervalle leicht zu überprüfende, Beziehungen:

1. $x \cdot E_\ell = |x| \cdot E_\ell$.
2. $|x| \geq |y| \Rightarrow y \cdot E_\ell \subseteq x \cdot E_\ell$.
3. $x \cdot E_\ell \pm y \cdot E_\ell = |x| \cdot E_\ell + |y| \cdot E_\ell = (|x| + |y|)E_\ell$.
4. $(x \pm y \cdot E_k)E_\ell = x \cdot E_\ell \pm y \cdot E_k E_\ell$.
5. $E_\ell \cdot E_k = E_{\ell+k-1}$.
6. $E_k\left[y(x + E_\ell) + z(x + E_\ell)\right] \subseteq (|y| + |z|)E_k(x + E_\ell)$.

Bemerkung: Zum Beweis der sechsten können die Regeln 2–4 herangezogen werden.

Interessanter ist die folgende Möglichkeit, Potenzen des Fehlerterms $(1 + E_\ell)$ und damit den relativen Fehler des Produkts von n Werten in ℓ–stelliger Arithmetik abschätzen zu können (vgl. [Br], Kap. 2. 4. 1, Lemma 2).

Lemma 2.1.1 *Unter der Voraussetzung*

$$0 < n \leq \sqrt{2 \cdot b^{\ell-1}} - 1$$

für $\ell, n \in \boldsymbol{N} \setminus \{0\}$ und $\ell \geq 2$ gilt

$$(1 + E_\ell)^n \subseteq 1 + (n+1)E_\ell \quad .$$

Bemerkung: Für die Praxis stellt die Voraussetzung des Lemmas keine bedeutende Einschränkung dar. Im Falle einer 13-stelligen BCD-Arithmetik führt sie etwa zu der Beschränkung $n \leq 1414212$.

Ohne spezielle Voraussetzungen gelten die folgenden Abschätzungen.

Lemma 2.1.2 *Für $\varepsilon \geq 0$ und $n \in N$ gilt*

$$1 \leq (1+\varepsilon)^n \leq 1 + n \cdot \varepsilon \left(\exp(n \cdot \varepsilon) - \frac{n \cdot \varepsilon}{2} \right) \quad .$$

Beweis: Für alle $\varepsilon \geq 0$ und $n \in N$ gilt $(1+\varepsilon)^n \leq \exp(n \cdot \varepsilon)$. Die rechte Seite kann nun mit Hilfe der für alle $x \in \boldsymbol{R}_0^+$ geltenden Ungleichung

$$\exp(x) \leq 1 + x \left(\exp(x) - \frac{x}{2} \right)$$

weiter abgeschätzt werden, was auf die Behauptung führt.

Bemerkung: Eine spezielle Fassung dieses Lemmas, allerdings ohne Beweis, findet sich bei Wilkinson [Wi]. Dort wird $\varepsilon = 2^{-k}$ und $n \cdot \varepsilon < 0.1$ gesetzt und die Gültigkeit von $\left(1 + 2^{-k}\right)^n < 1 + 1.06 \cdot n \cdot 2^{-k}$ behauptet.

Eine untere Schranke liefert für $\varepsilon > -1$ und $n \in \boldsymbol{N}$ die Bernoullische Ungleichung

$$(1+\varepsilon)^n \geq 1 + n \cdot \varepsilon \quad .$$

Etwas schwieriger sind Summen von $n > 2$ Termen zu handhaben. Hier gilt:

Lemma 2.1.3 *Für die Summe S von N Werten $a_1, \ldots, a_N$ gilt*

$$\tilde{S} := a_1 +_\ell \ldots +_\ell a_N = S + \Delta_S$$

mit

$$\Delta_S \in E_\ell \sum_{k=2}^{N} (1 + E_\ell)^{N-k} (|a_1 + \ldots + a_k|) \quad .$$

(Beweis durch vollständige Induktion)

Bemerkung: Bei Braune ([Br], Kap. 2.4.2, Lemma 3) findet sich eine Abschätzung für die Summation der Größe nach geordneter Terme gleichen Vorzeichens, die sich allerdings jeweils um mindestens eine Größenordnung bezüglich der Basis b unterscheiden müssen.

2.1.2 Polynom-Auswertung nach dem Horner–Schema

In diesem Abschnitt soll mit Hilfe des zuvor vorgestellten Fehler–Kalküls eine Rundungsfehleranalyse verschiedener Varianten des Horner–Schemas durchgeführt werden. Zur algorithmischen Formulierung der Auswertung des Polynoms

$$P(x) = \sum_{k=0}^{N} a_k x^k$$

nach dem Horner–Schema verwenden wir folgende Bezeichnungen:

$$\begin{aligned} r_N &:= a_N, \\ r_{k-1} &:= a_{k-1} + x \cdot r_k, \quad k = N, \ldots, 1 \quad (N \quad \text{Horner–Schritte}) \quad . \end{aligned}$$

Nach Abarbeitung der Sequenz gilt $P(x) = r_0$.
Bei der Implementierung dieses Algorithmus auf dem Rechner treten jedoch Fehler in den Eingabedaten x und a_k $(k = 0, \ldots, N)$ bzw. bei der Ausführung der Addition und Multiplikation auf, die die Genauigkeit des Ergebnisses beeinflussen. Tatsächlich berechnet wird

$$\begin{aligned} \tilde{r}_N &:= a_N(1 + \varepsilon_{a_N}), \\ \tilde{r}_{k-1} &:= (a_{k-1}(1 + \varepsilon_{a_{k-1}}) + (x(1 + \varepsilon_x) \cdot \tilde{r}_k)(1 + \varepsilon_{\cdot}))(1 + \varepsilon_+), \quad k = N, \ldots, 1. \end{aligned}$$

Es gilt jetzt nur noch $P(x) + \Delta_P = \tilde{r}_0$.
Erfolgt die Rechnung in einem festen, ℓ-stelligen Raster so gilt ohne nähere Spezifikation des Rundungsmodus und unter der Annahme $\varepsilon_{a_k} \in E_\ell \quad (k = 0, \ldots, N)$ bzw. $\varepsilon_{\cdot}, \varepsilon_+ \in E_\ell$ die folgende Aussage.

Lemma 2.1.4 *Für den absoluten Fehler Δ_P bei der Auswertung eines Polynoms P an der mit einem relativen Fehler ε_x, $|\varepsilon_x| \leq 1$ behafteten Stelle x nach dem Horner–Schema gilt folgende Inklusion:*

$$\begin{aligned} \Delta_P \;\in\; & E_\ell(2 + E_\ell) \left\{ \sum_{k=0}^{N} \left| a_k x^k \right| (1 + E_\ell)^{2k} (1 + \varepsilon_x)^k + \sum_{k=1}^{N} \left| r_k x^k \right| (1 + E_\ell)^{2k-2} (1 + \varepsilon_x)^{k-1} \right\} \\ & + \varepsilon_x \sum_{k=1}^{N} r_k x^k (1 + E_\ell)^{2k} (1 + \varepsilon_x)^{k-1} \quad . \end{aligned}$$

Ist der konstante Term a_0 im verwendeten Raster exakt darstellbar, gilt schärfer:

$$\begin{aligned} \Delta_P \;\in\; & E_\ell \left\{ |a_0| + (2 + E_\ell) \left[\sum_{k=1}^{N} \left| a_k x^k \right| (1 + E_\ell)^{2k} (1 + \varepsilon_x)^k \right.\right. \\ & \left.\left. + \sum_{k=1}^{N} \left| r_k x^k \right| (1 + E_\ell)^{2k-2} (1 + \varepsilon_x)^{k-1} \right] \right\} + \varepsilon_x \sum_{k=1}^{N} r_k x^k (1 + E_\ell)^{2k} (1 + \varepsilon_x)^{k-1} \quad . \end{aligned}$$

Beweis: Nach Ausführung eines einzelnen Horner–Schrittes gilt für $k = N-1, \ldots, 1$ folgende Inklusion:

$$\begin{aligned} \tilde{r}_{k-1} &= \tilde{a}_{k-1} +_\ell \tilde{x} \cdot_\ell \tilde{r}_k \\ &\in \{a_{k-1}(1 + E_\ell) + x(r_k + \Delta_k)(1 + \varepsilon_x)(1 + E_\ell)\} (1 + E_\ell) \\ &\subseteq a_{k-1}(1 + E_\ell)^2 + x(r_k + \Delta_k)(1 + \varepsilon_x)(1 + E_\ell)^2 \\ &\subseteq a_{k-1}(1 + E_\ell)^2 + x r_k (1 + \varepsilon_x)(1 + E_\ell)^2 + x\Delta_k (1 + \varepsilon_x)(1 + E_\ell)^2 \\ &\subseteq a_{k-1}(1 + E_\ell)^2 + x r_k (1 + E_\ell)^2 + \varepsilon_x x r_k (1 + E_\ell)^2 + x\Delta_k (1 + \varepsilon_x)(1 + E_\ell)^2 \\ &\subseteq a_{k-1} + x r_k + a_{k-1}(2E_\ell + E_\ell^2) + x r_k (2E_\ell + E_\ell^2) + x\Delta_k (1 + \varepsilon_x)(1 + E_\ell)^2 \\ &\quad + \varepsilon_x x r_k (1 + E_\ell)^2 \\ &\subseteq r_{k-1} + E_\ell(2 + E_\ell)\left(|a_{k-1}| + |x r_k|\right) + x\Delta_k (1 + \varepsilon_x)(1 + E_\ell)^2 + \varepsilon_x x r_k (1 + E_\ell)^2 \quad . \end{aligned}$$

Für den absoluten Fehler Δ_{k-1} gilt also

$$\Delta_{k-1} \in E_\ell(2+E_\ell)\left(|a_{k-1}| + |xr_k|\right) + x\Delta_k(1+\varepsilon_x)(1+E_\ell)^2 + \varepsilon_x x r_k (1+E_\ell)^2 \quad .$$

Induktionsanfang: $(N = 1)$

$$\begin{aligned}
\Delta_0 \;\in\;& E_\ell(2+E_\ell)\left(|a_0| + |xr_1|\right) + x\Delta_1(1+\varepsilon_x)(1+E_\ell)^2 + \varepsilon_x x r_1(1+E_\ell)^2 \\
& \quad \text{mit } \Delta_1 \in a_1 E_\ell \text{ folgt} \\
\subseteq\;& E_\ell\left\{(2+E_\ell)\left(|a_0| + |xr_1|\right) + \left|a_1 x^1\right|(1+\varepsilon_x)(1+E_\ell)^2\right\} + \varepsilon_x x r_1(1+E_\ell)^2 \\
& \text{da } (2+E_\ell) \geq 1 \text{ gilt, folgt} \\
\subseteq\;& E_\ell\left\{(2+E_\ell)\left(|a_0| + |xr_1|\right) + (2+E_\ell)\left|a_1 x^1\right|(1+\varepsilon_x)(1+E_\ell)^2\right\} + \varepsilon_x x r_1(1+E_\ell)^2 \\
\subseteq\;& E_\ell(2+E_\ell)\left\{\sum_{k=0}^{1}\left|a_k x^k\right|(1+E_\ell)^{2k}(1+\varepsilon_x)^k + \sum_{k=1}^{1}\left|r_k x^k\right|(1+E_\ell)^{2k-2}(1+\varepsilon_x)^{k-1}\right\} \\
& + \varepsilon_x \sum_{k=1}^{1} r_k x^k (1+E_\ell)^{2k}(1+\varepsilon_x)^{k-1}
\end{aligned}$$

Induktionsschritt: $(N-1$ nach $N)$

$$\begin{aligned}
\Delta_1 \;\in\;& E_\ell(2+E_\ell)\left\{\sum_{k=0}^{N-1}\left|a_{k+1}x^k\right|(1+E_\ell)^{2k}(1+\varepsilon_x)^k \right. \\
& \left. + \sum_{k=1}^{N-1}\left|r_{k+1}x^k\right|(1+E_\ell)^{2k-2}(1+\varepsilon_x)^{k-1}\right\} + \varepsilon_x \sum_{k=1}^{N-1} r_{k+1}x^k(1+E_\ell)^{2k}(1+\varepsilon_x)^{k-1}
\end{aligned}$$

Einsetzen führt auf

$$\begin{aligned}
\Delta_0 \;\in\;& E_\ell(2+E_\ell)\left(|a_0| + |xr_1|\right) + x\Delta_1(1+\varepsilon_x)(1+E_\ell)^2 + \varepsilon_x x r_1(1+E_\ell)^2 \\
\subseteq\;& E_\ell(2+E_\ell)\left(|a_0| + |xr_1|\right) + E_\ell\,(2+E_\ell)\left\{\sum_{k=0}^{N-1}\left|a_{k+1}x^{k+1}\right|(1+E_\ell)^{2k+2}(1+\varepsilon_x)^{k+1}\right. \\
& \left. + \sum_{k=1}^{N-1}\left|r_{k+1}x^{k+1}\right|(1+E_\ell)^{2k}(1+\varepsilon_x)^k\right\} + \varepsilon_x \sum_{k=1}^{N-1} r_{k+1}x^{k+1}(1+E_\ell)^{2k+2}(1+\varepsilon_x)^k \\
& + \varepsilon_x x r_1 (1+E_\ell)^2 \\
\subseteq\;& E_\ell\,(2+E_\ell)\left\{\sum_{k=0}^{N}\left|a_k x^k\right|(1+E_\ell)^{2k}(1+\varepsilon_x)^k + \sum_{k=1}^{N}\left|r_k x^k\right|(1+E_\ell)^{2k-2}(1+\varepsilon_x)^{k-1}\right\} \\
& + \varepsilon_x \sum_{k=1}^{N} r_k x^k (1+E_\ell)^{2k}(1+\varepsilon_x)^{k-1} \quad .
\end{aligned}$$

Den zweiten Teil des Lemmas erhält man durch Einsetzen der Beziehung für Δ_1 in

$$\Delta_0 \in E_\ell(2+E_\ell)\left|xr_1\right| + E_\ell\left|a_0\right| + x\Delta_1(1+\varepsilon_x)(1+E_\ell)^2 + \varepsilon_x x r_1(1+E_\ell)^2 \quad .$$

□

Bemerkung: Eine schwächere — weil die Auswirkung evtl. auftretender Auslöschung überschätzende — Version dieses Lemmas für den relativen Fehler ε_P findet sich bei Braune ([Br], Kap. 2.4.4, Lemma 5). Dort wird allerdings keine Störung des Argumentes x zugelassen.

Die hier bewiesene Version des Lemmas spiegelt die Verhältnisse beim Rechnen in einem festen, ℓ-stelligen Raster wieder, wie sie zum Beispiel bei Verwendung der IEEE-Gleitpunktarithmetik herrschen.
Langzahl–Arithmetiken werden in der Regel jedoch nicht von der Hardware unterstützt und müssen softwaremäßig realisiert werden. Dies eröffnet zusätzliche Möglichkeiten. Es können Mantissen variabler Länge verwendet oder Rundungen eingespart werden, um so partiell mit höherer Genauigkeit zu arbeiten. Das Horner–Schema bietet hierzu verschiedene Ansatzpunkte.
Zunächst kann die in jedem Horner–Schritt stattfindende Multiplikation exakt durchgeführt werden, was in unserer Terminologie $\varepsilon_{\cdot} = 0$ bedeutet. Die Stellenzahl verdoppelt sich dabei höchstens und eine Rundungsoperation wird ebenfalls eingespart, da erst nach der Addition wieder ins ursprüngliche ℓ-stellige Raster gerundet wird. Für ein festes Raster setzt dieses Vorgehen die Existenz einer multiply–and–add–Operation voraus, die den gleichen Fehlerabschätzungen wie die anderen Grundrechenarten genügt. Eine entsprechende Modifikation führt auf die folgende Aussage.

Lemma 2.1.5 *Für den absoluten Fehler Δ_P bei der Auswertung eines Polynoms P an der mit einem relativen Fehler ε_x, $|\varepsilon_x| \leq 1$ behafteten Stelle x nach dem wie oben beschriebenen, abgewandelten Horner–Schema gilt folgende Inklusion:*

$$\begin{aligned}\Delta_P \;\in\; & E_\ell \left\{ (2+E_\ell) \sum_{k=0}^{N-1} \left| a_k x^k \right| (1+E_\ell)^k (1+\varepsilon_x)^k + \left| a_N x^N \right| (1+E_\ell)^N (1+\varepsilon_x)^N \right. \\ & \left. + \sum_{k=1}^{N} \left| r_k x^k \right| (1+E_\ell)^{k-1} (1+\varepsilon_x)^{k-1} \right\} + \varepsilon_x \sum_{k=1}^{N} r_k x^k (1+E_\ell)^k (1+\varepsilon_x)^{k-1} \quad .\end{aligned}$$

Ist der konstante Term a_0 exakt darstellbar, gilt schärfer:

$$\begin{aligned}\Delta_P \;\in\; & E_\ell \left\{ |a_0| + (2+E_\ell) \sum_{k=1}^{N-1} \left| a_k x^k \right| (1+E_\ell)^k (1+\varepsilon_x)^k + \left| a_N x^N \right| (1+E_\ell)^N (1+\varepsilon_x)^N \right. \\ & \left. + \sum_{k=1}^{N} \left| r_k x^k \right| (1+E_\ell)^{k-1} (1+\varepsilon_x)^{k-1} \right\} + \varepsilon_x \sum_{k=1}^{N} r_k x^k (1+E_\ell)^k (1+\varepsilon_x)^{k-1} \quad .\end{aligned}$$

Beweis: Nach Ausführung eines einzelnen so abgewandelten Horner–Schrittes gilt für $k = N-1, \ldots, 1$ folgende Inklusion:

$$\begin{aligned}\tilde{r}_{k-1} &= \tilde{a}_{k-1} +_\ell \tilde{x} \cdot \tilde{r}_k \\ &\in \left\{ a_{k-1}(1+E_\ell) + x(r_k + \Delta_k)(1+\varepsilon_x) \right\} (1+E_\ell) \\ &\subseteq a_{k-1}(1+E_\ell)^2 + x(r_k+\Delta_k)(1+\varepsilon_x)(1+E_\ell) \\ &\subseteq a_{k-1} + x r_k + a_{k-1}(2E_\ell + E_\ell^2) + x r_k E_\ell + x\Delta_k (1+\varepsilon_x)(1+E_\ell) + \varepsilon_x x r_k (1+E_\ell) \\ &\subseteq r_{k-1} + E_\ell \left(|a_{k-1}| + |x r_k| \right) + E_\ell (1+E_\ell) \, |a_{k-1}| + x\Delta_k (1+\varepsilon_x)(1+E_\ell) \\ &\quad + \varepsilon_x x r_k (1+E_\ell) \quad .\end{aligned}$$

Induktionsanfang: $(N = 1)$

$$\begin{aligned}
\Delta_0 &\in E_\ell\left(|a_0| + |xr_1|\right) + E_\ell(1+E_\ell)\,|a_0| + x\Delta_1(1+\varepsilon_x)(1+E_\ell) + \varepsilon_x x r_1(1+E_\ell) \\
&\quad \text{mit } \Delta_1 \in a_1 E_\ell \text{ folgt} \\
&\subseteq E_\ell\left\{|a_0| + |xr_1| + (1+E_\ell)\,|a_0| + \left|a_1 x^1\right|(1+\varepsilon_x)(1+E_\ell)\right\} + \varepsilon_x x r_1(1+E_\ell) \\
&\subseteq E_\ell\left\{(2+E_\ell)\sum_{k=0}^{0}\left|a_k x^k\right|(1+E_\ell)^k(1+\varepsilon_x)^k + \left|a_1 x^1\right|(1+E_\ell)(1+\varepsilon_x)\right. \\
&\quad \left. + \sum_{k=1}^{1}\left|r_k x^k\right|(1+E_\ell)^{k-1}(1+\varepsilon_x)^{k-1}\right\} + \varepsilon_x \sum_{k=1}^{1} r_k x^k (1+E_\ell)^k(1+\varepsilon_x)^{k-1}
\end{aligned}$$

Induktionsschritt: $(N-1$ nach $N)$

$$\begin{aligned}
\Delta_1 &\in E_\ell\left\{(2+E_\ell)\sum_{k=0}^{N-2}\left|a_{k+1} x^k\right|(1+E_\ell)^k(1+\varepsilon_x)^k + \left|a_N x^{N-1}\right|(1+E_\ell)^{N-1}(1+\varepsilon_x)^{N-1}\right. \\
&\quad \left. + \sum_{k=1}^{N-1}\left|r_{k+1} x^k\right|(1+E_\ell)^{k-1}(1+\varepsilon_x)^{k-1}\right\} + \varepsilon_x \sum_{k=1}^{N-1} r_{k+1} x^k (1+E_\ell)^k(1+\varepsilon_x)^{k-1}
\end{aligned}$$

Einsetzen in

$$\Delta_0 \in E_\ell\left(|a_0| + |xr_1|\right) + E_\ell(1+E_\ell)\,|a_0| + x\Delta_1(1+\varepsilon_x)(1+E_\ell) + \varepsilon_x x r_1(1+E_\ell)$$

und Umsortieren führen schließlich auf die Behauptung. Der Beweis des zweiten Teils des Lemmas ergibt sich durch Einsetzen in

$$\Delta_0 \in E_\ell\left(|a_0| + |xr_1|\right) + E_\ell\,|a_0| + x\Delta_1(1+\varepsilon_x)(1+E_\ell) + \varepsilon_x x r_1(1+E_\ell) \quad .$$

□

Bemerkung: Die beiden Teilaussage des Lemmas lassen sich etwas abschwächen und sind dann leichter zu handhaben:

$$\begin{aligned}
\Delta_P &\in E_\ell\left\{(2+E_\ell)\sum_{k=0}^{N}\left|a_k x^k\right|(1+E_\ell)^k(1+\varepsilon_x)^k + \sum_{k=1}^{N}\left|r_k x^k\right|(1+E_\ell)^{k-1}(1+\varepsilon_x)^{k-1}\right\} \\
&\quad + \varepsilon_x \sum_{k=1}^{N} r_k x^k (1+E_\ell)^k(1+\varepsilon_x)^{k-1}
\end{aligned}$$

bzw.

$$\begin{aligned}
\Delta_P &\in E_\ell\left\{|a_0| + (2+E_\ell)\sum_{k=1}^{N}\left|a_k x^k\right|(1+E_\ell)^k(1+\varepsilon_x)^k + \sum_{k=1}^{N}\left|r_k x^k\right|(1+E_\ell)^{k-1}(1+\varepsilon_x)^{k-1}\right\} \\
&\quad + \varepsilon_x \sum_{k=1}^{N} r_k x^k (1+E_\ell)^k(1+\varepsilon_x)^{k-1} \quad .
\end{aligned}$$

Eine weitere Modifikation besteht in der zusätzlichen Bereitstellung der Polynomkoeffizienten mit doppelter Genauigkeit, d.h. $\varepsilon_{a_n} \in E_{2\ell}$ $(n = 0, \ldots, N)$. Erst die Addition führt dann

wieder ins ℓ-stellige Ausgangsraster. Das umgekehrte Vorgehen, also exakte Durchführung der Addition, ist wegen des unkontrollierten Anwachsens der Stellenzahl nicht zu empfehlen. Es bringt gegenüber der Kombination der beiden anderen vorgeschlagenen Modifikationen auch keinen großen Gewinn mehr. Im vorausgehenden Lemma ist lediglich der Faktor $(2+E_\ell)$ bei der ersten Summe innerhalb der geschweiften Klammer durch Eins zu ersetzen.

Lemma 2.1.6 *Für den absoluten Fehler Δ_P bei der Auswertung eines Polynoms P an der mit einem relativen Fehler ε_x, $|\varepsilon_x| \leq 1$ behafteten Stelle x nach dem zuletzt oben beschriebenen, abgewandelten Horner–Schema gilt folgende Inklusion:*

$$\begin{aligned}\Delta_P \;\in\; & E_\ell \left\{ (1+E_{l+1}+E_{2l}) \sum_{k=0}^{N-1} \left|a_k x^k\right| (1+E_\ell)^k (1+\varepsilon_x)^k + \left|a_N x^N\right| (1+E_\ell)^N (1+\varepsilon_x)^N \right. \\ & \left. + \sum_{k=1}^{N} \left|r_k x^k\right| (1+E_\ell)^{k-1} (1+\varepsilon_x)^{k-1} \right\} + \varepsilon_x \sum_{k=1}^{N} r_k x^k (1+E_\ell)^k (1+\varepsilon_x)^{k-1} \quad .\end{aligned}$$

Ist der konstante Term a_0 exakt darstellbar, gilt etwas schärfer:

$$\begin{aligned}\Delta_P \;\in\; & E_\ell \left\{ |a_0| + (1+E_{l+1}+E_{2l}) \sum_{k=1}^{N-1} \left|a_k x^k\right| (1+E_\ell)^k (1+\varepsilon_x)^k \right. \\ & \left. + \left|a_N x^N\right| (1+E_\ell)^N (1+\varepsilon_x)^N + \sum_{k=1}^{N} \left|r_k x^k\right| (1+E_\ell)^{k-1} (1+\varepsilon_x)^{k-1} \right\} \\ & + \varepsilon_x \sum_{k=1}^{N} r_k x^k (1+E_\ell)^k (1+\varepsilon_x)^{k-1} \quad .\end{aligned}$$

Beweis: Nach Ausführung eines einzelnen so abgewandelten Horner–Schrittes gilt für $k = N-1, \ldots, 1$ folgende Inklusion:

$$\begin{aligned}\tilde{r}_{k-1} &= \tilde{a}_{k-1} +_\ell \tilde{x} \cdot \tilde{r}_k \\ &\in \{a_{k-1}(1+E_{2\ell}) + x(r_k+\Delta_k)(1+\varepsilon_x)\}\,(1+E_\ell) \\ &\subseteq a_{k-1}(1+E_\ell)(1+E_{2\ell}) + x(r_k+\Delta_k)(1+\varepsilon_x)(1+E_\ell) \\ &\subseteq a_{k-1} + x r_k + a_{k-1}(E_\ell + E_{2\ell} + E_{3\ell-1}) + x r_k E_\ell + x\Delta_k(1+\varepsilon_x)(1+E_\ell) \\ &\quad + \varepsilon_x x r_k (1+E_\ell) \\ &\subseteq r_{k-1} + E_\ell\,(|a_{k-1}| + |x r_k|) + E_\ell(E_{\ell+1}+E_{2\ell})\,|a_{k-1}| + x\Delta_k(1+\varepsilon_x)(1+E_\ell) \\ &\quad + \varepsilon_x x r_k (1+E_\ell) \quad .\end{aligned}$$

Der Rest des Beweises verläuft vollkommen anlog. □

Bemerkung: Eine dem vorherigen Lemma entsprechende Abschwächung ist hier in dieser Form nicht möglich, da der Faktor $(1+E_{\ell+1}+E_{2\ell})$ nicht notwendigerweise größer als Eins ist.

In der vorliegenden Form sind die vorangehenden Lemmata nicht besonders hilfreich, da sie unhandliche Potenzen der Fehlerterme $(1+E_\ell)$ bzw. $(1+\varepsilon_x)$ enthalten. Hier bietet sich Lemma 2.1.1 an. Wir wenden es zunächst auf Lemma 2.1.4 in seiner abgeschwächten Form an und setzen dabei zusätzlich $\varepsilon_x = 0$ voraus.

Lemma 2.1.7 *Für den absoluten Fehler Δ_P bei der Auswertung eines Polynoms P an der Stelle x nach dem Horner–Schema gilt unter den Voraussetzungen von Lemma 2.1.1 folgende Inklusion:*

$$\Delta_P \quad \in \quad E_\ell(2+E_\ell)\left\{(1+E_\ell)\sum_{k=0}^{N}\left|a_kx^k\right| + (1+2E_\ell)\sum_{k=1}^{N}k\left|a_kx^k\right| + E_\ell\sum_{k=1}^{N}k^2\left|a_kx^k\right|\right\} \quad .$$

Ist der konstante Term a_0 im verwendeten Raster exakt darstellbar, gilt schärfer:

$$\Delta_P \quad \in \quad E_\ell\left\{|a_0| + (2+E_\ell)\left[(1+E_\ell)\sum_{k=1}^{N}\left|a_kx^k\right| + (1+2E_\ell)\sum_{k=1}^{N}k\left|a_kx^k\right| + E_\ell\sum_{k=1}^{N}k^2\left|a_kx^k\right|\right]\right\}.$$

Beweis: Nach Lemma 2.1.4 und unter der Voraussetzung $\varepsilon_x = 0$ gilt:

$$\begin{aligned}
\Delta_P \quad &\in \quad E_\ell(2+E_\ell)\left\{\sum_{k=0}^{N}\left|a_kx^k\right|(1+E_\ell)^{2k} + \sum_{k=1}^{N}\left|r_kx^k\right|(1+E_\ell)^{2k-2}\right\} \\
&\subseteq \quad E_\ell(2+E_\ell)\left\{\sum_{k=0}^{N}\left|a_kx^k\right|(1+(2k+1)E_\ell) + \sum_{k=1}^{N}\left|r_kx^k\right|(1+(2k-1)E_\ell)\right\} \\
&\subseteq \quad E_\ell(2+E_\ell)\left\{\sum_{k=0}^{N}\left|a_kx^k\right|(1+(2k+1)E_\ell) + \sum_{k=1}^{N}\left(\sum_{m=k}^{N}|a_mx^m|\right)(1+(2k-1)E_\ell)\right\} \\
&\subseteq \quad E_\ell(2+E_\ell)\left\{\sum_{k=0}^{N}\left|a_kx^k\right|(1+(2k+1)E_\ell) + \sum_{k=1}^{N}\left(\sum_{m=1}^{k}(1+(2m-1)E_\ell)\right)\left|a_kx^k\right|\right\} \\
&\subseteq \quad E_\ell(2+E_\ell)\left\{\sum_{k=0}^{N}\left|a_kx^k\right|(1+(2k+1)E_\ell) + \sum_{k=1}^{N}(k+k^2E_\ell)\left|a_kx^k\right|\right\} \\
&\subseteq \quad E_\ell(2+E_\ell)\left\{(1+E_\ell)\sum_{k=0}^{N}\left|a_kx^k\right| + (1+2E_\ell)\sum_{k=1}^{N}k\left|a_kx^k\right| + E_\ell\sum_{k=1}^{N}k^2\left|a_kx^k\right|\right\} \quad .
\end{aligned}$$

Der Beweis des zweiten Teils der Behauptung erfolgt analog mit Hilfe des entsprechenden Teils von Lemma 2.1.4. □

Bemerkung: Falls zusätzlich zu den Voraussetzungen des Lemmas $a_kx^k \geq 0$ für alle $k = 0, \ldots, n$ gilt, lassen sich obige Aussagen leichter lesbar formulieren. Sie lauten dann

$$\Delta_P \in E_\ell(2+E_\ell)\left\{(1+E_\ell)P(x) + (1+3E_\ell)xP'(x) + E_\ell x^2P''(x)\right\}$$

bzw. falls a_0 exakt darstellbar ist

$$\Delta_P \in E_\ell\left\{a_o + (2+E_\ell)\left[(1+E_\ell)(P(x)-a_0) + (1+3E_\ell)xP'(x) + E_\ell x^2P''(x)\right]\right\} \quad .$$

Die linearisierte Version von Lemma 2.1.5 lautet wie folgt.

Lemma 2.1.8 *Für den absoluten Fehler Δ_P bei der Auswertung eines Polynoms P an der Stelle x nach der ersten beschriebenen Modifikation des Horner–Schemas gilt unter den Voraussetzungen von Lemma 2.1.1 folgende Inklusion:*

$$\Delta_P \quad \in \quad E_\ell\left\{(2+3E_\ell+E_\ell^2)\sum_{k=0}^{N}\left|a_kx^k\right| + (1+\frac{5}{2}E_\ell+E_\ell^2)\sum_{k=1}^{N}k\left|a_kx^k\right| + \frac{E_\ell}{2}\sum_{k=1}^{N}k^2\left|a_kx^k\right|\right\} \quad .$$

Ist der konstante Term a_0 im verwendeten Raster exakt darstellbar, gilt schärfer:

$$\begin{aligned}\Delta_P \quad &\in \quad E_\ell\Big\{|a_0| + (2+3E_\ell+E_\ell^2)\sum_{k=1}^{N}\left|a_kx^k\right| + (1+\frac{5}{2}E_\ell+E_\ell^2)\sum_{k=1}^{N}k\left|a_kx^k\right| \\ &\quad + \frac{E_\ell}{2}\sum_{k=1}^{N}k^2\left|a_kx^k\right|\Big\} \quad .\end{aligned}$$

Bemerkung: Auch die Aussagen dieses Lemmas lassen sich unter der zusätzlichen Voraussetzung $a_kx^k \geq 0$ für alle $k = 0, \ldots, n$ leichter lesbar formulieren. Sie lauten dann

$$\Delta_P \in E_\ell\Big\{(2+3E_\ell+E_\ell^2)P(x) + (1+3E_\ell+E_\ell^2)xP'(x) + \frac{E_\ell}{2}x^2P''(x)\Big\}$$

bzw.

$$\Delta_P \in E_\ell\Big\{a_0 + (2+3E_\ell+E_\ell^2)(P(x)-a_0) + (1+3E_\ell+E_\ell^2)xP'(x) + \frac{E_\ell}{2}x^2P''(x)\Big\} \quad .$$

Der Vollständigkeit halber sei hier auch noch die linearisierte Version von Lemma 2.1.6 angegeben. Diese kam allerdings in den durchgeführten Implementierungen nicht zur Anwendung.

