Schnelle hoch genaue Einschließung von Werten arithmetischer Ausdrücke mit beliebiger vorgegebener Genauigkeit

von
Axel Rogat

Dissertation
zur Erlangung des Grades eines Doktors der Naturwissenschaften

vorgelegt dem Fachbereich Mathematik
der Bergischen Universität Gesamthochschule Wuppertal

Juli 2002

Einleitung

Die vorliegende Arbeit beschäftigt sich mit der Entwicklung und der Implementation (in Form einer `C++`-Klassenbibliothek) eines Algorithmus, der reelle arithmetische Ausdrücke mit beliebiger vorgegebener Genauigkeit auswertet. Als Grundlage dafür wird außerdem eine BCD-Intervall-Arithmetik für Grundrechenarten und Standardfunktionen mit beliebiger dynamischer Genauigkeit entwickelt.

Unter einem (reellen) arithmetischen Ausdruck in m Variablen verstehen wir zunächst informell eine Abbildung $F : D \to \mathbb{R}$, $D \subset \mathbb{R}^m$, die sich rekursiv aus endlich vielen arithmetischen Grundrechenarten und Standardfunktionen zusammensetzt. Eine exakte Definition über Codelisten findet sich später in Kapitel 5.

Zu entwickeln und zu implementieren ist ein Algorithmus, der den exakten Wert des Ausdrucks, $y = F(x_1, \ldots, x_m)$, verifiziert einschließt, derart dass die Einschließung $[y]$ in einem noch exakt zu definierenden Sinn auf eine beliebige Stellenzahl p „hoch genau“ ist. Grob gesprochen soll das bedeuten, dass die Grenzen der Einschließung Maschinenzahlen mit p Stellen (zu einer gegebenen Basis b) sind und sich erst in der letzten Stelle um 1 oder maximal 2 unterscheiden.

Ein verifizierter Einschluss kann, sofern keine Überläufe oder Überschreitungen von Definitionsbereichen auftreten, durch eine übliche Auswertung mit einer Maschinen-Intervallarithmetik garantiert werden. Typischerweise hat man auf dem Rechner nur eine Arithmetik mit fest vorgegebener Maximalgenauigkeit p zur Verfügung. Durch Rundungsfehler wird die so gelieferte Einschließung in den seltensten Fällen mit p Stellen hoch genau sein, und für jedes $p' \leq p$ lässt sich ein Ausdruck angeben, der mit dieser Arithmetik nicht mit der Genauigkeit p' berechnet werden kann.

Mit der auf dem Rechner verfügbaren Arithmetik kann man iterativ Näherungslösungen bestimmen, beispielsweise unter Verwendung von Methoden zur Lösung von Gleichungssystemen (nicht-linearen, wenn beliebige Standardfunktionen vorkommen dürfen), siehe z.B. [Fischer]. Diese Algorithmen können aber nicht garantieren, dass die gewünschte Genauigkeit auch letztlich erreicht wird (wenn nicht doch, explizit oder implizit, eine Arithmetik beliebiger Genauigkeit, beispielsweise durch staggered correction, verwendet wird).

Wenn man eine Familie von ineinander eingebetteten Arithmetiken mit unterschiedlichen Genauigkeiten zur Verfügung hat oder besser sogar eine Arithmetik, die mit beliebiger vorgegebener Genauigkeit arbeitet, kann man alle Operationen, die im Ausdruck vorkommen, mit einer festen größeren Stellenzahl berechnen. Dabei stellt sich aber das Problem, dass man bei beliebigen Operationen und unbeschränkten Bereichen für die Wertebereiche der Operanden keine a-priori-Abschätzungen angeben kann, die die Bestimmung einer Stellenzahl $p' > p$ ermöglichen würden, welche ein auf p Stellen hoch genaues Ergebnis garantiert. Ggf. müsste bei diesem einfachen Ansatz also die Stellenzahl so lange schrittweise erhöht und der gesamte Ausdruck mit ihr neu berechnet werden, bis Hochgenauigkeit erreicht ist. Wenn nur einige Teilausdrücke die hohe Stellenzahl erzwingen, werden die anderen dabei unnötig präzise bestimmt.

Die hier benutzte, zuerst in [Steins] vorgestellte Methode ist ein Wiederberechnungsverfahren. Es verwendet die Einschließungen aller vorkommenden Zwischenergebnisse während ei-

nes ersten normalen intervallmäßigen Auswertungsvorgangs, um für einen zweiten Durchgang vorab abschätzen zu können, um wie viel genauer die jeweiligen Teilausdrücke im Hinblick auf ein hoch genaues Gesamtergebnis berechnet werden müssen. Es handelt sich nicht um ein iteratives Verfahren; es ist normalerweise nur ein einziger zweiter Auswertungsvorgang notwendig. Als Ausnahmen ergeben sich aus technischen Gründen der Fall, dass während dieses zweiten Schritts eine zuvor eingeschlossene Null ausgeschlossen werden kann (und Fälle, in denen auf Benutzerwunsch Teilausdrücke verfeinert werden, um eine Bereichsüberschreitung des Arguments einer Funktion zu vermeiden).

Ein wesentliches Ziel der vorliegenden Arbeit war es, eine deutliche Beschleunigung des Gesamtverfahrens zu erzielen, da die Implementation zu [Steins] teilweise schon sehr lange Laufzeiten benötigte. Am eigentlichen Algorithmus zur Wiederberechnung wurden keine wesentlichen Modifikationen vorgenommen. Dagegen konnte einiges an den Implementationsdetails und besonders an der unterliegenden dynamischen Intervallarithmetik verbessert werden. Außerdem wurden einige kleine Fehler behoben, die teilweise hoch genaue Ergebnisse verhinderten. Sowohl der Auswertungsalgorithmus, wie auch die gesamte dafür benötigte dynamische Intervallarithmetik wurden im Rahmen dieser Arbeit als `C++`-Klassenbibliothek vollständig neu implementiert. Zu [Steins] war auch eine Implementation zur Verwendung mit dem Computer-Algebrasystem REDUCE erstellt worden, worauf hier verzichtet wurde. Ebenso wurde die dort von REDUCE her in die `C++`-Umgebung übertragene Auswahlmöglichkeit der unterliegenden Arithmetik *zur Laufzeit* nicht implementiert; aus Geschwindigkeitsgründen, und da sie keine wesentlichen Vorteile mit sich bringt. Design-Entscheidungen wie diese werden in den entsprechenden Abschnitten ausführlich besprochen.

Die Performance-Steigerungen beruhen auf dem Einsatz ganz unterschiedlicher Techniken auf den verschiedenen Ebenen. Bei den Grundoperationen wurden beispielsweise schnellere Methoden (etwa eine rekursive Integer-Multiplikation und eine verbesserte Intervall-Multiplikation) verwendet. Außerdem wurde dort viel technisches Feintuning betrieben, unter anderem durch Implementation spezieller Varianten für den Fall typischer Gegebenheiten auf dem Rechner, durch Einsatz vorberechneter Tabellen, binäre Fallunterscheidungen und mehr.

Bei der Berechnung transzendenter Konstanten über Kettenbrüche wird eine Mindestanzahl notwendiger Iterationsschritte vorab geschätzt, so dass diese Anzahl von Schritten ohne Divisionen und ohne Genauigkeitstest ablaufen kann. Außerdem wurden, wo sinnvoll, mehrere Iterationsschritte geclustert, was die Verschiebung von Berechnungen von der Langzahl- in die normale Integer-Arithmetik erlaubt. Bei der Berechnung von Standardfunktionen über Potenzreihen wird eine Mindestanzahl notwendiger Glieder aus einer Tabelle interpoliert, was die sonst lange Aufmultiplikationsphase von Potenzen und Fakultäten abkürzt und fast immer ganz umgeht. Das bei den Standardfunktionen verwendete Horner-Verfahren wurde – bei den Funktionen, die dies zulassen – durch Einbeziehen der Koeffizienten des Approximationspolynoms in die Horner-Klammerung modifiziert und beschleunigt. Für einige Funktionen war es sinnvoll, spezielle Routinen bei großen bzw. sehr kleinen Argumenten einzusetzen. Statt ein Intervall-Newton-Verfahren bei der Quadratwurzel zu verwenden, kommen wir mit einem einfachen Newton-Verfahren und einer nachgeschalteten Testquadrierung aus. Einige Funktionen werden nun schneller über eine Potenzreihe berechnet, die bei [Steins] mit dem

Newton-Verfahren (Logarithmus) oder über algebraische Identitäten mit anderen Funktionen (Arcussinus) berechnet wurden. Intervall-Sinus und -Cosinus kommen nun in einigen Fällen mit nur einer Punktauswertung aus, wo zuvor noch zwei nötig waren. Neu und bei [Steins] nicht implementiert sind coth, arccot, arcoth, root (n-te Wurzel), abs (für $\mathbb{R}^2$ bzw. $\mathbb{C}$) und arg ($\mathbb{C}$-Polarwinkel, in Programmiersprachen manchmal `atan2` genannt). Schließlich wurde an einigen Stellen sinnvoll die vorhandene (Hardware-, IEEE-) Arithmetik eingesetzt, beispielsweise für einen schon auf 16 Stellen genauen Startwert für das Newton-Verfahren bei der Quadratwurzel und bei Oberschranken für x^N bei den Restgliedabschätzungen.

Bei der Berechnung arithmetischer Ausdrücke konnten Verbesserungen dadurch erzielt werden, dass der bei der Neuberechnung eines Operanden einer binären Operation möglicherweise gewonnene Spielraum der Berechnung des anderen Operanden zugute kommen gelassen wird. Außerdem werden dort die Berechnungen notwendiger Stellenzahlen selbst nur mit der gebotenen Genauigkeit durchgeführt.

Die Gewinnfaktoren bewegen sich (im anvisierten Stellenbereich von 10 bis 50) zwischen 2 und 4 (Grundrechenarten), 5 bis 30 (Standardfunktionen), 15 bis 50 (Auswertung von Ausdrücken); Ausnahmeerscheinungen bei der Einschließung von π und e sind Faktoren bis 1000.

An nicht geschwindigkeitsbezogenen Verbesserungen ist z.B. Folgendes zu nennen. Die Spezifikation der Zwischenergebnisse beim Wiederberechnungsverfahren war zu ändern, da sonst in bestimmten Fällen falsche Ergebnisse produziert werden müssen. Die beiden Berechnungsdurchgänge des Verfahrens sind bei uns nun mit unterschiedlichen Genauigkeiten durchführbar, die der Benutzer auf Wunsch angeben kann. (Je nach Beschaffenheit des Ausdrucks kann es günstiger sein, zunächst schnell und ungenau zu rechnen, da der zweite Schritt ohnehin fast alle Zwischenergebnisse neu berechnen muss, oder bereits im ersten Durchgang besonders genau, da eine gute Chance besteht, den zweiten Durchgang zu vermeiden.) Es ist vom Benutzer einstellbar, dass auf mögliche Bereichsüberschreitungen bei der Auswertung von Ausdrücken automatisch durch einen Verfeinerungsversuch des betreffenden Operanden reagiert wird (Beispiele im entsprechenden Kapitel zeigen, wie sinnvoll dies meist sein kann.) Arithmetische Ausdrücke können aus Zeichenketten (z.B. Benutzereingaben) geparst werden (eine solche Funktionalität war im Steins'schen Plot-Programm enthalten, aber nicht allgemein in der Bibliothek zugänglich), und dort sind Wertänderungen vorkommender Variabler möglich, ohne dass der Ausdruck neu aufgebaut werden muss.

In einer Konfigurationsdatei kann die Arithmetik (zur Compilationszeit) konfiguriert und an die Gegebenheiten des jeweiligen Rechners angepasst werden (z.B. 32 vs. 64 Bit, Geschwindigkeit vs. Speicherbedarf, überlaufgesicherte Exponenten oder nicht, Exception-Handling oder nicht). Dort kann auch gewählt werden, ob die Fließkommazahlen immer ganz, nur teilweise oder (aus Geschwindigkeitsgründen) nie normalisiert werden sollen. Fließkomma-Additionen und Subtraktionen werden nicht mehr intern exakt berechnet und anschließend gerundet, was bei [Steins] bei Operanden stark unterschiedlicher Größenordnung zu extrem langen Mantissen, Speichermangel und Abstürzen führen konnte.

Diese Arbeit entwickelt die notwendigen mathematischen Grundlagen und Verfahren und stellt zugleich die entstandene Implementation vor. Bei Letzterem kann natürlich nicht auf

Details eingegangen werden; es werden aber die zu Grunde liegenden Design-Entscheidungen und der Aufbau der Bibliothek dargestellt, sowie einige spezielle verwendete Verfahren behandelt. Da das Verhalten der Arithmetik spätere Entscheidungen bei Standardfunktionen und Gesamt-Algorithmus beeinflusst, orientiert sich der Aufbau der Arbeit am Aufbau der Bibliothek aus ihren Einzelteilen im „Bottom-Up"-Sinn. Daher werden wir auf sinnvolle Weise zwischen mathematisch und an der Implementation orientierten Teilen hin und her wechseln.

Im ersten Kapitel werden zunächst einige benötigte Hilfsmittel, beispielsweise einige Resultate aus der Intervall-Analysis, vorgestellt und in der ganzen Arbeit verwendete Schreibweisen eingeführt. Danach werden Datenformate und grundlegende Implementationsentscheidungen der Fließkomma- und Intervallarithmetik besprochen. Das darauf folgende Kapitel behandelt, notwendigerweise sehr ausführlich, die verwendeten Methoden zur Approximation der Standardfunktionen. Zum Schluss wenden wir uns ausführlich dem Wiederberechnungsverfahren und letztlich dessen Implementation zu.

Abschließend sei bemerkt, dass diese Arbeit im Wesentlichen nach den Regeln der Rechtschreibreform gesetzt wurde, wodurch sich auch für einige fachliche Begriffe wie „hoch genau" eine ungewohnte Schreibung ergab.

Mein Dank gilt meinem Doktorvater Prof. Dr. G. Heindl für die gute Zusammenarbeit in den vergangenen Jahren und für Unterstützung und Verbesserungsvorschläge beim Erstellen dieser Arbeit, und Herrn Prof. Dr. H.-J. Buhl für Themenstellung und die Übernahme des Korreferats. Weiterhin danke ich Andreas Steins, der die Grundlagen entwickelte, auf denen ich hier aufbauen konnte, insbesondere für das Überlassen seiner Quellcodes, und meinem guten Freund Oğuz Varol für Zuhören, Korrekturlesen und Beta-Testen.

Wuppertal, im Juli 2002 Axel Rogat

Inhaltsverzeichnis

1 Grundlagen

1.1 Gleitpunktraster

Ein **Gleitpunktsystem** $S(b, \ell, e_{min}, e_{max}) \subset \mathbb{Q}$ zur Basis $b \geq 2$ mit $\ell \geq 1$ Mantissenstellen und Exponentenbereich von e_{min} bis e_{max} ist wie üblich wie folgt definiert:

$$S(b, \ell, e_{min}, e_{max}) := \left\{ \sigma \cdot b^e \cdot \sum_{k=1}^{\ell} a_k b^{-k} \;\middle|\; \begin{array}{l} \sigma \in \{-1, +1\},\ e \in \mathbb{Z},\ e_{min} \leq e \leq e_{max}, \\ a_k \in \mathbb{N}_0,\ 0 \leq a_k < b,\ a_1 \neq 0 \end{array} \right\} \cup \{0\}.$$

Wir wollen nur normalisierte Zahlen, also solche im obigen Sinn, zulassen. Der Summenteil einer Zahl x liegt in $[b^{-1}, 1-b^{-\ell}] \subset [b^{-1}, 1[$ und wird als Mantisse $\mathrm{mant}(x)$ bezeichnet; e heißt Exponent (auch „Charakterisktik“ $\mathrm{char}(x)$), σ Vorzeichen der Zahl $\mathrm{sign}(x)$.

In dieser Arbeit werden wir immer von Systemen $S(b, k) = S(b, k, -\infty, +\infty)$ ausgehen, also ohne Beschränkung des Exponentenbereichs. In der implementierten Arithmetik kann je nach Einstellung der größte *Exponent* ca. 10 hoch 8 Milliarden sein, was für alle praktischen Zwecke ausreichen dürfte. Wenn ein Über- oder Unterlauf auftritt, wird eine Exception „Overflow“ geworfen, auf die das *Benutzerprogramm* geeignet reagieren muss. Es wird also bei Unterläufen nie auf 0 gerundet; ebenso wenig gibt es Pseudo-Gleitpunktzahlen wie $\pm\infty$ oder `NaN` in der IEEE-Definition, die in Berechnungen weiter verwendet werden dürften. Man beachte aber die in Abschnitt 2.6.2 beschriebenen Effekte.

Wir arbeiten immer mit einer *Familie* solcher Gleitpunktsysteme, mit beliebiger, theoretisch unbeschränkter Mantissenlänge:

$$S(b) := \bigcup_{\ell \geq 1} S(b, \ell).$$

Die enthaltenen Gleitpunktsysteme fester Stellenzahl werden manchmal auch als ℓ-stellige Gleitpunktraster bezeichnet, die Gesamtfamilie als *dynamisches* Raster.

Zu beachten ist noch Folgendes: Jedes $x \in \mathbb{R} \setminus \{0\}$ hat eine eindeutige b-adische Darstellung als ein Grenzwert

$$x = \lim_{n \to \infty} \left(\pm\, b^e \cdot \sum_{k=1}^{n} a_k b^{-k} \right),$$

mit $0 \leq a_k < b$, $a_1 \neq 0$, und so, dass es kein $N \in \mathbb{N}$ gibt, so dass $a_k = b - 1$ für alle $k \geq N$ gilt. Es ist dennoch nicht $S(b) = \mathbb{R}$, da jedes Element aus $S(b)$ Element eines speziellen $S(b, \ell)$ ist und $S(b)$ somit nur rationale Zahlen enthält, deren b-adische Darstellung zwar beliebig lang, aber endlich ist.

Offenbar gilt $S(b, \ell_1) \subseteq S(b, \ell_2) \Leftrightarrow \ell_1 \leq \ell_2$ (d.h. $S(b, \ell_2)$ „ist feiner“ als $S(b, \ell_1)$). Für ein $x \in S(b)$ sei

$$\mathrm{prec}(x) = \min\{\ell \mid x \in S(b, \ell)\},$$

die Mindestzahl von Stellen, die notwendig sind, um x (exakt) darstellen zu können. Für zur Basis b nicht endlich darstellbare Zahlen x kann man $\mathrm{prec}(x) := \infty$ definieren.

In der realen Implementation wird für die interne Darstellung der Mantissenlänge der Datentyp `long int` verwendet, der sie typischerweise auf $2^{31} - 1$ beschränkt (lange vorher dürfte allerdings üblicherweise ein Speicherüberlauf stattfinden). Es wird gegebenenfalls wiederum eine Exception geworfen.

Für die Zwecke unserer theoretischen Betrachtungen werden wir Exponentenbereich und den Bereich für die Mantissenlängen als unbeschränkt auffassen.

Für $x \in S(b,p) \setminus \{0\}$ bezeichne $\mathrm{succ}_p(x) = \min\{y \in S(b,p) \mid y > x\}$ den Nachfolger in $S(b,p)$, analog $\mathrm{pred}_p(x) = \max\{y \in S(b,p) \mid y < x\}$ den Vorgänger. Man beachte, dass diese beiden bei unbeschränktem Exponentenbereich nie 0 sind.

Die Menge aller *abgeschlossenen* Intervalle in $D \subseteq \mathbb{R}$ sei als ID bezeichnet, insbesondere schreiben wir $I\mathbb{R}$. Analog zu Obigem sind Intervallsysteme und -Raster zu verstehen:

$$IS(b,\ell) = \{\ [a,b] \subset \mathbb{R} \mid a,b \in S(b,\ell)\ \}, \qquad IS(b) = \{\ [a,b] \subset \mathbb{R} \mid a,b \in S(b)\ \}.$$

Auf Implementationsebene können Ober- und Untergrenze eines Intervalls tatsächlich unterschiedlich lange Mantissen haben, beispielsweise durch Normalisierung nach arithmetischen Operationen (Entfernen von Nullen am Ende).

1.2 Fehlerkalkül

1.2.1 Rundungen und gerichtet gerundete Operationen

Für eine reelle Zahl

$$x = \sigma \cdot b^e \cdot \sum_{k=1}^{\infty} a_k b^{-k}$$

schreiben wir

$$\mathrm{trunc}_{\,\ell}\,(x) = \sigma \cdot b^e \cdot \sum_{k=1}^{\ell} a_k b^{-k} \in S(b,\ell)$$

für die nach der ℓ-ten Stelle ($\ell \geq 1$) abgebrochene Darstellung (*truncation*).

Eine **Rundung** auf ℓ Stellen ist eine Projektion $p_\ell : \mathbb{R} \to S(b,\ell)$, wobei man normalerweise fordert, dass es zwischen x und $p_\ell(x)$ keine weiteren Punkte in $S(b,\ell)$ gibt.

Wie üblich verwenden wir die drei typischen Rundungen $\bigtriangledown_\ell(x) := \max\{\underline{x} \in S(b,\ell) \mid \underline{x} \leq x\}$ nach unten, $\bigtriangleup_\ell(x) := \min\{\overline{x} \in S(b,\ell) \mid x \leq \overline{x}\}$ nach oben und $\bigcirc_\ell$ zur nächsten Gleitpunktzahl; $\bigcirc_\ell(x) \in \{\bigtriangledown_\ell(x), \bigtriangleup_\ell(x)\}$, so dass $|\bigcirc_\ell(x) - x| = \min\{|\bigtriangledown_\ell(x) - x|, |\bigtriangleup_\ell(x) - x|\}$. (Bei identischem Abstand muss eine gesonderte Vereinbarung getroffen werden. Üblicherweise wird die betragsgrößere Zahl verwendet, „Rundung nach außen“, oder ggf. eine statistisch ausgewogene Verteilung bevorzugt, etwa bei Binärdarstellung die Rundung auf eine Zahl mit gerade vielen binären Einsen).

Da wir meistens mit Einschließungen in Intervalle arbeiten, werden wir es hauptsächlich mit den beiden ersten „gerichteten“ Rundungen zu tun haben.

Zu exakten arithmetischen Operationen $\bullet : S(b) \to \mathbb{R}$, $\circ : S(b) \times S(b) \to \mathbb{R}$ (bzw. ggf. mit geeigneten Teilmengen von $S(b)$) gibt es zugehörige gerichtet gerundete Operationen in das ℓ-stellige Raster, formal durch die nachgeschaltete Rundungsoperation (auch wenn das nicht der Implementation entsprechen mag oder kann). Wir schreiben $x \mathbin{\overline{\circ}_\ell} y := \nabla_\ell(x \circ y)$, analog $\mathbin{\underline{\circ}_\ell}$, $\mathbin{\overline{\bullet}_\ell}$, $\mathbin{\underline{\bullet}_\ell}$. (Die implementierten Fassungen der Grundrechenarten und der Wurzel liefern diese Ergebnisse. Dass wir uns bei den anderen Standardfunktionen ggf. mit etwas weniger begnügen, wird an geeigneter Stelle behandelt.)

Für den relativen Fehler bei ∇_ℓ für ein $x > 0$, $x \notin S(b, \ell)$, gilt

$$|\varepsilon_\nabla| = \left|\frac{\nabla_\ell(x) - x}{x}\right| = \left|\frac{\mathrm{trunc}_\ell(x) - x}{x}\right| = \sum_{k=\ell+1}^{\infty} a_k b^{-k} \Big/ \sum_{k=1}^{\infty} a_k b^{-k} \le \frac{b^{-\ell}}{b^{-1}} = b^{-\ell+1}.$$

Analoges gilt für $x < 0$ bzw. $\triangle_\ell$. (Für $\bigcirc_\ell$ ist $|\varepsilon_\circ| \le \lfloor \frac{b}{2} \rfloor b^{-\ell}$, für $b = 10$ also $|\varepsilon_x| \le \frac{1}{2} \cdot 10^{-\ell+1}$.)

Man beachte, dass wir immer die Existenz einer solchen *globalen* Schranke für die relativen Fehler bei Rundungen und arithmetischen Operationen voraussetzen werden. Durch Werfen einer Exception bei Über- und Unterläufen wird die Entscheidung in Problemfällen (sinnvollerweise) auf den Benutzer verschoben.

In den beiden Arbeiten [Braune] und [Krämer] wird eine an das Landau-Symbol angelehnte „Fehlerglied"-Schreibweise für den relativen Fehler eingeführt:

- Die Fehlerschranke $\varepsilon(\ell) := b^{-\ell+1}$ für ein ℓ-stelliges Gleitpunktraster zur Basis b ist eine obere Schranke für den relativen Fehler bei Gleitpunktoperationen wie Rundungen oder Grundrechenarten.
- Analog zur Schreibweise $x = \mathcal{O}(n)$ schreibt man dann $x = \varepsilon_\ell$, falls es ein $\delta \in \mathbb{R}$ mit $|\delta| \le 1$ gibt, so dass $x = \delta \cdot \varepsilon(\ell)$.

In diesen beiden Arbeiten wird auf diese Weise eine Intervallschreibweise wie $[-\varepsilon(\ell), +\varepsilon(\ell)]$ (bzw. entsprechende Behandlung von Absolutbeträgen) umgangen. Es muss nicht für jeden speziellen Fall ein konkretes (möglichst ungünstiges) δ angegeben werden.

Wie schon in [Steins] werden wir dagegen die konkretere, wenn auch manchmal umständlichere Intervallschreibweise verwenden, die Autor und Leserschaft vertrauter sein wird. Sie hat aber vor allem den Vorteil, dass Ergebnisse mit Hilfe der implementierten hoch genauen Intervallarithmetik (für festes b und ℓ) direkt nachkontrolliert werden können. Beispielsweise konnte so der Auswertefehler bei der Berechnung von Standardfunktionen über Potenzreihen auf dem Rechner eingeschlossen und so die Güte der theoretischen Ergebnisse ermittelt werden. Etwa wurde so offensichtlich, dass die ursprünglichen Abschätzungen beim Kosinus recht grob waren, woraufhin sie durch zusätzliche Terme verbessert wurden.

Definition: Das **relative Fehlerintervall** für $S(b, \ell)$ sei

$$E_\ell := [-b^{-\ell+1}, +b^{-\ell+1}].$$

Damit gilt für alle $x \in \mathbb{R}$

$$\nabla_\ell(x) \in x \cdot (1 + E_\ell), \qquad \triangle_\ell(x) \in x \cdot (1 + E_\ell),$$

und auch für die gerichtet gerundeten Grundrechenarten $\circ \in \{+, -, *, /\}$ (die ja „maximal genau“ implementiert sind)

$$x \bigtriangledown_\ell y \in (x \circ y) \cdot (1 + E_\ell), \qquad x \bigtriangleup_\ell y \in (x \circ y) \cdot (1 + E_\ell).$$

1.2.2 Einschließungen

Eine Funktion $f : D \to \mathbb{R}$, $D \subseteq \mathbb{R}$, liefert rein mengentheoretisch eine Funktion $If : ID \to \wp(\mathbb{R})$ über $If(I) := \{f(x) \mid x \in I\}$. Falls f auf D stetig ist, nimmt f auf jedem $I \in ID$ Minimum und Maximum (und alle Zwischenwerte) an, d.h. wir haben eine Intervallfunktion $If : ID \to \mathbb{IR}$. Entsprechend sind auch Bezeichnungen wie $|I|$ zu verstehen.

Eine Einschließung einer reellen Größe x ist ein (abgeschlossenes) Intervall $I \subset \mathbb{R}$ mit $x \in I$, eine Einschließung durch Maschinenzahlen ein Intervall $I' \subset S(b)$ mit $x \in I'$. Durch Rundung der Grenzen nach außen (in ein geeignetes ℓ-stelliges Raster) kann man aus einer allgemeinen Einschließung eine Einschließung in Maschinenzahlen erhalten.

Wir wollen im Weiteren für solch eine (allgemeine) Einschließung auch $[x]$ schreiben, wobei entweder eine beliebige Einschließung von x gemeint ist oder eine spezielle, deren genaue Grenzen dem jeweiligen Zusammenhang zu entnehmen sind. Unter- und Obergrenze seien als $\underline{x}, \overline{x}$ bezeichnet, oder auch als $[x]_{\inf}, [x]_{\sup}$. (Die letztere Schreibweise macht viele Beziehungen deutlich lesbarer als $\inf([x])$, $\sup([x])$.)

Solche Einschließungen sind bei uns üblicherweise der Raum, in dem sich Approximationen an x bewegen können. Diese Näherungen sind mit absoluten bzw. relativen Fehlern gegenüber x behaftet. Entsprechend definieren wir diese Fehler für Einschließungen als die maximalen Fehlerbeträge:

Definition: Eine Einschließung $[x] \in \mathbb{IR}$ trägt einen

- absoluten Fehler, den Durchmesser $\operatorname{diam}[x] = \overline{x} - \underline{x}$,
- labelFehler!relativerrelativen Fehler $\varepsilon([x]) = \dfrac{\operatorname{diam}[x]}{|[x]|_{\inf}}$ (falls $0 \notin [x]$).

Eine reelle Zahl x kann offenbar wie folgt in ein Intervall $[x]_\ell \in IS(b, \ell)$ eingeschlossen werden:

$$[x]_\ell \;:=\; [\bigtriangledown_\ell x\,,\, \bigtriangleup_\ell x] \;\subseteq\; x \cdot (1 + E_\ell).$$

Analog erfolgt die ℓ-stellige Einschließung des Ergebnisses einer Grundrechenart $\circ \in \{+, -, *, /\}$ wie folgt:

$$[x \circ y]_\ell := \left[x \bigtriangledown_\ell y, x \bigtriangleup_\ell y\right] \subseteq (x \circ y) \cdot (1 + E_\ell).$$

Diese Intervalle enthalten maximal zwei Punkte aus $S(b, \ell)$, und genau dann einen, wenn x bzw. $x \circ y \in S(b, \ell)$.

Definition: Eine Funktion $f : D \to \mathbb{R}$, $D \subseteq \mathbb{R}$, heißt durch $f_\ell : \tilde{D} \to IS(b, \ell)$, $\tilde{D} \subseteq D \cap S(b)$, als Maschineneinschließung implementiert, wenn $f(x) \in f_\ell(x)$ für alle $x \in \tilde{D}$. Dabei wird $\tilde{D}$ nicht genau festgelegt, sollte aber natürlich möglichst groß sein.

Es ist zu beachten, dass eine beliebige Funktion intervallmäßig *nicht* notwendigerweise derart implementierbar ist, dass die Ergebnisintervalle jeweils maximal 2 Elemente enthalten.

Man kann einen beliebig kleinen relativen Fehler fordern; wenn die resultierende Einschließung sehr eng gerade einen ℓ-Rasterpunkt echt einschließt, wird die ℓ-Rundung drei Punkte enthalten, den Vorgänger und den Nachfolger des eingeschlossenen Rasterpunkts. Darüber hinaus gibt es Fälle, in denen auch mit beliebig hoher Stellenzahl keine Verbesserung möglich ist.

Ein simples Beispiel ist das aus [Krämer] (Seite 13). Wir betrachten $f : \mathbb{R} \to \mathbb{R}$, $f(x) = (\frac{1}{3}) \cdot x$. Die Implementationen sind natürlich so zu verstehen, dass vorkommende Konstanten mit beliebiger Genauigkeit eingeschlossen werden können. Es soll hier also nicht mit einer Division durch 3 gearbeitet werden. Jede derartige Implementation wird angewandt auf die Stelle $x = 3$ ein Intervall mit mindestens drei Elementen liefern. Gleichgültig, mit wie viel Stellen ℓ Genauigkeit die Einschließung $[\frac{1}{3}]_\ell$ bestimmt wird, die Untergrenze ist kleiner, die Obergrenze größer als $\frac{1}{3}$. Damit ist das Produkt mit $x = 3$ kleiner bzw. größer als 1, und die Rundungen nach außen müssen eine Zahl kleiner bzw. größer 1 produzieren.

Dasselbe Phänomen tritt beispielsweise bei Ausdrücken mit Funktion und Umkehrfunktion auf, etwa $\exp(\ln(x))$ (in diesem Fall für $x \neq 1$), bei transzendenten Konstanten, etwa $\sin(\pi)$, etc. Es ist aber natürlich nicht auf diese Fälle beschränkt.

Angesichts dessen kommen wir zu folgenden Bezeichnungen:

Definition: Eine Implementation durch Maschineneinschließungen $f_\ell : \tilde{D} \to IS(b, \ell)$ von $f : D \to \mathbb{R}$, heißt auf ℓ Stellen

- **hoch genau**, wenn $f_\ell(x)$ für alle $x \in \tilde{D}$ mit $f(x) \neq 0$ höchstens **drei** Elemente aus $S(b, \ell)$ (und nicht die Null) enthält,
- **maximal genau**, wenn $f_\ell(x)$ für alle $x \in \tilde{D}$ mit $f(x) \neq 0$ höchstens **zwei** Elemente aus $S(b, \ell)$ (und nicht die Null) enthält.

Man beachte, dass für $f(x) \neq 0$ diese Intervalle nicht die 0 enthalten (bei unbeschränktem Exponentenbereich). Analog heißen Maschinenintervalle I, die die Bedingungen in der Definition erfüllen, hoch genau bzw. maximal genau. Wir wollen für dieses Prädikat die Schreibweise $accurate_p(I)$ für maximale Genauigkeit bzw. $accurate_p^2(I)$ für Hochgenauigkeit verwenden.

Obiges sind Forderungen an den relativen Fehler und würden für $f(x) = 0$ (bei unbeschränktem Exponentenbereich) keinen Sinn machen. Für die Implementation unseres Wiederberechnungs-Algorithmus werden wir dort fordern, dass, falls die Null in der berechneten Einschließung enthalten ist, der *absolute* Fehler durch $b^{-\ell}$ beschränkt ist.

Im Hinblick auf die Erzeugung bzw. die Verwendung von hoch genauen Maschineneinschließungen wollen wir vorab die folgenden einfachen, aber wichtigen Tatsachen festhalten:

Lemma 1 *a) Eine auf ℓ Stellen hoch genaue Einschließung hat einen relativen Fehler von maximal $2 \cdot b^{-\ell+1}$.*

b) *Eine Einschließung, deren relativer Fehler maximal $b^{-\ell}$ ist, erzeugt bei anschließender Rundung auf ℓ Stellen eine auf ℓ Stellen hoch genaue Einschließung.*

Beweis: Die betrachtete Einschließung sei $[\underline{x}, \overline{x}]$, o.B.d.A. sei $[x] > 0$.

a) Zunächst mögen beide Grenzen dieselbe Charakteristik c haben. Innerhalb des Bereichs dieser Charakteristik haben die Punkte des ℓ-stelligen Rasters den Abstand $b^{-\ell}\, b^c$. Also ist der Fall $\underline{x} = m \cdot b^c,\ \ \overline{x} = (m + 2b^{-\ell}) \cdot b^c$ zu betrachten, und es gilt

$$\frac{\overline{x} - \underline{x}}{\underline{x}} \;=\; \frac{2b^{-\ell} \cdot b^c}{m \cdot b^c} \;=\; \frac{2b^{-\ell}}{m} \;\leq\; \frac{2b^{-\ell}}{b^{-1}} \;=\; 2b^{-\ell+1}.$$

Wenn die Grenzen unterschiedliche Charakteristik haben (c sei die Charakterisktik der Obergrenze), ist nur der Fall $\underline{x} = (1 - b^{-\ell}) \cdot b^c,\ \ \overline{x} = (1 + b^{-\ell+1}) \cdot b^c$ zu betrachten, in dem gilt

$$\frac{\overline{x} - \underline{x}}{\underline{x}} \;=\; \frac{b^{-\ell+1} - b^{-\ell}}{1 - b^{-\ell}} \;=\; \frac{1 + b^{-1}}{1 - b^{-\ell}} \cdot b^{-\ell+1}.$$

Der Quotient ist für $b \geq 3$ oder $\ell \geq 2$ nach oben durch 2 beschränkt, und diese Voraussetzungen sind für alle sinnvollen Arithmetiken erfüllt.

b) Die Behauptung folgt daraus, dass eine Einschließung $[\underline{x}, \overline{x}]$ mit relativem Fehler $\varepsilon \leq b^{-\ell}$ maximal einen Punkt des ℓ-stelligen Rasters enthalten kann. c sei die Charakteristik der Untergrenze, dann haben die in Frage kommenden Rasterpunkte mindestens den Abstand $b^{-\ell}\, b^c$, aber

$$\varepsilon \;\leq\; b^{-\ell} \quad\Rightarrow\quad \overline{x} - \underline{x} \;\leq\; \underline{x} \cdot b^{-\ell} \;\leq\; (1 - b^{-\ell}) b^c \cdot b^{-\ell}. \qquad \square$$

Hier liegt eine Quelle für Flüchtigkeitsfehler bei späteren Abschätzungen. Wenn ein Schritt in unserem Wiederberechnungsalgorithmus oder die Auswertung einer Standardfunktion ein auf ℓ Stellen hoch genaues Ergebnis liefern soll, muss der gesuchte Wert mit einem relativen Fehler von maximal $b^{-\ell}$ eingeschlossen werden. Wenn wir dagegen ein hoch genaues Ergebnis eines anderen Schritts verwenden, können wir nur von einem relativen Fehler von $2b^{-\ell+1}$ ausgehen.

1.2.3 Beziehungen aus der Intervall-Arithmetik

Wir werden in späteren Kapiteln relativ komplexe Beziehungen zur Fortpflanzung der bei Rundungen entstehenden Fehler betrachten müssen. Dabei ist bei Operationen mit den Fehlerintervallen Vorsicht geboten, da man leicht fälschlicherweise Beziehungen wie das Distributivgesetz auf Intervalle anwenden könnte. Wir wollen daher hier kurz einiges zusammenstellen, was wir des öfteren verwenden werden (siehe z.B. auch [AlHe]).

- Für $a \in \mathbb{R},\ B, C \in \mathbb{IR}$ gilt das Distributivgesetz

$$a(B + C) = aB + AC, \tag{I_1}$$

denn es ist

$$\begin{aligned} a(B + C) \;&= \{a(b + c) \mid b \in B,\ c \in C\} \\ &= \{ab + ac \mid b \in B,\ c \in C\} \\ &= \{ab \mid b \in B\} + \{ac \mid c \in C\} \;= aB + aC. \end{aligned}$$

- Für beliebige Intervalle $A, B, C \subseteq \mathbb{R}$ gilt dagegen nur das Subdistributivgesetz

$$A(B+C) \subseteq AB + AC, \tag{I_2}$$

was daran liegt, dass auf der rechten Seite das Intervall A zweimal unabhängig voneinander vorkommt, also:

$$\begin{aligned} A(B+C) &= \{a(b+c) \mid a \in A,\ b \in B,\ c \in C\} \\ &= \{ab + ac \mid a \in A,\ b \in B,\ c \in C\} \\ &\subseteq \{a'b + a''c \mid a', a'' \in A,\ b \in B,\ c \in C\} \\ &= \{a'b \mid a' \in A,\ b \in B\} + \{a''c \mid a'' \in A,\ b \in C\} \quad = AB + AC. \end{aligned}$$

Es gilt nicht einmal unbedingt Gleichheit, wenn B und C Punktintervalle sind, d.h. es gilt nur $(b+c)A \subseteq bA + cA$ für $b, c \in \mathbb{R}$.

Gleichheit gilt z.B. dann, wenn B und C beide keine negativen oder beide keine positiven Punkte enthalten:

$$A(B+C) = AB + AC \qquad \text{für } B, C \geq 0 \text{ oder } B, C \leq 0 \tag{I_2'}$$

Sei z.B. $B \geq 0$, $C \geq 0$ (analog für $B \leq 0$, $C \leq 0$).

Für $A \geq 0$ gilt: $A(B+C) = [A_{\inf}(B+C)_{\inf}, A_{\sup}(B+C)_{\sup}]$
$= [A_{\inf}(B_{\inf} + C_{\inf}), A_{\sup}(B_{\sup} + C_{\sup})] = [A_{\inf}B_{\inf}, A_{\sup}B_{\sup}] + [A_{\inf}C_{\inf}, A_{\sup}C_{\sup}]$
$= AB + AC$.

Für $A \leq 0$ gilt $A(B+C) = -((-A)(B+C)) = -((-A)B + (-A)C) = AB + AC$.

Für $0 \in A^\circ$ gilt $A(B+C) = A \cdot (B+C)_{\sup} = A \cdot B_{\sup} + A \cdot C_{\sup} = AB + AC$.

- Für ein symmetrisch um 0 gelegenes Intervall A gilt

$$BA = |B|A. \tag{I_3}$$

Das ist klar, denn, falls 0 im Innern von A, ist immer $BA = bA$ mit $b := \max\{|b_1|, |b_2|\}$. Insbesondere ist das Produkt wieder symmetrisch.

- Für ein symmetrisch um 0 gelegenes Intervall A gilt

$$AB \pm AC = A(|B| + |C|). \tag{I_4}$$

Wegen $AB - AC = AB + (-AC) = AB + (-A)C = AB + AC$ brauchen wir (I_4) nur für die Addition zu zeigen. Für $B, C \geq 0$ folgt dies aus (I_2'), ebenso für $B, C \leq 0$ über
$AB + AC = (-A)B + (-A)C = A(-B) + A(-C) = A|B| + A|C| = A(|B| + |C|)$.

Für $0 \in B$, $C \geq 0$ (analog $B \leftrightarrow C$) hat man mit $b := \max\{|b_1|, b_2\}$
$A(|B| + |C|) = A([0,b] + [c_1, c_2]) = A([c_1, b + c_2]) = (b + c_2)A = bA + c_2A = AB + AC$.

Für $0 \in B$, $C \leq 0$ (analog $B \leftrightarrow C$) folgt daraus auch
$AB + AC = AB + (-A)C = AB + A(-C) = A(|B| + |C|)$.

Falls $0 \in B$, $0 \in C$, gilt mit $c := \max\{|c_1|, c_2\}$
$A(|B| + |C|) = A([0,b] + [0,c]) = A[0, b+c] = (b+c)A = bA + cA = AB + AC$.

Für die (symmetrischen) relativen Fehlerintervalle E_ℓ und $x, y \in \mathbb{R}$ gilt daher

$(I_3) \Rightarrow$
$$xE_\ell = |x|E_\ell, \tag{E_1}$$

$(I_4) \Rightarrow$
$$xE_\ell + yE_\ell = (|x| + |y|)E_\ell, \tag{E_2}$$

$$E_k E_\ell = E_{k+\ell-1}, \tag{E_3}$$

$$(x \pm yE_k)E_\ell = xE_\ell \pm yE_k E_\ell. \tag{E_4}$$

Letzteres ist offenbar richtig für $y = 0$. Ansonsten gilt für $x \geq 0$

$$(x \pm yE_k)E_\ell \;=\; |x \pm yE_k|E_\ell \;=\; (x + |y|b^{-k+1})E_\ell = xE_\ell + |y|b^{-k+1}E_\ell \;=\; xE_\ell \pm yE_k E_\ell,$$

bzw. analog für $x < 0$

$$(x \pm yE_k)E_\ell \;=\; (x - |y|b^{-k+1})E_\ell \;=\; -(|x| + |y|b^{-k+1})E_\ell \;=\; xE_\ell \pm yE_k E_\ell.$$

2 Die dynamische Intervall-Arithmetik

2.1 Design-Entscheidungen

Unser Algorithmus zur hoch genauen Auswertung arithmetischer Ausdrücke benötigt eine Intervall-Arithmetik beliebiger dynamischer Genauigkeit. Damit sind sowohl die Grundrechenarten und andere grundlegende Operationen (Vergleiche, Rundungen, etc.) gemeint, wie auch der komplette Satz der gewünschten Standardfunktionen, auch für beliebig große, kleine oder von der Mantisse her lange Argumente.

Die Genauigkeitsforderungen an den gesamten Ausdruck haben üblicherweise höhere Genauigkeitsforderungen an die Teilausdrücke zur Folge, und es kann selbst bei Beschränkung der Gesamtgenauigkeit keine Schranke für die zwischenzeitlich notwendigen Stellenzahlen angegeben werden. Erstens darf der Ausdruck beliebig tief verschachtelt sein; zweitens gibt es Stellenforderungen, die beliebig mit der Größe der beteiligten Operanden wachsen (etwa bei trigonometrischen Funktionen oder Ausdrücken wie $\cosh^2(x) - \sinh^2(x) = 1$ bei großen Argumenten).

- Eine erste Forderung an die Arithmetik ist also die (im Rahmen der Gegebenheiten auf dem Rechner) unbeschränkte Mantissenlänge.

Außerdem kann eine Auswertung mit beliebig hoher Genauigkeit nur dann gewährleistet sein, wenn zumindest einmal die Eingangsdaten exakt vorliegen. Eine Störung dieser Daten hätte üblicherweise eine Minimalstörung des gesamten Ausdrucks zur Folge, die nicht unterschritten werden könnte.

- Da die Eingangsdaten in den meisten Fällen dezimal vorliegen dürften, ist eine zweite Forderung also, dass die Arithmetik zur Basis 10 (oder ggf. Potenzen von 10) arbeitet.

Erwünscht, aber nicht unbedingt notwendig ist auch ein unbeschränkter oder jedenfalls sehr großer Exponentenbereich.

Bereits bestehende Arithmetiken (etwa die aus der GNU multiple precision library) arbeiten aus Geschwindigkeitsgründen zur Basis 2 und sind deshalb nicht für uns verwendbar. Im Rahmen dieser Arbeit wurde eine eigene dynamische dezimale Arithmetik implementiert, die im Weiteren vorgestellt werden soll.

Zur Arbeit [Steins] wurde bereits eine solche Arithmetik implementiert. Bei ersten eigenen Tests stellte sich aber heraus, dass sehr deutliche Performance-Gewinne möglich sein würden. Die Datenstrukturen sind daher (aus Kompatibilitätsüberlegungen zu Beginn) praktisch mit denen der Steins'schen Version identisch. Die Routinen der Arithmetik wurden aber vollständig neu geschrieben.

Als Arithmetik-Typen werden folgende bereitgestellt:

`BigInt`	„beliebig“ große ganze Zahlen,
`BigRational`	rationale Zahlen (Nenner und Zähler aus `BigInt`),
`BigFloat`	Fließkommazahlen mit variabler, praktisch unbeschränkter Mantissenlänge und großem Exponentenbereich (die Mantissen werden mit `BigInt` dargestellt),
`BFInterval`	Intervalle (mit Grenzen aus `BigFloat`),
`BigComplex`	komplexe Zahlen (Real- und Imaginärteil aus `BigFloat`; bisher nur ansatzweise implementiert, insbesondere nur die simplen Standardfunktionen),
`BCInterval`	komplexe Intervalle (Rechtecke in der komplexen Ebene).

Sie können selbstverständlich alle eigenständig, außerhalb des Auswertungsalgorithmus verwendet werden. Bei den Fließkommatypen ist zu beachten, dass Addition, Subtraktion und Multiplikation auf Wunsch (d.h. mit den normalen Operatoren, ohne Angabe einer Stellenforderung) *exakt* durchgeführt werden – Vorsicht, Addition und Subtraktion können dann schnell extrem lange Mantissen erzeugen. Ansonsten ist bei jeder Operation die gewünschte Stellenzahl des Ergebnisses anzugeben. Die Grundrechenarten und die Quadratwurzel sind maximal genau, die anderen Standardfunktionen hoch genau implementiert. Die komplexen Klassen sind im Hinblick auf eine spätere Erweiterung des Auswertungsalgorithmus auf das Komplexe implementiert worden. Aus Zeitgründen konnten aber von den Standardfunktionen bislang nur die einfachsten implementiert werden.

Bemerkung: Wir weisen hier darauf hin, dass der Auswertungsalgorithmus in der Implementation zu [Steins] auch mit anderen Arithmetiken als der dynamischen betrieben werden kann. Noch zur Laufzeit können alternativ die auf dem Rechner ggf. in Hardware vorhandene (typischerweise IEEE-kompatible) Arithmetik, die selbst implementierte Arithmetik mit fester Stellenzahl ohne Wiederberechnung (mit und ohne Intervalle) oder die Fremdbibliotheken FILIB (Fast Interval Library) und BIAS (Basic Interval Arithmetic Subroutines) ausgewählt werden. Diese Umschaltmöglichkeit zeigt die ursprüngliche Herkunft der Steins'schen Implementation aus der Umgebung des Algebrasystems REDUCE. Sie bedingt allerdings natürlich bei jeder Einzeloperation innerhalb des Ausdrucks eine Umlenkung auf die passende Methode des eingestellten Arithmetik-Typs und damit eine leichte Verlangsamung der Auswertung.

Keines dieser anderen Formate kann aber eine hoch genaue Auswertung gewährleisten. IEEE und die eigene Arithmetik ohne Intervalle liefern gar keine Einschließungen. Die Intervall-Formate sind auf eine feste Stellenzahl beschränkt – die eigene Intervall-Arithmetik auf eine vorher angebbare feste, aber beliebig große Stellenzahl, und FILIB und BIAS überhaupt fest auf üblicherweise ca. 16 Dezimalstellen.

Gedacht ist das Konzept umschaltbarer Arithmetik für die Situation, dass in einem Großteil eines numerischen Verfahrens normale (Intervall-)Arithmetik völlig ausreicht und erst in der Endphase hoch genau ausgewertet werden muss. Der Übergang ist dann durch einen einzigen Befehl möglich, während der Rest des Verfahrens unberührt bleiben kann.

Es gibt nun mehrere Vorgehensweisen. Möglicherweise ist das Verfahren ohnehin zweigeteilt, beispielsweise in eine schnelle Berechnung eines Startwerts für ein verfeinerndes Intervallverfahren. Dann ist diese Umschaltmöglichkeit nicht notwendig. Oder es werden tatsächlich

innerhalb einer Iterationsschleife irgendwann höhere Genauigkeiten erforderlich. Dann kann eine entsprechende Fallunterscheidung auf dieser Ebene, also vor dem Auswerten des betreffenden Ausdrucks geschehen (was den Code natürlich weniger elegant macht). Bei der Implementation zu [Steins] findet diese Entscheidung transparent für den Benutzer, aber an jeder Einzelstelle des Ausdrucks statt.

- Es war daher eine frühe Design-Entscheidung, den Auswertungsalgorithmus für arithmetische Ausdrücke *immer* mit der selbst implementierten *beliebig genauen Intervall-Arithmetik* (mit fester beliebig großer Stellenzahl) oder mit der *dynamischen Intervall-Arithmetik* (mit garantiertem hoch genauen Endresultat) arbeiten zu lassen.

Genauer bedeutet das Folgendes: Der Algorithmus arbeitet mit zwei Durchgängen (Passes). Im ersten wird der Ausdruck bei jeder Operation mit fester Genauigkeit ausgewertet, in der zweiten ggf. so weit verfeinert wie notwendig. Man kann wahlweise nur den ersten Pass auslösen (das liefert eine garantierte, aber nicht notwendigerweise hoch genaue Einschließung) oder beide. Genaueres dazu in Abschnitt 5.8 zur Implementation des Algorithmus.

Geschwindigkeitstests zu den Arithmetiken sind in Kapitel 6 aufgeführt. Da einzelne Ergebnisse vorab in den einzelnen Kapitel genannt werden, sei hier vorab Folgendes erwähnt: Die Tests erfolgten auf zwei unterschiedlichen Systemen – auf einem Intel-L*i*nux-PC (Pentium II, 400 MHz) und einer SUN Ultra Sparc Workstation (333 MHz), wobei sich bei direkt vergleichbarem 32-Bit-Code der PC als ca. 3.5 mal so schnell wie die SUN erwies. Wenn Routinen untereinander oder unsere Routinen mit denen zu [Steins] verglichen werden, kommt es meist nicht auf die Wahl des Systems an; ansonsten wird explizit darauf hingewiesen.

2.2 Assertions

Als lebensrettend haben sich des öfteren die an fast allen sinnvollen Stellen eingesetzten Assertions erwiesen, also Überprüfungen von Vor- oder Nachbedingungen von Methoden oder anderen Codestücken zur Laufzeit, die für die endgültige Version durch eine Compileroption ausgeblendet werden können.

So werden etwa die BCD-Strings vor und nach jeder Low-Level-Operation mit Assertions auf ihre Konsistenz überprüft. Auf diese Weise konnten beispielsweise einige subtil versteckte Fehler in der relativ komplizierten rekursiv implementierten Multiplikation gefunden werden.

Eine andere Technik, die Assertions verwendet, sei am Beispiel des Konstruktors der Klasse `BFInterval` aus zwei `BigFloat`-Grenzen erläutert. Die normale, nach außen zur Verfügung gestellte Version vertauscht die beiden angegebenen Grenzen im Bedarfsfall stillschweigend. In den internen Arithmetik-Routinen dürfte aber immer gesichert sein, dass die berechnete Untergrenze die Obergrenze nicht überschreitet. Aus Geschwindigkeitsgründen – um den bei langen, fast gleichen Zahlen langsamen Vergleich zu vermeiden – gibt es daher einen internen (privaten) Konstruktor ohne die entsprechende Überprüfung. Dort allerdings sichert *zur Entwicklungszeit* eine Assertion, dass das Programm mit einer passenden Fehlermeldung abgebrochen wird, falls dem Entwickler doch einmal eine Falschangabe unterläuft (wie bei der Intervall-Multiplikation geschehen). Würde immer der sichere Konstruktor mit Vertauschung verwendet, würden solche Unstimmigkeiten gar nicht gefunden (aber die Laufzeit würde unter unnötigen Vertauschungen leiden).

2.3 Exceptions

Alle beteiligten Klassen (zur Grundarithmetik, für die Standard-Funktionen und für die Behandlung der Ausdrücke und deren Wiederberechnung) werfen im Ausnahmefall eine `C++`-Exception. Die Implementation zu [Steins] aus dem Jahr 1995 konnte davon noch nicht Gebrauch machen, da Exceptions zu diesem Zeitpunkt von den wenigsten Compilern vollständig unterstützt wurden. Dort wird im Ernstfall das Programm mit einer Fehlermeldung abgebrochen.

Codiert wird die Information über den Auslöser in einem Objekt der Klasse `XArithException`. Solche Auslöser sind beispielsweise

- arithmetische Ausnahmen (Division durch 0; Definitionsbereich bei Standardfunktionen),
- Overflow (der Exponentenbereich wurde überschritten – im Positiven oder Negativen, d.h. hierunter fällt auch ein Underflow),
- Speichermangel (z.B. nicht genügend Speicher für das exaktes Ergebnis einer Addition: sehr lange Mantisse),
- ungültige Genauigkeitsangabe (dezimale Stellenforderung ≤ 0),
- Index-Fehler (Abfrage einer nicht vorhandenen Dezimalstelle; Zugriff auf eine nicht existente Variable in einem Ausdruck, der aus einem String geparst wurde),
- Fehler in String-Konstanten (Format-Fehler in Integer-, Fließkomma- oder Intervall-Repräsentationen aus Strings oder Dateien; Syntax-Fehler in einem zu parsenden Ausdruck).

Ohne den Exception-Mechanismus müssten fast alle Routinen als zusätzlichen variablen Parameter eine Status-Meldung liefern, der nach jedem Aufruf getestet werden müsste. Das würde den Code sehr unübersichtlich, fehleranfällig (wenn man die Abfrage einmal vergisst) und langsamer machen.

`XArithException` ist von der Standard-Klasse `exception` abgeleitet, es werden momentan von ihr wiederum aber keine Unterklassen gebildet, da einige nicht ganz aktuelle Compiler Probleme bekommen (Objekte der spezialisierten Klassen werden nicht als Objekte der Oberklasse abgefangen). Der Benutzer fängt also ein allgemeines Objekt `e` der Klasse `XArithException` ab und kann über die Methode `e.get_type()` die Art des Fehlers feststellen. Am einfachsten kann er eine Fehlermeldung im Klartext über `e.what()` ausgeben. Der Text enthält den Namen der auslösenden Klasse, eine Kurzbezeichnung (wie „Definitionsbereich überschritten“) und ggf. eine kurze Zusatzinformation (wie „>1“ bei arcsin).

Nicht alle intern erzeugten Exceptions gelangen nach außen zum Benutzer:

- Zum einen fangen die Standard-Funktionen Exceptions der unterliegenden Grundarithmetik ab, um eine genauere Fehlermeldung erzeugen zu können. Beispielsweise fängt `exp` einen Exponentenüberlauf ab, damit der Benutzer eine `exp`-bezogene (und nicht etwa eine auf eine intern verwendete Multiplikation bezogene) Exception erhält.

- Teilweise dienen die Exceptions auch zur Kommunikation zwischen den verschiedenen Ebenen. So fängt die Wiederberechnungs-Routine zum Tangens eine Exception des Intervall-Tangens ab, die erzeugt wird, wenn das Argument-Intervall eine Polstelle enthielt. Im Rahmen der Wiederberechnung kann dann (wenn dies eingestellt wurde) versucht werden, zunächst das Argument zu verfeinern. Diese Routine höheren Levels braucht aber auf diese Weise nicht vorher selbst den (relativ komplizierten) Polstellen-Test durchzuführen – was sonst ja insgesamt zweimal geschehen würde.

Einige Exception-Typen besitzen Untertypen. So signalisiert der „`domain_error`“ (Überschreitung des Definitionsbereichs einer Standardfunktion) bei intervallmäßigen Auswertungen, ob *das gesamte Argument-Intervall* außerhalb des Definitionsbereichs liegt oder nur Teile. Das wird beispielsweise beim Funktionsplotter `xariplot` ausgenutzt, um zunächst Definitionslücken auszuschließen. Teilintervalle mit Domain-Errors werden feiner unterteilt, um die Problemstellen enger einzuschließen (was nicht sinnvoll ist, wenn die Funktion auf dem gesamten Intervall nicht definiert ist).

Das Erzeugen einer Exception unseres Typs geschieht nicht direkt mit `throw`, sondern mit einer Hilfsroutine `xarith_error`, die das eigentliche Exception-Objekt zusammenbaut und wirft. Auf diese Weise kann die Klassenbibliothek auch einfacher auf Systeme ohne Exceptions portiert werden. Die Stellen, an denen intern Exceptions *abgefangen* werden, müssten aber dennoch einzeln per Hand geändert werden.

2.4 `BigInt`

Die Klasse `BigInt` stellt eine Arithmetik für (fast) beliebig große Integerzahlen zur Verfügung. Die Größe ist nur durch den Typ beschränkt, der die Länge (in internen Elementen gemessen) speichert, und für ihn wird im Moment der `C++`-Typ `long` verwendet. Auf den meisten Systemen liegt die damit größte darstellbare Länge bei etwa 2 Milliarden, wodurch wiederum die eigentliche Beschränkung nur im auf dem Rechner verfügbaren Speicher zu suchen ist.

`BigInt` ist nicht nur zur Verwendung als eigentliche Integerzahlen gedacht. Die Klasse stellt auch die Mantissen der später zu besprechenden Fließkommazahlen. Aus diesem Grund ist auch sie bereits zur Basis 10 implementiert, muss aber auf fast jedem System die vom Prozessor zur Verfügung gestellten binären Formate benutzen, was deutliche Geschwindigkeitseinbußen bedeuten muss (Behandlung von Überträgen, etc.). Wer also eine reine Ganzzahl-Arithmetik benötigt, ist vermutlich besser mit einer Implementation zur Basis 2 bedient (etwa der aus der GNU MP-Library).

2.4.1 BCD-Format

Grundlage für die Design-Entscheidungen für das zu verwendende interne Zahlenformat ist, dass die Arithmetik rein auf Hochsprache-Ebene implementiert sein sollte und daher nur Zugriff über die in `C++` enthaltenen Ganzzahlformate und -operationen hat. Beispielsweise stellen viele Prozessoren ein eigenes BCD-Ganzzahlformat zur Verfügung, das aber von `C++` nicht ansprechbar ist und das deshalb für uns nicht in Frage kommt. Für den Fall, dass

eine mögliche spätere Erweiterung das (mit systemabhängigen Assembler-Teilen) ändern wollte, sind jedenfalls alle betroffenen Stellen innerhalb der Klassen `BCDstring` und `BigInt` abgekapselt.

- Natürlich sollen auf der Maschine in Hardware vorhandene Operationen so weit wie möglich ausgenutzt werden. Daher ist es nicht sinnvoll, als kleinstes Arithmetik-Element, also als BCD-Ziffer, eine Ziffer zur Basis 10, d.h. ein Element aus $\{0, 1, \ldots, 9\}$, zu wählen. Es wird also vielmehr als eigentliche Basis 10^n verwendet, wobei n zumindest so gewählt werden muss, dass $\{0, \ldots, 10^n - 1\}$ noch in einem von `C++` verwendbaren Format Platz findet. Wenn 16-Bit-Binärzahlen (meist `short int`) verwendet werden sollen, ist die Basis der Wahl daher 10000, bei 32-Bit (meist `long int`) 100 000 000.

- Außerdem muss aber berücksichtigt werden, dass *Zwischenergebnisse* der Grundoperationen von `C++` aus ebenfalls noch zugänglich sein müssen. Bei Additionen mit Überträgen können maximal zwei BCD-Ziffern und ein kleiner Übertrag anfallen, so dass das Ergebnis bei obigen Wahlen noch immer in den gewählten Binärtypen Platz hat.

- Problematisch werden allerdings Multiplikation und Division. Auf 32-Bit-Maschinen gibt es zwar typischerweise Operationen, die zwei 32-Bit-Binärzahlen zu einer 64-Bit-Zahl multiplizieren; das Ergebnis wird z.B. auf zwei Prozessorregister verteilt. Von `C++` aus sind aber nur Operationen *mit gleicher Länge* für Operanden und Ergebnis zugänglich. 32-Bit-Zahlen sind zwar multiplizierbar, das Ergebnis ist aber wiederum 32 Bit groß. Ein Überlauf wird *nicht signalisiert*, wobei der berechnete Wert dann z.B. das korrekte Resultat modulo 2^{32} ist.

  Wenn man also nicht obige Multiplikationen auf `C++`-Ebene kompliziert simulieren oder doch auf Assembler zurückgreifen will, ist man für die BCD-Ziffern bei 32-Bit-Maschinen auf 16 Bit beschränkt, bei 64-Bit-Maschinen auf 32 Bit. Analoges gilt für Divisionen.

- Zusätzlich ist zu bedenken, dass es bei Verwendung von 16-Bit-Zahlen als BCD-Ziffer für eine Multiplikation $16 \times 16 \to 32$ in `C++` notwendig ist, beide Operanden zuvor explizit von 16 nach 32 Bit umzuwandeln (was auch echten Code erzeugt, beispielsweise den Intel-Prozessorbefehl `ext`). Man kann im Hinblick auf Performance daher am besten damit bedient sein, jede 16-Bit-BCD-Ziffer intern verschwenderisch im 32-Bit-Format zu speichern. Dadurch wird natürlich der doppelte Speicherplatz benötigt, die Arithmetik insgesamt aber beschleunigt.

Die BCD-Zahl 1234567890 mit $\underline{3}$ BCD-Ziffern zur Basis 10000 bzw. $\underline{10}$ Dezimalziffern sieht beispielsweise wie folgt aus:

0	0	1	2	3	4	5	6	7	8	9	0

Als Beispiel für die etwas gewöhnungsbedürftige Handhabung dieser Zahlen bedenke man, dass es schon relativ aufwändig ist, ihre Länge in Dezimalstellen zu bestimmen. Bei den implementierten Routinen wird die Länge der führenden Ziffer nicht über eine Schleife mit Division, sondern entweder möglichst schnell über eine geschachtelte binäre Fallunterscheidung (< 100, dann < 10, etc.) oder (nur bei der kleineren Basis 10000) über eine vorberechnete Tabelle realisiert.

Im Hinblick auf obige Überlegungen wurde entschieden, die Arithmetik (zur Compilationszeit) frei, aber dann fest konfigurierbar zu machen. Alle betroffenen Low-Level-Routinen enthalten entsprechende Parameter, die in der Header-Datei `XArithConfig.h` festgelegt werden. Zwei Typen können den Gegebenheiten auf dem jeweiligen Rechner angepasst werden:

- `BCDelem` ist der Typ für eine BCD-Ziffer,
- `BCDelem2` ist der Typ für das Ergebnis einer Multiplikation BCD×BCD.

Vor allen Multiplikationen im Code werden die `BCDelem`-Operanden durch einen Typecast nach `BCDelem2` umgewandelt. Wenn allerdings die beiden Typen (wie oben beschrieben) zusammenfallen sollten, wird bei diesem Cast *kein Code* erzeugt.

Eine mögliche und typische Wahl ist also `BCDelem=short` und `BCDelem2`=long, oder – wie momentan voreingestellt – `BCDelem=BCDelem2=long`. Auf 64-Bit-Maschinen macht die Kombination `BCDelem=long`, `BCDelem2=long long` (64 Bit) Sinn.

Ein weiterer in `XArithConfig.h` definierter Typ ist `BCD_st` („size type“), der für die Speicherung der Länge der BCD-Zahlen (in BCD-Ziffern gemessen) verwendet wird. In Anlehnung an die Steins'sche Implementation trägt er im obersten Bit auch das Vorzeichen der Zahl. Er ist als `long` vordefiniert, d.h. die Maximallänge auf einer typischen Maschine ist $2^{31}-1$, womit sich bei der BCD-Basis 10000 als größte darstellbare Zahl $10000^{2^{31}-1}-1$ ergibt.

2.4.2 `BCDstring`

Diese Klasse ist eine Low-Level-Klasse, die nicht nach außen zur Verfügung gestellt wird, sondern allein als Hilfsklasse von `BigInt` benutzt wird. Diese Aufteilung orientiert sich auch an der in der Implementation zu [Steins], die Trennung wurde nur etwas deutlicher festgeschrieben.

Es sind die Objekte der Klasse `BCDstring`, die die eigentlichen BCD-Ziffern der Mantisse speichern. Auf dieser Ebene sind aber auch die grundlegenden arithmetischen Operationen definiert, auf die High-Level-Operationen aus `BigInt` zurückgeführt werden (etwa Addition bzw. Subtraktion *positiver* Strings, Vorzeichenunterscheidungen finden auf höherer Ebene statt). `BigInt`-Objekte besitzen einen *Pointer* auf ein `BCDstring`-Objekt, und mehrere `BigInt`-Objekte können sich auf diese Weise ein `BCDstring`-Objekt teilen.