Lemma 2.1.9 *Für den absoluten Fehler Δ_P bei der Auswertung eines Polynoms P an der Stelle x nach der zweiten beschriebenen Modifikation des Horner–Schemas gilt unter den Voraussetzungen von Lemma 2.1.1 folgende Inklusion:*

$$\begin{aligned}\Delta_P \quad &\in \quad E_\ell\Big\{(1+E_\ell)\sum_{k=0}^{N}\left|a_kx^k\right| + \frac{3}{2}E_\ell\sum_{k=1}^{N}k\left|a_kx^k\right| + \frac{1}{2}E_\ell\sum_{k=1}^{N}k^2\left|a_kx^k\right| \\ &\quad + (E_{\ell+1}+2E_{2\ell}+E_{3\ell-1})\sum_{k=0}^{N-1}\left|a_kx^k\right| + (E_{2\ell}+E_{3\ell-1})\sum_{k=0}^{N-1}\left|a_kx^k\right|\Big\} \quad .\end{aligned}$$

Ist der konstante Term a_0 im verwendeten Raster exakt darstellbar, gilt schärfer:

$$\begin{aligned}\Delta_P \quad &\in \quad E_\ell\Big\{|a_0| + (1+E_\ell)\sum_{k=1}^{N}\left|a_kx^k\right| + \frac{3}{2}E_\ell\sum_{k=1}^{N}k\left|a_kx^k\right| + \frac{1}{2}E_\ell\sum_{k=1}^{N}k^2\left|a_kx^k\right| \\ &\quad + (E_{\ell+1}+2E_{2\ell}+E_{3\ell-1})\sum_{k=0}^{N-1}\left|a_kx^k\right| + (E_{2\ell}+E_{3\ell-1})\sum_{k=0}^{N-1}\left|a_kx^k\right|\Big\} \quad .\end{aligned}$$

Die bisher aufgeführten linearisierten Versionen decken nur den Fall $\varepsilon_x = 0$ ab. Die Berechnung vieler Standardfunktionen verlangt jedoch die Auswertung von Polynomen in x^2. Es soll deshalb abschließend noch der Fall $\varepsilon_x \in E_\ell$ betrachtet werden. Die entsprechende Version von Lemma 2.1.4 lautet:

Lemma 2.1.10 *Für den absoluten Fehler Δ_P bei der Auswertung eines Polynoms P an der mit dem Fehler $\varepsilon_x \in E_\ell$ behafteten Stelle x nach dem Horner–Schemas gilt unter den*

Voraussetzungen von Lemma 2.1.1 folgende Inklusion:

$$\begin{aligned}\Delta_P \quad &\in \quad E_\ell\,(2+E_\ell)\left\{(1+E_\ell)\sum_{k=0}^{N}\left|a_kx^k\right| + (1+\frac{7}{2}E_\ell)\sum_{k=1}^{N}k\left|a_kx^k\right| + \frac{3}{2}E_\ell\sum_{k=1}^{N}k^2\left|a_kx^k\right|\right\} \\ &\quad + E_\ell\left\{(1+\frac{3}{2}E_\ell)\sum_{k=1}^{N}ka_kx^k + \frac{3}{2}E_\ell\sum_{k=1}^{N}k^2a_kx^k\right\} \quad .\end{aligned}$$

Ist der konstante Term a_0 im verwendeten Raster exakt darstellbar, gilt schärfer:

$$\begin{aligned}\Delta_P \quad &\in \quad E_\ell\,(2+E_\ell)\left\{|a_0| + (1+E_\ell)\sum_{k=1}^{N}\left|a_kx^k\right| + (1+\frac{7}{2}E_\ell)\sum_{k=1}^{N}k\left|a_kx^k\right| + \frac{3}{2}E_\ell\sum_{k=1}^{N}k^2\left|a_kx^k\right|\right\} \\ &\quad + E_\ell\left\{(1+\frac{3}{2}E_\ell)\sum_{k=1}^{N}ka_kx^k + \frac{3}{2}E_\ell\sum_{k=1}^{N}k^2a_kx^k\right\} \quad .\end{aligned}$$

Bemerkung: Auch die Aussagen dieses Lemmas lassen sich unter der zusätzlichen Voraussetzung $a_kx^k \geq 0$ für alle $k = 0, \ldots, n$ leichter lesbar formulieren. Sie lauten dann

$$\Delta_P \in E_\ell\left\{(2+3E_\ell+E_\ell^2)P(x) + (3+14E_\ell+5E_\ell^2)xP'(x) + (\frac{9}{2}E_\ell + \frac{3}{2}E_\ell^2)x^2P''(x)\right\}$$

bzw.

$$\begin{aligned}\Delta_P \quad &\in \quad E_\ell\left\{(2+E_\ell)a_0 + (2+3E_\ell+E_\ell^2)(P(x)-a_0) + (3+14E_\ell+5E_\ell^2)xP'(x)\right. \\ &\quad \left. + (\frac{9}{2}E_\ell + \frac{3}{2}E_\ell^2)x^2P''(x)\right\} \quad .\end{aligned}$$

Die entsprechende Version von Lemma 2.1.5 lautet:

Lemma 2.1.11 *Für den absoluten Fehler Δ_P bei der Auswertung eines Polynoms P an der mit dem Fehler $\varepsilon_x \in E_\ell$ behafteten Stelle x nach der ersten beschriebenen Modifikation des Horner–Schemas gilt unter den Voraussetzungen von Lemma 2.1.1 folgende Inklusion:*

$$\begin{aligned}\Delta_P \quad &\in \quad E_\ell\left\{(2+3E_\ell+E_\ell^2)\sum_{k=0}^{N}\left|a_kx^k\right| + (1+4E_\ell+2E_\ell^2)\sum_{k=1}^{N}k\left|a_kx^k\right| + E_\ell\sum_{k=1}^{N}k^2\left|a_kx^k\right|\right\} \\ &\quad + E_\ell\left\{(1+2E_\ell)xP'(x) + E_\ell x^2P''(x)\right\} \quad .\end{aligned}$$

Ist der konstante Term a_0 im verwendeten Raster exakt darstellbar, gilt schärfer:

$$\begin{aligned}\Delta_P \quad &\in \quad E_\ell\left\{|a_0| + (2+3E_\ell+E_\ell^2)\sum_{k=1}^{N}\left|a_kx^k\right| + (1+4E_\ell+2E_\ell^2)\sum_{k=1}^{N}k\left|a_kx^k\right|\right. \\ &\quad \left. + E_\ell\sum_{k=1}^{N}k^2\left|a_kx^k\right|\right\} + E_\ell\left\{(1+2E_\ell)xP'(x) + E_\ell x^2P''(x)\right\} \quad .\end{aligned}$$

Bemerkung: Auch die Aussagen dieses Lemmas lassen sich unter der zusätzlichen Voraussetzung $a_kx^k \geq 0$ für alle $k = 0, \ldots, n$ leichter lesbar formulieren. Sie lauten dann

$$\Delta_P \in E_\ell\left\{(2+3E_\ell+E_\ell^2)P(x) + (2+7E_\ell+2E_\ell^2)xP'(x) + 2E_\ell x^2P''(x)\right\}$$

bzw.

$$\Delta_P \in E_\ell\left\{a_0 + (2+3E_\ell+E_\ell^2)(P(x)-a_0) + (2+7E_\ell+2E_\ell^2)xP'(x) + 2E_\ell x^2P''(x)\right\} \quad .$$

2.2 Standardfunktionen über Potenzreihen

Die generelle Vorgehensweise bei der Berechnung von Standardfunktionen über Potenzreihen sieht wie folgt aus.
Im einfachsten Fall ist keine Argumentreduktion erforderlich und der Gesamtfehler ε_{ges} setzt sich aus dem Approximationsfehler ε_{app} und dem Auswertefehler ε_P zusammen. Ersterer rührt vom vorzeitigen Abbruch der Potenzreihenentwicklung her und kann für alle betrachteten Potenzreihen mit Hilfe des Argumentes x und des Polynomgrades N betragsmäßig nach oben abgeschätzt werden.
Der Auswertefehler wiederum ist — wie sich im folgenden herausstellen wird — mit Hilfe der im vorherigen Abschnitt hergeleiteten Lemmata für alle verwendeten Potenzreihen *unabhängig* vom Polynomgrad N abschätzbar. Es ist dazu lediglich eine Beschränkung des Argumentes x erforderlich, die sich aber in der Regel aus Konvergenzgründen sowieso ergibt. Falls das Horner–Schema unter Verwendung ℓ–stelliger Arithmetik durchgeführt wird, können also jeweils Schranken

$$|\varepsilon_P| \leq c \cdot b^{1-\ell} \quad \text{mit} \quad c \in \boldsymbol{R}^+ \tag{2.1}$$

angegeben werden. Der Gesamtfehler der Polynomapproximation ergibt sich insgesamt als

$$\varepsilon_{ges} = \varepsilon_P + \varepsilon_{app} \cdot (1 + \varepsilon_P) \quad .$$

Soll insgesamt ein auf k Stellen hochgenaues Ergebnis geliefert werden, ist also

$$|\varepsilon_P + \varepsilon_{app} \cdot (1 + \varepsilon_P)| \leq b^{-k}$$

zu fordern. Einsetzen von (2.1) und Auflösen nach ε_{app} führt auf

$$|\varepsilon_{app}| \leq b^{-k} \cdot \frac{1 - c \cdot b^{1-\ell+k}}{1 + c \cdot b^{1-\ell}} \quad . \tag{2.2}$$

Diese Ungleichung kann nur erfüllt werden, falls $1 - c \cdot b^{1-\ell+k} > 0$ gilt. Eine einfache Umformung führt auf die Forderung

$$\ell > k + 1 + \log_b(c) \quad . \tag{2.3}$$

Ist die zur Durchführung des Horner–Schemas notwendige Stellenzahl ℓ entsprechend gewählt, kann im Rückschluß mit Hilfe von (2.2) eine obere Schranke für den Approximationsfehler angegeben werden. Dies erlaubt abschließend die Bestimmung des erforderlichen Polynomgrades N, da wie bereits erwähnt für das Argument x stets obere Schranken zu wählen sind. Es sind dabei allerdings nachträglich die Voraussetzungen von Lemma 2.1.1 zu verifizieren, d. h. die Gültigkeit der dazu hinreichenden Bedingung

$$\ell \geq 2 \cdot \log_b(N + 1) - \log_b(2) + 1 \quad . \tag{2.4}$$

zu prüfen.

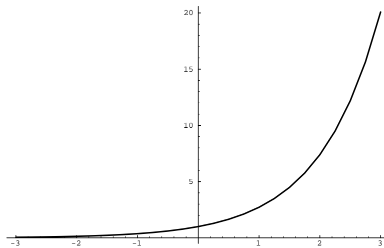
Bemerkung: Bei der Implementierung von intervallwertigen Standardfunktionen nach diesem Schema ist es nicht erforderlich, das Horner–Schema unter Verwendung von ℓ-stelliger

Intervall–Arithmetik durchzuführen. Es reicht vielmehr aus, mit Hilfe entsprechend gerichtet gerundeter Operationen die Unter– bzw. Obergrenze des Ergebnisintervalls zu bestimmen. Die fehlende Intervallgrenze kann daraus über den Gesamtfehler ε_{ges} berechnet werden, da sowohl der Auswertefehler ε_P (a priori) und der Approximationsfehler ε_{app} (wird berechnet) bekannt sind.

Falls doch eine Argumentreduktion erforderlich ist, werden zuvor mit den in Kapitel 1 hergeleiteten Formeln Schranken für die Beträge der relativen Fehler ε_x des Argumentes x und ε_f der Funktionsauswertung von f an der gestörten Stelle $\tilde{x} = x \cdot (1 + \varepsilon_x)$ berechnet. Wenn dies a priori nicht möglich ist, muß die Auswertung zunächst mit einer heuristisch festgelegten Anzahl von Schutzziffern durchgeführt werden. Die ermittelte Schranke für $|\varepsilon_x|$ ist dann bei der Argumentreduktion entsprechend zu berücksichtigen. Auch hierbei kann der Versuch einer a priori Analyse (wie etwa bei den trigonometrischen Funktionen) versagen und eine wiederholte Berechnung erforderlich werden.

Bemerkung: Wie beim Horner–Schema kann auf die Verwendung von Intervall–Arithmetik verzichtet werden, wenn mit Hilfe geeignet gerichteter Rundungen die Unter– oder Obergrenze des Ergebisses berechnet wird. Aus ihr kann die fehlende Intervallgrenze mit Hilfe der vorgegebenen Schranke für den absoluten oder relativen Fehler berechnet werden. Dies gilt natürlich *nicht*, falls eine a priori Analyse nicht möglich ist und ein Wiederberechnungsschritt erforderlich wird. Zu dessen Durchführung werden die Resultate einer intervallmäßigen Auswertung ja explizit benötigt!

2.2.1 Die Exponentialfunktion



Zur Berechnung der Exponentialfunktion wird die für alle $x \in \boldsymbol{R}$ konvergierende Potenzreihe

$$\exp(x) = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

herangezogen. Da für negative Argumente die Gefahr der Auslöschung besteht, wird für sie die Identität

$$\exp(x) = \frac{1}{\exp(-x)} \tag{2.5}$$

verwendet.

Approximationsfehler bei Abbruch nach dem N–ten Glied

Für $x > 0$ gilt bekanntlich folgende Abschätzung des relativen Approximationsfehlers von $1 + x + \ldots + \frac{x^N}{N!}$:

$$|\varepsilon_{app}| \leq \frac{x^{N+1}}{(N+1)!} \quad . \tag{2.6}$$

Für große x führt dies zu unakzeptablen Werten für N, daher ist in der Regel eine Argumentreduktion erforderlich.

Argumentreduktion

Zunächst wird für $x > \ln(b)$ mit Hilfe der Identität

$$\exp(x) = b^{[x/\ln(b)]} \exp(x - \ln(b)[x/\ln(b)]) \tag{2.7}$$

das Argument auf das Intervall $[0, \ln(b))$ reduziert.
Für $b = 10$ ist auch dieses noch unbefriedigend. Falls im verwendeten dynamischen Raster die Division durch Zwei fehlerfrei durchführbar ist, bietet sich die iterierte Anwendung der Identität

$$\exp(x) = \exp(\frac{x}{2})^2 \tag{2.8}$$

zur weiteren Reduktion des Arguments an. Für $b = 10$ und $x \in [0, \ln(b))$ ist dies bis zu viermal erforderlich, um schließlich auf ein in der Praxis akzeptables Argument

$$\tilde{x} \in [0, \frac{\ln(10)}{2^4}] \subseteq [0, 0.14392]$$

zu kommen.

Fehler der Polynomauswertung nach Horner

Der bei der Durchführung des Horner–Schemas entstehende relative Fehler ε_P wird über Lemma 2.1.8 abgeschätzt. Da zusätzlich $a_0 = 1$ exakt darstellbar ist, $a_n x^n \geq 0$ gilt und stets $P(x) \neq 0$ erfüllt ist, kann die folgende Fassung zur Anwendung gebracht werden:

$$\varepsilon_P \in \frac{E_\ell}{P(x)} \left\{ 1 + (2 + 3E_\ell + E_\ell^2)(P(x) - 1) + (1 + 3E_\ell + E_\ell^2) x P'(x) + \frac{E_\ell}{2} x^2 P''(x) \right\} \quad .$$

Unter Ausnutzung der Tatsache, daß $\frac{P'(x)}{P(x)}$ und $\frac{P''(x)}{P(x)}$ jeweils kleiner als Eins sind und $P(x) \in [1, \exp(x))$ gilt, ergibt sich

$$\varepsilon_P \in E_\ell \left\{ 1 + (2 + 3E_\ell + E_\ell^2)(1 - \frac{1}{\exp(x)}) + (1 + 3E_\ell + E_\ell^2) x + \frac{E_\ell}{2} x^2 \right\} \quad .$$

Für $b = 10$, $\ell \geq 5$ und $x \leq 0.14392$ führt dies auf

$$\varepsilon_P \in 1.4121 \cdot E_\ell \quad ,$$

woraus nach (2.3) die Notwendigkeit zweier Schutzziffern (d. h. $\ell = k+2$) folgt. Gemäß (2.2) folgt daraus für den Approximationsfehler

$$|\varepsilon_{app}| \leq 8.5867 \cdot 10^{-(k+1)} \quad .$$

Fehleranalyse der Argumentreduktion

Die Behandlung negativer Argumente gemäß (2.5) erfordert die Betrachtung der Fehlerfortpflanzung bei der Invertierung fehlerbehafteter Größen. Es gilt

$$\frac{1}{x(1+\varepsilon_x)} = \frac{1}{x} \cdot (1 - \frac{\varepsilon_x}{1+\varepsilon_x}) \quad .$$

Da für eine auf ℓ Stellen hochgenaue Auswertung der Exponentialfunktion $2 \cdot b^{1-\ell}$ eine obere Schranke für den Absolutbetrag des relativen Fehlers darstellt, ergibt sich für eine insgesamt auf k Stellen hochgenaue Berechnung nach Division die Forderung

$$\begin{aligned} \frac{2 \cdot b^{1-\ell}}{1 - 2 \cdot b^{1-\ell}} &\leq b^{-k} \\ \ell &\geq k + 1 + \log_b(2 + 2 \cdot b^{-k}) \quad . \end{aligned}$$

Im Fall $b = 10$ besteht somit die Notwendigkeit, zwei zusätzliche signifikante Ziffern von $\exp(-x)$ zu berechnen.
Die eigentliche Reduktion des Arguments nach (2.7) wird gemäß Kapitel 1, (1.14) behandelt, allerdings unter Verwendung relativer Fehler. Bei einer geforderten Genauigkeit von b^{-k} ergeben sich die folgenden Schranken für die Güte der Funktionsauswertung und der Argumentreduktion:

$$\left|\varepsilon_{\exp(\tilde{x})}\right| \leq \frac{\exp(x)}{\exp(\tilde{x})} \cdot \frac{b^{-k}}{2} = \frac{1}{\exp(x \cdot \varepsilon_x)} \cdot \frac{b^{-k}}{2}$$

bzw.

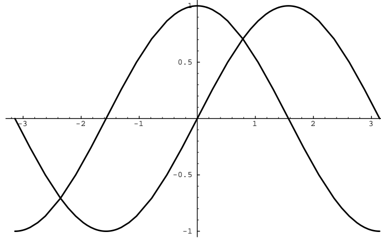
$$|\varepsilon_x| \leq \frac{\exp(x)}{x \cdot \exp(\xi)} \cdot \frac{b^{-k}}{2} \leq \frac{1}{x \cdot \exp(x \cdot |\varepsilon_x|)} \cdot \frac{b^{-k}}{2}$$

für ein $\xi \in \operatorname{conv}\{x, x(1+\varepsilon_x)\}$. Im Fall $b = 10$, $x \in [0, \log(b)]$ und $|\varepsilon_x| \leq 2 \cdot 10^{-4}$, d. h. für ein auf mindestens fünf Stellen hochgenau reduziertes Argument, folgt daraus die Notwendigkeit zweier zusätzlicher signifikanter Ziffern für Argumentreduktion und Funktionsauswertung.
Zur entsprechend genauen Berechnung von $\tilde{x}$ nach (2.7) ist gegebenenfalls ein Wiederberechnungsschritt erforderlich, da eine a priori Betrachtung nicht möglich ist. Mit der Abkürzung $q := \lfloor x/\log(b) \rfloor$ gilt für den relativen Fehler des reduzierten Arguments nach klassischer Fehlerrechnung

$$|\varepsilon_x| = \left|\frac{q \cdot \log(b)}{x - q \cdot \log(b)}\right| \cdot \left|\varepsilon_{\log(b)}\right| \quad .$$

Wird eine weitere Reduktion des Arguments nach (2.8) erforderlich, ist für $b = 10$ die Zahl der signifikanten Stellen der Funktionsauswertung noch einmal um Zwei zu erhöhen, um nach der notwendigen Rücktransformation die ursprünglich geforderte Genauigkeit zu erreichen.

2.2.2 Der Sinus und Kosinus



Sinus und Kosinus werden über die global konvergierenden Potenzreihen

$$\sin(x) = x \cdot \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k}}{(2k+1)!}$$

bzw.

$$\cos(x) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k}}{(2k)!}$$

berechnet. Sie werden hier zusammen betrachtet, da zur Berechnung jeder der beiden Funktionen in bestimmten Fällen auch die Potenzreihe der anderen herangezogen wird.

Approximationsfehler bei Abbruch nach dem N–ten Glied

Für $\sin(x) \neq 0$ gilt im Falle des Sinus folgende Abschätzung des relativen Approximationsfehlers

$$|\varepsilon_{app}| \leq \frac{x^{2N+2}}{(2N+3)!} \left| \frac{x}{\sin(x)} \right| \quad . \tag{2.9}$$

Für den Kosinus ergibt sich unter der entsprechenden Voraussetzung $\cos(x) \neq 0$

$$|\varepsilon_{app}| \leq \frac{x^{2N+2}}{(2N+2)!} \left| \frac{1}{\cos(x)} \right| \quad . \tag{2.10}$$

Beide Abschätzungen zeigen, daß zum Erreichen eines befriedigenden Konvergenzverhaltens eine Argumentreduktion notwendig ist.

Argumentreduktion

Beide Potenzreihen werden in der Praxis nur auf dem Intervall $[-\frac{\pi}{4}, \frac{\pi}{4}]$ ausgewertet. Zur Reduktion des Argumentes x wird zunächst $j := \lfloor \frac{4x}{\pi} \rfloor$ bestimmt und danach $\tilde{x} := x - \lfloor \frac{j+1}{2} \rfloor \frac{\pi}{2}$ berechnet. Die Ermittlung des eigentlichen Funktionswertes erfolgt anschließend durch Auswertung der nach folgender Tabelle auszuwählenden Reihe:

$\lfloor \frac{j+1}{2} \rfloor$ mod 4	$\sin(x)$	$\cos(x)$
0	$\sin(\tilde{x})$	$\cos(\tilde{x})$
1	$\cos(\tilde{x})$	$-\sin(\tilde{x})$
2	$-\sin(\tilde{x})$	$-\cos(\tilde{x})$
3	$-\cos(\tilde{x})$	$\sin(\tilde{x})$

.

Fehleranalyse der Argumentreduktion

Die einzuhaltenden Genauigkeiten bei geforderter k–stelliger hochgenauer Auswertung werden wie bei der Exponentialfunktion gemäß Kapitel 1, (1.14) ermittelt. Es ergeben sich jeweils die folgenden relativen Fehlerschranken:

$$\left|\varepsilon_{\sin(\tilde{x})}\right| \le \left|\frac{\sin(x)}{\sin(\tilde{x})}\right| \cdot \frac{b^{-k}}{2} \qquad \text{und} \qquad |\varepsilon_x| \le \left|\frac{\sin(x)}{x \cdot \cos(\xi)}\right| \cdot \frac{b^{-k}}{2}$$

bzw.

$$\left|\varepsilon_{\cos(\tilde{x})}\right| \le \left|\frac{\cos(x)}{\cos(\tilde{x})}\right| \cdot \frac{b^{-k}}{2} \qquad \text{und} \qquad |\varepsilon_x| \le \left|\frac{\cos(x)}{x \cdot \sin(\xi)}\right| \cdot \frac{b^{-k}}{2} \quad ,$$

jeweils für ein $\xi \in \operatorname{conv}\{x, x \cdot (1 + \varepsilon_x)\}$.
Unter Ausnutzung der Relationen $|\sin(x)| \le |x|$ für $|x| \le \frac{\pi}{4}$ und $|\cos(x)| \le 1$ für alle x sowie für ein $\xi \in \operatorname{conv}\{x, \tilde{x}\}$ ergeben sich folgende Abschätzungen:

$$\begin{aligned}
\left|\frac{\sin(x)}{x \cdot \cos(\xi)}\right| &\ge \left|\frac{\sin(x)}{x}\right| \\
\left|\frac{\sin(x)}{\sin(\tilde{x})}\right| &= \left|\frac{\sin(x)}{\sin(x) + \varepsilon_x \cdot x \cdot \cos(\xi)}\right| \ge \frac{1}{1 + |\varepsilon_x| \left|\frac{x \cdot \cos(\xi)}{\sin(x)}\right|} \ge \frac{1}{1 + |\varepsilon_x| \left|\frac{x}{\sin(x)}\right|} \\
\left|\frac{\cos(x)}{x \cdot \sin(\xi)}\right| &\ge \left|\frac{\cos(x)}{x \cdot \xi}\right| \ge \left|\frac{\cos(x)}{x^2 \cdot (1 + |\varepsilon_x|)}\right| \\
\left|\frac{\cos(x)}{\cos(\tilde{x})}\right| &= \left|\frac{\cos(x)}{\cos(x) - \varepsilon_x \cdot x \cdot \sin(\xi)}\right| \ge \frac{1}{1 + |\varepsilon_x| \left|\frac{x^2 \cdot (1 + |\varepsilon_x|)}{\cos(x)}\right|} \quad .
\end{aligned}$$

Bei der zweiten und vierten Ungleichung wurde der Mittelwertsatz der Differentialrechnung jeweils auf den Nenner der linken Seite angewandt. Zu konkreten Werten gelangt man wie bei der Exponentialfunktion durch eine Minimalforderung an den relativen Fehler ε_x, etwa $|\varepsilon_x| \le 2 \cdot 10^{-4}$ und Ausnutzung der Tatsache, daß $|x| \le \frac{\pi}{4}$ gilt. Für $b = 10$ resultiert daraus in allen Fällen die Notwendigkeit, Funktionsauswertung und Argumentreduktion jeweils auf zwei zusätzliche signifikante Ziffern genau durchzuführen.
Ohne Einschränkungen des Definitionsbereichs oder Voraussetzungen an das verwendete Raster sind keine a priori Aussagen über die für die Argumentreduktion notwendige Anzahl signifikanter Stellen von π möglich. Es wird deshalb zunächst mit einer heuristisch bestimmten Anzahl von Schutzziffern gearbeitet und eine Einschließung

$$\tilde{X} := x - \lfloor \frac{j+1}{2} \rfloor \frac{\pi}{2}$$

für das reduzierte Argument berechnet.
Sollte die dabei erzielte Genauigkeit nicht ausreichen, ist ein Wiederberechnungsschritt erforderlich. Die Bestimmung der notwendigen Zahl signifikanter Stellen von π erfolgt mittels klassischer Fehlerrechnung. Da x und $\lfloor \frac{j+1}{2} \rfloor$ exakt sind, ergibt sich bei fehlerfreier Durchführung der Subtraktion

$$|\varepsilon_x| = \left| \frac{\lfloor \frac{j+1}{2} \rfloor \frac{\pi}{2}}{x - \lfloor \frac{j+1}{2} \rfloor \frac{\pi}{2}} \right| \cdot |\varepsilon_\pi| \leq \left| \frac{\lfloor \frac{j+1}{2} \rfloor \frac{\pi}{2}}{\inf(|\tilde{X}|)} \right| \cdot |\varepsilon_\pi| \quad .$$

Fehler der Polynomauswertung nach Horner

Bei genauerer Betrachtung zeigt sich, daß tatsächlich jeweils Polynome in $u = x^2$ auszuwerten sind. Es sind dies

$$P_{\text{sin}}(u) = \sum_{k=0}^{N} (-1)^k \frac{u^k}{(2k+1)!}$$

bzw.

$$P_{\text{cos}}(u) = \sum_{k=0}^{N} (-1)^k \frac{u^k}{(2k)!} \quad .$$

Bemerkung: Auf die Indizierung von P_{sin} und P_{cos} mit dem Polynomgrad N wurde der Übersichtlichkeit halber verzichtet.

Zur Fehleranalyse wird dementsprechend Lemma 2.1.11 herangezogen. Da zusätzlich $a_0 = 1$ in beiden Fällen exakt darstellbar ist, können die folgenden Fassungen zur Anwendung gebracht werden:

$$\begin{aligned} \varepsilon_{P_{\text{sin}}} \in \; & \frac{E_\ell}{P_{\text{sin}}(u)} \left\{ 1 + (2 + 3E_\ell + E_\ell^2) \sum_{k=1}^{N} \frac{u^k}{(2k+1)!} + (1 + 4E_\ell + 2E_\ell^2) \sum_{k=1}^{N} \frac{k \cdot u^k}{(2k+1)!} \right. \\ & \left. + E_\ell \sum_{k=1}^{N} \frac{k^2 \cdot u^k}{(2k+1)!} \right\} + \frac{E_\ell}{P_{\text{sin}}(u)} \left\{ (1 + 2E_\ell) u P'_{\text{sin}}(u) + E_\ell u^2 P''_{\text{sin}}(u) \right\} \end{aligned}$$

und

$$\begin{aligned} \varepsilon_{P_{\text{cos}}} \in \; & \frac{E_\ell}{P_{\text{cos}}(u)} \left\{ 1 + (2 + 3E_\ell + E_\ell^2) \sum_{k=1}^{N} \frac{u^k}{(2k)!} + (1 + 4E_\ell + 2E_\ell^2) \sum_{k=1}^{N} \frac{k \cdot u^k}{(2k)!} \right. \\ & \left. + E_\ell \sum_{k=1}^{N} \frac{k^2 \cdot u^k}{(2k)!} \right\} + \frac{E_\ell}{P_{\text{cos}}(u)} \left\{ (1 + 2E_\ell) u P'_{\text{cos}}(u) + E_\ell u^2 P''_{\text{cos}}(u) \right\} \quad . \end{aligned}$$

Wegen des alternierenden Verhaltens der Koeffizienten von P_{sin} bzw. P_{cos} erlauben diese Darstellungen keine von Polynomgrad N unabhängige Aussage durch den direkten Grenzübergang $N \to \infty$. Die Absolutbeträge von Leibniz–Reihen können jedoch sehr einfach mit Hilfe der ersten beiden Summanden abgeschätzt werden. Für sie gilt

$$\left| \sum_{k=0}^{\infty} (-1)^k a_k x^k \right| \in \text{conv}\{|a_0 - a_1 x|\, , |a_0|\} \quad .$$

Anschließend kann dann der Grenzübergang $N \to \infty$ erfolgen. Wir führen dies hier nur für $P_{\sin}$ durch, da die entsprechende Betrachtung für $P_{\cos}$ vollkommen analog verläuft.

$$\begin{aligned}
\varepsilon_{P_{\sin}} \in\ & \frac{E_\ell}{P_{\sin}(u)}\left\{1+(2+3E_\ell+E_\ell^2)\sum_{k=1}^{N}\frac{u^k}{(2k+1)!}+(1+4E_\ell+2E_\ell^2)\sum_{k=1}^{N}\frac{k\cdot u^k}{(2k+1)!}\right.\\
& \left.+E_\ell\sum_{k=1}^{N}\frac{k^2\cdot u^k}{(2k+1)!}\right\}+\frac{E_\ell}{P_{\sin}(u)}\left\{(1+2E_\ell)\frac{u}{3!}+E_\ell\frac{2u^2}{5!}\right\}\\
\subseteq\ & \frac{E_\ell}{P_{\sin}(u)}\left\{1+(2+3E_\ell+E_\ell^2)\sum_{k=0}^{N-1}\frac{u^{k+1}}{(2k+3)!}+(1+4E_\ell+2E_\ell^2)\sum_{k=0}^{N-1}(k+1)\frac{u^{k+1}}{(2k+3)!}\right.\\
& \left.+E_\ell\sum_{k=0}^{N-1}(k+1)^2\frac{u^{k+1}}{(2k+3)!}\right\}+\frac{E_\ell}{P_{\sin}(u)}\left\{(1+2E_\ell)\frac{u}{3!}+E_\ell\frac{2u^2}{5!}\right\}\\
\subseteq\ & \frac{E_\ell}{P_{\sin}(u)}\left\{1+(2+3E_\ell+E_\ell^2)\frac{u}{6}\sum_{k=0}^{N-1}\frac{u^k}{(2k+1)!}+(1+4E_\ell+2E_\ell^2)\frac{u}{6}\sum_{k=0}^{N-1}\frac{u^k}{(2k+1)!}\right.\\
& \left.+E_\ell\frac{u}{4}\sum_{k=0}^{N-1}\frac{u^k}{(2k+1)!}\right\}+\frac{E_\ell}{P_{\sin}(u)}\left\{(1+2E_\ell)\frac{u}{3!}+E_\ell\frac{2u^2}{5!}\right\}\\
\subseteq\ & \frac{E_\ell}{1-\frac{u}{3!}}\left\{1+(6+17E_\ell+6E_\ell^2)\frac{u}{12}\frac{\sinh(x)}{x}+(1+2E_\ell)\frac{u}{3!}+E_\ell\frac{2u^2}{5!}\right\}\\
\subseteq\ & \frac{E_\ell}{1-\frac{u}{3!}}\left\{1+(6+17E_\ell+6E_\ell^2)\frac{x\sinh(x)}{12}+(1+2E_\ell)\frac{u}{3!}+E_\ell\frac{2u^2}{5!}\right\}
\end{aligned}$$

Die entsprechende Rechnung für $P_{\cos}$ ergibt

$$\varepsilon_{P_{\cos}} \in \frac{E_\ell}{1-\frac{u}{2!}}\left\{1+(4+13E_\ell+5E_\ell^2)\frac{u\cosh(x)}{2}+(1+2E_\ell)\frac{u}{2!}+E_\ell\frac{2u^2}{4!}\right\} \quad .$$

Für $b=10$, $\ell \geq 5$, $|x| \leq \frac{\pi}{4}$, d.h. $0 < u \leq \frac{\pi^2}{16}$ ergibt sich insgesamt

$$\varepsilon_{P_{\sin}} \in 1.6095 \cdot E_\ell \qquad \text{bzw.} \qquad \varepsilon_{P_{\cos}} \in 4.2551 \cdot E_\ell \quad .$$

In beiden Fällen muß das Horner–Schema also mit zwei Schutzziffern arbeiten. Gemäß (2.2) folgt daraus für die Approximationsfehler

$$|\varepsilon_{app}| \leq 8.3892 \cdot 10^{-(k+1)} \qquad \text{bzw.} \qquad |\varepsilon_{app}| \leq 5.7425 \cdot 10^{-(k+1)} \quad .$$

Für kleinere Basen wie etwa $b=2$ ist gemäß dieser Abschätzung die Berechnung des Kosinus über die zugehörige Potenzreihe sehr ungünstig, insbesondere wenn an eine hardwaremäßige Implementierung gedacht ist. Es wären *vier* Schutzbits erforderlich. In der Praxis wird für kleinere Basen deshalb die Identität $\cos(x) = \sin(x+\frac{\pi}{2})$ verwendet.