Bei einer Zuweisung in der Klasse `BigInt` werden nicht die Ziffern kopiert, lediglich der Pointer. Innerhalb des `BCDstring`-Objekts wird ein Zähler mitgeführt, der Buch darüber führt, wie viele `BigInt`-Objekte momentan noch Zugriff haben. Erst, wenn der `BCDstring` von keinem `BigInt` mehr benutzt wird, wird er gelöscht.

Die Aufteilung hat im Wesentlichen zwei Gründe. Erstens ist es allgemein durchaus wünschenswert und eine kluge Technik („lazy evaluation“), möglicherweise aufwändige Operationen hinauszuschieben, bis sie tatsächlich unumgänglich werden. Zweitens ist es beim Design

der Sprache C++ und bei der Arbeitsweise der meisten Compiler praktisch nicht zu vermeiden, dass viele temporäre Objekte entstehen, je nach Güte des Compilers mehr oder weniger überflüssig viele – etwa bei der Rückgabe von neu erzeugten Objekten, die auf Objekte der Aufrufebene kopiert werden. Wenn jedes Objekt der Klasse BigInt seine eigene Mantisse bei sich führen würde, würden, beispielsweise bei der Übergabe von Argumenten und der Rückgabe von Ergebnissen, sehr viele unnötige Kopiervorgänge stattfinden.

Zur Illustration folgt ein Ausschnitt der Definition der Klasse, in dem die Datenmember und Konstruktoren sichtbar sind.

```
class BCDstring
 {
   private:
      static long num_objects;
      ...
      BCD_st space;           // number of allocated BCD elements
      BCD_st size;            // actual number of used elements (+sign)
      BCDelem *digits;        // pointer to BCD elements, warning: see below!

      long refCount;     // number of referring (BigInt) objects -1 !!!
                         // *only handled in BigInt* using routines of BCDstring
   public:
      BCDstring();                     // value = 0
      BCDstring(const BCDstring &);    // copy constructor (copies the digits!)
      BCDstring(long l);               // init from a long integer
      BCDstring(const char *);         // from string representation of an integer
      ~BCDstring();                    // destructor
      ...
 };
```

Die High-Level-Klasse BigInt hat nur einen einzigen (nicht-statischen) Daten-Member, nämlich den Pointer auf das zu verwendende BCDstring-Objekt. Die Methoden zur Verwaltung der Mehrfachverwendung (und der Zähler) liegen in BCDstring, müssen aber von BigInt aus aufgerufen werden. Der Hauptgrund dafür ist, dass sich dynamisch angelegte BCDstring-Objekte nicht selbst automatisch löschen können, wenn sie den Zähler dekrementieren und feststellen, dass sie nicht mehr gebraucht werden.

BigInt stellt die typischerweise gewünschten Konstruktoren, Operatoren für Zuweisungen, Vergleiche und arithmetische Operationen zur Verfügung, Wandlungen in andere Typen, Ein- und Ausgabe-Operatoren, etc. Der Funktionsumfang ist zu groß, als dass er hier aufgelistet werden sollte.

Als ein Design-Fehler hat es sich erwiesen, das Vorzeichen der Zahl im BCDstring-Objekt zu kodieren, was ganz zu Beginn hauptsächlich aus Kompatibilitätsgründen zu [Steins] geschah. Auf diese Weise muss bei einem Vorzeichenwechsel leider die komplette Ziffernfolge kopiert werden. Gerade bei dem im Aufbau letzten Teil der Implementation, dem Algorithmus zur Auswertung arithmetischer Ausdrücke, werden sehr viele Absolutbeträge benötigt, die nun ggf. ein solches Kopieren erzwingen.

Eine Auslagerung des Vorzeichens von BCDstring nach BigInt stellt einen relativ schweren Eingriff in diese Klassen dar und konnte aus Zeitgründen nicht mehr vorgenommen werden. Sie sollte bei einem Ausbau der Arithmetik aber auf jeden Fall berücksichtigt werden.

2.4.3 Die Addition und Subtraktion

Die Addition und Subtraktion zweier `BCDstring`s sind in kanonischer Weise stellenweise von rechts nach links implementiert.

Es sei noch einmal darauf hingewiesen, dass die Summe zweier BCD-Ziffern und der mögliche Übertrag von 1 immer noch im verwendeten Binärtyp Platz findet. Nach jeder Stellenaddition wird also überprüft, ob das Ergebnis `d` größer oder gleich der verwendeten Basis `BCDbase` (z.B. = 10000) ist. Falls ja, ist die tatsächliche neue Ziffer `d-BCDbase`, und man hat einen Übertrag von 1. Analog wird bei der Subtraktion getestet, ob das Ergebnis kleiner als 0 wurde und dann ggf. `BCDbase` addiert, um die eigentliche Ziffer zu erhalten.

Aus Geschwindigkeitsgründen wurden viele verschiedene Versionen für unterschiedliche Zwecke programmiert und optimiert: Beispielsweise gibt es solche, die für das Ergebnis immer einen neuen Speicherbereich verwenden, und solche, die den linken Operanden verändern und erst dann Speicher umkopieren, wenn es durch einen längeren rechten Operanden oder einen Überlauf notwendig wird. Es gibt Versionen, die nur eine BCD-Ziffer addieren (sinnvoll für kleine ganze Zahlen, insbesondere für die Operatoren `++` und `--`), oder nur eine `long`-Zahl und dabei die speziellen Gegebenheiten auf dem Rechner ausnutzen. Wenn das Ziel ein neuer Speicherbereich ist, der kleinere der beiden Operanden abgearbeitet ist und kein Übertrag mehr auftritt, wird der Rest des größeren Operanden mit `memcpy` und nicht stellenweise auf `C++`-Ebene kopiert.

Durch obige Vorgehensweisen und weitere Optimierungen auf niedriger Ebene (günstige Schleifensteuerung, Aufrollen bestimmter Programmteile, andere Konstantendefinition) wird gegenüber der Version zu [Steins] eine Geschwindigkeitssteigerung um ca. den Faktor 2 erreicht.

2.4.4 Die Multiplikation

Im Hinblick auf größere Genauigkeitsforderungen wurde die Multiplikation in einer rekursiven Form realisiert. Für das untere Ende der Rekursion, und da sich die Rekursion wegen des zusätzlich anfallenden Verwaltungsaufwands erst ab einer gewissen Operandenlänge lohnt, wurde natürlich auch eine „klassische“ (stellenweise) Multiplikation implementiert.

2.4.4.1 Multiplikation klassisch

Zunächst betrachten wir die Multiplikation einer Langzahl mit einer BCD-Ziffer. Diese ist auch so als eigenständige Routine implementiert und wird automatisch von den High-Level-Operatoren aufgerufen, wenn einer der Operanden einer Multiplikation nur eine Ziffer lang ist oder wenn mit einer normalen Integerzahl multipliziert werden soll, die klein genug ist.

Um die Darstellung zu vereinfachen, setzen wir hier Standard-Übersetzungsparameter voraus, d.h. es gibt eine 32-Bit-Multiplikation im System, und eine BCD-Ziffer liegt in $[0, 9999]$. Das Produkt zweier solcher Ziffern liegt in $[0, 99980001]$, ein erster möglicher Übertrag also in $[0, 9998]$. Damit ist ab der zweiten Stelle ein Wert 99989999 möglich. Auf jeden Fall passt das Ergebnis noch immer ohne Überlauf in einen 32-Bit-Wert.

Anders als bei der Addition kann der Überlauf hier nicht mehr durch einen einfachen Vergleich und die eigentliche Ziffer durch Subtraktion ermittelt werden. Wir benötigen hier

tatsächlich eine Division durch die Basis `BCDbase` (z.B. 10000). Da wir Quotient (den Übertrag) und Rest (die eigentliche Ziffer) benötigen, liegt es nahe, die `C`-Bibliotheksfunktion `div` zu benutzen, die beides gleichzeitig liefert und intern nur einen einzigen Prozessorbefehl braucht (so wird auch bei [Steins] verfahren).

Es stellt sich aber heraus, dass `div` wesentlich *langsamer* ist als eine aufeinander folgende Division und Modulo-Operation auf `C`-Ebene, *falls* der Divisor eine echte Konstante zur Compilationszeit ist. Dann nämlich werden vom Compiler spezielle Sequenzen von Additionen, Subtraktionen und Schiebebefehlen erzeugt, die insgesamt mehr als 10% schneller als der allgemeine Prozessorbefehl zur Division sind (das wurde sowohl auf Intel-L*i*nux-Rechnern wie den SUNs festgestellt). Es reicht *nicht*, die Basis als `const` zu deklarieren, da dann dennoch ihr Wert nicht vom Compiler eingesetzt wird (der verwendete GNU-Compiler verfährt jedenfalls so, und der Wert könnte mit `const_cast` und/oder Pointern tatsächlich geändert werden). Sie wurde daher als `enum`-Konstante definiert (momentan global, da einige Compiler mit Namespaces Schwierigkeiten hatten).

Die mehrstellige klassische Multiplikation ($x, y > 0$) benutzt wie üblich

$$x \cdot y = x \cdot \sum_{k=0}^{m-1} y_k \cdot b^k = \sum_{k=0}^{m-1} (y_k \cdot x) \cdot b^k,$$

multipliziert also den linken Operanden nacheinander mit jeweils einer Ziffer des rechten und addiert die Teilergebnisse stellenverschoben aufeinander. In der Implementation wird die bei der Teilmultiplikation entstandene aktuelle Ziffer direkt auf den Ergebnisbereich addiert. Man hat nun also (bei Standard-Parametern) maximal einen aufaddierten Wert von 99999998, was immer noch mit der internen 32-Bit-Arithmetik erledigt werden kann.

Der Verwaltungsaufwand wird geringer, wenn man für x den längeren Operanden wählt. Wenn eine Ziffer 0 ist, wird die entsprechende Teilmultiplikation ausgelassen. Es lohnt aber nicht, die Ziffer auch mit 1 zu vergleichen, da bei der Basis 10000 die Trefferquote zu gering ist.

Die Routine wurde gegenüber der zu [Steins] folgendermaßen beschleunigt (Geschwindigkeitsfaktor bei normalen Längen der Operanden ca. 2.5 bis 4):

- Wir beginnen mit der niederwertigsten Stelle von y, wodurch der Aufwand durch Überträge, die sich nach weit links durchziehen, verkleinert wird.
- Der Ergebnisbereich wird nicht vollständig vorab mit Nullen initialisiert. Das erste Ergebnis wird in den unteren Teil hinein berechnet, nur die obere „Hälfte“ wird danach mit einer Systemroutine gelöscht.

Die Low-Level Routine heißt

```
multiply_digits_norecurse(const BCDelem *D1,  BCD_st len1,
   const BCDelem *D2,  BCD_st len2,  BCDelem *DT,  BCD_st lenT);
```

und geht davon aus, dass `len1`$\geq$`len2`, was vorher durch die aufrufenden High-Level-Operatoren `*`, `*=` etc. aus `BigInt` sichergestellt wird. Es ist sinnvoll, das nicht innerhalb der Routine zu überprüfen und ggf. die Operanden zu vertauschen, da die Routine auch von der rekursiven Routine aus aufgerufen wird, und da in der entsprechenden Situation immer klar ist, welcher Operand der längere ist (was aber natürlich durch Assertions abgesichert ist).

2.4.4.2 Multiplikation rekursiv

Die Implementation der rekursiven Multiplikation lehnt sich an die Darstellung in [Knuth] (Seite 278ff) an. Die Operanden werden dabei jeder in zwei Teile zerlegt. Die Multiplikation wird zurückgeführt auf 3 Multiplikationen von Zahlen etwa halber Operandenlänge und einige Additionen und Verschiebungen, wie weiter unten erläutert werden wird. Dabei reduziert sich der Aufwand von n^2 (wobei n=Anzahl Ziffern zur Basis $b \geq 2$ bei gleich langen Operanden) auf etwa $n^{1.585}$. Gegenüber der klassischen Version sind also beliebig große Geschwindigkeitszuwächse zu erzielen – wenn die Längen nur groß genug sind.

Es ist zu beachten, dass der Verwaltungsaufwand dieser Zerlegung relativ groß ist, so dass sich der Einsatz erst ab einer bestimmten Operandenlänge lohnt. Diese hängt von der genauen Implementation von Addition und Verschiebungen, sowie von der Systemarchitektur (Rekursion, temporärer Speicher) ab und muss im Einzelfall durch Zeitmessungen bestimmt werden. Auf den verwendeten SUN-Rechnern war das beispielsweise ab etwa $k_{min} = 24$ Dezimalziffern, auf den L*i*nux-Rechnern erst ab etwa $k_{min} = 48$ Dezimalziffern der Fall. Als Geschwindigkeitsgewinn erhält man dort bei etwa 100 Stellen einen Faktor 1.2, bei 1000 Stellen etwa 13. Die Beispiele für den Wiederberechnungsalgorithmus, z.B. in Abschnitt 5.8.6, zeigen aber, dass man auch leicht bei 16-stelliger Endforderung zwischenzeitlich 50 oder 100 Stellen benötigen kann!

Die Zerlegung orientiert sich am Operanden größerer Länge. Es ist dabei essentiell, dass die niederwertigen Operandenteile gleiche Länge haben.

Die Operanden mögen x, y heißen mit $n = \text{len}(x)$, $m = \text{len}(y)$ und $n \geq m$. Sei $\ell := \lfloor \frac{n+1}{2} \rfloor$ die Länge des jeweils niederwertigen Teils der Zerlegung.

a) Wenn einer der Operanden kürzer ist als k_{min}, wird die Rekursion nicht fortgeführt und die klassische Multiplikation verwendet.

b) Wenn $m \leq \ell$, kann y nicht passend zerlegt werden. Selbst wenn $\ell < m \leq \ell + k_{min}$, lohnt sich die eigentliche Zerlegung nicht, da der entsprechend kurze Teil von y eine Stufe tiefer eine weitere Rekursion verhindert.

   In diesem Fall wird x in zwei Hälften zerlegt und die eigentliche Zerlegung eine Stufe nach unten verschoben, in der sicheren Erwartung, dass nach ggf. mehreren solche Schritten die Bestandteile von x kurz genug werden. Es ist $x = x_1 \cdot b^\ell + x_2$, also

$$x \cdot y = (x_1 \cdot b^\ell + x_2) \cdot y = (x_1 \cdot y) \cdot b^\ell + x_2 \cdot y$$

   mit $\text{len}(x_1),\ \text{len}(x_2) \leq \ell$. Es wird zweimal rekursiv die Multiplikation von etwa der halben Länge von x aufgerufen, die Ergebnisse werden stellenverschoben aufaddiert.

c) Wenn dagegen $m > \ell + k_{\min}$, können beide Operanden sinnvoll zerlegt werden:

$$\begin{aligned} x &= x_1 \cdot b^\ell + x_2, \quad \text{len}(x_2) \leq \ell,\ \text{len}(x_1) = \ell \text{ oder } = \ell - 1, \\ y &= y_1 \cdot b^\ell + y_2, \quad \text{len}(y_2) \leq \ell,\ \text{len}(y_1) \leq \text{len}(x_1) \leq \ell. \end{aligned}$$

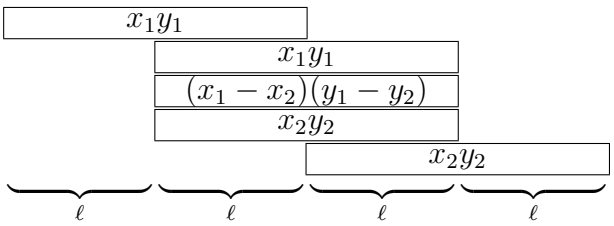
   Es gilt nicht notwendigerweise $\text{len}(x_2) = \text{len}(y_2) = \ell$, da sich führende Nullen ergeben haben könnten. Es resultiert folgende „klassische“ Zerlegung

$$x \cdot y = x_1 \cdot y_1 \cdot b^{2\ell} + x_1 \cdot y_2 \cdot b^\ell + x_2 \cdot y_1 \cdot b^\ell + x_2 \cdot y_2 \cdot b^0,$$

die vier Multiplikationen der Länge maximal ℓ enthält und noch keinen Vorteil bietet. Folgende weitere Umformung reduziert die Anzahl halblanger Multiplikationen aber auf drei:

$$\begin{aligned} x \cdot y &= (b^{2\ell} + b^\ell) \cdot x_1 \cdot y_1 \; - \; b^\ell \cdot x_1 \cdot y_1 \\ &\quad + b^\ell \cdot (x_1 \cdot y_2 + x_2 \cdot y_1) \\ &\quad + (b^\ell + 1) \cdot x_2 \cdot y_2 \; - \; b^\ell \cdot x_2 \cdot y_2 \\ &= (b^{2\ell} + b^\ell) \cdot x_1\, y_1 + b^\ell \cdot (x_1\, y_2 - x_1\, y_1 + x_2\, y_1 - x_2\, y_2) + (b^\ell + 1) \cdot x_2\, y_2 \\ &= (b^{2\ell} + b^\ell) \cdot x_1\, y_1 \; + \; b^\ell \cdot (x_1 - x_2)\,(y_2 - y_1) \; + \; (b^\ell + 1) \cdot x_2\, y_2. \end{aligned}$$

Letztlich benötigt diese Version drei halblange Multiplikationen, zwei Subtraktionen und zehn halblange Additionen (wenn man die seltene mögliche Fortpflanzung eines Übertrags nach ganz links als volle Addition berechnet). Die Situation ist im Bild rechts dargestellt.



Um eine möglichst hohe Geschwindigkeit zu erreichen, ist die genaue Durchführung dieser rekursiven Version recht aufwändig geworden und soll hier nur in Kurzform beschrieben werden. Bei der Implementation wird ein Zwischenpuffer für ein Ergebnis der Länge 4ℓ benötigt.

- Es wird zunächst (durch rekursiven Aufruf) x_1y_1 direkt in den Ergebnisbereich hinein und x_2y_2 (rekursiv) in den Puffer hinein berechnet (es wird dafür nur die untere Hälfte des Puffers benötigt).

- Dann wird die untere Hälfte von x_2y_2 in den Ergebnisbereich kopiert (mit einer schnellen Kopierroutine).

- Als Nächstes werden die untere Hälfte von x_2y_2, die obere Hälfte von x_2y_2 (beide aus dem Puffer) und die untere Hälfte von x_1y_1 (in einem Schritt) addiert und in das zweite Viertel des Ergebnisses geschrieben. Dabei stammt x_1y_1 auch aus dem Ergebnisbereich, liegt aber ℓ Stellen weiter links, so dass es keine Probleme mit etwaigen Überschreibungen gäbe. Am linken Ende dieses Viertels wird gestoppt und der Übertrag aufbewahrt.

  Damit ist der Ergebnisbereich gefüllt worden, ohne dass vorher eine Initialisierung mit Nullen notwendig gewesen wäre.

- Nun wird auf das dritte Viertel die obere Hälfte von x_1y_1 und die obere Hälfte von x_2y_2 addiert, wobei mit dem Übertrag aus der letzten Addition begonnen wird. Es muss natürlich beachtet werden, dass x_2y_2 (z.B. durch führende Nullen) kürzer sein kann als der im Bild dafür dargestellte Bereich. In diesem Fall brauchen ab einer gewissen Stelle nur noch Ziffern aus x_1y_1 verwendet zu werden.

  Ein Übertrag muss hier möglicherweise bis ganz nach links durchgezogen werden, es kann aber keinen Gesamtüberlauf geben, so dass dabei das Erreichen des linken Rands gar nicht überprüft werden muss.

Nun ist x_2y_2 vollständig berücksichtigt worden, und der Pufferbereich ist wieder frei verwendbar.

- Die Terme $x_1 - x_2$ und $y_2 - y_1$ werden immer *betragsmäßig* berechnet, wofür schnelle Vorab-Vergleiche notwendig sind. Beide haben die Maximallänge ℓ und finden in den oberen beiden Vierteln des Puffers Platz. Anschließend werden sie (rekursiv) in die untere Hälfte des Puffers hinein multipliziert. Das Ergebnis wird schließlich (vorzeichenabhängig) auf die mittleren Viertel des Ergebnisbereichs addiert oder von ihnen subtrahiert. Ein Übertrag kann wieder bis ganz nach links durchlaufen.

Die rekursive Low-Level Routine heißt

```
multiply_digits_recurse(const BCDelem *D1,  BCD_st len1,
   const BCDelem *D2,   BCD_st len2,   BCDelem *DT,   BCD_st lenT);
```

und ist diejenige, die von den High-Level-Operatoren immer direkt aufgerufen wird. Sie stellt jeweils zu Beginn fest, welcher der drei Fälle bezüglich der Operandenlänge vorliegt, und ruft ggf. die nicht-rekursive Routine auf.

Die rekursive Version wurde dadurch noch leicht beschleunigt, dass der benötigte Speicher für Temporärergebnisse nicht in jeder Rekursionsstufe angelegt und am Ende wieder freigegeben wird. Vielmehr wird vorab ein Maximalbedarf geschätzt und dann mit einem statischen Puffer und einem Zeiger ein entsprechender Stackauf- und -abbau selbst durchgeführt. Der Puffer bleibt auch über Multiplikationsaufrufe hinweg erhalten, braucht also ggf. nur einmal während des Programmlaufs angelegt zu werden.

Eine Multiplikationsstufe mit halber Operandenlänge ℓ benötigt $4\lceil \ell/2 \rceil$ BCD-Ziffern Zwischenspeicher. Für die Abschätzung nach oben wird angenommen, dass beide Operanden gleich lang sind, diese Länge wird außerdem auf die nächstgrößere Zweierpotenz aufgerundet, und wir gehen von einem Abstieg hinunter bis zu einer Ziffer aus. Dann ist der Gesamtspeicherbedarf durch 4ℓ beschränkt.

Wir wollen noch den (Zeit-)Aufwand dieser geänderten Fassung nach oben abschätzen, wobei wir uns auf die Multiplikation zweier gleich langer Zahlen beschränken (nach der Skizze in [Knuth]). Der Einfachheit halber geht die Abschätzung außerdem davon aus, dass die Rekursion hinab bis zur Operandenlänge 1 getrieben wird.

Es sei $t(n)$ die Zeit, die für die Multiplikation zweier Zahlen der Länge n benötigt wird. Da die Aufteilung stets symmetrisch ist, liegt immer Fall c) vor.

Wir betrachten zunächst den Fall gerader Länge. Offenbar gilt

$$t(2\ell) \leq 3t(\ell) + c_1 \cdot \ell, \qquad t(1) \leq c_2.$$

mit Konstanten c_1 und c_2, wobei der zweite Teil von den beteiligten Additionen und Shifts eingebracht wird, deren Aufwand maximal linear in der Länge ist. Für $\ell = 1$ erhalten wir

$$t(2) \leq 3c_2 + c_1 =: c.$$

Dann gilt per Induktion

$$t(2^k) \leq c(3^k - 2^k) \text{ für alle } k \geq 1.$$

Der Anfang $k = 1$ definierte oben c. Ansonsten haben wir

$$\begin{aligned} t(2^k) &= t(2 \cdot 2^{k-1}) \;\le\; 3t(2^{k-1}) + c_1 \cdot 2^{k-1} \overset{\text{Ind.}}{\le} 3c \cdot (3^{k-1} - 2^{k-1}) + c_1 \cdot 2^{k-1} \\ &= c \cdot 3^k - 2^{k-1}(3c - c_1) \;=\; c \cdot 3^k - 2^k(\tfrac{9}{2}c_2 + c_1) \;\le\; c(3^k - 2^k). \end{aligned}$$

Für eine allgemeine Länge n gilt nun

$$t(n) \le t(2^{\lceil \log_2 n \rceil}) \le c \cdot (3^{\lceil \log_2 n \rceil} - 2^{\lceil \log_2 n \rceil}) \le c \cdot 3^{1+\log_2 n} = 3c \cdot 3^{\log_2 n} = 3c \cdot n^{\log_2 3} \approx 3c \cdot n^{1.585}.$$

Die Ordnung kann von 1.585 noch weiter reduziert werden (vergleiche etwa [Knuth], S. 283, Algorithm C). Man benötigt dann aber extrem aufwändige Datenstrukturen, und dieser Aufwand lohnt sich vermutlich erst weit jenseits des von uns angepeilten Stellenbereichs.

2.4.4.3 Quadrieren

Das Quadrieren kommt gegenüber der klassischen Multiplikation mit etwa der Hälfte der Operationen aus. Es wird verwendet

$$\Big(\sum_{k=0}^{n-1} x_k b^k\Big)^2 = \sum_{k=0}^{n-1} x_k^2 b^{2k} + \sum_{k=0}^{n-2}\Big((2x_k)b^k \sum_{j=0}^{k-1} x^j b^j\Big).$$

Die Version zu [Steins] ermittelt zunächst die zweite Summe ohne den Faktor 2, multipliziert das Ergebnis mit 2 und addiert anschließend die quadratischen Terme. Die vorliegende Implementation arbeitet dagegen schneller wie folgt:

- Der Ergebnisbereich wird nicht vorab mit Nullen belegt, sondern im ersten Schritt mit den Quadraten der Ziffern gefüllt. Jedes belegt gerade zwei Ziffern des Ergebnisses ohne Überschneidungen.
- Darauf werden nacheinander die gemischten Terme direkt aufaddiert, wobei in einem k-Schritt $f := 2x_k$ fest ist und vor der Berechnung der inneren Summe ermittelt wird.

Zur Illustration ist im Diagramm rechts das Quadrieren von 5981 (zur Basis 10) dargestellt.

Diese Version benötigt statt n^2 nur $\frac{n(n+3)}{2} - 1$ einzelne Multiplikationen; dennoch ist die Ordnung natürlich immer noch $\mathcal{O}(n^2)$.

								$\leftarrow 5^2 \vert 9^2 \vert 8^2 \vert 1^2$
			9	8	1	0		$\leftarrow (2 \cdot 5) \cdot 981$
				1	4	5	8	$\leftarrow (2 \cdot 9) \cdot 81$
						1	6	$\leftarrow (2 \cdot 8) \cdot 1$
3	5	7	7	2	3	6	1	

Die rekursive Version der Multiplikation würde erst bei extrem hohen Stellenzahlen schneller als diese hier, weswegen in der momentanen Implementation keine entsprechende Überprüfung durchgeführt wird.

2.4.5 Potenzieren

Das Potenzieren $z := x^y$, $y \in \mathbb{N}$, wird natürlich nicht durch einfaches Aufmultiplizieren, sondern zumindest über die Binärdarstellung des Exponenten durchgeführt, in `C++`-ähnlichem Pseudocode (Voraussetzung $y \neq 0$):

```
BigInt f(x);
while ( (y&1)==0 )
 {
   y>>=1;
   f*=f;
 }
BigInt z(f);
while ( (y>>=1) !=0 )
 {
   f*=f;
   if ( (y&1)!=0 ) z*=f;
 }
```

Die erste Schleife dient zur Vermeidung einer Initialisierung mit 1 und anschließender Multiplikation mit x. Dieser Algorithmus benötigt allerdings nicht immer die minimal mögliche Anzahl von Multiplikationen (für eine ausführliche Behandlung dieses Problems siehe z.B. [Knuth], S. 441-466).

2.4.6 Die Division

Die Division von langen Integerzahlen (mit Rest) wurde in Anlehnung an [Knuth] (S. 257, Algorithm D) implementiert. Es handelt sich im Wesentlichen um das Verfahren, das als schriftliche Division in der Grundschule gelehrt wird.

In jedem Schritt muss eine Teildivision $q := \lfloor x'/y \rfloor$ durchgeführt werden, wobei x' von gleicher Länge wie y oder eine Stelle länger ist. Diese nächste Stelle q des Ergebnisses muss gleichsam „geraten“ werden, üblicherweise auf den ersten wenigen Stellen von x' und y basierend. Anschließend wird das geratene $\hat{q}$ mit y multipliziert und von x' subtrahiert, wobei eventuell festgestellt wird, dass das Produkt größer als x' oder kleiner gleich $x' - y$ geworden ist. Dann müssen $\hat{q}$ und x' entsprechend korrigiert werden.

Die bei uns (und [Knuth]) geratene Stelle $\hat{q}$ basiert auf den führenden beiden Ziffern von x' und der führenden Ziffer von y:

$$\hat{q} := \min\left\{ \left\lfloor \frac{x'_0 b + x'_1}{y_1} \right\rfloor, b - 1 \right\}.$$

In unserer Standard-Implementation zur Basis $b = 10000$ liegt $(x'_0 x'_1)_b$ in $[0, 99999999]$, d.h. wir können für diesen Schritt die im System vorausgesetzte 32-Bit-Division benutzen.

Nun ist dieses $\hat{q}$ niemals zu klein (siehe [Knuth], 4.3.1, Theorem A), und unter der Voraussetzung $y_1 \geq \lfloor b/2 \rfloor$ maximal um 2 zu groß (Theorem B). Um die Voraussetzung zu erreichen, muss vor Beginn der Division der Bruch passend erweitert werden. Der Faktor $d := \lfloor b/(y_1 + 1) \rfloor$ garantiert dies, ohne den Divisor zu verlängern, und die Multiplikationen $d \cdot x$, $d \cdot y$ sind nur solche mit einer Ziffer. Der sich ergebende Quotient bleibt natürlich unverändert; allerdings muss der Rest wieder durch die Ziffer d dividiert werden. Entsprechend hat die Low-Level-Routine einen Parameter, der bestimmt, ob der Rest überhaupt gefragt ist; ansonsten kann diese Korrektur entfallen.

Zusätzlich führen wir den in [Knuth] (4.3.1, Algorithm D, Teil D3) vorgeschlagenen Schnelltest durch, ob $\hat{q}$ zu groß ist; er kommt mit BCD-Ziffern statt mit ganzen Langzahlen aus.

Die Idee dazu liefert der Fall, dass x drei- und y zweistellig ist. Dann ist $\hat{q}$ genau dann zu groß, falls

$$x_0 b^2 + x_1 b + x_2 \ < \ \hat{q}(y_1 b + y_2).$$

Umgeformt liefert dies

$$\hat{q} y_2 \ > \ b(x_0 b + x_1 - \hat{q} y_1) + x_2.$$

In dieser Form wird der Test durchgeführt. Falls der Klammerterm bereits größer als b sein sollte, ist die Bedingung offensichtlich nicht erfüllt (und wir stoppen hier, denn bei der Multiplikation mit b würden wir einen BCD-Überlauf erhalten). Falls ansonsten die Bedingung gilt, wird $\hat{q}$ um 1 erniedrigt und der Test erneut durchgeführt. So werden alle $\hat{q}$ abgefangen, die um zwei zu groß sind, und viele, die um eins zu groß sind (siehe [Knuth]). Bei einem langen Testlauf mit Zufallszahlen war das resultierende $\hat{q}$ noch in 0.002% der Fälle um eins zu groß.

2.4.7 Die Fakultät

Eine Funktion zur Berechnung der Fakultät ist bei der Implementation einer Langzahl-Integer-Arithmetik natürlich ohnehin Pflicht. Außerdem wird sie im Rahmen dieser Arbeit später benötigt bei der Auswertung der Standardfunktionen (Kapitel 4) über Potenzreihen, in deren Koeffizienten Fakultäten vorkommen. Die einzelnen Koeffizienten gehen zwar iterativ auseinander durch eine Multiplikation oder rückwärts durch eine Division auseinander hervor; zur Bestimmung der notwendigen Anzahl von Gliedern braucht man jedoch nicht vorn zu beginnen und benötigt eine höhere Fakultät als „Startwert“.

Die Fakultätsroutine wird auf die folgenden zwei Weisen gegenüber direktem Aufmultiplizieren aller Faktoren deutlich beschleunigt.

a) Zunächst wird versucht, möglichst viele der Faktoren (vom Typ `long int`) vorzumultiplizieren und nur ihr Produkt zu einer bestehenden Langzahl (vom Typ `BigInt`) hinzuzumultiplizieren. Beispielsweise ist das (bei 32-Bit-Integerzahlen) für vier aufeinander folgende Faktoren ≤ 216 oder für zwei aufeinander folgende Faktoren ≤ 46341 möglich.

   Mit der implementierten Multiplikation ist so bis ca. $n = 200$ (alles Vierfach-Vorprodukte) eine Beschleunigung auf das Doppelte zu verzeichnen. Ab ca. $n = 1000$ lohnt sich diese Version dagegen nicht mehr.

b) Leider wird so aber auch nie die Stärke der implementierten Multiplikation ($\mathcal{O}(n^{1.6})$) ausgespielt, da der zweite Faktor immer wesentlich kürzer als der erste sein wird. Damit die rekursive Berechnung ausgelöst und möglichst effektiv werden kann, muss der kürzere Faktor aber mindestens halb so lang sein wie der längere.

   Daher wird nun noch versucht, die Fakultät „von oben nach unten“ halbwegs gleichmäßig in möglichst große Faktoren aufzusplitten. Es findet dazu ein Vorschlag aus [HaPa] Verwendung.

Es werden zunächst alle vorkommenden Primfaktoren 2 abgespalten und die übrigbleibenden ungeraden Faktoren in Gruppen gegliedert:

$$n! = 2^{n-\sigma_2(n)} \prod_{k\geq 1} \left(\prod_{\frac{n}{2^k} < 2m+1 \leq \frac{n}{2^{k-1}}} (2m+1) \right)^k .$$

Dabei ist $\sigma_2(n)$ die Anzahl der Einsen in der Binärdarstellung von n und wird als eine schnelle Routine `numbits` implementiert, die n in Bytes zerlegt und dann mit einer Tabelle für die Werte von $0 \ldots 255$ arbeitet. Die Berechnung der Potenzen erfolgt mit der beschleunigten `pow`-Funktion aus Abschnitt 2.4.5.

Die Faktoren

$$p(n_1, n_2) := \prod_{n_1 < 2m+1 \leq n_2} (2m+1)$$

werden durch „binäre Teilung“ berechnet:

$$P(n_1, n_2) = P(n_1, n_m) \cdot P(n_m, n_2), \qquad \text{mit } n_m = \left\lfloor \frac{n_1 + n_2}{2} \right\rfloor,$$

wodurch eine relativ gleichmäßige Verteilung der auf die beiden Faktoren der Multiplikations-Routine erreicht wird. Wie schon in [HaPa] bemerkt, würde diese Teilung keine Vorteile bringen, wenn die Multiplikation mit der Ordnung $\mathcal{O}(n^2)$ implementiert wäre. Außerdem ist es günstiger, die Faktoren in Reihenfolge von absteigendem statt aufsteigendem k zu berechnen, da die beiden Operanden der Multiplikation dann ausgeglichenere Größenordnung haben.

Die Funktion, die $P(n_1, n_2)$ berechnet, verwendet die binäre Teilung, solange $n_2 - n_1 \geq 8$. Für kleinere Differenzen versucht sie wie die Version aus a), Vorprodukte innerhalb des 32-Bit-Bereichs von `long int` zu bilden.

Diese Version schlägt die aus Teil a) ab ca. $n = 200$ und gewinnt bei größeren Faktoren wegen der schnelleren Multiplikation deutlicher. Bei $n = 1000$ ist sie beispielsweise doppelt so schnell, bei $n = 10000$ sechsmal so schnell.

Die implementierte Routine benutzt dementsprechend für $n \leq 200$ Teil a), sonst b).

2.4.8 Diverse weitere Operationen

Vergleiche wurden in natürlicher Weise implementiert (zuerst werden Vorzeichen, dann Längen, bei Gleichheit dann von links nach rechts die Ziffern verglichen). `BCDstring` implementiert eine Funktion `compare` (mit Rückgabewerten analog zu `strcmp`), auf der Ebene von `BigInt` wird dies zu den Vergleichsoperatoren `==`, `>=`, etc. abstrahiert. Zusätzlich gibt es aus Gründen der Beschleunigung eine spezialisierte Vergleichsroutine mit `long`. Außerdem wurde ein stellenverschobener Vergleich implementiert, der für die Mantissen von `BigFloat` bei Nicht-Normalisierung verwendet und im entsprechenden Kapitel beschrieben wird.

Verschiebungen um dezimale Stellen sind einfach zu realisieren, wenn der Betrag der Verschiebung ein Vielfaches der Länge einer BCD-Ziffer ist, also bei den Standard-Parametern

von 4. Dann braucht nur wortweise der Speicher umkopiert zu werden. Bei Verschiebungen mit Rest 1, 2 oder 3 bei Division durch 4 müssen die BCD-Ziffern aufgeteilt werden, wozu leider jeweils eine Division mit Rest durch `BCDbase` notwendig ist. Es wurde überlegt, für den Fall, dass „nur“ zur Basis 10000 gearbeitet wird, hier Tabellen zu verwenden, die für alle 10000 möglichen BCD-Ziffern die um 1, 2 und 3 verschobenen Ziffern und hinausgeschobenen „Reste“ enthalten. Diese Tabellen würden aber schon sehr groß und müssten zu Beginn jedes Programms vorberechnet werden, weswegen bislang auf diese Möglichkeit verzichtet wurde.

Die Low-Level-Routinen aus `BCDstring` werden in `BigInt` in alle Operatoren umgesetzt, die es für die eingebauten Integer-Typen gibt, also die normalen arithmetischen Operatoren, die kombinierten Zuweisungsoperatoren, Prä- und Postinkrement- und -dekrement-Operatoren, Shift-Operatoren (um dezimale Ziffern), `div` (Quotient und Rest), Vorzeichen, Absolut-Betrag, Parität, etc.

Außerdem gibt es Umwandlungsroutinen von (`unsigned`) `long` und nach (`unsigned`) `long` (Letzteres löst bei zu großen Argumenten eine Exception aus). Es gibt spezialisierte beschleunigte Versionen all dieser Routinen für den Fall, dass der Typ `long` dem typischen 32-Bit-Dualformat entspricht (das wird in `XArithConfig.h` durch eine Präprozessor-Konstante definiert).

Zur Verwendung hauptsächlich von `BigFloat` aus gedacht sind Hilfsroutinen zum Bestimmen der Länge in Dezimalziffern (nicht BCD-Ziffern), zum Auslesen einzelner Dezimalziffern, zum Zählen von Nullen am niederwertigen Ende, zum Multiplizieren und Dividieren von BCD-Ziffern mit Zehnerpotenzen, etc.

Letztlich sind Ein- und Ausgaberoutinen von und in Strings oder Streams implementiert, wobei momentan immer die komplette Ganzzahl (ohne Breitenbeschränkung) ausgegeben wird.

Beispiel: Wie folgt kann man eine gerade positive Zahl ≥ 6 als Summe zweier Primzahlen erhalten. Die Tests sind dabei natürlich sehr primitiv und langsam und nur zur Demonstration gedacht, wie die Arithmetik-Routinen von einem `C++`-Programm aus zu verwenden sind!

```
bool is_prime(const BigInt &n)
 {
   if (n<=2 || n.is_even()) return false;
   if (n==3) return true;

   for ( BigInt f1(3) ;; f1+=2 )
    {
      BigInt r, f2=div(n,f1,r);
      if (r.is_zero()) return false;
      if (f2<=f1) return true;
    }
 }

bool goldbach(const BigInt &n, BigInt &s1, BigInt &s2)
 {
   assert( n>=6 && n.is_even() );

   for ( s1=3, s2=n-3; s1<=s2; s1+=2, s2-=2 )
      if ( is_prime(s1) && ( s1==s2 || is_prime(s2) ) )
```

```
        return true;
  return false;
}

void main()
{
  BigInt n,s1,s2;
  for (;;)
   {
     cout << "n="; cin >> n;
     if ( n>=6 && n.is_even() )
        if (goldbach(n,s1,s2))
           cout << n << " = " << s1 << " + " << s2 << endl;
        else
           cout << "widerlegt???\n";
   }
}
```

2.5 BigRational

Diese Klasse für rationale Zahlen ist gleichsam ein Nebenprodukt von `BigInt` und wird nicht für die Fließkommaarithmetik oder den Wiederberechnungsalgorithmus benötigt. Sie war zunächst gedacht für die Entwicklung einiger Standardfunktionen über Potenzreihen, deren Koeffizienten etwas kompliziertere Brüche sind (tan und tanh), die aus Geschwindigkeitsgründen aber nun anders implementiert sind.

Die Klasse benutzt für Zähler und Nenner `BigInt`, wobei immer der Zähler das Vorzeichen trägt. Nach jeder Operation wird automatisch gekürzt (Euklidischer Algorithmus). Es gibt Konstruktoren und Zuweisungsoperatoren von `BigRational`, `BigInt`, `long` und mit Paaren von `BigInt` bzw. `long`. Es werden Operatoren für die Grundrechenarten zur Verfügung gestellt, normal und mit Zuweisung kombiniert, sowie solche mit einem `BigInt`-Operanden. In der Klasse `BigFloat` gibt es außerdem die gerichtet gerundeten Grundrechenarten mit einem `BigRational`- und einem `BigFloat`-Argument. Der Ausgabeoperator verwendet das Format „Zähler/Nenner“.

Beispiel: Folgendes Programmstück berechnet rationale Näherungen an $\sqrt{2}$ und gibt sie als rationale Zahlen und als 16-stellige Dezimalbruch-Näherungen (zur nächsten Zahl gerundet) aus. Nach fünf Schritten gibt es 16-stellig keine Verbesserung mehr.

```
BigRational x(2);
BigFloat f, f1;

do
 {
   f1 = f;
   x = x/2 + 1/x;
   f = BigFloat(x,16);
   cout << x << " ~ " << f << endl;
 }
while ( f != f1 );
```

Die Ausgabe ist folgende:

```
3/2 ~ 1.5
17/12 ~ 1.416666666666667
577/408 ~ 1.41421568627451
665857/470832 ~ 1.41421356237469
886731088897/627013566048 ~ 1.414213562373095
1572584048032918633353217/1111984844349868137938112 ~ 1.414213562373095
```

2.6 BigFloat

2.6.1 Aufbau der Fließkommazahlen

Das Fließkommaformat unterscheidet sich in einem Detail von den ansonsten üblichen:

- Die Fließkommazahlen sind so organisiert, dass die Mantissen ganze Zahlen *größer oder gleich 1* sind, sie stehen also gleichsam immer *links* vom Dezimalpunkt. Dadurch ergibt sich gegenüber den üblichen Darstellungen mit Mantissen, die als Zahlen kleiner als 1 aufgefasst werden, eine Exponentenverschiebung um die Länge der Mantisse.

Statt in der Art $12.34 = 0.1234 \cdot 10^2$ (oder ggf. $= 1.234 \cdot 10^1$) wird bei uns also in der Art $12.34 = 1234 \cdot 10^{-2}$ dargestellt. Bei unbeschränktem Exponentenbereich sind die beiden Darstellungen äquivalent.

Genauer trägt die Mantisse in der Implementation zusätzlich das Vorzeichen, und für 0 wird als Mantisse 0 gewählt (und der Exponent ist in diesem Fall bedeutungslos).

Grund für diese Design-Entscheidung war zunächst hauptsächlich die Kompatibilität zu den Steins'schen Routinen. Außerdem erscheint es bei dynamisch langen Mantissen für den Programmierer natürlicher, sie nach links wachsen zu lassen.

Zusätzlich ist ein möglichst großer Exponentenbereich wünschenswert. Auf typischen 32-Bit-Rechnern macht es wenig Sinn, den entsprechenden Typ nur 16 Bit groß zu wählen. Die Mantissen werden (bei der Voreinstellung) ohnehin schon ein wenig speicherverschwenderisch verwaltet, und bei Rechnen mit Exponenten wird immer 32-bitweise gearbeitet, d.h. es würde jedes Mal ein Erweiterungsbefehl auf 32 Bit erzeugt. 64-Bit-Typen (`long long`) liefern dagegen einen Exponentenbereich, der nun wirklich für praktisch alle Anwendungen völlig überflüssig groß ist (Exponenten bis zehn hoch 9 Trillionen).

Das Vorzeichen des Exponenten wird bei den meisten hardwareunterstützten Arithmetiken in einem Exzess-Format dargestellt (bei IEEE-64 beispielsweise mit Charakteristik 1022). Das hat teilweise historische Gründe und hilft auch im Zusammenhang mit speziellen Exponenten-Werten, mit denen Unterläufe und Werte wie `NaN` und $\pm\infty$ dargestellt werden. Bei unserer Arithmetik macht das wenig Sinn, daher verwenden wir einen echt vorzeichenbehafteten Typen.

- Als Typ für die Exponenten wird daher intern ein vorzeichenbehafteter 32-Bit-Integertyp verwendet, typischerweise (`signed`) `long`.

Für verschiedene Abschätzungen ist es notwendig, dass die benutzten relativen Fehlerschranken (z.B. die der Rundungen) global gültig sind. Daher ist es nicht möglich, bei Unterläufen auf 0 zu runden oder irgendeine Form nicht normalisierter Zahlen zu verwenden (vergleiche aber Abschnitt 2.6.2 zur Normalisierung). Bei Überläufen könnte man zwar mit einem Wert analog `NaN` arbeiten; es kann dann aber ggf. sehr lang sinnlos mit diesem Wert weitergerechnet werden, oder es wären entsprechende Abfragen notwendig. Daher entscheiden wir für unsere Arithmetik wie folgt:

- Bei Über- und Unterläufen soll immer eine Exception geworfen werden.

Über- und Unterläufe schlagen sich in (ins Positive oder Negative) überlaufenden Exponenten nieder. Bei uns gibt es theoretisch eine zusätzliche Art von Überlauf, wenn die Mantisse die Maximallänge überschreiten soll. Derartiges dürfte aber normalerweise schon weit vorher durch einen Speichermangel auffallen. Daher wird bei uns in solchen Fällen eine Exception „overlong mantissa“ und kein eigentlicher Überlauf geworfen.

In `C++`, wie in den meisten anderen Hochsprachen, wird nun aber ein Überlauf bei Integer-Operationen nicht signalisiert, *weder* durch ein echtes, abfangbares Signal im Sinn einer Exception, *noch* durch einen speziellen Wert, *noch* durch Setzen einer globalen Variablen wie `errno`.

Für unsere Arithmetik ist es aber essentiell, auf Überläufe reagieren zu können, da man sonst einfach ohne Meldung falsche Ergebnisse erhalten würde. Daher muss *vor* einer Über- oder Unterlauf-gefährdeten Operation daraufhin getestet werden. Die Arithmetik zu [Steins] führt nicht konsequent solche Tests durch. Beispielsweise erhält man dort bei der Multiplikation von `1e2147483646` mit 10 den Wert `0.1e−2147483648`, d.h. der Exponentenbereich „überschlägt sich“ ungemeldet!

Wir könnten nun die entsprechenden Abfragen über den ganzen Code verteilen. Stattdessen implementieren wir für diesen Zweck eine eigene Klasse:

- Für die Exponenten wird eine Klasse `SafeLong` verwendet, die intern vor jeder Rechnung mit Exponenten auf Über- bzw. Unterlauf testet und ggf. eine Exception wirft.

Genauer wird der tatsächlich verwendete Typ namens `BF_et` („`BigFloat` exponent type“) in der Konfigurationsdatei `XArithConfig.h` per `typedef` festgelegt; voreingestellt ist eben `SafeLong`.

`SafeLong` implementiert fast alle Operationen, die es für `long` gibt (z.B. keine Verschiebungen), und kann bei Vergleichen, Ausgaben, Division und Modulo direkt an die eingebauten `C++`-Versionen weitergeben. Nur bei den Operatoren zu Addition, Subtraktion und Multiplikation muss es sich echt einschalten. Der Code wird natürlich ein wenig verlangsamt; die wirklich langsamen Operationen sind aber die auf den Mantissen, so dass sich der Effekt normalerweise nicht negativ bemerkbar macht.

Man kann aus Geschwindigkeitserwägungen natürlich auch einfach `long` verwenden, muss aber bedenken, dass die oben beschriebenen Effekte auftreten können. Wenn man völlig auf der sicheren Seite sein will, ist auch vorstellbar, für die Exponenten `BigInt` zu verwenden. Dann könnten erst Überläufe auftreten, wenn der *Exponent* mindestens 2^{31} Dezimalstellen

hat. Die Verwendung von `BigInt` würde aber einen ganz erheblichen Geschwindigkeitsverlust bedeuten. (Die Implementation ist darauf auch noch nicht vorbereitet; es müssten diverse Typumwandlungen nach `long` eingebaut werden. Ein besserer Schritt wäre vermutlich zunächst die Verwendung von `long long` in `SafeLong`.)

Die Daten-Member von `BigFloat` ergeben sich also einfach wie folgt:

```
class BigFloat
 {
   private:
      BigInt mant;
      BF_et expo;
   ...
 };
```

Nach außen hin gibt es zwei Routinen, den Exponenten abzufragen. `exponent` liefert den „physischen“ Exponenten, also den in der internen Darstellung wie oben beschrieben. `order` gibt den „abstrakten“ Exponenten zurück, und zwar – in Anlehnung an andere Systeme – so, als läge eine Darstellung mit Mantissen in $[1, 10[$ vor, also mit *einer* dezimalen Stelle *links* vom Komma (z.B. $\text{order}(9.876 \cdot 10^{-4}) = -4$). In den einleitenden Abschnitten wurde dagegen die Darstellung mit einer Mantisse kleiner als 1 als ästhetischer empfunden. Die Funktion konnte nicht den naheliegenderen Namen `char` (Charakteristik) tragen, da dieser in `C++` ein Schlüsselwort ist.

Es ist für $x \in \mathbb{R} \setminus \{0\}$, unabhängig von Fließkomma-Darstellungen, aber zu einer festen Basis $b \geq 2$, $\text{order}(x) \in \mathbb{Z}$ eindeutig definiert durch

$$b^{\text{order}(x)} \;\leq\; |x| \;<\; b^{\text{order}(x)+1}.$$

Es folgt $\text{order}(x) \leq \log_{10} |x| < \text{order}(x)+1$, also $\text{order}(x) = \lfloor \log_b |x| \rfloor$. In der Implementation berechnet sich der Wert aus Mantissenlänge und dem physischen Exponenten: ($m \in \mathbb{Z}^+$, $e \in \mathbb{Z}$):

$$\text{order}(m \cdot 10^e) = \lfloor \log_{10}(m \cdot 10^e) \rfloor = \lfloor e + \log_{10}(m) \rfloor = e + \text{length}(m) - 1.$$

2.6.2 Normalisierung

Es bleibt zu überlegen, was jeweils nach Abschluss einer arithmetischen Operation mit Nullen am rechten, niederwertigen Ende der Mantisse geschehen soll („trailing zeroes“). Man könnte sie (beispielsweise aus Geschwindigkeitsgründen) stehen lassen, nur komplette BCD-Nullen entfernen oder wirklich alle dezimalen Nullen entfernen (echtes Normalisieren).

Ohne Letzteres gibt es keine eindeutige Darstellung jeder darstellbaren Zahl. Eine andere Philosophie könnte allerdings die Mantissenlänge immer als die Anzahl definitiv bekannter Ziffern deuten und daher sogar ein Auffüllen mit entsprechend vielen Nullen fordern.

$$12314.5 \;=\; \boxed{1}\boxed{2}\boxed{3}\boxed{1}\boxed{4}\boxed{5}\boxed{0}\boxed{0}\;\texttt{E-3} \;=\; \boxed{0}\boxed{0}\boxed{1}\boxed{2}\boxed{3}\boxed{1}\boxed{4}\boxed{5}\;\texttt{E-1}$$

- Die Erkennung von dezimalen Nullen innerhalb einer BCD-Ziffer erfordert allgemein mehrere Divisionen mit Rest. Falls eine Entfernung notwendig ist, muss dann ein

Rechts-Shift um 1, 2 oder 3 durchgeführt werden, der bei jeder BCD-Ziffer wiederum eine Division mit Rest auslöst. Dadurch kann allein dieses Normalisieren dann beispielsweise wesentlich länger dauern als die vorangegangene eigentliche Operation (etwa eine Addition oder Subtraktion, die bei Überträgen ganz ohne Divisionen auskommt).

- Wenn gar nicht normalisiert wird, ergibt sich zwar bei jeder Operation ein direkter Geschwindigkeitsgewinn; andererseits ist schwer abzuschätzen, welche unnützen Aktionen die stehen gelassenen Nullen bei den nachfolgenden Berechnungen auslösen. Wenn immer gerichtet gerundete Operationen verwendet werden, ist die Länge dieses Anhängsels aber immer entsprechend beschränkt. Auch im späteren Auswertungsalgorithmus für arithmetische Ausdrücke kann es auf diese Weise kein unkontrolliertes Anwachsen von Stellenzahlen geben.

  Andererseits erschweren (und verlangsamen) nicht-normalisierte Zahlen die Fließkomma-Vergleiche (siehe dazu den nächsten Abschnitt).

  Ein Kompromiss (nur auf den Längeneffekt bezogen) ist das Entfernen nur von kompletten BCD-Nullen („0000“), die schnell erkannt und durch einfache Kopiervorgänge beseitigt werden können.

- Allerdings ergibt sich, wenn nicht echt normalisiert wird, möglicherweise eine versteckte Vergrößerung des Exponentenbereichs (um die Anzahl der Endnullen). Es könnten dann einige Over- und Underflows nicht auftreten, die bei einer Normalisierung ausgelöst würden.

  Problematisch wird das nur bei Algorithmen, die sich fest auf den Exponentenbereich verlassen. Ansonsten kann dieser Effekt sogar erwünscht sein, da möglicherweise einige sehr knappe Over- oder Underflows umgangen werden und dennoch numerisch korrekte Ergebnisse geliefert werden, sowohl bei Standardfunktionen, wie beim Wiederberechnungsalgorithmus.

  Bei so umgangenen Underflows ergeben sich *nicht* die numerischen Probleme wie im nicht-normalisierten Bereich anderer Fließkommasysteme, bei denen gleichsam ein Festkommasystem eingebettet wird und die Fehlerkonstanten (bei Rundungen etc.) keine globale Gültigkeit haben.

In der Implementation zu [Steins] wird nie normalisiert, außer durch expliziten Funktionsaufruf (dann dezimal). Bei uns stehen mehrere Möglichkeiten zur Verfügung, die besprochenen Effekte zu behandeln.

- Zum einen gibt es ebenfalls Methoden der Klasse `BigFloat`, die eine vollständige oder nur BCD-bezogene Normalisierung durchführen (`normalize` bzw. `normalize_simple`). Algorithmen, die nur an einigen Stellen den Exponentenbereich strikt beachtet sehen müssen, können die betroffenen Werte so zwangsweise normalisieren und dadurch ggf. eine Underflow-Exception auslösen.

- Das Verhalten der Arithmetik kann aber auch zur Compilationszeit fest eingestellt werden, durch die passende Definition der Präprozessor-Konstanten `NORMALIZE_MODE`

in der Konfigurationsdatei `XArithConfig.h`. Bei einem Wert 0 wird nie normalisiert (außer durch expliziten Aufruf), bei 1 nur BCD-weise und bei 2 nach jeder Operation.

Die Rechts-Shifts lassen sich nicht gut umgehen (siehe den `BigInt`-Abschnitt). Die Implementation der Erkennung der Nullen innerhalb der am weitesten rechts stehenden BCD-Ziffer, die nicht 0 ist, konnte aber – nur für die voreingestellte „kleine“ Basis 10000 – deutlich beschleunigt werden durch Einsatz einer Tabelle, die für jede der 10000 möglichen BCD-Ziffern vorberechnet die Anzahl der Nullen am rechten Ende enthält. Sie wird beim Start des Programms automatisch angelegt. Falls eine größere BCD-Basis (etwa 100000000) verwendet wird, fällt diese Möglichkeit offensichtlich aus, und es wird automatisch auf die Standard-Version mit Divisionen zurückgegriffen.

Durch diese Tabelle wird die Möglichkeit, immer zu normalisieren, auch im Normalfall problemlos. In Kapitel 6 finden sich entsprechende Vergleiche bei der Berechnung von Standardfunktionen und beim Wiederberechnungsalgorithmus. Gegenüber der einfachen Version ergibt sich oft keine messbare Verlangsamung, maximal eine um ca. 5%, gegenüber der Version ohne Normalisierung schlimmstenfalls 14%. Allerdings zeigt sich auch, dass es in diesen Beispielen auch ohne Normalisieren keinen nachteiligen Längenwachstumseffekt gibt.

2.6.3 Vergleiche

Vergleiche sind, was die Gesamtperformance angeht, eine nicht zu unterschätzende Größe. Es ist *nicht* sinnvoll, sie über eine Subtraktion zu definieren, da diese auch bei langen Mantissen immer vollständig durchgeführt würde, während ein stellenweiser Vergleich von links nach rechts bei der ersten nicht übereinstimmenden Stelle aufhören kann. (Wenn bei der Verwendung von `BigFloat` ein Vergleich und im Ungleichheitsfall die Differenz benötigt werden, sollte man also den Vergleich explizit durchführen; im Gegensatz zu der bei anderen Arithmetik-Typen eingelernten Programmiertechnik.)

Problematische Situationen ergeben sich, falls nicht standardmäßig normalisiert wird, dadurch, dass gleichwertige Mantissenstellen in den beiden Operanden gegeneinander verschoben sein können, schlimmstenfalls um ein Nicht-Vielfaches der BCD-Länge.

Beispiel: Wir vergleichen 123456.789 mit 123456.782, wobei letztere Zahl nicht normalisiert bei der Vergleichsroutine ankommen soll:

$123456.789 = 123456789 \cdot 10^{-3} =$ | 0 | 0 | 0 | 1 | | 2 | 3 | 4 | 5 | | 6 | 7 | 8 | 9 | `E-3`

$123456.782 = 1234567820 \cdot 10^{-4} =$ | 0 | 0 | 1 | 2 | | 3 | 4 | 5 | 6 | | 7 | 8 | 2 | 0 | `E-4`

Wenn die Exponenten übereinstimmen, brauchen natürlich nur die Mantissen als Integer-Zahlen miteinander verglichen zu werden, d.h. wir können einen Vergleichsoperator zu `BigInt` benutzen, der einen ziffernweisen Test von links nach rechts durchführt.

In der Version zu [Steins] wird ansonsten *immer* die Mantisse der Zahl mit größerem Exponenten um die Exponentendifferenz nach links verschoben und dann ein Vergleich durchgeführt. Der Nachteil ist, dass immer ein temporäres Objekt für die verschobene Mantisse angelegt und wieder freigegeben werden muss, und dass immer die komplette Mantisse verschoben wird, obwohl die Mantissen sich danach bereits in den ersten paar Stellen unterscheiden könnten. (Es wird vorher nicht einmal das Vorzeichen geprüft.)

Wir gehen daher anders wie folgt vor (das Vorgehen ist für die Operatoren `==`, `!=`, `>=`, `<=`, `>`, `<` jeweils zu spezialisieren):

- Zunächst werden die Vorzeichen (-1, 0, $+1$) überprüft, anhand derer ggf. bereits das Ergebnis des Gesamtvergleichs abgelesen werden kann.
- Wenn ansonsten die Exponenten der beiden Fließkommazahlen übereinstimmen, brauchen nur die beiden Mantissen durch einen normalen `BigInt`-Vergleich miteinander verglichen zu werden.
- Sonst liegen unterschiedliche Exponenten vor. Wenn nun die Exponentendifferenz *nicht* gleich der *negativen Differenz der Mantissenlängen* ist, steht das Ergebnis des Vergleichs ebenso bereits fest.
- Der schwierigste Fall ist der, dass Exponentendifferenz gleich der negativen Längendifferenz ist. Nur hier müssen tatsächlich die Mantissenstellen selbst betrachtet werden, und zwar auf jeden Fall gegeneinander verschoben. Im Beispiel oben ist die Exponentendifferenz $(-3)-(-4)=1$ und die Längendifferenz $9-10=-1$, also problematisch.

  Dieser Vergleich wurde tatsächlich stellenweise implementiert, ohne eine explizite Verschiebung durchzuführen. Wenn die Mantissen um ein Vielfaches von 4 gegeneinander verschoben sind, können die BCD-Ziffern günstigerweise direkt stellenweise verglichen werden. Ansonsten müssen die BCD-Ziffern des einen und des anderen Operanden jeweils in 2 Teile aufgesplittet und diese paarweise miteinander verglichen werden. Die Größe der Teile hängt vom Betrag der Verschiebung (modulo der BCD-Länge) ab.

  Im Beispiel oben würden also nacheinander miteinander verglichen:

  $$001/001, \quad 2/2, \quad 345/345, \quad 6/6, \quad 789/782,$$

  und hier würde ein Unterschied festgestellt. Für das Aufsplitten werden leider echte Divisionen mit Rest notwendig.

Durch dieses Vorgehen wurden die Routinen gegenüber den Steins'schen Versionen in den Fällen, dass ein Mantissenvergleich notwendig ist, deutlich beschleunigt. Beispielsweise ergibt sich bei gleichem Vorzeichen und 32 Dezimalstellen ein Faktor 6 bei Unterschied in der letzten Stelle, bzw. ein Faktor 21 bei Unterschied in der ersten Stelle. Bei sehr langen Mantissen und Unterschied an der ersten Stelle können natürlich beliebig große Faktoren erzeugt werden.

2.6.4 Gerichtete Rundungen

Die Rundungen auf eine angegebene dezimale Stellenzahl sind kanonisch zu implementieren, in den Details aber recht kompliziert. Da Verschiebungen beteiligt sind und diese ggf. pro BCD-Stelle eine Division mit Rest benötigen, ist ihr Anteil am Gesamtaufwand bei einem arithmetischen Ausdruck nicht zu unterschätzen.

Wie auch bei [Steins] sind die Rundungen mit den Operatoren `^`, `|`, `&` (nach oben, unten, zur nächsten Zahl), bzw. `^=`, `|=`, `&=` erreichbar. Zusätzlich wurde eine Funktion `enclose` implementiert, die eine `BigFloat`-Zahl in ein Intervall mit p-stelligen Grenzen einschließt,

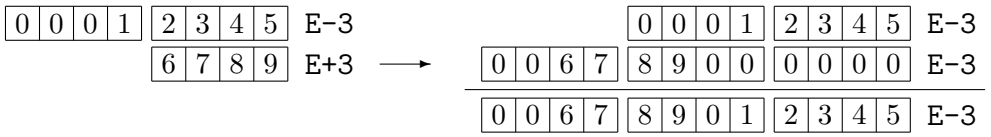
die (schneller) *nicht* nacheinander die Rundung nach oben bzw. unten aufruft, und die auch von der Intervall-Klasse aus benutzt wird. Außerdem werden die Rundungen intern am Ende der gerichtet gerundeten Multiplikation verwendet.

Wenn die angegebene Stellenzahl größer als die Mantissenlänge ist, wird die Mantisse einfach um die Differenz nach links geschoben. Wenn sie kleiner ist, kann bei einer positiven Zahl und Rundung nach unten, bzw. bei einer negativen Zahl und Rundung nach oben, nach rechts geschoben werden, und Ziffern am rechten Rand gehen verloren. Bei den jeweils umgekehrten Fällen muss vor dem Verschieben getestet werden, ob Ziffern ungleich 0 herausgeschoben werden würden, und nachher muss die verschobene Mantisse ggf. um ± 1 korrigiert werden. Wenn dabei wiederum ein Überlauf entsteht, d.h. die Mantisse wieder eine Stelle länger wird, muss leider noch einmal eine Stelle nachgeschoben werden. Bei der Rundung zur nächsten Zahl muss vor dem Verschieben für die führende Stelle des herausgeschobenen Teils getestet werden, ob sie größer oder gleich 5 ist.

2.6.5 Exakte Addition und Subtraktion

Bei Verwendung der Operatoren `+ - *` bzw. `+= -= *=` wird immer die exakte Operation durchgeführt.

Addition und Subtraktion können sehr leicht auf `BigInt`-Operationen zurückgeführt werden. Die Mantisse des Operanden mit dem größeren Exponenten muss um den Betrag der Exponenten-Differenz linksverschoben werden, dann findet Addition bzw. Subtraktion der Mantissen statt. Das Ergebnis hat als Exponent den kleineren der Exponenten der Operanden. Zu beachten ist, dass dabei die Exponenten-Arithmetik (die Differenz) eine Exception wegen eines Überlaufs werfen könnte.



Es ist hier aber allerhöchste Vorsicht geboten, da bei unbedachter Verwendung schnell extrem lange Mantissen entstehen können! Beispielsweise hat die exakte Summe $1 \cdot 10^{1000} + 1 \cdot 10^{-1000}$ bereits 2000 Stellen, während die Operanden-Mantissen nur einstellig waren. Da hier Exponenten im Milliarden-Bereich erlaubt sind, kann eine einzige solche Operation den Speicher zum Überlauf bringen.

Bei der Berechnung beispielsweise der Standard-Funktionen wird aus Genauigkeitsgründen an einigen Stellen *zwischenzeitlich* die exakte Addition oder Subtraktion verwendet. Dabei wird aber jeweils sichergestellt, dass die Operanden nicht völlig unterschiedliche Größenordnungen haben.

2.6.6 Gerichtet gerundete Addition und Subtraktion

Die gerichtet gerundeten Operationen sind (wie bei [Steins]) aus syntaktischen Gründen nur über Funktionsaufrufe wie `add_up(x,y,n)`, `sub_down(x,y,n)`, `mul(x,y,n)` (Letzteres rundet zur nächsten Zahl) erreichbar.

Alternativ wäre ein Design denkbar, bei dem man die Stellenzahl durch eine globale Variable setzt. Bei unserem späteren Wiederberechnungsverfahren ist es aber fast immer so, dass verschiedene Stellenzahlen innerhalb eines Ausdrucks gefordert sind. Außerdem ist diese Fassung etwas gefährlicher, da sich der Benutzer eventuell nicht an jeder Stelle darüber im Klaren ist, dass und mit wie vielen Stellen gerundet gerechnet wird.

In der Implementation zu [Steins] werden Addition und Subtraktion *immer exakt* durchgeführt und anschließend gerundet. Dadurch kann das oben beschriebene Phänomen der extrem langen Mantissen auftreten. Es wird dann sehr viel Hauptspeicher belegt und vom `C++`-Programm in einer Schleife mit Nullen gefüllt (was sehr lange dauern kann), und anschließend werden fast alle Stellen wieder weggerundet. Wenn nicht genügend Speicher vorhanden ist, stürzt das Programm ggf. mit einem Segmentation Fault ab (im Test hängte sich der L*i*nux-System sogar vollständig auf). Das kann beispielsweise im Wiederberechnungsalgorithmus geschehen, aber auch intern bei den Standardfunktionen. So wird etwa auch die Differenz bei $\sinh(x) = \frac{1}{2}(e^x - e^{-x})$ exakt berechnet, was sehr rasch zum obigen Effekt führt.

In unserer Implementation werden die Routinen zur gerundete Addition und Subtraktion (für die drei Rundungs-Richtungen) gesondert implementiert. Es ergibt sich in den sechs Kombinationen jeweils eine sehr umfangreiche Fallunterscheidung (über Exponentengrößen und Mantissenlängen und -lagen).

Wenn der Operand mit dem kleineren Exponenten hinter dem durch die gewünschte Stellenzahl beschriebenen Bereich liegt, braucht er gar nicht erst angeschaut zu werden (es muss aber natürlich der andere jeweils korrekt gerundet werden). Der Einfachheit halber wird der kleinere Operand nie aufgesplittet, d.h. wenn die gegeneinander verschobenen Mantissen aneinanderstoßen oder einander überlappen, wird eine exakte Addition oder Subtraktion durchgeführt und anschließend gerundet.

Beispiel: Beim folgenden Codestück

```
BigFloat x("1e+2140000000");
BigFloat y=div_up(1,x,16);
cout << add_up( x, y, 16 ) << endl;
cout << ( (x+y)^16 ) << endl;
```

ist die Ausgabe der ersten Operation

```
1.0000000000000001e+2140000000
```

und bei der zweiten wird eine Exception „overlong mantissa“ geworfen.

Aus den oben beschriebenen Gründen erschien diese Art der Implementation unumgänglich. Abgesehen davon ist es schwer zu beurteilen, ob dieses Vorgehen insgesamt statistisch zu einer Beschleunigung oder Verlangsamung gegenüber den exakten Versionen mit anschließender Rundung führt. In den Fallunterscheidungen sind nur wenige (Kurz-)Integer-Operationen mit Längen und Exponenten zusätzlich nötig, während ggf. eine lange Integer-Addition eingespart wird.

2.6.7 Multiplikation und Division

Die Fließkomma-Multiplikation wird zu einem Zweizeiler; sie wird auf die Multiplikation der Mantissen (`BigInt`) und die Kurz-Integer-Addition der Exponenten zurückgeführt. Bei der Multiplikation addieren sich die Mantissenlängen der beiden Operanden nur, so dass es nicht zu solch katastrophalen Effekten wie weiter oben bei Addition und Subtraktion kommen kann. Daher wird in unserer Implementation (wie bei der zu [Steins]) immer intern exakt multipliziert und anschließend gerundet.

Ebenso wird beim Fließkomma-Quadrieren mit `sqr` verfahren. Die *exakte* Potenzierung mit `pow` arbeitet mit der `pow`-Version von `BigInt` und liefert bei großen Exponenten natürlich auch sehr lange Mantissen. Die gerichtet gerundete Potenzierung arbeitet mit exp und ln und wird bei den Standardfunktionen behandelt.

Die Division kann im Allgemeinen ohnehin nicht exakt durchgeführt werden, so dass es nur gerichtet gerundete Versionen gibt. Die Mantisse des Dividenden wird für die gewünschte Stellenzahl passend verschoben, dann wird eine `BigInt`-Division der Mantissen durchgeführt. Die Division mit Rundung zur nächsten Zahl kann mit einer zusätzlichen Stelle und anschließendem Aufruf der Rundung durchgeführt werden. Bei den anderen muss nach der normalen Division der von der `BigInt`-Routine mitgelieferte Rest auf 0 getestet werden und je nach Rundung und Vorzeichen der Quotient um 1 erhöht oder erniedrigt werden.

Es gibt außerdem Routinen zur Inversion von ganzen Zahlen (`inverse_up(n,p)`, etc.), die intern den nach links verschobenen Zähler schneller produzieren. Sie finden bei den Koeffizienten der Potenzreihen bei den Standardfunktionen Verwendung und beschleunigen deren Berechnung zusätzlich.

2.6.8 Umwandlung von und nach IEEE

Natürlich dürfen Umwandlungsroutinen (als Typecasts oder `BigFloat`-Konstruktoren) mit dem auf dem Rechner bereits vorhandenen Fließkommaformat (nur `double`) nicht fehlen. In der Konfigurationsdatei `XArithConfig.h` wird festgelegt, ob es sich dabei um das IEEE-64-Format handelt oder nicht. Falls ja, werden spezialisierte Routinen verwendet, ansonsten langsamere Routinen, die mit der Arithmetik selbst arbeiten müssen. Außerdem wird dort die Byte-Reihenfolge auf dem jeweiligen System eingestellt, „little-endian“ (Intel) oder „big-endian“ (der Rest der Welt, insbesondere der SUN-Testrechner).

Die Routine `double`→`BigFloat` bei [Steins] testet leider nicht die speziellen Werte $\pm\infty$ und `NaN` und liefert bei der Umwandlung dann Phantasiewerte. Unsere Version zerlegt eigenständig das IEEE-Bitmuster und wirft in diesen Fällen eine Exception. Die gewünschte Stellenzahl und die Rundungsrichtung (-1, 0, $+1$) können als Parameter übergeben werden. Zusätzlich gibt es eine Version `enclose_double`, die die IEEE-Zahl in ein Intervall mit p-stelligen `BigFloat`s einschließt. Es macht nicht viel Sinn, Stellenzahlen viel größer als 16 anzugeben; nach den ersten 16 Stellen dürfte man meist nur noch Rundungs-Artefakte erhalten.

Bei `BigFloat`→`double` benutzt [Steins] einfach die `C`-Bibliotheksroutine `atof` (ASCII to float). Auf diese Weise kann, je nach Bibliotheksversion, ein Überlauf nicht festgestellt werden, und bei einem Unterlauf wird auf 0 gerundet. Außerdem stellte sich heraus, dass `atof`

zuweilen nicht exakt in den beiden niederwertigsten Bits arbeitet. Die Version zur vorliegenden Arbeit setzt daher eigenständig das IEEE-Bitmuster zusammen und wirft in obigen Fällen jeweils eine Exception. Da sie in der Hochsprache verfasst ist und exakter arbeitet, ist sie allerdings deutlich langsamer als `atof`. Alternativ könnte mit `atof` und Vorab-Vergleichen gearbeitet werden. Momentan gibt es nur eine Version, die zur nächsten `double`-Zahl rundet.

2.6.9 Diverse weitere Operationen

Zu Ein- und Ausgabe sei nur erwähnt, dass für das Lesen aus bzw. das Schreiben in Strings und Streams jeweils eine allgemeine Routine geschrieben wurde, die als Parameter die für das Lesen bzw. Schreiben eines Zeichens zuständige Funktion übergeben bekommt. Damit wird vermieden, dass Code dupliziert wird, und sichergestellt, dass das Lesen aus Strings bzw. Dateien sich nicht unabsichtlich unterschiedlich verhalten. Ähnlich wird bei den weiteren Datentypen verfahren werden.

Die Standardfunktionen mit `BigFloat`-Argumenten sind nicht innerhalb der Klasse selbst, sondern außerhalb definiert und benutzen nur die von außen zugänglichen Operationen. Sie werden im Kapitel zu den Standardfunktionen sehr ausführlich besprochen.

2.7 `BFInterval`

Diese Klasse implementiert abgeschlossene reelle Intervalle mit Grenzen, die durch `BigFloat`-Objekte dargestellt werden. Es ist dabei nicht notwendigerweise so, dass beide Grenzen dieselbe Mantissenlänge besitzen. Wenn nach jeder Operation normalisiert wird, wird das fast nie der Fall sein. Falls *nicht* normalisiert wird, haben Intervalle, die durch gerundete arithmetische Operationen erzeugt wurden, gleich lange Mantissen in Ober- und Untergrenze.

Die entsprechende Klasse in der Implementation zu [Steins] heißt `BigFloatInt`. Da es aber bereits Klassen `BigFloat` und `BigInt` gibt, erschien dieser Name etwas irreführend. Da es auch noch eine Klasse für komplexe Intervalle gibt, konnte nicht `BigInterval` gewählt werden.

Die einzigen benötigten Datenmember sind die beiden Grenzen.

```
class BFInterval
 {
   private:
      BigFloat Inf, Sup;
   ...
   public:
      const BigFloat & inf() const { return Inf; }
      const BigFloat & sup() const { return Sup; }
   ...
 };
```

Es gibt keine spezielle Darstellung für Punktintervalle. Alle Methoden stellen sicher, dass die Untergrenze immer kleiner oder gleich der Obergrenze ist. Von außen kann nicht eine einzelne Grenze manipuliert werden. Wie schon erwähnt, sortiert der Konstruktor aus zwei `BigFloat`-Grenzen diese im Bedarfsfall stillschweigend um; es gibt aber einen internen

Konstruktor ohne die entsprechende Überprüfung mit Absicherung über eine Assertion (zur Unterscheidung besitzt er ein drittes Dummy-Argument vom Typ `bool`).

Es gibt momentan nicht die Möglichkeit, ein leeres Intervall darzustellen. Die unterschiedlichen Durchschnitt-Operationen liefern bei leerem Durchschnitt als Signal entweder als Rückgabewert oder variablen Parameter ein `false`, die Versionen ohne solche Signalmöglichkeit werfen eine Exception.

2.7.1 Allgemeine Methoden

Es gibt die Operationen, die von einer Intervall-Arithmetik erwartet werden dürfen:

- Als Konstruktoren gibt es den Default-Konstruktor mit Wert 0 (Punktintervall) und die Typumwandlungs-Konstruktoren von `BigFloat`, `BFInterval`, `double`, `long` und analoge Zuweisungs-Operatoren. Es gibt Konstruktoren mit zwei `BigFloat`, zwei `double`, zwei `long` (automatische Sortierung); die analogen Zuweisungsmethoden tragen den Namen `set`. Bei allen Versionen mit `double` kann als optionaler Parameter die gewünschte Stellenzahl angegeben werden (Rundung nach außen).

- `mid` liefert den exakten Mittelpunkt (die Division durch 2 ist exakt darstellbar), `diam` den exakten Durchmesser (Vorsicht bei Grenzen weit auseinander liegender Größenordnungen!) und `diam(long p)` den auf `p` Stellen nach oben gerundeten Durchmesser.

- Die Methoden `contains` bzw. `is_contained_in` stellen fest, ob ein Punkt oder ein Intervall (ganz) im aktuellen Intervall enthalten sind. Wenn die Grenzen dabei nicht berührt werden dürfen, verwendet man `strictly_contains` und `is_strictly_contained_in`. Die Methode `intersects` testet, ob der Durchschnitt zweier Intervall nicht leer ist. Es gibt spezielle Versionen `is_zero`, `contains_zero`, `strictly_contains_zero` (die wegen der internen Darstellung der 0 bei `BigFloat` schneller arbeiten).

- Die Mengenoperationen Schnitt und konvexe Hülle heißen `intersection` und `intersect_with` (so wie `+=` zu verstehen), bzw. `conv_hull` und `build_hull_with`. Analog zu C-XSC gibt es zusätzlich die dazu gleichbedeutenden Operatoren `& &=` bzw. `| |=`. Letzteres darf nun nicht mit der Rundungs-Schreibweise von `BigFloat` verwechselt werden; für die Rundung nach außen auf `p` Stellen gibt es die Methoden `round` bzw. `rounded`.

- Die Methoden `x.absinf()` und `x.abssup()` geben das Infimum bzw. Supremum von $|x|$ zurück. Sie wurden implementiert, da bei den Wiederberechnungsmethoden häufig nur eine Grenze von $|x|$ benötigt wird und man so ohne ein temporäres Intervall-Objekt auskommt.

- Die Ausgabe-Routinen erzeugen das Format „`[` inf , sup `]`“ (überall ein Leerzeichen), auch bei Punktintervallen und immer mit der vollen Genauigkeit der Grenzen. Wenn weniger Stellen ausgegeben werden sollen, kann ja ein temporäres gerundetes Intervall verwendet werden. Die Leerzeichen wurden der Übersichtlichkeit halber verwendet, aber auch, da einige Terminals dann geschickter umbrechen können.

- Die Eingabe-Routinen (aus Strings oder Streams) akzeptieren die obige Form mit den eckigen Klammern (mit beliebig vielen oder gar keinen Leerräumen zwischen den Elementen), wobei bei einem Punktintervall das Komma und die zweite Grenze ausgelassen werden können. Die Versionen, die aus Dateien lesen, akzeptieren zusätzlich auch einfach zwei Fließkomma-Zahlen im `BigFloat`-Format, durch Leerraum getrennt.

2.7.2 Arithmetische Operatoren

Wie bereits bei `BigFloat` arbeiten die Operatoren `+ - +` und `+= -= *=` immer exakt. Bei Addition und Subtraktion sind also die gleichen Warnhinweise angebracht. Die nach außen gerundeten Operationen heißen `add`, `sub`, `mul div` bzw. `add_eq`, `sub_eq`, `mul_eq div_eq` und erhalten als letzten Parameter die gewünschte Stellenzahl. (Die Steins'schen Versionen von Addition und Subtraktion verwenden intern die exakten Operationen und runden anschließend, so dass es wiederum zu überlangen Mantissen kommen kann.)

Da eine spezielle Version der Multiplikation zum Einsatz kommt, kurz einige Vorbemerkungen.

Für eine Grundrechenart $\circ \in \{+, -, \cdot, /\}$ für Intervalle gilt immer

$$[\inf_1, \sup_1] \circ [\inf_2, \sup_2] = [\begin{array}{ll} \min & \{\inf_1 \circ \inf_2,\ \inf_1 \circ \sup_2,\ \sup_1 \circ \inf_2,\ \sup_1 \circ \sup_2\} \\ \max & \{\inf_1 \circ \inf_2,\ \inf_1 \circ \sup_2,\ \sup_1 \circ \inf_2,\ \sup_1 \circ \sup_2\} \end{array}],$$

Bei der Implementation auf dem Rechner müssen die Operationen für die neue Untergrenze natürlich ggf. nach unten, die für die Obergrenze nach oben gerundet werden.

Für Addition und Subtraktion müssen nicht alle vier reellen Operationen durchgeführt werden, so dass zur Implementation nichts weiter gesagt zu werden braucht:

$$\begin{aligned} [\ \inf_1, \sup_1\ ] + [\ \inf_2, \sup_2\ ] &= [\ \ \inf_1 + \inf_2,\quad \sup_1 + \sup_2\ \ ], \\ [\ \inf_1, \sup_1\ ] - [\ \inf_2, \sup_2\ ] &= [\ \ \inf_1 - \sup_2,\quad \sup_1 - \inf_2\ \ ]. \end{aligned}$$

Bei der Intervall-Multiplikation kann man die Anzahl der nötigen Multiplikationen durch Fallunterscheidungen über die Vorzeichen der Grenzen außer in einem Fall immer auf zwei begrenzen:

	$\sup_1 \le 0$	$\inf_1 \ge 0$	$\inf_1 < 0 < \sup_1$
$\sup_2 \le 0$	$[\sup_1 \sup_2\,,\ \inf_1 \inf_2\,]$	$[\sup_1 \inf_2\,,\ \inf_1 \sup_2\,]$	$[\sup_1 \inf_2\,,\ \inf_1 \inf_2\,]$
$\inf_2 \ge 0$	$[\inf_1 \sup_2\,,\ \sup_1 \inf_2\,]$	$[\inf_1 \inf_2\,,\ \sup_1 \sup_2\,]$	$[\inf_1 \sup_2\,,\ \sup_1 \sup_2\,]$
$\inf_2 < 0 < \sup_2$	$[\inf_1 \sup_2\,,\ \inf_1 \inf_2\,]$	$[\sup_1 \inf_2\,,\ \sup_1 \sup_2\,]$	s.u.

Der ungünstige verbleibende Fall ist der, dass beide Intervalle die Null im Innern enthalten:

$$\inf_1, \inf_2 < 0, \quad \sup_1, \sup_2 > 0;$$

dann gilt aus Vorzeichengründen offensichtlich

$$\begin{aligned} [\inf', \sup'] &:= [\ \inf_1, \sup_1\ ] \cdot [\ \inf_2, \sup_2\ ] \\ &= [\ \min\{\inf_1 \cdot \sup_2,\ \sup_1 \cdot \inf_2\},\ \max\{\inf_1 \cdot \inf_2,\ \sup_1 \cdot \sup_2\}\ ], \end{aligned}$$

so dass also immer noch alle vier Multiplikationen (und zwei Vergleiche) anstehen. Alle bekannten Implementationen arbeiten auch so.

Durch zusätzliche Vergleiche kann die Anzahl der Multiplikationen aber immer auf drei, in einigen Fällen auf nur zwei begrenzt werden. Voraussetzung für die Sinnhaftigkeit eines solchen Vorgehens ist allerdings, dass die Vergleiche um Größenordnungen schneller implementiert sind als die Multiplikationen, was bei der Langzahl-Arithmetik dieser Arbeit der Fall ist.

Das Vorgehen beruht auf folgender simpler Beobachtung:

1) $\sup_1 \leq |\inf_1|$

   1.1) $\sup_2 \geq |\inf_2| \implies \inf_1 \cdot \sup_2 \leq \sup_1 \cdot \inf_2 \implies \inf' = \inf_1 \cdot \sup_2$.

   1.2) sonst: $\sup_2 < |\inf_2| \implies \inf_1 \cdot \inf_2 > \sup_1 \cdot \sup_2 \implies \sup' = \inf_1 \cdot \inf_2$.

2) sonst: $\sup_1 > |\inf_1|$

   2.1) $\sup_2 \leq |\inf_2| \implies \sup_1 \cdot \inf_2 < \inf_1 \cdot \sup_2 \implies \inf' = \sup_1 \cdot \inf_2$.

   2.2) sonst: $\sup_2 > |\inf_2| \implies \sup_1 \cdot \sup_2 > \inf_1 \cdot \inf_2 \implies \sup' = \sup_1 \cdot \sup_2$.

In jedem der Fälle kann *eine* der Intervallgrenzen mit nur einer Multiplikation bestimmt werden, d.h. die Gesamtanzahl an notwendigen Multiplikationen ist auf drei gesunken.

Diese Beobachtung wurde zuerst veröffentlicht in [Heindl] (eleganter formuliert, aber etwas langsamer zu implementieren). Der dort entwickelte, in Pascal-XSC geschriebene Operator war im kritischen Fall gegenüber einer direkt vergleichbaren, ebenfalls in Pascal-XSC geschriebenen Version mit 4 Multiplikationen um einen Faktor 1.26 schneller, allerdings in den nicht kritischen Fällen gleich schnell und langsamer als der eingebaute Operator.

Letzteres mag teilweise daran liegen, dass *nicht* die Vorab-Unterscheidung durchgeführt wird, ob überhaupt der kritische Fall vorliegt, die ja allein die Vorzeichen der Grenzen (also prinzipiell ein Bit) zu testen hat. Es werden immer bei beiden Operanden Ober- und Untergrenze betragsmäßig verglichen. Außerdem werden bis zu 6 temporäre Objekte erzeugt, ggf. mit Vorzeichenwechsel gegenüber dem Original, was ja leider in unserer Arithmetik jeweils eine komplette Langzahl-Kopie erforderlich machen würde. Wir arbeiten daher genau wie oben beschrieben mit den Vorzeichentests aus der Tabelle und vergleichen nur im kritischen Fall.

Für die andere, durch obige Beobachtung noch nicht festgelegte Grenze sind so noch immer zwei Multiplikationen notwendig. Durch zwei weitere Vergleiche (zwischen Grenzen *unterschiedlicher* Intervalle) kann allerdings *manchmal* wiederum jeweils ein Produkt ausgeschlossen werden (statistisch in einem Drittel der Fälle). Ob allerdings der Einsatz dieser Zusatzentscheidungen noch lohnenswert ist, hängt stark von der Geschwindigkeit insbesondere der Vergleichsoperationen auf der Maschine ab.

a) für $\sup'$:

   a.1) $\sup_1 \geq |\inf_2|$:
   Falls $\sup_2 \geq |\inf_1| \implies \sup_1 \cdot \sup_2 \geq \inf_1 \cdot \inf_2 \implies \sup' = \sup_1 \cdot \sup_2$.

a.2) sonst ($\sup_1 < |\inf_2|$):
Falls $\sup_2 \leq |\inf_1| \implies \inf_1 \cdot \inf_2 > \sup_1 \cdot \sup_2 \implies \sup' = \inf_1 \cdot \inf_2$.

b) für $\inf'$:

b.1) $\sup_1 \geq \sup_2$:
Falls $\inf_2 \leq \inf_1 \implies \sup_1 \cdot \inf_2 \leq \inf_1 \cdot \sup_2 \implies \inf' = \sup_1 \cdot \inf_2$.

b.2) sonst ($\sup_1 < \sup_2$):
Falls $\inf_1 \leq \inf_2 \implies \inf_1 \cdot \sup_2 < \sup_1 \cdot \inf_2 \implies \inf' = \inf_1 \cdot \sup_2$.

Da bei uns Vergleiche (selbst wenn erst an der niederwertigsten Stelle ein Unterschied gefunden wird) wesentlich schneller sind als Langzahl-Multiplikationen, ergab sich, dass sich diese Zusatztests deutlich lohnen. Es wurden Geschwindigkeitsmessungen in den insgesamt $4 \cdot 4 = 16$ Fällen und in den 8 nicht kritischen Fällen durchgeführt.

Gegenüber einer Fassung, die immer mit vier Multiplikationen arbeitet, ergab sich etwa ein Geschwindigkeitsgewinn um einen Faktor 1.24 in den Fällen, in denen noch drei Multiplikationen notwendig sind, und um 1.78, falls man mit zweien auskommt (gewichtet gemittelt also um einen Faktor 1.42). Dabei bringen die zusätzlichen Vergleiche einen Faktor 1.04 ein.

Gegenüber der Fassung aus [Heindl] (übertragen in `C++` unter Verwendung von `BigFloat`) ergeben sich die Faktoren 1.58 bzw. 2.27 (gemittelt 1.81). In den nicht kritischen Fällen beträgt der gemessene Faktor 2.29. Wie nicht anders zu erwarten, ist der Heindl-Operator in dieser Arithmetik am schnellsten in den Fällen, in denen dort keine Vorzeichenwechsel notwendig sind (Faktor praktisch 1 bei drei Multiplikationen). Die genauen Messergebnisse sind in Kapitel 6 nachzulesen.

Bei der Intervalldivision tritt ein solch komplizierter Fall wie bei der Multiplikation einfach deswegen nicht auf, da das Nenner-Intervall die Null nicht enthalten darf. Man kommt so immer mit zwei Operationen aus:

	$\sup_1 \leq 0$	$\inf_1 \geq 0$	$\inf_1 < 0 < \sup_1$
$\sup_2 < 0$	$[\sup_1/\inf_2, \inf_1/\sup_2]$	$[\sup_1/\sup_2, \inf_1/\inf_2]$	$[\sup_1/\sup_2, \inf_1/\sup_2]$
$\inf_2 > 0$	$[\inf_1/\inf_2, \sup_1/\sup_2]$	$[\inf_1/\sup_2, \sup_1/\inf_2]$	$[\inf_1/\inf_2, \sup_1/\inf_2]$

Die Versionen der Standardfunktionen mit Intervallargument werden im entsprechenden Kapitel, jeweils im Anschluss an die Vorstellung der jeweiligen Version mit reellem Argument, besprochen. Die meisten realisierten Funktionen sind monoton oder haben nur wenige Extrema. Die kompliziertesten werden entsprechend die trigonometrischen Funktionen sein.

2.8 `BigComplex`

Die Klassen `BigComplex` und `BCInterval` finden aus Zeitgründen momentan noch nicht im eigentlichen Wiederberechnungsalgorithmus Verwendung. Im Hinblick auf eine mögliche Erweiterung wurden sie aber bereits in das Gesamtpaket der Klassen integriert. Sie werden hier entsprechend nur kurz beschrieben.

Auch Objekte von `BigComplex` werden durch ein Paar von `BigFloat`-Objekten beschrieben, hier durch Real- und Imaginärteil. Es gibt die typischen Konstruktoren mit keinem (0), einem (Imaginärteil 0) und zwei Argumenten, und entsprechende Zuweisungsoperatoren. Die einzige implementierte Rundung der ganzen komplexen Zahl rundet beide Komponenten zur nächsten Zahl des Rasters. Die Konjugation tauscht die Pointer auf die intern verwendeten `BCDstring`-Objekte aus und vermeidet temporäre Objekte. Addition, Subtraktion und Multiplikation können wieder exakt durchgeführt werden.

Für das Ausgabeformat kann bei der Methode `output` durch einen Parameter aus zweien ausgewählt werden, „`( re , im )`“ oder im Stil „re ± im`*i`“, wobei die Spezialfälle, dass eine oder beide Komponenten 0 oder der Imaginärteil betragsmäßig 1 sind, geeignet gesondert behandelt werden. Der Ausgabeoperator `<<` verwendet immer die Klammer-Version. Die Eingaberoutine aus Strings oder Streams akzeptiert die Klammerschreibweise, wobei das Komma und der Imaginärteil ausgelassen werden können. Beim Lesen aus Dateien werden zusätzlich zwei aufeinander folgende, durch Leerräume getrennte Fließkommadarstellungen angenommen.

2.9 `BCInterval`

Die Klasse `BCInterval` stellt komplexe Intervalle im Sinn des Produkts zweier reeller Intervalle, also Rechtecke in der komplexen Ebene dar. Die Objekte werden sinnvollerweise intern nicht als Paare von komplexen Zahlen (linke untere, rechte obere Ecke) dargestellt, sondern als Paar von reellen Intervallen. Die komplexe Struktur repliziert sich dann bei dieser Klasse genau wieder. Die andere Darstellungsmöglichkeit ist nicht natürlich, da die Objekte bei fast allen Operationen in ihre Grundbestandteile auseinander genommen werden müssten und die komplexe Struktur der Komponenten nicht genutzt werden kann (während man andersherum die Intervallstrukturen verwenden kann).

Es gibt die wünschenswerten Konstruktoren aus `BigFloat`, `BFInterval`, `BigComplex` und `BCInterval`. Die Methoden aus `BFInterval` werden in geeigneter Implementation auch hier implementiert. Die Durchmesserfunktion `get_diam` liefert beispielsweise den Durchmesser in reeller und in komplexer Richtung über Referenzparameter. Die arithmetischen Grundoperationen sind wieder exakt (außer Division) und nach außen gerundet vorhanden.

Der Betrag `abs` einer komplexen Zahl bzw. die Intervallversion davon sind über $|x + iy| = \sqrt{x^2 + y^2}$ implementiert, wobei das Quadrieren exakt durchgeführt wird, die Addition und die Wurzel mit zwei zusätzlichen Schutzziffern. Außerdem gibt es `arg` für die Bestimmung des Polarwinkels. (Die Herleitung zu beidem findet sich am Ende des Kapitels 4 zu den Standardfunktionen.)

Das Ausgabeformat schachtelt das Intervall-Format mit eckigen Klammern in das komplexe mit runden Klammern, also „`( [` re_1 `,` re_2 `] , [` im_1`,` im_2 `] )`“. Die Eingaberoutinen akzeptieren aber auch die umgekehrte Schachtelung „`[ (` re_1 `,` im_1 `) , (` re_2 `,` im_2 `) ]`“. Es kann jeweils das Komma und die zweite Komponente ausgelassen werden, beispielsweise „`[(1)]`“ (= Punktintervall reell 1). Aus Sicherheitsgründen werden vier aufeinander folgende Fließkommawerte *nicht* akzeptiert.

An komplexen Intervall-Standardfunktionen (mit komplexen Intervallen als Argumente) sind bisher nur zu Illustrationszwecken die einfachen Fälle, nämlich die separablen Funktionen exp, sin, cos, sinh und cosh implementiert. Ihr Real- und Imaginärteil ist jeweils als Produkt von zwei Funktionen nur des Real- bzw. nur des Imaginärteils darstellbar. Daher können ihre Intervallversionen durch intervallmäßige Auswertung dieses Produktausdrucks implementiert werden, ohne dass dabei Überschätzungen auftreten. Die Herleitung der Beziehungen beispielsweise für den komplexen Tangens wird dagegen ganz extrem aufwändig (siehe z.B. [Braune], Seiten 139-164), was den Rahmen dieser Arbeit sprengen würde. Die anderen Standardfunktionen bleiben daher einer zukünftigen Erweiterung vorbehalten.

3 Berechnung reeller Konstanten

Bei der Auswertung der Standardfunktionen werden an mehreren Stellen verschiedene reelle Konstanten, reell approximiert oder als Einschließungen, mit beliebig hoher Stellenzahl benötigt. Beispielsweise braucht man zur Argumentreduktion bei der Exponentialfunktion $\ln 10$, bei den trigonometrischen Funktionen π und beim natürlichen Logarithmus $\ln 2$, $\ln 5$ und $\ln 10$. Außerdem sollten e und π auch für den späteren Anwender interessant sein; sie können auch in die auszuwertenden arithmetischen Ausdrücke eingebaut werden und werden dann je nach Bedarf automatisch genau genug berechnet.

Für die Konstanten wurden analog zu [Steins] eigene Berechnungsroutinen implementiert. Die jeweilige Konstante wird nicht bei jedem Aufruf neu berechnet, vielmehr wird der Wert mit der bisher größten Stellenzahl intern weiter gespeichert und auf die gewünschte neue Stellenzahl gerundet zurückgeliefert. Außerdem liegt jede Konstante bereits beim Programmstart mit 100 Stellen vorberechnet vor.

Wir bemühen uns dennoch auch hier um eine Beschleunigung. Die Konstanten müssen natürlich vergleichsweise selten neu berechnet werden, und noch seltener mit sehr vielen Stellen. Wenn dies aber doch einmal geschieht, ist es angenehm, wenn die „Schrecksekunden“ entfallen, bevor ein Programm, das Standardfunktionen benutzt, überhaupt beginnen kann zu arbeiten! Beispielsweise benötigt die Implementation zu [Steins] (auf dem SUN-Rechner) für die Berechnung von π auf 1000 Stellen 43 Minuten, unsere Implementation 1,6 Sekunden.

3.1 Kettenbrüche

Durch die Forderung nach einer möglichen sukzessiven Erhöhung der Genauigkeit wäre es sehr ungünstig, Potenzreihen zur Berechnung heranzuziehen. Man kann nicht einfach weitere Glieder hinzunehmen, da ja alle Information über die hinter der Rundungsstelle gelegenen Ziffern aus den beim letzten Mal berechneten Gliedern vollständig verloren gegangen ist. Die Potenzreihe wäre immer ganz neu zu berechnen, und immer müsste die Anzahl zu berechnender Glieder und die notwendige Stellenzahl neu bestimmt werden.

Aus diesem Grund bieten sich Kettenbrüche zur Berechnung dieser Konstanten an. Zähler und Nenner aus dem jeweils letzten Schritt werden aufbewahrt, und bei einer Neuberechnung wird mit ihnen wieder neu aufgesetzt. Außerdem nähern sich Kettenbruchapproximationen dem wahren Wert abwechselnd von unten bzw. von oben an, liefern also auf natürliche Weise eine Einschließung. Es sind also in jedem Schritt zwei aufeinander folgende neue Kettenbrüche zu bestimmen, und als Abbruchkriterium kann – nach Durchführung zweier Fließkommadivisionen mit der gewünschten Stellenzahl – direkt die Hochgenauigkeit des entstehenden Intervalls dienen.

Die Division ist nun aber leider die mit Abstand langsamste Grundoperation der Arithmetik. Das ist auch mit Grund dafür, Kettenbrüche nicht für die Approximation der Standardfunktionen heranzuziehen. Die Implementation zu [Steins] führt in jedem Schritt zwei solche Divisionen durch und testet die Abbruchbedingung. Man erreicht aber schon eine deutliche Geschwindigkeitssteigerung, wenn man diese Division und den Durchmessertest nur z.B.

jedes zweite oder vierte Mal oder noch seltener durchführt, selbst wenn man dadurch am Schluss eventuell einige Brüche zu viel berechnet haben sollte.

In Erweiterung davon wird in der vorliegenden Implementation eine Mindestanzahl von Iterationen *geschätzt* und bis zu deren Erreichen *gar keine* Division durchgeführt. Hiermit wurden im Bereich bis $p = 1000$ Geschwindigkeitssteigerungen um Faktoren bis über 5000 erzielt.

Da wir *Konstante* berechnen, also keine Abhängigkeit von einem Argument x vorliegt, und wir also immer denselben Kettenbruch berechnen, wurde hierfür keine theoretische Fehlerabschätzung betrieben. Die Schätzung beruht auf einer Tabellierung der benötigten Schritte für etwa $p = 10$ bis 10000.

Für unsere Implementation benutzen wir die Kettenbrüche nur an dieser nicht zentralen Stelle und verweisen daher für Details und ausführliche Herleitungen der verwendeten Darstellungen z.B. auf [Wall], Abschnitt 90. Um die Besonderheiten bei den einzelnen Konstanten aber beschreiben zu können, erinnern wir hier kurz an übliche Schreibweisen und Beziehungen.

Ein Kettenbruch $q = [b_0, a_1 : b_1, a_2 : b_2, \ldots]$ (mit $a_i, b_i \in \mathbb{Z}_0^+$), ist definiert als die (endliche oder unendliche) Folge der Teilbrüche

$$q_0 = b_0, \quad q_1 = b_0 + \frac{a_1}{b_1}, \quad q_2 = b_0 + \frac{a_1}{b_1 + \frac{a_2}{b_2}}, \quad q_3 = b_0 + \frac{a_1}{b_1 + \frac{a_2}{b_2 + \frac{a_3}{b_3}}}, \quad \ldots$$

Formal ist die Definition für $a_i, b_i \in \mathbb{R}$ oder $\in \mathbb{C}$ sinnvoll, für $a_i, b_i \in \mathbb{Q}$ bei einigen Standardfunktionen auch in der Praxis. Die am häufigsten betrachteten Kettenbrüche arbeiten aber mit $a_i, b_i \in \mathbb{Z}$ (oder $\mathbb{N}_0$); so wollen wir es hier auch verstehen. Falls ein Nenner 0 wird, ist der entsprechende Teilbruch nicht definiert.

Der Kettenbruch heißt konvergent, falls die Folge der Teilbrüche konvergent ist (und nur endlich viele Nenner 0 sind). Unter dem Wert des Kettenbruchs ist dann der Grenzwert der Folge zu verstehen; auch er wird als $[b_0, a_1 : b_1, a_2 : b_2, \ldots]$ geschrieben. Wir setzen hier immer implizit Konvergenz voraus. Ein Kettenbruch heißt einfach, falls alle $a_i = 1$ sind. Die Klammerschreibweise wird dann zu $[b_0, b_1, b_2, b_3, \ldots]$ vereinfacht.

Teilbrüche q_j mit geradem j heißen gerade Teilbrüche, solche mit ungeradem j ungerade Teilbrüche. Für $a_i, b_i \in \mathbb{N}$ nähern sich die geraden Teilbrüche monoton wachsend von unten dem Wert des Kettenbruchs, die ungerade monoton fallend von oben. Ein gerader und der folgende ungerade Teilbruch bilden also eine Einschließung des Grenzwerts.

Man kann die Teilbrüche $q_j = \frac{A_j}{B_j}$ auf einfache Weise iterativ berechnen:

$$\begin{array}{rll} & A_0 := b_0, & B_0 := 1, \\ & A_1 := b_1 A_0 + a_1, & B_1 := b_1 B_0, \\ \text{für } j \geq 2: & A_j := b_j A_{j-1} + a_j A_{j-2}, & B_j := b_j B_{j-1} + a_j B_{j-2}. \end{array} \tag{3.1}$$

3.2 Berechnung von e

e hat folgende Darstellung als einfacher Kettenbruch mit einer Art Pseudo-Periode (vgl. [Wall], Abschnitt 91):

$$e = 2 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{4 + \ldots}}}}}, \qquad e = [2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10, \ldots].$$

Die ersten Einschließungen sind

$$[2, 3],\ [\tfrac{8}{3}, \tfrac{11}{4}],\ [\tfrac{19}{7}, \tfrac{87}{32}],\ [\tfrac{106}{39}, \tfrac{193}{71}],\ [\tfrac{1264}{465}, \tfrac{1457}{536}] \subseteq [2.7182^{836}_{795}].$$

Die maximale Länge von Nenner und Zähler ist etwa die halbe geforderte Stellenzahl.

Die Implementation zu [Steins] benutzt den oben angegebenen iterativen Auswertungsalgorithmus und berechnet in jedem Schritt b_j mit einer Fallunterscheidung.

Da die Pseudo-Periode der Darstellung 3 ist und wir jeweils zwei aufeinander folgende Näherungen für eine Einschließung brauchen, liegt es aber nahe, je *sechs* Schritte zu einem neuen zusammenzufassen („Clustern“). Statt 6 Langzahl-Additionen und 2 Multiplikationen mit einer Kurzzahl benötigt man so nur noch 2 Langzahl-Additionen, allerdings 4 Multiplikationen mit einer Kurzzahl. Es wird so Rechenaufwand von der Langzahl-Arithmetik in die normale Integer-Arithmetik verschoben, und der zusätzliche Aufwand bei den Kurzzahl-Koeffizienten fällt praktisch nicht ins Gewicht. Der Gewinn beträgt insgesamt etwa 20 bis 30 %.

Jeder solche zusammengefasste Schritt startet mit einem Index $j = 2+6n$, $n = 0, 1, 2, \ldots$. Es ist $b_{j+1} = b_{j+2} = 1$, und $b_j = \frac{2}{3}(j+1)$ entsteht im Programm am besten durch Heraufzählen $2, 6, 10, 14, \ldots$. In diesem Schritt sei $f := b_j$ abgekürzt; dann gilt:

$$\begin{array}{lcllcl} A_j &=& f \cdot A_{j-1} + A_{j-2}, & A_{j+1} &=& A_j + A_{j-1}, \\ A_{j+2} &=& A_{j+1} + A_j, & A_{j+3} &=& (f+2) \cdot A_{j+2} + A_{j+1}, \\ A_{j+4} &=& A_{j+3} + A_{j+2}, & A_{j+5} &=& A_{j+4} + A_{j+3} \end{array}$$

(analog für die B_j). Einsetzen liefert die Zähler A_{j+4}, A_{j+5} (analog die Nenner B_{j+4}, B_{j+5}) der neuen Unter- bzw. Obergrenze:

$$\begin{array}{lcl} A_{j+4} &=& 2(f^2 + 4f + 2)A_{j-1} + (2f + 7)A_{j-2}, \\ A_{j+5} &=& (4f^2 + 14f + 7)A_{j-1} + (4f + 12)A_{j-2}. \end{array}$$

Dabei geht der Koeffizient $c_{22} := 4f^2 + 14f + 7$ schnell (und ohne Überlaufgefahr, s.u.) aus dem Koeffizienten $c_{11} := 2(f^2 + 4f + 2)$ hervor als $c_{22} = 2(c_{11} - f) - 1$.

Da wir für die Koeffizienten mit normalen 32-Bit-Integern auskommen wollen, muss $f \leq 23168$ gelten, also $j \leq 34752$. Dadurch liegt die maximale dezimale Stellenzahl bei etwas über $p = 70000$ – da dies ohnehin völlig unrealistisch groß ist, wirft die Routine bei $p > 70000$ direkt eine Exception „internal limitation“. Bei wirklichem Bedarf könnte man (ab Erreichen dieser Stelle) `BigInt` statt `long` verwenden.

Es ist nicht sinnvoll, noch mehr Schritte zu clustern, da die Koeffizienten dann zu groß werden und schon wesentlich früher der Integer-Bereich überschritten wird.

Wie oben bereits angedeutet, wird eine Mindestzahl von Iterationen geschätzt, bevor überhaupt eine Division stattfindet. Momentan wird $n_{est} = 1.75 \cdot p^{0.85}$ verwendet (wobei diese Operationen schnell mit der unterliegenden Systemarithmetik, z.B. IEEE, durchgeführt werden). Im Bereich von $p = 10$ bis $p = 1000$ wird so selten ein Schritt zu viel, maximal und sehr selten zwei Schritte zu viel durchgeführt. Beispielsweise ist bei $p = 1000$ Stellen $n_{est} = 620$, d.h. es werden 103 zusammengefasste Iterationsschritte ohne Division durchgeführt (einer zu viel).

Wir benötigen auf dem Linux-Testrechner für $p = 1000$ etwa 0.015 Sekunden, unsere spätere Implementation der Exponentialfunktion benötigt auf $x = 1$ angewandt mit 0.105 Sekunden das Siebenfache. Die Implementation zu [Steins] braucht 156 Sekunden, mit der Exponentialfunktion dagegen nur 49 Sekunden. (Für die Messungen muss der Code natürlich so manipuliert werden, dass die Konstanten immer vollständig neu berechnet werden.)

Unsere Routine heißt

```
static BFInterval BFInterval::e(long precision);
```

3.3 Berechnung von π

π wird zur Argumentreduktion bei den trigonometrischen Funktionen benötigt.

Zur Auswertung verwenden wir $\pi = 4 \cdot \arctan 1$ und für $\arctan x$ folgende Kettenbruchdarstellung (vgl. [Wall], Abschnitt 90):

$$\arctan x = \cfrac{x}{1 + \cfrac{1x^2}{3 + \cfrac{4x^2}{5 + \cfrac{9x^2}{7 + \cfrac{16x^2}{9 + \cdots}}}}} \qquad \begin{aligned} \arctan(x) &= [0, x:1, (1x)^2:3, (2x)^2:5, (3x)^2:7, (4x)^2:9, \ldots] \\ \arctan(1) &= [0, 1:1, 1^2:3, 2^2:5, 3^2:7, 4^2:9, \ldots] \end{aligned}$$

Die ersten Einschließungen von π sind

$$4[0,1],\ 4[\tfrac{3}{4}, \tfrac{19}{24}],\ 4[\tfrac{160}{204}, \tfrac{1744}{2220}],\ 4[\tfrac{23184}{29520}, \tfrac{364176}{463680}],\ 4[\tfrac{6598656}{8401680}, \tfrac{135484416}{172504080}] \subseteq [3.1415^{94}_{88}].$$

Die Länge von Zähler und Nenner beträgt im relevanten Bereich etwa das 1.3-fache der gewünschten Stellenzahl.

Die Approximation läuft also nach folgendem Schema ab:

$A_0 := 0, \qquad B_0 := 1$
$A_1 := 1, \qquad B_1 := 1$
$j := 0$
repeat
  $j := j + 2$
  $A_j := (2j-1) \cdot A_{j-1} + (j-1)^2 \cdot A_{j-2}, \qquad B_j := (2j-1) \cdot B_{j-1} + (j-1)^2 \cdot B_{j-2}$
  $A_{j+1} := (2j+1) \cdot A_j + j^2 \cdot A_{j-1}, \qquad B_{j+1} := (2j+1) \cdot B_j + j^2 \cdot B_{j-1}$
until $accurate_p([\bigtriangledown_p(4A_j/B_j), \bigtriangleup_p(4A_{j+1}/B_{j+1})])$

Man kann leider diesmal schlecht mehrere Schritte zu einem zusammenfassen, da j in den Koeffizienten in der vierten Potenz vorkäme. Bei 32-Bit-Integerzahlen läge die Grenze bei $j = 142$, was bei der Forderung nach 107 Dezimalstellen bereits der Fall wäre (und 100 Stellen sind ohnehin fest eincodiert).

Als Schätzwert für die Mindestzahl an Schritten wird $n_{est} = 1.342 \cdot p^{0.9967}$ verwendet. Für $p = 500$ benötigt die vorliegende Implementation auf dem L*i*nux-Rechner 0.17 Sekunden, die zu [Steins] 283 Sekunden (auf der SUN-Workstation 0.38 zu 300 Sekunden).

Für $p \geq 35700$ werfen wir eine Exception „internal limitation“, da die Koeffizienten den 32-Bit-Bereich verlassen. Die Routine heißt

```
static BFInterval BFInterval::pi(long precision);
```

Bemerkung: Über das Additionstheorem des Arcustangens

$$\arctan x_1 + \arctan x_2 = \arctan \frac{x_1 + x_2}{1 - x_1 x_2}$$

lassen sich weitere Darstellungen von $\frac{\pi}{4}$ mit kleineren Argumenten von arctan herleiten (von Machin bzw. Meissel, siehe z.B. [Atlas], S. 308):

$$\begin{aligned} \tfrac{\pi}{4} &= \arctan 1 \\ &= 4 \arctan \tfrac{1}{5} - \arctan \tfrac{1}{239} \\ &= 8 \arctan \tfrac{1}{10} - 4 \arctan \tfrac{1}{515} - \arctan \tfrac{1}{239} \end{aligned}$$

Durch das gebrochene Argument muss man aber (direkt oder indirekt) mit rationaler Arithmetik arbeiten, und die beteiligten Zähler und Nenner werden ohne aufwändiges Kürzen rasch sehr groß. Die obigen Darstellungen sind für Potenzreihen geeigneter, für unsere Kettenbruchdarstellungen weniger gut.

3.4 Berechnung von $\ln 2$, $\ln 5$ und $\ln 10$

Die Konstanten $\ln 2$ und $\ln 5$ werden für die Argumentreduktion beim natürlichen Logarithmus, $\ln 10$ für die Argumentreduktion bei der Exponentialfunktion benötigt.

Der natürliche Logarithmus ist durch folgenden (nicht einfachen) Kettenbruch darstellbar (vgl. [Wall], Abschnitt 90):

$$\ln(1+x) = \cfrac{x}{1+\cfrac{1^2x}{2+\cfrac{1^2x}{3+\cfrac{2^2x}{4+\cfrac{2^2x}{5+\cfrac{3^2x}{6+\ldots}}}}}}, \quad \begin{aligned} \ln(1+x) &= [0, x:1, 1^2x:2, 1^2x:3, 2^2x:4, 2^2x:5, \ldots], \\ \ln(10) &= [0, 9:1, 1^2 9:2, 1^2 9:3, 2^2 9:4, 2^2 9:5, 3^2 9:6, \ldots] \end{aligned}$$

Die ersten Einschließungen sind damit:

$$\begin{aligned} \ln 2: &\quad [0,1], [\tfrac{2}{3}, \tfrac{7}{10}], [\tfrac{9}{13}, \tfrac{52}{75}], [\tfrac{131}{189}, \tfrac{1073}{1548}], [\tfrac{445}{642}, \tfrac{14161}{20430}], [\tfrac{34997}{50490}, \tfrac{37981}{54795}] &&\subseteq [0.6931471^{85}_{57}] \\ \ln 5: &\quad [0,4], [\tfrac{4}{3}, \tfrac{20}{11}], [\tfrac{36}{23}, \tfrac{260}{159}], [\tfrac{476}{297}, \tfrac{4892}{3033}], [\tfrac{2900}{1803}, \tfrac{117436}{72945}], [\tfrac{204436}{127035}, \tfrac{1141052}{708945}] &&\subseteq [1.609^{51}_{28}] \\ \ln 10: &\quad [0,9], [\tfrac{18}{11}, \tfrac{45}{14}], [\tfrac{99}{7}, \tfrac{900}{361}], [\tfrac{4473}{1991}, \tfrac{55611}{23684}], [\tfrac{136125}{59522}, \tfrac{82449}{35630}], [\tfrac{1175319}{511390}, \tfrac{6014781}{2608835}] &&\subseteq [2.298, 2.306] \end{aligned}$$

Man sieht bei $\ln 10$ die vergleichsweise langsame Konvergenz. Für $p = 100$ werden z.B. 360 Schritte (mit zum Schluss 768 Stellen von Zähler und Nenner) benötigt.

Daher wird $\ln 10$ in der vorliegenden Implementation als $\ln 2 + \ln 5$ berechnet, was gegenüber der direkten Version bei Stellenzahlen unter 50 ca. 20%, bei 200 Stellen bereits 60% Geschwindigkeitsgewinn einbringt (in der Praxis mehr, da $\ln 2$ und $\ln 5$ ja möglicherweise aus anderen Gründen bereits berechnet wurden). Für den entstehenden Rundungsfehler gilt (mit den offensichtlichen Bezeichnungen):

$$|\varepsilon_{10}| = \left|\frac{\widetilde{\ln}\,10 - \ln 10}{\ln 10}\right| = \left|\frac{\varepsilon_2 \ln 2 + \varepsilon_5 \ln 5}{\ln 10}\right|.$$

Nun ist $\ln 2 \approx 0.69$, $\ln 5 \approx 1.61$, daher setzen wir so an, dass (bei $b = 10$) $\ln 5$ mit einer Dezimalstelle mehr als $\ln 2$ berechnet wird ($\ln 5$ mit ℓ, $\ln 2$ mit $\ell-1$ Stellen), also $|\varepsilon_5| \le 10^{-\ell+1}$, $|\varepsilon_2| \le 10^{-\ell+2} = 10 \cdot 10^{-\ell+1}$. Dann wird

$$|\varepsilon_{10}| \le \frac{|\varepsilon_2| \ln 2 + |\varepsilon_5| \ln 5}{\ln 10} \le \frac{10 \cdot \ln 2 + \ln 5}{\ln 10} \cdot 10^{-\ell+1} \le 3.71 \cdot 10^{-\ell+1} \quad \overset{!}{\le} 10^{-p}.$$

Letzteres bedeutet $\ell \ge p + 1 + \log_{10} 3.71$, also $\ell := p + 2$, d.h. wir benötigen eine Schutzziffer bei $\ln 2$ und zwei bei $\ln 5$. Es ist zu beachten, dass so nach der Rundung auf p Stellen (bei ungünstiger Lage der Einschließung um eine Zahl des p-stelligen Rasters herum) nur ein relativer Gesamtfehler $\le 2 \cdot 10^{p+1}$ garantiert ist. Da wir aber für alle Konstanten einheitlich $\le 10^{p+1}$ garantieren wollen, wird in diesem Fall die Berechnung ggf. sukzessive mit erhöhten Stellenzahlen wiederholt (was von $p = 1$ bis $p = 1000$ genau neunmal vorkommt).

Es wurde auch überlegt, $\ln 2$ und $\ln 1.25$ zu berechnen (wegen $5 = 2^2 \cdot 1.25$ und $10 = 2^3 \cdot 1.25$). Für 1.25 wäre allerdings (wie bei $\arctan 1$) eine rationale Arithmetik innerhalb von Zähler und Nenner notwendig, die den möglichen Geschwindigkeitsgewinn zunichte machen würde.

Als Approximationsvorschrift erhält man also Folgendes für $\ln n$:

$A_0 := 0, \quad B_0 := 1$
$A_1 := n-1, \quad B_1 := 1$
$j := 0$
$f := 0$
repeat
  $j := j+2$
  $f := f + (n-1) \cdot (j-1)$
  $A_j := j \cdot A_{j-1} + f \cdot A_{j-2}, \qquad B_j := j \cdot B_{j-1} + f \cdot B_{j-2},$
  $A_{j+1} := (j+1) \cdot A_j + f \cdot A_{j-1} \qquad B_{j+1} := (j+1) \cdot B_j + f \cdot B_{j-1}$
until $accurate_p([\bigtriangledown_p (A_j/B_j), \bigtriangleup_p (A_{j+1}/B_{j+1})])$

Auch hier bietet es sich nicht an, zwei Schritte zu einem zusammenzufassen, da in den entstehenden Koeffizienten j in der vierten Potenz vorkommt.

Als Schätzwert n_{est} für die Anzahl von Schritten bis zu ersten Division wird $1.403p^{0.9902}$ bzw. $2.468p^{0.9956}$ verwendet. Ab etwa $p = 92000$ bzw. $p = 73000$ sind die Koeffizienten (f)

nicht mehr mit 32-Bit-Integer-Zahlen darstellbar, daher wird vorab eine Exception „internal limitation“ geworfen.

Für $p = 500$ benötigt die vorliegende Implementation von $\ln 10$ auf dem L*i*nux-Rechner 1.54 Sekunden, die Implementation zu [Steins] 19.5 Minuten. Die Routinen heißen

```
static BFInterval BFInterval::ln2(long precision);
static BFInterval BFInterval::ln5(long precision);
static BFInterval BFInterval::ln10(long precision);
```

4 Reelle Standardfunktionen mit beliebiger Genauigkeit

In diesem Kapitel behandeln wir sowohl die Standardfunktionen, die mit Hilfe von Potenzreihen ausgewertet werden, wie auch an jeweils passender Stelle die, die sich aus ihnen mittels algebraischer Operationen ermitteln lassen (beispielsweise der Tangens aus Sinus und Cosinus, bzw. der hyperbolische Cosinus aus der Exponentialfunktion). Einige Funktionen werden beide Darstellungen verwenden, etwa der hyperbolische Sinus, der bei großen Argumenten über die Exponentialfunktion, bei kleinen Argumenten wegen Auslöschungsgefahr bei $e^x - e^{-x}$ aber über die Potenzreihe bestimmt wird. Eine Ausnahme bildet die Quadratwurzel, die wir mit Hilfe des Newton-Verfahrens berechnen.

Wenn übliche Fließkommaarithmetiken mit fester oder jedenfalls begrenzter Stellenzahl zu Grunde gelegt werden, erscheint es eher ungünstig, mit Potenzreihen zu arbeiten. Die Güte der Approximation ist natürlich um den Entwicklungspunkt herum hervorragend, in größerem Abstand aber extrem schlecht. Abgebrochene Taylorreihen schneiden im Vergleich mit anderen polynomialen Approximationen nicht gerade gut ab. Bei gleichem Aufwand haben sie ein um Größenordnungen schlechteres Fehlerverhalten, bzw. benötigen bei gleichem Maximalfehler einen deutlich höheren Grad (vgl. z.B. [Muller], Kapitel 3).

Bei uns liegt aber nicht die Ausgangssituation eines an eine feste Maschinengenauigkeit gekoppelten Maximalfehlers vor; unser Fehler muss beliebig klein gehalten werden können. Interpolationsmethoden fallen für uns dadurch aus, dass die mit ihnen erreichbare Genauigkeit von der Anzahl von Stützstellen und Güte der dort vorberechneten Werte bzw. Ableitungen abhängt; eine sinnvolle Dynamik ist hier nicht vorstellbar. Potenzreihen sind dagegen leicht verlängerbar (typischerweise mit einfach und beliebig genau berechenbaren Koeffizienten), und bei der Taylor-Entwicklung ist die Fehlerabschätzung leicht an beliebige Grade anpassbar, so dass uns faktisch gar keine andere Wahl bleibt.

Im Hinblick auf das vorangegangene Kapitel sei noch erwähnt, dass Kettenbrüche wegen der Langsamkeit der dort besonders intensiv benutzten Division nicht einsetzbar sind; für unsere transzendenten Konstanten boten sie Vorteile wegen des möglichen Wiederaufsetzens der Rechnung zur schrittweisen Erhöhung der Genauigkeit.

An den meisten Stellen entspricht unser Vorgehen im Wesentlichen dem in [Steins], ebenso wurde der Großteil der Bezeichnungen in Anlehnung an diese Arbeit gewählt. Wir werden aber bei allen betrachteten Funktionen die Fehlerabschätzungen verbessern können. Durch geänderte Auswertungsverfahren lassen sich viele besonders kostspielige Operationen vermeiden, wodurch sich die Berechnungsgeschwindigkeit deutlich erhöht. Außerdem werden schnelle Methoden zur Bestimmung einer Mindestanzahl von Gliedern der Potenzreihe vorgeschaltet. Ferner beseitigen wir einige Unschönheiten, die bei der Implementation zu [Steins] teilweise zu falschen Ergebnissen oder Abstürzen führen.

Für diverse allgemeine Beziehungen, z.B. Reihendarstellungen und Additionstheoreme, die in diesem Kapitel Verwendung finden, sei z.B. auf [Atlas] verwiesen.

4.1 Vorbemerkungen

4.1.1 Bezeichnungen und globale Voraussetzungen

Wenn nichts anderes erwähnt wird, ist mit p (für „precision“) die gewünschte Stellenzahl des Ergebnisses eines Einschließungsversuchs bei einer Standardfunktion bezeichnet, mit ℓ die für die Horner- (oder Newton-) Auswertung intern verwendete. Als Generalvoraussetzung wollen wir hier

$$p \geq 3, \qquad \ell \geq 5 \tag{4.1}$$

festlegen. Dadurch kann beispielsweise in unseren späteren relativ komplexen Abschätzungen der Term $b^{-\ell+1}$ (in Vorfaktoren vor den eigentlichen Fehlerintervallen) nach oben durch b^{-4} abgeschätzt werden. Für noch geringere Genauigkeiten wäre man (außerhalb des Wiederberechnungsverfahrens für arithmetische Ausdrücke) mit den meist vorhandenen IEEE-Versionen ohnehin besser bedient. Eine Genauigkeit $p < 3$ wird in der Implementation intern auf $p := 3$ erhöht; die Endrundung findet aber mit der vom Benutzer gewünschten Stellenzahl, also auch ggf. mit $p = 1$ oder $p = 2$ statt. (Man beachte, dass dieses p in [Steins] als k bezeichnet wird, was wir wegen Verwechslungsgefahr mit dem Index k in diversen Reihen hier vermieden haben.)

Eine weitere Generalvoraussetzung sei

$$N \geq 3, \tag{4.2}$$

wobei $N + 1$ die Anzahl der verwendeten Glieder der jeweiligen Potenzreihe ist. (N oder manchmal $2N$ ist die höchste vorkommende *Ordnung*). Man kommt ohnehin nur für sehr geringe Stellenzahlen und Argumente extrem nah am Entwicklungspunkt mit weniger Gliedern aus (und Letztere werden ohnehin gesondert behandelt). Diese Voraussetzung ist daher nützlich, da wir an vielen Stellen für den relativen Fehler das exakte Ergebnis nach oben oder nach unten mit Hilfe einiger der ersten Reihenglieder abschätzen müssen. Für eine vernünftig enge Abschätzung sind manchmal auch vier Glieder notwendig.

Mit P bzw. P_N sei in diesem Abschnitt immer das jeweils verwendete Approximationspolynom (mit $N + 1$ Gliedern) gemeint, $\tilde{P}$ die mit den vom speziellen Fall abhängenden Rundungsfehler behaftete Version davon. Wenn N oder auch das Argument x aus dem Zusammenhang hervorgehen, werden wir diese Angaben gern auslassen.

4.1.2 Argumentreduktionen

Natürlich werden die Potenzreihen nicht global, sondern nur nah um den Entwicklungspunkt herum direkt verwendet. Zum einen ist der Konvergenzbereich der Reihen nur selten der ganze Definitionsbereich der Funktion (nur bei exp, sin, cos, sinh, cosh). Außerdem ist die Konvergenzgeschwindigkeit teilweise recht gering, so dass bei größeren Argumenten auch im Konvergenzbereich unverhältnismäßig viele Reihenglieder zur Berechnung herangezogen werden müssten. Zudem sind nur im Nahbereich um den Entwicklungspunkt herum die beteiligten Rundungs- und Verfahrensfehler vernünftig handhabbar.

Für in diesem Sinne zu große Argumente x ist eine Argumentreduktion notwendig, also die Anwendung algebraischer Identitäten, die die Auswertung auf ein genügend kleines Argument zurückführt. Danach muss das erhaltene Ergebnis ggf. wiederum rücktransformiert

werden. Bei der Exponentialfunktion wird z.B. $\exp(x) = (\exp(\frac{x}{2}))^2$ verwendet, beim Sinus $\sin(x + n \cdot 2\pi) = \sin(x)$ (aufwändig zu implementieren, aber keine Rücktransformation).

$x_{max} > 0$, manchmal als $\bar{x}$ abgekürzt, bezeichne immer den Wert, unterhalb dessen die Potenzreihe Verwendung finden soll, d.h. für $0 < x \leq \bar{x}$. (Ggf. wird in der Theorie zwar ein Intervall $[x_{min}, x_{max}] \ni 0$ verwendet, das intern in der Implementation auf Grund von Symmetrien aber wieder auf ein positives zurückgeführt wird.)

Die Bestimmung eines guten $\bar{x}$ kann sinnvoll nur durch praktische Tests erfolgen, da die Geschwindigkeiten der bei den Argumentreduktionen möglicherweise benötigten anderen Funktionen mit eingehen. Wenn man $\bar{x}$ ändert, sind leider auch die Abschätzungen zum Auswertefehler (siehe Abschnitte 4.1.7 und 4.1.9) anzupassen und die Tabelle aus Abschnitt 4.1.10 neu zu berechnen. Es wurde bei unserer Implementation daher immer nur ein gutes $\bar{x}$ bestimmt, nicht notwendigerweise ein (fast) optimales.

Bei beiden Richtungen der Transformation werden normalerweise Rundungsfehler gemacht werden, deren Behandlung im nächsten Abschnitt vorab kurz beschrieben wird. Die im Einzelfall genaue Vorgehensweise wird im Abschnitt der jeweiligen Funktion behandelt.

Gegebenenfalls werden mehrere Argumentreduktionen hintereinander notwendig. In der Implementation wird dies oft durch Hilfsfunktionen realisiert, die sich ineinander verschachtelt aufrufen. Bei jedem Aufruf wird normalerweise eine Erhöhung der geforderten Stellenzahl um mindestens 2 notwendig sein – allein durch den Unterschied etwas mehr als einer Größenordnung zwischen $2b^{-\ell+1}$ und $b^{-\ell}$ bei der Garantie der Hochgenauigkeit bzw. der Forderung für Hochgenauigkeit.

4.1.3 Auswertung bei gestörtem Argument

In mehreren Zusammenhängen werden wir uns damit befassen müssen, den Gesamtfehler der Auswertung einer Standardfunktion f zu kontrollieren, wenn sowohl die Auswertung inexakt erfolgt, wie auch bereits das ankommende Argument x gestört ist. Zunächst tritt diese Situation ein, wenn das Argument einer Standardfunktion durch eine vorangegangene Argumentreduktion nicht mehr exakt, sondern beispielsweise in Form einer engen Einschließung des exakt transformierten Argument bei der Potenzreihenberechnung ankommt. Ein analoger Fall liegt eine Stufe höher vor, wo im Auswertungsalgorithmus für arithmetische Ausdrücke entschieden werden muss, wie exakt der Teilausdruck berechnet werden muss, der das Argument einer Standardfunktion stellt, und wie exakt die eigentliche Funktionsauswertung erfolgen muss, um die gewünschte Gesamtgenauigkeit zu erreichen.

Das Argument x liege nun also nur in Form einer Einschließung $[x] \ni x$ vor. Wenn die Differenzierbarkeit der Funktion f auf $[x]$ gewährleistet ist, kann der Mittelwertsatz der Differentialrechnung angewendet werden. Für jedes $\tilde{x} \in [x]$ gibt es ein $\xi \in \mathrm{conv}\{x, \tilde{x}\}^\circ$, so dass

$$f(\tilde{x}) - f(x) \;=\; f'(\xi) \cdot (\tilde{x} - x).$$

Statt der exakten Funktion f sei die gestörte Funktion $\tilde{f}$ implementiert, d.h. die Auswertung $y = f(x)$ ist insgesamt als $\tilde{y} = \tilde{f}(\tilde{x})$ verfälscht. Dann hat man (mit $\Delta x := \tilde{x} - x$, $\Delta f := \tilde{f} - f$) für den absoluten Fehler

$$\Delta y \;:=\; \tilde{f}(\tilde{x}) - f(x) \;=\; (\tilde{f}(\tilde{x}) - f(\tilde{x})) + (f(\tilde{x}) - f(x)) \;=\; \Delta f(\tilde{x}) + f'(\xi) \cdot \Delta x.$$

Es sei $\Delta > 0$ die vorgegebene, beliebig kleine absolute Fehlertoleranz, d.h. es werde $|\Delta y| \leq \Delta$ gefordert. Wenn $[y] = f([x])$ nicht die Null enthält, kann auch der relative Fehler (z.B. über eine dezimale Stellenforderung) durch ein $\varepsilon > 0$ beschränkt werden, wobei sich als Forderung $|\Delta y| \leq \varepsilon \cdot |[y]|_{\inf}$ ergibt. Für p-stellige Hochgenauigkeit ergibt sich also (wegen Lemma 1) $\Delta := b^{-p} \cdot |[y]|_{\inf}$.

Angesichts der leichten Umrechnung betrachten wir hier zunächst nur absolute Fehler. Als Forderung haben wir erhalten (vgl. [Steins] Gleichung (1.11)):

$$|\Delta f(\tilde{x})| + |f'(\xi)| \cdot |\Delta x| \stackrel{!}{\leq} \Delta. \tag{4.3}$$

Im jeweiligen Fall ist $|f'(\xi)| \in |[f'([x])]|$ geeignet nach oben abzuschätzen. Bei den Argumentreduktionen wird ggf. $[x]$ sogar durch das gesamte (leicht nach oben vergrößerte) Intervall ersetzt, in dem die Potenzreihenentwicklung verwendet wird.

Man kann nun beispielsweise so ansetzen, dass die Fehlertoleranz auf beide Summanden gleich verteilt wird (vgl. [Steins], (1.14)), also:

$$|\Delta f(\tilde{x})| \leq \tfrac{\Delta}{2}, \quad |f'(\xi)| \cdot |\Delta x| \leq \tfrac{\Delta}{2}, \tag{4.4}$$

woraus sich jeweils eine Bedingung für die zu verwendenden Stellenzahlen bei Funktion und Argument ergibt. Natürlich sind andere Gewichtungen denkbar, etwa $\frac{1}{4} + \frac{3}{4}$, um z.B. beim Wiederberechnungsverfahren das Argument, das durch einen ganzen Teilbaum dargestellt wird, weniger genau berechnen zu müssen als die eine Funktionsauswertung.