Die zugehörigen Intervallfunktionen

Die Berechnung von Sinus und Kosinus für reelle Intervallargumente X erfordert einige Fallunterscheidungen (vgl. [Br], S. 85f). Hierbei gehen im wesentlichen die bei der Argumentreduktion ohnehin zu ermittelnden ganzzahligen Anteile $\lfloor \frac{4x}{\pi} \rfloor$ für $x \in \{\inf(X), \sup(X)\}$ ein.
Mit den Bezeichnungen $j_1 := \lfloor \frac{2\sup(X)}{\pi} \rfloor$ bzw. $j_2 := \lfloor \frac{2\inf(X)}{\pi} \rfloor$ und $d := j_2 - j_1$ ergeben sich die folgenden Fälle.
Für $d = 0$ liegen Unter- und Obergrenze des Argumentes im gleichen Intervall modulo $\frac{\pi}{2}$. Innerhalb dieser sind Sinus bzw. Kosinus jeweils streng monoton fallend oder steigend. Es ergibt sich jeweils

$$\sin(X) = \begin{cases} [\sin(\inf(X)), \sin(\sup(X))] & , \text{ für } \; j_1 \bmod 4 \in \{0,3\} \\ [\sin(\sup(X)), \sin(\inf(X))] & , \text{ für } \; j_1 \bmod 4 \in \{1,2\} \end{cases}$$

bzw.

$$\cos(X) = \begin{cases} [\cos(\inf(X)), \cos(\sup(X))] & , \text{ für } \; j_1 \bmod 4 \in \{2,3\} \\ [\cos(\sup(X)), \cos(\inf(X))] & , \text{ für } \; j_1 \bmod 4 \in \{0,1\} \end{cases} .$$

Für $d > 3$ ist beim Intervallsinus und -kosinus das Ergebnis stets $[-1, 1]$.
Für $0 < d \leq 3$ gibt mit den Definitionen

$$y_{\min} := \min\{\sin(\inf(X)), \sin(\sup(X))\} \quad \text{bzw.} \quad y_{\max} := \max\{\sin(\inf(X)), \sin(\sup(X))\}$$

die folgende Tabelle Auskunft über das Resultat des Intervallsinus:

$j_1 \bmod 4$	$d \bmod 4 = 1$	$d \bmod 4 = 2$	$d \bmod 4 = 3$
0	$[y_{\min}, 1]$	$[y_{\min}, 1]$	$[-1, 1]$
1	$[y_{\min}, y_{\max}]$	$[-1, y_{\max}]$	$[-1, y_{\max}]$
2	$[-1, y_{\max}]$	$[-1, y_{\max}]$	$[-1, 1]$
3	$[y_{\min}, y_{\max}]$	$[y_{\min}, 1]$	$[y_{\min}, 1]$

.

Bis auf Rotation um eine Zeile (bedingt durch die bereits erwähnte Identität $\cos(x) = \sin(x + \frac{\pi}{2})$) ergibt sich mit den entsprechenden Definitionen

$$y_{\min} := \min\{\cos(\inf(X)), \cos(\sup(X))\}$$

bzw.

$$y_{\max} := \max\{\cos(\inf(X)), \cos(\sup(X))\}$$

folgendes Bild für den Intervallkosinus:

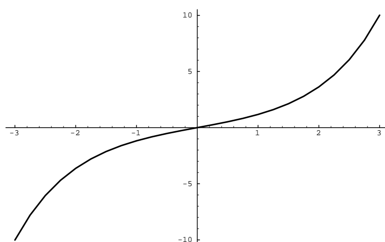
$j_1 \bmod 4$	$d \bmod 4 = 1$	$d \bmod 4 = 2$	$d \bmod 4 = 3$
0	$[y_{\min}, y_{\max}]$	$[-1, y_{\max}]$	$[-1, y_{\max}]$
1	$[-1, y_{\max}]$	$[-1, y_{\max}]$	$[-1, 1]$
2	$[y_{\min}, y_{\max}]$	$[y_{\min}, 1]$	$[y_{\min}, 1]$
3	$[y_{\min}, 1]$	$[y_{\min}, 1]$	$[-1, 1]$

.

Bemerkungen zur Implementierung

Es ist für die Laufzeit eines nach obigen Vorschriften implementierten Intervallsinus (oder -kosinus) sehr ungünstig, die Bestimmung der Werte für j_1 und j_2 unabhängig von der Berechnung (und damit auch der Argumentreduktion) der entsprechenden Punktroutinen durchzuführen. Die Fallunterscheidungen werden zwar etwas aufgebläht, dafür können aber direkt die entsprechenden Routinen für Argumente in $[-\frac{\pi}{4}, \frac{\pi}{4}]$ aufgerufen werden.

2.2.3 Der hyperbolische Sinus



Der hyperbolische Sinus wird für kleine Argumente über die zugehörige Potenzreihe

$$\sinh(x) = x \cdot \sum_{k=0}^{\infty} \frac{x^{2k}}{(2k+1)!}$$

berechnet.

Approximationsfehler bei Abbruch nach dem N–ten Glied

Für $x > 0$ gilt nach ([Br], S. 97) folgende Abschätzung des relativen Approximationsfehlers:

$$|\varepsilon_{app}| \leq \frac{x^{2N+2}}{(2N+3)!} \cosh(x) \quad . \tag{2.11}$$

Für große Argumente führt dies zu unakzeptablen Werten für N. Deshalb wird in diesen Fällen die Identität

$$\sinh(x) = \frac{e^x - e^{-x}}{2} \tag{2.12}$$

zur Berechnung verwendet, welche wiederum für kleine Argumente aufgrund von Auslöschung nicht geeignet ist. Wir verwenden hier wegen des Zusammenhangs mit der Argumentreduktion bei der Exponentialfunktion als Schranke den Wert $\bar{x} = \frac{\ln(10)}{4} \approx 0.57564$.

Fehler der Polynomauswertung nach Horner

Bei genauerer Betrachtung zeigt sich, daß tatsächlich das folgende Polynom in $u = x^2$ auszuwerten ist:

$$P(u) = \sum_{k=0}^{N} \frac{u^k}{(2k+1)!} \quad .$$

Zur Fehleranalyse wird dementsprechend Lemma 2.1.11 herangezogen. Da zusätzlich $a_0 = 1$ exakt darstellbar ist und $a_n u^n \geq 0$ gilt, kann die folgende Fassung zur Anwendung gebracht werden:

$$\varepsilon_P \in \frac{E_\ell}{P(u)} \left\{ 1 + (2 + 3E_\ell + E_\ell^2)(P(u) - 1) + (2 + 7E_\ell + 2E_\ell^2) u P'(u) + 2E_\ell u^2 P''(u) \right\} \quad .$$

Hierzu müssen die Ausdrücke $u\frac{P'(u)}{P(u)}$ bzw. $u^2 \frac{P''(u)}{P(u)}$ nach oben abgeschätzt werden. Nach ([Br], S. 98) gilt

$$u\frac{P'(u)}{P(u)} \leq \frac{u}{6} \quad .$$

Eine entsprechende Rechnung für den zweiten Ausdruck führt auf

$$u^2 \frac{P''(u)}{P(u)} \leq \frac{u}{4} \quad .$$

Einsetzen dieser Abschätzungen in die ursprüngliche Beziehung ergibt

$$\varepsilon_P \in E_\ell \left\{ 1 + (2 + 3E_\ell + E_\ell^2)(1 - \frac{x}{\sinh(x)}) + (2 + 7E_\ell + 2E_\ell^2)\frac{u}{6} + E_\ell \frac{u}{2} \right\} \quad .$$

Für $b = 10$, $\ell \geq 5$, $x \leq 0.57564$ ergibt sich

$$\varepsilon_P \in 1.2168 E_\ell \quad .$$

Aus (2.3) folgt somit die Notwendigkeit zweier Schutzziffern, also $\ell = k+2$. Dies führt gemäß (2.2) auf

$$|\varepsilon_{app}| \leq 8.7821 \cdot 10^{-(k+1)} \quad .$$

Fehleranalyse bei Verwendung der Exponentialfunktion

Klassische Fehlerrechnung führt, unter der Voraussetzung, daß die Division durch Zwei fehlerfrei durchführbar ist, auf

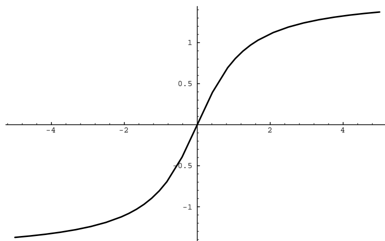
$$\begin{aligned}
|\varepsilon_{\sinh(x)}| &= \left| \frac{e^x}{2\sinh(x)} \varepsilon_{\exp(x)} - \frac{e^{-x}}{2\sinh(x)} \left(\varepsilon_/ - \frac{\varepsilon_{\exp(x)}}{1 + \varepsilon_{\exp(x)}} (1 + \varepsilon_/) \right) \right| \\
&\leq \left| \frac{\varepsilon}{2\sinh(x)} \left(2\cosh(x) + e^{-x} \frac{1+\varepsilon}{1-\varepsilon} \right) \right| \quad , \text{mit } \varepsilon := \max(|\varepsilon_{\exp(x)}|, |\varepsilon_/|) \\
&\leq \varepsilon \left| \coth(x) + \frac{e^{-x}}{2\sinh(x)} \frac{(1+\varepsilon)}{(1-\varepsilon)} \right| \quad .
\end{aligned}$$

Für $\varepsilon \leq 2 \cdot 10^{-4}$ ergibt sich

$$|\varepsilon_{\sinh(x)}| \leq 2.3876 \cdot \varepsilon \quad .$$

Dies führt für $b = 10$ zur Notwendigkeit zweier Schutzziffern.

2.2.4 Der Arcustangens



Die Umkehrfunktion des Tangens kann für Argumente x mit $|x| \leq 1$ über die Potenzreihe

$$\arctan(x) = x \cdot \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k}}{2k+1}$$

berechnet werden.

Approximationsfehler bei Abbruch nach dem N–ten Glied

Nach ([Kr], S. 50) gilt folgende Abschätzung des relativen Approximationsfehlers:

$$|\varepsilon_{app}| \leq \frac{x^{2N+2}}{2N+3} \cdot \frac{1}{1-x^2} \cdot \frac{1}{1-\frac{1}{3} \cdot x^2} \quad . \tag{2.13}$$

Für große Argumente führt dies zu unakzeptablen Werten für N. Eine Argumentreduktion ist deshalb erforderlich.

Argumentreduktion

Zunächst reicht es o.B.d.A. aus, nur positive Argumente zu betrachten, da $\arctan(-x) = -\arctan(x)$ gilt. Zur Argumentreduktion können dann folgende Identitäten herangezogen werden:

$$\arctan(x) = \begin{cases} 2\arctan(\frac{x}{1+\sqrt{1+x^2}}) & , \text{ für } \quad \bar{x} \leq x \leq 1 \\ \frac{\pi}{2} - \arctan(\frac{1}{x}) & , \text{ für } \quad x > 1 \end{cases} \quad .$$

Die erste der beiden Identitäten legt für $\bar{x}$ den Wert $\frac{1}{1+\sqrt{2}}$ nahe. Bei einer kleineren Schranke wäre die entsprechende Reduktion eventuell mehrfach durchzuführen (vgl. [Kr], S. 52). Der erhöhte Aufwand hierfür steht in keinem Verhältnis zur Erhöhung des Grades des Approximationspolynoms. Für Werte kleiner als $\bar{x}$ wird die Reihenentwicklung verwendet.

Fehleranalyse der Argumentreduktion

Der Gesamtfehler der Reduktion von Argumenten $x > 1$ ergibt sich nach klassischer Fehlerrechnung als

$$\begin{aligned}\varepsilon_{\arctan(x)} &= \frac{\frac{\pi}{2}}{\frac{\pi}{2} - \arctan(\frac{1}{x})} \cdot \varepsilon_{\frac{\pi}{2}} - \frac{\arctan(\frac{1}{x})}{\frac{\pi}{2} - \arctan(\frac{1}{x})} \cdot \varepsilon_{\arctan(\frac{1}{x})} \\ &= \frac{\frac{\pi}{2}}{\arctan(x)} \cdot \varepsilon_{\frac{\pi}{2}} - \frac{\arctan(\frac{1}{x})}{\arctan(x)} \cdot \varepsilon_{\arctan(\frac{1}{x})} \quad .\end{aligned}$$

Für die auftretenden Verstärkungsfaktoren von $\varepsilon_{\frac{\pi}{2}}$ und $\varepsilon_{\arctan(\frac{1}{x})}$ sind im Fall $x > 1$ die folgenden Abschätzungen gültig:

$$\left|\frac{\frac{\pi}{2}}{\frac{\pi}{2} - \arctan(\frac{1}{x})}\right| \leq 2 \qquad \text{bzw.} \qquad \left|\frac{\arctan(\frac{1}{x})}{\frac{\pi}{2} - \arctan(\frac{1}{x})}\right| \leq 1 \quad .$$

Durch die Forderungen

$$|\varepsilon_{\frac{\pi}{2}}| \leq \frac{\varepsilon_{\arctan(x)}}{4} \qquad \text{bzw.} \qquad |\varepsilon_{\arctan(\frac{1}{x})}| \leq \frac{\varepsilon_{\arctan(x)}}{2} \tag{2.14}$$

wird die vorgegebene Fehlerschranke $\varepsilon_{\arctan(x)}$ also nicht überschritten. Bei angestrebter k–stelliger hoher Genauigkeit folgt, daß $\frac{\pi}{2}$ auf $k + 1 + \log_b(8)$ Stellen genau zu berechnen ist. Für $b = 10$ bedeutet dies, daß zwei zusätzliche signifikante Ziffern zu bestimmen sind.
Die Fehleranalyse für den zweiten beteiligten Summanden wird nach Kapitel 1, (1.14) behandelt. Nach Einführung relativer Fehler ergibt sich mit der Bezeichnung $y := \frac{1}{x}$

$$\left|\varepsilon_{\arctan(\tilde{y})}\right| \leq \left|\frac{\arctan(y)}{\arctan(\tilde{y})}\right| \cdot \frac{\varepsilon_{\arctan(y)}}{2} \qquad \text{bzw.} \qquad |\varepsilon_y| \leq \left|\frac{\arctan(y)}{y \cdot \frac{1}{1+\xi^2}}\right| \cdot \frac{\varepsilon_{\arctan(y)}}{2}$$

für ein $\xi \in \operatorname{conv}\{y, \tilde{y}\}$. Die Anwendung des Mittelwertsatzes der Differentialrechnung auf den Nenner der rechten Seite der ersten Ungleichung führt mit der Abkürzung

$$\kappa := \frac{y}{\arctan(y)} \cdot \frac{1}{1 + \xi^2}$$

auf

$$\left|\varepsilon_{\arctan(\tilde{y})}\right| \leq \left|\frac{1}{1 + \varepsilon_y \cdot \kappa}\right| \cdot \frac{\varepsilon_{\arctan(y)}}{2} \qquad \text{bzw.} \qquad |\varepsilon_y| \leq \left|\frac{1}{\kappa}\right| \cdot \frac{\varepsilon_{\arctan(y)}}{2} \quad .$$

Die obere Schranke für $|\varepsilon_y|$ kann zur Vereinfachung der Schranke für $\left|\varepsilon_{\arctan(\tilde{y})}\right|$ herangezogen werden. Es ergibt sich

$$\left|\varepsilon_{\arctan(\tilde{y})}\right| \leq \left|\frac{1}{1 + \varepsilon_y \cdot \kappa}\right| \cdot \frac{\varepsilon_{\arctan(y)}}{2} \leq \left|\frac{1}{1 + \left|\frac{1}{\kappa}\right| \cdot \frac{\varepsilon_{\arctan(y)}}{2} \cdot |\kappa|}\right| \cdot \frac{\varepsilon_{\arctan(y)}}{2} = \frac{\varepsilon_{\arctan(y)}}{2 + \varepsilon_{\arctan(y)}} \quad .$$

Für $x > 1$ gilt nun $|\kappa| \leq \frac{4}{\pi}$. Insgesamt ergibt sich somit unter Berücksichtigung von (2.14)

$$\left|\varepsilon_{\arctan(\tilde{y})}\right| \leq \frac{b^{-k}}{4 + b^{-k}} \qquad \text{bzw.} \qquad |\varepsilon_y| \leq \frac{\pi}{16} \cdot b^{-k}$$

und daraus für $b = 10$ die Notwendigkeit von zwei zusätzlichen signifikanten Ziffern bei der Berechnung von Argument und Funktionswert.
Der verbleibende Fall von Argumenten x im Intervall $[\frac{1}{1+\sqrt{2}}, 1]$ wird unter der Voraussetzung, daß die Multiplikation mit Zwei fehlerfrei durchführbar ist, ebenfalls nach Kapitel 1, (1.14) behandelt. Mit der Abkürzung $y := \frac{x}{1+\sqrt{1+x^2}}$ und κ wie oben gilt nach Einführung relativer Fehler

$$|\varepsilon_y| \leq \left|\frac{1}{\kappa}\right| \cdot \frac{\varepsilon_{\text{arctan}(y)}}{2} \qquad \text{bzw.} \qquad \left|\varepsilon_{\text{arctan}(\tilde{y})}\right| \leq \frac{\varepsilon_{\text{arctan}(y)}}{2 + \varepsilon_{\text{arctan}(y)}} \quad .$$

In dem zu betrachtenden Intervall kann $|\kappa|$ grob durch Eins nach oben abgeschätzt werden. Dies führt auf

$$\left|\varepsilon_{\text{arctan}(\tilde{y})}\right| \leq \frac{b^{-k}}{2 + b^{-k}} \qquad \text{bzw.} \qquad |\varepsilon_y| \leq \frac{b^{-k}}{2} \quad .$$

Mit der weiteren Abkürzung $z := 1 + \sqrt{1+x^2}$ führt klassische Fehlerrechnung auf die Forderung

$$\frac{|\varepsilon_z|}{1 - |\varepsilon_z|} \cdot (1 + |\varepsilon_/|) + |\varepsilon_/| \leq |\varepsilon_y|$$

und mit $\varepsilon := max\{|\varepsilon_z|, |\varepsilon_/|\}$ folgt hieraus zusammen mit der zuvor hergeleiteten Schranke für $|\varepsilon_y|$

$$\varepsilon \leq \frac{b^{-k}}{4 + b^{-k}} \quad .$$

Unter der zusätzlichen Voraussetzung, daß die beiden bei der Berechnung von z erforderlichen Additionen und das Quadrieren von x fehlerfrei durchgeführt werden, sind bei der Auswertung der Quadratwurzel keine weiteren Schutzziffern erforderlich. Dies resultiert aus der Tatsache, daß der Verstärkungsfaktor von $\varepsilon_{\sqrt{1+x^2}}$ betragsmäßig durch Eins beschränkt ist. Die obige Voraussetzung ist in dem betrachteten Intervall durchaus legitim, da alle beteiligten Operanden die gleiche Größenordnung besitzen und keine aufwendigen Exponentenangleichungen erforderlich sind.
Für $b = 10$ und $k \geq 5$ folgt aus den vorangehenden Abschätzungen die durchgängige Notwendigkeit zweier Schutzziffern.

Fehler der Polynomauswertung nach Horner

Bei genauerer Betrachtung zeigt sich, daß tatsächlich ein Polynom in $u = x^2$ auszuwerten ist. Es hat die Form

$$P(u) = \sum_{k=0}^{N} (-1)^k \frac{u^k}{2k+1} \quad .$$

Zur Fehleranalyse wird dementsprechend Lemma 2.1.11 herangezogen. Da zusätzlich $a_0 = 1$ exakt darstellbar ist, kann die folgende Fassung zur Anwendung gebracht werden:

$$\begin{aligned} \varepsilon_P \quad \in \quad & \frac{E_\ell}{P(u)} \left\{ 1 + (2 + 3E_\ell + E_\ell^2) \sum_{k=1}^{N} \frac{u^k}{2k+1} + (1 + 4E_\ell + 2E_\ell^2) \sum_{k=1}^{N} \frac{k \cdot u^k}{2k+1} \right. \\ & \left. + E_\ell \sum_{k=1}^{N} \frac{k^2 \cdot u^k}{2k+1} \right\} + \frac{E_\ell}{P(u)} \left\{ (1 + 2E_\ell) u P'(u) + E_\ell u^2 P''(u) \right\} \quad . \end{aligned}$$

Wie bei Sinus und Kosinus ist wegen des alternierenden Verhaltens der Koeffizienten von P sowie dessen erster und zweiten Ableitung keine von Polynomgrad N unabhängige Aussage durch den direkten Grenzübergang $N \to \infty$ möglich. Dieses Problem wird auf analoge Weise gelöst. Es tritt jedoch das zusätzliche Problem auf, daß die in der ersten geschweiften Klammer auftretenden Summen nicht ebenso einfach wie beim Sinus bzw. Kosinus durch alleinige Indexverschiebung abgeschätzt werden können. Erst die in unserem konkreten Fall erfüllte Bedingung $u = x^2 < 1$ erlaubt einen Zugang über die geometrische Reihe (vgl. [Kr], S. 26, (1.3.6)) und damit schließlich eine vom Polynomgrad unabhängige Aussage. Für die einzelnen Summen ergibt sich:

$$\begin{aligned}
\sum_{k=1}^{N} \frac{u^k}{2k+1} &= \sum_{k=0}^{N-1} \frac{u^{k+1}}{2k+3} = \sum_{k=0}^{N-1} \frac{2k+1}{2k+3} \cdot \frac{u^{k+1}}{2k+1} \leq u \cdot \frac{\operatorname{artanh}(x)}{x} \\
\sum_{k=1}^{N} \frac{k \cdot u^k}{2k+1} &= \sum_{k=0}^{N-1} \frac{(k+1) \cdot u^{k+1}}{2k+3} \leq \frac{u}{2} \sum_{k=0}^{N-1} u^k \leq \frac{u}{2} \cdot \frac{1}{1-u} \quad .
\end{aligned}$$

Zur Behandlung der dritten Summe ist ein weiterer Kniff erforderlich. Wir nutzen aus, daß für $u = x^2 < 1$

$$\frac{\partial}{\partial u}\left(\frac{1}{1-u}\right) = \frac{1}{(1-u)^2} = \sum_{k=1}^{\infty} k u^{k-1}$$

gilt, wobei die letzte Reihe durch gliedweises Ableiten der geometrischen Reihe erhalten wird. Hiermit läßt sich nun auch die verbleibende Summe behandeln:

$$\begin{aligned}
\sum_{k=1}^{N} \frac{k^2 \cdot u^k}{2k+1} &= \sum_{k=0}^{N-1} \frac{(k+1)^2 \cdot u^{k+1}}{2k+3} \leq \frac{u}{2}\left(\sum_{k=0}^{N-1} u^k + u \sum_{k=0}^{N-1} u^{k-1}\right) \\
&\leq \frac{u}{2}\left(\frac{1}{1-u} + \frac{u}{(1-u)^2}\right) = \frac{u}{2}\frac{1}{(1-u)^2} \quad .
\end{aligned}$$

Einsetzen all dieser Abschätzungen in die ursprüngliche Beziehung führt auf

$$\begin{aligned}
\varepsilon_P \in \; & \frac{E_\ell}{1-\frac{u}{3}}\Big\{1 + (2 + 3E_\ell + E_\ell^2)x \cdot \operatorname{artanh}(x) + (1 + 4E_\ell + 2E_\ell^2)\frac{u}{2} \cdot \frac{1}{1-u} \\
& + E_\ell \frac{u}{2}\frac{1}{(1-u)^2} + (1 + 2E_\ell)\frac{u}{3} + E_\ell \frac{2u^2}{5}\Big\} \quad .
\end{aligned}$$

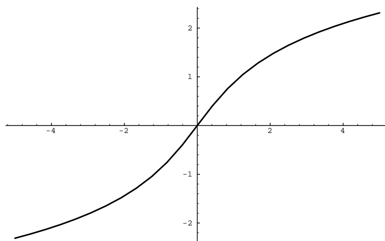
Für $b = 10$, $\ell \geq 5$ und $u = x^2 \leq \frac{1}{3+2\sqrt{2}}$ ergibt sich insgesamt

$$\varepsilon_P \in 1.6184 \cdot E_\ell$$

und damit die Notwendigkeit zweier Schutzziffern bei Forderung k–stelliger hoher Genauigkeit. Gemäß (2.2) folgt daraus für den Approximationsfehler

$$|\varepsilon_{app}| \leq 8.3802 \cdot 10^{-(k+1)} \quad .$$

2.2.5 Der Areasinus



Die Umkehrfunktion des hyperbolischen Sinus kann für Argumente x mit $|x| < 1$ über die Potenzreihe

$$\operatorname{arsinh}(x) = x \cdot \sum_{k=0}^{\infty} c_k \cdot x^{2k} \quad \text{mit} \quad c_k := \begin{cases} 1 & , \text{ für } \quad k = 0 \\ -c_{k-1} \frac{(2k-1)^2}{2k(2k+1)} & , \text{ für } \quad k > 0 \end{cases}$$

berechnet werden. In der Praxis wird die Reihe nur auf einem Intervall $(0, \bar{x}]$ mit geeignet gewählter Obergrenze $0 < \bar{x} \ll 1$ ausgewertet.
Für die verbleibenden Fälle werden die Identitäten

$$\operatorname{arsinh}(x) = \begin{cases} -\operatorname{arsinh}(-x) & , \text{ für } \quad x < 0 \\ 0 & , \text{ für } \quad x = 0 \\ \ln\left(x + \sqrt{x^2 + 1}\right) & , \text{ für } \quad x > \bar{x} \end{cases}$$

verwendet.

Approximationsfehler bei Abbruch nach dem N–ten Glied

Für $x > 0$ gilt nach ([Kr], S. 60 [1]) folgende Abschätzung des relativen Approximationsfehlers

$$|\varepsilon_{app}| \leq \frac{|c_{N+1}| x^{2N+2}}{1 - |c_1| x^2} = \frac{|c_{N+1}| x^{2N+2}}{1 - \frac{1}{6} x^2} \quad . \tag{2.15}$$

Fehleranalyse für $x > \bar{x}$

Die Fehleranalyse für Argumente $x > \bar{x}$ wird wie beim Arcustangens nach Kapitel 1, (1.14) behandelt. Mit den Abkürzungen $y := x + \sqrt{x^2 + 1}$ und $\kappa := \frac{y}{\ln(y)} \cdot \frac{1}{\xi}, \quad \xi \in \operatorname{conv}\{y, \tilde{y}\}$ ergibt sich wie beim Arcustangens nach Einführen relativer Fehler

$$\left|\varepsilon_{\ln(\tilde{y})}\right| \leq \frac{\varepsilon_{\ln(y)}}{2 + \varepsilon_{\ln(y)}} \qquad \text{bzw.} \qquad |\varepsilon_y| \leq \left|\frac{1}{\kappa}\right| \cdot \frac{\varepsilon_{\ln(y)}}{2} \quad .$$

[1] In der Originalarbeit befindet sich allerdings ein offensichtlicher Tippfehler.

Der Term $\frac{y}{\ln(y)}$ ist jedoch für $x \to \infty$ nach oben unbeschränkt. Hier kann die folgende Überlegung Abhilfe schaffen. Es gilt für $\xi \in \text{conv}\{y, \tilde{y} = y(1+\varepsilon_y)\}$, $y > 0$ und $|\varepsilon_y| < 1$

$$\frac{1}{\xi} \leq \frac{1}{y(1-|\varepsilon_y|)} = \frac{1}{(x+\sqrt{x^2+1})\cdot(1-|\varepsilon_y|)} \quad .$$

Hiermit ergibt sich nun folgende obere Schranke für κ:

$$\kappa \leq \frac{x+\sqrt{x^2+1}}{\ln(x+\sqrt{x^2+1})} \cdot \frac{1}{(x+\sqrt{x^2+1})\cdot(1-|\varepsilon_y|)} = \frac{1}{\ln(x+\sqrt{x^2+1})\cdot(1-|\varepsilon_y|)} \quad .$$

Mit einer Minimalforderung an ε_y, etwa $|\varepsilon_y| \leq 2 \cdot b^{-4}$ und damit einer auf mindestens fünf Stellen hochgenauen Berechnung des Argumentes des natürlichen Logarithmus, läßt sich aufgrund des Monotonieverhaltens des Logarithmus und der Tatsache, daß

$$x+\sqrt{x^2+1} \geq \bar{x}+\sqrt{\bar{x}^2+1} > 1$$

gilt, folgende Schranke für κ angeben:

$$\kappa \leq \frac{1}{\ln(\bar{x}+\sqrt{\bar{x}^2+1})\cdot(1-2\cdot b^{-4})} \quad .$$

Für $b = 10$ und $\bar{x} = \frac{1}{8}$ führt dies auf $\kappa \leq 8.0224$, woraus wiederum die Notwendigkeit dreier zusätzlicher signifikanter Ziffern bei der Berechnung des Argumentes des natürlichen Logarithmus folgt. Für die Auswertung des Logarithmus selbst reichen zwei Schutzziffern. Unter der Voraussetzung, daß die Addition von x und $\sqrt{x^2+1}$ exakt durchgeführt wird, ergibt die weitere Analyse unter Verwendung der Abkürzung $z = x^2+1$

$$\left|\varepsilon_{\sqrt{z}}\right| \leq \left|\frac{x+\sqrt{x^2+1}}{\sqrt{x^2+1}}\right| \cdot |\varepsilon_y| \quad .$$

Der Koeffizient von $|\varepsilon_y|$ ist monoton steigend in x und für $x \geq \frac{1}{8}$ durch 1.124 nach unten beschränkt. Die bei der Berechnung von z und der anschließenden Auswertung der Wurzelfunktion einzuhaltenden Genauigkeiten ergeben sich dann zu

$$|\varepsilon_z| \leq \frac{\sqrt{z}}{z \cdot \frac{1}{2\sqrt{\xi_z}}} \cdot \frac{\left|\varepsilon_{\sqrt{z}}\right|}{2} = \sqrt{\frac{\xi_z}{z}} \cdot \left|\varepsilon_{\sqrt{z}}\right|$$

für $\xi_z \in \text{conv}\{z, \tilde{z}\}$ bzw.

$$\left|\varepsilon_{\sqrt{\tilde{z}}}\right| \leq \frac{\left|\varepsilon_{\sqrt{z}}\right|}{2+\left|\varepsilon_{\sqrt{z}}\right|} \quad .$$

Aus der ersten Ungleichung erhält man wegen der Monotonie der Quadratwurzel

$$|\varepsilon_z| \leq \sqrt{(1-|\varepsilon_z|)} \cdot \left|\varepsilon_{\sqrt{z}}\right| \quad .$$

Mit einer der an $|\varepsilon_y|$ gestellten Mindestanforderung entsprechenden für $|\varepsilon_z|$ ergeben sich im vorliegenden Wertebereich die folgenden konkreten Schranken:

$$\left|\varepsilon_{\sqrt{z}}\right| \leq 1.124 \cdot |\varepsilon_y| \quad , \quad |\varepsilon_z| \leq 1.1238 \cdot |\varepsilon_y| \quad \text{und} \quad \left|\varepsilon_{\sqrt{\tilde{z}}}\right| \leq 0.56193 \cdot |\varepsilon_y| \quad .$$

Zieht man nun zusätzlich die zu Beginn hergeleitete Schranke für $|\varepsilon_y|$ ins Kalkül erhält man insgesamt

$$\left|\varepsilon_{\sqrt{z}}\right| \leq 0.070053 \cdot |\varepsilon_{\ln(y)}| \quad , \quad |\varepsilon_z| \leq 0.070041 \cdot \left|\varepsilon_{\ln(y)}\right| \quad \text{und} \quad \left|\varepsilon_{\sqrt{\tilde{z}}}\right| \leq 0.035022 \cdot \left|\varepsilon_{\ln(y)}\right| \quad .$$

Die Berechnung von z und das anschließende Ziehen der Quadratwurzel müssen also für $b = 10$ mit drei zusätzlichen signifikanten Ziffern erfolgen.