Bei hoch genauer Berechnung des Arguments mit ℓ Stellen bzw. m Stellen bei der Funktionsauswertung ist der jeweilige relative Fehler betragsmäßig durch $2b^{-\ell+1}$ bzw. $2b^{-m+1}$ beschränkt. So ergibt sich auch die Möglichkeit, die gleiche Stellenzahl $m = \ell$ oder eine entsprechende Gewichtung wie $m = \ell + 2$ zu fordern. Das führt auf nur eine zu erfüllende Gleichung, die je nach Funktion einfacher zu behandeln sein kann als die beiden von oben.

4.1.4 Grundlegende Beziehungen

Um die in den Abschätzungen des Auswertefehlers vorkommenden Potenzen $(1 + E_\ell)^n$ besser handhaben zu können, werden sie durch das folgende Lemma linearisiert. Es erscheint in [Steins] als Lemma 2.1.1, ohne Beweis, und ist abgeleitet von [Braune], Lemma 2, dort allerdings in der Fehlerglied-Notation. Der hier geführte Beweis verläuft analog zu dem von [Braune], verwendet aber unsere Intervallschreibweise.

Lemma 2 *Es seien $\ell, n \in \mathbb{N}^+$, $\ell \geq 2$. Falls*

$$n \leq \sqrt{2b^{\ell-1}} - 1,$$

dann folgt

$$(1 + E_\ell)^n \subseteq 1 + (n+1)\, E_\ell,$$

und beide Intervalle enthalten nicht die Null (außer beim rechten Intervall für $b = \ell = 2$ als unteren Rand).

Beweis: Es ist

$$\begin{aligned}(1+E_\ell)^n &= [\,(1-b^{-\ell+1})^n\,,\,(1+b^{-\ell+1})^n\,] &&> 0 \qquad (\text{da } \ell \geq 2),\\ 1+(n+1)E_\ell &= [\,1-(n+1)b^{-\ell+1}\,,\,1+(n+1)b^{-\ell+1}\,] &&> 0.\end{aligned}$$

Letzteres „>“ gilt genau dann, wenn $n < b^{\ell-1}-1$, aber nach Voraussetzung ist ja sogar $n \leq \sqrt{2b^{\ell-1}}-1$, und es ist $\sqrt{2b^{\ell-1}} \leq b^{\ell-1}$ mit Gleichheit genau für $b=\ell=2$.

Für die behauptete Inklusion sind die entsprechenden Ungleichungen in Ober- und Untergrenze zu zeigen:

a) Für die Untergrenze ist zu zeigen

$$1-(n+1)\,b^{-\ell+1} \;\leq\; (1-b^{-\ell+1})^n,$$

bzw. mit Hilfe der binomischen Reihe

$$1-(n+1)\,b^{-\ell+1} \;\leq\; 1-n\cdot b^{-\ell+1}+\sum_{k=2}^{n}\binom{n}{k}(-1)^k(b^{-\ell+1})^k,$$

d.h.

$$-b^{-\ell+1} \;\leq\; \sum_{k=2}^{n}\binom{n}{k}(-1)^k(b^{-\ell+1})^k.$$

Also sind wir fertig, wenn wir sogar zeigen können, dass die rechte Seite ≥ 0 ist. Das ist aber daran abzulesen, dass das führende Summenglied positiv ist, die Glieder im Vorzeichen alternieren und betragsmäßig streng monoton fallen:

$$\frac{\binom{n}{k+1}(b^{-\ell+1})^{k+1}}{\binom{n}{k}(b^{-\ell+1})^k} = \frac{n-k}{k+1}\,b^{-\ell+1} < n\,b^{-\ell+1} \overset{\text{Vorauss.}}{\leq} (\sqrt{2\,b^{\ell-1}}-1)\,b^{-\ell+1} < b^{\ell-1}\,b^{-\ell+1} = 1.$$

b) Für die Obergrenze ist zu zeigen

$$(1+b^{-\ell+1})^n \;\leq\; 1+(n+1)\,b^{-\ell+1},$$

äquivalent zu

$$\ln(1+(n+1)\,b^{-\ell+1})-n\cdot\ln(1+b^{-\ell+1}) \;\geq\; 0.$$

Wir verwenden hier die aus der Taylorentwicklung des Logarithmus um 1 stammende Beziehung

$$x-\frac{x^2}{2} \;\leq\; \ln(1+x) \;\leq\; x$$

(für $x \geq 0$) und erhalten

$$\begin{aligned}\ln(1+(n+1)\,b^{-\ell+1})-n\cdot\ln(1+b^{-\ell+1}) &\geq (n+1)b^{-\ell+1}-\tfrac{1}{2}(n+1)^2(b^{-\ell+1})^2-n\,b^{-\ell+1}\\ &= b^{-\ell+1}\cdot\Big(1-\frac{(n+1)^2}{2}\,b^{-\ell+1}\Big) \;\geq\; 0\\ \iff \qquad \frac{(n+1)^2}{2}\,b^{-\ell+1} &\leq 1\\ \iff \qquad n &\leq \sqrt{2b^{\ell-1}}-1,\end{aligned}$$

und Letzteres entspricht gerade unserer Voraussetzung. □

Bereits für das nach unserer Generalvoraussetzung minimale $\ell = 5$ bedeutet die Voraussetzung des Lemmas bei einer Basis $b = 10$ gerade $n \leq 140$, für $\ell = 10$ bereits $n \leq 44720$. Dennoch muss sie bei Verwendung des Lemmas natürlich gesichert sein.

Das Lemma wird in *allen* Abschätzungen für den Auswertefehler bei den Potenzreihen verwendet. Wir werden dort in den vorkommenden Ausdrücken (insbesondere Summen) darauf achten, nicht unnötigerweise auch $(1 + E_\ell)$ durch $(1 + 2E_\ell)$ abzuschätzen! Dadurch werden einige Ausdrücke durch Abspalten von Summanden etwas umständlich aussehen.

n ist typischerweise durch den maximal erlaubten Approximationsfehler, also durch die minimal erlaubte Anzahl von Gliedern vorgegeben. In der Implementation muss daher *immer* überprüft werden, ob ℓ entsprechend groß genug ist, anderenfalls muss es geeignet erhöht werden. Wir werden das bei der ersten betrachteten Funktion (der Exponentialfunktion) noch einmal genau darstellen.

4.1.5 Gesamtstrategie bei der Horner-Auswertung

Unter den Parametern der Auswertung wollen wir die folgenden beiden verstehen:

- ℓ – die Stellenzahl, mit der die arithmetischen Operationen in der Durchführung des Horner-Verfahrens arbeiten,
- N – die Anzahl der ausgewerteten Glieder der Potenzreihe *minus 1* (typischerweise der Maximalgrad oder der halbe Maximalgrad).

Die an der Auswertung beteiligten Fehler sind (wobei für die relativen Fehler jeweils $P(x) \neq 0$ bzw. $f(x) \neq 0$ vorausgesetzt wird):

$$\Delta_P = \Delta_P^N(x) = \tilde{P}_N(x) - P_N(x), \qquad \varepsilon_P = \varepsilon_P^N(x) = \frac{\Delta_P}{P(x)}, \tag{4.5}$$

der absolute bzw. relative Fehler, der bei der Auswertung des Approximationspolynoms $P = P_N$ gemacht wird,

$$\Delta_{app} = \Delta_{app}^N(x) = P_N(x) - f(x), \qquad \varepsilon_{app} = \varepsilon_{app}^N(x) = \frac{\Delta_{app}}{f(x)}, \tag{4.6}$$

der absolute bzw. relative Approximationsfehler, d.h. der Fehler, der durch Abbruch der Auswertung der Potenzreihe nach $N + 1$ Gliedern entsteht,

$$\Delta_{ges} = \Delta_{ges}^N(x) = \tilde{P}_N(x) - f(x), \qquad \varepsilon_{ges} = \varepsilon_{ges}^N(x) = \frac{\Delta_{ges}}{f(x)}, \tag{4.7}$$

der absolute bzw. relative Gesamtfehler.

Es gilt also

$$\begin{aligned} \tilde{P}(x) &= (1 + \varepsilon_P) \cdot P(x), \quad P(x) = (1 + \varepsilon_{app}) \cdot f(x), \quad \tilde{P}(x) = (1 + \varepsilon_{ges}) \cdot f(x), \\ \Delta_{ges} &= \tilde{P}(x) - P(x) + P(x) - f(x) = \Delta_P + \Delta_{app}, \end{aligned}$$

$$\begin{aligned}
\varepsilon_{ges} &= \frac{\tilde{P}(x) - f(x)}{f(x)} = \frac{(1+\varepsilon_P)P(x) - f(x)}{f(x)} = \frac{(1+\varepsilon_P)(1+\varepsilon_{app})f(x) - f(x)}{f(x)} \\
&= (1+\varepsilon_P)(1+\varepsilon_{app}) - 1 \\
&= \varepsilon_{app} + \varepsilon_P(1+\varepsilon_{app}) \\
&= \varepsilon_P + \varepsilon_{app}(1+\varepsilon_P). \qquad (4.8)
\end{aligned}$$

Je nach Zusammenhang lässt sich unter den letzten drei Zeilen besser die symmetrische oder eine der unsymmetrischen Beziehungen verwenden.

Der Approximationsfehler ε_{app} wird im jeweiligen Fall z.B. durch Betrachtung des Restglieds der Taylor-Entwicklung behandelt, aus der die Potenzreihe hervorgeht. Unser genaues Vorgehen wird in Abschnitt 4.1.10 dargestellt. Man beachte, dass dabei die Argumentbeschränkung $x \in [x_{min}, x_{max}]$ eingeht.

Es wird sich herausstellen, dass man Abschätzungen für Δ_P finden kann, die von N unabhängig sind – sofern zumindest die Generalvoraussetzung $N \geq 3$ und wiederum $x \in [x_{min}, x_{max}]$ erfüllt sind. Das kann beispielsweise dadurch geschehen, dass die in den Fehlerausdrücken entstehenden Summen (in denen die Koeffizienten und die Potenzen des Arguments des Polynoms erscheinen) nach oben durch vollständige Potenzreihen abgeschätzt werden, die zu anderen Standardfunktionen gehören, bzw. die geometrischen Reihen oder ähnlichen leicht handhabbaren Ausdrücken entsprechen.

Man beachte hier, dass diese Abschätzungen auch vom letztlich gewählten Rundungsmodus unabhängig sind. Es geht ein, dass die jeweiligen relativen Fehler in E_ℓ liegen, so dass Rundungen nach unten oder oben (für Unter- bzw. Obergrenzen einer Einschließung) oder zur nächsten Zahl abgedeckt sind.

Unsere Strategie (vgl. [Steins], S. 39) ergibt sich daher wie folgt:

1. Um ein auf p Stellen hoch genaues Gesamtergebnis zu erhalten, ist unsere Forderung
$$|\varepsilon_{ges}| \stackrel{!}{\leq} b^{-p},$$
bzw. schärfer mit (4.8)
$$|\varepsilon_{ges}| = |\varepsilon_P + \varepsilon_{app} \cdot (1+\varepsilon_P)| \leq |\varepsilon_P| + |\varepsilon_{app}| \cdot (1+|\varepsilon_P|) \stackrel{!}{\leq} b^{-p}, \qquad (4.9)$$

2. Für die Verwendung ℓ-stelliger Arithmetik bei der Polynom-Auswertung ergeben die Betrachtungen der nächsten Abschnitte jeweils eine von N unabhängige Abschätzung
$$|\varepsilon_P| \leq c \cdot b^{-\ell+1} \qquad (4.10)$$
mit einem $c \in \mathbb{R}^+$, das auch nicht von ℓ abhängig ist.

3. Damit nun (4.9) bei einem Approximationsfehler $\varepsilon_{app} \neq 0$ mit dem ε_P nach (4.10) überhaupt erfüllt werden kann, muss gelten
$$\begin{aligned}
& c \cdot b^{-\ell+1} < b^{-p} \\
\Leftrightarrow \quad & b^{-p+\ell-1} > c \\
\Leftrightarrow \quad & -p+\ell-1 > \log_b c \\
\Leftrightarrow \quad & \ell > p+1+\log_b c \qquad (4.11)
\end{aligned}$$

Aus dieser Beziehung erhält man die Mindestanzahl zusätzlich notwendiger Schutzziffern $\ell - p$. Mindestens muss außerdem unsere Generalvoraussetzung $\ell \geq 5$ gelten. Man beachte, dass die Anzahl der Schutzziffern nicht von x oder p abhängt, also vorab theoretisch und nicht im Programm berechnet wird. (Sehr wohl geht allerdings x_{max} ein.)

4. Mit diesem ℓ und der sich aus (4.10) ergebenden Schranke für $|\varepsilon_P|$ erhält man nun aus (4.9) eine hinreichende Bedingung für den jetzt noch erlaubten Approximationsfehler:

$$\begin{aligned} |\varepsilon_{ges}| &= |\varepsilon_P + \varepsilon_{app}(1+\varepsilon_P)| \\ &\leq |\varepsilon_P| + |\varepsilon_{app}| \cdot (1 + |\varepsilon_P|) \\ &< c \cdot b^{-\ell+1} + |\varepsilon_{app}| \cdot (1 + c \cdot b^{-\ell+1}) \overset{!}{\leq} b^{-p}, \end{aligned}$$

   bzw. nach Umformung die Forderung

$$|\varepsilon_{app}| \leq \frac{b^{-p} - c \cdot b^{-\ell+1}}{1 + c \cdot b^{-\ell+1}} = b^{-p} \cdot \frac{1 - c \cdot b^{-(\ell-p)+1}}{1 + c \cdot b^{-\ell+1}}. \tag{4.12}$$

5. Der erste Schritt, der nun *zur Laufzeit* durchgeführt wird, ist der folgende. Üblicherweise über das Restglied der Taylor-Entwicklung wird ein möglichst kleines N gefunden, so dass der Approximationsfehler unterhalb der in (4.12) gefundenen Schranke bleibt. Dazu wird das tatsächliche Argument x und nicht beispielsweise die Oberschranke x_{max} verwendet (siehe Abschnitt 4.1.10).

6. Danach muss noch geprüft werden, ob die Voraussetzungen von Lemma 2 erfüllt sind, das ja in allen Abschätzungen für ε_P verwendet wird. Hier geht das gerade gefundene N ein (die höchste vorkommende Potenz ist beispielsweise N oder $2N$). In Bemerkung a) wird beschrieben, wie dieser Schritt in der Implementation durchgeführt wird.

7. Mit den gefundenen Parametern ℓ und N kann nun das Horner-Verfahren durchgeführt werden. Es wird darin aber nie intervallmäßig gearbeitet. Vielmehr wird – fast doppelt so schnell – nur eine Ober- oder nur eine Untergrenze berechnet, mit Hilfe der jeweils passenden gerichteten Rundungen. Das Vorgehen wird in Bemerkung b) dargestellt.

8. Nach der Berechnung der Schranken wird das Intervall in das endgültig gewünschte p-stellige Raster gerundet.

Bemerkungen:

a) In der Implementation wird die Überprüfung von Lemma 2 wie folgt durchgeführt.

   Es sei n die höchste Potenz, in der der Term $(1 + E_\ell)$ in den Abschätzungen aus Punkt 2 vorkommt. Dann muss gelten

$$n \leq \sqrt{2b^{\ell-1}} - 1.$$

   Sollte das nicht so sein (was sehr selten der Fall ist), muss ℓ entsprechend erhöht werden; nach Umformung ergibt sich

$$\ell \geq 1 + \log_b \frac{(n+1)^2}{2}.$$

In der Implementation (zur Basis $b = 10$) wird $\log_{10}$ durch order ersetzt ($\operatorname{order} x \leq \log_{10}(x)$), und man erhält die etwas schärfere Forderung

$$\ell \geq 2 + \operatorname{order}\left\lfloor \frac{(n+1)^2+1}{2} \right\rfloor$$

In der letzteren Form wird der Test in der Implementation durchgeführt. Für $n \leq 46339$ kann der Test mit 32-Bit-Integerzahlen durchgeführt werden. Selbst für $n = 46339$ erhält man lediglich $\ell \geq 11$, so dass ℓ überhaupt nur selten geändert werden wird.

Für Integerzahlen wird anstelle von order eine Funktion `ndigs` benutzt, die eben die Anzahl der Dezimalstellen einer Integerzahl liefert. Sie ist nicht mit Divisionen, sondern schneller mit einer binären Verschachtelung von Vergleichen implementiert.

b) Wir beschreiben kurz, wie die Bestimmung der jeweils nicht mit dem Horner-Verfahren berechnete Grenze vonstatten geht. Nach der Argumentreduktion sind Argumente und Funktionswerte üblicherweise positiv, was wir hier der Einfachheit halber voraussetzen.

Wenn alle Reihenglieder positiv sind (bei exp, sinh, ggf. tan), berechnet man mit gerichteten Rundungen immer eine *Untergrenze*. Bei alternierendem Vorzeichen und positivem ersten Glied (sin, cos, arsinh, arcosh, ggf. tanh) erhält man bei geradem N eine *Obergrenze*, bei ungeradem N eine *Untergrenze*, und rundet entsprechend.

Die jeweils andere Grenze erhält man aus unseren Abschätzungen für den relativen Fehler. Aus N und der Restglied-Abschätzung aus Punkt 5 hat man automatisch eine obere Schranke $\bar{\varepsilon}_{app}$ für $|\varepsilon_{app}|$ erhalten. Aus dem möglicherweise inzwischen korrigierten ℓ und (4.10) erhält man eine jetzt gültige obere Schranke $\bar{\varepsilon}_P$ für $|\varepsilon_P|$. Aus beidem resultiert mit (4.8) eine obere Schranke $\bar{\varepsilon}_{ges} := \bar{\varepsilon}_{app} + \bar{\varepsilon}_P(1 + \bar{\varepsilon}_{app})$.

Hat man nun eine Unterschranke $\tilde{P} = \underline{f}$ berechnet, so verwendet man

$$f(x) \;=\; \frac{\tilde{P}(x)}{1+\varepsilon_{ges}} \;\leq\; \frac{\tilde{P}(x)}{1-\bar{\varepsilon}_{ges}} \;=:\; \bar{f} \tag{4.13}$$

zur Bestimmung einer garantierten Oberschranke $\bar{f}$.

Die Implementation zu [Steins] verwendet hier $\tilde{P}(1+\bar{\varepsilon}_{ges})$. Es ist $\frac{1}{1-\varepsilon} = (1+\varepsilon) + \frac{\varepsilon^2}{1-\varepsilon}$. Bei $b = 10$ ist wegen $\bar{\varepsilon}_{ges} \leq 10^{-p}$ der letzte Summand kleiner als $1.0011 \cdot 10^{-2p}$ und fällt dort durch vorangegangene Überschätzungen und Rundungen nie ins Gewicht.

Wir arbeiten auf der sicheren Seite mit $\tilde{P}(1 + \bar{\varepsilon}_{ges} + 10^{-\ell})$, d.h. wir benutzen den Nachfolger von $1 + \bar{\varepsilon}_{ges}$ in $(\ell+1)$-stelliger Arithmetik. Hinreichend dafür, dass $10^{-\ell}$ hier genügend groß ist, ist die Bedingung $\ell - p \leq p - 1$, d.h. die Anzahl der Schutzziffern darf höchstens $p-1$ betragen, und es ist $p \geq 3$. Wir werden fast immer genau zwei Schutzziffern benötigen. Wenn man $1.01 \cdot 10^{-\ell}$ benutzt, ist $\ell - p \leq p$ hinreichend, was im Fall von drei Schutzziffern benutzt wird.

Analog verwendet man im Fall einer im Horner-Verfahren bestimmten Oberschranke $\tilde{P} = \bar{f}$

$$f \;=\; \frac{\tilde{P}}{1+\varepsilon_{ges}} \;\geq\; \frac{\tilde{P}}{1+\bar{\varepsilon}_{ges}} \;=:\; \underline{f} \tag{4.14}$$

zur Bestimmung einer garantierten Unterschranke $\underline{f}$. Da $\frac{1}{1+\varepsilon} = (1-\varepsilon) + \frac{\varepsilon^2}{1+\varepsilon}$, können wir in diesem Fall tatsächlich direkt mit $\tilde{P}(1-\bar{\varepsilon}_{ges})$ arbeiten.

c) Die Bestimmung der Konstanten c im Schritt 2 kann sich recht trickreich gestalten. Es hat sich als günstig erwiesen, die Herleitungen unter Zuhilfenahme unserer Langzahlarithmetik auf dem Rechner überschlagsmäßig zu bestätigen. Dazu werden die beim Horner-Verfahren entstehenden absoluten Fehler garantiert in Intervalle eingeschlossen, wobei man natürlich mit höherer Stellenzahl als ℓ rechnen muss.

Selbstverständlich ist dies kein Beweis, da jeweils ein festes ℓ und ein festes N gewählt werden müssen. Auf diese Weise können aber schnell Rechenfehler in den aufwändigen Umformungen gefunden oder übergroße Überschätzungen aufgespürt werden. Immer wenn die Auswertung auf dem Rechner eine deutlich kleinere Konstante lieferte als die sich aus den Abschätzungen ergebende, wurde an diesen Abschätzungen noch gearbeitet. So wurde zum Beispiel beim Cosinus die Konstante $c \approx 4$ als wesentlich zu groß erkannt und konnte mit schärferen Abschätzungen auf $c \approx 1.5$ gedrückt werden.

4.1.6 Durchführung des Horner-Verfahrens

Das Approximationspolynom sei wie folgt gegeben:

$$P(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \ldots + a_N x^N. \tag{4.15}$$

Das Horner-Verfahren beruht auf folgender Klammerung:

$$P(x) = a_0 + x \cdot (a_1 + x \cdot (a_2 + x \cdot (a_3 + \ldots + x \cdot (a_N)\,)\,)\,\ldots\,). \tag{4.16}$$

Es benötigt N Multiplikationen, während ein kanonisches Aufmultiplizieren $2N-1$ Multiplikationen bräuchte und außerdem dadurch numerisch unsinnig wäre, dass es (in unserem Anwendungsfall Potenzreihen für $|x| < 1$) bei den betragsmäßig größeren Termen beginnen würde.

Die Klammerung ergibt, als Auswertungsverfahren (mit Zwischenergebnissen r_k) dargestellt:

$$\begin{aligned} &a) \quad r_N \;&&:= a_N, \\ &b) \quad r_k \;&&:= a_k + x \cdot r_{k+1} \qquad \text{für } k = N-1, \ldots, 0. \end{aligned}$$

Für die Zwischenergebnisse gilt dann

$$r_k = \sum_{m=k}^{N} a_k \, x^{m-k}$$

und insbesondere $P(x) = r_0$.

In einer Implementation des Verfahrens mit Fließkomma-Arithmetik finden an folgenden Stellen Rundungsfehler Eingang:

1. Die Koeffizienten a_k brauchen nicht exakt darstellbar zu sein. In unseren Potenzreihen haben sie die typische Form $\frac{1}{n}$, $n \in \mathbb{N}$, und sind dann bei dezimaler Arithmetik nur exakt darstellbar, wenn in n nur Primfaktoren 2 und 5 vorkommen.

2. Das Argument x kann durch eine vorangegangene Transformation gestört sein. Wenn beispielsweise nur gerade Potenzen von x vorkommen, handelt es sich bei P um ein Polynom im Argument x^2. Aus Geschwindigkeitsgründen wäre es aber nicht sinnvoll, ein exaktes x^2 mit der doppelten Stellenzahl von x durch die ganze Berechnung zu schleppen.

3. Multiplikation und Addition brauchen nicht exakt durchführbar zu sein. (Bei uns sind sie es theoretisch, was aber zu explodierenden Stellenzahlen führen würde und angesichts gestörter Koeffizienten ohnehin unsinnig wäre.)

Mit unserer beliebig genauen Fließkomma-Arithmetik haben wir durch exakte Berechnung von *Zwischenergebnissen* die Möglichkeit, einige Rundungsfehler zu vermeiden, die eine Arithmetik mit fester Stellenzahl nicht umgehen kann. Wie bei der Implementation zu [Steins] wird auch bei uns im Schritt b) die Multiplikation exakt durchgeführt, nach der Addition aber gerundet. Da die Fließkomma-Multiplikation in der aktuellen Fassung intern *immer exakt* arbeitet und die gerundeten Versionen nur nachträglich eine Rundung dieses exakten internen Ergebnisses durchführen, spart man auf diese Weise sogar echt eine Operation ein.

Andere in [Steins] vorgeschlagene Variationen bringen vergleichsweise wenig Verbesserung in der Fehlerfortpflanzung (insbesondere nicht weniger Schutzziffern, selten indirekt ein kleineres N), bedeuten aber eine deutliche Verlangsamung eines Schritts. Die Berechnung der Koeffizienten a_k mit doppelter Genauigkeit beispielsweise bringt eine doppelt so lange Division mit sich, und diese Operation ist nun gerade die langsamste der Grundoperationen.

In [Steins], Abschnitt 2.1.2, findet sich eine ganze Sammlung von Lemmas zur Abschätzung des Auswertefehlers unter Berücksichtigung obiger Rundungen. Sie sind angelehnt an die Betrachtungen in [Braune], Abschnitt 2.4.4, bzw. [Krämer], Abschnitt 1.3, unter Umsetzung der Fehlerterm-Schreibweise in die Intervall-Notation.

Wir beschränken uns hier auf einige Varianten, die auf unsere Gegebenheiten zugeschnitten sind. Im folgenden Abschnitt behandeln wir das Horner-Verfahren, so wie es oben dargestellt wurde (verwendet für arctan und artanh). In Abschnitt 4.1.8 betrachten wir eine gegenüber [Steins] modifizierte Version mit geänderter Klammerung. Sie ist nur bei fakultätsähnlichen Koeffizienten verwendbar, wird am häufigsten eingesetzt (bei exp, sin, cos, sinh, arcosh und weiter abgewandelt arsinh) und bringt dann eine erhebliche Beschleunigung.

4.1.7 Auswertefehler beim normalen Horner-Verfahren

Wir betrachten zunächst den recht allgemeinen Fall, bei dem alle Koeffizienten (bis auf a_0) und das Argument inexakt sind.

Lemma 3 *Das Approximationspolynom* (4.15) *sei wie in* (4.16) *geklammert und werde entsprechend ausgewertet. Das Argument sei gestört als* $\tilde{x} = x(1+\varepsilon_x)$*. Alle Koeffizienten seien gestört als* $\tilde{a}_k \in a_k \cdot (1+E_\ell)$*, nur* a_0 *sei im verwendeten Fließkommasystem exakt darstellbar. Die Multiplikationen mögen exakt durchgeführt werden, die Additionen sollen ins* ℓ*-stellige Raster gerundet werden.*

Dann gilt für den absoluten Auswertefehler $\Delta_P = \tilde{P} - P$:

$$\begin{aligned}\Delta_P \;\in\; & E_\ell \Big[|r_0| + \sum_{k=1}^{N-1} |a_k x^k| \,(1+\varepsilon_x)^k \,(1+E_\ell)^{k+1} + \sum_{k=1}^{N} |r_k x^k| \,(1+\varepsilon_x)^k \,(1+E_\ell)^k \Big] \\ & + \varepsilon_x \sum_{k=1}^{N} r_k x^k \,(1+\varepsilon_x)^{k-1} \,(1+E_\ell)^k.\end{aligned}$$

Beweis: Die verfälschten Zwischenergebnisse seien mit $\tilde{r}_k$ bezeichnet, die absoluten Abweichungen mit $\Delta_k := \tilde{r}_k - r_k$. Dann ergibt sich:

$$\begin{aligned}
\text{a)}\quad \tilde{r}_N &= \tilde{a}_N \;\in\; a_N \cdot (1+E_\ell) \;=\; r_N \cdot (1+E_\ell) \;=\; r_N + r_N E_\ell, \\
\Delta_N &\in r_N E_\ell. \\
\text{b)}\quad \tilde{r}_k &\in (\, a_k(1+E_\ell) + x(1+\varepsilon_x)(r_{k+1} + \Delta_{k+1}) \,)\,(1+E_\ell) \\
&= (a_k + a_k E_\ell + x r_{k+1} + x\varepsilon_x r_{k+1} + x(1+\varepsilon_x)\Delta_{k+1})\,(1+E_\ell) \\
&= (r_k + a_k E_\ell + x\varepsilon_x r_{k+1} + x(1+\varepsilon_x)\Delta_{k+1})\,(1+E_\ell) \\
&\subseteq r_k + r_k E_\ell + (a_k E_\ell + x\varepsilon_x r_{k+1} + x(1+\varepsilon_x)\Delta_{k+1})\,(1+E_\ell), \\
\Delta_k &\in r_k E_\ell + (a_k E_\ell + \varepsilon_x r_{k+1} x)(1+E_\ell) + x(1+\varepsilon_x)(1+E_\ell)\Delta_{k+1}. \\
\text{b}_0)\quad \tilde{r}_0 &\in (\, a_0 + x(1+\varepsilon_x)(r_1 + \Delta_1) \,)\,(1+E_\ell) \\
&= (\, r_0 + x\varepsilon_x r_1 + x(1+\varepsilon_x)\Delta_1 \,)\,(1+E_\ell) \\
&\subseteq r_0 + r_0 E_\ell + (x\varepsilon_x r_1 + x(1+\varepsilon_x)\Delta_1)\,(1+E_\ell), \\
\Delta_0 &\in r_0 E_\ell + r_1 x\varepsilon_x (1+E_\ell) + x(1+\varepsilon_x)(1+E_\ell)\Delta_1.
\end{aligned}$$

(Dabei gingen (I_1) und (I_2) ein; in den weiteren Rechnungen werden auch (E_1) bis (E_4) verwendet.)

Wir beweisen zunächst per Induktion über $N \geq 1$ Folgendes:

$$\Delta_1 \in r_1 E_\ell + \sum_{k=1}^{N-1} x^{k-1}(a_k E_\ell + x\varepsilon_x r_{k+1})(1+\varepsilon_x)^{k-1}(1+E_\ell)^k + \sum_{k=1}^{N-1} x^k r_{k+1} E_\ell (1+\varepsilon_x)^k (1+E_\ell)^k.$$

Für den Induktionsanfang $N = 1$ mit $\Delta_1 \in r_1 E_\ell$ ist nichts zu zeigen (leere Summen).

Induktionsschritt $N-1 \mapsto N$: Aus dem Verfahren mit Höchstgrad $N-1$ ergibt sich für das mit Höchstgrad N durch Indexverschiebung die Induktionsvoraussetzung

$$\Delta_2 \in r_2 E_\ell + \sum_{k=1}^{N-2} x^{k-1}(a_{k+1} E_\ell + x\varepsilon_x r_{k+2})(1+\varepsilon_x)^{k-1}(1+E_\ell)^k + \sum_{k=1}^{N-2} x^k r_{k+2} E_\ell (1+\varepsilon_x)^k (1+E_\ell)^k.$$

Dies setzen wir in die vorne hergeleitete Beziehung für Δ_k mit $k = 1$ ein:

$$\begin{aligned}
\Delta_1 \;\in\; & r_1 E_\ell + (a_1 E_\ell + x\varepsilon_x r_2)(1+E_\ell) + x(1+\varepsilon_x)(1+E_\ell)\Delta_2 \\
\subseteq\; & r_1 E_\ell + (a_1 E_\ell + x\varepsilon_x r_2)(1+E_\ell) + x r_2 (1+\varepsilon_x)(1+E_\ell) E_\ell \\
& + \sum_{k=1}^{N-2} x^k (a_{k+1} E_\ell + x\varepsilon_x r_{k+2})(1+\varepsilon_x)^k (1+E_\ell)^{k+1} \\
& + \sum_{k=1}^{N-2} x^{k+1} r_{k+2} E_\ell (1+\varepsilon_x)^{k+1} (1+E_\ell)^{k+1}
\end{aligned}$$

$$
\begin{aligned}
= \quad & r_1E_\ell + (a_1E_\ell + x\varepsilon_x r_2)(1+E_\ell) + xr_2(1+\varepsilon_x)(1+E_\ell)E_\ell \\
& + \sum_{k=2}^{N-1} x^{k-1}(a_kE_\ell + x\varepsilon_x r_{k+1})(1+\varepsilon_x)^{k-1}(1+E_\ell)^k \\
& + \sum_{k=2}^{N-1} x^k r_{k+1}E_\ell(1+\varepsilon_x)^k(1+E_\ell)^k,
\end{aligned}
$$

woraus sich durch Einbinden der allein stehenden Summanden in die passende Summe (für $k=1$) die Behauptung der Induktion ergibt.

Durch Einsetzen in die in Schritt b$_0$) gefundene Beziehung für $\Delta_P = \Delta_0$ erhält man:

$$
\begin{aligned}
\Delta_0 \quad & \in r_0E_\ell + r_1x\varepsilon_x(1+E_\ell) + r_1xE_\ell(1+\varepsilon_x)(1+E_\ell) \\
& \quad + \sum_{k=1}^{N-1} x^k(a_kE_\ell + x\varepsilon_x r_{k+1})(1+\varepsilon_x)^k(1+E_\ell)^{k+1} \\
& \quad + \sum_{k=1}^{N-1} x^{k+1}r_{k+1}E_\ell(1+\varepsilon_x)^{k+1}(1+E_\ell)^{k+1} \\
& \subseteq r_0E_\ell + r_1x\varepsilon_x(1+E_\ell) + r_1xE_\ell(1+\varepsilon_x)(1+E_\ell) \\
& \quad + E_\ell \sum_{k=1}^{N-1} |a_kx^k|(1+\varepsilon_x)^k(1+E_\ell)^{k+1} + E_\ell \sum_{k=1}^{N-1} |r_{k+1}x^{k+1}|(1+\varepsilon_x)^{k+1}(1+E_\ell)^{k+1} \\
& \quad + \varepsilon_x \sum_{k=1}^{N-1} x^{k+1}r_{k+1}(1+\varepsilon_x)^k(1+E_\ell)^{k+1} \\
& = r_0E_\ell + E_\ell \sum_{k=1}^{N-1} |a_kx^k|(1+\varepsilon_x)^k(1+E_\ell)^{k+1} + E_\ell \sum_{k=0}^{N-1} |r_{k+1}x^{k+1}|(1+\varepsilon_x)^{k+1}(1+E_\ell)^{k+1} \\
& \quad + \varepsilon_x \sum_{k=0}^{N-1} x^{k+1}r_{k+1}(1+\varepsilon_x)^k(1+E_\ell)^{k+1} \\
& = r_0E_\ell + E_\ell \sum_{k=1}^{N-1} |a_kx^k|(1+\varepsilon_x)^k(1+E_\ell)^{k+1} + E_\ell \sum_{k=1}^{N} |r_kx^k|(1+\varepsilon_x)^k(1+E_\ell)^k \\
& \quad + \varepsilon_x \sum_{k=1}^{N} x^k r_k(1+\varepsilon_x)^{k-1}(1+E_\ell)^k.
\end{aligned}
$$

Ausklammern von E_ℓ liefert die Behauptung. □

Dieses Lemma entspricht [Steins], Lemma 2.1.5 (mit einem durch günstigere Klammerung kleineren Faktor). Außerdem erscheint von der Formulierung her ein führender Term r_0 (statt a_0) sinnvoller, da für den relativen Fehler ohnehin noch durch r_0 dividiert werden muss.

Wir setzen an dieser Stelle (noch) nicht die Summenbeziehung für die r_k ein, die in [Steins] zur $P/P'/P''$-Formulierung bei den Lemmas 2.1.7 bis 2.1.11 führt, da es sich in einigen Fällen als günstig erweist, diese Terme speziell abzuschätzen.

Das normale Horner-Verfahren findet bei uns nur Anwendung bei arctan und artanh (bzw. ln), so dass wir uns auf deren Gegebenheiten einstellen können. In beiden Potenzreihen kommen nur ungerade Terme vor, also werten wir nach Ausklammerung von x ein Polynom in $u = x^2$ aus, d.h. unser Argument ist immer als gestört zu betrachten. Der Einfachheit

halber soll der relative Fehler der Störung aber im selben Intervall E_ℓ liegen wie das der Fehler der arithmetischen Operationen während der Auswertung. Im Anschluss an das eigentliche Verfahren wird mit x multipliziert; daher soll im letzten Schritt des Verfahrens noch gar nicht gerundet werden, erst nach dieser Multiplikation.

Lemma 4 *Das Approximationspolynom $P(u)$ sei wie in (4.16) geklammert und werde entsprechend ausgewertet. Das Argument u sei gestört als $\tilde{u} \in u(1+E_\ell)$. Alle Koeffizienten seien gestört als $\tilde{a}_k \in a_k \cdot (1+E_\ell)$, nur a_0 sei im verwendeten Fließkommasystem exakt darstellbar. Die Multiplikationen mögen exakt durchgeführt werden, bei jeder Addition soll ins ℓ-stellige Raster gerundet werden. Im letzten Schritt soll überhaupt nicht gerundet werden.*
Dann gilt für den absoluten Auswertefehler $\Delta_P = \tilde{P} - P$:

$$\Delta_P \in E_\ell \left[\sum_{k=1}^{N-1} |a_k u^k|(1+E_\ell)^{2k} + (2+E_\ell) \sum_{k=1}^{N} |r_k u^k|(1+E_\ell)^{2k-2} \right].$$

Beweis: Da der letzte Schritt exakt sein soll, können wir nicht die eigentliche Aussage von Lemma 3 verwenden, wohl aber die im Beweis gemachte Teilaussage für Δ_1. Da $(1+\varepsilon_u)^n \in (1+E_\ell)^n$, ergibt sich folgendes:

$$\begin{aligned}
\Delta_1 &\in r_1 E_\ell + \sum_{k=1}^{N-1} u^{k-1}(a_k E_\ell + u\varepsilon_u r_{k+1})(1+\varepsilon_u)^{k-1}(1+E_\ell)^k + \sum_{k=1}^{N-1} u^k r_{k+1} E_\ell (1+\varepsilon_u)^k (1+E_\ell)^k \\
&\subseteq r_1 E_\ell + \sum_{k=1}^{N-1} u^{k-1}(a_k E_\ell + u r_{k+1} E_\ell)(1+E_\ell)^{2k-1} + \sum_{k=1}^{N-1} r_{k+1} u^k E_\ell (1+E_\ell)^{2k} \\
&\subseteq r_1 E_\ell + E_\ell \sum_{k=1}^{N-1} |u^{k-1} a_k|(1+E_\ell)^{2k-1} + E_\ell \sum_{k=1}^{N-1} |r_{k+1} u^k|(1+E_\ell)^{2k-1} \\
&\quad + E_\ell \sum_{k=1}^{N-1} |r_{k+1} u^k|(1+E_\ell)^{2k} \\
&\subseteq E_\ell \left[|r_1| + \sum_{k=1}^{N-1} |a_k u^{k-1}|(1+E_\ell)^{2k-1} + (2+E_\ell) \sum_{k=1}^{N-1} |r_{k+1} u^k|(1+E_\ell)^{2k-1} \right]
\end{aligned}$$

Im letzten Schritt wird nicht gerundet, wohl ist aber das Argument verfälscht:

$$\begin{aligned}
\mathrm{b}_0) \quad \tilde{r}_0 &\in a_0 + u(1+E_\ell)(r_1 + \Delta_1) \\
&\subseteq r_0 + u r_1 E_\ell + u(1+E_\ell)\Delta_1, \\
\Delta_0 &\in u r_1 E_\ell + u(1+E_\ell)\Delta_1,
\end{aligned}$$

d.h. wir erhalten insgesamt

$$\begin{aligned}
\Delta_0 &\in E_\ell \left[|u r_1| + |u r_1|(1+E_\ell) + \sum_{k=1}^{N-1} |a_k u^k|(1+E_\ell)^{2k} + (2+E_\ell) \sum_{k=1}^{N-1} |r_{k+1} u^{k+1}|(1+E_\ell)^{2k} \right] \\
&= E_\ell \left[(2+E_\ell)|u r_1| + \sum_{k=1}^{N-1} |a_k u^k|(1+E_\ell)^{2k} + (2+E_\ell) \sum_{k=1}^{N-1} |r_{k+1} u^{k+1}|(1+E_\ell)^{2k} \right],
\end{aligned}$$

was der Behauptung entspricht. □

In der folgenden Form kann die Aussage bei arctan direkt verwendet werden:

Korollar 5 *Das Approximationspolynom lasse sich darstellen als $P(x) = x \cdot Q(x^2)$, wobei Q so ausgewertet werde wie in Lemma 4. Bei der abschließenden Multiplikation mit x werde in das ℓ-stellige Raster gerundet. Wenn die Stellenzahl ℓ die Voraussetzung für Lemma 2 erfüllt, gilt für den absoluten Gesamtfehler Δ_P:*

$$\begin{aligned}\Delta_P \;\in E_\ell\Big[\; & |r_0 x| \;+\; |r_1 x^3|(1+E_\ell)(2+E_\ell) \\ & + (2+E_\ell)\sum_{k=2}^{N}|r_k x^{2k+1}| \;+\; (1+E_\ell)^2\sum_{k=1}^{N-1}|a_k x^{2k+1}| \\ & + 2E_\ell(2+E_\ell)\sum_{k=2}^{N}|k r_k x^{2k+1}| \;+\; 2E_\ell(1+E_\ell)^2\sum_{k=1}^{N-1}|k a_k x^{2k+1}|\;\Big].\end{aligned}$$

Beweis: Für den Abschluss-Schritt gilt

$$\begin{aligned}\tilde{P} \;&\in\; x(Q+\Delta_Q)(1+E_\ell) \;\subseteq\; xQ + xQE_\ell + x\Delta_Q(1+E_\ell) \\ &=\; P + xr_0E_\ell + x\Delta_Q(1+E_\ell) \\ \Delta_P \;&\in\; xr_0E_\ell + x\Delta_Q(1+E_\ell),\end{aligned}$$

also haben wir mit Lemma 4 insgesamt

$$\begin{aligned}\Delta_P \;&\in E_\ell\left[\,|r_0x| + \sum_{k=1}^{N-1}|a_kx^{2k+1}|(1+E_\ell)^{2k+1} + (2+E_\ell)\sum_{k=1}^{N}|r_kx^{2k+1}|(1+E_\ell)^{2k-1}\,\right] \\ &\subseteq E_\ell\Big[\,|r_0x| + |r_1x^3|(1+E_\ell)(2+E_\ell) + (1+E_\ell)^2\sum_{k=1}^{N-1}|a_kx^{2k+1}|(1+E_\ell)^{2k-1} \\ &\qquad +(2+E_\ell)\sum_{k=2}^{N}|r_kx^{2k+1}|(1+E_\ell)^{2k-1}\,\Big] \\ &\overset{\text{L1}}{\subseteq} E_\ell\Big[\,|r_0x| + |r_1x^3|(1+E_\ell)(2+E_\ell) + (1+E_\ell)^2\sum_{k=1}^{N-1}|a_kx^{2k+1}|(1+2kE_\ell) \\ &\qquad +(2+E_\ell)\sum_{k=2}^{N}|r_kx^{2k+1}|(1+2kE_\ell)\,\Big].\end{aligned}$$

Aufteilen der beiden Summen liefert die Behauptung. □

Für artanh (resp. ln) werden wir folgende vereinfachte Version benutzen:

Korollar 6 *Wenn unter den Voraussetzungen von Korollar 5 zusätzlich das Argument und alle Koeffizienten a_k positiv sind, gilt für den absoluten Auswertefehler*

$$\begin{aligned}\Delta_P \;\subseteq\; E_\ell\Big[\; & r_0x \;+\; r_1x^3(1+E_\ell) \;+\; (2+E_\ell)(1+E_\ell)\sum_{k=1}^{N} r_k\,x^{2k+1} \\ & + 2E_\ell(2+E_\ell)(1+E_\ell)\sum_{k=1}^{N} k\,r_k\,x^{2k+1}\;\Big].\end{aligned}$$

Beweis: Durch die Zusatzvoraussetzung sind auch alle $r_k \geq 0$. In der Beziehung für Δ_1 in Lemma 4 erscheinen beim Ausklammern von E_ℓ nun keine Beträge. Man erhält

$$\begin{aligned}\Delta_1 \;&\in\; E_\ell\left[\, r_1 + \sum_{k=1}^{N-1} r_k u^{k-1}(1+E_\ell)^{2k-1} + \sum_{k=1}^{N-1} r_{k+1}u^{k-1}(1+E_\ell)^{2k}\,\right] \\ &\subseteq\; E_\ell\left[\, r_1(2+E_\ell) + (2+E_\ell)\sum_{k=1}^{N} r_k u^{k-1}(1+E_\ell)^{2k-2}\,\right],\end{aligned}$$

wobei zum Zusammenfassen der beiden Summen der ersten ein vorher nicht enthaltener positiver (kleiner) Summand für $k = N$ hinzugefügt wurde. Damit erhält man nun:

$$\begin{aligned}
\Delta_1 &\in E_\ell(2+E_\ell)\sum_{k=1}^{N} r_k u^{k-1}(1+E_\ell)^{2k-2}, \\
\Delta_0 &\in E_\ell\left[r_1 u + (2+E_\ell)\sum_{k=1}^{N} r_k u^k (1+E_\ell)^{2k-1}\right], \\
\Delta_P &\in E_\ell\left[r_0 x \;+\; r_1 x^3(1+E_\ell) \;+\; (2+E_\ell)(1+E_\ell)\sum_{k=1}^{N} r_k x^{2k+1}(1+E_\ell)^{2k-1}\right].
\end{aligned}$$

Anwenden von Lemma 2 und Aufteilen der Summe liefert die Behauptung. □

4.1.8 Modifiziertes Horner-Verfahren

Bei vielen der Standardfunktionen kann die Berechnung deutlich (um einen Faktor 2 und mehr) dadurch beschleunigt werden, dass die Koeffizienten auseinander gerissen, d.h. in die Ausklammerung des Horner-Schemas mit einbezogen werden. Das ist z.B. möglich, wenn sie Fakultäten oder fakultätsähnliche Terme enthalten. Ziel ist hierbei keine numerische Verbesserung; vielmehr werden Operationen zur Bestimmung der Koeffizienten eingespart, oder arithmetische Operationen können miteinander verschmolzen werden.

Beispielsweise ist das Approximationspolynom zur Exponentialfunktion

$$P(x) = \sum_{k=0}^{N} \frac{1}{k!}\, x^k,$$

was in normaler Horner-Klammerung wie folgt aussehen würde:

$$P(x) = 1 + x\cdot\left(1 + x\cdot\left(\frac{1}{2!} + x\cdot\left(\frac{1}{3!} + x\cdot\left(\frac{1}{4!} + \ldots + x\cdot\left(\frac{1}{(N-1)!} + x\cdot\frac{1}{N!}\right)\right)\cdots\right)\right)\right),$$

d.h. man hat folgendes Auswertungsverfahren:

$$\begin{aligned}
r_N &:= \frac{1}{N!}, \\
r_k &:= \frac{1}{k!} + x\cdot r_{k+1} \qquad \text{für } k = N-1,\ldots,0, \\
\Rightarrow\; & P(x) = r_0.
\end{aligned}$$

Nachteilig hierbei ist, dass in jedem Schritt eine Fakultät benötigt wird, die durch eine *lange* Integer-Division aus der vorhergehenden gewonnen werden muss (die erste, $1/N!$, erhält man automatisch aus der Bestimmung der notwendigen Anzahl auszuwertender Glieder), und zusätzlich wird eine *lange* Fließkomma-Division $1/k!$ durchgeführt.

Daher geht die vorliegende Implementation von folgender Klammerung aus:

$$P(x) = 1 + x\cdot\left(1 + \frac{x}{2}\cdot\left(1 + \frac{x}{3}\cdot\left(1 + \frac{x}{4}\cdot\left(\ldots\cdot\left(1 + \frac{x}{N-1}\cdot\left(1 + \frac{x}{N}\right)\right)\ldots\right)\right)\right)\right). \tag{4.17}$$

Abgesehen davon, dass man sich in jedem Schritt eine Division zur Berechnung der Nenner spart, sind die vorkommenden Divisoren auf diese Weise *kleine* Integerzahlen, die in der Implementation der BCD-Arithmetik in den normalen Anwendungsfällen durch eine BCD-Ziffer (z.B. $\in [0, 9999]$) dargestellt werden. Die entsprechenden Divisionen sind um eine Größenordnung schneller als die durch die ggf. schon recht großen Fakultäten.

Als Auswertungsverfahren ergibt sich:

$$\begin{aligned}
&a) \quad r_N &&:= x/N, \\
&b) \quad r_k &&:= (x \cdot (r_{k+1} + 1))/k \qquad \text{für } k = N-1, \ldots, 1, \\
&c) \quad r_0 &&:= r_1 + 1, \\
&\Rightarrow \quad P(x) &&= r_0.
\end{aligned}$$

In diesem speziellen Fall werden die Schritte $k = 1$, $k = 2$ noch gesondert behandelt werden, da die Divisionen durch 2 bzw. 1 exakt durchgeführt bzw. unterdrückt werden (siehe dazu den Abschnitt zur Exponentialfunktion).

Unsere Abschätzungen zum Auswertefehler bei diesem modifizierten Verfahren liefern in allen Fällen etwas kleinere Schranken als die in [Steins] berechneten. Dadurch bleibt zwar theoretisch auch mehr Raum für den erlaubten Approximationsfehler, was sich aber praktisch nie in einer kleineren Zahl zu berechnenden Glieder der Reihe niederschlägt.

Klammerungen wie hier bei exp sind auch möglich bei sin, cos, sinh und arcosh. Eine noch weiter modifizierte Klammerung wird für arcsin und arsinh benutzt. Für die Anwendung bei ersteren Standardfunktionen beweisen wir den folgenden Satz von Lemmas.

4.1.9 Auswertefehler beim modifizierten Horner-Verfahren

Bei sin, cos und sinh kommen in der Potenzreihe wiederum nur gerade oder nur ungerade Potenzen vor, in arcosh das Argument $x - 1$, so dass wir wieder von einem gestörten Argument und exaktem letzten Schritt ausgehen. Im Hinblick auf cos wird eine gesonderte Aussage für ein exakt darstellbares a_1 gemacht. Nur bei exp ist das Argument ungestört, eine entsprechende Modifikation wird im Kapitel zur Exponentialfunktion angegeben.

Lemma 7 *Die Koeffizienten des Approximationspolynoms $P(x) = P_N(x)$ seien rational und fakultätsähnlich in folgendem Sinn:*

$$P(x) = a_0 + \sigma a_0 a_1 x + a_0 a_1 a_2 x^2 + \sigma a_0 a_1 a_2 a_3 x^3 + \ldots + \sigma^N a_0 a_1 a_2 \ldots a_N x^N, \qquad \sigma \in \{-1, +1\}.$$

Das Argument x sei gestört mit relativem Fehler in E_ℓ. Das Horner-Verfahren werde so modifiziert, dass die Koeffizienten in die Klammerung mit einbezogen werden. In jedem Schritt des Verfahrens werde nur am Ende einmal gerundet, mit relativem Fehler in E_ℓ. Die $a_i \in \mathbb{Q}$ brauchen für $k \geq 1$ nicht notwendig im Fließkommaraster exakt darstellbar zu sein; a_0 sei exakt darstellbar, und der ganze letzte Schritt möge exakt durchgeführt werden.
Dann gilt für den absoluten Auswertefehler $\Delta_0 = \tilde{r}_0 - r_0$:

$$\Delta_0 \;\in\; a_0\, r_1\, E_\ell\,(2 + E_\ell) + E_\ell\,(2 + E_\ell) \sum_{k=2}^{N} \left| \Big(\prod_{i=0}^{k-1} a_i \Big) r_k\, x^{k-1} \right| (1 + E_\ell)^{2k-2}.$$

Wenn auch a_1 exakt dargestellt (bzw. die Multiplikation damit exakt durchgeführt) werden kann, gilt sogar:

$$\Delta_0 \;\in\; a_0\, r_1\, E_\ell + E_\ell\,(2+E_\ell) \sum_{k=2}^{N} \Big| \Big(\prod_{i=0}^{k-1} a_i \Big) r_k\, x^{k-1} \Big| \,(1+E_\ell)^{2k-3}.$$

Beweis: Aus der Klammerung

$$P(x) = a_0(1 + \sigma a_1 x(1 + \sigma a_2 x(1 + \ldots + \sigma a_N x^N)) \ldots)$$

erhalten wir das Verfahren

$$\begin{array}{lrll}
\text{a)} & r_N & := a_N \cdot x, & \\
\text{b)} & r_k & := a_k \cdot x \cdot (1 + \sigma\, r_{k+1}) & \text{für } k = N-1, \ldots, 1, \\
\text{c)} & r_0 & := a_0\,(1 + \sigma\, r_1), & \\
\Rightarrow & P(x) & = r_0. &
\end{array}$$

Da $a_k \in \mathbb{Q}$, wird die Multiplikation mit a_k als exakte Multiplikation mit dem Zähler und anschließende gerundete Division durch den Nenner dargestellt.

In den $\tilde{r}_k$ erhalten wir jeweils einen Term $(1+E_\ell)$ durch das gestörte x und einen durch die inexakte Division:

$$\begin{array}{lrcl}
\text{a)} & \tilde{r}_N & \in & a_N\, x\,(1+E_\ell)^2 \;=\; r_N\,(1+E_\ell)^2 \\
& & \subseteq & r_N\,(1 + E_\ell(2+E_\ell)) \;=\; r_N + r_N\, E_\ell(2+E_\ell) \\
& \Delta_N & \in & r_N(2+E_\ell)E_\ell \\
\text{b)} & \tilde{r}_k & \in & (a_k\, x\,(1+\sigma\,(r_{k+1}+\Delta_{k+1})))(1+E_\ell)^2 \\
& & = & (r_k + a_k\, x\, \sigma\, \Delta_{k+1})(1+E_\ell)^2 \\
& & \subseteq & r_k\,(1+E_\ell(2+E_\ell)) + a_k\, x\, \sigma\, \Delta_{k+1}\,(1+E_\ell)^2 \\
& & = & r_k + r_k\, E_\ell(2+E_\ell) + a_k\, x\, \sigma\, \Delta_{k+1}\,(1+E_\ell)^2 \\
& \Delta_k & \in & r_k\, E_\ell(2+E_\ell) + a_k\, x\, \sigma\, \Delta_{k+1}\,(1+E_\ell)^2 \\
\text{c)} & \tilde{r}_0 & = & a_0(1+\sigma\,(r_1+\Delta_1)) \;=\; r_1 + \sigma\, a_0\, \Delta_1 \\
& \Delta_0 & = & \sigma\, a_0\, \Delta_1
\end{array}$$

Durch sukzessives Einsetzen erhalten wir

$$\begin{array}{rcl}
\Delta_2 & \in & r_2 E_\ell(2+E_\ell) + \sigma a_2 x(1+E_\ell)^2 \Delta_3 \\
& \subseteq & r_2 E_\ell(2+E_\ell) + \sigma a_2 x(1+E_\ell)^2\,(r_3 E_\ell(2+E_\ell) + \sigma a_3 x(1+E_\ell)^2 \Delta_4) \\
& \subseteq & r_2 E_\ell(2+E_\ell) + a_2 r_3 x E_\ell(2+E_\ell)(1+E_\ell)^2 + a_2 a_3 x(1+E_\ell)^4 \Delta_4,
\end{array}$$

bzw. insgesamt

$$\Delta_2 \;\in\; \sum_{k=2}^{N} \Big(\prod_{i=2}^{k-1} a_i \Big) r_k x^{k-2} E_\ell(2+E_\ell)(1+E_\ell)^{2k-4}.$$

Wenn auch a_1 nicht exakt darstellbar ist, gilt nun entsprechend

$$\begin{array}{rcl}
\Delta_1 & \in & r_1 E_\ell(2+E_\ell) + \sigma a_1 x(1+E_\ell)^2 \Delta_2 \\
& \subseteq & r_1 E_\ell(2+E_\ell) + \displaystyle\sum_{k=2}^{N} \Big(\prod_{i=1}^{k-1} a_i \Big) r_k x^{k-1} E_\ell(2+E_\ell)(1+E_\ell)^{2k-2}
\end{array}$$

und mit $\Delta_0 = \sigma a_0 \Delta_1$:

$$\begin{aligned}\Delta_0 &\in a_0 r_1 E_\ell(2+E_\ell) + \sum_{k=2}^{N}\Big(\prod_{i=0}^{k-1} a_i\Big) r_k x^{k-1} E_\ell(2+E_\ell)(1+E_\ell)^{2k-2}.\\ &\subseteq a_0 r_1 E_\ell(2+E_\ell) + E_\ell(2+E_\ell)\sum_{k=2}^{N}\left|\Big(\prod_{i=0}^{k-1} a_i\Big) r_k x^{k-1}\right|(1+E_\ell)^{2k-2}.\end{aligned}$$

Falls a_1 exakt darstellbar ist, wird der Schritt b) modifiziert:

$$\begin{aligned}\text{b)}\quad \tilde{r}_1 &\in a_1 x\,(1+\sigma(r_2+\Delta_2))\,(1+E_\ell)\\ &= (r_1 + a_1 x\sigma\Delta_2)\,(1+E_\ell)\\ &\subseteq r_1 + r_1 E_\ell + a_1 x\sigma(1+E_\ell)\Delta_2\\ \Delta_1 &\in r_1 E_\ell + a_1 x\sigma(1+E_\ell)\Delta_2,\end{aligned}$$

so dass sich auch dort insgesamt das Behauptete ergibt:

$$\Delta_0 = \sigma a_0 \Delta_1 \in a_0 r_1 E_\ell + E_\ell(2+E_\ell)\sum_{k=2}^{N}\left|\Big(\prod_{i=0}^{k-1} a_i\Big) r_k x^{k-1}\right|(1+E_\ell)^{2k-3}. \qquad \square$$

Bemerkungen:

a) Es ist zu beachten, dass die Forderung, den letzten Schritt vollständig exakt durchzuführen, insbesondere den Term $1+\sigma r_1$, bei sehr kleinem r_1 zu großen Mantissenlängen führen kann. Das wird normalerweise nur bei extrem kleinen Argumenten der Funktion geschehen, bei denen eigentlich bereits das erste Glied der Potenzreihe für die Berechnung ausreichen würde. Aufgrund der Beschaffenheit unserer Abschätzungen müssen aber immer mindestens 3 Glieder berechnet werden.

Bei den Funktionen, deren Auswertung auf diesem Lemma beruht, werden sehr kleine Argumente (auch aus Geschwindigkeitsgründen) ohnehin gesondert behandelt, etwa $\cos x \approx 1$, $\sin x \approx x$ bei genügend kleinem x (abhängig von der Stellenforderung). Das wird in den zugehörigen Abschnitten besprochen werden.

Die Version zu [Steins] berechnet den letzten Schritt teilweise auch exakt und stürzt dann manchmal (z.B. bei sinh) vermutlich eben wegen überlanger Mantissen mit einem Segmentierungsfehler ab.

b) Im Fall $\sigma = -1$ muss in der Implementation bei der Berechnung nur einer oberen oder nur einer unteren Grenze einer Einschließung natürlich jeweils auf die korrekte Richtung der Rundung geachtet werden! Das Polynom zum Sinus sieht beispielsweise wie folgt aus:

$$P_N^{\sin}(x) = x\left(1 - \frac{x^2}{2\cdot 3}\Big(1 - \frac{x^2}{4\cdot 5}\Big(1 - \frac{x^2}{6\cdot 7}\Big(1 - \ldots\Big(1 - \frac{x^2}{(2N)(2N+1)}\Big)\Big)\ldots\Big)\right).$$

Es muss in geradzahligen Schritten (z.B. im letzten bei r_0) in die gewünschte Endrichtung gerundet werden, in ungeradzahligen Schritten in die umgekehrte Richtung. Das bedeutet, dass im N-ten (zuerst durchgeführten Schritt) immer nach oben gerundet werden muss. Üblicherweise werden jeweils zwei aufeinander folgende Schritte in einen Schleifendurchgang zusammengefasst. Bei Berechnung einer Obergrenze muss danach ein einzelner Schritt für r_1 mit Rundung nach unten nachgeholt werden.

Zur späteren Verwendung bei cos geben wir folgendes Korollar an.

Korollar 8 *Das Approximationspolynom sei darstellbar als*

$$P(x) = Q(x^2),$$

wobei $Q(u)$ wie in Lemma 7 klammerbar sei und wie dort ausgewertet werde. $u = x^2$ sei gerundet, mit einem relativen Fehler in E_ℓ. Auch der Koeffizient a_1 sei exakt darstellbar. Ganz am Schluss werde in das ℓ-stellige Raster gerundet. Dann gilt für den absoluten Gesamtfehler Δ_P:

$$\begin{aligned}\Delta_P \in E_\ell\Big[\ |r_0| + |a_0 r_1|\,(1+E_\ell) + (2+E_\ell)\sum_{k=2}^{N}\Big|\Big(\prod_{i=0}^{k-1} a_i\Big) r_k\, x^{2k-2}\Big| \\ + E_\ell(2+E_\ell)\sum_{k=2}^{N}\Big|\Big(\prod_{i=0}^{k-1} a_i\Big) r_k\, x^{2k-2}\Big|\,(2k-1)\ \Big].\end{aligned}$$

Beweis: Wir können die zweite in Lemma 7 angegebene Beziehung für Δ_0 verwenden, allerdings für das Argument x^2:

$$\Delta_0 \ \in\ a_0\, r_1\, E_\ell + E_\ell\,(2+E_\ell)\sum_{k=2}^{N}\Big|\Big(\prod_{i=0}^{k-1} a_i\Big) r_k\, x^{2k-2}\Big|\,(1+E_\ell)^{2k-3}.$$

Insgesamt gilt $P(x) = r_0$, bzw. gestört

$$\begin{aligned}\tilde{P} &\in (r_0+\Delta_0)(1+E_\ell)\\ &\subseteq r_0 + r_0E_\ell + \Delta_0(1+E_\ell),\\ \Delta_P &\in r_0E_\ell + (1+E_\ell)\Delta_0\\ &\subseteq r_0E_\ell + a_0r_1E_\ell(1+E_\ell) + E_\ell(2+E_\ell)\sum_{k=2}^{N}\Big|\Big(\prod_{i=0}^{k-1} a_i\Big) r_k\, x^{2k-2}\Big|\,(1+E_\ell)^{2k-2}.\end{aligned}$$

Durch Anwendung von Lemma 2 ergibt sich nun

$$\begin{aligned}\Delta_P \ \subseteq\ & r_0E_\ell + a_0r_1E_\ell(1+E_\ell) + E_\ell(2+E_\ell)\sum_{k=2}^{N}\Big|\Big(\prod_{i=0}^{k-1} a_i\Big) r_k\, x^{2k-2}\Big| \\ & + E_\ell^2(2+E_\ell)\sum_{k=2}^{N}\Big|\Big(\prod_{i=0}^{k-1} a_i\Big) r_k\, x^{2k-2}\Big|(2k-1),\end{aligned}$$

was nach Ausklammern von E_ℓ die Behauptung liefert. □

Für die Funktionen sin und sinh ist folgendes Korollar gedacht:

Korollar 9 *Das Approximationspolynom P sei darstellbar als*

$$P(x) = x \cdot Q(x^2),$$

wobei $Q(u)$ wie in Lemma 7 klammerbar sei und wie dort ausgewertet werde. $u = x^2$ sei gerundet, mit einem relativen Fehler in E_ℓ. Ganz am Schluss werde in das ℓ-stellige Raster gerundet. Dann gilt für den absoluten Gesamtfehler Δ_P:

$$\begin{aligned}\Delta_P \;\in\; E_\ell\Big[\; |r_0 x| + |a_0 r_1 x|(1+E_\ell)(2+E_\ell) + (2+E_\ell)\sum_{k=2}^{N}\Big|\Big(\prod_{i=0}^{k-1} a_i\Big)\, r_k x^{2k-1}\Big| \\ +2E_\ell(2+E_\ell)\sum_{k=2}^{N}\Big|\Big(\prod_{i=0}^{k-1} a_i\Big)\, r_k\, k\, x^{2k-1}\Big|\;\Big].\end{aligned}$$

Beweis: Da a_1 nicht notwendigerweise exakt darstellbar ist, verwenden wir hier die erste Beziehung für Δ_0 aus Lemma 7, umgesetzt auf das Argument x^2:

$$\Delta_0 \;\in\; a_0\, r_1\, E_\ell\,(2+E_\ell) + E_\ell\,(2+E_\ell)\sum_{k=2}^{N}\Big|\Big(\prod_{i=0}^{k-1} a_i\Big) r_k\, x^{2k-2}\Big|\,(1+E_\ell)^{2k-2}.$$

Insgesamt gilt $P = P(x) = x \cdot r_0$, bzw. gestört

$$\begin{aligned}\tilde{P} \;&\in\; x(r_0+\Delta_0)(1+E_\ell) \\ &=\; (P + x\Delta_0)(1+E_\ell) \\ &\subseteq\; P + PE_\ell + x(1+E_\ell)\Delta_0, \\ \Delta_P \;&\in\; r_0 x E_\ell + x\Delta_0(1+E_\ell) \\ &\subseteq\; r_0 x E_\ell + a_0 r_1 x E_\ell(1+E_\ell)(2+E_\ell) + E_\ell(2+E_\ell)\sum_{k=2}^{N}\Big|\Big(\prod_{i=0}^{k-1} a_i\Big) r_k\, x^{2k-1}\Big|\,(1+E_\ell)^{2k-2} \\ &\subseteq\; r_0 x E_\ell + a_0 r_1 x E_\ell(1+E_\ell)(2+E_\ell) + E_\ell(2+E_\ell)\sum_{k=2}^{N}\Big|\Big(\prod_{i=0}^{k-1} a_i\Big) r_k\, x^{2k-1}\Big| \\ &\qquad +2E_\ell^2(2+E_\ell)\sum_{k=2}^{N}\Big|\Big(\prod_{i=0}^{k-1} a_i\Big) r_k\, k\, x^{2k-1}\Big|,\end{aligned}$$

wobei im letzten Schritt Lemma 2 verwendet wurde. Ausklammern von E_ℓ liefert die Behauptung. □

Bemerkung: Die Koeffizienten einiger Potenzreihen (z.B. arcsin und arsinh) sind „teilweise fakultätsähnlich“ in folgendem Sinn:

$$P_\infty(x) = b_0 + \sigma a_1 b_1 x + a_1 a_2 b_2 x^2 + \sigma a_1 a_2 a_3 b_3 x^3 + a_1 a_2 a_3 a_4 b_4 x^4 + \ldots, \qquad \sigma \in \{-1, +1\},$$

so dass das Approximationspolynom wie folgt geklammert werden kann:

$$P_N(x) = b_0 + \sigma a_1 x(b_1 + \sigma a_2 x(b_2 + \sigma a_3 x(b_3 + \sigma a_4 x(b_4 + \ldots + \sigma a_N x \cdot b_N)))\ldots).$$

Das entsprechend modifizierte Horner-Verfahren hat dann folgende Gestalt:

$$\begin{aligned}&\text{a)}\quad r_N = b_N, \\ &\text{b)}\quad r_k = b_k + \sigma a_{k+1}\, r_{k+1}\, u, \qquad \text{für } k = N-1, \ldots, 0.\end{aligned}$$

Nun haben die a_k, b_k bei den erwähnten Funktionen die spezielle Gestalt $a_k = \frac{c_k}{d_k}$ und $b_k = \frac{1}{e_k}$ mit c_k, d_k, $e_k \in \mathbb{N}$, und es hat sich als vorteilhaft erwiesen, den Schritt b) durch Transformation auf den Hauptnenner wie folgt umzuformen:

$$r_k = \frac{d_{k+1} \; + \; \sigma e_k \, c_{k+1} \, r_{k+1} \, u}{e_k \, d_{k+1}} \, .$$

Dabei ist zu beachten, dass sich die Multiplikationen $e_k c_{k+1}$ und $e_k d_{k+1}$ im Integer-Bereich (`long`) abspielen. Außerdem entfällt eine abschließende Rundung, die durch die Division gleichsam mit erledigt wird. Man erhält so eine deutliche Beschleunigung.