Fehler der Polynomauswertung nach Horner

Bei genauerer Betrachtung zeigt sich, daß tatsächlich ein Polynom in $u = x^2$ auszuwerten ist. Es hat die Form

$$P(u) = \sum_{k=0}^{N} c_k \cdot u^k \quad .$$

Zur Fehleranalyse wird dementsprechend Lemma 2.1.11 herangezogen. Da zusätzlich $a_0 = 1$ exakt darstellbar ist, kann die folgende Fassung zur Anwendung gebracht werden:

$$\begin{aligned} \varepsilon_P \in \; & \frac{E_\ell}{P(u)} \left\{ 1 + (2 + 3E_\ell + E_\ell^2) \sum_{k=1}^{N} |c_k| \cdot u^k + (1 + 4E_\ell + 2E_\ell^2) \sum_{k=1}^{N} k \cdot |c_k| \cdot u^k \right. \\ & \left. + E_\ell \sum_{k=1}^{N} k^2 \cdot |c_k| \cdot u^k \right\} + \frac{E_\ell}{P(u)} \left\{ (1 + 2E_\ell) u P'(u) + E_\ell u^2 P''(u) \right\} \quad . \end{aligned}$$

Zur Abschätzung der vorkommenden Summen verfahren wir wie beim Arcustangens und erhalten die folgenden vom Polynomgrad N unabhängigen Aussagen:

$$\begin{aligned} \sum_{k=1}^{N} |c_k| \cdot u^k &= \sum_{k=0}^{N-1} |c_{k+1}| \cdot u^{k+1} = \sum_{k=0}^{N-1} \frac{(2k+1)^2}{2(k+1)(2k+3)} |c_k| \cdot u^{k+1} \\ &\leq u \sum_{k=0}^{N-1} |c_k| \cdot u^k \leq \frac{u}{1-u} \end{aligned}$$

sowie

$$\begin{aligned} \sum_{k=1}^{N} k \cdot |c_k| \cdot u^k &= \sum_{k=0}^{N-1} (k+1) \cdot |c_{k+1}| \cdot u^{k+1} = \sum_{k=0}^{N-1} \frac{(2k+1)^2}{2(2k+3)} |c_k| \cdot u^{k+1} \\ &= u \sum_{k=0}^{N-1} \left(k - \frac{2k-1}{2(2k+3)} \right) |c_k| \cdot u^k \leq \frac{u}{6} + u^2 \sum_{k=1}^{N-1} k \cdot |c_k| \cdot u^{k-1} \\ &\leq \frac{u}{6} + \left(\frac{u}{1-u} \right)^2 \end{aligned}$$

und

$$\begin{aligned}
\sum_{k=1}^{N} k^2 \cdot |c_k| \cdot u^k &= \sum_{k=0}^{N-1} (k+1)^2 \cdot |c_{k+1}| \cdot u^{k+1} = \sum_{k=0}^{N-1} \frac{(k+1)(2k+1)^2}{2(2k+3)} |c_k| \cdot u^{k+1} \\
&= u \sum_{k=0}^{N-1} \left(k^2 + \frac{1}{2}k + \frac{2k+1}{2(2k+3)} \right) |c_k| \cdot u^k \le u \sum_{k=0}^{N-1} \left(k^2 + \frac{1}{2}k + \frac{1}{2} \right) |c_k| \cdot u^k \\
&= u^3 \sum_{k=2}^{N-1} (k^2 - k)|c_k| \cdot u^{k-2} + \frac{3}{2}u^2 \sum_{k=1}^{N-1} k \cdot |c_k| \cdot u^{k-1} + \frac{u}{2} \sum_{k=0}^{N-1} |c_k| \cdot u^k \\
&\le 2\left(\frac{u}{1-u}\right)^3 + \frac{3}{2}\left(\frac{u}{1-u}\right)^2 + \frac{1}{2} \cdot \frac{u}{1-u} \quad .
\end{aligned}$$

Zusammen mit den Abschätzungen

$$P(u) \ge 1 - \frac{u}{6} \quad \text{bzw.} \quad |P'(u)| \le \frac{1}{6} \quad \text{und} \quad |P''(u)| \le \frac{3}{20}$$

ergibt sich insgesamt

$$\begin{aligned}
\varepsilon_P \in \; & \frac{E_\ell}{1 - \frac{u}{6}} \left\{ 1 + (2 + 3E_\ell + E_\ell^2) \frac{u}{1-u} + (1 + 4E_\ell + 2E_\ell^2) \left(\frac{u}{6} + \left(\frac{u}{1-u} \right)^2 \right) \right. \\
& \left. + E_\ell \left(2\left(\frac{u}{1-u}\right)^3 + \frac{3}{2}\left(\frac{u}{1-u}\right)^2 + \frac{1}{2} \cdot \frac{u}{1-u} \right) + (1 + 2E_\ell)\frac{u}{6} + E_\ell \frac{3}{20} u^2 \right\} \quad .
\end{aligned}$$

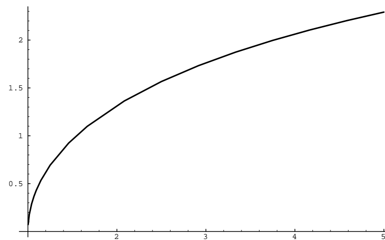
Für $b = 10$, $\ell \ge 5$ und $u = x^2 \le \frac{1}{64}$, d. h. $\bar{x} = \frac{1}{8}$ ergibt sich insgesamt

$$\varepsilon_P \in 1.04 \cdot E_\ell$$

und damit die Notwendigkeit zweier Schutzziffern bei Forderung k–stelliger hoher Genauigkeit. Gemäß (2.2) folgt daraus für den Approximationsfehler

$$|\varepsilon_{app}| \le 8.9591 \cdot 10^{-(k+1)} \quad .$$

2.2.6 Der Areakosinus



Die Umkehrfunktion des hyperbolischen Kosinus kann für Argumente x mit $x > 1$ über die Potenzreihe

$$\operatorname{arcosh}(x) = \sqrt{x^2-1} \cdot \sum_{k=0}^{\infty} c_k \cdot (x-1)^k \quad \text{mit} \quad c_k := \begin{cases} 1 & , \quad \text{für} \quad k = 0 \\ -c_{k-1}\frac{k}{2k+1} & , \quad \text{für} \quad k > 0 \end{cases}$$

berechnet werden. In der Praxis wird die Reihe nur auf einem Intervall $(1, \bar{x}]$ mit geeignet gewählter Obergrenze $1 < \bar{x}$ ausgewertet.
Für die verbleibenden Fälle werden die Identitäten

$$\operatorname{arcosh}(x) = \begin{cases} 0 & , \quad \text{für} \quad x = 1 \\ \ln\left(x + \sqrt{x^2-1}\right) & , \quad \text{für} \quad x > \bar{x} > 1 \end{cases}$$

verwendet.

Approximationsfehler bei Abbruch nach dem N–ten Glied

Für $x > 1$ gilt nach ([Kr], S. 69 [2]) folgende Abschätzung des relativen Approximationsfehlers:

$$|\varepsilon_{app}| \leq \frac{|c_{N+1}|(x-1)^{N+1}}{1 - |c_1|(x-1)} = \frac{|c_{N+1}|(x-1)^{N+1}}{1 - \frac{1}{3}(x-1)} \quad . \tag{2.16}$$

Fehleranalyse für $x > \bar{x}$

Die Fehleranalyse für Argumente $x > \bar{x}$ wird analog zum Areasinus behandelt. Mit den Abkürzungen $y := x + \sqrt{x^2-1}$ und $\kappa := \frac{y}{\ln(y)} \cdot \frac{1}{\xi} \quad , \xi \in \operatorname{conv}\{y, \tilde{y}\}$ ergibt sich wie dort

$$\left|\varepsilon_{\ln(\tilde{y})}\right| \leq \frac{\varepsilon_{\ln(y)}}{2 + \varepsilon_{\ln(y)}} \qquad \text{bzw.} \qquad |\varepsilon_y| \leq \left|\frac{1}{\kappa}\right| \cdot \frac{\varepsilon_{\ln(y)}}{2} \quad .$$

Auch in diesem Fall ist der Term $\frac{y}{\ln(y)}$ für $x \to \infty$ nach oben unbeschränkt. Wie beim Areasinus folgt aus der Abschätzung

$$\frac{1}{\xi} \leq \frac{1}{y(1-|\varepsilon_y|)} = \frac{1}{(x+\sqrt{x^2-1}) \cdot (1-|\varepsilon_y|)}$$

für κ unmittelbar

$$\kappa \leq \frac{x+\sqrt{x^2-1}}{\ln(x+\sqrt{x^2-1})} \cdot \frac{1}{(x+\sqrt{x^2-1}) \cdot (1-|\varepsilon_y|)} = \frac{1}{\ln(x+\sqrt{x^2-1}) \cdot (1-|\varepsilon_y|)} \quad .$$

Mit einer Minimalforderung an ε_y, etwa $|\varepsilon_y| \leq 2 \cdot b^{-4}$ und damit einer auf mindestens fünf Stellen hochgenauen Berechnung des Argumentes des natürlichen Logarithmus, läßt sich aufgrund des Monotonieverhaltens des Logarithmus und der Tatsache, daß

$$x + \sqrt{x^2-1} \geq \bar{x} + \sqrt{\bar{x}^2-1} > 1$$

[2] Auch hier enthält die Originalarbeit einen offensichtlichen Tippfehler.

gilt, folgende Schranke für κ angeben:

$$\kappa \leq \frac{1}{\ln(\bar{x} + \sqrt{\bar{x}^2 - 1}) \cdot (1 - 2 \cdot b^{-4})} \quad .$$

Für $b = 10$ und $\bar{x} = \frac{9}{8}$ führt dies auf $\kappa \leq 2.0209$, woraus für $k \geq 5$ wiederum die Notwendigkeit zweier zusätzlicher signifikanter Ziffern bei der Berechnung des natürlichen Logarithmus und dessen Arguments folgt. Unter der Voraussetzung, daß die Addition von x und $\sqrt{x^2 - 1}$ exakt durchgeführt wird, ergibt die weitere Analyse unter Verwendung der Abkürzung $z = x^2 - 1$

$$\left|\varepsilon_{\sqrt{z}}\right| \leq \left|\frac{x + \sqrt{x^2 - 1}}{\sqrt{x^2 - 1}}\right| \cdot |\varepsilon_y| \quad .$$

Der Koeffizient von $|\varepsilon_y|$ ist monoton fallend in x und für $x > 1$ durch 2 nach unten beschränkt. Die bei der Berechnung von z und der anschließenden Auswertung der Wurzelfunktion einzuhaltenden Genauigkeiten ergeben sich dann zu

$$|\varepsilon_z| \leq \frac{\sqrt{z}}{z \cdot \frac{1}{2\sqrt{\xi_z}}} \cdot \frac{\left|\varepsilon_{\sqrt{z}}\right|}{2} = \sqrt{\frac{\xi_z}{z}} \cdot \left|\varepsilon_{\sqrt{z}}\right|$$

für $\xi_z \in \operatorname{conv}\{z, \tilde{z}\}$ bzw.

$$\left|\varepsilon_{\sqrt{\tilde{z}}}\right| \leq \frac{\left|\varepsilon_{\sqrt{z}}\right|}{2 + \left|\varepsilon_{\sqrt{z}}\right|} \quad .$$

Aus der ersten Ungleichung erhält man wegen der Monotonie der Quadratwurzel

$$|\varepsilon_z| \leq \sqrt{(1 - |\varepsilon_z|)} \cdot \left|\varepsilon_{\sqrt{z}}\right| \quad .$$

Mit einer der an $|\varepsilon_y|$ gestellten Mindestanforderung entsprechenden für $|\varepsilon_z|$ ergeben sich im vorliegenden Wertebereich die folgenden konkreten Schranken:

$$\left|\varepsilon_{\sqrt{z}}\right| \leq 2 \cdot |\varepsilon_y| \quad , \quad |\varepsilon_z| \leq 1.9997 \cdot |\varepsilon_y| \quad \text{und} \quad \left|\varepsilon_{\sqrt{\tilde{z}}}\right| \leq 0.9998 \cdot |\varepsilon_y| \quad .$$

Zieht man nun zusätzlich die zu Beginn hergeleitete Schranke für $|\varepsilon_y|$ ins Kalkül erhält man insgesamt

$$\left|\varepsilon_{\sqrt{z}}\right| \leq 0.49482 \cdot |\varepsilon_{\ln(y)}| \quad , \quad |\varepsilon_z| \leq 0.49475 \cdot \left|\varepsilon_{\ln(y)}\right| \quad \text{und} \quad \left|\varepsilon_{\sqrt{\tilde{z}}}\right| \leq 0.24736 \cdot \left|\varepsilon_{\ln(y)}\right| \quad .$$

Die Berechnung von z und das anschließende Ziehen der Quadratwurzel müssen also für $b = 10$ im Gegensatz zum Areasinus mit nur zwei zusätzlichen signifikanten Ziffern erfolgen.

Fehler der Polynomauswertung nach Horner

Bei genauerer Betrachtung zeigt sich, daß tatsächlich ein Polynom in $u = x - 1$ auszuwerten ist. Es hat die Form

$$P(u) = \sum_{k=0}^{N} c_k \cdot u^k \quad .$$

Zur Fehleranalyse wird dementsprechend Lemma 2.1.11 herangezogen. Da zusätzlich $a_0 = 1$ exakt darstellbar ist, kann die folgende Fassung zur Anwendung gebracht werden:

$$\begin{aligned}\varepsilon_P \;\in\; & \frac{E_\ell}{P(u)}\left\{1+(2+3E_\ell+E_\ell^2)\sum_{k=1}^{N}|c_k|\cdot u^k+(1+4E_\ell+2E_\ell^2)\sum_{k=1}^{N}k\cdot|c_k|\cdot u^k\right.\\ & \left.+E_\ell\sum_{k=1}^{N}k^2\cdot|c_k|\cdot u^k\right\}+\frac{E_\ell}{P(u)}\left\{(1+2E_\ell)uP'(u)+E_\ell u^2P''(u)\right\} \quad .\end{aligned}$$

Zur Abschätzung der vorkommenden Summen verfahren wir wie beim Areasinus. Wir geben hier nur noch die Ergebnisse an, die Rechnungen verlaufen vollkommen analog. Es ergibt sich

$$\sum_{k=1}^{N}|c_k|\cdot u^k \le \frac{1}{2}\cdot\frac{u}{1-u}$$

sowie

$$\sum_{k=1}^{N}k\cdot|c_k|\cdot u^k \le \frac{1}{3}\cdot\frac{u}{1-u}+\frac{1}{2}\cdot\left(\frac{u}{1-u}\right)^2$$

und

$$\sum_{k=1}^{N}k^2\cdot|c_k|\cdot u^k \le \frac{1}{2}\cdot\left(\frac{u}{1-u}\right)^3+\frac{5}{4}\left(\frac{u}{1-u}\right)^2+\frac{1}{3}\cdot\frac{u}{1-u} \quad .$$

Zusammen mit den Abschätzungen

$$P(u)\ge 1-\frac{u}{3} \quad \text{bzw.} \quad |P'(u)|\le\frac{1}{3} \quad \text{und} \quad |P''(u)|\le\frac{4}{15}$$

erhält man

$$\begin{aligned}\varepsilon_P \;\in\; & \frac{E_\ell}{1-\frac{u}{3}}\left\{1+(2+3E_\ell+E_\ell^2)\frac{1}{2}\cdot\frac{u}{1-u}+(1+4E_\ell+2E_\ell^2)\left(\frac{1}{3}\cdot\frac{u}{1-u}+\frac{1}{2}\cdot\left(\frac{u}{1-u}\right)^2\right)\right.\\ & \left.+E_\ell\left(\frac{1}{2}\cdot\left(\frac{u}{1-u}\right)^3+\frac{5}{4}\left(\frac{u}{1-u}\right)^2+\frac{1}{3}\cdot\frac{u}{1-u}\right)+(1+2E_\ell)\frac{u}{3}+E_\ell\frac{4}{15}u^2\right\} \quad .\end{aligned}$$

Für $b = 10$, $\ell \ge 5$ und $u = x^2 \le \frac{1}{64}$, d. h. $\bar{x} = \frac{1}{8}$ ergibt sich insgesamt

$$\varepsilon_P \in 1.0319\cdot E_\ell \quad .$$

Der tatsächliche Wert des Areakosinus ergibt sich allerdings erst durch Multiplikation des zuvor betrachteten Polynoms mit dem Term $\sqrt{x^2-1}$. Unter der Voraussetzung, daß sich dessen Fehler $\varepsilon_{\surd}$ und der Gesamtfehler der Polynomapproxiamtion ε_{ges} die Waage halten sollen, ergibt sich bei geforderter k–stelliger hoher Genauigkeit

$$\varepsilon \le \frac{b^{-k}}{2+b^{-k}} \quad \text{mit} \quad \varepsilon := \max\{|\varepsilon_{\surd}|, |\varepsilon_{ges}|\} \quad .$$

Verwendet man diese schärfere Schranke zur Bestimmung der zur Durchführung des Horner–Schemas notwendigen Stellenzahl ℓ ergibt sich in Analogie zu (2.3)

$$\ell > k+1+\log_b(1.0319\cdot(2+b^{-k})) \quad .$$

Für $b = 10$ bedeutet dies $\ell = k + 2$. Eine obere Schranke für den Approximationsfehler liefert eine entsprechend modifizierte Fassung von (2.2)

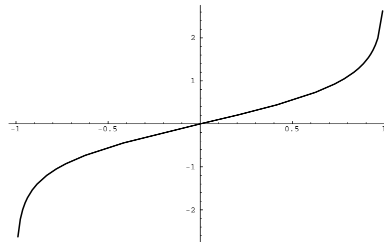
$$|\varepsilon_{app}| \leq \frac{\frac{10^{-k}}{2+10^{-k}} - |\varepsilon_P|}{1 + |\varepsilon_P|} = \frac{\frac{10}{2+10^{-k}} - 1.0319}{1 + 1.0319 \cdot 10^{-(k+1)}} \cdot 10^{-(k+1)}$$

woraus

$$|\varepsilon_{app}| \leq 3.9652 \cdot 10^{-(k+1)}$$

gefolgert werden kann. Falls der Radikand $x^2 - 1$ exakt ermittelt wird, reichen für die Berechnung der Quadratwurzel ebenfalls zwei zusätzliche Schutzziffern. Diese Annahme ist im vorliegenden Wertebereich keine schwerwiegende Einschränkung, da x^2 in der Nähe von 1 liegt und daher keine aufwendige Exponentenangleichung durchzuführen ist.

2.2.7 Der Areatangens



Die Umkehrfunktion des hyperbolischen Tangens kann für Argumente x mit $|x| < 1$ über die Potenzreihe

$$\operatorname{artanh}(x) = x \cdot \sum_{k=0}^{\infty} \frac{1}{2k+1} \cdot x^{2k}$$

berechnet werden. In der Praxis wird die Reihe nur auf einem Intervall $(0, \bar{x}]$ mit geeignet gewählter Obergrenze $0 < \bar{x} \ll 1$ ausgewertet.
Für die verbleibenden Fälle werden die Identitäten

$$\operatorname{artanh}(x) = \begin{cases} -\operatorname{artanh}(-x) & , \text{ für } \quad -1 < x < 0 \\ 0 & , \text{ für } \quad x = 0 \\ \frac{1}{2} \ln\left(\frac{1+x}{1-x}\right) & , \text{ für } \quad 1 > x > \bar{x} \end{cases}$$

verwendet.

Approximationsfehler bei Abbruch nach dem N–ten Glied

Für $x < 1$ gilt nach ([Br], S. 41) folgende Abschätzung des relativen Approximationsfehlers:

$$|\varepsilon_{app}| \leq \frac{x^{2N+2}}{2N+3} \cdot \frac{1}{1-x^2} \quad . \tag{2.17}$$

Fehleranalyse für $x > \bar{x}$

Die Fehleranalyse für Argumente $1 > x > \bar{x}$ wird analog zum Areasinus bzw. -kosinus behandelt. Zusätzlich wird dabei vorausgesetzt, daß die Division durch Zwei im verwendeten Raster fehlerfrei durchführbar ist. Mit den Abkürzungen $y := \frac{1+x}{1-x}$ und $\kappa := \frac{y}{\ln(y)} \cdot \frac{1}{\xi}$, $\xi \in \operatorname{conv}\{y, \tilde{y}\}$ gilt wie dort

$$\left|\varepsilon_{\ln(\tilde{y})}\right| \leq \frac{\varepsilon_{\ln(y)}}{2 + \varepsilon_{\ln(y)}} \qquad \text{bzw.} \qquad |\varepsilon_y| \leq \left|\frac{1}{\kappa}\right| \cdot \frac{\varepsilon_{\ln(y)}}{2} \quad .$$

Wie beim Areasinus folgt aus der Abschätzung

$$\frac{1}{\xi} \leq \frac{1}{y(1 - |\varepsilon_y|)} = \frac{1}{\frac{1+x}{1-x} \cdot (1 - |\varepsilon_y|)}$$

für κ unmittelbar

$$\kappa \leq \frac{\frac{1+x}{1-x}}{\ln(\frac{1+x}{1-x})} \cdot \frac{1}{\frac{1+x}{1-x} \cdot (1 - |\varepsilon_y|)} = \frac{1}{\ln(\frac{1+x}{1-x}) \cdot (1 - |\varepsilon_y|)} \quad .$$

Für $b = 10$, $\bar{x} = \frac{1}{8}$ gilt mit der Forderung $|\varepsilon_y| \leq 2 \cdot b^{-4}$ folgende Schranke für κ:

$$\kappa \leq \frac{1}{\ln(\frac{9}{7}) \cdot (1 - 2 \cdot b^{-4})} = 3.9799 \quad ,$$

woraus sich wiederum

$$\left|\varepsilon_{\ln(\tilde{y})}\right| \leq 0.49995 \cdot \varepsilon_{\ln(y)} \qquad \text{bzw.} \qquad |\varepsilon_y| \leq 0.12563 \cdot \varepsilon_{\ln(y)}$$

ergibt. Unter der Annahme, daß $1 + x$ und $1 - x$ jeweils exakt berechnet werden, müssen Division und Auswertung des Logarithmus mit zwei zusätzlichen signifikanten Stellen durchgeführt werden. Die obige Annahme ist aufgrund der Tatsache, daß $x \in (\frac{1}{8}, 1)$ gilt keine wesentliche Einschränkung.

Fehler der Polynomauswertung nach Horner

Bei genauerer Betrachtung zeigt sich, daß tatsächlich das folgende Polynom in $u = x^2$ auszuwerten ist:

$$P(u) = \sum_{k=0}^{N} \frac{u^k}{2k+1} \quad .$$

Zur Fehleranalyse wird dementsprechend Lemma 2.1.11 herangezogen. Da zusätzlich $a_0 = 1$ exakt darstellbar ist und $a_n u^n \geq 0$ gilt, kann die folgende Fassung zur Anwendung gebracht werden:

$$\varepsilon_P \in \frac{E_\ell}{P(u)} \left\{ 1 + (2 + 3E_\ell + E_\ell^2)(P(u) - 1) + (2 + 7E_\ell + 2E_\ell^2) u P'(u) + 2E_\ell u^2 P''(u) \right\} \quad .$$

Hierzu müssen die Ausdrücke $u\frac{P'(u)}{P(u)}$ bzw. $u^2\frac{P''(u)}{P(u)}$ nach oben abgeschätzt werden. Wie beim Arcustangens bereits praktiziert, ziehen wir nach einer Indexverschiebung die geometrische Reihe zu Hilfe und nutzen außerdem die Tatsache, daß $P(u) \geq 1$ gilt.
Wir erhalten

$$u\frac{P'(u)}{P(u)} \leq uP'(u) = u \sum_{k=0}^{N-1} \frac{k+1}{2k+3} u^k \leq \frac{u}{2} \cdot \frac{1}{1-u}$$

und

$$\begin{aligned} u^2 \frac{P''(u)}{P(u)} &\leq u^2 P''(u) = u^2 \sum_{k=0}^{N-2} \frac{(k+1)(k+2)}{2k+5} u^k \\ &\leq \frac{u^2}{2} \sum_{k=0}^{N-2} (k+1) u^k \leq \frac{u^2}{2} \left(\frac{u}{(1-u)^2} + \frac{1}{1-u} \right) \quad . \end{aligned}$$

Einsetzen dieser Abschätzungen in die ursprüngliche Beziehung ergibt

$$\begin{aligned} \varepsilon_P \in\ & E_\ell \left\{ 1 + (2 + 3E_\ell + E_\ell^2)(1 - \frac{x}{\operatorname{artanh}(x)}) + (2 + 7E_\ell + 2E_\ell^2) \frac{u}{2(1-u)} \right. \\ & \left. + E_\ell u^2 \left(\frac{u}{1-u} \right)^2 \right\} \quad . \end{aligned}$$

Für $b = 10$, $\ell \geq 5$, $x \leq \bar{x} = \frac{1}{8}$ ergibt sich

$$\varepsilon_P \in 1.0264 \cdot E_\ell$$

und damit die Notwendigkeit zweier Schutzziffern bei Forderung k–stelliger hoher Genauigkeit. Gemäß (2.2) folgt daraus für den Approximationsfehler

$$|\varepsilon_{app}| \leq 8.9727 \cdot 10^{-(k+1)} \quad .$$

2.3 Standardfunktionen über Newton-Verfahren

2.3.1 Intervall–Newton–Verfahren

Zur Berechnung einer Einschließung einer einfachen Nullstelle einer stetigen Funktion f kann das folgende, dem vereinfachten Newton–Verfahren ähnliche Intervall–Verfahren verwendet werden (vgl. [Kie], S. 61, Satz 10 oder [AH], S. 69, Theorem 1):

Satz 2.3.1 *Die reelle Funktion f sei stetig differenzierbar im Intervall $X^{(0)}$ und für alle $x \in X^{(0)}$ gelte $f'(x) \in D = [d_1, d_2]$ mit $0 < d_1 \leq d_2 < \infty$. Für $\hat{x} \in X^{(0)}$ gelte $f(\hat{x}) = 0$. Wird für ein beliebiges $x^{(t)} \in X^{(t)}$ definiert*

$$X^{(t+1)} := X^{(t)} \cap \left\{ x^{(t)} - \frac{f(x^{(t)})}{D} \right\} \quad (t = 0, 1, 2, \ldots)$$

so gilt

1. $X^{(t+1)} \subset X^{(t)}$.
2. $\hat{x} \in X^{(t)}$.
3. $\lim_{t \to \infty} X^{(t)} = \hat{x}$.

Bemerkung: Nach ([AH], S. 74, Theorem 3) führt die Wahl $x^{(t)} = \mathrm{mid}(X^{(t)})$ zu einem optimalen Verfahren in dem Sinne, daß die Durchmesser $\mathrm{diam}(X^{(t+1)})$ der Iterierten im Vergleich zu jeder anderen Wahl minimal werden.

Das skizzierte Verfahren hat die Konvergenzordnung 1. Zu einem Verfahren mit mindestens quadratischer Konvergenz gelangt man für eine stetig differenzierbare Funktion f, deren Ableitung f' die Intervall–Auswertungen $f'(X^{(t)}) \quad (t = 0, 1, 2, \ldots)$ erlaubt, durch folgende Modifikation. Das Intervall D wird in jedem Schritt durch

$$D^{(t)} := f'(X^{(t)}) \cap [l_1, l_2]$$

mit

$$0 < l_1 \leq f'(x) \leq l_2, \quad x \in X^{(0)}$$

ersetzt (vgl. [AH], S. 75, Theorem 4).

Bemerkung: Bei einer Implementierung eines Verfahrens nach obigen Vorschriften, muß die Berechnung von

$$x^{(t)} - \frac{f(x^{(t)})}{D} \quad \text{bzw.} \quad x^{(t)} - \frac{f(x^{(t)})}{D^{(t)}}$$

intervallmäßig erfolgen und dazu insbesondere eine garantierte Einschließung von $f(x^{(t)})$ berechnet werden. Die Qualität des Verfahrens hängt entscheidend von der Güte dieser Einschließung ab! Falls $x^{(t)} = \mathrm{mid}(X^{(t)})$ gewählt wird, ist es *nicht* erforderlich, auch diesen einzuschließen. Es muß lediglich gewährleistet sein, daß seine Berechnung nicht aus $X^{(t)}$ herausführt.

Es bleibt das Problem der Bestimmung eines geeigneten Startintervalls $X^{(0)}$. In der Regel ist nur eine Punkt-Näherung $\tilde{x}$ für die gesuchte Nullstelle $\hat{x}$ bekannt. Für die im Rahmen dieser Arbeit mit Hilfe des Newton–Verfahrens behandelten Funktionen Quadratwurzel und natürlicher Logarithmus kann der folgende Satz (vgl. [Er], Abschnitt 10.2) weiterhelfen.

Satz 2.3.2 *Die Funktion $f : [a, b] \to \boldsymbol{R}$ sei zweimal stetig differenzierbar, streng monoton wachsend und strikt konvex. Falls $f(a) < 0$ und $f(b) > 0$ erfüllt ist, gelten folgende Aussagen:*

1. *Es existiert genau ein $\hat{x} \in [a, b]$ mit $f(\hat{x}) = 0$.*
2. $a' := a - f(a)\frac{b-a}{f(b)-f(a)} \in (a, \hat{x}]$.
3. $b' := b - \frac{f(b)}{f'(b)} \in [\hat{x}, b)$.
4. $f(a') < 0$ *und* $f(b') > 0$.

Die Güte der Näherung b' kann durch

$$b' - \hat{x} \leq \frac{1}{m} f(b') \leq \frac{M}{2m}(b - b')^2$$

mit $m := \min\{f'(x) | \hat{x} \leq x \leq b'\} \leq \min\{f'(x) | a \leq x \leq b'\}$ *und* $M := \max\{f''(x) | b' \leq x \leq b\}$ *abgeschätzt werden.*

Bemerkung: Wie auch in [Er] ausgeführt, erlaubt dieser Satz ebenfalls die Konstruktion eines iterativen Intervall–Verfahrens zur Verbesserung der Einschließung einer einfachen Nullstelle. Die Folge der Obergrenzen entspricht dabei der der Iterierten des Newton–Verfahrens für den Startwert b. Die Folge der Untergrenzen ist eine Majorante der Iterationsfolge der Regula Falsi für die Startwerte a, b.

Mit Hilfe der a posteriori Fehlerabschätzung kann bei Kenntnis einer unteren Schranke für m aus einer Punkt-Näherung $x^{(0)} := \tilde{x}$ (mit $f(\tilde{x}) \neq 0$!) ein Kandidat für ein Startintervall $X^{(0)}$ bestimmt werden. Dazu ist zunächst unter Verwendung von Intervall–Arithmetik ein weiterer Newton–Schritt auszuführen. Aus der so erhaltenen Einschließung $[\tilde{\tilde{x}}]$ einer neuen Näherung $\tilde{\tilde{x}}$ wird wie folgt ein Kandidat für ein Startintervall $X^{(0)}$ bestimmt.

Einschließung:

1. $\boldsymbol{f(\tilde{x}) > 0:}$ $\quad X^{(0)} := \left[\inf\left([\tilde{\tilde{x}}]\right) - \delta, \min\left\{\tilde{x}, \sup\left([\tilde{\tilde{x}}]\right)\right\}\right]$ mit $\delta := \frac{1}{m} \cdot f\left(\sup\left([\tilde{\tilde{x}}]\right)\right)$.
2. $\boldsymbol{f(\tilde{x}) < 0:}$ $\quad X^{(0)} := \left[\tilde{x}, \sup\left([\tilde{\tilde{x}}]\right)\right]$.

Bemerkung: Die Verwendung von Intervall-Arithmetik bei der Berechnung einer weiteren Näherung ist wesentlich für die Anwendbarkeit von Satz (2.3.2)! Ebenso muß bei der praktischen Berechnung von δ durch entsprechend gerichtete Rundungen für eine sichere obere Schranke gesorgt werden.

$X^{(0)}$ ist dann ein geeignetes Startintervall für jedes der zuvor beschriebenen Intervall–Verfahren, da für deren Konvergenz $\hat{x}$ nicht notwendigerweise im Inneren von $X^{(0)}$ zu liegen braucht.

Sollte die Bestimmung des Vorzeichens von $f(\tilde{x})$ in der Praxis nicht möglich sein, kann der Versuch der Einschließung statt dessen mit der neuen Näherung $\tilde{x} := (1+\varepsilon) \cdot \sup\left([\tilde{\tilde{x}}]\right)$ für ein heuristisch gewähltes $\varepsilon > 0$ wiederholt werden. Dieser Wert ist unter den gegebenen Voraussetzungen stets größer als $\hat{x}$. In der Praxis muß bei seiner Berechnung selbstverständlich entsprechend gerichtet gerundet werden.
Die auf den ersten Blick kompliziert erscheinende Verbindung eines klassischen und eines Intervallverfahrens vermeidet den hohen Aufwand, der bei der ausschließlichen Verwendung des letzteren entsteht. Falls die Näherung $\tilde{x}$ schon ausreichend gut ist, stellt im günstigsten Fall bereits das durch Aufblähung erhaltene Startintervall $X^{(0)}$ eine Einschließung mit der geforderten Genauigkeit dar.