Da diese Auswertung bei arcsin und arsinh so speziell ist und außerdem die Kenntnis des jeweiligen σ ausgenutzt werden kann, geben wir hier keine allgemeine Aussage für solche Klammerungen an, sondern leiten die beiden Fehlerabschätzungen in den beiden zugehörigen Abschnitten gesondert her.

4.1.10 Bestimmung der Ordnung des Approximationspolynoms

Es gilt nun noch, ein genügend großes (aber möglichst kleines) N zu finden, so dass der relative Fehler bei Abbruch der Reihenauswertung nach $N+1$ Gliedern betragsmäßig unter der Schranke aus (4.12) verbleibt.

Aus den Restglied-Darstellungen der Potenzreihen ergeben sich für die einzelnen Fälle spezielle Bedingungen, beim Sinus beispielsweise (siehe Abschnitt 4.3.4):

$$\frac{x^{2N+2}}{(2N+3)!} \left| \frac{x}{\sin x} \right| \; \leq \; 8.7528 \cdot 10^{-p-1}.$$

Dabei ist 8.7528 die Konstante c aus (4.10) für die Sinus-Funktion, berechnet auf Seite 92.

In den Restgliedern kommen naturgemäß Potenzen von x und Fakultäten (oder ähnliche Terme) vor, die ein Auflösen nach N unmöglich machen. Bei Arithmetiken mit fester oder beschränkter Genauigkeit kann ein passendes N fest angegeben oder einer kleinen Tabelle entnommen werden (für sin siehe z.B. [Braune], Seite 53). In [Braune] und [Krämer] beispielsweise wird dabei das Restglied immer auf dem gesamten erlaubten Argumentintervall (typischerweise $]0, \bar{x}]$) betrachtet, was zu etwas zu großen N typischerweise für x am linken Rand führt.

Da wir theoretisch beliebige Genauigkeit erzielen können müssen, bleibt uns prinzipiell wenig anderes übrig, als die Bedingung mit steigendem N durchzutesten. Immerhin können wir dabei dann das tatsächliche x verwenden und kommen meist zu etwas kleineren N als globale Versionen.

Der Test könnte bei $N := 1$ beginnen, dann jeweils die Potenzen und die vorkommenden Fakultäten aufmultiplizieren und in jedem Schritt einen Vergleich durchführen. So geht beispielsweise auch [Steins] vor. Die Bestimmung von N kann dann einen erheblichen Teil der gesamten Laufzeit ausmachen.

In der vorliegenden Implementation wird daher ein Startwert N_0 *geschätzt*. Die erste Potenz (beim Sinus x^{2N_0+2}) und die erste Fakultät (beim Sinus $(2N_0+3)!$) müssen dann natürlich

vorab berechnet werden. Die zugehörigen Funktionen `pow` bzw. `BIfac` sind aber wesentlich schneller als ein Aufmultiplizieren, verbunden mit einem $\leq$-Test in jedem Schritt. Selbstverständlich sollte N_0 möglichst nicht zu groß sein, damit das Berechnen unnötig vieler Glieder der Potenzreihe vermieden wird. Angesichts der Laufzeitersparnis zu Beginn wäre ein um eins oder zwei zu großes N_0 aber unproblematisch.

Unsere Schätzung entsteht durch bilineares Interpolieren in einer vorberechneten Tabelle mit exakten N. Das Intervall (typischerweise $]0, x_{max}]$), in dem sich das reduzierte Argument x bewegt, wird in z.B. zehn Teilintervalle zerlegt ($j = j_x := 10\,\frac{x}{x_{max}}$), der Bereich von $p = 3$ bis $p = 1000$ in z.B. fünf Teilbereiche. Beispielsweise sieht die Tabelle zum Sinus wie folgt aus:

p	$j=1$	2	3	4	5	6	7	8	9	10
3	3	3	3	3	3	3	3	3	3	3
10	3	3	4	4	4	5	5	5	5	6
33	8	9	10	11	11	12	12	13	13	14
100	21	23	25	27	28	29	30	31	32	32
330	59	65	69	73	76	78	80	82	84	85
1000	157	171	181	188	194	199	204	208	212	216

Beim Sinus ist $x_{max} = \frac{\pi}{4} \approx 0.7854$. Der Wert zu $x = 0.5$ (zwischen $j = 6$ und $j = 7$) und $p = 80$ ergibt sich durch Interpolation aus dem Rechteck $\begin{smallmatrix}12 & 12\\ 29 & 30\end{smallmatrix}$, man erhält den Wert 23. Das exakte N ist 24.

Beim Sinus und $x = 0.5$ ist diese Version der Bestimmung von N gegenüber der mit $N_0 = 1$ beginnenden (bei gleicher unterliegender Arithmetik auf dem L*i*nux-Rechner) bei 50 Stellen um etwa den Faktor 2, bei 100 Stellen um den Faktor 5, bei 1000 Stellen um den Faktor 59 schneller.

Die Aufteilung in Teilintervalle kann der jeweiligen Funktion angepasst werden, um etwa unterschiedlichem Steigungsverhalten Rechnung zu tragen. Der Einfachheit halber wurde in der vorliegenden Implementation darauf verzichtet. Dadurch kann die Bestimmung des Startwerts leichter in eine Routine ausgelagert werden, die nur die Tabelle und die Grenzen der Intervalle übergeben zu bekommen braucht:

```
long Ntable_lookup( const BigFloat &x, long p,
   const BigFloat &x_factor,
   const long *Ntable, int p_intervals , int x_intervals );
```

In der aktuellen Version ist der Einfachheit halber immer `p_intervals`= 6 und `x_intervals`= 10 wie im Sinus-Beispiel.

Da zwei aufeinander folgende Schranken für $|\varepsilon_{app}|$ immer gleich Größenordnungen auseinander liegen, brauchen wir bei den Potenzen des Arguments x nicht die exakte Routine `pow(x,i)` zu verwenden. Stattdessen kommt eine Version zum Einsatz, die mit Hilfe der vorhandenen IEEE-Arithmetik (so vorhanden) eine etwas größere Approximation berechnet. Es wird dabei nicht vorausgesetzt, dass diese Arithmetik bis auf die letzte Stelle genau arbeitet, sondern eine gewisse Toleranz eingebaut. Das Argument, das bei der `C`-Funktion `pow` ankommt, liegt in $[1, 10]$. Exponenten über 300 werden wie unten angegeben aufgesplittet, so dass es nie zu einem IEEE-Über- oder -Unterlauf kommt.

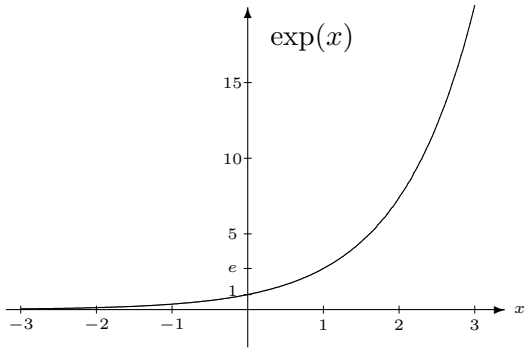
```
BigFloat approx_pow_up(const BigFloat &x, long l)
 {
   long o=order(x);
   double ieeearg=((x>>o)^12).double_value();
   if (l>300)
    {
      long k=l/300;
      return ((pow(BigFloat(pow(ieeearg,300.0)),k)
                  *BigFloat(pow(ieeearg,l-300*k)))^16) << (o*l);
    }
   return BigFloat( pow(ieeearg,(double)l) ) << (o*l) ;
 }
```

4.2 Die Exponentialfunktion

Wir verwenden zur Berechnung die auf ganz $\mathbb{R}$ konvergente Potenzreihe

$$\exp(x) \;=\; \sum_{k=0}^{\infty} \frac{x^k}{k!} \;=\; 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \ldots,$$

d.h. die, über die exp definiert ist, bzw. die Taylor-Reihe um den Entwicklungspunkt 0.



4.2.1 Argumentreduktion

Im Hinblick auf die Anzahl der zu berechnenden Glieder wird natürlich trotz globaler Konvergenz eine Argumentreduktion durchgeführt.

Man muss bedenken, dass es durch den großen Exponentenbereich der unterliegenden Fließkommaarithmetik bei $\exp(x)$ erst deutlich später zu Unter- bzw. Überläufen kommt als bei herkömmlichen Arithmetiken, bei Verwendung von 32-Bit-Zahlen für den Exponenten etwa bei $x = 4.9 \cdot 10^9$. Eine Argumentreduktion muss also auch in solchen Bereichen noch exakt genug implementiert sein.

Bei der Berechnung von exp (oder anderen Standardfunktionen) werden Unter- und Überläufe *nicht* selbst gesondert behandelt (etwa durch eine vorherige Argumentüberprüfung). Wenn der Typ `SafeLong` für die Exponenten verwendet wird, werden ganz automatisch entsprechende Exceptions geworfen (aus den verwendeten arithmetischen Grundoperationen heraus) – die exp-Routine fängt diese ab und wirft dann eine eigene, damit die Herkunft des Überlaufs für den Benutzer klar wird. Solche Exceptions können allerdings auch einmal durch *zwischenzeitliche* Überschreitungen des Exponentenbereichs ausgelöst werden, selbst wenn das Endergebnis wieder darstellbar wäre (was hier speziell bei exp aber nicht auftritt).

a) Bei negativen Argumenten $x < 0$ haben die Terme in der Potenzreihe alternierendes Vorzeichen, was wegen möglicher Auslöschung numerisch ungünstig ist. Daher wird für

negative Argumente die Beziehung

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

angewandt.

b) Sei also nun $x \geq 0$. Mit folgender Beziehung kann das Argument auf das Intervall $[0, \ln b[$ reduziert werden (wenn es nicht schon darin liegt):

$$\begin{aligned}\exp(x) &= \exp(\ln b \cdot \lfloor x/\ln b \rfloor + x - \ln b \cdot \lfloor x/\ln b \rfloor) \\ &= b^{\lfloor x/\ln b \rfloor} \cdot \exp(x - \ln b \cdot \lfloor x/\ln b \rfloor).\end{aligned}$$

Dabei ist die Multiplikation mit $b^{\lfloor x/\ln b \rfloor}$ eine reine Exponentenverschiebung, also fehlerfrei durchführbar. Hier ist es, wo in unserer Implementation die Konstante $\ln 10$ mit beliebiger Genauigkeit benötigt wird.

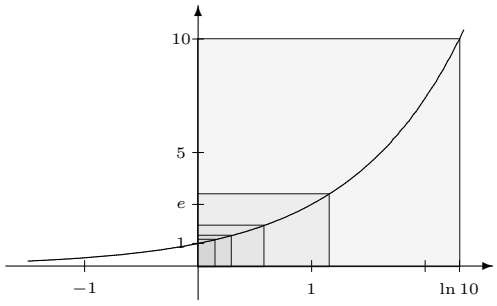
c) Nach der letzten Reduktion ist $x \in [0, \ln b[$, für den implementierten Fall $b = 10$ also in $[0, 2.30259]$. Leider werden für große x immer noch sehr viele Glieder der Potenzreihe benötigt (beispielsweise für $x = 2.3$ und $k = 16$ Stellen 23 Glieder).

Daher wird ggf. mehrere Male Gebrauch von folgender Beziehung gemacht:

$$\exp(x) = (\exp(\tfrac{x}{2}))^2.$$

wobei die Division durch 2 fehlerfrei durchführbar ist. Nach maximal viermaliger Anwendung erhalten wir $x \in [0, \frac{\ln 10}{16}[\subseteq [0, 0.14391157]$. Für $x = 0.1439$ und $k = 16$ Stellen werden beispielsweise noch 10 Glieder der Reihe benötigt.

Bei sehr hohen Stellenzahlen ist es manchmal (rechnerabhängig und abhängig von der Stellenzahl des Arguments) günstiger, nur drei Reduktionen durchzuführen, auf einigen Rechnern sind fünf günstiger.



Das hängt unter anderem von der Geschwindigkeit der Langzahl-Quadrierung ab und vom Verhältnis der Multiplikation mit einer BCD-Ziffer zu einer echten Langzahl-Multiplikation. Die vorliegende Implementation verwendet wie die zu [Steins] immer vier Reduktionen, was durchschnittlich einen guten Wert darstellt.

Links sind die bei uns verwendeten vier Bereiche dieser Reduktion dargestellt.

4.2.1.1 Fehleranalyse der Argumentreduktion

Wir müssen uns mit der Fortpflanzung der Fehler aus der Argumentreduktion in die eigentliche Reihenauswertung und aus ihr in die Rücktransformation beschäftigen.

Die einzelnen Reduktionen sind in der Implementation durch Hilfsroutinen gestaltet, die einander ggf. mit jeweils erhöhter Stellenforderung aufrufen („a) ruft b) ruft c) ruft die Reihenauswertung“).

a) Der Reduktionsschritt vor der Auswertung ($x \mapsto -x$) geschieht natürlich fehlerfrei. Es sei $y = \exp(-x)$ und $\tilde{y} = y(1+\varepsilon_y)$ das verfälschte Ergebnis. Die Inversion ist maximal genau implementiert; daher reicht es für p-stellige Hochgenauigkeit insgesamt, wenn sie p-stellig durchgeführt wird und der relative Fehler der exakten Inversion $1/\tilde{y}$ betragsmäßig kleiner als b^{-p} ist, d.h.

$$\left|\frac{\frac{1}{\tilde{y}} - \frac{1}{y}}{\frac{1}{y}}\right| \;=\; \left|\frac{y}{\tilde{y}} - 1\right| \;=\; \left|\frac{1}{1+\varepsilon_y} - 1\right| \;=\; \left|\frac{\varepsilon_y}{1+\varepsilon_y}\right| \;\overset{!}{\leq}\; b^{-p}.$$

Eine ℓ-stellige hoch genaue Auswertung garantiert nun $|\varepsilon_y| \leq 2b^{-\ell+1}$, d.h. als Kette hinreichender Forderungen ergibt sich (mit $\ell \geq 5$)

$$\begin{aligned}
\frac{2b^{-\ell+1}}{1-2b^{-\ell+1}} &\leq b^{-p}, \\
b^{\ell-p} &\geq b \cdot \frac{2}{1-2b^{-4}}, \\
\ell - p &\geq 1 + \log_{10}\left(\frac{2}{1-2b^{-4}}\right),
\end{aligned}$$

bzw. bei $b = 10$ die Forderung $\ell - p \geq 1.302$, d.h. wir benötigen zwei Schutzziffern, $\ell := p + 2$.

b) Die Reduktion b) ist nur für $x \geq \ln b$ durchzuführen. Da dies aber im Programm zunächst nur approximativ (mit einer Konstante mit wenigen Stellen) getestet wird, kann es sein, dass es sich *sicher* erst im Verlauf dieses Schritts ergibt. (Wenn für $\lfloor x/\ln b \rfloor$ der Wert 0 berechnet wird, wird der Rest des Schritts übersprungen.)

Das reduzierte Argument heiße $x_1 := x - \ln b \cdot \lfloor x/\ln b \rfloor$. Wenn die Reduktion und die Auswertung der Funktion hoch genau mit ℓ Stellen durchgeführt werden, liefern die Betrachtungen aus Abschnitt 4.1.3 als hinreichende Bedingung für p-stellige Hochgenauigkeit insgesamt:

$$\begin{aligned}
&2b^{-\ell+1} \cdot (\exp[x_1])_{\sup} + \exp([x_1])_{\sup} \cdot 2b^{-\ell+1} \cdot [x_1]_{\sup} \;\leq\; b^{-p} \cdot (\exp[x_1])_{\inf}, \\
b^{\ell-p} &\geq 2\,b \frac{\exp([x_1]_{\sup})}{\exp([x_1]_{\inf})}\,(1 + [x_1]_{\sup}), \\
b^{\ell-p} &\geq 2\,b\,\exp(2b^{-\ell+1}[x_1]_{\sup})\,(1 + [x_1]_{\sup}).
\end{aligned}$$

Für $b = 10$ gilt für das reduzierte Argument $[x_1] \subseteq [1, \ln 10 \cdot (1 + 2 \cdot 10^{-\ell+1})]$. Zusammen mit $\ell \geq 5$ erhalten wir die Forderung $10^{\ell-p} \geq 66.1$, also zwei Schutzziffern.

Es bleibt x_1 mit $p+2$ Stellen Genauigkeit einzuschließen. Wegen der vorkommenden Integer-Rundung kann man den maximalen Fehler *nicht a priori* abschätzen. Besonders wenn x in der Nähe von Vielfachen von $\ln b$ liegt, wird eine Einschließung problematisch.

Zunächst geht die vorliegende Implementation ähnlich wie die zu [Steins] vor. Es muss zunächst die Gauß-Klammer $f := \lfloor x/\ln b \rfloor$ sicher bestimmt werden. Dazu wird mit der Stellenzahl $\ell = p+2$ (für Logarithmus und Division) begonnen und eine Einschließung

von $x/\ln b$ berechnet. Solange dabei noch eine ganze Zahl echt eingeschlossen wird, wird die Stellenzahl um 2 erhöht und die Berechnung wiederholt.

Falls danach nun sicher $f = 0$, braucht Schritt b) nicht durchgeführt zu werden. (Das kann auftreten, wenn x nur sehr wenig kleiner ist als $\ln 10$, was am Beginn des Schritts nicht erkannt wurde.)

Ansonsten wird nun, anders als in der Implementation zu [Steins], für die eigentliche Berechnung von $[x_1]$ nicht noch eine ähnliche Schleife verwendet; die letztlich nötige Stellenzahl m für die Näherung $(\ln b)_m$ von $\ln b$ kann nun direkt bestimmt werden. Die Multiplikation kann exakt durchgeführt werden, die Subtraktion und die Berechnung von $\ln b$ sind maximal genau; daher ergibt sich folgende Kette hinreichender Bedingungen:

$$\begin{aligned} |(x - f \cdot (\ln b)_m) - (x - f \cdot \ln b)| &\leq b^{-p-2} \cdot |x - f \cdot \ln b|, \\ |\ln b - (\ln b)_m| &\leq b^{-p-2} \cdot \frac{(x - f \cdot \ln b)}{f}, \\ b^{-m+1} \cdot \ln b &\leq b^{-p-2} \cdot \frac{(x - f \cdot \ln b)}{f}, \\ m &\geq p + 3 - \log_b \left(\frac{x - f \ln b}{f \ln b} \right). \end{aligned}$$

Im Programm wird hierbei das Supremum der letzten Näherung von $\ln b$ verwendet, die Division nach unten gerundet, die Subtraktion wegen Auslöschung exakt durchgeführt und $\log_b$ durch order ersetzt. Anschließend wird $[x_1] = x - f \cdot [\ln b]_m$ berechnet.

c) Es mögen m Reduktionsschritte erfolgen, d.h. wir verwenden

$$\exp(x) = (\exp(\tfrac{x}{2^m}))^{2^m}.$$

Die Division durch Zweierpotenzen kann exakt durchgeführt werden. Eine mit ℓ Stellen hoch genaue Auswertung der Reihe garantiert einen relativen Fehler $\leq 2b^{-\ell+1}$. Wenn die (maximal 5) Quadrierungen exakt durchgeführt werden, pflanzt sich dieser zu maximal $(1 + 2b^{-\ell+1})^{2^m} - 1$ fort, was für $b = 10$ bei $m = 1, 2, 3$ zwei und für $m = 4, 5$ drei Schutzziffern bedeutet (was hier nicht ausgeführt werden soll).

Durch mehrfaches exaktes Quadrieren kann die Mantissenlänge allerdings relativ lang werden. Wenn nach allen Quadrierungen außer der letzten mit ℓ Stellen gerundet wird (abschließend kann mit p Stellen gerundet werden), erhält man für $b = 10$:

$$\begin{aligned} m = 2: \quad |\varepsilon_{ges}| &= |(1 + \varepsilon_{\exp})^4 (1 + \varepsilon_{sqr}) - 1| \leq 4.00161 \cdot |\varepsilon_{\exp}| + |\varepsilon_{sqr}| \\ &\leq 4.00161 \cdot 2 \cdot 10^{-\ell+1} + 10^{-\ell+1} = 9.00322 \cdot 10^{-\ell+1} \overset{!}{\leq} 10^{-p}, \\ m = 3: \quad |\varepsilon_{ges}| &= |(1 + \varepsilon_{\exp})^8 (1 + \varepsilon_{sqr})^6 - 1| \leq 6.00151 \cdot |\varepsilon_{sqr}| + 8.0105 \cdot |\varepsilon_{\exp}| \\ &\leq 22.02251 \cdot 10^{-\ell+1} \overset{!}{\leq} 10^{-p}, \\ m = 4: \quad |\varepsilon_{ges}| &= |(1 + \varepsilon_{\exp})^{16} (1 + \varepsilon_{sqr})^{14} - 1| \leq 14.0092 \cdot |\varepsilon_{sqr}| + 16.0467 \cdot |\varepsilon_{\exp}| \\ &\leq 46.1026 \cdot 10^{-\ell+1} \overset{!}{\leq} 10^{-p}, \end{aligned}$$

also schärfer $\ell \geq p + 1.955$, bzw. $\ell \geq p + 2.343$, bzw. $\ell \geq p + 2.6638$, d.h. für $m = 2$ werden weiterhin nur 2, bei $m = 3$ und $m = 4$ drei Schutzziffern benötigt. Bei Tests erwies es sich am schnellsten, bei $m \geq 3$ nur einmal zwischenzeitlich zu runden.

4.2.2 Auswertung der Potenzreihe

4.2.2.1 Geändertes Horner-Verfahren

Unser Approximationspolynom ist

$$P(x) = P_N(x) := \sum_{k=0}^{N} \frac{1}{k!} x^k.$$

Wie in Abschnitt 4.1.8 beschrieben, werden die Koeffizienten mit in die Horner-Klammerung einbezogen:

$$P(x) = 1 + x \cdot \left(1 + \tfrac{x}{2} \cdot \left(1 + \tfrac{x}{3} \cdot \left(1 + \tfrac{x}{4} \cdot \left(\ldots \cdot \left(1 + \tfrac{x}{N-1} \cdot \left(1 + \tfrac{x}{N}\right)\right)\ldots\right)\right)\right)\right). \tag{4.18}$$

Auch das sich ergebende Verfahren, das zwei lange Divisionen zur Ermittlung der Koeffizienten einspart, war auf Seite 67 bereits angegeben worden. Außer am Schluss wird genau bei den vorkommenden Divisionen gerundet. Die ersten *drei* Koeffizienten 1, 1, $\frac{1}{2}$ sind exakt darstellbar. Wir werden dennoch nach der Division durch 2 einmal runden, um die Stellenzahl nicht zu sehr anwachsen zu lassen (es wird aber eine schnellere, spezielle Routine verwendet). Die letzten beiden Schritte werden tatsächlich exakt durchgeführt, bis am Ende einmal ins ℓ-stellige Raster gerundet wird.

$$\begin{array}{lrll}
\text{a)} & r_N & := x/N, & \\
\text{b)} & r_k & := (x \cdot (r_{k+1} + 1))/k & \text{für } k = N-1, \ldots, 3, \\
\text{b}_2) & r_2 & := (x \cdot (r_3 + 1))/2 & \text{(wäre exakt möglich)}, \\
\text{c)} & r_1 & := x \cdot (r_2 + 1) & \text{(wird exakt durchgeführt)}, \\
\text{d)} & r_0 & := r_1 + 1 & \text{(Schlussrundung)}, \\
\Rightarrow & P(x) & = r_0, &
\end{array}$$

Es gilt für $k \geq 1$

$$r_k = \sum_{i=1}^{N+1-k} \frac{(k-1)!}{(k+i-1)!}\, x^i.$$

4.2.2.2 Fehlerabschätzungen des geänderten Verfahrens

Wir setzen die vorangegangene Argumentreduktion auf $x \in]0, x_{max}]$ voraus. Alle Zwischenergebnisse r_k und das Argument x sind daher positiv, weswegen wir beim Ausklammern vor den Fehlerintervallen die Beträge wegfallen lassen können.

Das Verfahren entspricht dem für Lemma 7, unter Wegfallen der vom gestörten Argument hervorgerufenen Terme $(1 + E_\ell)$. Genauer erhalten wir Folgendes für die absoluten Fehler $\Delta_k = \Delta r_k = \tilde{r}_k - r_k$:

a) $\tilde{r}_N \in \frac{x}{N} \cdot (1 + E_\ell) = r_N \cdot (1 + E_\ell) = r_N + r_N \cdot E_\ell$
$\Rightarrow \Delta_N \in r_N \cdot E_\ell$.

b) Für $k = N - 1, \ldots, 2$:

$$\begin{aligned}
\tilde{r}_k &= (x \cdot (\tilde{r}_{k+1} + 1)) \;/_\ell\; k \\
&\in x \cdot (r_{k+1} + \Delta_{k+1} + 1)/k \cdot (1 + E_\ell) \\
&= (\; r_k + x\Delta_{k+1}/k \;) \cdot (1 + E_\ell) \\
&\subseteq r_k + r_k E_\ell + x\Delta_{k+1}/k(1 + E_\ell) \\
\Rightarrow \quad \Delta_k &\in r_k E_\ell + \tfrac{x}{k}(1 + E_\ell)\Delta_{k+1}.
\end{aligned}$$

c) Hier findet keine Rundung statt.

$$\begin{aligned}
\tilde{r}_1 &= x(r_2 + 1 + \Delta_2) \\
&= r_1 + x\Delta_2 \\
\Rightarrow \quad \Delta_1 &= x\Delta_2.
\end{aligned}$$

d) Am Schluss dieses Schrittes wird in das ℓ-stellige Raster gerundet.

$$\begin{aligned}
\tilde{r}_0 &\in (r_1 + 1 + \Delta_1)(1 + E_\ell) \\
&= (r_0 + \Delta_1)(1 + E_\ell) \\
&= r_0 + r_0 E_\ell + (1 + E_\ell)\Delta_1 \\
\Rightarrow \quad \Delta_P = \Delta_0 &\in r_0 E_\ell + (1 + E_\ell)\Delta_1
\end{aligned}$$

Durch sukzessives Einsetzen (bzw. Induktion wie in Lemma 3) erhält man:

$$\begin{aligned}
\Delta_2 &\in r_2 E_\ell + \frac{x}{2}(1 + E_\ell)\Delta_3 \\
&\subseteq r_2 E_\ell + \frac{x}{2}(1 + E_\ell)\left(r_3 E_\ell + \frac{x}{3}(1 + E_\ell)\Delta_4\right) \\
&\subseteq r_2 E_\ell + \frac{x}{2} r_3 E_\ell (1 + E_\ell) + \frac{x^2}{3!}(1 + E_\ell)^2 \Delta_4,
\end{aligned}$$

bzw. insgesamt

$$\Delta_2 \in E_\ell \sum_{k=1}^{N-1} r_{k+1} \frac{x^{k-1}}{k!}(1 + E_\ell)^{k-1}.$$

Für den Endfehler ergibt sich also

$$\begin{aligned}
\Delta_P &\in r_0 E_\ell + (1 + E_\ell)\Delta_1 \quad \subseteq \quad r_0 E_\ell + x(1 + E_\ell)\Delta_2 \\
&\subseteq r_0 E_\ell + E_\ell \sum_{k=1}^{N-1} r_{k+1} \frac{x^k}{k!}(1 + E_\ell)^k \\
&= E_\ell \left[r_0 + \sum_{k=1}^{N-1} r_{k+1} \frac{x^k}{k!}(1 + E_\ell)^k \right]
\end{aligned}$$

Nun verwenden wir die Abschätzung $(1+E_\ell)^n \subseteq (1+(n+1)E_\ell)$ aus Lemma 1 (die Voraussetzungen müssen in der Implementation getestet werden, vergleiche Seite 58, Punkt 6):

$$\begin{aligned}
\Delta_P &\in E_\ell\left[r_0 + r_2 x(1+E_\ell) + \sum_{k=2}^{N-1} r_{k+1}\frac{x^k}{k!}(1+E_\ell)^k \right] \\
&\subseteq E_\ell\left[r_0 + r_2 x(1+E_\ell) + \sum_{k=2}^{N-1} r_{k+1}\frac{x^k}{k!}(1+(k+1)E_\ell) \right] \\
&\subseteq E_\ell\left[r_0 + r_2 x(1+E_\ell) + \sum_{k=2}^{N-1} r_{k+1}\frac{x^k}{k!} + E_\ell\sum_{k=2}^{N-1} r_{k+1}\frac{x^k}{k!}(k+1) \right] \\
&= E_\ell\left[r_0 + r_2 x E_\ell + \sum_{k=1}^{N-1} r_{k+1}\frac{x^k}{k!} + E_\ell\sum_{k=2}^{N-1} r_{k+1}\frac{x^k}{k!}(k+1) \right] \\
&\subseteq E_\ell\left[r_0 + (1+E_\ell)\underbrace{\sum_{k=1}^{N-1} r_{k+1}\frac{x^k}{k!}}_{A} + E_\ell\underbrace{\sum_{k=2}^{N-1} r_{k+1}\frac{x^k}{(k-1)!}}_{B} \right]
\end{aligned}$$

Für die Terme A und B ergibt sich durch Einsetzen der r_k und Umsortieren:

$$\begin{aligned}
A &= \sum_{k=1}^{N-1}\sum_{i=1}^{N-k}\frac{x^{k+i}}{(k+i)!} = \sum_{k=2}^{N}(k-1)\frac{x^k}{k!} = x\sum_{k=2}^{N}\frac{x^{k-1}}{(k-1)!} - \sum_{k=2}^{N}\frac{x^k}{k!} \\
&= x\sum_{k=1}^{N-1}\frac{x^k}{k!} - \sum_{k=2}^{N}\frac{x^k}{k!} = x\left(\sum_{k=0}^{N-1}\frac{x^k}{k!} - 1\right) - \sum_{k=2}^{N}\frac{x^k}{k!} \\
&\le x(e^x-1) - \frac{x^2}{2} - \frac{x^3}{6}, \\
B &= \sum_{k=2}^{N-1}\sum_{i=1}^{N-k} k\,\frac{x^{k+i}}{(k+i)!} = \sum_{k=3}^{N}\left(\frac{x^k}{k!}\sum_{i=2}^{k-1} i\right) = \sum_{k=3}^{N}\frac{x^k}{k!}\left(\frac{(k-1)k}{2} - 1\right) \\
&= \frac{1}{2}\sum_{k=3}^{N}\frac{x^k}{k!}(k-1)k - \sum_{k=3}^{N}\frac{x^k}{k!} = \frac{x^2}{2}\sum_{k=3}^{N}\frac{x^{k-2}}{(k-2)!} - \sum_{k=3}^{N}\frac{x^k}{k!} \\
&= \frac{x^2}{2}\sum_{k=1}^{N-2}\frac{x^k}{k!} - \sum_{k=3}^{N}\frac{x^k}{k!} = \frac{x^2}{2}\left(\sum_{k=0}^{N-2}\frac{x^k}{k!} - 1\right) - \sum_{k=3}^{N}\frac{x^k}{k!} \\
&\le \frac{x^2}{2}(e^x-1) - \frac{x^3}{6} - \frac{x^4}{24} = \frac{x^2}{2}\left(e^x - 1 - \frac{x}{3} - \frac{x^2}{12}\right).
\end{aligned}$$

Für $\varepsilon_P = \frac{\Delta_P}{r_0}$ schätzen wir r_0 durch die ersten drei Glieder nach unten ab (es ist ja sogar $N \ge 3$):

$$\begin{aligned}
\varepsilon_P &\in E_\ell\left[1 + \frac{A}{r_0}(1+E_\ell) + \frac{B}{r_0}E_\ell \right] \\
&\in E_\ell\left[1 \;+\; x\cdot\frac{(e^x-1)-\frac{x}{2}-\frac{x^2}{6}}{1+x+\frac{x^2}{2}}(1+E_\ell) \;+\; \frac{x}{2}\cdot\frac{e^x-1-\frac{x}{3}-\frac{x^2}{12}}{1+x+\frac{x^2}{2}}E_\ell \right]
\end{aligned}$$

Die Faktoren vor den Intervallen sind auf $]0, x_{max}]$ monoton steigend. Dann erhalten wir mit

$b = 10$, $x_{max} = \frac{\ln 10}{16} \leq 0.143912$, und $\ell \geq 5$ (also $E_\ell \subseteq [-10^{-4}, +10^{-4}]$):

$$\begin{array}{rrcl} & \varepsilon_P & \in & E_\ell \cdot (1 + 0.00989634 + 10^{-4} \cdot 0.006551) \\ & & \subseteq & 1.009897 \cdot E_\ell, \\ \text{d.h.} & |\varepsilon_P| & \leq & 1.009897 \cdot 10^{-\ell+1}, \end{array}$$

womit wir mit $c = 1.009897$ also eine Konstante für die Abschätzung (4.10) gefunden haben. Die Plausibilitätskontrolle mit dem Rechner (Einschließung der Δ_k für feste ℓ, N, siehe Seite 60) liefert in etwa 1.009876. Mit dem normalen Horner-Verfahren (und wesentlich gröberen Abschätzungen) wird in [Steins], Seite 42, die Konstante 1.4121 errechnet.

Aus (4.11)

$$\ell \geq p + 1 + \log_{10} c,$$

folgt schärfer $\ell \geq p + 1.0043$, d.h. wir benötigen zwei Schutzziffern.

4.2.2.3 Bestimmung von ℓ und N

Mit dieser Konstanten ergibt sich aus (4.12) als Schranke für den erlaubten Approximationsfehler (mit $\ell - p \geq 2$, $\ell \geq 5$):

$$|\varepsilon_{app}| \leq 10^{-p} \cdot \frac{1 - 1.009897 \cdot 10^{-2+1}}{1 + 1.009897 \cdot 10^{-5+1}},$$

also schärfer

$$|\varepsilon_{app}| \leq 8.9892 \cdot 10^{-p-1},$$

Das Restglied der Taylor-Reihe bei Abbruch nach dem N-ten Glied ist

$$|\Delta_N| \;=\; e^x - P_N(x) \;=\; \frac{f^{N+1}(\xi)}{(N+1)!}\, x^{N+1} \;=\; \frac{e^\xi}{(N+1)!}\, x^{N+1} \;<\; \frac{e^x}{(N+1)!}\, x^{N+1},$$

mit einem $\xi \in]0, x[$, d.h. für den relativen Approximationsfehler gilt

$$|\varepsilon_{app}| < \frac{x^{N+1}}{(N+1)!}.$$

Wir suchen demnach das kleinste N, so dass

$$\frac{x^{N+1}}{(N+1)!} \;\leq\; 8.9892 \cdot 10^{-p-1}.$$

Wie in Abschnitt 4.1.10, Seite 72 beschrieben, wird ein Startwert für N mittels Interpolation aus einer Tabelle gewonnen, Potenz und Fakultät mit den entsprechenden schnellen Routinen berechnet und die obige Bedingung getestet. Solange sie nicht erfüllt ist, wird N sukzessive um 1 erhöht, und weitere Fakultäten und Potenzen werden durch Aufmultiplikation erhalten. Die Tabelle für exp ist folgende:

k	$j=1$	2	3	4	5	6	7	8	9	10
3	3	3	3	3	3	3	3	3	3	3
10	4	5	5	5	6	6	6	6	7	7
33	12	14	15	15	16	17	17	18	18	18
100	33	36	39	40	42	43	44	45	46	47
330	96	105	110	114	118	121	123	126	128	130
1000	261	280	293	303	311	318	324	330	335	339

Wenn man statt vier Argumentreduktionen c) drei oder fünf durchführt (was gelegentlich schneller sein kann, siehe weiter vorn), müssen die beiden Konstanten bei $|\varepsilon_P|$ und $|\varepsilon_{app}|$ natürlich neu berechnet werden!

Nun haben wir alle Parameter für die eigentliche Auswertung zusammen. Da alle Reihenglieder positiv sind, also $r_0 < e^x$, berechnen wir also mit gerichteten Rundungen nur eine Untergrenze und aus den Fehlerabschätzungen eine Obergrenze (wie auf Seite 59 in Punkt 7 beschrieben).

4.2.3 Auswertung für betragskleine Argumente

Für sehr kleine positive x gilt $e^x \stackrel{>}{\approx} 1$. Das kann nicht nur zur Beschleunigung ausgenutzt werden, sondern ist auch aus folgendem Grund sinnvoll (vgl. die Bemerkung auf Seite 69):

Die Horner-Auswertung setzt die exakte Durchführung der letzten beiden Schritte voraus. Bei sehr kleinen Argumenten x werden dort die Terme r_2 und besonders r_1 sehr klein, und die Summen $1+r_1$ bzw. $1+r_2$ haben dann eine sehr große Mantissenlänge, was völlig unnötig Zeit und Speicher verbraucht. Eigentlich würde bereits das erste Glied der Potenzreihe ausreichen (also 1), die Abschätzungen sind aber nur für $N \geq 3$ verwendbar.

Es muss natürlich sichergestellt sein, dass der entstehende Fehler b^{-p} nicht überschreitet. Der entsprechende Test soll möglichst schnell durchführbar sein (da er im Normalfall selten ansprechen wird) und darf daher auch etwas grob sein. Es gilt (für $0 < x \leq 10^{-3}$):

$$\begin{aligned} e^x - 1 &= x + \frac{x^2}{2} + \frac{x^3}{3!} + \ldots \leq x\left(1 + \frac{x}{2}\left(1 + x + x^2 + x^3 + \ldots\right)\right) \\ &= x\left(1 + \frac{x}{2} \cdot \frac{1}{1-x}\right) \leq 1.00051\, x \stackrel{!}{\leq} b^{-p+1}. \end{aligned}$$

Das ist (bei $b = 10$) als Vergleich $x \leq 9.994 \cdot 10^{-p}$ (mit statisch definierter Konstanter) so in der Implementation durchführbar. Momentan wird allerdings schneller, aber wesentlich gröber $\text{order}(x) \leq -p - 1$ getestet, und wir werden bei den anderen Standardfunktionen auch ähnliche Schnellvarianten mit angeben.

Bei positiv ausfallendem Test erhalten wir hier als Einschließung $[1\,,\, 1 + b^{-p+1}]$ (p-stellige Obergrenze).

4.2.4 Die Implementation

Nur dieses Mal wollen wir auch die relevanten Ausschnitte aus dem Quelltext der Implementation (leicht editiert) abdrucken. Wir beginnen mit der eigentlichen Horner-Auswertung bei schon vollständig reduziertem Argument.

Die folgenden Zeilen behandeln die gerade beschriebenen sehr kleinen Argumente:

```
BFInterval I_exp_reduced(const BigFloat &x, long p)
 {
   if ( order(x) <= -p-1 )
      return BFInterval(1,1+BigFloat(1,-p+1),true);
```

Zunächst wird die Anzahl der Potenzreihenglieder bestimmt, da sie (wegen der Überprüfung der Voraussetzung von Lemma 2) auch in die Bestimmung der Stellenzahl ℓ eingeht:

```
long N=Ntable_lookup( x, p, exp_switch_const, Ntable, 6, 10);

BigFloat left = approx_pow_up( x, N );
static BigInt expAppErrConst("89892");
BigFloat right(expAppErrConst,-p-5);
BigInt fac = BIfac(N);

while ( left > right*fac )
 {
   ++N;
   left.mul_eq_up(x,p);
   fac *= N;
 }

--N;
```

Anders als die Implementation zu [Steins] vermeiden wir hier Divisionen.

Wir sammeln die Bestandteile der Schranke für den relativen Fehler, zunächst die für den Approximationsfehler, $x^{N+1}/(N+1)!$, die sich wie folgt ergibt:

```
BigFloat eps = div_up( left, fac, k );
```

Nun wird die Stellenzahl bestimmt:

```
long l=p+2;
long l2=N+1;
l2=2+ndigs((1+l2*l2)>>1);
if (l<l2) l=l2;
```

Mit ℓ besitzen wir nun eine Schranke für den Approximationsfehler $|\varepsilon_P| \leq 1.009897 \cdot 10^{-\ell+1}$. Er wird hier direkt zur Schranke für den relativen Gesamtfehler verrechnet.

```
static BigInt exp_epsAppConst("10099");
eps += BigFloat(exp_epsAppConst,-l-3) * add_up(BigFloat::one,eps,l);
```

Es folgt das eigentliche Horner-Verfahren:

```
BigFloat s = div_down( x, N, l);

for (long n=N-1; n>=2; --n)
 {
   ++s;
   s *= x;
   s.div_eq_down( n, l );
 }
++s;
s *= x;
++s;
```

Als letztes wird die Obergrenze des Intervalls bestimmt und das Ergebnisintervall auf die gewünschten k Stellen gerundet.

```
  (eps^=(l+1)).roundup_1();
  return BFInterval(s,s*add_up(BigFloat::one,eps,l),true);
}
```

Die äußerste von außen zugängliche Routine ist die folgende, die ggf. die Argumentreduktion a) vornimmt (das Abfangen von Exceptions ist ausgelassen):

```
BFInterval exp(const BigFloat &bf, long k)
{
  if (bf.is_zero()) return BFInterval::one;
  if (k<1) k=1;
  long p=k;
  if (p<3) p=3;
  if (bf.is_negative()) return exp_positive(-bf,p+2).inv(k);
  return exp_positive(bf,p).do_round(k);
}
```

Positive Argumente übernimmt die nächste Routine, die die Argumentreduktion b) auf $[0, \frac{\ln 10}{16}]$ durchführt:

```
BFInterval I_exp_positive(const BigFloat &bf, long k)
{
  static BigFloat ln10limit("2.3025850");
  if ( bf <= ln10limit) return I_exp_reduced1(bf,k);

  BFInterval lo; // inclusion of ln10
  BigFloat fl;   // floor ( x / ln10 )
  long l=k;      // number of digits necessary for ln10
  BFInterval X(bf);

  for (;;)
   {
     lo = BFInterval::ln10(l);
     BFInterval di = div( X, lo, l );
     fl = floor( di.inf() );
     if ( fl == floor(di.sup()) ) break;
     l += 2;
   }

  if (fl.is_zero())
     return exp_reduced1( bf, k+2 );

  l = k + 3 - order( div_down( bf, fl*lo.sup(), l ) - 1 );
  BFInterval ret = I_exp_reduced1( bf - BigFloat::ln10(l)*fl , k+2 );
  long f=fl.long_value();
  return BFInterval( ret.inf()<<f, ret.sup()<<f ).do_round(k);
}
```

Die Argumentreduktion unterhalb von $\ln 10$ wird von der folgenden Routine durchgeführt. Die Division durch 2^m erledigt man schneller durch Multiplikation mit 5^m und einer Exponentenverschiebung um m nach unten:

```
static BFInterval exp_reduced1(const BigFloat &x, long k)
{
```

```
  static BigFloat limit[]=
   { "0.14391156", "0.28782313", "0.57564627", "1.15129254" };
  if (x<=limit[2])
     if (x<=limit[1])
        if (x<=limit[0])
           return exp_reduced(x,k);
        else
           return sqr(exp_reduced((5*x)>>1,k+2));
     else
        return sqr(sqr(exp_reduced((25*x)>>2,k+2)));
  else
     if (x<=limit[3])
        return sqr(sqr(sqr(exp_reduced((125*x)>>3,k+3))).do_round(k+3));
     else
      return sqr(sqr(sqr(sqr(exp_reduced((625*x)>>4,k+3)))).do_round(k+3)));
}
```

4.2.5 Intervallversion

Da die Exponentialfunktion streng monoton steigend ist, ist die intervallmäßige Implementation denkbar einfach. Mit der Version aus den letzten Abschnitten werden die Funktionswerte an den Intervallgrenzen eingeschlossen und von der ersten Einschließung das Infimum, von der zweiten das Supremum verwendet:

```
BFInterval exp(const BFInterval &x, long k)
 {
   if (bfi.is_point()) return exp(x.inf(),k);

   return BFInterval( exp(x.inf(),k).inf(),
                      exp(x.sup(),k).sup(), true);
 }
```

4.2.6 Komplexe Versionen

Die komplexe Exponentialfunktion kann einfach auf reelle Standardfunktionen zurückgeführt werden. Mit $z = x + iy$ gilt $\mathrm{Re}(e^z) = e^x \cos(y)$ und $\mathrm{Im}(e^z) = e^x \sin(y)$. Für komplexe Argumente können also die Versionen der beteiligten Funktionen für reelle Argumente verwendet werden. Es kommen x und y jeweils nur einmal vor (die Exponentialfunktion ist separabel, siehe Seite 42), d.h. es ergibt sich keine Überschätzung, wenn die Version für komplexe Intervalle durch eine Intervall-Auswertung der obigen Ausdrücke implementiert wird.

Die Fehlerabschätzungen sind für Real- und Imaginärteil gleich. Die Multiplikation wird abschließend p-stellig gerundet. Mit $r := \mathrm{Re}(e^z)$ gilt:

$$\begin{aligned}\left|\frac{\tilde{r}-r}{r}\right| &= \left|\frac{\exp(x)(1+\varepsilon_{\exp})\cos(y)(1+\varepsilon_{\cos}) - \exp(x)\cos(x)}{\exp(x)\cos(y)}\right| \\ &= |(1+\varepsilon_{\exp})(1+\varepsilon_{\cos}) - 1| \;\leq\; |\varepsilon_{\exp}|(1+\varepsilon_{\cos}) + |\varepsilon_{\cos}| \quad \overset{!}{\leq}\; b^{-p}.\end{aligned}$$

Wenn exp und cos jeweils mit m Stellen berechnet werden, ist schärfer zu erfüllen

$$2 \cdot b^{-m+1}(1 + 2 \cdot b^{-4}) + 2 \cdot b^{-m+1} \overset{!}{\leq} b^{-p},$$

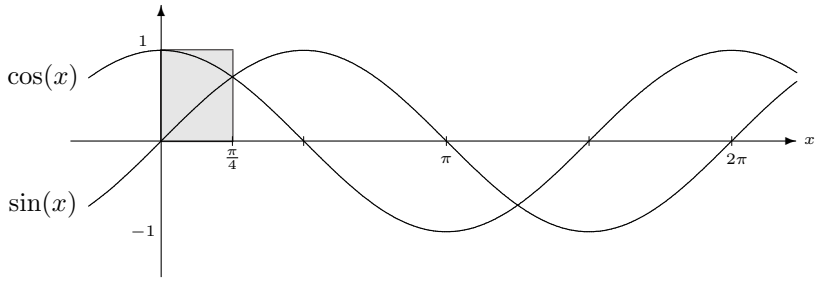
was bei $b = 10$ auf die Forderung $m \geq p + 1.6022$, also zwei Schutzziffern führt.

Bei der Implementation kann günstig die Funktion `sincos` aus dem nächsten Abschnitt verwendet werden, die sin und cos zugleich berechnet.

4.3 Sinus und Cosinus

Für diese beiden Funktionen verwenden wir die folgenden beiden Potenzreihen, die wahlweise durch Einsetzen von ix in die Reihe von $\exp(x)$ und Aufspaltung in Realteil und Imaginärteil oder durch Taylor-Entwicklung um 0 entstehen:

$$\begin{aligned}
\sin(x) &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + - \ldots &&= \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}, \\
\cos(x) &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + - \ldots &&= \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k}}{(2k)!}.
\end{aligned}$$



Die Reihen konvergieren ebenfalls global; der Bereich, in dem sie direkt Verwendung finden, ist im Bild grau unterlegt.

Wir wollen, zur späteren Verwendung an diversen Stellen der Fehlerabschätzungen, hier kurz festhalten, dass für $x > 0$ und *gerades* $n \geq 2$ folgende Einschließungen gelten:

$$\begin{aligned}
\sum_{k=0}^{n-1} (-1)^k \frac{x^{2k+1}}{(2k+1)!} \;&<\; \sin x \;<\; \sum_{k=0}^{n-2} (-1)^k \frac{x^{2k+1}}{(2k+1)!}, \\
\sum_{k=0}^{n-1} (-1)^k \frac{x^{2k}}{(2k)!} \;&<\; \cos x \;<\; \sum_{k=0}^{n} (-1)^k \frac{x^{2k}}{(2k)!},
\end{aligned}$$

was beispielsweise an den Restgliedern (siehe die Seiten 92, 95) abgelesen werden kann (oder für kleine x am alternierenden Vorzeichen und der betragsmäßigen fallenden Monotonizität). Insbesondere haben wir

$$1 - \frac{x^2}{2} \;<\; \cos x \;<\; 1 - \frac{x^2}{2} + \frac{x^4}{24}, \qquad x - \frac{x^3}{6} \;<\; \sin x \;<\; x.$$

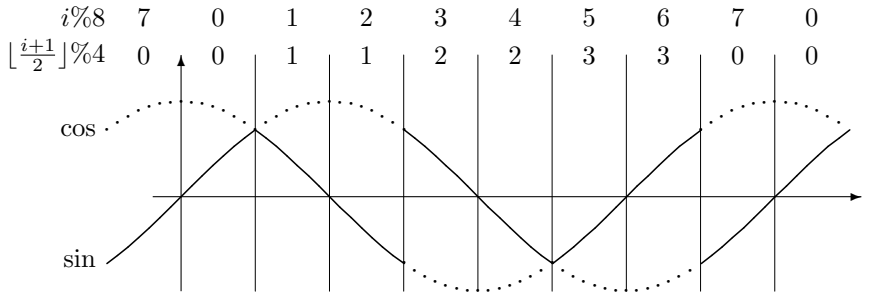
Analoges werden wir bei anderen Standardfunktionen benutzen.

4.3.1 Argumentreduktion

Wir werden keine Beschränkung des Arguments implementieren, da die unterliegende Arithmetik (theoretisch) keiner Genauigkeitsbeschränkung unterliegt. Es kann also durchaus sein, dass auch ein extrem großes x tatsächlich so genau aus einer anderen Rechnung hervorgegangen ist, dass eine Auswertung von sin und cos in ihm Sinn macht. Es liegt so allerdings in der Verantwortung des Benutzers unserer Routinen, dies zu beachten.

Es wird ggf. die Potenzreihe der jeweils anderen Funktion verwendet. Wir werden die Periodizität der Funktionen mit der Periode 2π, die Verschiebung $\cos x = \sin(x + \frac{\pi}{2})$ und die Symmetrien $\sin(-x) = -\sin(x)$, $\cos(-x) = \cos(x)$ ausnutzen. Dadurch braucht man die Funktionen nur auf dem Intervall $[-\frac{\pi}{4}, \frac{\pi}{4}]$ (alternativ auch $[0, \frac{\pi}{4}]$) zu betrachten. Die Transformation ist die Subtraktion eines ganzzahligen Vielfachen von $\frac{\pi}{2}$.

Wir betrachten die Funktionen in einem Raster der Breite $\frac{\pi}{4}$. Es sei x das Argument, in dem die Funktionen ausgewertet werden sollen, und $i := \lfloor \frac{x}{\pi/4} \rfloor$. Es ist „%“ die aus `C` entlehnte Abkürzung für den Modulo-Operator (Rest bei Division).



Offenbar sollte man im Bereich durchgezogener Linien die Sinus-Reihe, im gepunkteten Bereich die Cosinus-Reihe verwenden. Mit $j := \lfloor \frac{i+1}{2} \rfloor$ und

$$x_1 := x - j \cdot \tfrac{\pi}{2} \ \in \ [-\tfrac{\pi}{4}, +\tfrac{\pi}{4}]$$

ergibt sich die Auswahl der Reihe aus folgender Tabelle (vgl. [Steins], S. 44, bzw. [Braune], S. 32):

$j = 4n$	$x_1 = x - 2\pi n$	$\sin(x) = \sin(x_1)$	$\cos(x) = \cos(x_1)$
$j = 4n+1$	$x_1 = x - 2\pi n - \frac{\pi}{2}$	$\sin(x) = \sin(x_1 + \frac{\pi}{2}) = \cos(x_1)$	$\cos(x) = \cos(x_1 + \frac{\pi}{2}) = -\sin(x_1)$
$j = 4n+2$	$x_1 = x - 2\pi n - \pi$	$\sin(x) = \sin(x_1 + \pi) = -\sin(x_1)$	$\cos(x) = \cos(x_1 + \pi) = -\cos(x_1)$
$j = 4n+3$	$x_1 = x - 2\pi n - \frac{3\pi}{2}$	$\sin(x) = \sin(x_1 + \frac{3\pi}{2}) = -\cos(x_1)$	$\cos(x) = \cos(x_1 + \frac{3\pi}{2}) = \sin(x_1)$

Man beachte, dass wir nach dieser Reduktion noch negative Argumente x_1 zulassen.

Darüber hinausgehend könnte man folgende Identität benutzen, um das Argument der Cosinus-Reihe noch zu halbieren:

$$\cos 2x = 2\cos^2 x - 1.$$

Die entsprechende Beziehung für den Sinus ($\sin 2x = \sin x \cos x$) ist so leider nicht verwendbar.

Es stellt sich aber heraus, dass eine Reduktion unter $\frac{\pi}{4}$ die Potenzreihenauswertung nicht genügend beschleunigen würde. Durch eine Argumenthalbierung spart man etwa 2 Reihenglieder ein, der Aufwand der Transformationen entspricht etwa dem eines Horner-Schritts, aber man handelt sich zwei Schutzziffern ein.

4.3.1.1 Fehleranalyse der Argumentreduktion

Wir verwenden (4.4), was den maximal erlaubten relativen Fehler (bei p insgesamt gewünschten gültigen Stellen) auf die Funktionsauswertung $\tilde{f}(\tilde{x})$ und auf die Störung des Arguments $\tilde{x}$ gleichmäßig verteilt (der Index 1 des Reduzierten x_1 ist der Übersichtlichkeit halber unterdrückt):

$$|\varepsilon_f| = \left|\frac{\Delta f(\tilde{x})}{f(\tilde{x})}\right| \leq \frac{b^{-p}}{2}\left|\frac{f(x)}{f(\tilde{x})}\right|, \qquad |\varepsilon_x| \leq \left|\frac{\Delta x}{x}\right| \leq \frac{b^{-p}}{2}\left|\frac{f(x)}{x\cdot f'(\xi)}\right|$$

mit einem $\xi \in \mathrm{conv}\{x, \tilde{x}\}$. Es ist immer $-\frac{\pi}{4} \leq x \leq \frac{\pi}{4}$ vorausgesetzt.

a) Beim Sinus erhält man hieraus

$$|\varepsilon_{\sin}| \overset{!}{\leq} \frac{b^{-p}}{2}\cdot\left|\frac{\sin(x)}{\sin(\tilde{x})}\right|, \qquad \text{wobei}$$

$$\begin{aligned}\left|\frac{\sin(x)}{\sin(\tilde{x})}\right| &= \left|\frac{\sin(x)}{\sin(x+\Delta x)}\right| = \left|\frac{\sin(x)}{\sin(x)\cos(\Delta x)+\cos(x)\sin(\Delta x)}\right| \geq \left|\frac{\sin x}{\sin x + \Delta x}\right| \\ &\geq \frac{1}{1+|\varepsilon_x|\cdot\frac{x}{\sin x}}.\end{aligned}$$

Mit unserer Generalvoraussetzung $\ell \geq 5$, also $|\varepsilon_x| \leq 10^{-4}$, ergibt sich also hinreichend:

$$2\cdot b^{-\ell+1} \overset{!}{\leq} \frac{b^{-p}}{2}\cdot\frac{1}{1+2b^{-4}\frac{\pi}{2\sqrt{2}}},$$

bzw. $\ell \geq p+1.603$; wir brauchen also wieder einmal 2 Schutzziffern für die Auswertung der Potenzreihe.

Aus

$$|\varepsilon_x| \leq \frac{b^{-p}}{2}\left|\frac{\sin(x)}{x\cdot\cos(\xi)}\right|, \qquad \left|\frac{\sin x}{x\cdot\cos\xi}\right| \geq \left|\frac{\sin x}{x}\right| \geq \frac{2\sqrt{2}}{\pi} \gtrapprox 0.9$$

erhalten wir unsere zweite Forderung $2\cdot 10^{-\ell+1} \leq 0.45\cdot 10^{p}$, bzw. $\ell \geq p+1.648$ – auch zwei Schutzziffern für die Berechnung des Arguments.

b) Im Fall des Cosinus soll gelten

$$|\varepsilon_{\cos}| \leq \frac{b^{-p}}{2}\cdot\left|\frac{\cos x}{\cos\tilde{x}}\right|, \qquad \text{wobei}$$

$$\begin{aligned}\left|\frac{\cos(x)}{\cos(\tilde{x})}\right| &= \frac{\cos(x)}{\cos(x+\Delta x)} = \frac{\cos(x)}{\cos(x)\cos(\Delta x)-\sin(x)\sin(\Delta x)} \geq \frac{\cos(x)}{\cos(x)+|\sin(x)\,\varepsilon_x\,x|} \\ &= \frac{1}{1+|\tan(x)\,\varepsilon_x\,x|},\end{aligned}$$

d.h. hier ist zu fordern

$$2\cdot b^{-\ell+1} \leq \frac{b^{-p}}{2}\cdot\frac{1}{1+2b^{-4}\cdot\frac{\pi}{4}},$$

bzw. $2b^{-\ell+1} \leq 0.4999 \cdot 10^{-p}$, und wieder einmal müssen wir wegen $\ell \geq p + 1.6022$ mindestens zwei Schutzziffern bei der Auswertung der Reihe fordern.

Mit

$$|\varepsilon_x| \leq \frac{b^{-p}}{2} \left| \frac{\cos x}{x \sin \xi} \right|, \qquad \text{wobei}$$

$$\begin{aligned} \left| \frac{\cos x}{x \sin \xi} \right| &\geq \frac{\cos x}{|x \cdot (x + \Delta x)|} \geq \frac{\cos x}{|x||x + \varepsilon_x x|} \geq \frac{\cos x}{|x|(|x| + |\varepsilon_x x|)} = \frac{\cos x}{x^2(1 + |\varepsilon_x|)} \\ &\geq \frac{8\sqrt{2}}{\pi^2(1 + 2b^{-4})} > 1.146 \end{aligned}$$

kommen wir zur Forderung $2 \cdot 10^{-\ell+1} \leq 0.573 \cdot 10^{-p}$, bzw. $\ell \geq p + 1.543$, also auch zwei Schutzziffern für die Auswertung der Argumentreduktion.

Die Durchführung der Argumentreduktion

$$i := \lfloor \tfrac{4x}{\pi} \rfloor, \qquad j := \lfloor \tfrac{i+1}{2} \rfloor, \qquad x_1 := x - j \cdot \tfrac{\pi}{2}$$

erfolgt ähnlich wie bei der Exponentialfunktion. Wiederum kann man den Fehler bei der vorkommenden Integer-Rundung für i nicht a priori abschätzen. Es wird also sukzessive die Stellenzahl m der Einschließung von π (und die der Division) erhöht, bis $[\frac{4x}{(\pi)_m}]$ keine ganze Zahl mehr einschließt.

Es ist nicht sinnvoll, die Stellenzahl hier von Anfang an zu groß zu machen, da Probleme ja nur eng um Vielfache von $\frac{\pi}{4}$ herum auftreten. Falls x so ungünstig liegt, ist ein guter Schätzwert $m_0 = (\text{Mantissenlänge von } 4x) - (\text{Charakteristik von } 4x)$. Wir starten daher mit $m = p + 2$, bei einer Erhöhung setzen wir m auf $2 + \frac{m+m_0}{2}$, oberhalb von m_0 auf $m + 2$.

Wir berechnen nun eine Einschließung $[x_1] := (x - j \cdot \frac{[\pi]_m}{2})$. Falls dabei 0 eingeschlossen werden sollte, wird die Rechnung mit einem um 2 erhöhten m wiederholt. Die Stellenzahl m, die notwendig ist, damit $[x_1]$ letztlich $p + 2$-stellige Genauigkeit erhält, ergibt sich aus folgender Betrachtung:

$$\begin{aligned} |\tilde{x}_1 - x_1| &\leq b^{-p-2} |x_1|, \\ |(x - \tfrac{j}{2}(\pi)_m) - (x - \tfrac{j}{2}\pi)| &\leq b^{-p-2} |x_1|, \\ |\tfrac{j}{2}(\pi - (\pi)_m)| &\leq b^{-p-2} |x_1|, \\ \tfrac{j}{2} \cdot b^{-m+1} \cdot \pi &\leq b^{-p-2} |x_1|, \\ m &\geq p + 3 - \log_{10} |\tfrac{2x_1}{j\pi}|, \end{aligned}$$

und wir berechnen (unter Zuhilfenahme der vorherigen Einschließungen von x_1 und π)

$$m := p + 3 - \text{order} \left(\frac{2|[x_1]|_{\text{inf}}}{j \cdot [\pi]_{\text{sup}}} \right),$$

wobei die Division nur mit *einer Stelle* nach unten gerundet durchgeführt zu werden braucht, da ja nur die Größenordnung interessant ist. Alternativ kann man den Logarithmus-Term zerlegen, wobei sich zwar typischerweise eine höhere Stellenzahl ergibt, die Division aber entfällt:

$$m := p + 4 - \text{order } (2\,|[x_1]|_{\text{inf}}) + \text{order } (j\,[\pi]_{\text{sup}}).$$

4.3.2 Auswertung beim Sinus

Wie bei der Exponentialfunktion kann die Auswertung deutlich beschleunigt werden, wenn man die inversen Fakultäten mit in die Ausklammerung des Horner-Schemas einbezieht.

Für den Sinus ist die normale Klammerung (mit $u = x^2$):

$$P_N^{\sin}(x) = x\left(1 + u\Big(-\frac{1}{3!} + u\Big(+\frac{1}{5!} + u\Big(-\frac{1}{7!} + u\Big(\dots\Big(\frac{(-1)^{N-1}}{(2N-1)!} + u\frac{(-1)^N}{(2N+1)!}\Big)\Big)\dots\Big)\right),$$

d.h. in jedem Schritt wäre eine Division durch eine Integerzahl der Form $(2j+3)(2j+2)$ erforderlich, um den neuen Nenner zu erhalten. Die alternative Klammerung ist folgende:

$$P_N^{\sin}(x) = x\left(1 - \frac{u}{2\cdot 3}\Big(1 - \frac{u}{4\cdot 5}\Big(1 - \frac{u}{6\cdot 7}\Big(1 - \dots\Big(1 - \frac{u}{(2N)(2N+1)}\Big)\Big)\dots\Big)\right).$$

Das zugehörige Auswertungsverfahren sieht so aus:

$$\begin{aligned}
&a)\quad r_N &&:= \frac{u}{(2N)(2N+1)},\\
&b)\quad r_k &&:= \frac{u}{(2k)(2k+1)}(1 - r_{k+1}) \qquad \text{für } k = N-1,\dots,1,\\
&c)\quad r_0 &&:= 1 - r_1,\\
&\Rightarrow\quad P(x) &&= x \cdot r_0,
\end{aligned}$$

d.h. für $k \geq 1$

$$r_k = (2k-1)! \sum_{i=1}^{N-k+1} \frac{(-1)^{i-1}}{(2k+2i-1)!} x^{2i}.$$

Da $P_N(x) > \sin x$ für gerades N und $P_N(x) < \sin x$ für ungerades N, berechnen wir für gerades N eine Obergrenze und für ungerades N eine Untergrenze.