Konvergenzerhaltung durch Rasteranpassung

Bei durchgehender Rechnung in einem festen ℓ–stelligen Raster ist in der Regel *nicht* zu erwarten, daß die nachgeschalteten Intervall–Verfahren eine auf ℓ Stellen hochgenaue Einschließung liefern. Es stellt sich deshalb die Frage, wie sich eine Erhöhung der Stellenzahl auf die Güte des relativen Fehlers einer weiteren Newton–Iterierten auswirkt. Wir wollen dies am Beispiel des vereinfachten Newton–Verfahrens untersuchen. Bei Verwendung ℓ–stelliger Arithmetik gilt für den absoluten Fehler Δ_{t+1} einer aus $\tilde{x}_t$ berechneten neuen Näherung $\tilde{x}_{t+1}$ der Nullstelle $\hat{x}$ von f

$$
\begin{aligned}
\Delta_{t+1} &:= \tilde{x}_{t+1} - \hat{x} = \tilde{x}_t -_\ell \tilde{f}(\tilde{x}_t) /_\ell d - \hat{x} \\
&\in \left\{ \hat{x} + \Delta_t - (1+2E_\ell) \cdot \frac{f(\hat{x}+\Delta_t)}{d}(1+E_\ell) \right\} (1+E_\ell) - \hat{x} \\
&\subseteq \hat{x}E_\ell + \Delta_t(1+E_\ell) - (1+2E_\ell)(1+E_\ell)^2 \cdot \frac{f(\hat{x}+\Delta_t)}{d} \\
&\subseteq \hat{x}E_\ell + \Delta_t(1+E_\ell) - (1+2E_\ell)(1+E_\ell)^2 \cdot \frac{f(\hat{x}) + \Delta_t \cdot f'(\xi)}{d} \\
&\quad \text{mit } \xi \in \operatorname{conv}\{\hat{x}, \tilde{x}_t\} \\
&\subseteq \Delta_t \left(1 - \frac{f'(\xi)}{d}\right) + \hat{x}E_\ell + \Delta_t E_\ell + E_\ell(4 + 5E_\ell + 2E_\ell^2)\Delta_t \frac{f'(\xi)}{d} \quad .
\end{aligned}
$$

Dabei wurde davon ausgegangen, daß f hochgenau ausgewertet werden kann, also $\tilde{f}(x) \in (1+2E_\ell) \cdot f(x)$ für alle x im Definitionsbereich von f gilt. Außerdem muß $\ell > 2$ erfüllt sein, damit unabhängig von der Basis b stets $0 \notin (1+2E_\ell)$ gilt.
Unter der Voraussetzung $\hat{x} \neq 0$ liefert der Übergang zu relativen Fehlern

$$
\begin{aligned}
\varepsilon_{t+1} &:= \frac{\Delta_{t+1}}{\hat{x}} \\
&\in \varepsilon_t \left(1 - \frac{f'(\xi)}{d}\right) + E_\ell + \varepsilon_t E_\ell + E_\ell(4 + 5E_\ell + 2E_\ell^2)\varepsilon_t \frac{f'(\xi)}{d} \\
&\subseteq \varepsilon_t \left(1 - \frac{f'(\xi)}{d}\right) + E_\ell \left(1 + |\varepsilon_t| + (4 + 5E_\ell + 2E_\ell^2)|\varepsilon_t| \left|\frac{f'(\xi)}{d}\right|\right) \\
&\subseteq \varepsilon_t \left(1 - \frac{f'(\xi)}{d}\right) + E_\ell \left\{1 + |\varepsilon_t| \left(1 + (4 + 5E_\ell + 2E_\ell^2) \left|\frac{f'(\xi)}{d}\right|\right)\right\} \quad .
\end{aligned}
$$

Um tatsächlich eine Abnahme des relativen Fehlers ε_{t+1} zu erzielen, genügt es die hinreichende Bedingung

$$|\varepsilon_t| \left|1 - \frac{f'(\xi)}{d}\right| + b^{1-\ell} \left\{1 + |\varepsilon_t| \left(1 + (4 + 5b^{1-\ell} + 2b^{2-2\ell}) \left|\frac{f'(\xi)}{d}\right|\right)\right\} < |\varepsilon_t| \tag{2.18}$$

zu erfüllen. Der Faktor $\left|1 - \frac{f'(\xi)}{d}\right| \le q < 1$ ist dabei bestimmend für die theoretische Konvergenzrate des vereinfachten Newton–Verfahrens und kann bei Kenntnis einer Einschließung der gesuchten Nullstelle a priori abgeschätzt werden.

Bemerkung: Die oben aufgestellte Bedingung ist auf das entsprechende Intervall–Verfahren übertragbar. Dazu muß lediglich der Punkt d durch das Intervall D ersetzt werden und an geeigneter Stelle die Suprema der entstehenden Intervalle gewählt werden.

Eine entsprechende Analyse für das allgemeine Newton–Verfahren liefert, falls auch die Ableitung f' hochgenau auswertbar und zudem differenzierbar ist

$$\begin{aligned}
\Delta_{t+1} &:= \tilde{x}_{t+1} - \hat{x} = \tilde{x}_t -_\ell \tilde{f}(\tilde{x}_t) /_\ell \tilde{f}'(\tilde{x}_t) - \hat{x} \\
&\in \left\{\hat{x} + \Delta_t - \frac{(1+2E_\ell)}{(1+2E_\ell)} \cdot \frac{f(\hat{x}+\Delta_t)}{f'(\tilde{x}_t)}(1+E_\ell)\right\}(1+E_\ell) - \hat{x} \\
&\subseteq \Delta_t \left(1 - \frac{f'(\xi)}{f'(\hat{x})}\right) + \Delta_t^2 \cdot \frac{f'(\xi) f''(\eta)}{f'(\hat{x}) f'(\tilde{x}_t)} + \hat{x} E_\ell + \Delta_t E_\ell \\
&\quad - E_\ell \cdot \Delta_t \cdot \frac{f'(\xi)}{f'(\tilde{x}_t)} \cdot \left\{2 + E_\ell + (2 + \frac{4E_\ell}{1+2E_\ell})(1+E_\ell)^2\right. \\
&\quad \left. + 2 \cdot (1 + 2E_\ell + \frac{4E_\ell^2}{1+2E_\ell})(1+E_\ell)^2\right\} \quad \text{mit } \xi, \eta \in \operatorname{conv}\{\hat{x}, \tilde{x}_t\} \quad .
\end{aligned}$$

Bemerkung: Zur Herleitung wird eine weitere Rechenregel für die Fehlerintervalle E_ℓ benötigt. Sie lautet

$$\frac{1}{1 + x \cdot E_\ell} \subseteq 1 + x \cdot E_\ell + \frac{x^2 \cdot E_\ell^2}{1 + x \cdot E_\ell} \quad \text{für alle } x \in \boldsymbol{R}$$

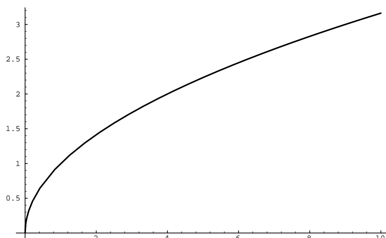
und kann z. B. mit Hilfe der geometrischen Reihe bewiesen werden.

Unter der Voraussetzung $\hat{x} \neq 0$ liefert der Übergang zu relativen Fehlern

$$\begin{aligned}
\varepsilon_{t+1} &:= \frac{\Delta_{t+1}}{\hat{x}} \\
&\in \varepsilon_t \left(1 - \frac{f'(\xi)}{f'(\hat{x})}\right) + \varepsilon_t^2 \cdot \hat{x} \cdot \frac{f'(\xi) f''(\eta)}{f'(\hat{x}) f'(\tilde{x}_t)} \\
&\quad + E_\ell \cdot \varepsilon_t \cdot \frac{f'(\xi)}{f'(\tilde{x}_t)} \cdot \left\{3 + |\varepsilon_t| + E_\ell + (2 + \frac{4E_\ell}{1+2E_\ell})(1+E_\ell)^2\right. \\
&\quad \left. + 2 \cdot (1 + 2E_\ell + \frac{4E_\ell^2}{1+2E_\ell})(1+E_\ell)^2\right\} \quad .
\end{aligned}$$

Bemerkung: Die Übertragung dieses Resultats auf das entsprechende Intervall-Verfahren ist leider *nicht* direkt möglich, da im Nenner nicht der Punkt $\tilde{f}(\tilde{x}_t)$ sondern das Intervall $\tilde{f}(\tilde{X}_t)$ steht.

2.3.2 Die Quadratwurzel



Die Quadratwurzel $\sqrt{z}$ einer reellen Zahl $z > 0$ wird üblicherweise mit Hilfe des gewöhnlichen Newton–Verfahrens als positive Nullstelle $\hat{x}$ der Funktion

$$f(x) = x^2 - z$$

bestimmt. Das Verfahren konvergiert für alle Startwerte $x_0 > \hat{x}$ streng monoton fallend gegen $\sqrt{z}$ (vgl. [Stoe], S. 245, Satz (5.5.5)). Die Iterationsvorschrift lautet

$$x_{t+1} := x_t - \frac{x_t^2 - z}{2 \cdot x_t} = \frac{1}{2}\left(x_t + \frac{z}{x_t}\right) \quad (t = 0, 1, 2, \ldots) \quad .$$

Vor Beginn des Verfahrens wird der im halboffenen Intervall $[b^k, b^{k+1}) \quad (k \in \boldsymbol{Z})$ liegende Radikand z je nach Parität des Exponenten k in eines der Intervalle $[b^{-1}, 1)$, bzw. $[b^{-2}, b^{-1})$ transformiert. Dies geschieht mittels Division durch eine gradzahlige Potenz der Basis b und ist somit fehlerfrei durchführbar. Im Anschluß wird für den so erhaltenen Wert z' eine Näherung $\tilde{x}$ von $\sqrt{z'}$ gemäß obiger Iteration bestimmt.

Bemerkung: Es ist für das weitere Vorgehen *nicht* wesentlich, wie die Näherung $\tilde{x}$ bestimmt wird. Dies kann etwa auch mit Hilfe einer im jeweiligen System verfügbaren Quadratwurzel–Routine erfolgen.

Um aus dieser Näherung durch Aufblähung ein Startintervall $X^{(0)}$ zu erhalten, wird eine untere Schranke m für die erste Ableitung von $x^2 - z'$ benötigt. Diese ergibt sich aufgrund der oben durchgeführten Transformation und der damit implizit verfügbaren a priori Information über die Lage der positiven Nullstelle von $x^2 - z'$ zu

$$m = \begin{cases} 2 \cdot b^{-1} & , \text{ falls } k \equiv 0 \bmod 2 \\ 2 \cdot \sqrt{b^{-1}} & , \text{ falls } k \equiv 1 \bmod 2 \end{cases} \quad .$$

Sollte die zur Rechtfertigung dieses Vorgehens erforderliche Bedingung

$$\tilde{x} \in \begin{cases} [b^{-1}, \sqrt{b^{-1}}) & , \text{ falls } k \equiv 0 \bmod 2 \\ [\sqrt{b^{-1}}, 1) & , \text{ falls } k \equiv 1 \bmod 2 \end{cases}$$

nicht erfüllt sein, kann das entsprechende Intervall unmittelbar zur Initialisierung von $X^{(0)}$ herangezogen werden. Es wäre ebenfalls denkbar, die Güte von $\tilde{x}$ mit Hilfe einiger Newton–Schritte zu verbessern. Der für die Bestimmung des Startintervalls $X^{(0)}$ zusätzlich erforderliche Vorzeichentest kann in dynamischen Rastern stets erfolgreich durchgeführt werden.
Zur eventuell erforderlichen Verbesserung der erhaltenen Einschließung wird dann die Iteration

$$X^{(t+1)} := X^{(t)} \cap \left\{ \mathrm{mid}(X^{(t)}) - \frac{\mathrm{mid}(X^{(t)})^2 - z'}{2 \cdot X^{(t)}} \right\} \qquad (t = 0, 1, 2, \ldots)$$

bis zur Erfüllung einer geeigneten Abbruchbedingung durchgeführt. Das Ergebnisintervall $\tilde{X}$ ist anschließend gegebenenfalls einer Rücktransformation zu unterziehen.
In dynamischen Rastern könnten bis auf die Division alle dazu erforderlichen Operationen exakt ausgeführt werden. Es ist jedoch trotzdem in der Regel nicht zu erwarten, daß das Verfahren beim Erreichen des Stagnationspunktes ($X^{(t+1)} = X^{(t)}$) eine hochgenaue Einschließung ermittelt hat. Wir wollen daher eine (2.18) entsprechende Bedingung herleiten, die es erlaubt anzugeben, wieviele zusätzliche Ziffern erforderlich sind, um in einem weiteren Verfahrensschritt tatsächlich eine Verbesserung der Einschließung zu erhalten. Wir gehen dabei davon aus, daß das Quadrieren x_t^2 und die Multiplikation mit Zwei exakt durchführbar sind. Die auf diese Verhältnisse angepaßte Analyse des vereinfachten Newton–Verfahrens liefert

$$\begin{aligned}
\Delta_{t+1} &:= \tilde{x}_{t+1} - \sqrt{z'} = \tilde{x}_t -_\ell (\tilde{x}_t^2 -_\ell z')/_\ell d - \sqrt{z'} \\
&\in \left\{ \sqrt{z'} + \Delta_t - \frac{(\sqrt{z'} + \Delta_t)^2 - z'}{d} \cdot (1 + E_\ell)^2 \right\} (1 + E_\ell) - \sqrt{z'} \\
&\subseteq \sqrt{z'} E_\ell + \Delta_t (1 + E_\ell) - \frac{2\sqrt{z'}\Delta_t + \Delta_t^2}{d} \cdot (1 + E_\ell)^3 \\
&\subseteq \Delta_t \left(1 - \frac{2\sqrt{z'} + \Delta_t}{d} \right) + \sqrt{z'} E_\ell + \Delta_t E_\ell + E_\ell (3 + 3E_\ell + E_\ell^2) \Delta_t \frac{2\sqrt{z'} + \Delta_t}{d} \quad .
\end{aligned}$$

Nach dem Übergang zu relativen Fehlern ($z' \neq 0$) erhält man

$$\begin{aligned}
\varepsilon_{t+1} &:= \frac{\Delta_{t+1}}{\sqrt{z'}} \\
&\in \varepsilon_t \left(1 - \frac{2\sqrt{z'} + \Delta_t}{d} \right) + E_\ell \left(1 + |\varepsilon_t| + E_\ell (3 + 3E_\ell + E_\ell^2) |\varepsilon_t| \left| \frac{2\sqrt{z'} + \Delta_t}{d} \right| \right) \\
&\subseteq \varepsilon_t \left(1 - \frac{\sqrt{z'}(2 + \varepsilon_t)}{d} \right) + E_\ell \left\{ 1 + |\varepsilon_t| \left(1 + E_\ell (3 + 3E_\ell + E_\ell^2) \left| \frac{\sqrt{z'}(2 + \varepsilon_t)}{d} \right| \right) \right\} \quad .
\end{aligned}$$

Um also in einem weiteren Schritt tatsächlich eine Verbesserung der Näherung zu erzielen, genügt es, die hinreichende Bedingung

$$b^{1-\ell} \left\{ 1 + |\varepsilon_t| \left(1 + b^{1-\ell}(3 + 3b^{1-\ell} + b^{2-2\ell}) \left| \frac{\sqrt{z'}(2 + \varepsilon_t)}{d} \right| \right) \right\} < |\varepsilon_t| \left(1 - \left| 1 - \frac{\sqrt{z'}(2 + \varepsilon_t)}{d} \right| \right)$$

zu erfüllen. Nimmt man an, daß $\varepsilon_t \in E_k$, $\ell > k \geq 2$ und $b = 10$ gelten, kann (2.19) mit Hilfe der a priori Informationen über die Lage von $\sqrt{z'}$ und der daraus ableitbaren groben Abschätzung $0 < \frac{\sqrt{z'}}{d} \leq \frac{1}{2}$ wie folgt konkretisiert werden:

$$10^{1-\ell}\left\{1 + 0.1\left(1 + 0.030301\left|\frac{2-0.1}{2}\right|\right)\right\} < 10^{1-k}\left(1 - \left|1 - \frac{2-0.1}{2}\right|\right)) \quad .$$

Logarithmieren und Umstellen führt schlußendlich auf die Forderung

$$\ell > k + \log_{10}(1.102878595) - \log_{10}(0.95) \quad .$$

Da die Differenz der beiden Logarithmen im Intervall $[0.06, 0.07]$ liegt, reicht also bereits eine zusätzliche Stelle, um die theoretische Verbesserung nicht durch den Einfluß von Rundungsfehlern zu verlieren. Mit zwei zusätzlichen Ziffern in *jedem* weiteren Schritt des Intervall–Verfahrens dürfte daher die Konvergenz gegen eine hochgenaue Einschließung mehr als gesichert sein.

Beispiel: (angeregt durch eine mündliche Mitteilung von R. Stiefken)

In der `ATARI`–Version von `PASCAL-SC`, die über eine 13–stellige `BCD`–Arithmetik verfügt, liefert das gewöhnliche Newton–Verfahren mit der umgeformten Verfahrensvorschrift

$$x_{t+1} := \frac{1}{2}\left(x_t + \frac{z}{x_t}\right) \quad (t = 0, 1, 2, \ldots)$$

und Rundung zur nächsten Gleitpunktzahl als Näherung der Wurzel von 0.3 den Wert $\tilde{x} = 0.547722557505$. Die Iteration stagniert an dieser Stelle, der Wert ist also ein Fixpunkt des Gleitpunkt–Verfahrens! Er besitzt eine Genauigkeit von 2 ulp, da $\sqrt{0.3} \in [0.547722557505^2_1]$ gilt.

Der Vorzeichentest liefert wie zu erwarten

$$0.547722557505^2 - 0.3 < 0 \quad .$$

Ein weiterer mit Intervall-Rechnung (!) durchgeführter Newton–Schritt (ohne die obige Umformung) ergibt

$$[\tilde{\tilde{x}}] = [0.547722557505^2_1] \quad .$$

(Der Wert 0.5477225575052 ist der entsprechende Fixpunkt bei Verwendung der Originalvorschrift $x_{t+1} := x_t - (x_t^2 - 0.3)/2x_t$!) Gemäß Teil 2 der Vorschrift zur Aufblähung kann direkt

$$X^{(0)} := [0.547722557505^2_0]$$

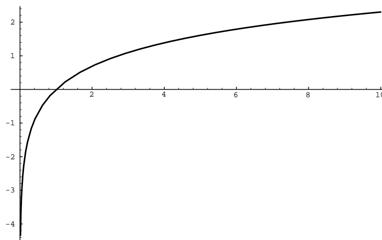
als Startintervall verwendet werden. Es handelt sich zwar bereits um eine hochgenaue Einschließung, trotzdem ist das Intervall–Verfahren in der Lage — bei durchgehender Verwendung 13–stelliger `BCD`–Arithmetik — die weitere Iterierte

$$X^{(1)} = [0.547722557505^2_1]$$

zu berechnen. Bereits nach einem Schritt ist also sogar eine maximal genaue Einschließung ermittelt, und dies ohne Verwendung von Skalarprodukt–Ausdrücken oder ähnlicher Hilfsmittel. Die Gültigkeit der Beziehung $0.5477225575051^2 \leq 0.3$ ist zudem mit 13–stelliger

BCD–Arithmetik nicht mehr zu überprüfen, wird jedoch durch das Intervall–Verfahren garantiert.
In dynamischen Rastern würde man selbstverständlich bei der Ermittlung des Startintervalls anders vorgehen. Bereits der mit Intervall–Arithmetik durchzuführende gewöhnliche Newton–Schritt wird mit höherer Genauigkeit ausgeführt. Bei einer entsprechend guten Ausgangsnäherung ist dann häufig die durch Aufblähung erhaltene Einschließung bereits hochgenau und das Intervall–Verfahren kommt überhaupt nicht mehr zur Anwendung.

2.3.3 Der natürliche Logarithmus



Der natürliche Logarithmus $\ln(z)$ einer reellen Zahl $z > 0$ kann mit Hilfe des gewöhnlichen Newton–Verfahrens als die einzige Nullstelle $\hat{x}$ der Funktion

$$f(x) = e^x - z$$

angenähert werden.Die Iterationsvorschrift lautet

$$x_{t+1} := x_t - \frac{e^{x_t} - z}{e^{x_t}} \quad (t = 0, 1, 2, \ldots) \quad .$$

Eine einfache Betrachtung zeigt, daß das Verfahren global konvergiert. Für Startwerte $x_0 < \hat{x}$ gilt nach einem Iterationsschritt $x_1 > \hat{x}$. Für solche Werte liefert das Verfahren streng monoton fallende Konvergenz.
Im Gegensatz zur Quadratwurzel verwenden wir als nachgeschaltetes Intervallverfahren das in Satz (2.3.1) beschriebene, dem vereinfachten Newton–Verfahren nachempfundene. Dabei wird $x^{(t)} = \text{mid}(X^{(t)})$ gewählt. Zwar ist die Intervallauswertung der Ableitung $f'(x) = e^x$ für die Iterierten $X^{(t)}$ möglich, führt aber bei Wahl von $\text{mid}(X^{(t)})$ für $x^{(t)}$ zur Notwendigkeit von zwei weiteren Auswertungen der Exponentialfunktion für $\inf(X^{(t)})$ bzw. $\sup(X^{(t)})$ in jedem Iterationsschritt. Selbst bei Wahl eines dieser Werte für $x^{(t)}$ wäre immer noch eine zusätzliche Auswertung der Exponentialfunktion an der verbleibenden Intervallgrenze erforderlich. Wir halten deshalb im Laufe des Verfahrens $D = e^{X^{(0)}}$ fest.

Zur Bestimmung von $X^{(0)}$ aus einer mit dem klassischen Verfahren gewonnenen Näherung $\tilde{x}$ dienen folgende Vorüberlegungen. Für ein im halboffenen Intervall $[b^k, b^{k+1})$ liegendes Argument z gilt die grobe Einschließung $\ln(z) \in \ln(b) \cdot [k, k+1)$. Falls $\tilde{x} \in \ln(b) \cdot [k, k+1)$ gilt, kann der Wert $m = b^k$ zusammen mit der nächsten intervallmäßig aus $\tilde{x}$ berechneten Newton–Iterierten $[\tilde{\tilde{x}}]$ zur Bestimmung von $X^{(0)}$ gemäß der Vorschrift zur Aufblähung verwendet werden. Sollte die Voraussetzung nicht erfüllt sein, kann selbstverständlich $X^{(0)} = \ln(b) \cdot [k, k+1]$ gewählt werden. Falls $X^{(0)}$ nicht bereits die gewünschte Genauigkeit besitzt, wird $D = e^{X^{(0)}}$ bestimmt und die Iteration

$$X^{(t+1)} := X^{(t)} \cap \left\{ \mathrm{mid}(X^{(t)}) - \frac{e^{\mathrm{mid}(X^{(t)})} - z}{D} \right\} \quad (t = 0, 1, 2, \ldots)$$

durchgeführt. Bei der praktischen Durchführung sind die bereits bei der Quadratwurzel angesprochenen Konvergenzprobleme zu berücksichtigen. Nach (2.18) genügt es, um in einem weiteren Schritt tatsächlich eine Verbesserung der Näherung zu erzielen, die hinreichende Bedingung

$$\begin{aligned} & b^{1-\ell} \left\{ 1 + |\varepsilon_t| \left(1 + (4 + 5b^{1-\ell} + 2b^{2-2\ell}) \frac{exp(\xi)}{\sup(D)} \right) \right\} \\ < \quad & |\varepsilon_t| \left(1 - \left| 1 - \frac{exp(\xi)}{\sup(D)} \right| \right) \quad \text{mit } \xi \in \mathrm{conv}\{\log(z), \mathrm{mid}(X^{(t)})\} \end{aligned} \tag{2.19}$$

zu erfüllen. Nimmt man an, daß $\varepsilon_t \in E_k$, $\ell > k \geq 2$ und $b = 10$ gelten, kann (2.19) mit Hilfe der a priori Informationen über die Lage von $\log(z)$ und der daraus ableitbaren groben Abschätzung $0 < \frac{exp(\xi)}{\sup(D)} \leq \frac{1}{10}$ wie folgt konkretisiert werden:

$$10^{1-\ell} \left\{ 1 + 0.1 \left(1 + 4.0502 \frac{1}{10} \right) \right\} < 10^{1-k} \left(1 - \left| 1 - \frac{1}{10} \right| \right)) \quad .$$

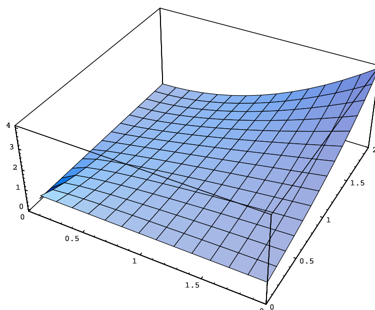
Logarithmieren und Umstellen führt schlußendlich auf die Forderung

$$\ell > k + \log_{10}(1.140502) + 1 \quad .$$

Da $\log_{10}(1.140502) \in [0.05, 0.06]$ gilt, reichen also zwei zusätzliche Stellen, um die theoretische Verbesserung nicht durch den Einfluß von Rundungsfehlern zu verlieren.

2.4 Standardfunktionen über algebraische Identitäten

2.4.1 Die allgemeine Potenzfunktion



Die allgemeine Potenzfunktion wird auf dem Umweg über Exponentialfunktion und natürlichen Logarithmus berechnet. Es gilt

$$x^y = e^{\ln(x)\cdot y} \quad \text{für} \quad x > 0 \quad .$$

Folgende Sonderfälle werden zusätzlich berücksichtigt:

$$x^y = \begin{cases} 0 & , \text{ falls } \; x = 0 \; \text{ und } \; y \neq 0 \\ 1 & , \text{ falls } \; x \neq 0 \; \text{ und } \; y = 0 \\ (-1)^{|y|} \cdot |x|^y & , \text{ falls } \; x < 0 \; \text{ und } \; y \in \boldsymbol{Z} \end{cases} \quad .$$

Der zuletzt aufgeführte Sonderfall kann entweder auf den allgemeinen Fall abgebildet werden oder aber durch iterierte Multiplikation mit dem Aufwand $O(\log_2(y))$ behandelt werden. Zur Fehleranalyse können dann Lemma (2.1.1) bzw. (2.1.2) herangezogen werden.

Fehleranalyse für den allgemeinen Fall

Wie die verwandte Exponentialfunktion wird die allgemeine Potenzfunktion gemäß Kapitel 1, (1.14) unter Verwendung relativer Fehler behandelt. Bei einer geforderten Genauigkeit von b^{-k} ergeben sich mit den Abkürzungen $z := \ln(x)\cdot y$ und $\kappa := z \cdot \frac{\exp(\xi)}{\exp(z)}$, $\xi \in \operatorname{conv}\{z, z(1+\varepsilon_z)\}$ die folgenden Schranken für die Güte der Funktionsauswertung und der Berechnung von z:

$$\left|\varepsilon_{\exp(\tilde{z})}\right| \leq \frac{\exp(z)}{\exp(\tilde{z})} \cdot \frac{b^{-k}}{2} \quad \text{bzw.} \quad |\varepsilon_z| \leq \left|\frac{1}{\kappa}\right| \cdot \frac{b^{-k}}{2} \quad .$$

Ohne Einschränkung des Definitionsbereichs läßt sich hieraus a priori keine Information über eventuell notwendige Schutzziffern ableiten. Es wird deshalb in der Regel ein Wiederberechnungsschritt erforderlich sein. Mit den bei der erstmaligen Auswertung erhaltenen

Einschließungen $Z := [\ln(x) \cdot y]$ und $[x^y] \in W := \exp(Z)$ lassen sich nötigenfalls wie folgt obere Schranken für die wiederholte Auswertung ermitteln:

$$\left|\varepsilon_{\exp(\tilde{z})}\right| \leq \frac{\sup(W)}{\inf(W)} \cdot \frac{b^{-k}}{2} \quad \text{bzw.} \quad |\varepsilon_z| \leq \left|\frac{1}{\sup(|Z|)\frac{\sup(W)}{\inf(W)}}\right| \cdot \frac{b^{-k}}{2} \quad .$$

Die bei der Bestimmung von $\ln(x)$ einzuhaltende Fehlerschranke ergibt sich für $\varepsilon := \max\{|\varepsilon_{\cdot}|, |\varepsilon_{\ln(x)}|\}$ dann aus der Forderung $\varepsilon \cdot (1+\varepsilon) \leq |\varepsilon_z|$.

Die zugehörige Intervallfunktion

Es ist zur Bestimmung von $Z := X^Y$ nicht notwendig, die allgemeine Potenzfunktion für alle vier in Frage kommenden Kombinationen der Grenzen der Intervallargumente X und Y auszuwerten. Nach ([Kr], S.47f) kann das Ergebnis $Z := [z_1, z_2]$ mit $z_1 := x_1^{y_1}$ und $z_2 := x_2^{y_2}$ durch folgende Wahlen von x_i, y_i, $i \in \{1, 2\}$ berechnet werden:

Bedingung		x_1	x_2	y_1	y_2
$\sup(X) \leq 1$,	$\sup(Y) \leq 0$	$\sup(X)$	$\inf(X)$	$\sup(Y)$	$\inf(Y)$
	$\inf(Y) \geq 0$	$\inf(X)$	$\sup(X)$	$\sup(Y)$	$\inf(Y)$
	$\inf(Y) < 0 < \sup(Y)$	$\inf(X)$	$\inf(X)$	$\sup(Y)$	$\inf(Y)$
$\inf(X) \geq 1$,	$\sup(Y) \leq 0$	$\sup(X)$	$\inf(X)$	$\inf(Y)$	$\sup(Y)$
	$\inf(Y) \geq 0$	$\inf(X)$	$\sup(X)$	$\inf(Y)$	$\sup(Y)$
	$\inf(Y) < 0 < \sup(Y)$	$\inf(X)$	$\inf(X)$	$\inf(Y)$	$\sup(Y)$
$\inf(X) < 1 < \sup(X)$,	$\sup(Y) \leq 0$	$\sup(X)$	$\inf(X)$	$\inf(Y)$	$\inf(Y)$
	$\inf(Y) \geq 0$	$\inf(X)$	$\sup(X)$	$\sup(Y)$	$\sup(Y)$
	$\inf(Y) < 0 < \sup(Y)$	*	*	*	*

.

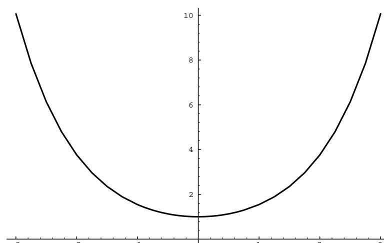
In den mit (*) gekennzeichneten Fällen ergibt sich das Ergebnisintervall Z gemäß der Vorschrift

$$\begin{aligned} Z \quad := \quad & [\exp(\min\{\sup(Y) \cdot \ln(\inf(X)), \inf(Y) \cdot \ln(\sup(X))\}), \\ & \exp(\max\{\sup(Y) \cdot \ln(\sup(X)), \inf(Y) \cdot \ln(\inf(X))\})] \quad . \end{aligned}$$

Auch bei der Intervallversion ist ein Sonderfall zu berücksichtigen. Für ein Punktintervall $Y = [y, y]$, $y \in \boldsymbol{N}$ und $0 \in X$ ist die allgemeine Potenzfunktion ebenfalls definiert. Es ergibt sich dann

$$X^Y := \begin{cases} [0, \sup(|X|)^y] & , \text{ falls } y \text{ gerade} \\ [\inf(X)^y, \sup(X)^y] & , \text{ falls } y \text{ ungerade} \end{cases} \quad .$$

2.4.2 Der hyperbolische Kosinus



Der hyperbolische Kosinus wird ausschließlich über die Identität

$$\cosh(x) = \frac{e^x + e^{-x}}{2} \tag{2.20}$$

berechnet. Im Gegensatz zum hyperbolischen Sinus besteht hierbei keine Gefahr der Auslöschung.

Fehleranalyse

In fast völliger Analogie zur entsprechenden Betrachtung beim hyperbolischen Sinus führt klassische Fehlerrechnung, unter der Voraussetzung, daß die Division durch Zwei fehlerfrei durchführbar ist, auf

$$\begin{aligned} |\varepsilon_{\cosh(x)}| &= \left| \frac{e^x}{2\cosh(x)} \varepsilon_{\exp(x)} + \frac{e^{-x}}{2\cosh(x)} \left(\varepsilon_/ - \frac{\varepsilon_{\exp(x)}}{1+\varepsilon_{\exp(x)}} (1+\varepsilon_/) \right) \right| \\ &\le \frac{\varepsilon}{2\cosh(x)} \left(2\cosh(x) + e^{-x} \frac{1+\varepsilon}{1-\varepsilon} \right) \quad, \text{ mit } \varepsilon := \max(|\varepsilon_{\exp(x)}|, |\varepsilon_/|) \\ &\le \varepsilon \left(1 + \frac{e^{-x}}{2\cosh(x)} \frac{(1+\varepsilon)}{(1-\varepsilon)} \right) \quad . \end{aligned}$$

Unter den zusätzlichen Voraussetzungen $\varepsilon \le 2 \cdot 10^{-4}$ und $x > 0$ folgt $|\varepsilon_{\cosh(x)}| \le 1.5003 \cdot \varepsilon$. Dies führt zur Notwendigkeit zweier Schutzziffern.