4.3.3 Fehlerabschätzungen des geänderten Verfahrens beim Sinus

Das für unseren Fall passende Lemma aus Abschnitt 4.1.8 ist Korollar 9. Mit $a_0 = 1$ und für $k \geq 1$

$$a_k = \frac{1}{2k(2k+1)}, \quad \text{also} \quad \prod_{k=0}^{n} a_i = \frac{1}{(2n+1)!},$$

erhalten wir daraus

$$\begin{aligned}
\Delta_P \;\in\; E_\ell \Big[\; &|r_0 x| + |r_1 x|(1+E_\ell)(2+E_\ell) + (2+E_\ell)\sum_{k=2}^{N}\Big| r_k \frac{x^{2k-1}}{(2k-1)!}\Big| \\
&+2E_\ell(2+E_\ell)\sum_{k=2}^{N}\Big| k\, r_k \frac{x^{2k-1}}{(2k-1)!}\Big| \;\Big]
\end{aligned}$$

Bei den r_k haben wir alternierendes Vorzeichen und betragsmäßig fallende Monotonie, so dass (bei $x > 0$)

$$0 \;<\; r_k \;\leq\; \frac{x^2}{(2k)(2k+1)} = \frac{(2k-1)!}{(2k+1)!}\, x^2,$$

was in unseren Abschätzungen für $k \geq 2$ völlig ausreicht. Es ergibt sich (mit $x > 0$):

$$\Delta_P \in E_\ell \Big[\, r_0 x + r_1 x(1+E_\ell)(2+E_\ell) + (2+E_\ell) \underbrace{\sum_{k=2}^{N} \frac{x^{2k+1}}{(2k+1)!}}_{A} + 2E_\ell(2+E_\ell) \underbrace{\sum_{k=2}^{N} k\, \frac{x^{2k+1}}{(2k+1)!}}_{B} \,\Big].$$

Für die beiden Summen A und B gilt:

$$\begin{aligned}
A &= \sum_{k=2}^{N} \frac{x^{2k+1}}{(2k+1)!} = \sum_{k=0}^{N} \frac{x^{2k+1}}{(2k+1)!} - x - \frac{x^3}{6} \leq \sinh x - x - \frac{x^3}{6}, \\
B &= \sum_{k=2}^{N} k \cdot \frac{x^{2k+1}}{(2k+1)!} = \frac{1}{2} \sum_{k=2}^{N} (2k+1) \frac{x^{2k+1}}{(2k+1)!} - \frac{1}{2} \sum_{k=2}^{N} \cdot \frac{x^{2k+1}}{(2k+1)!} \\
&= \frac{x}{2} \sum_{k=2}^{N} \frac{x^{2k}}{(2k)!} - \frac{1}{2} \sum_{k=2}^{N} \cdot \frac{x^{2k+1}}{(2k+1)!} = \frac{x}{2} \left(\sum_{k=0}^{N} \frac{x^{2k}}{(2k)!} - 1 - \frac{x^2}{2} \right) - \frac{1}{2} \sum_{k=2}^{N} \cdot \frac{x^{2k+1}}{(2k+1)!} \\
&\leq \frac{x}{2} \left(\cosh x - 1 - \frac{x^2}{2} \right) - \frac{1}{2} \cdot \frac{x^5}{120} = \frac{x}{2} \left(\cosh x - 1 - \frac{x^2}{2} - \frac{x^4}{120} \right).
\end{aligned}$$

Für $\varepsilon_P = \frac{\Delta P}{r_0 x}$ ergibt sich noch

$$\frac{r_1 x}{r_0 x} = \frac{1 - r_0}{r_0} = \frac{1}{r_0} - 1.$$

r_0 im Nenner wird nach unten durch die ersten vier Glieder abgeschätzt (es ist ja $N \geq 3$ vorausgesetzt). Dann erhalten wir insgesamt

$$\begin{aligned}
\varepsilon_P &\in E_\ell \cdot \Big[1 + \frac{r_1}{r_0}(1+E_\ell)(2+E_\ell) + (2+E_\ell)\frac{A}{x r_0} + 2E_\ell(2+E_\ell)\frac{B}{x r_0} \Big] \\
&\subseteq E_\ell \cdot \Big[1 + (1+E_\ell)(2+E_\ell) \left(\frac{1}{1 - \frac{x^2}{6} + \frac{x^4}{120} - \frac{x^6}{5040}} - 1 \right) + (2+E_\ell) \frac{\frac{\sinh x}{x} - 1 - \frac{x^2}{6}}{1 - \frac{x^2}{6}} \\
&\qquad + E_\ell(2+E_\ell) \frac{\cosh x - 1 - \frac{x^2}{2} - \frac{x^4}{120}}{1 - \frac{x^2}{6}} \Big].
\end{aligned}$$

Die Koeffizienten der Fehlerintervalle sind alle auf dem betrachteten Bereich $0 < x \leq \frac{\pi}{4}$ monoton steigend. Mit $b = 10$ und $\ell \geq 5$ ergibt sich daher schließlich

$$\begin{aligned}
\varepsilon_P &\in E_\ell \Big[1 + 0.11072074(1+E_4)(2+E_4) + 0.0035865698(2+E_4) + 0.0072521202 E_4(2+E_4) \Big] \\
&\subseteq E_\ell (1 + 0.22147469 + 7.1734983 \cdot 10^{-3} + 2.9009932 \cdot 10^{-6}) \\
&\subseteq 1.228652 \cdot E_\ell.
\end{aligned}$$

Die Plausibilitätskontrolle auf dem Rechner liefert etwa einen Faktor 1.228498. Mit dem normalen Horner-Verfahren wird in [Steins] 1.6095 berechnet.

Wiederum folgt mit (4.11), dass wir zwei Schutzziffern benötigen. Als Schranke für den Approximationsfehler erhält man mit $\ell \geq k+2 \geq 5$:

$$|\varepsilon_{app}| \leq 10^{-p} \cdot \frac{1 - 1.228651 \cdot 10^{-1}}{1 + 1.228651 \cdot 10^{-4}},$$

d.h.

$$|\varepsilon_{app}| \leq 8.7703 \cdot 10^{-p-1}.$$

4.3.4 Approximationsfehler beim Sinus

Aus dem Restglied der Taylor-Entwicklung erhalten wir folgende Abschätzung für den absoluten, bzw. relativen Approximationsfehler ($x > 0$):

$$\Delta_N(x) = (-1)^{N+1} \frac{\cos(\xi)}{(2N+3)!}\, x^{2N+3},$$

mit einem $\xi \in]0, x[$, also:

$$\left|\varepsilon_{app}^N(x)\right| \;=\; \left|\frac{\Delta_N(x)}{\sin(x)}\right| \;\leq\; \frac{x^{2N+3}}{(2N+3)!\,|\sin x|} \;=\; \frac{x^{2N+2}}{(2N+3)!} \cdot \frac{x}{\sin x},$$

Wir suchen also ein kleines N mit

$$\frac{x^{2N+2}}{(2N+3)!} \cdot \frac{x}{\sin x} \;\leq\; 8.7703 \cdot 10^{-p-1},$$

bzw. mit $\sin x > x - \frac{x^3}{6}$ eines mit

$$\frac{x^{2N+2}}{(2N+3)!} \cdot \frac{6}{6 - x^2} \;\leq\; 8.7703 \cdot 10^{-p-1},$$

implementiert als

$$x^{2N+2} \;\leq\; (2N+3)! \cdot (6 - x^2) \cdot 1.4618 \cdot 10^{-p-1},$$

wobei wir wieder (wie in 4.1.10 beschrieben) einen Startwert für N aus einer Tabelle interpolieren (die Tabelle erscheint bereits im Beispiel auf Seite 73):

p | $j=1$	2	3	4	5	6	7	8	9	10
3	3	3	3	3	3	3	3	3	3
10	3	3	4	4	4	5	5	5	5	6
33	8	9	10	11	11	12	12	13	13	14
100	21	23	25	27	28	29	30	31	32	32
330	59	65	69	73	76	78	80	82	84	85
1000	157	171	181	188	194	199	204	208	212	216

4.3.5 Auswertung beim Cosinus

Statt der normalen Horner-Klammerung (mit $u = x^2$)

$$P_N(x) = 1 + u\left(-\frac{1}{2!} + u\Big(\frac{1}{4!} + u\Big(-\frac{1}{6!} + u\Big(\dots\Big(\frac{(-1)^{N-1}}{(2N-2)!} + u\frac{(-1)^N}{(2N!)}\Big)\Big)\dots\right)$$

verwenden wir folgende:

$$P_N(x) = 1 - \frac{u}{2}\left(1 - \frac{u}{3\cdot 4}\Big(1 - \frac{u}{5\cdot 6}\Big(1 - \dots\Big(1 - \frac{u}{(2N-1)(2N)}\Big)\Big)\dots\right)$$

und folgendes Auswertungsverfahren:

$$\begin{aligned}
&a) \quad r_N &&:= \frac{u}{(2N-1)(2N)},\\
&b) \quad r_k &&:= \frac{u}{(2k-1)(2k)}(1 - r_{k+1}) \qquad \text{für } k = N-1, \dots, 2,\\
&c) \quad r_1 &&:= \frac{u}{2}(1 - r_2),\\
&d) \quad r_0 &&:= 1 - r_1,\\
&\Rightarrow \quad P(x) &&= r_0.
\end{aligned}$$

Dabei ist r_1 gesondert behandelt, da die Division durch 2 exakt erfolgen kann.

Wie beim Sinus wird bei geradem N die Obergrenze, bei ungeradem N die Untergrenze des Ergebnisintervalls berechnet und die andere Grenze aus den Beziehungen zum relativen Fehler.

4.3.6 Fehlerabschätzungen des geänderten Verfahrens beim Cosinus

Für diesen Fall ist Korollar 8 anwendbar. Mit $a_0 = 1$ und für $k \geq 1$

$$a_k = \frac{1}{(2k-1)(2k)}, \quad \text{also} \quad \prod_{k=0}^{n} a_k = \frac{1}{(2n)!}$$

erhalten wir

$$\begin{aligned}
\Delta_P \in E_\ell\Big[\ |r_0| + |r_1|\,(1 + E_\ell) + (2 + E_\ell)\sum_{k=2}^{N}\Big|\frac{1}{(2k-2)!}\,r_k\,x^{2k-2}\Big|\\
+E_\ell(2 + E_\ell)\sum_{k=2}^{N}\Big|\frac{1}{(2k-2)!}\,r_k\,x^{2k-2}\Big|(2k-1)\ \Big]
\end{aligned}$$

Für $k \geq 1$ gilt für die r_k:

$$r_k = (2k-2)!\sum_{i=1}^{N-k+1}\frac{(-1)^{i-1}}{(2k+2i-2)!}\,x^{2i},$$

und durch das alternierende Vorzeichen und die betragsmäßig fallende Monotonie gilt

$$0 < r_k \;\leq\; \frac{x^2}{(2k-1)(2k)} \;=\; \frac{(2k-2)!}{(2k)!}\,x^2.$$

Damit ergibt sich

$$\Delta_P \in E_\ell\Big[\; r_0 + r_1\,(1+E_\ell) + (2+E_\ell)\underbrace{\sum_{k=2}^{N}\frac{x^{2k}}{(2k)!}}_{A} + E_\ell(2+E_\ell)\underbrace{\sum_{k=2}^{N}\frac{x^{2k}}{(2k)!}\,(2k-1)}_{B}\;\Big]$$

Wir erhalten für die Summen A und B:

$$\begin{aligned}
A &= \sum_{k=2}^{N}\frac{x^{2k}}{(2k)!} = \sum_{k=0}^{N}\frac{x^{2k}}{(2k)!} - 1 - \frac{x^2}{2} \;\leq\; \cosh x - 1 - \frac{x^2}{2},\\
B &= \sum_{k=2}^{N}\frac{x^{2k}}{(2k)!}(2k-1) \;=\; \sum_{k=2}^{N}\frac{x^{2k}}{(2k)!}2k - \sum_{k=2}^{N}\frac{x^{2k}}{(2k)!} \;=\; x\sum_{k=1}^{N}\frac{x^{2k+1}}{(2k+1)!} - \sum_{k=2}^{N}\frac{x^{2k}}{(2k)!}\\
&= x\left(\sum_{k=0}^{N}\frac{x^{2k+1}}{(2k+1)!} - x\right) - \sum_{k=2}^{N}\frac{x^{2k}}{(2k)!} \leq\; x(\sinh x - x) - \frac{x^4}{24}.
\end{aligned}$$

Mit analogen Abschätzungen für r_0 wie beim Sinus ergibt sich nun insgesamt

$$\begin{aligned}
\varepsilon_P \;&\in\; E_\ell\Big[\,1 + \frac{r_1}{r_0}\,(1+E_\ell) + (2+E_\ell)\frac{A}{r_0} + E_\ell(2+E_\ell)\frac{B}{r_0}\,\Big]\\
&\subseteq\; E_\ell\Big[\,1 + (1+E_\ell)\Big(\frac{1}{1-\frac{x^2}{2}+\frac{x^4}{24}-\frac{x^6}{720}} - 1\Big) + (2+E_\ell)\cdot\frac{\cosh x - 1 - \frac{x^2}{2}}{1-\frac{x^2}{2}}\\
&\qquad + E_\ell(2+E_\ell)\cdot\frac{x(\sinh x - x) - \frac{x^4}{24}}{1-\frac{x^2}{2}}\,\Big]
\end{aligned}$$

Die Faktoren der Fehlerintervalle sind im betrachteten Bereich $0 < x \leq \frac{\pi}{4}$ wiederum monoton steigend. Mit $b = 10$, $\ell \geq 5$ erhalten wir daher

$$\begin{aligned}
\varepsilon_P \;&\in\; E_\ell\,(1 + 0.414263 + 0.0468056 + 1.43298\cdot 10^{-5})\\
&\subseteq\; 1.46109\cdot E_\ell.
\end{aligned}$$

Die Kontrolle auf dem Rechner ergibt einen Faktor 1.46005. Für das normale Horner-Verfahren (und mit gröberen Abschätzungen) wird in [Steins] die Konstante 4.2551 berechnet (was dort *vier* Schutzziffern bei $b = 2$ bedeutet).

Nichtsdestoweniger benötigen wir wieder einmal zwei Schutzziffern.

4.3.7 Approximationsfehler beim Cosinus

Mit der gerade berechneten Konstante erhalten wir aus (4.12) die Forderung

$$|\varepsilon_{app}| \leq 10^{-p}\cdot\frac{1 - 1.46109\cdot 10^{-1}}{1 + 1.46109\cdot 10^{-4}},$$

d.h.

$$|\varepsilon_{app}| \leq 8.5377 \cdot 10^{-p-1}.$$

Aus dem Restglied der Taylor-Entwicklung ergibt sich (für $x > 0$)

$$\Delta_N(x) = (-1)^{N+1} \frac{\cos(\xi)}{(2N+2)!} x^{2N+2},$$

mit einem $\xi \in]0, x[$, also:

$$\left|\varepsilon_{app}^N(x)\right| \;=\; \left|\frac{\Delta_N(x)}{\cos(x)}\right| \;\leq\; \frac{x^{2N+2}}{(2N+2)!} \cdot \frac{1}{\cos x},$$

d.h. wir suchen ein kleines N mit

$$\frac{x^{2N+2}}{(2N+2)!} \; \frac{1}{\cos x} \;\leq\; 8.5377 \cdot 10^{-p-1},$$

bzw. mit $\cos x > 1 - \frac{x^2}{2}$ eines mit

$$\frac{x^{2N+2}}{(2N+2)!} \cdot \frac{2}{2-x^2} \;\leq\; 8.5377 \cdot 10^{-p-1},$$

implementiert als

$$x^{2N+2} \;\leq\; (2N+2)! \cdot (2-x^2) \cdot 4.2689 \cdot 10^{-p-1},$$

wobei wir wiederum einen Startwert aus einer Tabelle interpolieren. Die Tabelle zum Cosinus sieht wie folgt aus:

k	$j=1$	2	3	4	5	6	7	8	9	10
3	3	3	3	3	3	3	3	3	3	3
10	3	4	4	4	5	5	5	6	6	6
33	8	9	10	11	12	12	13	13	14	14
100	21	24	26	27	28	29	30	31	32	33
330	59	66	70	73	76	78	80	82	84	86
1000	157	171	181	188	194	200	204	209	213	216

4.3.8 Auswertung für betragskleine Argumente

Für betragskleine Argumente gilt $\sin(x) \lessapprox x$, was aus Geschwindigkeitsgründen und um große Mantissenlängen beim letzten Schritt $r_0 := a_0(1-r_1)$ zu vermeiden, ausgenutzt werden sollte.

Es soll als Einschließung $[\, x(1-b^{-p}) \,,\, x \,]$ verwendet werden können. Für $0 < x \leq 10^{-3}$ gilt:

$$\begin{aligned}
\frac{x - \sin x}{x} \;&=\; \frac{\frac{x^3}{3!} - \frac{x^5}{5!} + \frac{x^7}{7!} - + \ldots}{x} \;\leq\; \frac{\frac{x^3}{3!}(1 - \frac{x^2}{4\cdot 5}) + \frac{x^7}{7!}(1 - \frac{x^2}{8\cdot 9}) + \ldots}{x} \\
&\leq\; \frac{\frac{x^3}{6}\,(1 + x^4 + x^8 + \ldots)}{x} \;=\; \frac{x^2}{6} \cdot \frac{1}{1-x^4} \;\leq\; 0.166667 \cdot x^2 \quad \overset{!}{\leq}\; b^{-p},
\end{aligned}$$

Daraus ergeben sich (für $b = 10$) die etwas schärferen Forderungen

$$\begin{aligned} x &\overset{!}{\leq} 2.449 \cdot 10^{-\lfloor \frac{p}{2} \rfloor} \quad \text{für } p \text{ gerade,} \\ x &\overset{!}{\leq} 0.7745 \cdot 10^{-\lfloor \frac{p}{2} \rfloor} \quad \text{für } p \text{ ungerade,} \end{aligned}$$

was wiederum gröber (aber sehr schnell) getestet werden kann als

$$\operatorname{order} x \leq -\left\lfloor \frac{p}{2} \right\rfloor - 2.$$

Analog wird beim Cosinus für kleine positive Argumente $\cos(x) \overset{<}{\approx} 1$ verwendet. Es soll als Einschließung $[1 - b^{-p}, 1]$ verwendet werden (p-stellige Untergrenze). Bei $0 < x \leq 10^{-3}$ gilt folgende Abschätzung:

$$\begin{aligned} 1 - \cos x &= \frac{x^2}{2!} - \frac{x^4}{4!} + \frac{x^6}{6!} - + \ldots \leq \frac{x^2}{2}\left(1 + x^4 + x^8 + \ldots\right) \\ &= x^2 \cdot \frac{1}{2(1 - x^4)} \leq 0.500001 \cdot x^2 \overset{!}{\leq} b^{-p}, \end{aligned}$$

was zum selben Grobtest wie beim Sinus führt; es wird dann als Einschließung $[\operatorname{pred}_p(1)\,,\, 1]$ berechnet.

4.3.9 `sincos`

Es ist zusätzlich eine Funktion `sincos` implementiert, die Sinus und Cosinus eines Arguments gemeinsam berechnet:

```
void I_sincos(const BigFloat &x, BFInterval &si, BFInterval &co, long k);
```

Dabei wird jeweils zunächst die eine der beiden Funktionen berechnet, die gerade über die (etwas schnellere) Sinus-Potenzreihe berechnet wird. Die andere wird über $\sin^2 x + \cos^2 x = 1$ bestimmt. Dadurch braucht nur eine Argumentreduktion vorgenommen zu werden, und die Implementation der Quadratwurzel ist deutlich schneller als die der trigonometrischen Funktionen über die Potenzreihe.

Mit $s := \sin(x)$ und den offensichtlichen Bezeichnungen der Fehler hat man

$$|\cos x| = \sqrt{1 - \sin^2 x}, \qquad |\widetilde{\cos}\ x| = \sqrt{1 - s^2(1+\varepsilon_s)^2} \cdot (1 + \varepsilon_{\surd})$$

(die Subtraktion kann im betroffenen Bereich problemlos exakt durchgeführt werden), und es gilt für den relativen Fehler

$$\begin{aligned} |\varepsilon_{\cos}| &= \left| \frac{\sqrt{1 - s^2(1+\varepsilon_s)^2} \cdot (1 + \varepsilon_{\surd}) - \sqrt{1 - s^2}}{\sqrt{1 - s^2}} \right| \\ &= \left| \varepsilon_{\surd} - \frac{s^2 \varepsilon_s (2 + \varepsilon_s)(1 + \varepsilon_{\surd})}{2\sqrt{\xi}\sqrt{1 - s^2}} \right|, \qquad \xi \in \operatorname{conv}\{s^2, s^2(1+\varepsilon_s)^2\} \\ &\leq |\varepsilon_{\surd}| + \frac{|s|}{\sqrt{1 - s^2}} \cdot \frac{(2 + \varepsilon_s)(1 + \varepsilon_{\surd})}{2(1 - |\varepsilon_s|)} \cdot |\varepsilon_s| \overset{!}{\leq} b^{-p}. \end{aligned}$$

Die Wurzelberechnung muss schon einmal mit zwei Schutzziffern durchgeführt werden, und dann ist $|\varepsilon_{\surd}| \leq b^{-p-1}$. Der erste Quotient ist ≤ 1, und wenn die Rechnungen zumindest mit 5 Stellen durchgeführt werden, erhält man als Forderung für die Stellenzahl ℓ der Sinusberechnung

$$\frac{(1+b^{-4})^2}{1-2b^{-4}} \cdot 2b^{-\ell+1} \stackrel{!}{\leq} (b-1) \cdot b^{-p-1},$$

was bei $b = 10$ bedeutet $10^{-\ell+1} \leq 4.4982 \cdot 10^{-p-1}$, bzw. $\ell \geq p + 1.347$, also ebenfalls zwei Schutzziffern für die Sinusberechnung.

Insgesamt wird dann $|\varepsilon_{\cos}| \leq 0.300081 \cdot 10^{-p}$, was nach Berechnung einer Unterschranke zur Bestimmung einer garantierten Oberschranke verwendet werden kann.

Es ist momentan noch keine spezielle Intervall-Version von `sincos` implementiert; hier werden die einzelnen Intervall-Versionen der beiden Funktionen bemüht.

4.3.10 Intervallversion des Cosinus

Wir betrachten zunächst den Cosinus, weil bei ihm in 0 gerade eine Extremalstelle liegt und die gleich folgenden Fallunterscheidungen symmetrischer werden als beim Sinus.

Die trigonometrischen Funktionen sind, was die intervallmäßige Implementation angeht, unter den reellen Funktionen die kompliziertesten. Es sind umfangreiche Fallunterscheidungen notwendig, um festzustellen, ob das Argumentintervall Extremalstellen einschließt, und um dem Monotonieverhalten Rechnung zu tragen.

Die Fallunterscheidungen aus [Braune], die in [Steins] übernommen werden, sind manchmal zu grob, so dass die jeweilige Funktion ggf. unnötigerweise an beiden Intervallgrenzen ausgewertet wird, obwohl eine Auswertung ausreichen würde. Das liegt daran, dass nur noch ganzzahlige Anteile der Art $\lfloor \frac{x}{\pi/2} \rfloor$ (und nicht $\lfloor \frac{x}{\pi/4} \rfloor$) betrachtet werden und das Monotonieverhalten nicht mehr vollständig ausgenutzt werden kann. Außerdem wird dort gelegentlich das Maximum bzw. Minimum zweier Sinus bzw. Cosinus durch Berechnen beider Werte ermittelt, obwohl man bereits an den Argumenten ablesen kann, welches davon den benötigten Wert liefern wird (genauer weiter unten). In den entsprechenden Fällen ist unsere Version also fast doppelt so schnell.

Das Argumentintervall sei $x = [x_1, x_2]$, o.B.d.A. mit $x_1 < x_2$ (Punktintervalle werden vorab behandelt). Die Werte des Cosinus an den Grenzen sollen $c_1 := \cos x_1$, $c_2 := \cos x_2$ heißen (sie brauchen aber noch nicht berechnet zu werden). In der Implementation werden c_1 und c_2 bei Bedarf durch die Punktroutinen der letzten Abschnitte eingeschlossen, und es wird – je nach Verwendung im Ergebnis – die Obergrenze oder die Untergrenze verwendet. So sind unten angegebene Intervalle der Art $[c_1, c_2]$ zu verstehen.

Die Reduktion an beiden Grenzen liefere die reduzierten Argumente $\bar{x}_1,\ \bar{x}_2 \in [-\frac{\pi}{4}, +\frac{\pi}{4}]$ (eingeschlossen als $[\bar{x}_1]$, $[\bar{x}_2]$), außerdem die ganzzahligen Anteile

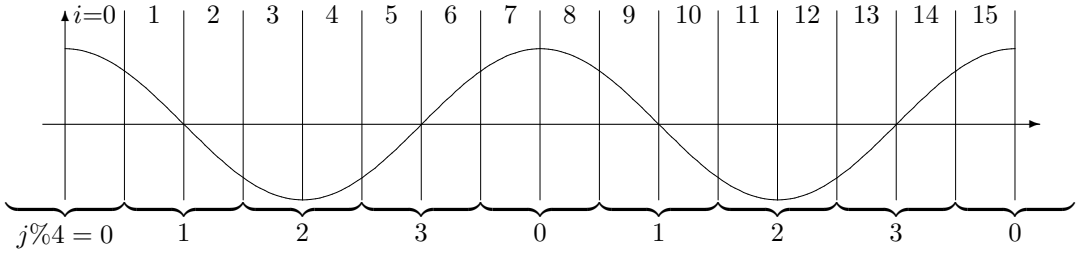
$$i_1 := \lfloor \tfrac{x_1}{\pi/4} \rfloor, \qquad i_2 := \lfloor \tfrac{x_2}{\pi/4} \rfloor,$$

also $i_1 \leq i_2$. Falls $i_2 - i_1 \geq 8$, so muss sowohl eine Maximal- wie eine Minimalstelle in x liegen, d.h. das Ergebnisintervall ist $[-1, +1]$.

Ansonsten verschieben wir das Intervall um ein Vielfaches von 2π so, dass x_1 in $[0, 2\pi]$ zu liegen kommt (damit bleiben das Ergebnis und die reduzierten Argumente unverändert), algorithmisch formuliert:

$i_1 := i_1 \bmod 8$,
$i_2 := i_2 \bmod 8$,
falls $i_2 < i_1$: $\quad i_2 := i_2 + 8$.

Dabei muss die Modulo-Operation *algebraisch* durchgeführt werden, also $\exists q \in \mathbb{Z},\ r \in \{0, \ldots, b-1\} \ : \ a = q \cdot b + r \iff : \ r = a \bmod b$; z.B. $(-1) \bmod 8 = 7$. Der normale `C++`-Operator `%` zieht das Vorzeichen heraus und kann nicht direkt verwendet werden.

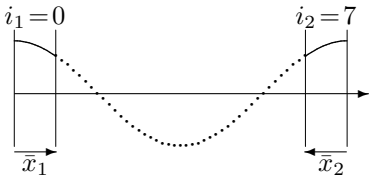


Die eigentliche Fallunterscheidung ist nun in unten stehender Tabelle aufgeführt und unter Hinzunahme des vorangegangenen Diagramms gut zu verstehen. Wir besprechen hier nur den ersten Fall ausführlich:

Falls $i_1 = 0$, so ist der Cosinus im Intervall x monoton fallend, solange $i_2 = 0, 1, 2, 3$, also $[\cos(x)] := [c_2, c_1]$. Für $i_2 = 4, 5, 6$ enthält x eine Minimalstelle, und es gilt $c_2 < c_1$, also $[\cos(x)] := [-1, c_1]$. (Für $i_2 \geq 8$ hatten wir schon oben das Ergebnis $[-1, 1]$ festgestellt.)

Falls aber gerade $i_2 = 7$, liegen x_1 und x_2 in zueinander symmetrischen Streifen (siehe das Bild unten rechts). Als Ergebnisintervall erhält man $[-1, \max\{c_1, c_2\}]$.

Man kann nun aber bereits an den reduzierten Argumenten $\bar{x}_1$, $\bar{x}_2$ ablesen, welches den größeren Cosinus besitzt. Beide werden die Cosinus-Reihe in $[-\frac{\pi}{4}, +\frac{\pi}{4}]$ benutzen, und die beiden Ränder im Bild werden durch die Argumentreduktion auf 0 abgebildet. Es ist $\bar{x}_1 > 0$ und $\bar{x}_2 < 0$.



Der Cosinus nimmt also sein Maximum an in $x_0 = \min\{[\bar{x}_1]_{\inf}, -[\bar{x}_2]_{\sup}\}$, wobei sich die Intervalle durchaus schneiden dürfen. (Natürlich gilt nie exakt $-\bar{x}_2 = \bar{x}_1$, da sonst die beiden Argumente exakt symmetrisch um eine Minimalstelle herumliegen würden, also $x_1 + x_2 = 2\pi n$ für ein ungerades n, aber $x_1, x_2 \in \mathbb{Q}$.)

In der Tabelle bedeutet eine mit a) markierte Maximumsbestimmung eine des obigen Typs, in der dort ausgewertet wird, wo das reduzierte Argument betragsmäßig am kleinsten ist. Da bei der Reduktion immer Vielfache von $\frac{\pi}{2}$ (nicht $\frac{\pi}{4}$) auf 0 abgebildet werden, ergibt sich ein symmetrischer Fall (markiert mit b)), wenn der linke Rand in einem Streifen mit *ungeradem* i_1 liegt. Dann muss dort ausgewertet werden, wo das reduzierte Argument betragsmäßig am größten ist.

i_1	i_2	[cos(x)]
0	$0,1,2,3$	$[c_2, c_1]$
	$4,5,6$	$[-1, c_1]$
	7	$[-1, \max\{c_1, c_2\}]^{a)}$
	≥ 8	$[-1, 1]$
1	$1,2,3$	$[c_2, c_1]$
	$4,5$	$[-1, c_1]$
	6	$[-1, \max\{c_1, c_2\}]^{b)}$
	7	$[-1, c_2]$
	≥ 8	$[-1, 1]$
2	$2,3$	$[c_2, c_1]$
	4	$[-1, c_1]$
	5	$[-1, \max\{c_1, c_2\}]^{a)}$
	$6,7$	$[-1, c_2]$
	≥ 8	$[-1, 1]$
3	3	$[c_2, c_1]$
	4	$[-1, \max\{c_1, c_2\}]^{b)}$
	$5,6,7$	$[-1, c_2]$
	≥ 8	$[-1, 1]$

i_1	i_2	[cos(x)]
4	$4,5,6,7$	$[c_1, c_2]$
	$8,9,10$	$[c_1, 1]$
	11	$[\min\{c_1, c_2\}, 1]^{a)}$
	≥ 12	$[-1, 1]$
5	$5,6,7$	$[c_1, c_2]$
	$8,9$	$[c_1, 1]$
	10	$[\min\{c_1, c_2\}, 1]^{b)}$
	11	$[c_2, 1]$
	≥ 12	$[-1, 1]$
6	$6,7$	$[c_1, c_2]$
	8	$[c_1, 1]$
	9	$[\min\{c_1, c_2\}, 1]^{a)}$
	$10,11$	$[c_2, 1]$
	≥ 12	$[-1, 1]$
7	7	$[c_1, c_2]$
	8	$[\min\{c_1, c_2\}, 1]^{b)}$
	$9,10,11$	$[c_2, 1]$
	≥ 12	$[-1, 1]$

Da die Beziehungen für das Ergebnisintervall jeweils mehrfach vorkommen, arbeitet auch unsere Implementation mit einer solchen Tabelle. Dafür erhalten die durchzuführenden Operationen eine Nummer zwischen 0 und 10, beispielsweise 0 für $[c_1, c_2]$ und 10 für $[-1, 1]$. Ein statisches zweidimensionales `int`-Array `codetab` wird mit den Codes gefüllt, die obiger Tabelle entsprechen. Mit einem `switch codetab[i1][i2]` wird dann einfach der entsprechende Fall ausgelöst.

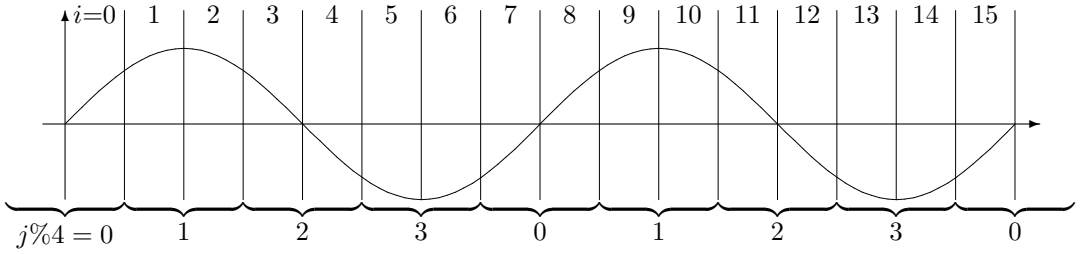
```
// coding:
// 0) [c1,c2]                          6) [-1,c1]
// 1) [c2,c1]                          7) [-1,c2]
// 2) [c1,+1]                          8) [-1,max(c1,c2)] a)
// 3) [c2,+1]                          9) [-1,max(c1,c2)] b)
// 4) [min(c1,c2),+1] a)              10) [-1,+1]
// 5) [min(c1,c2),+1] b)

static int codetab[8][16]=
 {
    {  1, 1, 1, 1, 6, 6, 6, 8,10,10,10,10,10,10,10,10 },
    { -1, 1, 1, 1, 6, 6, 9, 7,10,10,10,10,10,10,10,10 },
    { -1,-1, 1, 1, 6, 8, 7, 7,10,10,10,10,10,10,10,10 },
    { -1,-1,-1, 1, 9, 7, 7, 7,10,10,10,10,10,10,10,10 },
    { -1,-1,-1,-1, 0, 0, 0, 0, 2, 2, 2, 4,10,10,10,10 },
    { -1,-1,-1,-1,-1, 0, 0, 0, 2, 2, 5, 3,10,10,10,10 },
    { -1,-1,-1,-1,-1,-1, 0, 0, 2, 4, 3, 3,10,10,10,10 },
    { -1,-1,-1,-1,-1,-1,-1, 0, 5, 3, 3, 3,10,10,10,10 }
 };

switch ( codetab[i1][i2] )
 { ...
```

Offensichtlich ist für $i_2 \geq 12$ das Ergebnisintervall immer $[-1, 1]$. Durch die Tabellentechnik macht es zeitlich aber keinen Sinn, dies als gesonderte Fallunterscheidung voranzustellen.

4.3.11 Intervallversion des Sinus



Der Intervall-Sinus ist völlig analog zum Intervall-Cosinus zu behandeln. Die Tabelle ergibt sich im Wesentlichen durch eine Verschiebung und Umnummerierung aus der zum Cosinus. Mit den zu oben analogen Bezeichnungen erhält man:

i_1	i_2	[sin(x)]
0	$0, 1$	$[s_1, s_2]$
	2	$[s_1, 1]$
	3	$[\min\{s_1, s_2\}, 1]^{a)}$
	$4, 5$	$[s_2, 1]$
	≥ 6	$[-1, 1]$
1	1	$[s_1, s_2]$
	2	$[\min\{s_1, s_2\}, 1]^{b)}$
	$3, 4, 5$	$[s_2, 1]$
	≥ 6	$[-1, 1]$
2	$2, 3, 4, 5$	$[s_2, s_1]$
	$6, 7, 8$	$[-1, s_1]$
	9	$[-1, \max\{s_1, s_2\}]^{a)}$
	≥ 10	$[-1, 1]$
3	$3, 4, 5$	$[s_2, s_1]$
	$6, 7$	$[-1, s_1]$
	8	$[-1, \max\{s_1, s_2\}]^{b)}$
	9	$[-1, s_2]$
	≥ 10	$[-1, 1]$

i_1	i_2	[sin(x)]
4	$4, 5$	$[s_2, s_1]$
	6	$[-1, s_1]$
	7	$[-1, \max\{s_1, s_2\}]^{a)}$
	$8, 9$	$[-1, s_2]$
	≥ 10	$[-1, 1]$
5	5	$[s_2, s_1]$
	6	$[-1, \max\{s_1, s_2\}]^{b)}$
	$7, 8, 9$	$[-1, s_2]$
	≥ 10	$[-1, 1]$
6	$6, 7, 8, 9$	$[s_1, s_2]$
	$10, 11, 12$	$[s_1, 1]$
	13	$[\min\{s_1, s_2\}, 1]^{a)}$
	≥ 14	$[-1, 1]$
7	$7, 8, 9$	$[s_1, s_2]$
	$10, 11$	$[s_1, 1]$
	12	$[\min\{s_1, s_2\}, 1]^{b)}$
	13	$[s_2, 1]$
	≥ 14	$[-1, 1]$

Im Programm erscheint diese Tabelle dann wie folgt:

```
static int codes[8][16]=
 {
    {  0, 0, 2, 4, 3, 3,10,10,10,10,10,10,10,10,10,10 },
    { -1, 0, 5, 3, 3, 3,10,10,10,10,10,10,10,10,10,10 },
    { -1,-1, 1, 1, 1, 1, 6, 6, 6, 8,10,10,10,10,10,10 },
    { -1,-1,-1, 1, 1, 1, 6, 6, 9, 7,10,10,10,10,10,10 },
    { -1,-1,-1,-1, 1, 1, 6, 8, 7, 7,10,10,10,10,10,10 },
    { -1,-1,-1,-1,-1, 1, 9, 7, 7, 7,10,10,10,10,10,10 },
    { -1,-1,-1,-1,-1,-1, 0, 0, 0, 0, 2, 2, 2, 4,10,10 },
    { -1,-1,-1,-1,-1,-1,-1, 0, 0, 0, 2, 2, 5, 3,10,10 }
 };
```

4.3.12 Komplexe Versionen

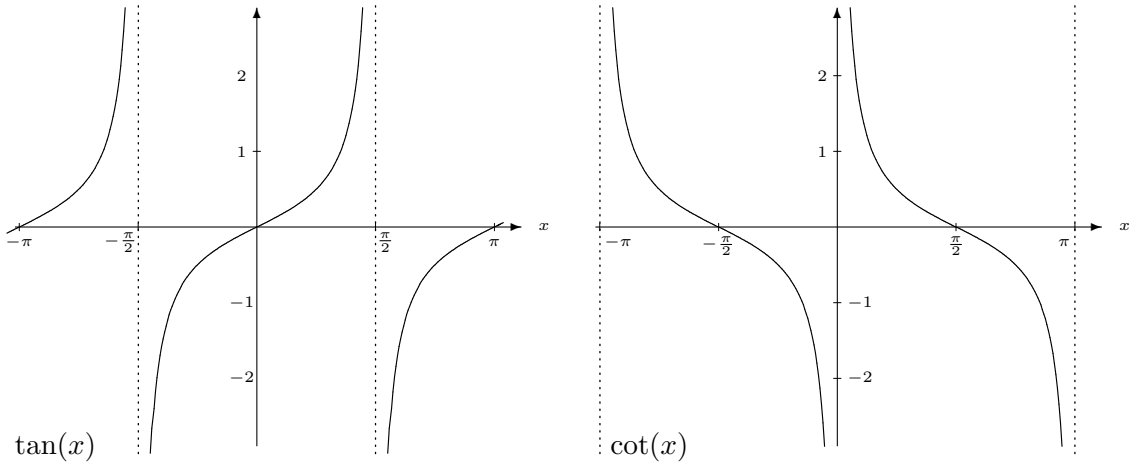
Auch die komplexen Funktionen sin und cos sind separabel, genauer gilt

$$\begin{aligned}\sin(x+iy) &= \sin(x)\cdot\cosh(y) + i\cdot\cos(x)\cdot\sinh(y),\\ \cos(x+iy) &= \cos(x)\cdot\cosh(y) - i\cdot\sin(x)\cdot\sinh(y).\end{aligned}$$

Man kann also wiederum ohne Überschätzung jeweils mit einer Intervall-Auswertung dieser Ausdrücke arbeiten. Die Fehlerabschätzungen entsprechen genau denen bei der komplexen Exponentialfunktion (siehe Seite 85), d.h. die beteiligten vier reellen Standardfunktionen müssen mit jeweils zwei Schutzziffern berechnet werden.

Günstigerweise kann man hier die beiden Funktionen `sincos` und `sinhcosh` aufrufen, die die beiden jeweiligen Werte zugleich einschließen und eine Intervallreduktion bzw. eine Auswertung der Exponentialfunktion einsparen können.

4.4 Tangens und Cotangens



4.4.1 Die Potenzreihe des Tangens

Die aus der Taylorreihe stammende Reihenentwicklung des Tangens wird (wie schon bei [Steins], aber anders als bei [Braune]) für die vorliegende Implementation *nicht* verwendet:

$$\begin{aligned}\tan(x) &= \sum_{n=1}^{\infty}(-1)^{n-1}\,\frac{2^{2n}(2^{2n}-1)B_{2n}}{(2n)!}\,x^{2n-1} \qquad (x\in\,]-\tfrac{\pi}{2},+\tfrac{\pi}{2}[)\\ &= x+\frac{1}{3}x^3+\frac{2}{15}x^5+\frac{17}{315}x^7+\frac{62}{2835}x^9+\frac{1382}{155925}x^{11}+\frac{21844}{6081975}x^{13}+\ldots\end{aligned}$$

Dabei sind die $B_k\in\mathbb{Q}$ die Bernoulli-Zahlen, definiert durch

$$B_0 := 1, \qquad B_n := -\frac{1}{n+1}\cdot\sum_{k=0}^{n-1}\binom{n+1}{k}B_k \qquad (n\geq 1),$$

so dass $B_{2n+1}=0$ für $n\geq 1$; zur Illustration der Größenordnungen:

$$B_1 = -\tfrac{1}{2},\ B_2 = \tfrac{1}{6},\ B_4 = -\tfrac{1}{30},\ B_6 = \tfrac{1}{42},\ B_8 = -\tfrac{1}{30},\ B_{10} = \tfrac{5}{66},\ B_{12} = -\tfrac{691}{2730},$$
$$B_{14} = \tfrac{7}{6},\ B_{16} = -\tfrac{3617}{510},\ B_{18} = \tfrac{43867}{798},\ B_{20} = -\tfrac{174611}{330}.$$

Bei einer Arithmetik mit vorgegebener maximaler Stellenzahl ist die Anzahl der maximal notwendigen Koeffizienten fest vorgegeben, und diese können mit der maximal notwendigen Genauigkeit vorberechnet werden.

In unserem Fall müssten *zur Laufzeit* so viele Koeffizienten berechnet werden, wie jeweils gebraucht werden, was sehr aufwändig ist. Zumindest müssten sie also nach der Berechnung für die nächste Verwendung gespeichert werden. Zu diesem Zweck kann ihre Berechnung nicht mit Fließkomma-Arithmetik einer Genauigkeit durchgeführt werden, die der gerade geforderten Genauigkeit des Tangens entspricht, vielmehr müsste man mit rationaler Arithmetik arbeiten (mit Kürzen nach jeder Operation) und Zähler und Nenner vollständig aufbewahren. Wie man bereits bei den letzten der oben angegebenen Brüche erkennt, steigt deren Größe rasch an, und noch schneller, wenn man das Kürzen auslässt.

Im Rahmen dieser Implementation wurde eine solche Testversion für den Tangens geschrieben, unter Verwendung der Klasse `BigRational`. Bernoulli-Zahlen und Tangens-Koeffizienten werden in dynamischen Arrays gespeichert. Die Test-Implementation ist nicht ganz naiv, aber auch nicht vollständig durchoptimiert. Natürlich werden vorkommende Zweierpotenzen, Fakultäten und ggf. mehrfach vorkommende Binomialkoeffizienten dabei nicht immer wieder neu berechnet. Dennoch benötigte die Bereitstellung der Koeffizienten für eine 50-stellige Tangensberechnung über 5 Sekunden.

4.4.2 Berechnung über Sinus und Cosinus

Die Verwendung von

$$\tan x = \frac{\sin x}{\cos x} \qquad (x \notin \frac{\pi}{2} + \pi\mathbb{Z})$$

ist, besonders bei höheren Stellenzahlen, sogar von den Konvergenzgeschwindigkeiten her geeigneter. Beispielsweise benötigt die Tangens-Reihe bei $x = 0.5$ etwa so viele Reihenglieder wie geforderte Dezimalstellen. Sinus und Cosinus zusammen kommen mit weniger Gliedern aus (bei 16 bis 100 Stellen zwischen 5 und 40 %). Der Cotangens wird entsprechend als

$$\cot x = \frac{\cos x}{\sin x} \qquad (x \notin \pi\mathbb{Z})$$

berechnet.

Die Argumentreduktion braucht natürlich jeweils nur *einmal* durchgeführt zu werden. Aus der Tabelle zu Sinus und Cosinus auf Seite 87 ergibt sich nun:

$j = 4n$	$\tan(x) = \sin(x_1)/\cos(x_1)$	$\cot(x) = \cos(x_1)/\sin(x_1)$
$j = 4n+1$	$\tan(x) = -\cos(x_1)/\sin(x_1)$	$\cot(x) = -\sin(x_1)/\cos(x_1)$
$j = 4n+2$	$\tan(x) = \sin(x_1)/\cos(x_1)$	$\cot(x) = \cos(x_1)/\sin(x_1)$
$j = 4n+3$	$\tan(x) = -\cos(x_1)/\sin(x_1)$	$\cot(x) = -\sin(x_1)/\cos(x_1)$,

d.h. es ist nur eine Fallunterscheidung j gerade/ungerade vorzunehmen.

Die Division kann mit p Stellen maximal genau durchgeführt werden, so dass wir nur fordern müssen, dass der relative Fehler beim exakten Quotienten betragsmäßig durch b^{-p} beschränkt ist. Allgemein gilt für den relativen Gesamtfehler bei der Division $\tilde{a}/\tilde{b}$ mit fehlerbehafteten Größen $\tilde{a} = a(1+\varepsilon_a)$, $\tilde{b} = b(1+\varepsilon_b)$ (mit $|\varepsilon_b| < 1$):

$$|\varepsilon_{ges}| \;=\; \left|\frac{\frac{a(1+\varepsilon_a)}{b(1+\varepsilon_b)} - a/b}{a/b}\right| \;=\; \left|\frac{1+\varepsilon_a}{1+\varepsilon_b} - 1\right| \;=\; \left|\frac{\varepsilon_a - \varepsilon_b}{1+\varepsilon_b}\right| \;\leq\; \frac{|\varepsilon_a| + |\varepsilon_b|}{1+\varepsilon_b}. \tag{4.19}$$

Wenn sin und cos mit ℓ Stellen hoch genau berechnet werden, ergibt sich (mit $\ell \geq 5$) die Forderung

$$\frac{2b^{-\ell+1} + 2b^{-\ell+1}}{1 - 2b^{-\ell+1}} \;\leq\; \frac{4}{1-2b^{-4}} \cdot b^{-\ell+1} \;\overset{!}{\leq}\; b^{-p},$$

bzw. für $b = 10$ schärfer $\ell \geq p + 1.603$, d.h. die Funktionsauswertungen müssen mit mindestens zwei Schutzziffern erfolgen.

4.4.3 Berechnung über Sinus und Wurzel

Analog zu `sincos` kann man natürlich versuchen, die eine der beiden beteiligten Funktionen durch die andere und die (viel schnellere) Quadratwurzel auszudrücken. Es stellt sich heraus, dass es noch schneller ist, außerdem ein Additionstheorem zu verwenden, das das Argument für die verwendete Potenzreihe verkleinert. Beim Sinus allein lohnte ein solches Vorgehen nicht, da in seinem Additionstheorem zusätzlich der Cosinus erscheint, aber hier werden ja ohnehin beide Funktionen direkt oder indirekt benötigt.

Wir betrachten für den Tangens die Darstellung

$$\tan 2x = \frac{\sin 2x}{\cos 2x} = \frac{2\,\sin x\,\cos x}{1 - 2\sin^2 x} = \frac{2\,\sin x\,\sqrt{1-\sin^2 x}}{1-2\sin^2 x} = \frac{2\,\sqrt{1-\cos^2 x}\,\cos x}{2\cos^2 x - 1}$$

Da die Potenzreihen um 0 herum ausgewertet werden, wird immer der vorletzte Term Verwendung finden. Der Cosinus wird für kleine Argumente fast 1, so dass es beim letzten Term zu Auslöschungseffekten käme.

Wir werden weiter unten sehen, dass für alle beteiligten Operationen nur zwei Schutzziffern notwendig sind, d.h. genau so viele wie bei der Version mit Sinus und Cosinus aus dem vergangenen Abschnitt (siehe aber die Bemerkung am Ende des Abschnitts). Daher benutzt die aktuelle Version *immer* die Darstellung über die Argumentreduktion. Sie erwies sich als um ca. 15% bis 30% schneller als die Version des letzten Abschnitts, ist aber weniger oft maximal genau. Es kann daher zur Compilationszeit (über einer Präprozessor-Konstante) gewählt werden, welche Fassung verwendet werden soll.

Für das Argument x des zu berechnenden Tangens wird die Argumentreduktion durchgeführt, die j und das reduzierte Argument x_1' liefern. Es sei dann $x_1 := \frac{1}{2}x_1'$, d.h. $-\frac{\pi}{8} <$

$x_1 < +\frac{\pi}{8}$. Man erhält Folgendes:

$$\tan(x) = \begin{cases} \dfrac{2\sin(x_1)\sqrt{1-\sin^2 x_1}}{1-2\sin^2 x_1} & \text{für } j \text{ gerade,} \\ \dfrac{2\sin^2 x_1 - 1}{2\sin(x_1)\sqrt{1-\sin^2 x_1}} & \text{für } j \text{ ungerade.} \end{cases}$$

Es braucht o.B.d.A. nur $x_1 > 0$ betrachtet zu werden. Die Quadratwurzel braucht nicht mit Intervallargument aufgerufen zu werden. Wir berechnen mit gerichteten Rundungen die Obergrenze einer Einschließung des obigen Ausdrucks und die Untergrenze aus einer Fehlerabschätzung.

Zunächst sei j gerade. Es sei $s := \sin x_1 > 0$, die relativen Fehlerterme sind in offensichtlicher Weise bezeichnet. Dann hat man:

$$y := \frac{2s\ \sqrt{1-s^2}}{1-2s^2}, \qquad \tilde{y} = \frac{2s(1+\varepsilon_1)\ \sqrt{1-s^2(1-\varepsilon_2)^2}\ (1+\varepsilon_{\surd})(1+\varepsilon_d)}{1-2s^2(1+\varepsilon_1)^2},$$

$$\begin{aligned} 0 \le \varepsilon_y &= \frac{(1+\varepsilon_1)\ \sqrt{1-s^2(1-\varepsilon_2)^2}\ (1+\varepsilon_{\surd})(1+\varepsilon_d)(1-2s^2)}{(1-2s^2(1+\varepsilon_1)^2)\ \sqrt{1-s^2}} - 1 \\ &= \frac{(1+\varepsilon_1)\ \sqrt{1+\frac{s^2\varepsilon_2(2-\varepsilon_2)}{(1-s^2)}}\ (1+\varepsilon_{\surd})(1+\varepsilon_d)(1-2s^2)}{1-2s^2(1+\varepsilon_1)^2} - 1 \\ &\le \frac{(1+\varepsilon_1)\left(1+\frac{s^2\varepsilon_2(2-\varepsilon_2)}{2(1-s^2)}\right)(1+\varepsilon_{\surd})(1+\varepsilon_d)(1-2s^2)}{1-2s^2(1+\varepsilon_1)^2} - 1 \end{aligned}$$

Letzteres liefert nach längerem Umformen bei $b = 10$ unter Verwendung von $\ell \ge 5$, $s \le \sin\frac{\pi}{8}$ (und $0 \le \varepsilon \le 2 \cdot 10^{-\ell+1}$ für alle vorkommenden relativen Fehler ε):

$$\varepsilon_y \le \frac{9.65875154}{1.20690676} \cdot 10^{-\ell+1} \ \le\ 8.0029 \cdot 10^{-\ell+1},$$

d.h. für $10^{-p} \le 8.0029 \cdot 10^{-\ell+1}$ müssen wir nur zwei Schutzziffern fordern. Wenn die berechnete Obergrenze $\bar{y}$ ist, erhalten wir also eine Untergrenze als $\bar{y} \cdot (1 - 8.003 \cdot 10^{-p-1})$. Da die beteiligte Konstante relativ groß ist, und damit wir möglichst oft eine maximal genaue Einschließung erhalten, wird in der aktuellen Version doch mit drei Schutzziffern gerechnet, was die Gesamtberechnung nur wenig (im relevanten Stellenbereich um weniger als 5 %) verlangsamt.

Bei ungeradem j muss (abgesehen vom Vorzeichen) der Kehrwert berechnet werden. Aus einer völlig analogen Fehlerrechnung ermittelt man auch dort die Notwendigkeit von zwei Schutzziffern (und eine noch etwas größeren Konstante). Beim Cotangens sind einfach die beiden Beziehungen auszutauschen.

4.4.4 Auswertung für sehr kleine Argumente

Für sehr kleine x ist $\tan(x) \gtrapprox x$. Es gilt (für $0 < x \leq 10^{-3}$):

$$\begin{aligned}\frac{\tan x - x}{x} &= \frac{\frac{x^3}{3} + \frac{2}{15}x^5 + \frac{17}{315}x^7 + \ldots}{x} \;\leq\; \frac{x^2}{3}\,(1 + x^2 + x^4 + x^6 + \ldots) \\ &= x^2 \cdot \frac{1}{3(1-x^2)} \;\leq\; 0.333334\,x^2 \quad \overset{!}{\leq}\; b^{-p}.\end{aligned}$$

Der vergröberte Test ist derselbe wie beim Sinus, und die dann berechnete Einschließung $[\,x\,,\, x(1+b^{-p})\,]$.

4.4.5 Intervall-Tangens und -Cotangens

Bei diesen beiden Funktionen kommt es nur darauf an, festzustellen, ob eine Definitionslücke im Argumentintervall $x = [x_1, x_2]$ enthalten ist, also $\frac{\pi}{2} + \pi z$ beim Tangens bzw. πz beim Cotangens ($z \in \mathbb{Z}$). In diesem Fall ist eine Exception zu werfen. Zwischen diesen Stellen sind die Funktionen monoton steigend (Tangens) bzw. fallend (Cotangens).

x_1 und x_2 können niemals exakt auf einer (transzendenten) Definitionslücke liegen. Lediglich der eine Fall $x_i = 0$ beim Cotangens muss vorab getestet werden. Die Argumentreduktion garantiert mit dynamischer Genauigkeit, dass die Einschließungen der reduzierten Argumente nicht 0 enthalten, so dass auch nicht durch Überschätzung eine Definitionslücke eingeschlossen würde.

Außerdem schenken uns die Argumentreduktionen wiederum die ganzzahligen Anteile i_1 und i_2 bei Division durch $\frac{\pi}{4}$. Es liegt jedenfalls eine Definitionslücke zwischen x_1 und x_2, falls $i_2 - i_1 \geq 4$, und ansonsten genau dann, wenn (vgl. auch [Braune], S. 73–77)

$$(i_1 + 2) \bmod 4 > (i_2 + 2) \bmod 4 \quad \text{(tan)}, \qquad \text{bzw.} \qquad i_1 \bmod 4 > i_2 \bmod 4 \quad \text{(cot)}.$$

Da der Modulo-Operator `%` in `C` nicht algebraisch rechnet (sondern um 0 symmetrisch), behelfen wir uns wie folgt (Codestück aus dem Tangens):

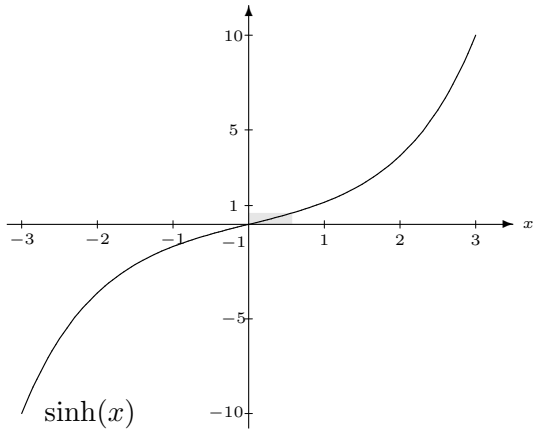
```
bool domain_error=false;
if (I2-I1>=4) domain_error=true;
else if (I1>=-2)
 {
   if ( (I1+2)/4!=(I2+2)/4 ) domain_error=true;
 }
else
 {
   if ( I2>=-2 ||  (1-I1)/4!=(1-I2)/4 ) domain_error=true;
 }
if (domain_error) ...
```

4.5 Der hyperbolische Sinus

Der hyperbolische Sinus

$$\sinh x = \frac{e^x - e^{-x}}{2}$$

wird wie bei [Braune] und [Steins] je nach Argument über die Potenzreihe oder eben über die Exponentialfunktion berechnet. Aufgrund der Symmetrie $\sinh(-x) = -\sinh x$ brauchen wir nur positive Argumente zu betrachten.



Die obige Darstellung ist numerisch ungünstig lediglich bei kleinen Argumenten x. Es kann aber für $e^x - e^{-x} > 1$, d.h. $x > \ln \frac{\sqrt{5}+1}{2} \approx 0.4812$, keine Auslöschung eintreten. Da das Argument der Exponentialfunktion ggf. in mehreren Schritten auf ein Intervall $[0, \frac{\ln 10}{2^n}]$ reduziert wird, wählen wir die etwas größere Schranke $\frac{\ln 10}{4} \approx 0.5756$ (wie in der Arbeit [Steins], in der zugehörigen Implementation wird dagegen mit 0.5 gearbeitet).

Für kleinere Argumente wird die Potenzreihe

$$\sinh x \;=\; \sum_{k=0}^{\infty} \frac{x^{2k+1}}{(2k+1)!} \;=\; x + \frac{x^3}{3!} + \frac{x^5}{5!} + \ldots$$

herangezogen, wobei wiederum ein modifiziertes Horner-Verfahren verwendet wird.

4.5.1 Berechnung über die Exponentialfunktion

4.5.1.1 Berechnung für große Argumente

Für große Argumente x wird $\sinh x$ sehr gut durch $\frac{1}{2}e^x$ approximiert, und der e^{-x}-Term (d.h. insbesondere eine Division) braucht nicht berechnet zu werden. Wir berechnen die Obergrenze von $\frac{1}{2}e^x$ und erhalten eine Untergrenze aus unten stehender Fehlerabschätzung.

Das ist nicht nur eine Maßnahme zur Beschleunigung der Rechnung: Wenn die Subtraktion intern *immer völlig exakt* durchgeführt wird, führt dies bei großen x zu extrem langen Mantissen, die nicht nur überflüssig zu berechnen sind, sondern ggf. gar nicht mehr im Speicher Platz finden. Die Implementation zu [Steins] arbeitet beispielsweise so und nimmt keine entsprechende Überprüfung vor. Sie stürzt in solchen Fällen mit einem „Segmentation Fault“ ab (ebenso bei cosh und tanh). Nach unserer noch zu beschreibenden Vorab-Behandlung können wir dagegen exakt subtrahieren (im nächsten Abschnitt).

Die Grenze, ab der der e^{-x}-Term weggelassen werden kann, hängt natürlich von der gewünschten Gesamtstellenzahl ab. Da die Bedingung $x \geq \frac{\ln 10}{4}$ fest eingeht, setzen wir hier ausnahmsweise $b = 10$ voraus. Es gilt folgende Beziehung für den relativen Fehler:

$$\varepsilon_{\sinh}(x) = \frac{\frac{e^x(1+\varepsilon_e)}{2} - \frac{e^x - e^{-x}}{2}}{\frac{e^x - e^{-x}}{2}} = \frac{e^x\varepsilon_e + e^{-x}}{e^x - e^{-x}} = \varepsilon_e \underbrace{\frac{e^x}{e^x - e^{-x}}}_{\leq 1.4625} + \frac{1}{e^{2x} - 1}.$$

Wenn die Exponentialfunktion mit ℓ Stellen ausgewertet wird, ist der erste Summand $\leq 2.925 \cdot 10^{-\ell+1}$. Wir fordern naheliegenderweise für den zweiten Summanden eine Schranke in der gleichen Größenordnung, allerdings nicht zu groß, um möglichst oft maximal genau zu bleiben. Um eine einfach zu überprüfende Bedingung zu erhalten, fordern wir daher $e^{2x} \geq 10^{\ell-1}$, d.h. $\frac{2x}{\ln 10} \geq \ell - 1$, näherungsweise getestet als

$$0.8686 \cdot x \geq \ell - 1.$$

Bei 16 Stellen erhält man damit beispielsweise $x \geq 17.27$. Für den relativen Gesamtfehler ergibt sich dann

$$\varepsilon_{\sinh} \leq 2.925 \cdot 10^{-\ell+1} + \frac{1}{10^{\ell-1}-1} \leq 10^{-\ell+1}\left(2.925 + \frac{1}{1-10^{-4}}\right) \leq 3.926 \cdot 10^{-\ell+1}$$

Wir benötigen also 2 Schutzziffern, und aus dieser Schranke erhalten wir eine Untergrenze für unsere Einschließung.

4.5.1.2 Berechnung mit beiden Termen

Für kleinere x als im letzten Abschnitt müssen beide Terme eingehen. Mit den exakten bzw. inexakten Auswertungen

$$\sinh x = \frac{1}{2}\left(e^x - \frac{1}{e^x}\right), \qquad \widetilde{\sinh}\, x = \frac{1}{2}\left(e^x(1+\varepsilon_e) - \frac{1+\varepsilon_/}{e^x(1+\varepsilon_e)}\right)$$

mit exakt durchgeführter Subtraktion und Division durch 2 ergibt sich folgende Fehleranalyse:

$$\begin{aligned}
|\varepsilon_{\sinh}| &= \frac{1}{2\sinh x}\left|e^x(1+\varepsilon_e) - \frac{1+\varepsilon_/}{e^x(1+\varepsilon_e)} - e^x + \frac{1}{e^x}\right| \\
&= \frac{1}{2\sinh x}\left|e^x\varepsilon_e + \frac{1}{e^x}\left(\varepsilon_e - \frac{\varepsilon_/ + \varepsilon_e^2}{1+\varepsilon_e}\right)\right| \;=\; \frac{1}{\sinh x}\left|\varepsilon_e \cosh x - \frac{1}{2e^x}\cdot\frac{\varepsilon_/ + \varepsilon_e^2}{1+\varepsilon_e}\right| \\
&= \left|\varepsilon_e \coth x - \frac{1}{e^{2x}-1}\cdot\frac{\varepsilon_/ + \varepsilon_e^2}{1+\varepsilon_e}\right| \qquad \leq\; |\varepsilon_e| \coth x + \frac{1}{e^{2x}-1}\cdot\frac{|\varepsilon_/| + |\varepsilon_e^2|}{1+\varepsilon_e}.
\end{aligned}$$

Wenn Division maximal genau und Auswertung von exp hoch genau mit ℓ Stellen durchgeführt werden, erhält man (für $b = 10$ und mit $x \geq \frac{\ln 10}{4}$) als Forderung für p-stellige Hochgenauigkeit insgesamt:

$$10^{-\ell+1} \cdot \left(2 \cdot 1.925 + 0.4625 \cdot \frac{1 + 4 \cdot 10^{-4}}{1 - 2 \cdot 10^{-4}}\right) \overset{!}{\leq} 10^{-p},$$

bzw. $4.3128 \cdot 10^{-\ell+1} \leq 10^{-p}$, also $\ell \geq p + 1.635$, wiederum zwei Schutzziffern.

Wir führen hier eine volle Intervalldivision durch (also zwei Punkt-Divisionen), da sich ein Umweg über eine Grenze und die Fehlerabschätzung zeitlich gar nicht lohnt und wir so außerdem häufiger maximale Genauigkeit erzielen.

4.5.2 Berechnung über die Potenzreihe

Die Potenzreihe entspricht der des Sinus, nur mit durchgehend positivem Vorzeichen der Summanden. Entsprechend gibt es in Auswertung und Fehlerabschätzungen viele Parallelen. Es ist wiederum ein Polynom in $u = x^2$ auszuwerten.

$$\sinh x \;=\; x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \ldots \;=\; x \sum_{k=0}^{\infty} \frac{x^{2k}}{(2k+1)!} \;=\; x \sum_{k=0}^{\infty} \frac{u^k}{(2k+1)!}$$

Im Vergleich zum Sinus ist zu beachten, dass (für $x > 0$) immer gilt $P_N(x) < \sinh(x)$, d.h. wir berechnen immer eine Untergrenze.