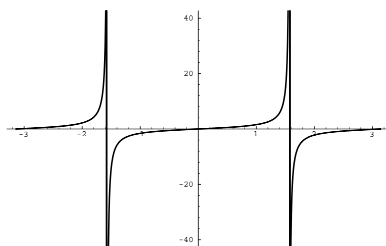
Die zugehörige Intervallfunktion

Auf den offenen Halbstrahlen $(-\infty, 0)$ bzw. $(\infty, 0)$ ist der hyperbolische Kosinus jeweils streng monoton fallend bzw. steigend. Aufgrund der zusätzlich vorhandenen Symmetrie er-

gibt sich für den hyperbolischen Intervallkosinus insgesamt folgendes Berechnungsschema:

$$\cosh(X) := \begin{cases} [\cosh(\sup(X)), \cosh(\inf(X))] & , \text{ falls } \sup(X) < 0 \\ [1, \cosh(\sup(|X|))] & , \text{ falls } 0 \in X \\ [\cosh(\inf(X)), \cosh(\sup(X))] & , \text{ falls } 0 < \inf(X) \end{cases} .$$

2.4.3 Der Tangens



Zur Berechnung des Tangens kann die Reihenentwicklung

$$\tan(x) = \sum_{n=1}^{\infty} (-1)^{n-1} \frac{2^{2n}(2^{2n}-1)B_{2n}}{(2n)!} \cdot x^{2n-1}$$

mit

$$B_0 := 1, \quad B_n := -\frac{1}{n+1} \cdot \sum_{k=0}^{n-1} \binom{n+1}{k} B_k \quad \text{für} \quad n > 0$$

herangezogen werden. Die Berechnung der Bernoulli–Zahlen B_n und der dabei auftretenden Binomialkoeffizienten ist sehr aufwendig und muß für dynamische Raster mittels rationaler Arithmetik ausgeführt werden. Auch nach einer Reduktion des Argumentes auf das Intervall $[0, \frac{\pi}{4})$ konvergiert die Reihe jedoch sehr langsam.
Auf der anderen Seite ist die Verwendung der Identität

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

wegen der Notwendigkeit zweier Funktionsauswertungen ebenfalls nicht optimal. Dies läßt sich jedoch etwas lindern, indem die beiden Funktionsrümpfe in einer einzigen Routine zusammengefaßt werden. Die jeweils anfallende Argumentreduktion mit der dabei eventuell erforderlichen Wiederberechnung braucht so nur einmal durchgeführt zu werden.

Fehleranalyse bei Rückführung auf Sinus und Kosinus

Hier ist zunächst die Fehlerfortpflanzung bei der Division von fehlerbehafteten Größen zu untersuchen. Es gilt

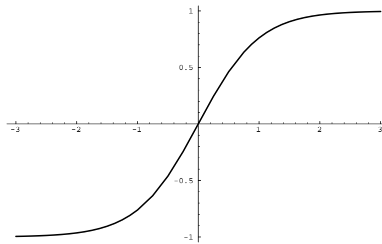
$$\frac{x \cdot (1+\varepsilon_x)}{y \cdot (1+\varepsilon_y)} = \frac{x}{y} \cdot (1+\varepsilon_x)(1-\frac{\varepsilon_y}{1+\varepsilon_y}) = \frac{x}{y} \cdot (1+\frac{\varepsilon_x-\varepsilon_y}{1+\varepsilon_y}) \quad .$$

Für $\varepsilon := \max(|\varepsilon_x|, |\varepsilon_y|)$ folgt somit

$$|\varepsilon_{\frac{x}{y}}| \leq \frac{2\varepsilon}{1-\varepsilon} \quad .$$

Hieraus ergibt sich für die Basis $b = 10$, daß für Sinus und Kosinus jeweils zwei zusätzliche signifikante Stellen zu berechnen sind.

2.4.4 Der hyperbolische Tangens



Wie beim Tangens kommen in der Reihenentwicklung

$$\tanh(x) = \sum_{n=1}^{\infty} \frac{2^{2n}(2^{2n}-1)B_{2n}}{(2n)!} \cdot x^{2n-1}$$

des hyperbolischen Tangens die Bernoulli–Zahlen B_n vor. Daraus ergeben sich die gleichen Nachteile wie im Falle des Tangens.
Das Ausweichen auf die Identität

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x}-1}{e^{2x}+1}$$

ist in diesem Fall jedoch nicht so problematisch, da für Argumente größeren Betrages nur eine einzige Auswertung der Exponentialfunktion notwendig ist. Wegen der Auslöschungsgefahr im Zähler ist dieser Weg wie beim hyperbolischen Sinus für betragskleine Argumente $x \leq \bar{x}$ nicht empfehlenswert. Hier müssen hyperbolischer Sinus und Kosinus separat berechnet werden. Wie beim Tangens hat dies für $b = 10$ auf jeweils zwei zusätzliche signifikante Stellen zu erfolgen.

Fehleranalyse bei Verwendung der Exponentialfunktion

Zunächst reicht o.B.d.A. die Betrachtung positiver Argumente aus, da $\tanh(-x) = -\tanh(x)$ gilt. Setzt man voraus, daß Inkrementieren bzw. Dekrementieren um Eins fehlerfrei durchführbar sind, erhält man mit Hilfe der im vorhergehenden Abschnitt hergeleiteten Fehlerformel für einen Quotienten fehlerbehafteter Größen und der nach klassischer Fehlerfortpflanzung ermittelten relativen Fehler in Zähler und Nenner

$$|\varepsilon_{\tanh(x)}| = \left| \frac{\frac{e^{2x}}{e^{2x}-1}\varepsilon_{\exp(2x)} - \frac{e^{2x}}{e^{2x}+1}\varepsilon_{\exp(2x)}}{1 + \frac{e^{2x}}{e^{2x}+1}\varepsilon_{\exp(2x)}} \right| \quad .$$

Mit der Abkürzung $\varepsilon := |\varepsilon_{\exp(2x)}|$ und unter Berücksichtigung der Tatsache, daß $\frac{e^{2x}}{e^{2x}+1} < 1$ gilt, ergibt sich daraus

$$|\varepsilon_{\tanh(x)}| \leq \frac{\varepsilon(1 + \frac{e^{2x}}{e^{2x}-1})}{1-\varepsilon} \quad .$$

Für praktische Anwendungen muß der für $x > 0$ monoton fallende Term $\frac{e^{2x}}{e^{2x}-1}$ weiter abgeschätzt werden. Hierzu ist die Spezifikation eines Schwellwertes $\bar{x}$ erforderlich. Aus den gleichen Gründen wie beim hyperbolischen Sinus bietet sich für die Basis $b = 10$ der Wert $\frac{\ln(10)}{4} \approx 0.57564$ an. Hieraus folgt für $x > \bar{x}$

$$\frac{e^{2x}}{e^{2x}-1} \leq 1.4625 \quad .$$

Bei angestrebter k–stelliger hoher Genauigkeit ergibt sich schließlich folgende Forderung

$$\varepsilon \leq \frac{b^{-k}}{2.4625 + b^{-k}} \quad .$$

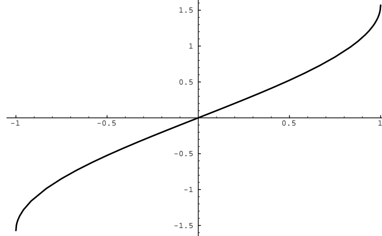
Für $b = 10$ und $k \geq 5$ bedeutet dies, daß zwei zusätzliche signifikante Ziffern von e^{2x} zu bestimmen sind.

Bemerkungen zur effizienten Implementierung

Falls der hyperbolische Tangens für Argumente kleinen Betrages über die separat aufzurufenden Funktionen für den hyperbolischen Sinus bzw. Kosinus berechnet wird, sollte der Schwellwert $\bar{x}$ so gewählt werden, daß bei der Berechnung von $\sinh(x)$ *keine* Argumentreduktion mehr vorgenommen werden muß. Es kann dann direkt die entsprechenden Routine für reduzierte Argumente aufgerufen werden.

Bei der Verwendung der Exponentialfunktion sollte die explizite Multiplikation mit Zwei für gewisse Argumente unterbleiben. Bei unserer Implementierung der e–Funktion sind dies Argumente x mit $\bar{x} < |x| < \ln(10) \approx 2.3025$. Die Multiplikation solcher x mit Zwei macht eine aufwendigere Argumentreduktion, die unter Umständen sogar einen Wiederberechnungsschritt erfordert, nötig. Dies läßt sich vermeiden, indem unter Verwendung zusätzlicher Schutzziffern statt $\exp(2x)$ besser $\exp(x)^2$ berechnet wird.

2.4.5 Der Arcussinus



Bei der Berechnung der Umkehrfunktion des Sinus werden insgesamt vier Fälle unterschieden. Diese sind

$$\arcsin(x) = \begin{cases} -\arcsin(-x) & , \text{ für } \; x < 0 \\ 0 & , \text{ für } \; x = 0 \\ \arctan\left(\frac{x}{\sqrt{1-x^2}}\right) & , \text{ für } \; 0 < x < 1 \\ \frac{\pi}{2} & , \text{ für } \; x = 1 \end{cases} .$$

Fehleranalyse für $x \in (0,1)$

Die Fehleranalyse für Argumente $x \in (0,1)$ wird wie beim Arcustangens nach Kapitel 1, (1.14) behandelt. Mit den Abkürzungen $y := \frac{x}{\sqrt{1-x^2}}$ und $\kappa := \frac{y}{\arctan(y)} \cdot \frac{1}{1+\xi^2}$, $\xi \in \operatorname{conv}\{y, \tilde{y}\}$ erhält man bei geforderter k–stelliger hoher Genauigkeit nach Einführen relativer Fehler

$$\left|\varepsilon_{\arctan(\tilde{y})}\right| \leq \frac{b^{-k}}{2 + b^{-k}} \qquad \text{bzw.} \qquad |\varepsilon_y| \leq \left|\frac{1}{\kappa}\right| \cdot \frac{b^{-k}}{2} .$$

Der Term $\frac{y}{\arctan(y)}$ ist jedoch für $x \to 1$, d. h. $y \to \infty$ nach oben unbeschränkt. Hier kann die folgende Überlegung Abhilfe schaffen. Es gilt für $\xi \in \operatorname{conv}\{y, \tilde{y} = y(1+\varepsilon_y)\}$

$$\frac{1}{1+\xi^2} \leq \frac{1}{1+y^2(1-|\varepsilon_y|)^2} = \frac{1-x^2}{1+x^2(|\varepsilon_y|^2 - 2|\varepsilon_y|)} .$$

Hiermit ergibt sich nun folgende obere Schranke für κ:

$$\begin{aligned} \kappa &\leq \frac{x(1-x^2)}{\sqrt{1-x^2} \cdot \arctan\left(\frac{x}{\sqrt{1-x^2}}\right)} \cdot \frac{1}{1+x^2(|\varepsilon_y|^2 - 2|\varepsilon_y|)} \\ &= \frac{x\sqrt{1-x^2}}{\arctan\left(\frac{x}{\sqrt{1-x^2}}\right)} \cdot \frac{1}{1+x^2(|\varepsilon_y|^2 - 2|\varepsilon_y|)} \leq \frac{1}{1-x^2|\varepsilon_y|(2-|\varepsilon_y|)} . \end{aligned}$$

Mit einer Minimalforderung an ε_y, etwa $|\varepsilon_y| \leq 2 \cdot b^{-4}$ und damit einer auf mindestens fünf Stellen hochgenauen Berechnung des Argumentes des Arcustangens, läßt sich sich im Fall $b = 10$ folgende Schranke für κ angeben:

$$\kappa \leq \frac{1}{1 - 2 \cdot 10^{-4}(2 - 2 \cdot 10^{-4})} \leq 1.0005 \quad .$$

Dies hat für $k \geq 5$ zunächst die Konsequenz, daß für den Arcustangens zwei zusätzliche signifikante Ziffern zu berechnen sind.
Mit der weiteren Abkürzung $z := \sqrt{1 - x^2}$ führt klassische Fehlerrechnung auf die Forderung

$$\frac{|\varepsilon_z|}{1 - |\varepsilon_z|} \cdot (1 + |\varepsilon_/|) + |\varepsilon_/| \leq |\varepsilon_y|$$

und mit $\varepsilon := max\{|\varepsilon_z|\,, |\varepsilon_/|\}$ folgt hieraus zusammen mit der zuvor hergeleiteten Schranke für $|\varepsilon_y|$:

$$\frac{2 \cdot \varepsilon}{1 - \varepsilon} \leq |\varepsilon_y| \leq \frac{b^{-k}}{2.001}$$

und damit:

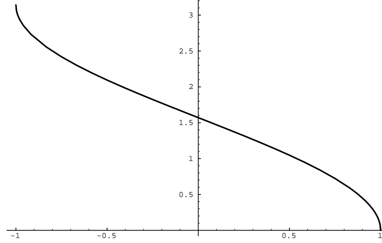
$$\varepsilon \leq \frac{b^{-k}}{4.002 + b^{-k}} \quad .$$

Die bei der Berechnung von $r := 1 - x^2$ und der anschließenden Auswertung der Wurzelfunktion einzuhaltenden Genauigkeiten ergeben sich dann wie beim Areakosinus zu

$$|\varepsilon_r| \leq \sqrt{\frac{\xi_r}{r}} \cdot |\varepsilon| \quad \text{mit} \quad \xi_r \in \text{conv}\{r, \tilde{r}\} \qquad \text{bzw.} \qquad \left|\varepsilon_{\sqrt{\tilde{r}}}\right| \leq \frac{|\varepsilon|}{2 + |\varepsilon|} \leq \frac{b^{-k}}{8.004 + 3 \cdot b^{-k}} \quad .$$

Aus der ersten Ungleichung erhält man wegen der Monotonie der Quadratwurzel $|\varepsilon_r| \leq \sqrt{(1 - |\varepsilon_r|)} \cdot |\varepsilon|$.
Mit einer der an $|\varepsilon_y|$ gestellten Mindestanforderung entsprechenden für $|\varepsilon_r|$ ergibt sich, daß die Berechnung von r und die Division für $b = 10$ und $k \geq 5$ mit zwei zusätzlichen signifikanten Ziffern erfolgen müssen. Beim Ziehen der Quadratwurzel ist hingegen nur noch eine weitere Schutziffer erforderlich.

2.4.6 Der Arcuskosinus



Bei der Berechnung der Umkehrfunktion des Kosinus werden insgesamt fünf Fälle unterschieden. Diese sind:

$$\arccos(x) = \begin{cases} \pi & , \text{ für } x = -1 \\ \arctan\left(\frac{\sqrt{1-x^2}}{x}\right) + \pi & , \text{ für } -1 < x < 0 \\ \frac{\pi}{2} & , \text{ für } x = 0 \\ \arctan\left(\frac{\sqrt{1-x^2}}{x}\right) & , \text{ für } 0 < x < 1 \\ 0 & , \text{ für } x = 1 \end{cases} .$$

Fehleranalyse für $x \in (0,1)$

Die Fehleranalyse für Argumente $x \in (0,1)$ wird analog zum Arcussinus behandelt. Mit den Abkürzungen $y := \frac{\sqrt{1-x^2}}{x}$ und $\kappa := \frac{y}{\arctan(y)} \cdot \frac{1}{1+\xi^2}$, $\xi \in \text{conv}\{y, \tilde{y}\}$ erhält man bei geforderter k–stelliger hoher Genauigkeit

$$\left|\varepsilon_{\arctan(\tilde{y})}\right| \leq \frac{b^{-k}}{2+b^{-k}} \qquad \text{bzw.} \qquad |\varepsilon_y| \leq \left|\frac{1}{\kappa}\right| \cdot \frac{b^{-k}}{2} \quad .$$

Auch hier ist die Abschätzung von $\frac{y}{\arctan(y)}$ nicht unmittelbar möglich, allerdings ist jetzt der Fall $x \to 0$ der problematische. Ein zum Arcussinus analoger Ansatz führt auf

$$\frac{1}{1+\xi^2} \leq \frac{1}{1+y^2(1-|\varepsilon_y|)^2} = \frac{x^2}{1+(1-x^2)(|\varepsilon_y|^2 - 2|\varepsilon_y|)} \quad .$$

Hiermit ergibt sich nun folgende obere Schranke für κ:

$$\begin{aligned} \kappa \quad &\leq \quad \frac{\sqrt{1-x^2} \cdot x^2}{x \cdot \arctan\left(\frac{\sqrt{1-x^2}}{x}\right)} \cdot \frac{1}{1+(1-x^2)(|\varepsilon_y|^2 - 2|\varepsilon_y|)} \\ &= \quad \frac{\sqrt{1-x^2} \cdot x}{\arctan\left(\frac{\sqrt{1-x^2}}{x}\right)} \cdot \frac{1}{1+(1-x^2)(|\varepsilon_y|^2 - 2|\varepsilon_y|)} \leq \frac{1}{1-(1-x^2)|\varepsilon_y|(2-|\varepsilon_y|)} \quad . \end{aligned}$$

Wie beim Arcussinus führt die Minimalforderung $|\varepsilon_y| \leq 2 \cdot b^{-4}$ im Fall $b = 10$ auf die Schranke

$$\kappa \leq \frac{1}{1 - 2 \cdot 10^{-4}(2 - 2 \cdot 10^{-4})} \leq 1.0005 \quad .$$

Auch hier entsteht für $k \geq 5$ zunächst die Notwendigkeit, zwei zusätzliche signifikante Ziffern bei der Berechnung des Arcustangens zu bestimmen.
Mit der weiteren Abkürzung $z := \sqrt{1 - x^2}$ führt klassische Fehlerrechnung auf die Forderung

$$|\varepsilon_z|\,(1 + |\varepsilon_/|) + |\varepsilon_/| \leq |\varepsilon_y|$$

und mit $\varepsilon := max\{|\varepsilon_z|\,, |\varepsilon_/|\}$ und der ohnehin notwendigerweise zu erfüllenden Zusatzbedingung $\varepsilon \leq |\varepsilon_y|$ folgt hieraus

$$\varepsilon \leq \frac{|\varepsilon_y|}{2 + |\varepsilon_y|} \quad .$$

Zusammen mit der zuvor hergeleiteten Schranke für $|\varepsilon_y|$ erhält man:

$$\varepsilon \leq \frac{b^{-k}}{4.002 + b^{-k}} \quad .$$

Für $k \geq 5$ und $b = 10$ ergeben sich hieraus die gleichen Konsequenzen für die Anzahl der benötigten Schutzziffern wie beim Arcussinus.

Fehleranalyse für $x \in (-1, 0)$

Hier ist zunächst die abschließende Addition von π und die durch sie hervorgerufene Fehlerverstärkung zu betrachten. Es stellt sich heraus, daß trotz unterschiedlichen Vorzeichens der Summanden keine Auslöschungsgefahr besteht. Klassische Fehlerrechnung ergibt bei Verwendung der Abkürzung $y := \frac{\sqrt{1-x^2}}{x}$

$$\varepsilon_{\text{arccos}(x)} = \frac{\arctan(y)}{\arctan(y) + \pi} \cdot \varepsilon_{\arctan(y)} + \frac{\pi}{\arctan(y) + \pi} \cdot \varepsilon_\pi \quad .$$

Da für Argumente $x \in (-1, 0)$ der Ausdruck $\arctan\left(\frac{\sqrt{1-x^2}}{x}\right)$ nur Werte in $(-\frac{\pi}{2}, 0)$ annimmt, ergibt sich mit $\varepsilon := max\{\left|\varepsilon_{\arctan(y)}\right|, |\varepsilon_\pi|\}$ die Forderung

$$3 \cdot \varepsilon \leq b^{-k} \quad .$$

Für $b = 10$ hat dies zur Konsequenz, daß zwei zusätzliche Ziffern von π benötigt werden. Die restliche Analyse verläuft vollkommen analog zum vorherigen Abschnitt, da das negative Vorzeichen entweder durch Quadrieren oder Betragsbildung nicht ins Gewicht fällt. Es ist lediglich durchgehend die oben hergeleitete, etwas schärfere Fehlerschranke $\frac{b^{-k}}{3}$ einzusetzen. Im Falle der Basis 10 bleibt die Anzahl erforderlicher Schutzziffern jedoch unverändert.

Kapitel 3

Implementierungen

3.1 Designgrundlagen

Die im Rahmen dieser Arbeit vorgenommenen Implementierungen des in Kapitel 1 vorgestellten Wiederberechnungsverfahren wurden unter besonderer Berücksichtigung der folgenden Vorgaben erstellt:

- Transparenz und einfache Benutzung für den normalen Anwender.
- leichte Erweiterbarkeit durch den fortgeschrittenen Benutzer.
- Portabilität.

Dem letzten Aspekt wurde insbesondere durch die Erstellung hardware–unabhängiger Bibliotheken für Langzahlarithmetik und hochgenaue Standardfunktionen für Punkt- und Intervallargumente in dynamischen Rastern Rechnung getragen. Die theoretischen Grundlagen dazu wurden in Kapitel 2 ausführlich behandelt.
Es soll an dieser Stelle nicht verschwiegen werden, daß eine hardware–unabhängige Langzahlarithmetik selbstverständlich nie die Leistungsfähigkeit einer durch einen arithmetischen Koprozessor unterstützten IEEE–Gleitpunktarithmetik erreichen kann. Hieraus resultieren erheblich höhere Laufzeiten als bei der gewöhnlichen Rechnung. Dies ist aber zu tolerieren, wenn man sich vor Augen führt, daß die auf eine vorgebbare Stellenzahl hochgenaue Auswertung arithmetischer Ausdrücke selbstverständlich nicht die breite Palette spezialisierter Algorithmen der kontrollierten Numerik ([HHKR]) ersetzen will. Sie stellt vielmehr ein zusätzliches Hilfsmittel dar, das durch gezielte Anwendung und geschickte Integration in bestehende Verfahren dazu beitragen kann, qualitativ bessere numerische Resultate zu erhalten.
Vor diesem Hintergrund gewinnt der erste genannte Designgrundsatz zusätzliches Gewicht. Ein Software–Tool mag noch so mächtig sein; wenn es nur umständlich handhabbar und schwierig zu erlernen ist, wird der potentielle Anwender es nicht akzeptieren. Ist diese Hürde überwunden, entsteht in der Regel bald der Wunsch, eine Anpassung an die eigenen Bedürfnisse vorzunehmen. Dem wurde in beiden Implementierungen durch eine offene, klar dokumentierte Schnittstelle Rechnung getragen. Diese nutzt die Möglichkeiten der jeweiligen

Implementierungssprache derart, daß Ergänzungen durch den Benutzer *keine* endlose Kette von Modifikationen im Gesamtsystem nach sich ziehen.

3.2 REDUCE

Das Computer–Algebra–System `REDUCE` bietet in der Version `3.3a` eine im Modul "`ALG1.R`" leidlich dokumentierte Schnittstelle für benutzerdefinierte Arithmetiken, die aber, wenn ihre Funktionsweise einmal verstanden ist, einen für den Benutzer vollkommen transparenten Zugang ermöglicht. Diese Eigenschaft sowie die Offenlegung der Quelltexte und die daraus resultierende Verfügbarkeit auf vielen verschiedenen Rechnerplattformen führten zur Wahl von `REDUCE` als Kandidat für eine Implementierung des in Kapitel 1 vorgestellten Wiederberechnungsverfahrens.

3.2.1 Die Benutzer–Schnittstelle

Anders als etwa *Mathematica* ([Wo]), das numerische Resultate nur nach Aufruf einer speziellen Funktion liefert, trennt `REDUCE` ([Hea]) zunächst einmal streng zwischen symbolischem und numerischem Rechnen. Zwischen beiden Modi kann mit den Befehlen

```
on numval;   % numerischer Modus ein
```

bzw.

```
off numval;  % numerischer Modus aus
```

umgeschaltet werden. In der numerischen Betriebsart stehen verschiedene sogenannte "arithmetic domains" zur Verfügung. Im Standardlieferumfang befinden sich u. a. die Gleitpunktarithmetiken:

– `FLOAT`.

– `BIGFLOAT`.

Sie ermöglichen das Rechnen mit den im System verwendeten Maschinenzahlen (in der Regel eines der IEEE–Formate) sowie die Verwendung einer softwaremäßig realisierten überlangen Gleitpunktarithmetik zur Basis 10 mit vorgebbarer Mantissenlänge und theoretisch unbeschränktem Exponentenbereich ([Sa]). Eine breite Palette von Standardfunktionen wird von beiden unterstützt.
Die Auswahl eines dieser arithmetischen Bereiche erfolgt nach der Umschaltung in den numerischen Modus durch

```
on float;
```

bzw.

```
on bigfloat;
```

Die Festlegung der Mantissenlänge `N` erfolgt für die Domäne `BIGFLOAT` durch den Aufruf

```
precision N;
```

Die auf dem in Kapitel 1 vorgestellten Wiederberechnungsverfahren basierende selbstverifizierende Arithmetik wird vollkommen analog durch die Eingabe

```
on accurate;
```

aktiviert und die gewünschte Anzahl signifikanter Stellen `N` aller vom System zurückgegebenen Werte durch

```
accuracy N;
```

spezifiziert.

Das folgende Protokoll einer Beispielsitzung zeigt, wie die am Ende von Kapitel 1 zur Demonstration des Wiederberechnungsverfahrens erörterte Frage, ob $e^{\pi\sqrt{163}} \notin \boldsymbol{Z}$ gilt, mit Hilfe von `REDUCE` unter Verwendung der neuen Domäne `ACCURATE` geklärt werden kann:

```
reduce 3.3a (15 jan 1988):

1: on numval;

2: on bigfloat;

3: precision 31;

4: exp(pi*sqrt(163));

26253 74126 40768 743.9 99999 99999 92

5: on accurate;

*** domain mode bigfloat changed to accurate

6: accuracy 31;

7: input 4;

[2.625374126407687439999999999992E+17,
 2.625374126407687439999999999993E+17]
```

3.2.2 Implementationsdetails

Verwendete Datenstrukturen

`REDUCE` ist weitgehend in `RLISP`, einer prozeduralen `LISP`–Erweiterung, implementiert. Der `RLISP`–Interpreter gehört zum System und setzt wiederum auf `LISP` auf. Es ist daher nicht

verwunderlich, daß die zur Realisierung der selbstverifizierenden Arithmetik verwendete Datenstruktur auf Listen basiert. Ihr Aufbau ist im folgenden mit einer Mischung aus Backus–Naur–Form und RLISP–Syntax beschrieben. Punkte "." und runde Klammern "(", bzw. ")" sind dabei nicht im metasprachlichen Sinne zu verstehen.

```
<tagged evaluated arithmetic expression> ::=
        ('!:ibfexpr!: . (<evaluated arithmetic expression>)
```

Bemerkung: Das sog. Tag "!:ibfexpr!:" dient der Typkennung zur Laufzeit, da LISP eine untypisierte Sprache ist.

```
<evaluated arithmetic expression> ::=
        ( <bigfloat interval> . <atomic expression> ) |
        ( <bigfloat interval> . ( <operator>
                                  <evaluated arithmetic expression>
                                  {<evaluated arithmetic expression>} ))

<bigfloat interval> ::= ( '!:ibf!: . ( <bigfloat> . <bigfloat> ) )
```

Bemerkung: Die beiden BIGFLOAT's repräsentieren Unter– und Obergrenze des Intervalls und zwar in dieser Reihenfolge.

```
<atomic expression> ::= <domain element> |
                        'e |
                        'pi
```

Bemerkung: Zulässige Wahlen für <domain element> stellen Elemente arithmetischer Bereiche dar, die eine intervallmäßige Einschließung mit vorgebbarer relativer bzw. absoluter Fehlerschranke erlauben.

```
<operator> ::= 'minus |
               'plus |
               'difference |
               'times |
               'quotient |
               <standard function>

<standard function> ::= 'fabs  |
                        'sqrt  |
                        'exp   |
                        'log   |
                        'sin   |
                        'cos   |
                        'tan   |
                        'asin  |
```

```
'acos  |
'atan  |
'sinh  |
'cosh  |
'tanh  |
'asinh |
'acosh |
'atanh |
...
```

Bemerkungen:

- Die Punkte sollen andeuten, daß die Menge der unterstützten Funktionen erweitert werden kann.
- Die allgemeine Potenzfunktion taucht hier nicht auf, da sie bis auf Spezialfälle stets mittels der Identität `pow(x,y) = exp(log(x)*y)` berechnet wird.
- Die Implementierung der arithmetischen Operationen auf Werten vom Typ `<evaluated arithmetic expression>` erfolgt in kanonischer Art und Weise und wird deshalb hier nicht ausgeführt.

Verwendete Programmierschnittstelle

Die Integration der auf der zuvor beschriebenen Datenstruktur basierenden Arithmetik in das System geschieht über die folgenden Anweisungen:

```
module ibfdmain;  % REDUCE unterstuetzt ein Modulkonzept !

% externe globale Variable

% !:prec!:     fuer bigfloat aktuell gueltige Stellenzahl
% !:ibfacc!:   gibt die geforderte signifikante Stellenzahl an
% domainlist!* enthaelt die im System verfuegbaren arithmetischen Bereiche

global '(!:prec!: !:ibfacc!: domainlist!*);


% accurate als neue arithmetic domain mit dem
% tag-feld '!:ibfexpr!: vereinbaren

switch accurate;

domainlist!* := union('(!:ibfexpr!:), domainlist!*);

put('accurate,'tag,'!:ibfexpr!:);
```

```
put('!:ibfexpr!:,'dname,'accurate);

flag('(!:ibfexpr!:),'field);

% Konvertierungsroutinen aus anderen domains deklarieren

put('!:ibfexpr!:,'i2d,'i2ibfexpr!*);       % integer
put('!:bf!:,'!:ibfexpr!:,'bf2ibfexpr!*);   % bigfloat
put('!:ft!:,'!:ibfexpr!:,'ft2ibfexpr!*);   % float
put('!:rn!:,'!:ibfexpr!:,'rn2ibfexpr!*);   % rational

% arithmetische Praedikate

put('!:ibfexpr!:,'minusp,'ibfexpr!_minusp!*);   % < 0 ?
put('!:ibfexpr!:,'zerop,'ibfexpr!_zerop!*);     % = 0 ?
put('!:ibfexpr!:,'onep,'ibfexpr!_onep!*);       % = 1 ?

% arithmetische Operationen

put('!:ibfexpr!:,'minus,'ibfexpr!_minus!*);   % unaeres Minus

put('!:ibfexpr!:,'plus,'ibfexpr!_plus!*);
put('!:ibfexpr!:,'difference,'ibfexpr!_difference!*);
put('!:ibfexpr!:,'times,'ibfexpr!_times!*);
put('!:ibfexpr!:,'quotient,'ibfexpr!_quotient!*);

% Die Funktion prepfn stellt normalerweise die Schnittstelle zum
% symbolischen Simplifizierer dar, wird in unserem Fall jedoch
% zum Einklinken des eventuell erforderlichhen Wiederberechnungs-
% schrittes verwendet

put('!:ibfexpr!:,'prepfn,'ibfexpr!_prep!*);

% Ausgaberoutine

put('!:ibfexpr!:,'prifn,'ibfexpr!_prin!*);

% Umwandlung in Standardquotienten

put('!:ibfexpr!:,'simpfn,'!:ibfexpr!:simp);
symbolic procedure !:ibfexpr!:simp(u); ('!:ibfexpr!: . u) ./ 1;


symbolic procedure ibfexpr!_prep!*(u);
```

```
   % vorgeschaltete Routine, die eine Typpruefung zur Laufzeit
   % durchfuehrt und ggf. die eigentlich fuer die Wiederberechnung
   % zustaendige Routine aufruft

   if ibfexprp!:(u) then
   <<
      ibfexpr!_prep1(cdr u)
   >>

   else u;


symbolic procedure ibfexpr!_prep1(u);

   begin scalar a, dm, r, i;

      % Den urspruenglichen Ausdruck extrahieren und eine
      % symbolische Vereinfachung vorschalten. Dieser
      % Schritt kann unterbleiben, allerdings werden dann
      % dem System bekannte Identitaeten wie 'sin(pi)=0'
      % nicht mehr angewendet. Im Gegenzug verringert sich
      % dafuer aber die Laufzeit der Routine.

      a := extract!_expr!:(u);

      dm := dmode!*;
      dmode!* := nil;
      r  := reval(a);
      dmode!* := dm;

      % Falls das Resultat eine rationale oder ganze Zahl ist, den
      % Ausdruck verwerfen und statt dessen den vereinfachten
      % Term in das entsprechende Rueckgabeformat konvertieren.

      if eqcar(r,'quotient) and fixp(cadr r) and fixp(caddr r) then
      <<
         r := mkrn(cadr r,caddr r);
         r := '!:ibfexpr!: . (rn2ibf!:(r,!:ibfacc!:) . r);
      >>

      else if not fixp(r) then
      <<
         % Wiederberechnung auf !:ibfacc!: Stellen durchfuehren
```

```
         r := ibfrecompute!:(u,!:ibfacc!:);
         i := car r;

         % Falls erforderlich, eine zweite Wiederberechnung nachschalten

         if not ibfincludes0!:(i) and not ibfaccurate!:(i,!:ibfacc!:) then
            r := ibfrecompute!:(r,!:ibfacc!:);

         r := '!:ibfexpr!: . r;
      >>

      return r;

   end;

% Definition der zum Spezifizieren der signifikanten Stellenzahl
% vorgesehenen Funktion und der von ihr verwendeten Variablen

!:ibfacc!: := 10;  % default value wie bei bigfloat

initdmode 'accurate;


symbolic procedure accuracy n;   % analog zu precision fuer bigfloat's

   if (n < 1) then
      !:ibfacc!:

   else
   <<
      !:ibfacc!: := n;
      n
   >>;

% accuracy als Operator im symbolischen Modus deklarieren

flag('(accuracy),'opfn);

% Liste der verfuegbaren Standardfunktionen und Konstanten

deflist('((fabs ibfexpr!_abs!*)     % Absolutbetrag
          (sqrt ibfexpr!_sqrt!*)    % Quadratwurzel
          (exp ibfexpr!_exp!*)      % Exponentialfunktion
```

```
          (expt ibfexpr!_expt!*)    % allgemeine Potenzfunktion
          (log ibfexpr!_ln!*)       % natuerlicher Logarithmus
          (sin ibfexpr!_sin!*)      % Sinus
          (cos ibfexpr!_cos!*)      % Kosinus
          (tan ibfexpr!_tan!*)      % Tangens
          (asin ibfexpr!_asin!*)    % Arcussinus
          (acos ibfexpr!_acos!*)    % Arcuskosinus
          (atan ibfexpr!_atan!*)    % Arcustangens
          (sinh ibfexpr!_sinh!*)    % hyperbolischer Sinus
          (cosh ibfexpr!_cosh!*)    % hyperbolischer Kosinus
          (tanh ibfexpr!_tanh!*)    % hyperbolischer Tangens
          (asinh ibfexpr!_asinh!*)  % Areasinus
          (acosh ibfexpr!_acosh!*)  % Areakosinus
          (atanh ibfexpr!_atanh!*)  % Areatangens
%
          (e ibfexpr!_e!*)          % Eulersche Zahl
          (pi ibfexpr!_pi!*)),      % Kreiszahl Pi
        '!:ibfexpr!:);

%       ...

endmodule;   % ibfdmain
```

Bemerkung: Die Definition der Arithmetik–Routinen unterbleibt wie bereits erwähnt aus Platzgründen an dieser Stelle.