4.5.2.1 Auswertung mit dem Horner-Verfahren

Wir verwenden also als Approximationspolynom

$$P_N(x) := \sum_{k=0}^{N} \frac{x^{2k+1}}{(2k+1)!} = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \ldots + \frac{x^{2N+1}}{(2N+1)!},$$

in folgender Weise geklammert:

$$P_N(x) = x \left(1 + \frac{u}{2 \cdot 3}\left(1 + \frac{u}{4 \cdot 5}\left(1 + \frac{u}{6 \cdot 7}\left(1 + \ldots \left(1 + \frac{u}{(2N)(2N+1)}\right)\right)\right)\ldots\right)\right),$$

so dass wir folgendes Verfahren erhalten:

$$\begin{array}{lll}
\text{a)} & r_N & := \dfrac{u}{(2N)(2N+1)}, \\
\text{b)} & r_k & := \dfrac{u}{(2k)(2k+1)}(1 + r_{k+1}) \qquad \text{für } k = N-1, \ldots, 1, \\
\text{c)} & r_0 & := x(1 + r_1), \\
\Rightarrow & P(x) & = r_0,
\end{array}$$

4.5.2.2 Fehlerabschätzung des geänderten Verfahrens

Wie beim Sinus ist hier Korollar 9 verwendbar. Mit $a_0 = 1$ und für $k \geq 1$

$$a_k = \frac{1}{(2k)(2k+1)}, \quad \text{also} \quad \prod_{k=0}^{n} a_k = \frac{1}{(2n+1)!},$$

liefert es uns folgende Aussage:

$$\Delta_P \in E_\ell \Big[r_0 x + r_1 x(1+E_\ell)(2+E_\ell) + (2+E_\ell) \sum_{k=2}^{N} r_k \frac{x^{2k-1}}{(2k-1)!} + 2E_\ell(2+E_\ell) \sum_{k=2}^{N} k\, r_k \frac{x^{2k-1}}{(2k-1)!} \Big].$$

Für $k > 0$ ist

$$r_k = (2k-1)! \sum_{i=1}^{N-k+1} \frac{x^{2i}}{(2k+2i-1)!},$$

was nicht so direkt abschätzbar ist wie die alternierende Version beim Sinus. Wir erhalten Folgendes:

$$\Delta_P \;\in\; E_\ell\Big[\, r_0 x \,+\, r_1 x(1+E_\ell)(2+E_\ell) \,+\, (2+E_\ell)\underbrace{\sum_{k=2}^{N}\sum_{i=1}^{N-k+1}\frac{x^{2k+2i-1}}{(2k+2i-1)!}}_{A} \\ +\,2E_\ell(2+E_\ell)\underbrace{\sum_{k=2}^{N}\sum_{i=1}^{N-k+1}k\frac{x^{2k+2i-1}}{(2k+2i-1)!}}_{B}\,\Big].$$

Nun gilt für die beiden Summen A und B (mit $N \geq 3$):

$$\begin{aligned}
A &= \sum_{k=2}^{N}\sum_{i=1}^{N-k+1}\frac{x^{2k+2i-1}}{(2k+2i-1)!} = \sum_{k=2}^{N}\frac{x^{2k+1}}{(2k+1)!}\,(k-1) \\
&= \frac{1}{2}\sum_{k=2}^{N}\frac{x^{2k+1}}{(2k+1)!}(2k+1) - \frac{3}{2}\sum_{k=2}^{N}\frac{x^{2k+1}}{(2k+1)!} \\
&= \frac{x}{2}\left(\sum_{k=0}^{N}\frac{x^{2k}}{(2k)!}(2k+1) - 1 - \frac{x^2}{2}\right) - \frac{3}{2}\sum_{k=2}^{N}\frac{x^{2k+1}}{(2k+1)!} \\
&\leq \frac{x}{2}\left(\cosh x - 1 - \frac{x^2}{2} - \frac{3x^4}{120} - \frac{3x^6}{5040}\right), \\
B &= \sum_{k=2}^{N}\sum_{i=1}^{N-k+1}k\,\frac{x^{2k+2i-1}}{(2k+2i-1)!} = \sum_{k=2}^{N}\left(\frac{x^{2k+1}}{(2k+1)!}\sum_{i=2}^{k}i\right) = \sum_{k=2}^{N}\frac{x^{2k+1}}{(2k+1)!}\left(\frac{k(k+1)}{2} - 1\right) \\
&= \frac{1}{2}\sum_{k=2}^{N}k^2\,\frac{x^{2k+1}}{(2k+1)!} + \frac{1}{2}\sum_{k=2}^{N}k\,\frac{x^{2k+1}}{(2k+1)!} - \sum_{k=2}^{N}\frac{x^{2k+1}}{(2k+1)!} \\
&= \frac{1}{8}\sum_{k=2}^{N}(2k+1)2k\,\frac{x^{2k+1}}{(2k+1)!} + \frac{1}{8}\sum_{k=2}^{N}(2k+1)\,\frac{x^{2k+1}}{(2k+1)!} - \frac{9}{8}\sum_{k=2}^{N}\frac{x^{2k+1}}{(2k+1)!} \\
&= \frac{x^2}{8}\sum_{k=1}^{N-1}\frac{x^{2k+1}}{(2k+1)!} + \frac{x}{8}\sum_{k=2}^{N}\frac{x^{2k}}{(2k)!} - \frac{9}{8}\sum_{k=2}^{N}\frac{x^{2k+1}}{(2k+1)!} \\
&= \frac{x^5}{48} - \left(\frac{9-x^2}{8}\right)\sum_{k=2}^{N-1}\frac{x^{2k+1}}{(2k+1)!} - \frac{9}{8}\cdot\frac{x^{2N+1}}{(2N+1)!} + \frac{x}{8}\sum_{k=0}^{N}\frac{x^{2k}}{(2k)!} - \frac{x}{8} - \frac{x^3}{16} \\
&\leq \frac{x}{8}\cosh x - \frac{x}{8} - \frac{x^3}{16} + \frac{x^5}{48} - \frac{9-x^2}{8}\cdot\frac{x^5}{120} \\
&\leq \frac{x}{8}\cosh x - \frac{x}{8} - \frac{x^3}{16} + \frac{11}{960}x^5 + \frac{x^7}{960}.
\end{aligned}$$

Außerdem gilt

$$\frac{r_1 x}{r_0 x} = \frac{r_1}{r_0} = \frac{r_0 - 1}{r_0} = 1 - \frac{1}{r_0},$$

und r_0 in den Nennern von ε_P kann nach unten durch die ersten beiden Glieder, nach oben durch $\frac{\sinh x}{x}$ abgeschätzt werden.

Damit erhält man nun insgesamt

$$\begin{aligned}\varepsilon_P \;\in\; E_\ell \Bigg[1 + (1+E_\ell)(2+E_\ell)\left(1 - \frac{x}{\sinh x}\right) + (2+E_\ell)\,\frac{\cosh x - 1 - \frac{x^2}{2} - \frac{x^4}{40} - \frac{x^6}{1680}}{2(1+\frac{x^2}{6})} \\ +2E_\ell(2+E_\ell)\,\frac{\frac{1}{8}\cosh x - \frac{1}{8} + \frac{x^2}{16} + \frac{11}{960}x^4 + \frac{x^6}{960}}{1+\frac{x^2}{6}} \Bigg].\end{aligned}$$

Die Faktoren der Fehlerterme sind wieder monoton steigend für den betrachteten Bereich $0 < x < \frac{\ln 10}{4}$, so dass mit $b = 10$, $\ell \geq 5$ gilt:

$$\begin{aligned}\varepsilon_P \;&\in\; E_\ell \Big[1 + 0.10634627 + 0.0017620428 + 1.6412598 \cdot 10^{-5} \Big] \\ &\subseteq\; 1.108125\, E_\ell.\end{aligned}$$

Die Kontrolle auf dem Rechner ergibt einen Faktor von etwa 1.108107. Mit dem normalen Horner-Verfahren wird in [Steins] 1.2168 errechnet. Wieder einmal brauchen wir also zwei Schutzziffern.

4.5.2.3 Approximationsfehler

Aus dieser Konstante resultiert folgende Schranke für den Approximationsfehler:

$$\begin{aligned}|\varepsilon_{app}| \;&\leq\; 10^p \cdot \frac{1 - 1.108125 \cdot 10^{-1}}{1 + 1.108125 \cdot 10^{-4}}, \quad \text{bzw.} \\ |\varepsilon_{app}| \;&\leq\; 8.8909 \cdot 10^{-p-1}.\end{aligned}$$

Das Restglied aus der Taylor-Entwicklung lautet (mit $x > 0$):

$$R_N(x) = \frac{\cosh(\xi)}{(2N+3)!}\, x^{2N+3}, \qquad \xi \in]0, x[,$$

woraus wir für den Approximationsfehler erhalten:

$$\begin{aligned}|\varepsilon_{app}(x)| = -\varepsilon_{app} \;&= \frac{\cosh(\xi)}{\sinh(x)} \cdot \frac{x^{2N+3}}{(2N+3)!} \leq \frac{\cosh(x)}{\sinh(x)} \cdot \frac{x^{2N+3}}{(2N+3)!} \\ &\leq \frac{\cosh(x)}{x} \cdot \frac{x^{2N+3}}{(2N+3)!} = \cosh(x) \cdot \frac{x^{2N+2}}{(2N+3)!}\end{aligned}$$

Da $\cosh(x)$ auf $[0, \frac{\ln 10}{4}]$ nur zwischen 1 und 1.171 variiert, kann man es durch den Wert am rechten Rand ersetzen. Der entsprechende Test ist dann

$$x^{2N+2} \leq (2N+3)! \cdot 7.597 \cdot 10^{-p-1},$$

wieder mit einem Start-N mit Interpolation aus einer vorberechneten Tabelle, die hier angegeben ist:

p	$j=1$	2	3	4	5	6	7	8	9	10
3	3	3	3	3	3	3	3	3	3	3
10	3	3	3	3	4	4	4	4	4	5
33	7	8	9	10	10	11	11	11	12	12
100	20	22	24	25	26	27	28	29	29	30
330	57	62	66	69	72	74	76	77	79	80
1000	151	164	173	180	185	190	195	198	202	205

Wie zu erwarten, erweist es sich nicht als sinnvoll, das vorkommende $\cosh(x)$ auf $[0, \frac{\ln 10}{4}]$ z.B. durch eine quadratische Funktion $q(x) = 1 + c \cdot x^2$ (mit $c \approx 0.513961$) zu beschränken, da der Term gegenüber Potenzen und Fakultäten fast nicht ins Gewicht fällt und nur sehr selten ein Reihenglied einspart.

4.5.2.4 Auswertung bei betragskleinen Argumenten

Für kleine $x > 0$ ist $\sinh(x) \gtrapprox x$, was wieder zur Beschleunigung und zur Vermeidung großer Mantissenlängen verwendet wird. Wir betrachten

$$\frac{\sinh x - x}{x} \;=\; \frac{\frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \dots}{x} \;\le\; \frac{x^2}{6}\,(1 + x^2 + x^4 + \dots) \;=\; x^2 \cdot \frac{1}{6(1-x^2)} \;\overset{!}{\le}\; b^{-p},$$

was für $0 < x \le 10^{-3}$ auf denselben Test wie beim Sinus führt, und es wird dann dann $[\,x\,,\, x(1+b^{-p})\,]$ berechnet.

4.5.3 Intervallversion

Da der hyperbolische Sinus auf ganz $\mathbb{R}$ monoton steigend ist, ist zur intervallmäßigen Implementation nichts weiter zu sagen (vgl. `exp`, Seite 85).

4.5.4 Komplexe Versionen

Auch der hyperbolische Sinus ist separabel; es ist

$$\sinh(x + iy) = \sinh(x) \cdot \cos(y) + i \cdot \cosh(x) \sin(y).$$

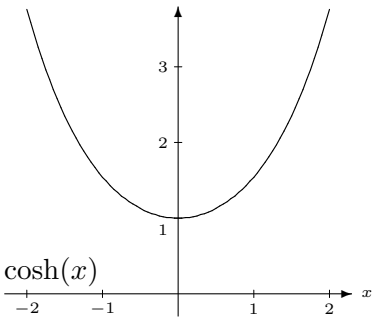
Es kann also, zusammen mit der Fehlerabschätzung analog zu exp, wieder eine einfache Intervallauswertung mit zwei Schutzziffern durchgeführt werden. Es werden dabei `sincos` und `sinhcosh` verwendet.

4.6 Der hyperbolische Cosinus

Beim hyperbolischen Cosinus

$$\cosh x = \frac{e^x + e^{-x}}{2}$$

besteht keinerlei Gefahr der Auslöschung, so dass er immer problemlos über diese Beziehung berechnet werden kann. Es gilt $\cosh(-x) = \cosh x$ und $\cosh(0) = 1$, so dass wir uns für unsere Betrachtungen auf $x > 0$ beschränken können.



Die Betrachtung, für welche Argumente der Term e^{-x} weggelassen werden kann (und dann sollte), ist analog zum hyperbolischen Sinus:

$$\varepsilon_{\cosh} = \frac{e^x\varepsilon_e - e^{-x}}{e^x + e^{-x}} = \varepsilon_e \underbrace{\frac{e^x}{e^x + e^{-x}}}_{\leq 1} - \frac{1}{e^{2x}+1}.$$

Bei $b = 10$ erhält man für $e^{2x} \geq 10^{\ell-1}$ (wieder getestet als $0.8686\,x \geq \ell - 1$):

$$|\varepsilon_{\cosh}| \leq |\varepsilon_\ell| + \frac{1}{10^{\ell-1}+1} \leq |\varepsilon_\ell| + 10^{-\ell+1} \leq 3 \cdot 10^{-\ell+1},$$

d.h. wir berechnen mit 2 Schutzziffern $\frac{1}{2}e^x$ als Untergrenze und aus dieser Abschätzung eine Obergrenze.

Bei Durchführung mit beiden Termen ist die exakte bzw. inexakte Auswertung:

$$\cosh x = \frac{1}{2}\left(e^x + \frac{1}{e^x}\right), \qquad \widetilde{\cosh}\, x = \frac{1}{2}\left(e^x(1+\varepsilon_e) + \frac{1+\varepsilon_/}{e^x(1+\varepsilon_e)}\right),$$

so dass wir für den relativen Fehler folgendes erhalten:

$$\begin{aligned}
\varepsilon_{\cosh}(x) &= \frac{1}{\cosh x}\left(e^x(1+\varepsilon_e) + e^{-x}\frac{1+\varepsilon_/}{1+\varepsilon_e} - e^x - e^{-x}\right) \\
&= \frac{1}{2\cosh x}\left(e^x\varepsilon_e + e^{-x}\left(-\varepsilon_e + \frac{\varepsilon_/ + \varepsilon_e^2}{1+\varepsilon_e}\right)\right) \;=\; \frac{1}{\cosh x}\left(e^x \sinh x + \frac{1}{2e^x}\cdot\frac{\varepsilon_/ + \varepsilon_e^2}{1+\varepsilon_e}\right) \\
&= \varepsilon_e \tanh x + \frac{1}{e^{2x+1}}\cdot\frac{\varepsilon_/ + \varepsilon_e^2}{1+\varepsilon_e},
\end{aligned}$$

d.h. für $x > 0$ gilt

$$\begin{aligned}
|\varepsilon_{\cosh}(x)| &\leq |\varepsilon_e| \cdot \underbrace{\tanh x}_{\leq 1} + \underbrace{\frac{1}{e^{2x}+1}}_{\leq 1/2} \cdot \frac{|\varepsilon_/ + \varepsilon_e^2|}{1+\varepsilon_e} \qquad \leq 2 \cdot b^{-\ell+1} + \frac{1}{2}\cdot b^{-\ell+1} \cdot \frac{1+4\,b^{-4}}{1-2\,b^{-4}} \\
&= b^{-\ell+1}\left(2 + \frac{1+4b^{-4}}{2(1-2b^{-4})}\right).
\end{aligned}$$

Für $b = 10$ bedeutet das $|\varepsilon_{\cosh}| \leq 2.5004 \cdot 10^{-\ell+1}$, also $\ell \geq p + 1.398$, d.h. wir brauchen wieder zwei Schutzziffern. Wir führen hier wieder eine volle Intervalldivision durch.

4.6.1 Intervallversion

Wegen $\cosh(-x) = \cosh x$ und $\cosh(x) \geq 1\ \forall x \in \mathbb{R}$ ist bei einer intervallmäßigen Auswertung genauso zu verfahren wie beim Quadrieren x^2.

4.6.2 `sinhcosh`

Es sind zusätzliche Versionen (mit reellem und mit Intervall-Argument), die die Werte beider Funktionen sinh und cosh zugleich einschließen. Es wird so vermieden, dass zweimal die Exponentialfunktion ausgewertet (und ggf. zweimal invertiert) wird. Diese Funktionen sind besonders sinnvoll einsetzbar in den komplexen Versionen von sin, cos, sinh und cosh.

Dabei werden die Fallunterscheidungen von sinh und cosh über die Größe des Arguments ineinander verflochten. Alle besprochenen Fälle (sehr kleine, sehr große Argumente) werden dabei abgedeckt.

```
void sinhcosh(const BigFloat &x, BFInterval &si, BFInterval &co, long k);
void sinhcosh(const BFInterval &x, BFInterval &si, BFInterval &co, long p);
```

4.6.3 Komplexe Versionen

Auch der hyperbolische Sinus ist separabel, genauer:

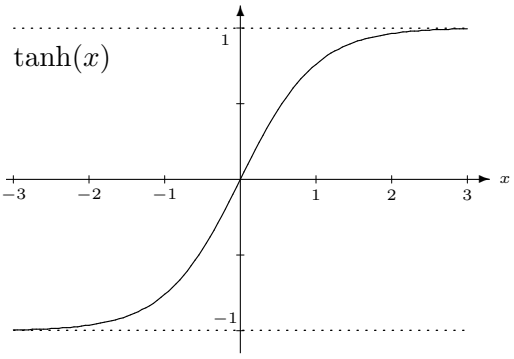
$$\cosh(x+iy) = \cosh(x)\cdot\cos(y) + i\cdot\sinh(x)\cdot\sin(y).$$

Die Fehlerabschätzung ist wieder analog zu exp, wir benötigen also jeweils zwei Schutzziffern. Die Intervallversion kann durch intervallmäßige Auswertung implementiert werden. Es können `sincos` und `sinhcosh` aufgerufen werden.

4.7 Der hyperbolische Tangens

Der hyperbolische Tangens ist (auf ganz $\mathbb{R}$) definiert als

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



Der Wertebereich ist $]-1,1[$. Eventuell überschätzte Grenzen werden bei uns (anders als bei [Steins]) nachträglich korrigiert.

Die Potenzreihenentwicklung ist die des Tangens mit alternierendem Vorzeichen, also

$$\begin{aligned}\tanh x &= \sum_{n=1}^{\infty} \frac{2^{2n}(2^{2n}-1)B_{2n}}{(2n)!}\, x^{2n-1} \qquad (x \in \,]-\tfrac{\pi}{2}, +\tfrac{\pi}{2}[)\\ &= x - \frac{1}{3}x^3 + \frac{2}{15}x^5 - \frac{17}{315}x^7 + \frac{62}{2835}x^9 - \frac{1382}{155925}x^{11} + \frac{21844}{6081975}x^{13} - + \ldots.\end{aligned}$$

Da also wiederum jeweils zur Laufzeit genügend viele Koeffizienten mit aufwändiger rationaler Arithmetik berechnet werden müssten (siehe Seite 101), ist diese Darstellung im Rahmen unserer Implementation nicht sinnvoll verwendbar. Wir benutzen daher die zuerst angegebenen Formeln.

Wegen $\tanh(-x) = -\tanh x$ setzen wir hier wieder $x > 0$ voraus. Es ist $\lim_{x\to\infty} \tanh x = 1$, und es lohnt sich, eine spezielle Betrachtung für große Argumente anzustellen. Die Formeln mit e^x kranken bei kleinen Argumenten an Auslöschungsgefahr, wo dann die erste (mit zwei Funktionsaufrufen sinh und cosh) verwendet werden muss. Wir unterscheiden daher im Folgenden zwischen kleinen, mittleren und großen Argumenten.

Die Version zu [Steins] verwendet *immer* $\frac{\sinh x}{\cosh x}$ (coth ist dort nicht implementiert). Bei kleinen Argumenten und kleinen Stellenzahlen stürzt sie wegen der Probleme von $\sinh x$ ab, bei großen Argumenten wegen überlanger Mantissen. Bei extrem großen Argumenten bricht die intern verwendete Exponentialfunktion das Programm mit einem Overflow ab, obwohl $\tanh x \approx 1$.

4.7.1 Berechnung für große Argumente

Wegen des raschen Abfallens des e^{-x}-Terms ist $\tanh x$ schnell fast nicht mehr von 1 zu unterscheiden. Wir untersuchen, ab wann man (bei p geforderten Stellen) als Ergebnisintervall $[1 - b^{-p}, 1]$ erhält (siehe auch [Braune], S. 107):

$$1 - \tanh x \le b^{-p} \quad \Leftrightarrow \quad 1 - \frac{e^{2x}-1}{e^{2x}+1} \le b^{-p} \quad \Leftrightarrow \quad \frac{2}{e^{2x}+1} \le b^{-p} \quad \Leftrightarrow \quad x \ge \frac{1}{2}\ln(2b^p - 1).$$

Es gilt nun

$$\frac{1}{2}\ln(2b^p - 1) \le \frac{1}{2}\ln(2b^p) = \frac{1}{2}(\ln 2 + p \ln b),$$

weswegen in der Implementation mit $b = 10$ der Test in der Form $x \ge 0.3466 + 1.152p$ (mit Vorab-Schnelltest $x > p$) implementiert ist. Für $p = 16$ erhält man damit beispielsweise die Bedingung $x \ge 18.7786$.

4.7.2 Berechnung für mittelgroße Argumente

Die Darstellung

$$\tanh x = \frac{e^{2x}-1}{e^{2x}+1}$$

kann für kleine Argumente wegen numerischer Probleme im Zähler nicht verwendet werden.

Keine Auslöschungsgefahr besteht für

$$e^{2x} - 1 \ge 1 \quad \Longleftrightarrow \quad x \ge \tfrac{1}{2}\ln 2 \approx 0.3466.$$

Maximal die führende Ziffer kann ausgelöscht werden für

$$e^{2x} \ge 1 + \tfrac{1}{b} \quad \Longleftrightarrow \quad x \ge \tfrac{1}{2}\ln\tfrac{b+1}{b},$$

für $b = 10$ heißt das $x \ge \frac{1}{2}\ln 1.1 \approx 0.04766$, d.h. im Bereich $x \in [0.04766, 0.3466[$ fallen zusätzliche Schutzziffern an. Noch kleinere x werden im nächsten Abschnitt behandelt.

In beiden Fällen gilt folgende Abschätzung des relativen Fehlers:

$$\begin{aligned}
|\varepsilon_{\tanh}| &= \left| \frac{\frac{e^{2x}(1+\varepsilon_e)-1}{e^{2x}(1+\varepsilon_e)+1}(1+\varepsilon_/) - \frac{e^{2x}-1}{e^{2x}+1}}{\frac{e^{2x}-1}{e^{2x}+1}} \right| \\
&= \left| \frac{(e^{2x}(1+\varepsilon_e)-1)(e^{2x}+1)(1+\varepsilon_/) - (e^{2x}(1+\varepsilon_e)+1)(e^{2x}-1)}{(e^{2x}(1+\varepsilon_e)+1)(e^{2x}-1)} \right| \\
&= \left| \frac{\varepsilon_/(1+\varepsilon_e)e^{4x} + \varepsilon_e(2+\varepsilon_/)e^{2x} - \varepsilon_/}{(e^{2x}(1+\varepsilon_e)+1)(e^{2x}-1)} \right| \\
&\leq \underbrace{\frac{e^{4x}\,(1+\varepsilon_e)-1}{(e^{2x}(1-|\varepsilon_e|)+1)(e^{2x}-1)}}_{\leq 1.00126} |\varepsilon_/| \;+\; \underbrace{\frac{(2+\varepsilon_/)\,e^{2x}}{(e^{2x}(1-|\varepsilon_e|)+1)(e^{2x}-1)}}_{\leq 10.477} |\varepsilon_e|
\end{aligned}$$

Die Angaben der unterklammerten Werte gelten (mit $b = 10$, $|\varepsilon_/| \leq 10^{-4}$, $|\varepsilon_e| \leq 2 \cdot 10^{-4}$) für $x \geq 0.04766$. Bei p gewünschten Stellen erhält man daraus die Forderung nach 4 Schutzziffern für die Exponentialfunktion. Das klingt relativ viel; es ist aber zu beachten, dass unterhalb von $\frac{\ln 10}{16} \approx 0.1439$ keine Argumentreduktion bei der Exponentialfunktion mehr stattfindet, dort also günstigerweise keine weiteren Schutzziffern anfallen.

Der mittelgroße Bereich wird in Teilstücke aufgeteilt, in denen nur drei oder zwei Schutzziffern nötig sind. Die Unterteilung orientiert sich an den Bereichen, in denen unterschiedlich viele Reduktionen innerhalb der Exponentialfunktion stattfinden.

1. Für $x \in [0.04766, 0.0.07196]$ verwenden die e^{2x}-Form (4 Schutzziffern, aber keine Argumentreduktion in exp).

2. Für $x \in [0.07196, 0.1439]$ wird *nicht* e^{2x} berechnet, sondern $(e^x)^2$ (dadurch gibt es in exp keine Argumentreduktion, die Quadrierung findet intern exakt statt).

3. Für $x \in [0.1439, 0.3466]$ wird wieder die e^{2x}-Form benutzt, es sind nur noch drei Schutzziffern notwendig (innerhalb der Exponentialfunktion wird aber intern eine Reduktion mit zwei Schutzziffern durchführt).

4. Für $x \geq 0.3466$ sind nur noch zwei Schutzziffern notwendig (und in exp wird ggf. mehrfach mit maximal drei Schutzziffern reduziert).

4.7.3 Berechnung für kleine Argumente

Für $x \leq 0.04766$ (bei $b = 10$) sind die e^x-Darstellungen wegen Auslöschungsgefahr nicht anwendbar. [Braune] verwendet die Potenzreihe, was bei unserer dynamischen Arithmetik wegen des Aufwands bei den Koeffizienten ungünstig ist. Wir benutzen also $\tanh x = \frac{\sinh x}{\cosh x}$.

In diesem Bereich wird $\sinh x$ über die Potenzreihe berechnet, $\cosh x$ wie immer über die Exponentialfunktion, aber bei diesen kleinen Argumenten immerhin ohne Argumentreduktion.

Für den relativen Fehler erhält man dieselbe Beziehung wie bei $\tan x = \frac{\sin x}{\cos x}$, also sind zwei Schutzziffern für die beiden Funktionsaufrufe nötig.

4.7.4 Berechnung für sehr kleine Argumente

Für kleine Argumente $x > 0$ gilt $\tanh(x) \lessapprox x$, so dass wir bei genügend kleinem relativen Fehler als Einschließung $[\,x(1-b^{-p})\,,\,x\,]$ verwenden können. Es gilt (für $0 < x \le 10^{-3}$):

$$\begin{aligned}\frac{x-\tanh x}{x} &= \frac{\frac{x^3}{3}-\frac{2}{15}x^5+\frac{17}{315}x^7-+\ldots}{x} \le \frac{\frac{x^3}{3}\,(1+x^4+x^8+\ldots)}{x}\\ &\le \frac{x^2}{3}\,\frac{1}{1-x^4} = 0.333334\,x^2 \overset{!}{\le} b^{-p},\end{aligned}$$

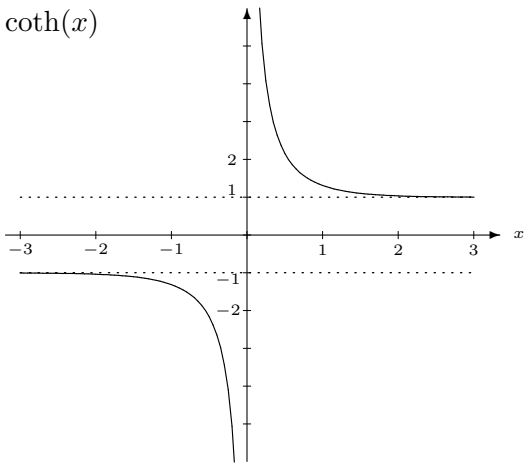
woraus sich gröber $x \le 1.732 \cdot b^{-p/2}$ ergibt, was wiederum mit $\operatorname{order} x \le -\lfloor \frac{p}{2} \rfloor - 2$ getestet werden kann.

4.7.5 Der hyperbolische Cotangens

Diese Funktion, definiert für $x \neq 0$ durch

$$\coth x = \frac{1}{\tanh x} = \frac{\cosh x}{\sinh x},$$

kann sehr einfach durch die erste Beziehung oder schneller durch die zweite, in völliger Analogie zu den Fallunterscheidungen von tanh realisiert werden. Die vorliegende Implementation benutzt Letzteres. Wegen $\coth(-x) = -\coth(x)$ brauchen wir wiederum nur positive Argumente zu betrachten.



Auch diese Funktion kann bei großen Argumenten $x > 0$ durch ± 1 approximiert werden. Es gilt:

$$\coth x - 1 = \frac{e^{2x}+1}{e^{2x}-1} - 1 = \frac{2}{e^{2x}-1} \overset{!}{\le} b^{-p+1} \quad\Leftrightarrow\quad x \ge \frac{1}{2}\ln(2b^{p-1}+1).$$

Hier haben wir nun (unter Anwendung des Mittelwertsatzes):

$$\frac{1}{2}\ln(2b^{p-1}+1) \le \frac{1}{2}\Big(\ln(2b^{p-1}) + \frac{1}{2b^{p-1}}\Big) = \frac{1}{2}\Big(\ln 2 + (p-1)\,\ln b + \frac{1}{2}\,b^{-p+1}\Big).$$

Für $b = 10$ und mit der Standardvoraussetzung $p \ge 3$ ist der Test daher als $x \ge -0.8022 + 1.152p$ implementiert. Falls er positiv ausfällt, wird $[1, 1+b^{-p+1}]$ berechnet (p-stellige Obergrenze).

Bei mittelgroßen Argumenten wird die Darstellung $\frac{e^{2x}-1}{e^{2x}+1}$ verwendet, wobei sich analog zu tanh die folgende Abschätzung ergibt:

$$|\varepsilon_{\coth}| \le \underbrace{\frac{e^{4x}(1+\varepsilon_e)-1}{(e^{2x}(1-|\varepsilon_e|)-1)(e^{2x}-1)}}_{\le 1.0034}\,|\varepsilon_/| + \underbrace{\frac{(2+\varepsilon_/)e^{2x}}{(e^{2x}(1-|\varepsilon_e|)-1)(e^{2x}-1)}}_{\le 10.499}\,|\varepsilon_e|,$$

wobei die Werte der Unterklammerung wieder für ein beliebiges $x \ge 0.04766$ gelten. Die Beziehung entspricht praktisch der des hyperbolischen Tangens, und daher findet auch dieselbe Unterteilung in Bereiche mit zwei bis vier Schutzziffern statt wie dort.

Bei kleinen Argumenten wird, analog zu tanh, mit $\frac{\cosh x}{\sinh x}$ (und sinh über die Potenzreihe) gearbeitet, wobei wie bei tanh zwei Schutzziffern anfallen.

4.7.6 Intervallversionen

$\tanh x$ ist auf ganz $\mathbb{R}$ monoton steigend und wird entsprechend simpel intervallmäßig implementiert.

Bei coth muss die Polstelle in 0 beachtet werden (es wird eine passende Exception geworfen); ansonsten ist die Funktion monoton fallend und entsprechend einfach intervallmäßig zu implementieren.

4.8 Der Arcussinus

Der Arcussinus ist zu verstehen als die Umkehrfunktion des Sinus auf $[-\frac{\pi}{2}, +\frac{\pi}{2}]$, er ist also nur für $|x| \leq 1$ definiert. Für $x = \pm 1$ geben wir direkt eine Einschließung von $\pm\frac{\pi}{2}$ zurück. Wegen $\arcsin(-x) = -\arcsin(x)$ brauchen wir nur positive Argumente zu betrachten.

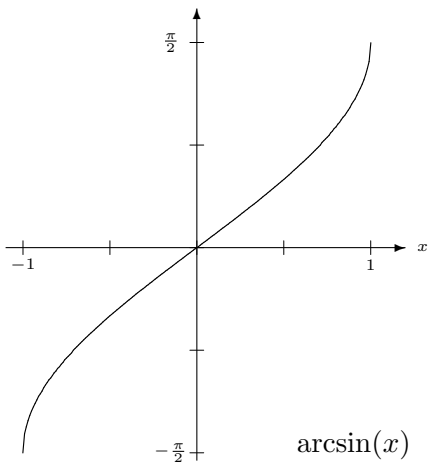
Für $|x| < 1$ wird $\arcsin(x)$ durch folgende Potenzreihe dargestellt:

$$\begin{aligned}\arcsin(x) &= \sum_{k=0}^{\infty} c_k \cdot x^{2k+1} \qquad \text{mit } c_k = \frac{1 \cdot 3 \cdot 5 \cdot 7 \cdot \ldots \cdot (2k-1)}{2 \cdot 4 \cdot 6 \cdot 8 \cdot \ldots \cdot (2k)} \cdot \frac{1}{2k+1} \\ &= x \;+\; \frac{1}{2} \cdot \frac{x^3}{3} \;+\; \frac{1 \cdot 3}{2 \cdot 4} \cdot \frac{x^5}{5} \;+\; \frac{1 \cdot 3 \cdot 5}{2 \cdot 4 \cdot 6} \cdot \frac{x^7}{7} \;+ \ldots \\ &= x \;+\; \frac{1}{6}\, x^3 \;+\; \frac{3}{40}\, x^5 \;+\; \frac{5}{112}\, x^7 \;+\; \frac{35}{1152}\, x^9 \;+\; \frac{63}{2816}\, x^{11} \;+\; \frac{231}{13312}\, x^{13} + \ldots\end{aligned}$$

Für Argumente nicht in der Nähe von 0 kann eine Argumentreduktion folgender Art durchgeführt werden (siehe [Braune], S. 79):

$$\arcsin(x) = \arcsin(x_0) + \arcsin\left(\frac{x^2 - x_0^2}{x\sqrt{1-x_0^2} + x_0\sqrt{1-x^2}}\right),$$

(für $x > x_0$ einzusetzen), die aber leider in unserem Fall nicht gut verwendbar ist. Bei einer Arithmetik mit fester Stellenzahl können der Arcussinus des Teilungspunkts x_0 und die entsprechende Wurzel fest vorberechnet werden; bei uns müssten diese beiden ggf. mit der jeweils benötigten hohen Genauigkeit zur Laufzeit nachberechnet werden.



Als Teilungspunkte bieten sich z.B. $x_0 = \frac{1}{2}$ mit $\arcsin(x_0) = \frac{\pi}{6}$ und $x_0' = \frac{\sqrt{2-\sqrt{3}}}{2} \approx 0.2588$ mit $\arcsin(x_0') = \frac{\pi}{12}$ an. Beispielsweise ergibt sich für $x > x_0$ die Gleichung

$$\arcsin(x) = \frac{\pi}{6} + \arcsin\left(\frac{4x^2 - 1}{2\sqrt{3} \cdot x + 2\sqrt{1-x^2}}\right),$$

in der die Konstanten π und $\sqrt{3}$ (z.B. noch über Kettenbrüche implementierbar) vorkommen, und zusätzlich fallen einige Schutzziffern im Argument an. Die entsprechende Formel mit x_0' wird entsprechend komplizierter, und um das Argument in den sinnvollen Bereich der Potenzreihenauswertung zu bringen, müssten schon beide angewandt werden.

Stattdessen kann die Identität

$$\arcsin(x) = \arctan \frac{x}{\sqrt{1-x^2}}$$

verwendet werden ($-1 < x < 1$). [Steins] (siehe dort S. 79) benutzt *immer* diese Formel und zieht nie die Potenzreihe heran.

Der Schwellwert $\bar{x}$, von dem an die Potenzreihe wegen zu großen Aufwands nicht mehr benutzt werden sollte, hängt auch von den Geschwindigkeiten der Implementation von Arcustangens und Quadratwurzel ab und kann nur durch entsprechende Messungen gefunden werden. In unserem Fall liegt die Grenze (bei gängigen Stellenzahlen von 16 bis 100) etwas über 0.2. Wir benutzen allerdings 0.19, da bei ca. 0.198 gerade die Grenze einer Argumentreduktion beim Arcustangens liegt (siehe dort), und da $x = 0.19$ in etwa dieses Argument für arctan liefert).

4.8.1 Berechnung der Koeffizienten

Für Arithmetiken mit vorgegebener Maximalgenauigkeit wird immer nur eine bestimmte Maximalanzahl von Koeffizienten benötigt, die dann vorberechnet werden können. Wir brauchen dagegen beliebig viele, die zur Laufzeit berechnet werden müssen.

Für aufsteigende Berechnung (bei der Bestimmung der Anzahl N der Glieder) bzw. für absteigende Berechnung (innerhalb des Horner-Verfahrens) gilt $x_0 = 1$ und für $k \geq 1$

$$c_k = c_{k-1}\frac{(2k-1)^2}{(2k)(2k+1)}, \qquad c_{k-1} = c_k\frac{(2k)(2k+1)}{(2k-1)^2}.$$

Wenn wir nun aber bei der Bestimmung von N, wie bei den bisher vorgekommenen Potenzreihen durchgeführt, nicht bei $N = 1$ (oder jedenfalls sehr klein) beginnen wollen, ist es günstiger, die Beziehung aus der Definition der c_k umzustellen, damit wir die schnellen Routinen zur Fakultät und zur Berechnung von Zweierpotenzen verwenden können.

Es gilt:

$$c_k = \frac{1 \cdot 3 \cdot 5 \cdot 7 \cdot \ldots \cdot (2k-3) \cdot (2k-1)}{2^k \cdot (1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot (k-1) \cdot k)} \cdot \frac{1}{2k+1}.$$

Für gerades k hat man dann

$$\begin{aligned} c_k &= \frac{(k+1)(k+3) \cdot \ldots \cdot (2k-3)(2k-1)}{2^k \cdot (2 \cdot 4 \cdot 6 \cdot 8 \cdot \ldots \cdot (k-2)k)} \cdot \frac{1}{2k+1} \\ &= \frac{(k+1)(k+3) \cdot \ldots \cdot (2k-3)(2k-1)}{2^{\frac{3k}{2}}(1 \cdot 2 \cdot 3 \cdot \ldots \cdot \frac{k}{2})(2k+1)} = \frac{(k+1)(k+3) \cdot \ldots \cdot (2k-3)(2k-1)}{2^{\frac{3k}{2}} \cdot (\frac{k}{2})! \cdot (2k+1)}, \end{aligned}$$

bzw. für ungerades k

$$\begin{aligned} c_k &= \frac{(k+2)(k+4)\cdot\ldots\cdot(2k-3)(2k-1)}{2^k\cdot(2\cdot 4\cdot 6\cdot 8\cdot\ldots\cdot(k-3)(k-1))}\cdot\frac{1}{2k+1} \\ &= \frac{(k+2)(k+4)\cdot\ldots\cdot(2k-3)(2k-1)}{2^{k+\lfloor\frac{k}{2}\rfloor}(1\cdot 2\cdot 3\cdot\ldots\cdot\lfloor\frac{k}{2}\rfloor)(2k+1)} = \frac{(k+2)(k+4)\cdot\ldots\cdot(2k-3)(2k-1)}{2^{k+\lfloor\frac{k}{2}\rfloor}\cdot\lfloor\frac{k}{2}\rfloor!\cdot(2k+1)}, \end{aligned}$$

wobei in beiden Fällen $\lfloor\frac{k}{2}\rfloor$ Faktoren im Zähler stehen.

[Steins] benutzt, wie erwähnt, hier nicht die Potenzreihendarstellung, wohl aber beim Areasinus, der dieselbe Reihe mit alternierendem Vorzeichen besitzt. Dort berechnet seine Implementation bei der Bestimmung von N ab $N := 2$ die c_N mit der iterativen Beziehung, also insbesondere mit einer Division pro Schritt (wir brauchen oft insgesamt nur eine). Unsere Version hat den zusätzlichen Vorteil, dass nicht mehr ganz so große Brüche entstehen und dadurch bei der späteren Rückrechnung im Horner-Verfahren die dortigen Divisionen etwas schneller werden. So erhält man beispielsweise für c_9 in der voll ausmultiplizierten Version bzw. in unserer Folgendes (ein Faktor 3 verbleibt):

$$c_9 = \frac{34459425}{3530096640}, \qquad \text{bzw.} \qquad c_9 = \frac{36465}{3735552}.$$

In der Implementation werden beim Zähler wieder immer zwei Faktoren als normale Integerzahlen vormultipliziert. In der Implementation erhält man für $4 \le k \le 46342$ folgendes Codestück zur Berechnung von c_k (als `numerator`/`denominator`):

```
long k2=k>>1;
BigInt denominator = pow(BigInt::two,k+k2) * BIfac(k2) * (2*k+1);

if ((k&1)==0) ++k;
else k+=2;

BigInt numerator(k*(k+2));
for ( long n=(k2>>1)-1 ; n>0 ; --n )
 {
   k+=4;
   numerator *= k*(k+2);
 }

if ((k2&1)!=0) numerator *= (k+4);
```

Da bei der Durchführung des Horner-Verfahrens auch Faktoren $(2k+1)(2k+2)$ vorkommen, die ebenfalls noch in (vorzeichenbehaftete) 32-Bit-Integerzahlen passen sollen, brechen wir unsere Routine ab, falls $N > 23169$ wird (Exception „internal limitation“). Falls einmal eine noch höhere Genauigkeit gewünscht sein sollte, müsste in solchen Fällen mit `BigInt` gerechnet werden.

Das oben angegebene Codestück berechnet die Koeffizienten exakt, was für die Abschätzung des Approximationsfehlers gar nicht unbedingt notwendig ist. Die tatsächliche Implementation von arcsin arbeitet nicht mit der exakten Routine `pow` für die Zweierpotenz, sondern mit einer etwas kleineren Approximation, die mit Hilfe der IEEE-Arithmetik berechnet wird (analog zu `approx_pow_up`, Seite 73). Ebenso wird bei der Aufmultiplikation im Zähler alle 4 Schritte nach oben gerundet. Da zwei aufeinander folgende Schranken für den Approximationsfehler (bei steigendem Grad) gleich Größenordnungen auseinander liegen, reichen diese Approximationen völlig aus.

4.8.2 Auswertung der Potenzreihe

Da alle Reihenglieder positiv sind, berechnen wir wieder mit nach unten gerichteten Rundungen die Untergrenze einer Einschließung, die Obergrenze ergibt sich aus unseren Fehlerabschätzungen.

Wie in Abschnitt 4.1.9 bereits angedeutet, verwenden wir hier eine spezielle Klammerung, die die teilweise fakultätsähnliche Struktur der Koeffizienten ausnutzt:

$$\begin{aligned} P_N(x) &= x\left[1+\frac{1}{2}\cdot\frac{x^2}{3}+\frac{1\cdot 3}{2\cdot 4}\cdot\frac{x^4}{5}+\frac{1\cdot 3\cdot 5}{2\cdot 4\cdot 6}\cdot\frac{x^6}{7}+\frac{1\cdot 3\cdot 5\cdot 7}{2\cdot 4\cdot 6\cdot 8}\cdot\frac{x^8}{9}+\ldots\right] \\ &= x\left[1+\frac{1}{2}\,x^2\left(\frac{1}{3}+\frac{3}{4}\,x^2\left(\frac{1}{5}+\frac{5}{6}\,x^2\left(\frac{1}{7}+\ldots+\frac{2N-1}{2N}\,x^2\cdot\left(\frac{1}{2N+1}\right)\right)\ldots\right)\right)\right]. \end{aligned}$$

Ein Horner-Schritt kommt mit nur einer Division aus, wenn die beiden beteiligten Brüche auf den Hauptnenner gebracht werden. Die entstehenden Multiplikationen spielen sich im `long`-Bereich ab, sind also vom Aufwand her völlig harmlos. Wie erwähnt wurde, handeln wir uns allerdings die Beschränkung $N \le 23169$ ein.

Das entstehende Verfahren sieht wie folgt aus ($u = x^2$):

$$\begin{aligned} &\text{a)} \quad r_N := \frac{1}{2N+1}, \\ &\text{b)} \quad r_k := \frac{1}{2k+1}+\frac{2k+1}{2k+2}\,u\,r_{k+1} = \frac{(2k+2)+(2k+1)^2\,u\,r_{k+1}}{(2k+1)(2k+2)} \quad \text{für } k = N-1,\ldots,1, \\ &\text{c)} \quad r_0 := 1+\frac{1}{2}\,x^2\,r_1 \qquad \text{(exakt durchführbar)} \\ &\text{d)} \quad P(x) = x\cdot r_0. \end{aligned}$$

4.8.3 Auswertefehler beim Horner-Verfahren

Schritt c) kann exakt durchgeführt werden. Die Division durch 2 ist exakt berechenbar, und die Addition ist unproblematisch, da sehr kleine Argumente wieder gesondert behandelt werden. Es wird erst bei der Endmultiplikation $x \cdot r_0$ gerundet.

$$\begin{aligned} &\text{a)} \quad \tilde{r}_N \in \frac{1}{2N+1}\,(1+E_\ell), \quad \Delta_N \in r_N E_\ell. \\ &\text{b)} \quad \tilde{r}_k = \frac{(2k+2)+(2k+1)^2\,u\,(1+E_\ell)\,(r_{k+1}+\Delta_{k+1})}{(2k+1)(2k+2)}\,(1+E_\ell) \\ &\qquad \subseteq \left(r_k+\frac{(2k+1)\,u\,r_{k+1}E_\ell+(2k+1)\,u\,\Delta_{k+1}(1+E_\ell)}{2k+2}\right)(1+E_\ell) \\ &\qquad \subseteq r_k+r_kE_\ell+\left(\frac{2k+1}{2k+2}\,u\,r_{k+1}E_\ell+\frac{2k+1}{2k+2}\,u\,\Delta_{k+1}(1+E_\ell)\right)(1+E_\ell) \\ &\qquad \Delta_k \in r_kE_\ell+\frac{2k+1}{2k+2}\,u\,(r_{k+1}E_\ell+\Delta_{k+1}(1+E_\ell))(1+E_\ell) \\ &\text{c)} \quad \tilde{r}_0 = 1+\tfrac{1}{2}\,u\,(r_1+\Delta_1), \qquad \Delta_0 = +\tfrac{1}{2}\,u\cdot\Delta_1. \end{aligned}$$

d) $\tilde{P} \in (r_0 + \Delta_0)\, x\, (1 + E_\ell) \subseteq x\, r_0 + x\, r_0\, E_\ell + x\, \Delta_0\, (1 + E_\ell)$,
$\Delta_P \in x\, r_0\, E_\ell + x\, (1 + E_\ell)\, \Delta_0$.

Für die vorkommenden Teilbrüche schreiben wir abkürzend

$$q_k := \frac{1 \cdot 3 \cdot 5 \cdot \dots \cdot (2k-1)}{2 \cdot 4 \cdot 6 \cdot \dots \cdot (2k)} \;=\; \prod_{i=1}^{k} \frac{2i-1}{2i} \qquad (= (2k+1)c_k\,)$$

und

$$q_k^n := \frac{(2n-1)(2n+1)(2n+3)\dots(2k-1)}{(2n)(2n+2)(2n+4)\dots(2k)} \;=\; \prod_{i=n}^{k} \frac{2i-1}{2i} \qquad \text{für } n \le k,$$

für $n > k$ sei $q_k^n = 1$ (leeres Produkt). Offenbar gilt $q_k^1 = q_k$ und die „Verlängerung“ $q_k q_n^{k+1} = q_n$.

Durch sukzessives Einsetzen erhält man (beachte $x > 0$, alle $r_k > 0$):

$$\begin{aligned}
\Delta_P \;&\in r_0 x E_\ell + \frac{1}{2}\, x^3\, (1 + E_\ell)\Delta_1 \\
&\subseteq r_0 x E_\ell + q_1\, r_1\, x^3\, E_\ell(1 + E_\ell) + q_2\, x^5\, (r_2 E_\ell + \Delta_2(1 + E_\ell))(1 + E_\ell)^2 \\
&\subseteq r_0 x E_\ell + q_1\, r_1\, x^3\, E_\ell(1 + E_\ell) + q_2\, r_2\, x^5\, E_\ell((1 + E_\ell)^2 + (1 - E_\ell)^3) \\
&\quad + q_3\, r_3\, x^7 E_\ell(1 + E_\ell)^4 - q_3\, x^7\, (1 + E_\ell)^5 \Delta_3
\end{aligned}$$

bzw. insgesamt

$$\begin{aligned}
\Delta_P \;&\in E_\ell \left[r_0 x + \frac{1}{2} r_1 x^3 (1 + E_\ell) + q_k\, x^{2k+1}\, r_k \left((1 + E_\ell)^{2k-2} + (1 + E_\ell)^{2k-1} \right) \right] \\
&= E_\ell \left[r_0 x + \frac{1}{2} r_1 x^3 (1 + E_\ell) + (2 + E_\ell) \sum_{k=2}^{N} q_k\, x^{2k+1}\, r_k (1 + E_\ell)^{2k-2} \right].
\end{aligned}$$

Für die Zwischenergebnisse r_k gilt

$$r_k \;=\; \sum_{i=0}^{N-k} q_{k+i}^{k+1} \cdot \frac{x^{2i}}{2k + 2i + 1},$$

womit sich Folgendes ergibt:

$$\Delta_P \;\in E_\ell \left[r_0 x + \frac{1}{2} r_1 x^3 (1 + E_\ell) + (2 + E_\ell) \sum_{k=2}^{N} \sum_{i=0}^{N-k} q_{k+i}\, \frac{x^{2k+2i+1}}{2k + 2i + 1} (1 + E_\ell)^{2k-2} \right],$$

bzw. nach Anwendung von Lemma 2

$$\begin{aligned}
\Delta_P \;\in\; E_\ell \Big[r_0 x + \frac{1}{2} r_1 x^3 (1 + E_\ell) + (2 + E_\ell) \overbrace{\sum_{k=2}^{N} \sum_{i=0}^{N-k} q_{k+i}\, \frac{x^{2k+2i+1}}{2k + 2i + 1}}^{A} \\
+ E_\ell (2 + E_\ell) \underbrace{\sum_{k=2}^{N} \sum_{i=0}^{N-k} (2k-1)\, q_{k+i}\, \frac{x^{2k+2i+1}}{2k + 2i + 1}}_{B} \Big].
\end{aligned}$$

Für die beiden Summen A und B gilt (mit $N \geq 3$):

$$\begin{aligned}
A &= \sum_{k=2}^{N}(k-1)\,q_k\,\frac{x^{2k+1}}{2k+1} \;\leq\; \frac{3}{8}\cdot\frac{x^5}{5} \;+\; \frac{15}{48}\sum_{k=3}^{N}(k-1)\frac{x^{2k+1}}{2k+1} \\
&= \frac{3x^5}{40} + \frac{15}{96}\sum_{k=3}^{N}(2k+1)\frac{x^{2k+1}}{2k+1} - \frac{45}{96}\sum_{k=3}^{N}\frac{x^{2k+1}}{2k+1} = \frac{3x^5}{40} + \frac{15}{96}x^7\sum_{k=0}^{N-3}x^{2k} - \frac{45}{96}\sum_{k=3}^{N}\frac{x^{2k+1}}{2k+1} \\
&\leq \frac{3x^5}{40} + \frac{15}{96}\cdot\frac{x^7}{1-x^2} - \frac{45}{96}\cdot\frac{x^7}{7} \;=\; \frac{3x^5}{40} + \frac{5x^7}{32}\left(\frac{1}{1-x^2} - \frac{3}{7}\right),
\end{aligned}$$

$$\begin{aligned}
B &= \sum_{k=2}^{N}(k^2-1)\,q_k\,\frac{x^{2k+1}}{2k+1} \;\leq\; \frac{9}{8}\cdot\frac{x^5}{5} + \frac{15}{48}\sum_{k=3}^{N}(k^2-1)\frac{x^{2k+1}}{2k+1} \\
&\leq \frac{9x^5}{40} + \frac{15}{96}\sum_{k=3}^{N}(2k+1)k\frac{x^{2k+1}}{2k+1} \;-\; \frac{15}{96}\sum_{k=3}^{N}k\,\frac{x^{2k+1}}{2k+1} \;-\; \frac{15}{48}\sum_{k=3}^{N}\frac{x^{2k+1}}{2k+1} \\
&\leq \frac{9x^5}{40} + \frac{15x^7}{192}\cdot\frac{6-4x^2}{(1-x^2)^2} - \frac{25}{224}x^7 \;=\; \frac{9x^5}{40} + \frac{5x^7}{32}\left(\frac{3-x^2}{(1-x^2)^2} - \frac{5}{7}\right).
\end{aligned}$$

Für den relativen Auswertefehler $\varepsilon_P = \frac{\Delta_P}{r_0 x}$ kann r_0 nach unten durch die ersten drei Terme abgeschätzt werden, $n(x) := 1 + \frac{x^2}{6} + \frac{3}{40}x^4$. Dann hat man

$$\begin{aligned}
\varepsilon_P &\in E_\ell\left[1 + \left(1 - \frac{1}{r_0}\right)(1+E_\ell) + (2+E_\ell)\frac{1}{r_0}\cdot\frac{A}{x} + E_\ell(2+E_\ell)\frac{1}{r_0}\cdot\frac{B}{x}\right] \\
&\subseteq E_\ell\left[1 + \left(1 - \frac{x}{\arcsin(x)}\right)(1+E_\ell) + (2+E_\ell)\,\frac{1}{n(x)}\cdot\left(\frac{3x^4}{40} + \frac{5x^6}{32}\left(\frac{1}{1-x^2} - \frac{3}{7}\right)\right)\right. \\
&\qquad \left. + E_\ell(2+E_\ell)\,\frac{1}{n(x)}\cdot\left(\frac{9x^4}{40} + \frac{5x^6}{32}\left(\frac{3-x^2}{(1-x^2)^2} - \frac{5}{7}\right)\right)\right].
\end{aligned}$$

Die Faktoren der Fehlerterme sind auf $[0,1]$ monoton steigend. Für $x \leq 0.19$ (und $b = 10$, $\ell \geq 5$) ergibt sich damit

$$\begin{aligned}
\varepsilon_P &\in E_\ell\,[1 + 0.0060799838 + 2.0320094 \cdot 10^{-4} + 6.1908880 \cdot 10^{-8}] \\
&\subseteq 1.006284 \cdot E_\ell.
\end{aligned}$$

Die Plausibilitätskontrolle auf dem Rechner ergibt einen Faktor 1.006283. Wir benötigen also zwei Schutzziffern. Für die Schranke für den Approximationsfehler erhalten wir damit

$$|\varepsilon_{app}| \leq 8.9928 \cdot 10^{-p-1}.$$

4.8.4 Approximationsfehler

Für den relativen Approximationsfehler bei Abbruch nach $N+1$ Termen gilt

$$\begin{aligned}
|\varepsilon_{app}| &= \sum_{k=N+1}^{\infty} c_k x^{2k+1} \Big/ \sum_{k=0}^{\infty} c_k x^{2k+1} \;=\; \sum_{k=N+1}^{\infty} c_k x^{2k} \Big/ \sum_{k=0}^{\infty} c_k x^{2k} \;\leq\; c_{N+1}\sum_{k=N+1}^{\infty} x^{2k} \Big/ 1 \\
&= c_{N+1}\cdot x^{2N+2}\cdot\frac{1}{1-x^2},
\end{aligned}$$

d.h. wir brauchen ein kleines N mit

$$c_{N+1} \cdot x^{2N+2} \cdot \frac{1}{1-x^2} \;\le\; 8.9928 \cdot 10^{-p-1},$$

implementiert als (num=Zähler, den=Nenner):

$$num(c_{N+1}) \cdot x^{2N+2} \;\le\; den(c_{N+1}) \cdot [\,(1-x^2) \cdot 8.9928 \cdot 10^{-p-1}\,],$$

wobei ein Startwert aus folgender Tabelle interpoliert wird:

p	$j=1$	2	3	4	5	6	7	8	9	10
3	3	3	3	3	3	3	3	3	3	3
10	3	3	3	3	4	4	4	5	5	5
33	9	10	12	13	15	16	17	18	19	21
100	28	34	39	43	47	51	55	59	63	67
330	94	114	131	145	159	172	186	199	212	225
1000	289	350	400	444	486	527	568	608	648	689

4.8.5 Berechnung für sehr kleine Argumente

Für betragskleine x wird $\arcsin(x) \gtrapprox x$ verwendet. Genauer gilt (für $0 < x \le 10^{-3}$):

$$\begin{aligned}\frac{\arcsin(x)-x}{x} &= \frac{\frac{1}{6}x^3 + \frac{3}{40}x^5 + \frac{5}{112}x^7}{x} \;\le\; \frac{1}{6}x^2\,(1+x^2+x^4+\ldots)\\ &= \frac{1}{6(1-x^2)}\,x^2 \;\le\; 0.166667\,x^2 \quad \overset{!}{\le}\; b^{-p},\end{aligned}$$

bzw. schärfer $x \le 2.449\,b^{-p/2}$, was genau wie beim Sinus schnellgetestet wird. Wenn der Test positiv ausfällt, wird als Einschließung $[x\,,\,x(1+b^{-p})]$ verwendet.

4.8.6 Berechnung über den Arcustangens

Hier ähnelt unser Vorgehen wieder dem in [Steins]. Für $\bar{x} < x < 1$ (mit momentan $\bar{x} = 0.19$) soll $\arcsin(x)$ als $\arctan(y)$ berechnet werden mit (exakt bzw. gestört)

$$y := \frac{x}{\sqrt{1-x^2}}, \qquad \tilde{y} := y \cdot \frac{1+\varepsilon_/}{1+\varepsilon_\surd}.$$

Der relative Fehler ε_y ergibt sich aus

$$\frac{1+\varepsilon_/}{1+\varepsilon_\surd} = (1+\varepsilon_/)(1-\varepsilon_\surd) + \varepsilon_\surd^2 \cdot \frac{1+\varepsilon_/}{1+\varepsilon_\surd} = 1+\varepsilon_/ - \varepsilon_\surd - \varepsilon_/\varepsilon_\surd + \varepsilon_\surd^2 \cdot \frac{1+\varepsilon_/}{1+\varepsilon_\surd},$$

also

$$\varepsilon_y = \varepsilon_/ - \varepsilon_\surd - \varepsilon_/\varepsilon_\surd + \varepsilon_\surd^2 \cdot \frac{1+\varepsilon_/}{1+\varepsilon_\surd}.$$

Aus dem Mittelwertsatz ergibt sich ($|x| < 1$):

$$\arctan\tilde{y} = \arctan(y + y\varepsilon_y) = (\arctan y) + \frac{y}{1+\xi^2}\,\varepsilon_y$$

mit einem $\xi \in \operatorname{conv}\{y, y(1+\varepsilon_y)\}$.

Damit erhalten wir

$$\begin{aligned}
|\varepsilon_{\arcsin}| &= \left|\frac{\widetilde{\arctan}\,\tilde{y} - \arctan y}{\arctan y}\right| = \left|\frac{(\arctan \tilde{y})(1+\varepsilon_a) - \arctan y}{\arctan y}\right| \\
&= \left|\frac{\varepsilon_a \cdot \arctan y + \frac{y}{1+\xi^2}(1+\varepsilon_a)\varepsilon_y}{\arctan y}\right| \leq |\varepsilon_a| + \frac{y}{\arctan y} \cdot \frac{1}{1+\xi^2}(1+\varepsilon_a)|\varepsilon_y| \\
&\leq |\varepsilon_a| + \frac{x}{\sqrt{1-x^2}\cdot \arcsin(x)} \cdot \frac{1}{1+y^2(1-|\varepsilon_y|)^2}(1+\varepsilon_a)|\varepsilon_y| \\
&= |\varepsilon_a| + \frac{x}{\sqrt{1-x^2}\cdot \arcsin(x)} \cdot \frac{1-x^2}{1+x^2(-2|\varepsilon_y|+\varepsilon_y^2)}(1+\varepsilon_a)|\varepsilon_y| \\
&\leq |\varepsilon_a| + \underbrace{\frac{x\sqrt{1-x^2}}{\arcsin(x)}}_{\leq 0.97582} \cdot \underbrace{\frac{1+|\varepsilon_a|}{1-x^2|\varepsilon_y|(2-|\varepsilon_y|)}}_{\leq 0.99981} |\varepsilon_y|
\end{aligned}$$

Der erste unterklammerte Term ist in $[0,1]$ monoton fallend in x und nimmt sein Maximum daher in $\bar{x}$ an. Beim zweiten ergibt sich die Schranke aus $x < 1$ und der Minimalbedingung $|\varepsilon_a|, |\varepsilon_y| \leq 2 \cdot 10^{-4}$ (für $b = 10$).

Aus der Forderung $|\varepsilon_{\arcsin}| \leq 10^{-p}$ ergibt sich zunächst $|\varepsilon_a| \leq 10^{-p}$ und damit zwei Schutzziffern für den Arcustangens, damit hat man dann $|\varepsilon_a| \leq 2 \cdot 10^{-p-1}$. Der restliche Freiraum kann für ε_y verwendet werden, also ist zu fordern

$$\begin{aligned}
|\varepsilon_y| &\leq \frac{10^{-p} - 2 \cdot 10^{-p-1}}{0.97563}, \\
\text{bzw.} \quad |\varepsilon_y| &\leq 8.19982 \cdot 10^{-p-1}.
\end{aligned}$$

Aus der oben angegebenen Beziehung für ε_y erhält man nun

$$\begin{aligned}
|\varepsilon_y| &\leq |\varepsilon_/| + |\varepsilon_\surd| \cdot \left(1 + |\varepsilon_/| + |\varepsilon_\surd| \cdot \frac{1+|\varepsilon_/|}{1-|\varepsilon_\surd|}\right) \\
&\leq 10^{-\ell+1} + 10^{-\ell+1} \cdot \left(1 + 10^{-4} + 10^{-4} \cdot \frac{1+10^{-4}}{1-\cdot 10^{-4}}\right) \\
&\leq 2.00021 \cdot 10^{-\ell+1} \overset{!}{\leq} 8.19982 \cdot 10^{-p-1},
\end{aligned}$$

woraus sich für die Mindeststellenzahl ℓ für Division und Quadratwurzel $\ell \geq p + 1.3873$ ergibt, also auch hier zwei Schutzziffern.

Insgesamt erhält man außerdem

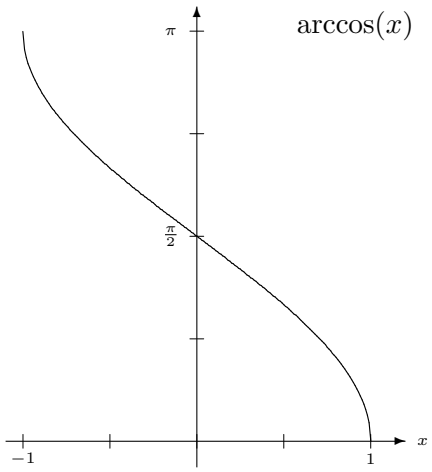
$$|\varepsilon_{\arcsin}| \leq 3.9515 \cdot 10^{-p-1}.$$

Wir benutzen also keine Intervallversion des Arcustangens, sondern berechnen nur eine Oberschranke aus einer Oberschranke für y. Eine Unterschranke erhalten wir über diese Abschätzung.

4.8.7 Intervallversion

Da auch arcsin monoton wächst, ergibt sich die intervallmäßige Implementation kanonisch.

4.9 Der Arcuscosinus



Der Arcuscosinus wird aufgefasst als Umkehrfunktion des Cosinus auf $[0, \pi]$. Er ist natürlich ebenfalls nur für $|x| \leq 1$ definiert. Für $x = 1$ geben wir direkt 0, für $x = 0$ eine Einschließung von $\frac{\pi}{2}$, für $x = -1$ eine Einschließung von π zurück.

Wegen $\arccos(x) = \frac{\pi}{2} - \arcsin(x)$ ist die Potenzreihenentwicklung die negative des Arcussinus mit dem zusätzlichen Summanden $\frac{\pi}{2}$, und man kann die Berechnung auf die Arcussinus-Routine zurückführen.

Alternativ lassen sich (wie in [Steins]) folgende Identitäten benutzen:

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

Wir benutzen den Arcussinus in dem Bereich $[-\bar{x}, +\bar{x}]$, in dem er über eine Potenzreihe direkt berechnet wird (in der aktuellen Version mit $\bar{x} = 0.19$). Dort wird zusätzlich eine Einschließung von $\frac{\pi}{2}$ benötigt, von der aber (außer beim ersten Mal) eine gute Chance besteht, dass sie bereits in (ggf. fast) der gewünschten Genauigkeit vorliegt.

Außerhalb dieses Bereichs benutzen wir die passende der oben angegebenen Beziehungen über den Arcustangens. (Der Arcussinus würde dort ja ebenfalls auf den Arcustangens zurückgeführt.)

Für die Fehlerabschätzung unterscheiden wir vier Fälle:

a) $0 < x \leq \bar{x}$ (Berechnung über arcsin).

Hier haben wir

$$\begin{aligned} |\varepsilon_{\arccos}| &= \left|\frac{(\frac{\pi}{2}(1+\varepsilon_1) - \arcsin(x)(1+\varepsilon_2))(1+\varepsilon_3) - (\frac{\pi}{2} - \arcsin(x))}{\arccos(x)}\right| \\ &= \left|\frac{\frac{\pi}{2}(\varepsilon_1 + \varepsilon_3 + \varepsilon_1\varepsilon_3) - \arcsin(x)(\varepsilon_2 + \varepsilon_3 + \varepsilon_2\varepsilon_3)}{\arccos(x)}\right|. \\ &\leq \frac{\frac{\pi}{2}(|\varepsilon_1| + |\varepsilon_3|(1+\varepsilon_1)) + \arcsin(x)(|\varepsilon_2| + |\varepsilon_3|(1+\varepsilon_2))}{\arccos(x)} \\ &= \frac{\frac{\pi}{2}}{\arccos(x)}|\varepsilon_1| + \frac{\arcsin(x)}{\arccos(x)}|\varepsilon_2| + \frac{\frac{\pi}{2}(1+\varepsilon_1) + \arcsin(x)(1+\varepsilon_2)}{\arccos(x)}|\varepsilon_3| \\ &\leq 1.1386 \cdot 10^{-\ell+1} + 0.1386 \cdot 2 \cdot 10^{-\ell+1} + 1.2773 \cdot 10^{-\ell+1} \overset{!}{\leq} 10^{-p}, \end{aligned}$$

bzw. $\ell \geq p + 1.4303$, also reichen zwei Schutzziffern (in allen beteiligten Operationen) aus. Dadurch ergibt sich

$$|\varepsilon_{\arccos}| \leq 2.6931 \cdot 10^{-p-1}.$$

Wir berechnen eine obere Grenze und mit Hilfe dieser Abschätzung die untere Grenze einer Einschließung.

b) $-\bar{x} \leq x < 0$ (Auch Berechnung über arcsin).

Man erhält zunächst dieselben Abschätzungen wie in a), durch das unterschiedliche Vorzeichen von $\arcsin(x)$ aber letztlich

$$|\varepsilon_{\arccos}| \;\leq\; |\varepsilon_1| + 0.1085\,|\varepsilon_2| + 1.00013\,|\varepsilon_3| \;\leq\; 2.21712 \cdot 10^{-\ell+1} \;\overset{!}{\leq}\; b^{-p}.$$

Man hat also noch etwas günstiger zu erfüllen $\ell \leq p + 1.4303$, also auch jeweils zwei Schutzziffern.

c) $\bar{x} < x < 1$ (Berechnung über arctan ohne π).