Die im Modul "`ibfrecmp`" angesiedelte Routine `ibfrecompute(.,.)` überprüft zunächst, ob der Ausdruck bereits der Spezifikation des Wiederberechnungsverfahrens genügt. Ist dies nicht der Fall, wird die Wiederberechnung mit Hilfe einer weiteren globalen Liste mit dem Tag "`!:ibfrec!:`" delegiert. Nur für die Grundrechenarten, das unäre Minus und einige atomare Ausdrücke erledigt sie diese Aufgabe direkt.

```
symbolic procedure ibfrecompute!:(c,k);

   begin scalar  i,r,prec,arg,c1,c2,kond,err,ops,func;
         integer l,n;

      if fixp(k) and (k > 0) then
      <<
         i := car c;   % Intervall
         r := cdr c;   % 'op

         if ibfincludes0!:(i) and leq!:(ibfwidth!:(i),make!:bf(1,(-k))) then
            return c
```

```
else if not ibfincludes0!:(i) and ibfaccurate!:(i,k) then
   return c

else   % Ausdruck hat nicht die geforderte Genauigkeit
<<
   % Rationale Zahlen und Float's werden gesondert behandelt

   if rn!:(r) then
      return (rn2ibf!:(r,k) . r)

   else if eqcar(r,'!:ft!:) or floatp(r) then
   <<
      prec := !:prec!:; !:prec!: := k + 2;
      i := ibfround!:(ft2ibf!:(if floatp(r) then
            mkfloat(r) else r),k);
      !:prec!: := prec;
      return (i . r);
   >>

   else if (r eq 'e) or (r eq 'pi) then
      return (apply1(get(r,'!:ibf!:),k) . r)

   else if eqcar(r,'minus) then  % unary operator !
   <<
      c1 := ibfrecompute!:(cadr r,k);
      i  := ibfminus!:(car c1);

      return (i . list('minus,c1));
   >>

   else if eqcar(r,'plus) then
      ...

   else if eqcar(r,'difference) then
      ...

   else if eqcar(r,'times) then
      ...

   else if eqcar(r,'quotient) then
      ...

   else
```

```
        <<
           func := get(car r,'!:ibfrec!:);

           if func eq nil then
              rederr list("RECOMPUTATION OF ",car r,
                      " IMPOSSIBLE IN IBFRECOMPUTE!:");

           return apply2(func,c,k);
        >>;
     >>;
  >>

  else
     ibferrmsg 'ibfrecompute!:;

end;
```

Bemerkung: Der Code für die Grundrechenarten wurde hier aus Platzgründen nicht wiedergegeben.

Die in der Liste "`!:ibfrec!:`" versammelten Funktionen arbeiten (fast) alle nach dem gleichen Prinzip. Zunächst werden für den vorliegenden Operator oder die Standardfunktion die bei der Wiederberechnung zu erreichenden absoluten bzw. relativen Genauigkeiten bestimmt. Anschließend wird für die ggf. erforderliche Wiederberechnung des Argumentes bzw. der Operanden wieder das globale `ibfrecompute(.,.)` aufgerufen, womit die Rekursion sich vollendet. Für den zugegebenermaßen trivialen Fall des Absolutbetrages sieht die zugehörige Wiederberechnungsroutine wie folgt aus:

```
symbolic procedure rec!_ibffabs!:(c,k);

   begin scalar c1;

      c1 := ibfrecompute!:(caddr c,k);

      return (ibfabs!:(car c1) . list('fabs,c1));

   end;
```

Durch konsistente Erweiterung der beiden Listen "`!:ibfrec!:`" und "`!:ibfexpr!:`" können also zusätzliche Funktionen vollkommen transparent und ohne größeren Aufwand hinzugefügt werden. Wäre der Absolutbetrag dem System nicht bereits bekannt, würden die Anweisungen

```
   put('!:ibfexpr!:, 'fabs, 'ibfexpr!_abs!*);
   put('!:ibfrec!:, 'fabs, 'rec!_ibffabs!:);
```

ihn nahtlos in die Domäne `ACCURATE` integrieren.

3.3 C++

Die Realisierung einer selbstverifizierenden Arithmetik mit Hilfe der Programmiersprache C++ ([Str],[ES]) liegt aus verschiedenen Gründen nahe. Zunächst unterstützt C++ ein Operatorkonzept, das die Verwendung benutzerdefinierter Arithmetiken erheblich vereinfacht, da nicht von der gewohnten Notation abgewichen werden muß. Zusammen mit dem mächtigen Klassenkonzept, das die Implementierung eigener Datentypen und der zugehörigen Operationen mit einer klar definierten Schnittstelle erlaubt, stellt es eine erwägenswerte Alternative zu den nur auf kommerzieller Basis verfügbaren (X)SC–Produkten dar. Dank des unter Beachtung der GNU Public License (kurz GPL) frei erhältlichen C++–Compilers der Free Software Foundation (FSF), ist diese Programmiersprache zudem auf fast allen gängigen Rechnerplattformen verfügbar.
Die eigentliche Implementierung besteht aus zwei schichtartig übereinander liegenden Teilen. Der „untere“ dient der Bereitstellung der für den Wiederberechnungsalgorithmus erforderlichen Datenstrukturen. Darüber liegt eine Schnittstellenklasse `ArithmeticDomainElement` (kurz `ADE`), deren einzige Aufgabe darin besteht, verschiedenste Arithmetiken in transparenter Weise zur Verfügung zu stellen. Sie ist ähnlich einfach zu handhaben, wie das domain interface des Computer–Algebra–Systems `REDUCE` (vgl. vorigen Abschnitt).
Beim Design beider Teile wurde besonderer Wert auf die Beachtung des objektorientierten Paradigmas gelegt. Dies resultiert in der intensiven Verwendung von Vererbung, virtuellen Funktionen und Polymorphie. Konsequenterweise erfolgt auch die — für erfahrene C++–ProgrammiererInnen durchaus mögliche — Erweiterung des Systems durch Ableiten von vorgegebenen Basisklassen.

3.3.1 Die Benutzer–Schnittstelle

Wie schon bei der Beschreibung der `REDUCE`–Implementierung soll zunächst anhand eines kleinen Beispielprogramms die Verwendung des neu geschaffenen Typs `ADE` illustriert werden. Wir betrachten die gleiche Problemstellung, führen aber diesmal die erste Auswertung intervallmäßig (mit der Domäne aus `BIGFLOATINT`) aus, nachdem zuvor die zu verwendende Stellenzahl mit der Funktion `precision()` festgelegt wurde. Anschließend wird zur Domäne `ACCURATE` gewechselt und die Berechnung wiederholt. Die vorher festgelegte Stellenzahl wird jetzt allerdings als Genauigkeitsvorgabe interpretiert, d. h. *jede* Zuweisung erfolgt mit der geforderten Anzahl signifikanter Ziffern. Der Aufruf einer weiteren Funktion `accuracy()` wie in der `REDUCE`–Implementierung ist nicht erforderlich.

```
#include "ADE.h"

int main()
{
    ADE Pi, x;

    precision(31);

    SetActualDomain(BIGFLOATINT);
```

```
    Pi = 4 * atan("1");
    x = exp(Pi * sqrt("163"));
    cout << "x = " << x << endl;

    SetActualDomain(ACCURATE);
    Pi = 4 * atan("1");
    x = exp(Pi * sqrt("163"));
    cout << "x = " << x << endl;

    return 0;
}
```

Bemerkung: Die Konstante π ist für den Typ ADE nicht vordefiniert und muß deshalb explizit berechnet werden.

Das Programm erzeugt die folgende Aussgabe:

```
x = [262537412640768743.9999999999955,262537412640768744.0000000000087]
x = [262537412640768743.9999999999991,262537412640768743.9999999999993] .
```

Bemerkung: Die etwas gröbere Einschließung resultiert aus der internen Verwendung zweier zusätzlicher Schutzziffern und deshalb teilweise entfallender Wiederberechnungsschritte.

3.3.2 Eine Klassenhierarchie für Wiederberechnungsverfahren

Im folgenden soll eine C++–Klassenhierarchie vorgestellt werden, die die Implementierung von Wiederberechnungsverfahren unter Berücksichtigung der zu Beginn dieses Kapitels aufgeführten Designgrundlagen ermöglicht. Im Gegensatz zu der `REDUCE`–Implementierung wird der abstrakte Datentyp Liste hier nicht explizit verwendet. Eine vergleichbare Struktur wird vielmehr implizit durch die geschickte Verschachtelung der beiden C++–Klassen `EvaluatedExpr` bzw. `ArithmeticExpr` generiert.
Diese sind wie folgt definiert:

```
class ArithmeticExpr {

public:

    virtual ArithmeticExpr* clone() const = 0;

    virtual BigFloatInt recompute(BigFloatInt&, long) = 0;

    virtual const char *const ClassName() { return "ArithmeticExpr"; }

    virtual ~ArithmeticExpr() {}
};
```

```
class EvaluatedExpr {

    BigFloatInt value;
    ArithmeticExpr *expr;

public:

    EvaluatedExpr() { expr = NULL; }

    EvaluatedExpr(double d) : value(d)
    {
        expr = (ArithmeticExpr *) new DoubleAtom(d);
    }

    EvaluatedExpr(BigFloatInt ibf, ArithmeticExpr *e) : value(ibf)
    {
        expr = e→clone();
    }

    EvaluatedExpr(const EvaluatedExpr& ee)
    {
        value = ee.value;
        expr = ee.expr→clone();
    }

    EvaluatedExpr& operator=(const EvaluatedExpr&);

    EvaluatedExpr* clone() const
    {
        return new EvaluatedExpr(value, expr);
    }

    BigFloatInt Value() { return value; }

    void recompute(long, int = FALSE);

    ~EvaluatedExpr() { delete expr; }
};
```

Bemerkung: Der Typ `BigFloatInt` stellt eine Intervallarithmetik basierend auf dem dynamischen Gleitpunkttyp `BigFloat` zur Verfügung. Für beide Typen sind die üblichen Operatoren und Standardunktionen (vgl. Kapitel 2) definiert. Der Typ BigFloat ist dem entsprechenden Typ in `REDUCE` nachempfunden und basiert wie dieser auf einem im Prinzip beliebig langen ganzzahligen Typ (genannt `BigInt`), der für die Darstellung der Mantisse verwendet

wird. Aus Effizienzgründen wird für den Exponenten lediglich der Typ `long` herangezogen. Der Typ `BigInt` nutzt die populäre Technik, gerade so viele Zehnerstellen in einem Maschinenwort unterzubringen, daß bei Multiplikation und Addition von Überträgen keine Überläufe auftreten. Die Modulstruktur der bereits etwas in die Jahre gekommenen MIRACL–Bibliothek ([Sc]) diente als Vorbild, um diesen Ansatz einer Quasi–BCD–Arithmetik zu einer vollwertigen C++–Bibliothek auszubauen. In den zugehörigen low level Routinen (Dateien "`bb*.h bb*.C`") wurde ein „sicherer" Zeigertyp verwendet, bei dem es sich um eine Weiterentwicklung des Typs `CheckedPtrToT` von Stroustrup ([Str], Abschnitt 7.10) handelt, der im Original einige gravierende Schwächen aufweist. Diese Pointer werden bei der Konstruktion jeweils an Speicherbereiche bestimmter Größe gebunden. Der zugehörige Dereferenzierungs–Operator überprüft bei jedem Zugriff, ob eine Bereichsverletzung vorliegt, und bricht das Programm ggf. ab. Die Definition der entsprechenden Klasse `smallBCDptr` findet sich in der gleichnamigen Headerdatei. Über die Verwendung „sicherer" oder gewöhnlicher Zeiger kann mittels bedingter Compilierung durch Definition des Makros `CHECKPTR` entschieden werden. Eine Template–Version (`SavePtr<T>`) wurde ebenfalls erstellt, kam aber, da nicht alle Übersetzer dieses Feature bereits voll unterstützen, nicht zur Verwendung. Außerdem wurde in den low level Routinen und bei der Implementierung der Klasse `BigInt` intensiv vom assert–Makro Gebrauch gemacht, um zumindest in Ansätzen Eiffel–ähnliche Vor- und Nachbedingungen zu realisieren (siehe [MeB]). Viele wertvolle Hinweise zu den bei der Implementierung von Langzahl–Arithmetiken auftretenden Fragestellungen finden sich in [Knu], Abschnitt 4. 3 (Multiple–Precision Arithmetic).

Die Klasse `ArithmeticExpr` dient ausschließlich als Basisklasse für weitere abzuleitende Klassen, die die verschiedenen Typen arithmetischer Ausdrücke wie Summen, Differenzen, Produkte, Quotienten, Funktionen und Konstanten repräsentieren. In der oben wiedergegebenen Form erlaubt sie deshalb naturgemäß noch keine Kodierung komplexer arithmetischer Ausdrücke.
Wir wollen nun am Beispiel des relativ einfachen Ausdrucks

$$3.14 + 2.71$$

zeigen, wie durch Ableiten weiterer Klassen von `ArithmeticExpr` komplexe Ausdrücke aufgebaut werden können. In unserem Beispiel sind dies die Klassen `DoubleAtom` bzw. `Sum`, die wie folgt definiert werden können:

```
class DoubleAtom : public ArithmeticExpr {

    double x;

public:

    DoubleAtom(double d = 0.0) : x(d) {}

    ArithmeticExpr* clone() const
    {
```

```
        return (ArithmeticExpr *) new DoubleAtom(x);
    }

    ~DoubleAtom() {}

    BigFloatInt recompute(BigFloatInt&, long);

    virtual const char *const ClassName() { return "DoubleAtom"; }
};


class Sum : public ArithmeticExpr {

    EvaluatedExpr *x, *y;

public:
    Sum() : x(NULL), y(NULL) {}

    Sum(const EvaluatedExpr& v, const EvaluatedExpr& w)
    {
        x = v.clone();
        y = w.clone();
    }

    ArithmeticExpr* clone() const
    {
        return (ArithmeticExpr *) new Sum(*x, *y);
    }

    ~Sum() { delete x; delete y; }

    BigFloatInt recompute(BigFloatInt&, long);

    virtual const char *const ClassName() { return "Sum"; }
};
```

Zusammen mit dem entsprechenden binären Operator `+` für den Typ `EvaluatedExpr`

```
EvaluatedExpr operator+(const EvaluatedExpr& x, const EvaluatedExpr& y)
{
    EvaluatedExpr r;

    r.value = x.value + y.value;
    expr = (ArithmeticExpr *) new Sum(x, y);
```

```
        return r;
    }
```

und einem zusätzlichen Konstruktor

```
    EvaluatedExpr::EvaluatedExpr(double d) : value(d)
    {
        expr = (ArithmeticExpr *) new DoubleAtom(d);
    }
```

stellt das folgende Programmsegment für einen C++–Compiler dann kein Hindernis mehr dar:

```
    EvaluatedExpr x;

    x = EvaluatedExpr(3.14) + 2.71;
```

Der explizite Aufruf des Konstruktors `EvaluatedExpr(double)` ist erforderlich, da der Compiler andernfalls zunächst die beiden `double`-Konstanten addieren und erst bei der Zuweisung ein Objekt vom Typ `EvaluatedExpr` erzeugen würde. Mit dieser kleinen Hilfestellung wird folgender Code generiert:

```
    x.operator=(operator+(EvaluatedExpr(3.14), EvaluatedExpr(2.71)));
```

Das eigentliche Wiederberechnungsverfahren wird über die Memberfunktionen

```
    void EvaluatedExpr::recompute(long k, int top)
    {
        if (includes0(value))
        {
            if (width(value) > BigFloat(1,-k))
                value = expr→recompute(value, k);

            // second recomputation if zero is no longer included
            // and(!) we are at the top level of an expression

            if ((top == TRUE) && (!includes0(value)))
                value = expr→recompute(value, k);
        }

        else if (!accurate(value, k))
            value = expr→recompute(value, k);

    }
```

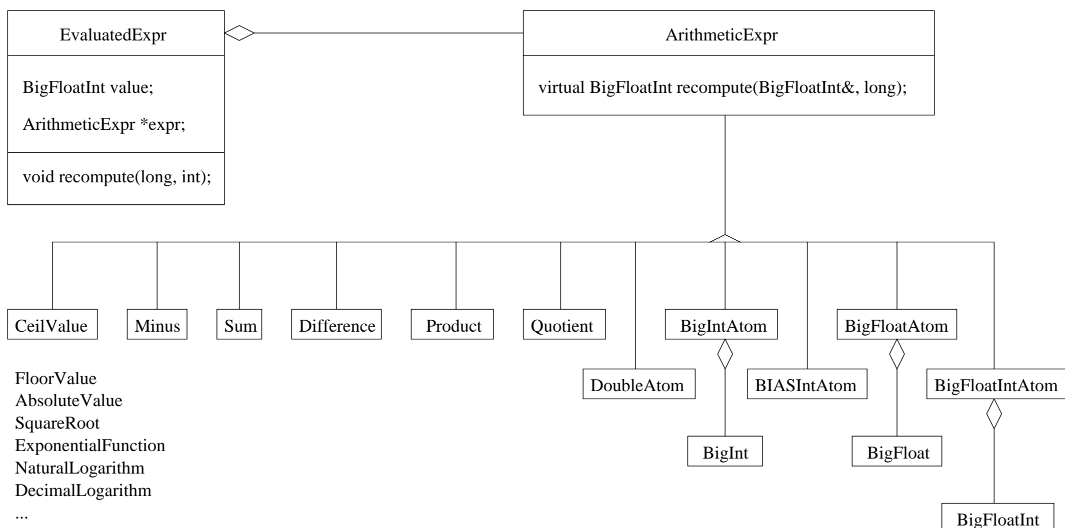
bzw.

BigFloatInt Sum::recompute(BigFloatInt&, **long**);

realisiert. Erstere überprüft lediglich, ob der Wert des arithmetischen Ausdrucks bereits die geforderte Genauigkeit besitzt (vgl. dazu die Spezifikation des Wiederberechnungsverfahrens aus Kapitel 1). Falls dies nicht der Fall ist, wird die Wiederberechnung der beteiligten Operanden bzw. Argumente an die entsprechende — von `ArithmeticExpr` abgeleitete — Klasse delegiert. Dies geschieht durch den Aufruf der virtuellen Memberfunktion `recompute(BigFloatInt&, long)`, die als Rückgabewert den mit der gewünschten Genauigkeit neu berechneten Wert liefert. Die dazu in der Regel erforderlichen Einschließungen des Resultats und der Operanden bzw. Argumente sind als Aufrufparameter `value` und über die Memberfunktion `EvaluatedExpr::Value()` der beteiligten Operanden bzw. Argumente verfügbar.
Das Hinzufügen weiterer Operatoren und Funktionen erfolgt nun vollkommenen analog durch Überladen des entsprechenden Operators bzw. Definition der entsprechenden Funktion für den Typ `EvaluatedExpr` und Ableiten einer geeigneten Klasse von `ArithmeticExpr`. Ist auch für diese die Memberfunktion
`recompute(BigFloatInt& value, long k)` geeignet definiert, integriert sich der neue Operator bzw. die Funktion nahtlos in die bestehende Hierarchie und damit auch in das Wiederberechnungsverfahren.

Die Klassenhierarchie des Typs `EvaluatedExpr` im Überblick

Des besseren Verständnisses halber soll die Klassenhierarchie des Typs `EvaluatedExpr` noch einmal (fast) vollständig in graphischer Form wiedergegeben werden. Dazu wurde die "object model notation" nach Rumbaugh et al. ([Rum]) in einer an die Sprache C++ angepaßten Variante verwendet.



Bemerkung: Die in den nicht atomaren, von `ArithmeticExpr` abgeleiteten Klassen enthaltenen Zeiger auf Objekte vom Typ `EvaluatedExpr` wurden der Übersichtlichkeit halber nicht mehr berücksichtigt.

Bemerkungen zur effizienten Implementierung

Der Aufbau der beteiligten Klassen ist der besseren Verständlichkeit wegen leicht vereinfacht wiedergegeben. Erfahrene C++–ProgrammiererInnen werden sofort bemerken, daß die Implementierung von Copy–Konstruktor und Zuweisungsoperator für den Typ `EvaluatedExpr` besondere Sorgfalt erfordert und insbesondere explizit erfolgen muß. Die andernfalls vom Compiler automatisch generierten Versionen führen nur eine „oberflächliche" Kopie (engl. - swallow copy) durch, ohne auch die „hinter den Zeigern liegenden" Objekte zu duplizieren. Dies wird in der Regel zu Zugriffen auf bereits destruierte Objekte und damit zum Programmabsturz führen (siehe [MeS], Tip 11).
Die einfachste Gegenmaßnahme stellt das generelle „tiefe" Kopieren (engl. deep copy) dar, wobei stets Zeiger auf neu erzeugte Kopien der beteiligten Objekte zugewiesen werden. Sie ist auch in den oben vorgestellten Programmfragmenten dargestellt und verwendet die eigens für diesen Zweck bereitgestellte Memberfunktion `EvaluatedExpr::clone()`, die wiederum entsprechend zu definierende Memberfunktionen der „Erben" von `ArithmeticExpr` aufruft. Dieses sichere Vorgehen führt allerdings zu erheblichen Laufzeiteinbußen, da insbesondere Copy–Konstruktor–Aufrufe von C++–Übersetzern in großer Zahl generiert werden. Effektives Gegensteuern ist aber zum Glück mittels Verwendung sogenannter Handle–Klassen und einer lazy–copy Strategie möglich. Die Handle–Klassen enthalten zusätzlich zum eigentlichen Zeiger einen Referenzzähler, der beim Erstellen einer „oberflächlichen" Kopie inkrementiert und beim Destruieren des zugehörigen Objektes dekrementiert wird. Erst wenn der Zustand des Objektes in einer Weise geändert werden soll, die unerwünschte Seiteneffekte nach sich zöge, wird zuvor mittels deep copy eine identische Kopie (ein Clone) mit eigenem Referenzzähler erzeugt.

3.3.3 Eine C++–Schnittstelle für arithmetische Datentypen

Die in der Überschrift genannte Schnittstellenfunktionen wird — wie sollte es anders sein — von einer C++–Klasse `ArithmeticDomainElement` (kurz ADE) übernommen. Sie hat folgende öffentliche Schnittstelle:

```
class ArithmeticDomainElement {

    DomainImpl *impl;
    DomainType tag;

    // further implementation details omitted

public:

// various constructors:

    ArithmeticDomainElement();
    ArithmeticDomainElement(long, unsigned long d = 1);
    ArithmeticDomainElement(double);
```

```
        ArithmeticDomainElement(const char *);

    // copy constructor:

        ArithmeticDomainElement(const ArithmeticDomainElement&);

    // assignment operator:

        ArithmeticDomainElement& operator=(const ArithmeticDomainElement&);

    // destructor:

        ~ArithmeticDomainElement();

    // unary arithmetic operators:

        friend ArithmeticDomainElement operator+(const
ArithmeticDomainElement&);
        friend ArithmeticDomainElement operator-(const
ArithmeticDomainElement&);

    // binary arithmetic operators:

        friend ArithmeticDomainElement operator+(
            const ArithmeticDomainElement&,
            const ArithmeticDomainElement&);
        friend ArithmeticDomainElement operator-(
            const ArithmeticDomainElement&,
            const ArithmeticDomainElement&);
        friend ArithmeticDomainElement operator*(
            const ArithmeticDomainElement&,
            const ArithmeticDomainElement&);
        friend ArithmeticDomainElement operator/(
            const ArithmeticDomainElement&,
            const ArithmeticDomainElement&);

    // arithmetic assignment operators:

        ArithmeticDomainElement& operator+=(const ArithmeticDomainElement&);
        ArithmeticDomainElement& operator-=(const ArithmeticDomainElement&);
        ArithmeticDomainElement& operator*=(const ArithmeticDomainElement&);
        ArithmeticDomainElement& operator/=(const ArithmeticDomainElement&);

    // comparison operators:
```

```
    friend int operator<(const ArithmeticDomainElement&,
        const ArithmeticDomainElement&);
    friend int operator>(const ArithmeticDomainElement&,
        const ArithmeticDomainElement&);
    friend int operator<=(const ArithmeticDomainElement&,
        const ArithmeticDomainElement&);
    friend int operator>=(const ArithmeticDomainElement&,
        const ArithmeticDomainElement&);
    friend int operator==(const ArithmeticDomainElement&,
        const ArithmeticDomainElement&);
    friend int operator!=(const ArithmeticDomainElement&,
        const ArithmeticDomainElement&);

// miscellaneous functions:

    friend ArithmeticDomainElement ceil(const ArithmeticDomainElement&);
    friend ArithmeticDomainElement floor(const ArithmeticDomainElement&);
    friend ArithmeticDomainElement fabs(const ArithmeticDomainElement&);
    friend ArithmeticDomainElement sqrt(const ArithmeticDomainElement&);
    friend ArithmeticDomainElement log(const ArithmeticDomainElement&);
    friend ArithmeticDomainElement log10(const ArithmeticDomainElement&);
    friend ArithmeticDomainElement exp(const ArithmeticDomainElement&);
    friend ArithmeticDomainElement sin(const ArithmeticDomainElement&);
    friend ArithmeticDomainElement cos(const ArithmeticDomainElement&);
    friend ArithmeticDomainElement tan(const ArithmeticDomainElement&);
    friend ArithmeticDomainElement asin(const ArithmeticDomainElement&);
    friend ArithmeticDomainElement acos(const ArithmeticDomainElement&);
    friend ArithmeticDomainElement atan(const ArithmeticDomainElement&);
    friend ArithmeticDomainElement sinh(const ArithmeticDomainElement&);
    friend ArithmeticDomainElement cosh(const ArithmeticDomainElement&);
    friend ArithmeticDomainElement tanh(const ArithmeticDomainElement&);
    friend ArithmeticDomainElement asinh(const ArithmeticDomainElement&);
    friend ArithmeticDomainElement acosh(const ArithmeticDomainElement&);
    friend ArithmeticDomainElement atanh(const ArithmeticDomainElement&);
    friend ArithmeticDomainElement pow(const ArithmeticDomainElement&,
        const ArithmeticDomainElement&);

// i/o routines:

    friend istream& operator≫(istream&, ArithmeticDomainElement&);
    friend ostream& operator≪(ostream&, const ArithmeticDomainElement&);

// retrieving and setting actual arithmetic domain:
```

```
        friend DomainType GetActualDomain();
        friend DomainType SetActualDomain(DomainType);

    // retrieving and setting precision:

        static long precision();
        static long precision(long);
    };

    typedef ArithmeticDomainElement ADE;
```

Alle üblicherweise für arithmetische Typen (wie etwa `double`) verfügbaren Operatoren und Funktionen sind also zugänglich. Der besondere Nutzen dieses „meta–arithmetischen" Typs verbirgt sich hinter der Funktion `SetActualDomain(DomainType)`. Sie erlaubt die Wahl der verwendeten Datenrepräsentation und der zugehörigen Arithmetik an nahezu beliebiger Stelle innerhalb eines Programms und zwar während der Laufzeit(!). Momentan sind die folgenden arithmetischen Domänen verfügbar:

- `NONE`.
- `IEEEDOUBLE`.
- `BIAS`.
- `RATIONAL`.
- `BIGFLOAT`.
- `BIGFLOATINT`.
- `ACCURATE`.

Der Typ `NONE` ist — wie der Name nahlegt — gültig, falls noch keine Domäne vom Programm aktiviert wurde. Er wird ebenfalls vom Default–Konstruktor vergeben. Bis auf wenige Ausnahmen, die wir später noch detaillierter diskutieren werden, führt jeder Versuch einer Operation mit Variablen dieses Typs zum vorzeitigen Programmabbruch. Zusätzlich wird der Name der fehlgeschlagenen Memberfunktion auf den standard error stream ausgegeben. Die Domäne `BIAS` bietet eine Schnittstelle zu der frei erhältlichen Maschinen–Intervallarithmetik `BIAS` ([Kn1]) und dem zugehörigen C++–Binding `PROFIL` ([Kn2]). `RATIONAL` schließlich ermöglicht das Rechnen mit rationalen Zahlen beliebiger Größe. Die Bedeutung der verbleibenden Domänennamen geht aus den bisherigen Ausführungen hervor.
Es ist möglich, eigene arithmetische Typen zu integrieren. Die dazu nötigen Schritte werden im Rahmen der Darstellung der hinter `ADE` stehenden Klassenhierarchie und ihrer Funktionsweise ersichtlich.

Die Hauptmotivation für eine solche Schnittstellenklasse ist die Tatsache, daß für viele Probleme im "scientific computing" optimal angepaßte aber in der Regel nicht durch Koprozessoren oder andere geeignete Hardware unterstützte arithmetische Typen existieren. Hier sind unter anderem rationale Zahlen im Bereich der Computergeometrie oder selbstverifizierende Arithmetiken in der (kontrollierten) Numerik zu nennen.
Es wäre aus Gründen der Effizienz also naheliegend, diese Arithmetiken nur gezielt in „kritischen“ Programmabschnitten einzusetzen. Was ein „kritisches“ Codesegment ist, hängt im allgemeinen von den aktuellen Werten der involvierten Variablen ab. Diese sind aber eben *erst zur Laufzeit* bekannt, daher die Notwendigkeit, die Arithmetik auch zu diesem Zeitpunkt noch beeinflussen zu können. Ein Beispiel, wie auch klassische numerische Verfahren hiervon profitieren können, findet sich in [Hei1] bzw. [Hei2]. Dank einer einzigen hochgenauen Auswertung einer problemspezifischen Funktion kann nach anschließender ε–Inflation — einer in der kontrollierten Numerik häufig angewendeten Technik — bereits eine sehr gute Einschließung der gesuchten Lösung bestimmt werden.

Implementierung der Arithmetik für den Typ ADE

Auch hier wird erneut die in C++ mögliche Polymorphie ausgenutzt. Die Klasse `ADE` enthält einen Zeiger auf ein Objekt vom Typ `DomainImpl`. Dahinter verbirgt sich aber in der Tat stets ein Objekt einer abgeleiteten Klasse. `DomainImpl` dient wieder ausschließlich als Basisklasse, die die Schnittstelle zu `ADE` festlegt:

```
class DomainImpl {

protected:

    DomainRepr *pdata;

public:

// flag is used to trigger actions (e. g. recomputuation) of derived classes:

    virtual DomainImpl* clone(int flag) const;

// default constructor:

    DomainImpl() { pdata = NULL; }

// virtual destructor:

    virtual ~DomainImpl() {}

// for nicer error messages:

    virtual const char *const ClassName() const { return "DomainImpl"; }
```

```
// conversion to given domain:

    virtual DomainImpl* ToDomainImpl(DomainType);

// unary minus:

    virtual void minus(void);

// arithmetic operators:

    virtual void add(const DomainImpl&);
    virtual void subtract(const DomainImpl&);
    virtual void multiply(const DomainImpl&);
    virtual void quotient(const DomainImpl&);

// comparison operators:

    virtual int less(const DomainImpl&);
    virtual int greater(const DomainImpl&);
    virtual int leq(const DomainImpl&);
    virtual int geq(const DomainImpl&);
    virtual int equal(const DomainImpl&);
    virtual int neq(const DomainImpl&);

// miscellaneous functions:

    virtual void ceil();
    virtual void floor();
    virtual void fabs();
    virtual void sqrt();
    virtual void log();
    virtual void log10();
    virtual void exp();
    virtual void sin();
    virtual void cos();
    virtual void tan();
    virtual void asin();
    virtual void acos();
    virtual void atan();
    virtual void sinh();
    virtual void cosh();
    virtual void tanh();
    virtual void asinh();
```

```
        virtual void acosh();
        virtual void atanh();
        virtual void pow(const DomainImpl&);

    // i/o routines:

        virtual ostream& operator<<(ostream&);
    };
```

Bemerkung: Entgegen der üblichen Praxis sind alle Operatoren und Funktionen nicht als Friends der Klasse realisiert. Nur Memberfunktionen können virtuell sein und damit in abgeleiteten Klassen problemlos überladen werden.

Die Klasse `DomainImpl` kann naturgemäß noch keine Annahmen über die interne Repräsentation der Daten abzuleitender Klassen machen. Sie enthält deshalb einen Zeiger auf ein Objekt vom Typ `DomainRepr`. Dessen Deklaration fordert lediglich die Unterstützung des bereits ausgiebig diskutierten Cloning–Mechanismus.

```
    class DomainRepr {

    public:

    // flag is supplied by DomainImpl, see comment:

        virtual DomainRepr* clone(int flag) const = 0;

        virtual ~DomainRepr() {}

    };
```

Die in Zusammenarbeit mit der Klasse `ADE` wichtigste Aufgabe kommt der Memberfunktion `DomainImpl::ToDomainImpl(DomainType)` zu, die für jede abgeleitete Klasse die Konvertierung in die für `ADE` momentan gültige arithmetische Domäne leistet. Insbesondere erfolgt diese Umwandlung erst wenn es wirklich erforderlich wird und nicht bereits beim Umschalten zwischen verschieden Domänen. Objekte vom Typ `ADE` behalten ihren „internen", durch (Copy–)Konstruktion oder Zuweisung erworbenen Typ bis dieser ggf. durch eine erneute Zuweisung geändert wird. Dadurch werden überflüssige Umwandlungen und die Akkumulation von Konvertierungsfehlern vermieden. Explizite Umwandlungen können aber weiterhin wie oben angedeutet durch Zuweisung forciert werden.

Bemerkung: Die Implementierung dieser Umwandlungsroutinen ist nicht unproblematisch. Insbesondere wenn zwischen Zahldarstellungen zu verschiedenen Basen oder von Intervall– in Punktdomänen konvertiert werden soll, können unerwartete Effekte auftreten. Die im Rahmen dieser Arbeit erstellten Konvertierungen von IEEE– zu BCD–Formaten erfüllen die Anforderungen an eine validierte Arithmetik. Für die umgekehrte Richtung wurden die

`BIAS/PROFIL`–Routinen bzw. die Routinen der C–Standardbibliothek verwendet, die Zeichenketten in den entsprechenden Typ umwandeln. Deren Güte hängt stark vom jeweiligen System ab. Wenn höhere Ansprüche an die Qualität dieser Konvertierungen gestellt werden, können (falls verfügbar) die entsprechenden Routinen der `XSC`–Bibliotheken verwendet werden.

Mit dieser Umwandlungsroutine und den weiteren Memberfunktionen können für `ADE` alle Funktionen und Operatoren leicht definiert werden. Wir wollen dies am Beispiel der Addition diskutieren. Der zu ihrer Implementierung erforderliche Programmcode sieht wie folgt aus:

```
ArithmeticDomainElement operator+(const ArithmeticDomainElement& x,
    const ArithmeticDomainElement& y)
{
    ArithmeticDomainElement z;
    DomainImpl *pdi;
    DomainType D = GetActualDomain();

    z.tag = D;

    if (D != x.tag) z→impl = x→impl→ToDomainImpl(D);
    else z→impl = x→impl→clone(0);

    pdi = y→impl→ToDomainImpl(D);
    z→impl→add(*pdi);
    if (pdi != y→impl) delete pdi;

    return z;
}
```

Die folgenden Schritte werden hier durchgeführt. Zunächst wird falls nötig der erste beteiligte Operand in die aktuelle arithmetische Domäne konvertiert. Sollte dies nicht erforderlich sein, muß dennoch ein Clone des Operanden erzeugt werden, da der nachfolgende Aufruf seiner virtuellen Memberfunktion `DomainImpl::add(const DomainImpl&)` andernfalls unerwünschte Seiteneffekte zeitigen würde. Ist auch der zweite Operand konvertiert worden, kann die Addition erfolgen. Um kein Speicherleck zu hinterlassen, muß nun ggf. das von der Konvertierung des zweiten Operanden herrührende temporäre Objekt destruiert werden. Anschließend wird das Ergebnis zurückgegeben. Für unäre Operatoren und Funktionen mit einem Argument vereinfacht sich die Vorgehensweise geringfügig, da naturgemäß kein zweiter Operand zu betrachten ist. Auch der Ausgabeoperator kann vollkommen analog implementiert werden.

Bemerkung: Der Memberfunktion `clone(int)` wird hier wie in fast allen anderen Fällen der Wert Null übergeben. Die einzige Ausnahme von dieser Regel stellt der Zuweisungsoperator dar, der den Wert Eins verwendet. Bis auf die Klasse `EvaluatedExprDomainImpl` jedoch,

die zur Domäne `ACCURATE` gehört, wird dieser Wert von allen anderen von `DomainImpl` abgeleiteten Klassen ignoriert. Ein von Null verschiedener Wert veranlaßt die Funktion `EvaluatedExprDomainImpl::clone(int flag)` dazu, zusätzlich zum eigentlichen Clonen über die Funktion `EvaluatedExpr::recompute(long, int)` eine Wiederberechnung zu initiieren. Dieser wiederum werden der aktuelle Wert des statischen Memberdatums `prec` von `ADE` als erster und der Wert `TRUE` als zweiter Parameter übergeben.
Der Parameter `flag` wurde eingeführt, um den Einsatz des Wiederberechnungsverfahrens gezielt steuern zu können. Die Alternative, bei jeder Operation der Domäne `ACCURATE` eine Wiederberechnung durchzuführen, wurde aus Gründen der Effizienz verworfen.
Das zweite Argument von `EvaluatedExpr::recompute(long, int)` ist erforderlich, um die unnötige, eventuell erfolgende zweite Wiederberechnung auf tieferen Stufen eines arithmetischen Ausdrucks zu unterbinden. Diese ist gemäß der Spezifikation des Algorithmus nur auf dem obersten Level erforderlich. Daher wurde der Parameter mit dem Default–Wert `FALSE` versehen.

Ausgewählte Designaspekte des Typs ADE

Die im vorigen Abschnitt skizzierte Implementierung der Arithmetik sowie der Vergleichs- und des Ausgabeoperators für den Typ ADE sind "straight forward", wenn man das Zusammenspiel der beteiligten Klassen einmal verinnerlicht hat.
Etwas anders stellt sich die Situation für den Eingabeoperator

```
friend istream& operator>>(istream&, ArithmeticDomainElement&);
```

sowie den Konstruktor

```
ArithmeticDomainElement(const char *);
```

dar. Die Problematik ist in beiden Fällen vergleichbar.
Würde etwa der Eingabeoperator analog zum Ausgabeoperator implementiert, könnte folgende Situation entstehen:

```
int main()
{
    ADE x;

    SetActualDomain(IEEEDOUBLE);
    cin >> x;
}
```

Da die Variable `x` mit dem Defaultkonstruktor konstruiert wurde, hat sie den Typ `NONE`. Das Programm wird also mit einer Fehlermeldung abbrechen, da kein sinnvoller Eingabeoperator verfügbar ist.

Bemerkung: Würde in obigem Beispiel der Ausgabeoperator aufgerufen, wäre die Reaktion die gleiche. Dies ist jedoch gerechtfertigt, da `x` noch keinen gültigen Wert (und damit implizit auch keinen arithmetischen Typ) besitzt.

Ein möglicher Ausweg wäre die Definition einer temporären Variablen – etwa `tmp` – der aktuellen Domäne innerhalb des Eingabeoperators. Daran anschließend müßten der Aufruf der entsprechend überladenen Memberfunktion `tmp.operator≫(istream&)` und die Zuweisung des so erhaltenen Wertes erfolgen.
Der zuvor beschriebene Kunstgriff scheint zunächst auch eine gangbare Lösung für den Konstruktor

```
ArithmeticDomainElement(const char *);
```

zu sein. Hier ist die Situation jedoch noch komplizierter, wie das folgende — leicht abgewandelte — Programmsegment verdeutlicht:

```
int main()
{
    ADE x("1.234");

    SetActualDomain(IEEEDOUBLE);
}
```

Zum Zeitpunkt des Konstruktoraufrufs ist keine arithmetische Domäne gültig. In welchen Typ sollte also die Zeichenkette "`1.234`" umgewandelt werden? Diese Problem stellt sich in ähnlicher Weise bei der Initialisierung globaler Variablen.
Als Fazit ergibt sich leider die Notwendigkeit einen kleinen Stilbruch zu begehen, da das bisher so bewährte Konzept der Polymorphie nicht mehr trägt. Wegen der bereits erwähnten engen Verwandtschaft zwischen dem Einlesen aus Streams und der Konstruktion mittels Zeichenketten wurde auch der Eingabeoperator aus Konsistenzerwägungen in der nachfolgend beschriebenen Weise implementiert. (Für Experten: Genauer könnte im Gegenteil sogar der Konstruktor mit Hilfe des Eingabeoperators und String Streams realisiert werden.)
Die Eingaberoutine analysiert den Input Stream und wählt — nach dem Überspringen führender Trenn- und Leerzeichen (engl. white spaces) — gemäß folgender Vorschrift einen arithmetischen Typ aus:

1. `BIGFLOATINT`, falls als erstes Zeichen '[' erkannt wird.
2. `RATIONAL`, falls nach einem evtl. auftretenden Vorzeichen als erstes nicht–numerisches Zeichen '/' erkannt wird.
3. `BIGFLOAT`, sonst.

Die Nichtberücksichtigung der aktuellen Domäne hat den Vorteil, daß alle Eingaben ohne Verlust von Genauigkeit erfolgen können. Als Nachteil wird dafür erkauft, daß beim Erweitern der von ADE unterstützten Domänen Eingabeformate hinzukommen könnten, die mit keinem der Typen `BIGFLOAT(INT)` oder `RATIONAL` verträglich sind. Die Eingaberoutine müßte also entsprechend angepaßt werden, was einen Verstoß gegen ein wesentliches Paradigma von C++ darstellt. Es besagt, daß Änderungen und Erweiterungen von Klassenhierarchien durch Ableiten und Überladen erfolgen sollten, da direkte Manipulationen an Memberfunktionen die Verfügbarkeit des Quelltextes voraussetzen.

Zusätzliche Funktionen für den Typ ADE

Sollen dem System neue Funktionen hinzugefügt werden, so bestehen dazu prinzipiell zwei Möglichkeiten:

- Die Funktion ist in Termen des Typs `ADE` formulierbar, d. h. mit Hilfe der vorhandenen Funktionen und Operatoren zu implementieren.
- Die Funktion erlaubt dies nicht und muß in die Klassenhierarchie integriert werden.

Natürlich sollte der ersten Alternative unbedingter Vorzug eingeräumt werden. Nur so steht die Funktion auf einen Schlag *für alle* Domänen zu Verfügung, die die zu ihrer Implementierung verwendeten Funktionen und Operatoren unterstützen.

Beispiel: Der Kotangens etwa könnte wie folgt implementiert werden:

```
ADE cot(const ADE& x)
{
    ADE y = sin(x);

    if (!((y < 0) || (y > 0)))
    {
        cerr << "ADE cot(const ADE&):"
            << "argument is multiple of pi" << endl;
        exit(1);
    }

    y = cos(x) / y;
    return y;
}
```

Bemerkung: Die Abfrage, ob eine Nullstelle des Sinus vorliegt, mag auf den ersten Blick etwas umständlich erscheinen. Man sollte sich allerdings bei Verwendungs des Typs `ADE` stets vergegenwärtigen, daß sich dahinter auch Intervall–Domänen verbergen können!

Neue Funktionen in die Klassenhierarchie zu integrieren ist dagegen sehr viel aufwendiger und sollte deshalb nur in begründeten Ausnahmefällen erwogen werden. Selbst wenn die Änderungen auf ein Minimum beschränkt werden, sind zumindest die folgenden Klassen betroffen:

- `ADE`.
- `DomainImpl`.
- von `DomainImpl` abgeleitete Klassen, die die neue Funktion unterstützen sollen.

Die notwendigen Modifikation an den ersten beiden sind eher formaler Natur. Sie gehen aus der Beschreibung des Zusammenspiels der entsprechenden Klassen hervor und sind — einen entsprechenden Editor vorausgesetzt — mittels "copy and paste" nebst minimaler Änderungen zu bewerkstelligen. Der für die von `DomainImpl` abgeleiteten Klassen zu treibende Aufwand fällt dagegen schon stärker ins Gewicht. Für die Domäne `ACCURATE` etwa sind die nötigen Schritte bereits am Beispiel des binären Additionsoperators besprochen worden. Auf alle Fälle ist jedoch eine komplette Neuübersetzung aller Module erforderlich, da sich sowohl die Schnittstelle von `ADE` als auch die Basisklasse `DomainImpl` *aller* anderen Klassen verändert.

Integration weiterer arithmetischer Domänen

Da zu dieser Aufgabenstellung bereits sechs Musterlösungen (darunter auch das Fremdprodukt `BIAS/PROFIL`) vorliegen, dürfte die Integration weiterer Arithmetiken wie etwa der `dotprecision` oder `multiple-precision` Typen aus C–XSC kein Problem darstellen. Lediglich die Existenz einer „sauberen“, den Konventionen für arithmetische Typen entsprechenden, C++–Schnittstelle sollte gesichert sein.
Auf drei unbedingt erforderliche, aber leicht zu übersehende Teilaufgaben eines solchen Portierungsprojektes soll hier abschließend noch hingewiesen werden:

1. Der Enumerationstyp `DomainType` und das statische char–Array `DomainNames` (in "`DomainImpl.h`" bzw. "`DomainImpl.cc`") müssen um den neuen Typ erweitert werden (siehe Kommentar in "`DomainImpl.cc`").

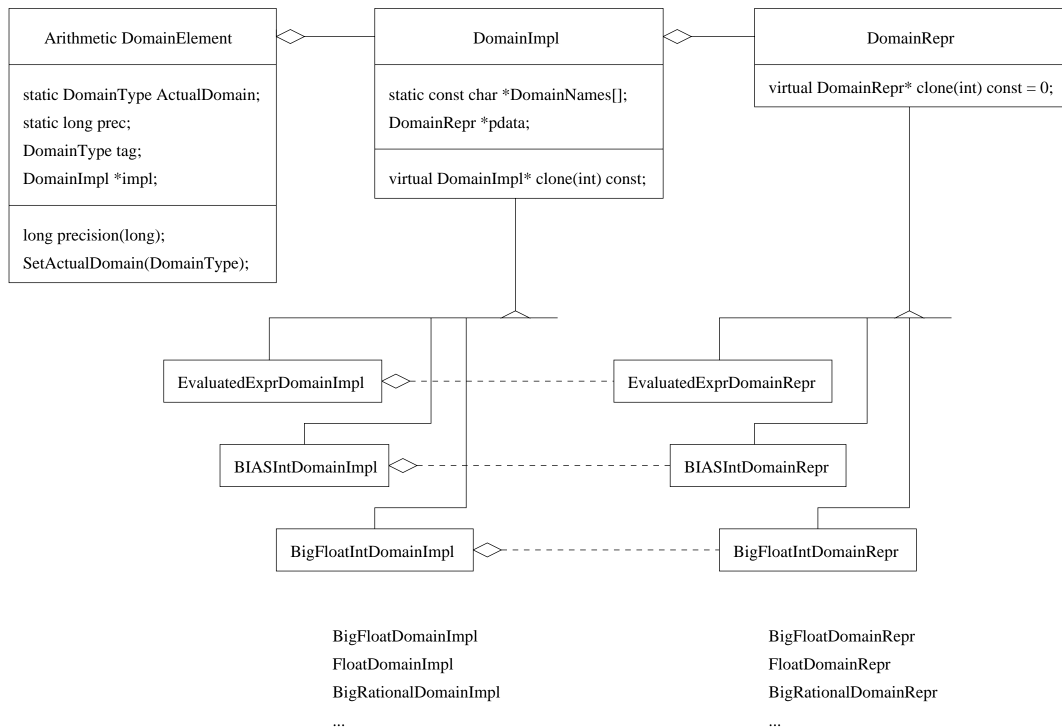
2. Sollen andere Domänen in den neuen Typ konvertierbar sein, muß deren Memberfunktion
`ToDomainImpl` entsprechend angepaßt werden.

3. Falls der neue Typ ein mit `BIGFLOAT(INT)` bzw. `RATIONAL` unverträgliches Eingabeformat besitzt, müssen der Eingabeoperator `ADE::operator<<(istream&, ADE&)` und der Konstruktor
`ADE::ADE(const char *)` geeignet modifiziert werden.

Bemerkung: Sollte der dritte Punkt entfallen, das Eingabeformat des neuen Typs also kompatibel sein, muß unter Punkt 2 unbedingt die „passende“ der in Punkt 3 aufgeführten Domänen in den neuen Typ konvertierbar gemacht werden. Andernfalls werden weder Eingabe noch Konstruktion aus Zeichenketten von der neuen Domäne unterstützt!

Die Klassenhierarchie des Typs `ADE` im Überblick

Zum Abschluß dieses Kapitels soll auch die Klassenhierarchie „hinter“ dem Typ `ADE` noch einmal (fast) vollständig wiedergegeben werden:



Anhang A

Beispiel–Applikationen

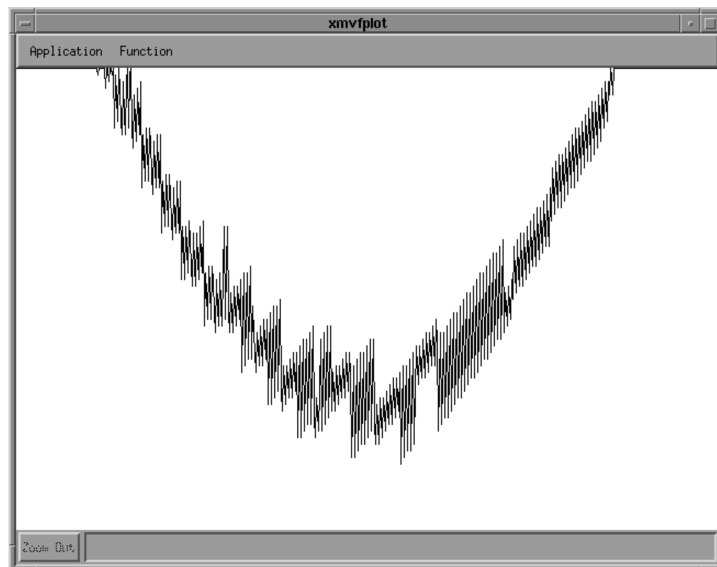
Abschließend sollen zwei OSF/Motif basierte Anwendungen vorgestellt werden, die den meta–arithmetischen Typ `ADE` verwenden. Beide Applikationen wurden mit Hilfe einer leicht modifizierten Version des Application Frameworks von Douglas Young, das in [Yo] entwickelt wird, erstellt. Sie sind beide vorzüglich zur Demonstration vieler Phänomene und Probleme numerischer Rechnungen auf dem Computer geeignet. Außerdem haben sie als Testwerkzeuge für die im Rahmen dieser Arbeit vorgenommenen C++–Implementierungen wertvolle Dienste geleistet.

A.1 Der X11-Funktionsplotter `xmvfplot`

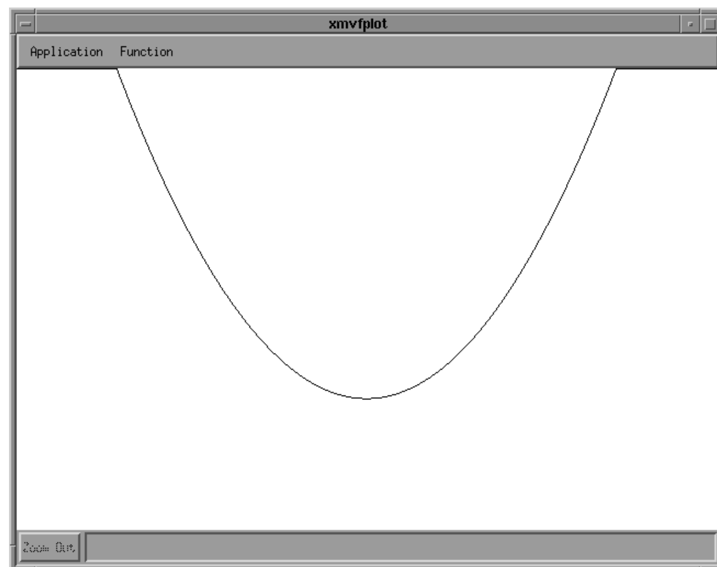
Der 2D–Funktionsplotter `xmvfplot` ist zwar ein einfacher Repräsentant seiner Spezies, unterstützt jedoch die wichtigsten Anforderungen an ein solches Programm. Möglich sind u. a. die interaktive Eingabe der zu zeichnenden Funktion, das Vergrößern von Ausschnitten (mit Rücknahme) sowie die Verfolgung der aktuellen Maus–Koordinaten in einer Statuszeile.
Die wesentliche Neuerung ist allerdings nur während des Eingabe–Dialoges sichtbar und verbirgt sich hinter dem links unten, oberhalb von OK– und CANCEL–Button liegenden Option-Menü. Es erlaubt den Zugriff auf die Funktion `SetActualDomain(DomainType)` und damit die Umschaltung der verwendeten Arithmetik. Lediglich die Unterstützung der Domäne `RATIONAL` wurde nicht realisiert, da für diese keine Standardfunktionen implementiert sind. Die zur Unterstützung der interaktiven Funktionseingabe geschaffene C++–Klasse `FuncEval` erlaubt die Verarbeitung von Funktionsstrings und die (verifizierte) Auswertung für vorgebbare Werte. Die Bedeutung dieser Funktionalität für Einschließungsalgorithmen wird in [Ste] hervorgehoben. Es handelt sich bei dieser Klasse um eine Portierung eines C–Moduls zur „Funktionsberechnung aus Eingabestrings" ([Scha]), der allerdings in der Originalfassung grobe Fehler enthält. Einer der in dem zugehörigen Artikel abgebildeten Funktionsplots paßt ganz offensichtlich nicht zur angegebenen Funktion.
Die beiden folgenden Bilder zeigen einen Plot des Graphen des Polynoms

$$p(x) = 170.4 \cdot x^3 - 356.41 \cdot x^2 + 168.97 \cdot x + 18.601$$

im Bereich $[1.09160791, 1.09160805] \times [-2 \cdot 10^{-13}, 5 \cdot 10^{-13}]$ berechnet unter Verwendung der Domänen `BIGFLOAT` (ein qualitativ vergleichbares Ergebnis liefert auch `IEEEDOUBLE`)



und ACCURATE



In beiden Fällen wurde mit 16–stelliger Arithmetik gerechnet bzw. auf ebenso viele Stellen (automatisch) verifiziert.

Die beiden positiven Nullstellen des obigen Polynoms liegen sehr nahe zusammen, wie die Zerlegung in Linearfaktoren

$$p(x) = 170.4 \cdot (x - \frac{5+\sqrt{35}}{10})(x - \frac{5-\sqrt{35}}{10})(x - \frac{18601}{17040})$$

und die unter Verwendung von ADE berechneten Einschließungen

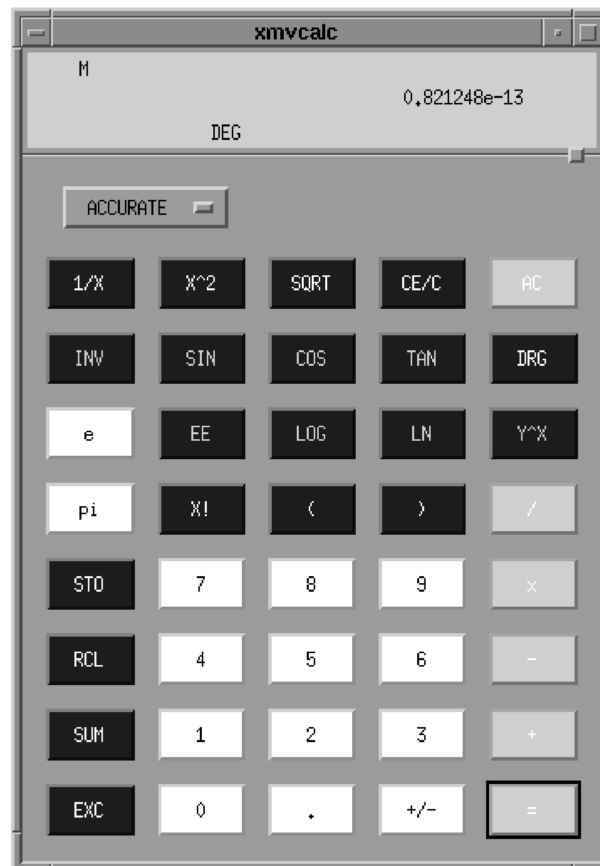
$$\frac{5+\sqrt{35}}{10} = 1.09160797830996^{2}_{1} \quad \text{und} \quad \frac{18601}{17040} = 1.09160798122065^{8}_{7}$$

zeigen.

Bemerkung: Die Zerlegung wurde mit `REDUCE` berechnet, da sowohl *Mathematica* wie auch MAPLE die positiven Nullstellen nicht separieren konnten und eine doppelte Nullstelle bei $x = 1.09161$ vermuteten. Diese Aussage wird jedoch von dem zweiten Plot klar wiederlegt; selbst das erste nicht verifiziert berechnete Bild läßt bereits vermuten, daß dies nicht zutreffen kann. Erst wenn das Polynom explizit mit rationalen Koeffizienten eingegeben wird sind beide System in der Lage, die korrekte Faktorisierung zu berechnen. Es ist leider eine vom Autor häufig gemachte Erfahrung, daß zumindest *Mathematica* bei der Trennung zwischen symbolischem und numerischem Rechnen die nötige Schärfe vermissen läßt. Auch bei der Angabe der Anzahl der signifikanten Stellen eines numerischen Resultats wird offensichtlich eine Heuristik verwendet, die eher zur Überschätzung der tatsächlichen Güte neigt.

A.2 Der X11-Taschenrechner `xmvcalc`

Die X11-Anwendung `xmvcalc` ist, wie das folgende Bild zeigt, ein Clone des bekannten X11-Programms `xcalc`, das auf fast jeder unter X laufenden Workstation verfügbar ist.



Es wurde einem objektorientierten Redesign unterworfen und dabei gleichzeitig an das Young'sche Framework angepaßt. En passant wurden ebenfalls zwei schwere Fehler im TI-Modus (algebraischer Eingabemodus, nur dieser wurde portiert!) des Originals korrigiert.
Natürlich gestattet auch diese Anwendung wieder die Wahl der verwendeten Arithmetik über ein Option–Menü, das sich links unterhalb des Displays befindet. Auch hier wird die Domäne `RATIONAL` aus den bereits angeführten Gründen nicht unterstützt. Die Display–Logik wurde so modifiziert, daß sie bei Intervall–Arithmetiken zur Darstellung von Resultaten ggf. in einen zweizeiligen Modus umschaltet. Die direkte Eingabe von Intervallen ist jedoch nicht möglich!

Bemerkung: Im Display ist das verifizierte Resultat der Auswertung des im vorigen Abschnitt behandelten Polynoms an der Stelle $x = 1.091608$ sichtbar.

Literaturverzeichnis

[ASt] M. Abramowitz, I. A. Stegun, Handbook of Mathematical Functions, Dover Publications, Inc. , New York, 1970.

[AK] E. Adams, U. Kulisch (Editors), Scientific Computing with Automatic Result Verification, Academic Press, San Diego, 1993.

[AH] G. Alefeld, J. Herzberger, Introduction to Interval Computations, Academic Press, New York, 1983.

[B] H. Böhm, Berechnung von Polynomnullstellen und Auswertung arithmetischer Ausdrücke mit garantierter, maximaler Genauigkeit, Dissertation, Universität Karlsruhe, 1983.

[Br] K. Braune, Hochgenaue Standardfunktionen für reelle und komplexe Punkte und Intervalle in beliebigen Gleitpunktrastern, Dissertation, Universität Karlsruhe, 1987.

[Er] F. Erwe, Differential- und Integralrechnung I, Bibliographisches Institut AG, Mannheim, 1962.

[ES] M. A. Ellis, B. Stroustrup, The Annotated C++ Reference Manual, Addison–Wesley, Reading, 1990.

[Fi1] H.–C. Fischer et al. , Evaluation of Arithmetic Expressions with Guaranteed High Accuracy, in [KSH], p. 149–158.

[Fi2] H.–C. Fischer et al. , Auswertung arithmetischer Ausdrücke mit garantierter hoher Genauigkeit, Siemens Forschungs- und Entwicklungsberichte, Nr. 5, 1987, S. 171–177.

[Fi3] H.–C. Fischer, Schnelle automatische Differentiation, Einschließungsmethoden und Anwendungen, Dissertation, Universität Karlsruhe, 1990.

[Ha] R. Hammer, Maximal genaue Berechnung von Skalarproduktausdrücken und hochgenaue Auswertung von Programmteilen, Dissertation, Karlsruhe, 1992.

[HHKR] R. Hammer, M. Hocks, U. Kulisch, D. Ratz, Numerical Toolbox for Verified Computing I: Basic Numerical Problems, Springer, Berlin, Heidelberg, New York, 1993.

[Hea] A. C. Hearn, REDUCE, User's Manual Version 3.3, Rand Corporation, Santa Monica, 1987.

[Hei1] G. Heindl, Inclusion methods in solving engineering problems, Interner Bericht der integrierten Arbeitsgruppe Mathematische Probleme aus dem Ingenieurbereich, Fachbereich Mathematik der BUGH Wuppertal, 1994.

[Hei2] G. Heindl, Inclusion methods in solving engineering problems, in [Her].

[Her] J. Herzberger (Editor), Topics in validated computations: proceedings of IMACS–GAMM International Workshop on Validated Computations, Oldenburg, Deutschland, 30. August – 3. September, 1993.

[Kie] I. Kießling et al. , Genaue Rechnerarithmetik — Intervallrechnung und Programmieren mit PASCAL–SC, Teubner, Stuttgart, 1988.

[Kh] A. Ya. Khinchin, Continued Fractions, Phoenix Books, University of Chicago Press, Chicago and London, 1964.

[Kl1] R. Klatte et al. , PASCAL–XSC, Sprachbeschreibung mit Beispielen, Springer, Berlin, 1991.

[Kl2] R. Klatte et al ., C–XSC A C++ Class Library for Extended Scientific Computing, Springer, Berlin, 1993.

[Kn1] O. Knüppel, BIAS — Basic Interval Arithmetic Subroutines, Technical Report 93.3, Berichte des Forschungsschwerpunktes Informations- und Kommunikationstechnik, TUHH, 1993.

[Kn2] O. Knüppel, PROFIL — Programmer's Runtime Optimized Fast Interval Library, Technical Report 93.4, Berichte des Forschungsschwerpunktes Informations- und Kommunikationstechnik, TUHH, 1993.

[Knu] D. E. Knuth, The Art of Computer Programming, 2 : Seminumerical Algorithms, Addison–Wesley, Reading, Mass. , 1969.

[KM] U. W. Kulisch, W. L. Miranker (Editors), A New Approach to Scientific Computing, Academic Press, New York, 1983.

[KSH] U. Kulisch, H. J. Stetter, J. Herzberger (Editors), Scientific Computation with Automatic Result Verification, Computing Supplementum 6, Springer, Wien, New York, 1988.

[Kr] W. Krämer, Inverse Standardfunktionen für reelle und komplexe Intervallargumente mit a priori Fehlerabschätzungen für beliebige Datenformate, Dissertation, Universität Karlsruhe, 1987.

[Lo1] R. Lohner, Interval Arithmetic in Staggered Correction Format, in [AK], p. 301–321, 1993.

[Lo2] R. Lohner, Precise Evaluation of Polynomials in Several Variables, in [KSH], p. 139–148.

[MeB] B. Meyer, Eiffel: the language, Prentice–Hall, Inc. , New York, 1992.

[MeS] S. Meyer, Effektiv C++ Programmieren: 50 Möglichkeiten zur Verbesserung Ihrer Programme, Addison Wesley, Bonn, München, Paris, . . . , 1992.

[MT] W. L. Miranker, R. A. Toupin, Accurate Scientific Computations, Lecture Notes in Computer Science Nr. 235, Springer, Berlin, 1986.

[Pe] O. Perron, Irrationalzahlen, Göschens Lehrbücherei, de Gruyter, Berlin, 1960.

[Ri] P. L. Richmann, Automatic Error Analysis for Determining Precision, Communications of the ACM, Vol. 15, No. 9, p. 813–817, 1972.

[Rum] J. Rumbaugh, Object–Oriented Modeling and Design, Englewood Cliffs, Prentice Hall, New Jersey, 1991.

[Ru1] S. M. Rump, New Results on Verified Inclusions, in [MT].

[Ru2] S. M. Rump, Solving Algebraic Problems with High Accuracy, in [KM].

[Sa] T. Sasaki, An Arbitrary Precision Real Arithmetic Package in `REDUCE`, in *EUROSAM 1979*, 1979, p. 358–368.

[Scha] D. Schaum, Die Auswertung mathematischer Funktionen in C, ST Journal 1/86, Data Becker Verlag, Düsseldorf, 1986.

[Schu] G. Schumacher, Genauigkeitsfragen bei algebraisch–numerischen Algorithmen auf Skalar- und Vektorrechnern, Dissertation, Universität Karlsruhe, 1989.

[Sc] M. Scott, MIRACL – Multiprecision Integer and Rational Arithmetic C Library, National Institute for Higher Education, Ballymun, Dublin, Ireland, 1987, `ftp://ftp.compapp.dcu.ie/pub/crypt/other/miracl-3.23.zip`.

[Ste] H. J. Stetter, Inclusion Algorithms with Functions as Data, in [KSH], p. 213–224.

[Str] B. Stroustrup, Die C++ Programmiersprache, Addison–Wesley, Bonn, München, Paris, . . . , 1992.

[Stoe] J. Stoer, Einführung in die Numerische Mathematik I, 3. Auflage, Springer, 1979.

[Wi] J. H. Wilkinson, Rounding Errors in Algebraic Processes, Prentice–Hall, Inc. , New York, 1963.

[Wo] St. Wolfram, *Mathematica* — Ein System für Mathematik auf dem Computer, 2. Auflage, Addison–Wesley, Bonn, München, Paris, . . . , 1992.

[Yo] D. Young, Object Oriented Programming with C++ and OSF/Motif, Prentice–Hall, Inc. , New York, 1992.