Hier ist das Vorgehen ähnlich wie bei arcsin, wobei wir den Vorteil haben, dass $x = 0$ nicht in unserem Auswertungsbereich liegt. Sei analog (exakte Subtraktion und Quadrierung)

$$y := \frac{\sqrt{1-x^2}}{x}, \qquad \tilde{y} := y \cdot (1+\varepsilon_/)(1+\varepsilon_{\sqrt{}}),$$

dann ergibt sich diesmal der relative Fehler bei der Berechnung von y zu

$$\varepsilon_y = \varepsilon_{\sqrt{}} + \varepsilon_/ + \varepsilon_{\sqrt{}}\varepsilon_/.$$

Mit den Bezeichnungen aus arcsin ergibt sich zunächst völlig analog

$$\begin{aligned}
|\varepsilon_{\arccos}| &= |\varepsilon_a| + \frac{1}{1+\xi^2} \cdot \frac{y}{\arctan y} \cdot (1+\varepsilon_a)\,|\varepsilon_y| \\
&= |\varepsilon_a| + \frac{1}{1+\xi^2} \cdot \frac{\sqrt{1-x^2}}{x\,\arccos(x)}\,(1+\varepsilon_a)\,|\varepsilon_y| \\
&\leq |\varepsilon_a| + \frac{\sqrt{1-x^2}}{x \cdot \arccos(x)} \cdot \frac{1+\varepsilon_a}{1+y^2(1-|\varepsilon_y|)^2}\,|\varepsilon_y| \\
&= |\varepsilon_a| + \frac{x\sqrt{1-x^2}}{\arccos(x)} \cdot \frac{1+\varepsilon_a}{1-(1-x^2)|\varepsilon_y|(2-|\varepsilon_y|)}\,|\varepsilon_y|
\end{aligned}$$

Für $b = 10$ und mit $\bar{x} = 0.19$ erhält man aus Letzterem die Forderung

$$|\varepsilon_a| + 1.00059\,|\varepsilon_y| \overset{!}{\leq} 10^{-p},$$

was zunächst natürlich zwei Schutzziffern für die Arcustangens-Berechnung bedeutet, und damit $|\varepsilon_a| \leq 2 \cdot 10^{-p-1}$. Damit bleibt

$$\begin{aligned}
1.00059 \cdot |\varepsilon_y| &\overset{!}{\leq} 10^{-p} - 2 \cdot 10^{-p-1}, \\
\text{bzw.} \qquad |\varepsilon_y| &\overset{!}{\leq} 7.9952 \cdot 10^{-p-1}.
\end{aligned}$$

Aus der Beziehung für ε_y von oben erhält man nun Folgendes:

$$\begin{aligned} |\varepsilon_y| &\leq |\varepsilon_{\surd}|(1+|\varepsilon_{/}|)+|\varepsilon_{/}| \leq 2\cdot 10^{-\ell+1}\,(1+10^{-4})+10^{-\ell+1} \\ &= 3.0002\cdot 10^{-\ell+1} \overset{!}{\leq} 7.9952\cdot 10^{-p-1}, \end{aligned}$$

bzw. $\ell \geq p+1.5744$, also reichen auch für Quadratwurzel und Division zwei Schutzziffern.

Insgesamt erhält man dann

$$|\varepsilon_{\arccos}| \leq 5.002\cdot 10^{-p-1}.$$

Um keine Intervallversion des Arcustangens zu bemühen, berechnen wir nur eine Unterschranke $\tilde{y}$ von y, daraus eine Oberschranke von $\arctan\tilde{y}$ und aus dieser Fehlerabschätzung eine Unterschranke des Arcuscosinus.

d) $-1 < x < -\bar{x}$ (Berechnung über arctan mit π).

Die Bezeichnungen y, $\tilde{y}$, ε_y, ξ sowie die Beziehung von zu $\arctan\tilde{y}$ können natürlich aus Teil c) übernommen werden. Mit der zusätzlichen Addition von π haben wir hier

$$\begin{aligned} |\varepsilon_{\arccos}| &= \left|\frac{(\pi(1+\varepsilon_\pi)+(\arctan(y)+\frac{\varepsilon_y y}{1+\xi^2})(1+\varepsilon_a))(1+\varepsilon_+)-\pi-\arctan(y)}{\arccos(x)}\right| \\ &\leq \left|\frac{\pi(\varepsilon_\pi+\varepsilon_+(1+\varepsilon_\pi)+\arctan(y)(\varepsilon_a+\varepsilon_+(1+\varepsilon_a))+\frac{\varepsilon_y y}{1+\xi^2}(1+\varepsilon_a)(1+\varepsilon_+)}{\arccos(x)}\right| \\ &\leq \underbrace{\frac{\pi}{\arccos(x)}}_{\leq 1.7831}|\varepsilon_\pi| + \underbrace{\frac{\pi(1+\varepsilon_\pi)+|\arctan(y)|(1+\varepsilon_a)}{\arccos(x)}}_{\leq 2.5664}|\varepsilon_+| + \underbrace{\frac{|\arctan(y)|}{\arccos(x)}}_{\leq 0.7831}|\varepsilon_a| \\ &\quad + \underbrace{\frac{|y|}{1+y^2(1-|\varepsilon_y|)^2}(1+\varepsilon_a)(1+\varepsilon_+)}_{\leq 0.5003}|\varepsilon_y|. \end{aligned}$$

Die Unterklammerungen gelten für $b = 10$. Da die ersten beiden Faktoren etwas groß sind, berechnen wir die schnellen Operationen π (meist schon vorberechnet) und die Addition mit einer Stelle mehr als den Arcustangens und y (diese mit ℓ Stellen). Dann erhält man als Forderung:

$$\begin{aligned} |\varepsilon_{\arccos}| &\leq 0.17831\cdot 10^{-\ell+1}+0.25664\cdot 10^{-\ell+1}+0.7831\cdot 2\cdot 10^{-\ell+1} \\ &\quad +0.5003\cdot 2\cdot 10^{-\ell+1} = 3.00175\cdot 10^{-\ell+1} \overset{!}{\leq} 10^{-p}. \end{aligned}$$

Also ist $\ell := p+2$ erforderlich, d.h. zwei Schutzziffern bei arctan und y, drei bei π und der Addition. Die zusätzlichen Toleranzen ergeben für y als Forderung $|\varepsilon_y| \leq 1.5988\cdot 10^{-p}$, also

$$|\varepsilon_y| = |\varepsilon_{\surd}| + |\varepsilon_{/}|(1+\varepsilon_{\surd}) \leq 10^{-m+1} + (1+10^{-4})\cdot 10^{-m+1} \overset{!}{\leq} 1.5988\cdot 10^{-p},$$

bei m-stelliger Division und Wurzelberechnung, also braucht man zwei Schutzziffern, $m := p + 2$, so dass $|\varepsilon_y| \leq 0.20001 \cdot 10^{-p}$. Insgesamt erhält man dann

$$|\varepsilon_{\arccos}| \leq 3.0019 \cdot 10^{-p-1},$$

so dass wir wieder nur eine Oberschranke zu berechnen haben und hieraus eine Unterschranke erhalten.

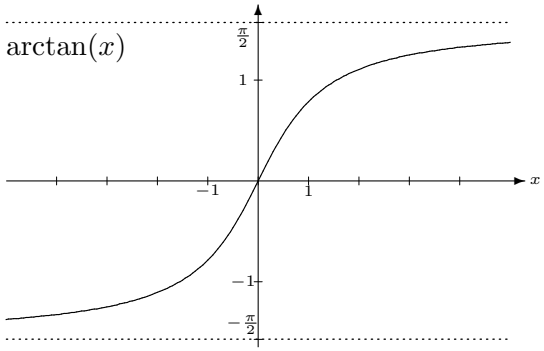
4.9.1 Intervallversion

Auch hier ist nicht viel zu tun, da der Arcuscosinus auf seinem Definitionsbreich monoton fallend ist.

4.10 Der Arcustangens

Der Arcustangens, zu verstehen als die Umkehrfunktion des Tangens auf $]-\frac{\pi}{2}, +\frac{\pi}{2}[$, wird für $|x| < 1$ durch folgende Potenzreihe dargestellt:

$$\begin{aligned} \arctan(x) &= \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{2k+1} \\ &= x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + - \dots \end{aligned}$$



Wegen $\arctan(-x) = -\arctan(x)$ brauchen wir wieder nur $x > 0$ zu betrachten.

4.10.1 Argumentreduktion

Für $x \geq 1$ ist eine Argumentreduktion erforderlich (sie benutzt eine Symmetrie). Da die Reihe nicht sehr schnell konvergiert, wird über ein Additionstheorem auch unterhalb von 1 eine weitere Reduktion vorgenommen. Für $x = 1$ gibt unsere Routine direkt eine genügend genaue Einschließung von $\frac{\pi}{4}$ zurück.

a) Für $x > 1$ erlaubt die Identität

$$\arctan(x) = \frac{\pi}{2} - \arctan\frac{1}{x} \quad \text{für } x > 0$$

die Reduktion in das Intervall $]0, 1[$.

Für die Berechnung des exakten Arcustangens für das gestörte Argument $\frac{1}{x}(1+\varepsilon_/)$ gilt (mit dem Mittelwertsatz):

$$\arctan\left(\frac{1}{x}(1+\varepsilon_/)\right) = \arctan\left(\frac{1}{x}\right) + \frac{1}{1+\frac{\xi^2}{x^2}} \cdot \frac{1}{x} \cdot \varepsilon_/ = \arctan\left(\frac{1}{x}\right) + \frac{1}{x+\frac{1}{x}\xi^2} \cdot \varepsilon_/$$

mit einem $\xi \in \operatorname{conv}\{1, 1+\varepsilon_/\}$. Damit erhalten wir für den relativen Gesamtfehler dieser Reduktion (exakte Subtraktion, sehr große Argumente werden noch gesondert behandelt):

$$\begin{aligned}\varepsilon_{\arctan} &= \frac{1}{\arctan(x)}\left(\frac{\pi}{2}(1+\varepsilon_\pi) - \left(\arctan\left(\frac{1}{x}\right) + \frac{1}{x+\frac{1}{x}\xi^2}\varepsilon_/\right)(1+\varepsilon_a) - \frac{\pi}{2} + \arctan\left(\frac{1}{x}\right)\right) \\ &= \frac{\frac{\pi}{2}}{\arctan(x)}\cdot\varepsilon_\pi - \frac{\arctan\frac{1}{x}}{\arctan(x)}\cdot\varepsilon_a - \frac{\frac{1}{x+\frac{1}{x}\xi^2}}{\arctan(x)}\cdot\varepsilon_/\,(1+\varepsilon_a),\end{aligned}$$

d.h.

$$\begin{aligned}|\varepsilon_{\arctan}| &\le \frac{\frac{\pi}{2}}{\arctan(x)}|\varepsilon_\pi| + \frac{\arctan\frac{1}{x}}{\arctan(x)}|\varepsilon_a| + \frac{1}{(x+\frac{1}{x}(1-|\varepsilon_/|)^2)\arctan(x)}|\varepsilon_/|\,(1+\varepsilon_a) \\ &\le \underbrace{\frac{\frac{\pi}{2}}{\arctan 1}}_{=2}\cdot|\varepsilon_\pi| + \underbrace{\frac{\arctan 1}{\arctan 1}}_{=1}\cdot|\varepsilon_a| + \frac{1}{\underbrace{(x+\frac{1}{x}(1-2b^{-4})^2)}_{\ge 1.766}\arctan(x)}\cdot|\varepsilon_/|\,(1+\varepsilon_a).\end{aligned}$$

Für $b = 10$ gilt bei der letzten Klammer sogar ≥ 1.9996 und damit insgesamt:

$$|\varepsilon_{\arctan}| \;\le\; 2\cdot|\varepsilon_\pi| + |\varepsilon_a| + 0.63688\cdot|\varepsilon_/| \;\le\; 4.6369\cdot 10^{-\ell+1} \;\overset{!}{\le}\; 10^{-p}$$

bei Berechnung mit ℓ Stellen bei $\frac{\pi}{2}$, dem Arcustangens und der Division, also sind zwei Schutzziffern zu fordern. Danach ist $|\varepsilon_{\arctan}| \le 0.46369\cdot 10^p$, was nach Berechnung einer Obergrenze für die Bestimmung eine Untergrenze verwendet wird.

b) Aus dem Additionstheorem des Arcustangens

$$\arctan(x_1) + \arctan(x_2) = \arctan\left(\frac{x_1+x_2}{1-x_1x_2}\right)$$

folgt

$$\begin{aligned}2\arctan y &= \arctan\frac{2y}{1-y^2}, \\ \arctan(x) &= 2\arctan\frac{\sqrt{1+x^2}-1}{x} = 2\arctan\frac{x}{\sqrt{1+x^2}+1}.\end{aligned}$$

Die Transformation $x \mapsto \frac{x}{1+\sqrt{1+x^2}}$ kann nun mehrere Male angewandt werden, um das Argument so weit zu verkleinern, dass es für die Berechnung über die Potenzreihe geeignet ist. Nach einer Transformation ist das Argument $x < \sqrt{2}-1 \approx 0.414214$, nach zwei Transformationen $x < 0.198913$. Es ist aber zu beachten, dass eine Quadratwurzel vorkommt, die selbst einen gewissen Aufwand mitbringt.

Damit für $x = 0.9$ das Restglied der Taylor-Entwicklung (siehe die spätere Abschätzung) etwa 16-stellige Genauigkeit erlaubt, werden z.B. 154 Glieder der Potenzreihe, für 50-stellige Genauigkeit 519 Glieder benötigt. Für das einfach transformierte Argument ≈ 0.3837 braucht man 17 bzw. 57 Glieder, für das zweifach transformierte ≈ 0.1853

noch 10 bzw. 32 Glieder. *Eine* Argumentreduktion lohnt daher immer, mehr als zwei nicht. In unserer Implementation wird nur eine durchgeführt.

Wir betrachten zunächst den Fehler, der bei der Berechnung des transformierten Arguments y gemacht wird. Das Quadrieren und die beiden Additionen werden exakt durchgeführt. Dann haben wir

$$\tilde{y} = \frac{x(1+\varepsilon_/)}{1+(1+\varepsilon_{\surd})\sqrt{1+x^2}},$$

$$|\varepsilon_y| \leq \frac{1+\sqrt{1+x^2}}{1+(1+\varepsilon_{\surd})\sqrt{1+x^2}}\,|\varepsilon_/| + \frac{\sqrt{1+x^2}}{1+(1+\varepsilon_{\surd})\sqrt{1+x^2}}\,|\varepsilon_{\surd}|.$$

Die Faktoren der beiden Fehler sind für $x \geq 0$ monoton steigend. Für $x < 1$ gilt dann

$$|\varepsilon_y| \leq 1.00006\varepsilon_/ + 0.5859\varepsilon_{\surd}.$$

Für den Gesamtfehler gilt (wieder unter Anwendung des Mittelwertsatzes mit einem $\xi \in \mathrm{conv}\{1, 1+\varepsilon_y\}$):

$$\begin{aligned} |\varepsilon_{\arctan}| &= \left|\frac{(\arctan(y) + \frac{y}{1+y^2\xi^2}\varepsilon_y)(1+\varepsilon_a) - \arctan(y)}{\arctan(y)}\right| \\ &\leq |\varepsilon_a| + (1+\varepsilon_a)\,\frac{y}{(1+y^2)\arctan(y)}\,|\varepsilon_y|. \end{aligned}$$

Der Bruch ist für $x > 0$ monoton fallend und nach einer Argumentreduktion ($x < 0.41422$) kleiner als 0.9005, nach zwei Reduktionen ($x < 0.198914$) kleiner als 0.9747. In jedem Fall werden also zwei Schutzziffern benötigt. Im ersten Fall ist dann $|\varepsilon_{\arctan}| \leq 3.429 \cdot 10^{-p-1}$.

4.10.2 Auswertung der Potenzreihe

Da keine Fakultäten oder ähnliche Terme vorkommen, die beim Ausklammern einbezogen werden könnten, benutzen wir hier das normale Horner-Verfahren.

$$\begin{aligned} P_N(x) &= x\left(1 - \frac{u}{3} + \frac{u^2}{5} - \frac{u^3}{7} + - \ldots + (-1)^N \frac{u^N}{2N+1}\right) \\ &= x\left(1 + u\left(-\frac{1}{3} + u\left(\frac{1}{5} + u\left(-\frac{1}{7} + \ldots + u \cdot \frac{(-1)^N}{2N+1}\right)\right)\ldots\right)\right) \end{aligned}$$

Das Verfahren ergibt sich wie folgt:

$$\begin{aligned} &a) \quad r_N := \frac{(-1)^N}{2N+1}, \\ &b) \quad r_k := \frac{(-1)^k}{2k+1} + r_{k+1} \cdot u \qquad \text{für } k = N-1, \ldots, 1, \\ &c) \quad r_0 := 1 + u \cdot r_1 \\ &\quad \Rightarrow \quad P(x) = x \cdot r_0. \end{aligned}$$

Für gerades N ist wiederum eine obere, für ungerades N eine untere Schranke zu berechnen.

4.10.3 Auswertefehler beim Horner-Verfahren

Es ergibt sich für $k = 1, \ldots, N$:

$$r_k = (-1)^k \sum_{i=0}^{N-k} (-1)^i \frac{x^{2i}}{2k+2i+1}.$$

Da wiederum das Vorzeichen der Summanden alterniert und ihre Beträge streng monoton fallen, gilt

$$|r_k| \;\le\; \frac{1}{2k+1} \;=\; a_k.$$

Damit liefert uns Korollar 5 folgende Aussage (mit $x > 0$):

$$\begin{aligned}
\Delta_P \;&\in E_\ell \Big[\; |r_0 x| \;+\; |r_1 x^3|(1+E_\ell)(2+E_\ell) \\
&\quad + (2+E_\ell) \sum_{k=2}^{N} \frac{x^{2k+1}}{2k+1} \;+\; (1+E_\ell)^2 \sum_{k=1}^{N-1} \frac{x^{2k+1}}{2k+1} \\
&\quad + 2E_\ell(2+E_\ell) \sum_{k=2}^{N} k\,\frac{x^{2k+1}}{2k+1} \;+\; 2E_\ell(1+E_\ell)^2 \sum_{k=1}^{N-1} k\,\frac{x^{2k+1}}{2k+1} \;\Big] \\
&\subseteq E_\ell \Big[\; |r_0 x| \;+\; |r_1 x^3|(1+E_\ell)(2+E_\ell) + \frac{x^3}{3}(1+E_\ell)^2(1+2E_\ell) \\
&\quad + [2+E_\ell+(1+E_\ell)^2] \underbrace{\sum_{k=2}^{N} \frac{x^{2k+1}}{2k+1}}_{A} + 2E_\ell[2+E_\ell+(1+E_\ell)^2] \underbrace{\sum_{k=2}^{N} k\,\frac{x^{2k+1}}{2k+1}}_{B} \;\Big].
\end{aligned}$$

Für die beiden Summen A und B erhalten wir:

$$\begin{aligned}
A \;&=\; \sum_{k=0}^{N} \frac{x^{2k+1}}{2k+1} - x - \frac{x^3}{3} \;\le\; \operatorname{artanh} x - x - \frac{x^3}{3}, \\
B \;&=\; \sum_{k=2}^{N} k\,\frac{x^{2k+1}}{2k+1} \;=\; \frac{1}{2}\sum_{k=2}^{N}(2k+1)\frac{x^{2k+1}}{2k+1} - \frac{1}{2}\sum_{k=2}^{N}\frac{x^{2k+1}}{2k+1} \;=\; \frac{1}{2}\sum_{k=2}^{N} x^{2k+1} - \frac{1}{2}\sum_{k=2}^{N}\frac{x^{2k+1}}{2k+1} \\
&\le\; \frac{x^5}{2}\cdot\frac{1}{1-x^2} - \frac{1}{2}\cdot\frac{x^5}{5} \;=\; \frac{x^5}{2}\left(\frac{1}{1-x^2} - \frac{1}{5}\right),
\end{aligned}$$

wobei bei A die Areatangens-Reihe und bei B die geometrische Reihe verwendet wird.

Ferner ist $\frac{r_1 x^3}{r_0 x} = \frac{1}{r_0} - 1$, und in den Nennern wird r_0 durch die ersten 4 Terme nach unten abgeschätzt (es ist $N \ge 3$). So ergibt sich:

$$\begin{aligned}
\varepsilon_P \;=\; \tfrac{\Delta_P}{r_0 x} \;\in\; E_\ell \Big[\; 1 \;+\; &\left(\frac{1}{1-\frac{x^2}{3}+\frac{x^4}{5}-\frac{x^6}{7}} - 1\right)(1+E_\ell)(2+E_\ell) \\
&+ \frac{x^2}{3\left(1-\frac{x^2}{3}+\frac{x^4}{5}-\frac{x^6}{7}\right)}(1+E_\ell)^2(1+2E_\ell) + \frac{\frac{\operatorname{artanh} x}{x} - 1 - \frac{x^2}{3}}{1-\frac{x^2}{3}+\frac{x^4}{5}-\frac{x^6}{7}}\,(3+3E_\ell+E_\ell^2) \\
&+ \frac{x^4\left(\frac{1}{1-x^2}-\frac{1}{5}\right)}{1-\frac{x^2}{3}+\frac{x^4}{5}-\frac{x^6}{7}}\,E_\ell(3+3E_\ell+E_\ell^2) \;\Big].
\end{aligned}$$

Die Koeffizienten der Fehlerintervalle sind auf $[0,1[$ monoton steigend. Für den Bereich $0 < x < 0.190744$ (nach zwei Argumentreduktionen) gilt daher (mit $b = 10$, $\ell \geq 5$):

$$\begin{aligned}\varepsilon_P &\subseteq E_\ell\,[1 + 0.026091030 + 0.013366082 + 9.7938495 \cdot 10^{-4} + 4.002554 \cdot 10^{-7}] \\ &\subseteq 1.040437 \cdot E_\ell,\end{aligned}$$

und für den Bereich $0 < x < 0.414214$ (nach nur einer Argumentreduktion) gilt

$$\begin{aligned}\varepsilon_P &\subseteq E_\ell\,[1 + 0.10817202 + 0.060307846 + 0.021255737 + 9.3758396 \cdot 10^{-6}] \\ &\subseteq 1.189745 \cdot E_\ell,\end{aligned}$$

Die Rechnerkontrolle liefert die Werte 1.040427 bzw. 1.189614. In [Steins] wird für den zweiten Fall die Konstante 1.6184 errechnet. In jedem Fall sind zwei Schutzziffern erforderlich.

4.10.4 Approximationsfehler

Für den Approximationsfehler nach einer bzw. zwei Transformationen ist damit die erlaubte Schranke

$$|\varepsilon_{app}| \leq 8.9586 \cdot 10^{-p-1}, \qquad \text{bzw.} \qquad |\varepsilon_{app}| \leq 8.8092 \cdot 10^{-p-1},$$

Es gilt (bei $0 < x < 1$) für den relativen Approximationsfehler:

$$\begin{aligned}|\varepsilon_{app}| &= \left|\frac{P_N(x) - \arctan(x)}{\arctan(x)}\right| = \left|\sum_{k=N+1}^{\infty} (-1)^k \frac{x^{2k+1}}{2k+1}\right| \Big/ \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{2k+1} \\ &= \left[\frac{x^{2N+3}}{2N+3} - \left(\frac{x^{2N+5}}{2N+5} - \frac{x^{2N+7}}{2N+7}\right) - \left(\frac{x^{2N+9}}{2N+9} - \frac{x^{2N+11}}{2N+11}\right) - \ldots\right] \Big/ \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{2k+1} \\ &\leq \frac{x^{2N+3}}{2N+3} \Big/ \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{2k+1} \quad \leq \quad \frac{x^{2N+3}}{2N+3} \cdot \frac{1}{x - \frac{x^3}{3}} \\ &= \frac{x^{2N+2}}{2N+3} \cdot \frac{1}{1 - \frac{x^2}{3}}.\end{aligned}$$

Die Abschätzung aus [Krämer], Seite 50, die in [Steins] übernommen wird, enthält durch eine unnötige Überschätzung einen zusätzlichen Faktor $\frac{1}{1-x^2}$ (das alternierende Vorzeichen wird nicht ausgenutzt). Bei den kleinen Argumenten x, für die die Potenzreihe verwendet wird, macht er sich aber fast nicht bemerkbar.

Wir suchen (im kleineren Bereich) also ein möglichst kleines N, so dass

$$\frac{x^{2N+2}}{2N+3} \cdot \frac{3}{3 - x^2} \;\leq\; 8.9586 \cdot 10^{-p-1},$$

implementiert als

$$x^{2N+2} \;\leq\; (2N+2) \cdot \left[(3 - x^2) \cdot 2.9862 \cdot 10^{-p-1}\right].$$

Der Startwert für N wird aus folgender Tabelle interpoliert:

k	$j=1$	2	3	4	5	6	7	8	9	10
3	3	3	3	3	3	3	3	3	3	3
10	3	4	4	5	6	7	8	9	10	11
33	11	14	17	20	22	25	29	32	36	40
100	35	45	54	62	71	80	90	101	114	127
330	118	151	180	209	239	270	304	341	389	427
1000	360	460	550	638	728	824	926	1038	1162	1301

4.10.5 Auswertung für sehr kleine Argumente

Für kleine positive x wird $\arctan(x) \overset{<}{\approx} x$ ausgenutzt, also ggf. $[\,x(1-b^{-p})\,,\,x\,]$ als Einschließung berechnet. Für $0 < x \leq 10^{-3}$ gilt

$$\begin{aligned}\frac{x-\arctan x}{x} &= \frac{\frac{x^3}{3}-\frac{x^5}{5}+\frac{x^7}{7}-+\ldots}{x} \leq \frac{\frac{x^3}{3}\,(1+x^4+x^8+\ldots)}{x} \\ &= \frac{x^2}{3}\,\frac{1}{1-x^4} \leq 0.333334\,x^2 \overset{!}{\leq} b^{-p},\end{aligned}$$

woraus man gröber $x \leq 1.732 \cdot b^{-p}$ und damit denselben Test wie beim Sinus erhält.

4.10.6 Auswertung für sehr große Argumente

Über die Symmetrie erhält man den analogen Test für große Argumente. Für große $x > 1$ gilt $\arctan(x) \overset{<}{\approx} \frac{\pi}{2}$. Diese Sonderbehandlung braucht nicht aus numerischen oder implementatorischen Gründen durchgeführt zu werden; der Test ist aber wieder schnell durchführbar und erspart eine Reduktion durch Inversion auf den Fall sehr kleiner Argumente. Aus der Forderung (für $x \geq 10$)

$$\frac{\frac{\pi}{2}-\arctan x}{\frac{\pi}{2}} = \frac{\arctan(\frac{1}{x})}{\frac{\pi}{2}} \overset{!}{\leq} b^{-\ell}$$

ergibt sich etwas schärfer die Forderung

$$\arctan(\tfrac{1}{x}) \leq \tfrac{1}{x} \overset{!}{\leq} \tfrac{2}{\pi} \cdot \cdot b^{-\ell},$$

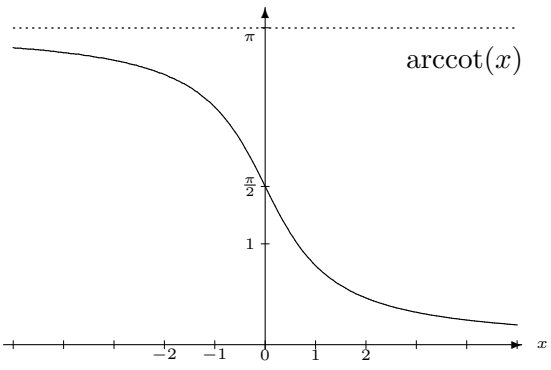
bzw. schärfer $x \geq 1.571 \cdot b^{\ell}$, was noch gröber durch $\operatorname{order}(x) \geq \ell + 1$ getestet wird. Eine kurze Betrachtung zeigt, dass minimal $\frac{\pi}{2}$ mit 3 Schutzziffern berechnet werden und $\ell = p+2$ gewählt muss. Wenn P die Obergrenze der Einschließung von $\frac{\pi}{2}$ ist, kann dann $[\operatorname{pred}_p(P), P]$ als Einschließung des Arcustangens zurückgegeben werden.

4.10.7 Intervallversion

Auch hier ist wegen der Monotonizität auf ganz $\mathbb{R}$ nichts zu tun.

4.11 Der Arcuscotangens

Der Arcuscotangens ist zu verstehen als die Umkehrfunktion des Cotangens auf dem Intervall $]0, \pi[$.



Wegen $\cot(x) = \tan(\frac{\pi}{2} - x)$ gilt immer

$$\operatorname{arccot}(x) = \tfrac{\pi}{2} - \arctan(x),$$

und für $x > 0$ außerdem

$$\operatorname{arccot}(x) = \arctan(\tfrac{1}{x}).$$

Entsprechend wird die Berechnung natürlich immer auf den Arcustangens zurückgeführt. Da aber auch bei dem intern bereits die Konstante π gebraucht wird (für $|x| \geq 1$), sollte die Formel so ausgewählt werden, dass insgesamt nur eine Konstante vorkommt. Daraus resultiert folgende Fallunterscheidung:

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

Das Argument des Arcustangens liegt so immer in $[0, 1]$, wo die Arcustangens-Routine (ggf. nach einer Argumentreduktion) die Potenzreihe verwendet, so dass dort keine weitere Konstante vorkommt.

a) $x \leq -1$. Es wird berechnet (mit den üblichen Bezeichnungen)

$$\widetilde{\operatorname{arccot}}\, x = \Big(\pi(1+\varepsilon_\pi) - \arctan\Big(\frac{1+\varepsilon_/}{|x|}\Big)(1+\varepsilon_{\arctan})\Big)(1+\varepsilon_-),$$

und es ist

$$\arctan\Big(\frac{1+\varepsilon_/}{|x|}\Big) = \arctan\Big(\frac{1}{|x|}\Big) + \frac{1}{1+\xi^2} \cdot \frac{\varepsilon_/}{|x|}$$

für ein $\xi \in \operatorname{conv}\{\frac{1}{|x|}, \frac{1}{|x|}(1+\varepsilon_/)\}$. Dann gilt

$$|\varepsilon_{\arctan}| = \left| \frac{\Big(\pi(1+\varepsilon_\pi) - (\arctan(\frac{1}{|x|}) + \frac{\varepsilon_/}{|x|(1+\xi^2)})(1+\varepsilon_{\arctan})\Big)(1+\varepsilon_-) - \pi + \arctan(\frac{1}{|x|})}{\pi - \arctan(\frac{1}{|x|})} \right|$$

$$\leq |\varepsilon_-| + \underbrace{\frac{\pi}{\pi - \arctan(\frac{1}{|x|})}}_{\leq\, 4/3} (1+\varepsilon_-)\,|\varepsilon_\pi| + \underbrace{\frac{\arctan(\frac{1}{|x|})}{\pi - \arctan(\frac{1}{|x|})}}_{\leq\, 1/3} (1+\varepsilon_-)\,|\varepsilon_{\arctan}|$$

$$+ \underbrace{\frac{1}{(|x| + \frac{1}{|x|}(1-|\varepsilon_/|)^2)(\pi - \arctan(\frac{1}{|x|}))}}_{\leq 0.2123} (1+\varepsilon_-)(1+\varepsilon_{\arctan})\,|\varepsilon_/|.$$

Wenn nun alle vorkommenden Operationen mit $\ell \geq 5$ Dezimalstellen durchgeführt werden, erhält man die Forderung $4.5462 \cdot 10^{-\ell+1} \leq 10^{-p}$, bzw. $\ell \geq p + 1.6844$, also sind wieder überall zwei Schutzziffern ausreichend, und dann ist (vor der Endrundung auf p Stellen)

$$|\varepsilon_{\mathrm{arccot}}| \;\leq\; 0.45462 \cdot 10^{-p}.$$

b) $-1 < x < 0$. Hier gilt für den relativen Fehler

$$\begin{aligned} |\varepsilon_{\mathrm{arccot}}| &= \left| \frac{\Big(\frac{\pi}{2}(1+\varepsilon_\pi) + \arctan|x|(1+\varepsilon_{\arctan})\Big)(1+\varepsilon_+) - \frac{\pi}{2} - \arctan|x|}{\frac{\pi}{2} + \arctan|x|} \right| \\ &\leq |\varepsilon_+| + \underbrace{\frac{\frac{\pi}{2}}{\frac{\pi}{2} + \arctan|x|}}_{\leq 1}(1+\varepsilon_+)\,|\varepsilon_\pi| + \underbrace{\frac{\arctan|x|}{\frac{\pi}{2} + \arctan|x|}}_{\leq\, 1/3}(1+\varepsilon_+)\,|\varepsilon_{\arctan}|. \end{aligned}$$

Für jeweils ℓ Dezimalstellen bei der Berechnung folgt als Bedingung

$$10^{-\ell+1} + 1.0001 \cdot 2 \cdot 10^{-\ell+1} + 0.6668 \cdot 10^{-\ell+1} = 3.6669 \cdot 10^{-\ell+1} \leq 10^{-p},$$

bzw. $\ell \geq p + 1.4261$, also wiederum 2 Schutzziffern. Anschließend hat man

$$|\varepsilon_{\mathrm{arccot}}| \;\leq\; 0.36669 \cdot 10^{-p}.$$

c) $0 < x < 1$. Die Fehlerabschätzungen sind identisch mit denen aus Teil b).

d) $x \geq 1$. Hier ergibt sich die Fehlerabschätzung wie folgt (ξ wie in a)):

$$\begin{aligned} |\varepsilon_{\mathrm{arccot}}| &= \left| \frac{(\arctan(\frac{1}{|x|}) + \frac{\varepsilon_/}{|x|(1+\xi^2)})(1+\varepsilon_{\arctan}) - \arctan(\frac{1}{|x|})}{\arctan(\frac{1}{|x|})} \right| \\ &\leq |\varepsilon_{\arctan}| + \frac{1}{x(1+\xi^2)\arctan(\frac{1}{x})}\,(1+\varepsilon_{\arctan})\,|\varepsilon_/| \\ &\leq |\varepsilon_{\arctan}| + \underbrace{\frac{1}{(x + \frac{1}{x}(1+|\varepsilon_/|)^2)\arctan(\frac{1}{x})}}_{\leq\, 1}\,(1+\varepsilon_{\arctan})\,|\varepsilon_/|. \end{aligned}$$

Bei ℓ Dezimalstellen für Division und Arcustangens erhält man die Forderung $3.0001 \cdot 10^{-\ell+1} \leq 10^{-p}$, bzw. $\ell \geq p + 1.478$, also zwei Schutzziffern, so dass danach gilt

$$|\varepsilon_{\mathrm{arccot}}| \;\leq\; 0.30001 \cdot 10^{-p}.$$

In den Fällen mit verrundetem Argument des Arcustangens wird nur jeweils eine Unterschranke des Arcustangens berechnet und die jeweils andere Grenze der Gesamteinschließung über die Fehlerabschätzung.

4.11.1 Intervallversion

Da der Arcuscotangens auf ganz $\mathbb{R}$ monoton fallend ist, ergibt sich die Intervallversion wieder kanonisch.

4.12 Der Areasinus

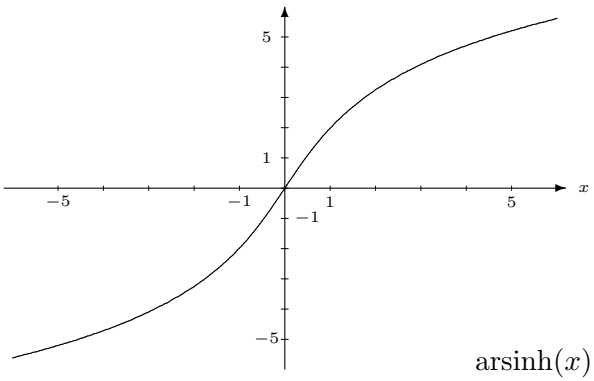
Der Areasinus ist die Umkehrfunktion des hyperbolischen Sinus. Für $x \geq 0$ erhält man durch Umstellen von $\sinh y = \frac{e^{2y}-1}{2e^y}$:

$$\operatorname{arsinh}(x) = \ln(x + \sqrt{1+x^2}).$$

Für große (positive) Argumente gilt sogar

$$\operatorname{arsinh}(x) \gtrapprox \ln(2x).$$

Es ist $\operatorname{arsinh}(-x) = -\operatorname{arsinh}(x)$, so dass wir bei unseren Betrachtungen $x > 0$ voraussetzen.



arsinh(x)

Für kleine Argumente wird arsinh besser über seine Potenzreihe berechnet (für $|x| < 1$ gegen $\operatorname{arsinh}(x)$ konvergent):

$$\begin{aligned}
\operatorname{arsinh} x &= \sum_{k=0}^{\infty} c_k \cdot x^{2k+1} \qquad \text{mit } c_k = (-1)^k \cdot \frac{1\cdot 3\cdot 5\cdot 7\cdot \ldots \cdot (2k-1)}{2\cdot 4\cdot 6\cdot 8\cdot \ldots \cdot (2k)} \cdot \frac{1}{2k+1} \\
&= x \;-\; \frac{1}{2}\cdot\frac{x^3}{3} \;+\; \frac{1\cdot 3}{2\cdot 4}\cdot\frac{x^5}{5} \;-\; \frac{1\cdot 3\cdot 5}{2\cdot 4\cdot 6}\cdot\frac{x^7}{7} \;+-\ldots \\
&= x \;-\; \frac{1}{6}\,x^3 \;+\; \frac{3}{40}\,x^5 \;-\; \frac{5}{112}\,x^7 \;+\; \frac{35}{1152}\,x^9 \;-\; \frac{63}{2816}\,x^{11} \;+\; \frac{231}{13312}\,x^{13} -+\ldots
\end{aligned}$$

Die Reihe entspricht der des Arcussinus mit alternierendem Vorzeichen. Zur Berechnung der Koeffizienten c_k sei daher auf den entsprechenden Abschnitt 4.8.1 verwiesen.

Ab welchem Schwellwert $\bar{x} < 1$ das Umsteigen auf die Logarithmus/Wurzel-Darstellung lohnt, hängt von der Geschwindigkeit der Implementation dieser beiden Funktionen ab und sollte in praktischen Tests ermittelt werden. Wir benutzen $\frac{1}{16} = 0.0625$ – es ergab sich ein kleinerer Wert als $\frac{1}{8}$ bei [Steins], da die Logarithmusberechnung deutlich schneller ist und die Koeffizienten der arsinh-Reihe vergleichsweise aufwändig zu berechnen sind.

4.12.1 Auswertung der Potenzreihe

Die Auswertung verläuft völlig analog zum Arcussinus. Die benutzte Klammerung ist also folgende:

$$\begin{aligned}
P_N(x) &= x\left[1 - \frac{1}{2}\cdot\frac{x^2}{3} + \frac{1\cdot 3}{2\cdot 4}\cdot\frac{x^4}{5} - \frac{1\cdot 3\cdot 5}{2\cdot 4\cdot 6}\cdot\frac{x^6}{7} + \frac{1\cdot 3\cdot 5\cdot 7}{2\cdot 4\cdot 6\cdot 8}\cdot\frac{x^8}{9} -+\ldots\right] \\
&= x\left[1 - \frac{1}{2}\,x^2\left(\frac{1}{3} - \frac{3}{4}\,x^2\left(\frac{1}{5} - \frac{5}{6}\,x^2\left(\frac{1}{7} - \ldots - \frac{2N-1}{2N}\,x^2\cdot\left(\frac{1}{2N+1}\right)\right)\ldots\right)\right)\right].
\end{aligned}$$

Vor allem durch die eingesparten Divisionen (bei der Berechnung der Koeffizienten) wird unsere Version insgesamt (auf dem Linux-Rechner bei $x = \frac{1}{16}$) bereits bei 30 Stellen um den Faktor 7.6 mal so schnell wie die zu [Steins].

Das entstehende Verfahren sieht wie folgt aus ($u = x^2$):

$$\begin{aligned}
&\text{a)} \quad r_N &&:= \frac{1}{2N+1},\\
&\text{b)} \quad r_k &&:= \frac{1}{2k+1} - \frac{2k+1}{2k+2}\, u\, r_{k+1} = \frac{(2k+2) - (2k+1)^2\, u\, r_{k+1}}{(2k+1)(2k+2)} \quad \text{für } k = N-1, \ldots, 1,\\
&\text{c)} \quad r_0 &&:= 1 - \frac{1}{2}\, x^2\, r_1\\
&\text{d)} \quad &&P(x) = x \cdot r_0.
\end{aligned}$$

Es ist zu beachten, dass wiederum bei geradem N die Obergrenze, bei ungeradem N die Untergrenze einer Einschließung berechnet wird. Durch die Art der Aufteilung kommen so allerdings diesmal gemischte Rundungen innerhalb eines Schritts b) zustande. Bei geradem k wird die Division in Endrichtung gerundet, das $u = x^2$ dagegen in die andere Richtung (umgekehrt bei ungeradem k).

4.12.2 Auswertefehler beim Horner-Verfahren

Man erhält zunächst dieselbe Fehlerabschätzung wie beim Arcussinus, da das Vorzeichen jeweils in der Multiplikation mit dem Fehlerintervall E_ℓ verschwindet (Bezeichnung q_k wie dort auf Seite 121):

$$\begin{aligned}
\Delta_P \;&\in E_\ell \left[x r_0 + \frac{1}{2} x^3 r_1 (1+E_\ell) + \sum_{k=2}^{N} \frac{1 \cdot 3 \cdot \ldots \cdot (2k-1)}{2 \cdot 4 \cdot \ldots \cdot (2k)}\, x^{2k+1}\, r_k \left((1+E_\ell)^{2k-2} + (1+E_\ell)^{2k-1} \right) \right]\\
&= E_\ell \left[x r_0 + \frac{1}{2} x^3 r_1 (1+E_\ell) + (2+E_\ell) \sum_{k=2}^{N} q_k\, x^{2k+1}\, r_k (1+E_\ell)^{2k-2} \right]\\
&\subseteq E_\ell \left[x r_0 + \frac{1}{2} x^3 r_1 (1+E_\ell) + (2+E_\ell) \sum_{k=2}^{N} q_k\, x^{2k+1}\, r_k (1+(2k-1)E_\ell) \right]
\end{aligned}$$

Durch das alternierende Vorzeichen können wir allerdings die Zwischenergebnisse r_k einfacher als bei arcsin abschätzen. Alle r_k sind positiv, und aus ihrer Definition folgt direkt

$$r_k < \frac{1}{2k+1},$$

was ab r_2 als Abschätzung völlig ausreicht. Wir erhalten:

$$\begin{aligned}
\Delta_P \;&\in E_\ell \left[x r_0 + \frac{1}{2} x^3 r_1 (1+E_\ell) + (2+E_\ell) \sum_{k=2}^{N} q_k\, \frac{x^{2k+1}}{2k+1} (1+(2k-1)E_\ell) \right]\\
&\subseteq E_\ell \Bigg[x r_0 + \frac{1}{2} x^3 r_1 (1+E_\ell) + (2+E_\ell) \underbrace{\sum_{k=2}^{N} q_k\, \frac{x^{2k+1}}{2k+1}}_{A} + E_\ell (2+E_\ell) \underbrace{\sum_{k=2}^{N} q_k\, \frac{x^{2k+1}}{2k+1} (2k-1)}_{B} \Bigg].
\end{aligned}$$

Es ist $q_k \le \frac{3}{8}$ für $k \ge 2$. Dann erhält man für die beiden Summen Folgendes:

$$\begin{aligned} A &= \sum_{k=2}^{N} q_k \frac{x^{2k+1}}{2k+1} \;\le\; \frac{3}{8}\sum_{k=2}^{N}\frac{x^{2k+1}}{2k+1} \;\le\; \frac{3}{8}\left(\operatorname{artanh} x - x - \frac{x^3}{3}\right), \\ B &= \sum_{k=2}^{N} q_k \frac{x^{2k+1}}{2k+1}(2k-1) = \sum_{k=2}^{N} q_k(2k+1)\frac{x^{2k+1}}{2k+1} - 2\sum_{k=2}^{N} q_k \frac{x^{2k+1}}{2k+1} \\ &= \sum_{k=2}^{N} q_k x^{2k+1} - 2\sum_{k=2}^{N} q_k \frac{x^{2k+1}}{2k+1} \le \frac{3}{8}\sum_{k=2}^{N} x^{2k+1} - 2\sum_{k=2}^{N} q_k \frac{x^{2k+1}}{2k+1} \\ &\le \frac{3}{8}\cdot\frac{x^5}{1-x^2} - \frac{3}{20}\,x^5. \end{aligned}$$

Bei $\varepsilon_P = \frac{\Delta_P}{r_0}$ schätzen wir r_0 mit den ersten zwei bzw. vier Gliedern ab (es gilt ja die Generalvoraussetzung $N \ge 3$), also $r_0 \ge 1 - \frac{x^2}{6} + \frac{3}{40}x^4 - \frac{5}{112}x^6$. Dann haben wir

$$\begin{aligned} \varepsilon_P \;\in E_\ell\Bigg[& 1 + (1+E_\ell)\left(\frac{1}{1-\frac{x^2}{6}+\frac{3}{40}x^4-\frac{5}{112}x^6} - 1\right) + (2+E_\ell)\cdot\frac{3}{8}\cdot\frac{\frac{\operatorname{artanh} x}{x} - 1 - \frac{x^2}{3}}{1-\frac{x^2}{6}} \\ & + E_\ell(2+E_\ell)\cdot\frac{x^4}{1-\frac{x^2}{6}}\cdot\left(\frac{3}{8(1-x^2)} - \frac{3}{20}\right)\Bigg]. \end{aligned}$$

Mit $b = 10$, $x \le \frac{1}{16}$ und $\ell \ge 5$ ergibt das

$$\begin{aligned} \varepsilon_P &\subseteq E_\ell \cdot (\,1 + 0.00065039 + 2.2972\cdot 10^{-6} + 6.9162\cdot 10^{-10}\,) \\ &\subseteq 1.000653 \cdot E_\ell. \end{aligned}$$

Wir benötigen also wiederum zwei Schutzziffern, und für den Approximationsfehler ergibt sich die Forderung

$$|\varepsilon_{app}| \le 8.9984 \cdot 10^{-p-1}.$$

4.12.3 Approximationsfehler

Die Koeffizienten c_k haben alternierendes Vorzeichen und fallen betragsmäßig monoton. Mit dem Approximationspolynom $P_N(x)$ bei Abbruch nach dem N-ten Glied und mit $0 < x < 1$, gilt also für den relativen Approximationsfehler (diesmal in Übereinstimmung mit [Krämer] und [Steins]):

$$|\varepsilon_{app}| = \left|\frac{\operatorname{arsinh} x - P_N(x)}{\operatorname{arsinh} x}\right| = \Big|\sum_{k=N+1}^{\infty} c_k x^{2k+1}\Big| \Big/ \sum_{k=0}^{\infty} c_k x^{2k+1} \le \frac{|c_{N+1}|\cdot x^{2N+3}}{x - \frac{x^3}{6}} = |c_{N+1}|\cdot\frac{x^{2N+2}}{1-\frac{x^2}{6}}.$$

Wie üblich wird aus einer Tabelle ein Start-N interpoliert, mit der Routine aus dem letzten Abschnitt $|c_{N+1}|$ berechnet. Danach wird ggf. N erhöht und mit der iterativen Methode $|c_{N+1}|$, bis

$$\begin{aligned} |c_{N+1}|\cdot\frac{x^{2N+2}}{1-\frac{x^2}{6}} &\le 8.9984\cdot 10^{-p-1}, \qquad \text{bzw.} \\ |c_{N+1}|\cdot x^{2N+2} &\le 1.49973\cdot 10^{-p-1}\cdot(6-x^2). \end{aligned}$$

Die Tabelle für arsinh sieht wie folgt aus:

p	$j=1$	2	3	4	5	6	7	8	9	10
3	3	3	3	3	3	3	3	3	3	3
10	3	3	3	3	3	3	3	3	4	3
33	7	8	8	9	10	10	11	11	12	12
100	22	25	28	30	32	34	35	37	38	40
330	74	85	94	101	108	114	120	125	130	135
1000	225	261	288	310	330	349	366	382	398	413

4.12.4 Auswertung mit Logarithmus und Quadratwurzel

Für $x > \frac{1}{16}$ verwenden wir die Beziehung

$$\operatorname{arsinh} x = \ln u, \qquad u := x + \sqrt{1+x^2},$$

und leiten im Folgenden her, mit wie vielen Schutzziffern die Auswertung des Arguments und des Logarithmus zu erfolgen hat. Für sehr große x wird sogar $\operatorname{arsinh} x \approx \ln(2x)$ verwendet, wofür im nächsten Abschnitt die Fehlerabschätzung folgen wird.

Es werde u gestört berechnet als $\tilde{u} = u(1+\varepsilon_u)$. Dann gilt für den relativen Gesamtfehler

$$\begin{aligned}
|\varepsilon_{\operatorname{arsinh}}| &= \left|\frac{\widetilde{\ln} u - \ln u}{\ln u}\right| = \left|\frac{(\ln(u) + \frac{\varepsilon_u u}{\xi})(1+\varepsilon_{\ln}) - \ln u}{\ln u}\right|, \qquad \xi \in \operatorname{conv}\{u, u(1+\varepsilon_u)\} \\
&= \left|\frac{\varepsilon_{\ln} \ln(u) + \frac{\varepsilon_u u}{\xi}(1+\varepsilon_{\ln})}{\ln u}\right| \;\le\; |\varepsilon_{\ln}| + \left|\frac{u}{\xi \ln u}\right| (1+\varepsilon_{\ln})|\varepsilon_u| \\
&\le |\varepsilon_{\ln}| + \frac{1+\varepsilon_{\ln}}{1-|\varepsilon_u|} \cdot \frac{1}{\ln u}\,|\varepsilon_u| \;\le\; |\varepsilon_{\ln}| + 16.014\,|\varepsilon_u|,
\end{aligned}$$

Letzteres wegen $x > 0.0625$, also $\ln u > 0.06245$, und unter der Voraussetzung, dass u und $\ln u$ zumindest mit 5 Stellen berechnet werden. Hieraus folgt für $b = 10$ bereits, dass der Logarithmus mit 2 Schutzziffern ermittelt werden muss, und dann ist $|\varepsilon_{\ln}| \le 0.2 \cdot 10^{-p}$. Unsere Forderung an das Argument u ist daher

$$|\varepsilon_u| \;\overset{!}{\le}\; 0.04995 \cdot 10^{-p}.$$

Da x nicht extrem groß (oder klein) ist, kann $1+x^2$ zwischenzeitlich exakt berechnet werden, wird aber als Argument für die Wurzel gerundet. Die äußere Addition kann ebenfalls exakt durchgeführt werden, da sich Wurzel und x in der Größenordnung nicht besonders unterscheiden (maximal für das kleinste $x = \frac{1}{16}$ um eineinhalb Stellen).

Das gestörte u lässt sich also wie folgt darstellen:

$$\begin{aligned}
\tilde{u} &= x + \sqrt{(1+x^2)(1+\varepsilon_1)}(1+\varepsilon_2) = x + \sqrt{1+x^2}\,\sqrt{1+\varepsilon_1}\,(1+\varepsilon_2) \\
&= x + \sqrt{1+x^2}\,(1+\varepsilon_1')\,(1+\varepsilon_2), \qquad \varepsilon_1' \in \operatorname{conv}\left\{\frac{\varepsilon_1}{2}, \frac{\varepsilon_1}{2\sqrt{1+\varepsilon_1}}\right\}.
\end{aligned}$$

Damit ergibt sich für den relativen Fehler

$$\begin{aligned}
|\varepsilon_u| &= \left|\frac{\tilde{u}-u}{u}\right| = \left|\frac{\sqrt{1+x^2}(1+\varepsilon_1')(1+\varepsilon_2) - \sqrt{1+x^2}}{x+\sqrt{1+x^2}}\right| = \left|\frac{(\varepsilon_1' + \varepsilon_2 + \varepsilon_1'\varepsilon_2)\sqrt{1+x^2}}{x+\sqrt{1+x^2}}\right| \\
&\le \frac{\sqrt{1+x^2}}{x+\sqrt{1+x^2}}\,((1+\varepsilon_2)|\varepsilon_1'| + |\varepsilon_2|).
\end{aligned}$$

Der Quotient ist monoton fallend, ist also durch den Wert in $x = \frac{1}{16}$ beschränkt. Wenn wir außerdem wieder mindestens mit fünf Dezimalstellen rechnen, ergibt sich als Forderung

$$|\varepsilon_u| \;\leq\; 0.4708\,|\varepsilon_1| \;+\; 0.9413\,|\varepsilon_2| \;\overset{!}{\leq}\; 0.04995 \cdot 10^{-p}.$$

Beim Ansatz mit gleichen Stellenzahlen $(0.4708+0.9413)10^{-\ell+1} \leq 0.04995 \cdot 10^{-p}$ ergeben sich drei notwendige Schutzziffern. Wenn man die Toleranz auf beide Summanden gleichmäßig verteilt, erhält man für die Stellenzahl m der Wurzelberechnung $m \geq p + 2.276$ und für die Stellenzahl ℓ für das Argument $\ell \geq p + 1.975$, also nur zwei Schutzziffern. Da letztere Konstante aber recht groß und das Argument leicht zu berechnen ist, rechnen wir doch in beiden Fällen mit drei Schutzziffern. Damit ergibt sich für den relativen Gesamtfehler

$$|\varepsilon_{\text{arsinh}}| \leq 0.2 \cdot 10^{-p} + 16.014(0.4708 + 0.9413 \cdot 10^{-p+2}) \leq 0.4262 \cdot 10^{-p}.$$

Wie rechnen wiederum nicht intervallmäßig. Wir berechnen eine Unterschranke des Arguments der Wurzel, verwenden die Untergrenze der Einschließung der Wurzel und dann die des Logarithmus. Aus der letzten Abschätzung erhalten wir dann eine gesicherte Oberschranke.

4.12.5 Auswertung für sehr große Argumente

Bei großen x kann der Areasinus direkt durch $\ln(2x)$ approximiert werden. Es muss natürlich sicher gestellt werden, dass der Gesamtfehler noch den Genauigkeitsforderungen entspricht.

Auf diese Weise vermeidet man auch einen zwischenzeitlichen Überlauf bei der Berechnung von x^2, obwohl das Endergebnis wieder darstellbar wäre. [Krämer] verwendet diese Berechnungsweise nur aus diesem Grund und daher nur für sehr große Argumente. Wir gehen ähnlich wie dort vor, aber wir versuchen, einen schnellen Test zu finden, wann der Fehler klein genug bleibt, da wir *immer* so arbeiten wollen, wenn es möglich ist. Unsere Abschätzungen werden zum Schluss daher sehr grob sein.

Der Vergleich der beiden Berechnungsweisen sieht wie folgt aus:

$$\begin{aligned}
\ln(x+\sqrt{1+x^2}) &= \ln(2x + (\sqrt{1+x^2} - x)) \;=\; \ln\Big(2x + x\Big(\sqrt{1+\tfrac{1}{x^2}} - 1\Big)\Big) \\
&\leq \ln(2x + x((1+\tfrac{1}{2x^2}) - 1)) \;=\; \ln(2x + (\tfrac{1}{2x})) \\
&= \ln(2x) + \tfrac{1}{\xi \cdot 2x}, \qquad 2x < \xi < 2x + \tfrac{1}{2x} \\
&\leq \ln(2x) + \tfrac{1}{4x^2}
\end{aligned}$$

Damit ergibt sich für den relativen Gesamtfehler (die Multiplikation mit 2 kann exakt durchgeführt werden):

$$\begin{aligned}
|\varepsilon_{\text{arsinh}}| &= \left|\frac{\widetilde{\ln}\,(2x) - \ln(2x) + \ln(2x) - \ln(x+\sqrt{1+x^2})}{\ln(x+\sqrt{1+x^2})}\right| \\
&= \left|\frac{\varepsilon_{\ln}\,\ln(2x) + \ln(2x) - \ln(x+\sqrt{1+x^2})}{\ln(x+\sqrt{1+x^2})}\right| \\
&\leq \frac{\ln(2x)}{\ln(x+\sqrt{1+x^2})}\,|\varepsilon_{\ln}| + \frac{\ln(x+\sqrt{1+x^2}) - \ln(2x)}{\ln(x+\sqrt{1+x^2})}. \\
&\leq |\varepsilon_{\ln}| + \frac{1}{4x^2 \cdot \ln(x+\sqrt{1+x^2})} \qquad \overset{!}{\leq} b^{-p}.
\end{aligned}$$

Wenn die Logarithmusberechnung mit zwei Schutzziffern erfolgt, muss für den Bruch gelten ($b = 10$):

$$\begin{aligned} 4x^2 \cdot \ln(x + \sqrt{1+x^2}) &\geq 1.25 \cdot 10^p, \\ 3.2 \cdot x^2 \cdot \ln(2x) &\geq 10^p, \\ 7.368 \cdot x^2 \cdot \log_{10}(2x) &\geq 10^p, \\ 2\,\mathrm{order}(x) + \mathrm{order}(\mathrm{order}(2x)) &\geq p - 0.867. \end{aligned}$$

Momentan wird allerdings schneller getestet

$$2\,\mathrm{order}(x) \geq p,$$

im Hinblick darauf, dass so große Argumente, dass der doppelte order-Term aussschlaggebend wird, doch relativ selten sein werden.

4.12.6 Auswertung für sehr kleine Argumente

Für sehr kleine (positive) x wird $\mathrm{arsinh}(x) \overset{<}{\approx} x$ verwendet. Es gilt (für $0 < x \leq 10^{-3}$):

$$\begin{aligned} \frac{x - \mathrm{arsinh}\, x}{x} &= \frac{\frac{1}{6}x^3 - \frac{3}{40}x^5 + \frac{5}{112}x^7 - + \ldots}{x} \leq \frac{\frac{1}{6}x^3\,(1 + x^4 + x^8 + \ldots)}{x} \\ &= \frac{x^2}{6}\,\frac{1}{1-x^4} \leq 0.166667\,x^2 \overset{!}{\leq} b^{-p}, \end{aligned}$$

was denselben Test wie beim Sinus anwendbar macht. Wenn er positiv ist, wird als Einschließung $[x(1-b^{-p})\,,\, x]$ berechnet.

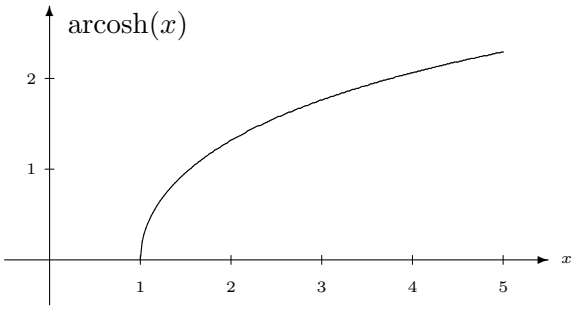
4.12.7 Intervallversion

Der Areasinus ist auf ganz $\mathbb{R}$ streng monoton steigend, so dass zur intervallmäßigen Implementation nicht spezielles zu bemerken ist.

4.13 Der Areacosinus

Der Areacosinus ist zu verstehen als die Umkehrfunktion des hyperbolischen Cosinus auf $\mathbb{R}_0^+$, d.h. er ist nur auf $[1, +\infty[$ definiert. Für $y \geq 0$ erhält man durch Umstellen der Definition von cosh:

$$\begin{aligned} \cosh y &= \frac{e^{2y}+1}{2e^y}, \\ \mathrm{arcosh}\, x &= \ln(x + \sqrt{x^2-1}). \end{aligned}$$



Für kleine Argumente wird dagegen wiederum die Potenzreihe verwendet.

Nun ist $\operatorname{arcosh} x$ aber in 1 nicht differenzierbar ($f'(x) = \frac{1}{\sqrt{x^2-1}}$, $x > 1$). In [Krämer] (ab Seite 65) wird über die Betrachtung einer (auf ganz $\mathbb{R}$ differenzierbaren) Hilfsfunktion und deren Taylor-Entwicklung folgende Darstellung hergeleitet:

$$\operatorname{arcosh} x = \sqrt{x^2-1} \cdot \sum_{k=0}^{\infty} c_k \, (x-1)^k, \qquad 1 < x < 2,$$

mit

$$c_k = (-1)^k \, \frac{1 \cdot 2 \cdot 3 \cdot \ldots \cdot k}{3 \cdot 5 \cdot 7 \cdot \ldots \cdot (2k+1)}.$$

Die Potenzreihe beginnt wie folgt:

$$P(u) = 1 - \frac{1}{3}u + \frac{2}{15}u^2 - \frac{2}{35}u^3 + \frac{8}{315}u^4 - \frac{8}{693}u^5 + \frac{16}{3003}u^6 - \frac{16}{6435}u^7 + \frac{128}{109395}u^8 - + \ldots$$

Für $x \geq 2$ wird also eine Argumentreduktion benötigt, für kleinere x ist eine wegen der geringen Konvergenzgeschwindigkeit sinnvoll.

4.13.1 Berechnung der Koeffizienten

Iterativ gilt offensichtlich $c_0 = 1$ und für $k \geq 1$

$$c_k = -c_{k-1} \cdot \frac{k}{2k+1}.$$

Auch hier ist es sinnvoll, für das Start-N die Bestimmung der Anzahl notwendiger Reihenglieder eine Beziehung für die c_k anzugeben, die die schnelleren Routinen zur Potenzierung und Fakultät benutzen kann. Für gerades k gilt

$$c_k = \frac{1 \cdot 2 \cdot 3 \cdot \ldots \cdot k}{3 \cdot 5 \cdot 7 \cdot \ldots \cdot (2k+1)} = \frac{2 \cdot 4 \cdot 6 \cdot \ldots \cdot k}{(k+1)(k+3)\ldots(2k+1)} = \frac{2^{\frac{k}{2}} \cdot (\frac{k}{2})!}{(k+1)(k+3)\ldots(2k+1)}$$

und für ungerades k

$$c_k = -\frac{1 \cdot 2 \cdot 3 \cdot \ldots \cdot k}{3 \cdot 5 \cdot 7 \cdot \ldots \cdot (2k+1)} = -\frac{2 \cdot 4 \cdot 6 \cdot \ldots \cdot (k-1)}{(k+2)(k+4)\ldots(2k+1)} = -\frac{2^{\lfloor \frac{k}{2} \rfloor} \cdot \lfloor \frac{k}{2} \rfloor!}{(k+2)(k+4)\ldots(2k+1)},$$

mit jeweils $\lfloor \frac{k}{2} \rfloor + 1$ Gliedern im Nenner.

4.13.2 Auswertung der Potenzreihe

Wir können hier wieder in derselben Art klammern wie bei exp, sin, cos und sinh:

$$\begin{aligned} P_N(x) &= 1 - \frac{1}{3}\,u + \frac{1\cdot 2}{3\cdot 5}\,u^2 - \frac{1\cdot 2\cdot 3}{3\cdot 5\cdot 7}\,u^3 + \frac{1\cdot 2\cdot 3\cdot 4}{3\cdot 5\cdot 7\cdot 9}\,u^4 - + \ldots + (-1)^N \frac{1 \cdot 2 \cdot 3 \cdot \ldots \cdot N}{3\cdot 5\cdot 7\cdot \ldots \cdot (2N+1)}\,u^N \\ &= 1 - \frac{1}{3}\,u\left(1 - \frac{2}{5}\,u\left(1 - \frac{3}{7}\,u\left(1 - \frac{4}{9}\,u\left(\ldots - \frac{N}{2N+1}\,u\right)\right)\right)\ldots\right) \end{aligned}$$

Das entsprechende Verfahren ist also:

$$\begin{aligned}
&\text{a)} \quad r_N &&:= \frac{u \cdot N}{2N+1}, \\
&\text{b)} \quad r_k &&:= \frac{u \cdot k \cdot (1 - r_{k+1})}{2k+1} \qquad \text{für } k = N-1, \ldots, 1, \\
&\text{c)} \quad r_0 &&:= 1 - r_1, \\
&\Rightarrow \quad P(u) &&= r_0.
\end{aligned}$$

In der Durchführung wird wieder mit geeigneten gerichteten Rundungen bei geradem N eine Obergrenze, bei ungeradem N eine Untergrenze berechnet.

4.13.3 Auswertefehler beim Horner-Verfahren

Eine Abschätzung des Auswertefehlers beim Polynom ergibt sich aus Lemma 7 unter Anpassung an ein exaktes Argument. Alternativ kann man schnell direkt Folgendes ablesen:

$$\begin{aligned}
&\text{a)} \quad \Delta_N &&\in r_N E_\ell \\
&\text{b)} \quad \Delta_k &&\in r_k E_\ell - \frac{k}{2k+1}\, u\,(1+E_\ell)\,\Delta_{k+1} \qquad \text{für } k = N-1, \ldots, 1, \\
&\text{c)} \quad \Delta_0 &&\in r_0 E_\ell - (1+E_\ell)\Delta_1,
\end{aligned}$$

was insgesamt ergibt:

$$\Delta_P \in E_\ell \left[r_0 + \sum_{k=0}^{N-1} |c_k|\, r_{k+1}\, u^k\, (1+E_\ell)^{k+1} \right].$$

Es gilt

$$r_k = \sum_{i=1}^{N-k+1} (-1)^{i-1} \frac{k(k+1)(k+2)\ldots(k+i-1)}{(2k+1)(2k+3)\ldots(2k+2i-1)}\, u^i,$$

also wegen des alternierenden Vorzeichens und fallender Beträge

$$0 \;<\; r_k \;\leq\; \frac{k}{2k+1}\, u.$$

Außerdem verwenden wir, dass für $k \geq 2$ gilt

$$|c_k| \;<\; \frac{1}{3} \cdot \frac{1}{2^{k-1}}.$$

Insgesamt ergibt sich

$$\begin{aligned}
\Delta_P \;&\in\; E_\ell \Big[\, r_0 + r_1\,(1+E_\ell) + \frac{2}{15}\, u^2\,(1+E_\ell)^2 + \sum_{k=3}^{N} |c_{k-1}|\, \frac{k}{2k+1}\, u^k\,(1+E_\ell)^k \,\Big] \\
&\subseteq\; E_\ell \Big[\, r_0 + r_1\,(1+E_\ell) + \frac{2}{15}\, u^2\,(1+E_\ell)^2 + \frac{4}{3} \sum_{k=3}^{N} \frac{1}{2^k}\, \frac{k}{2k+1} u^k (1+E_\ell)^k \,\Big]
\end{aligned}$$

$$\begin{aligned}
&\overset{\text{L1}}{\subseteq} \; E_\ell \Big[\, r_0 + r_1\,(1+E_\ell) + \frac{2}{15}\,u^2\,(1+E_\ell)^2 + \frac{4}{3}\sum_{k=3}^{N}\frac{1}{2^k}\,\frac{k}{2k+1}u^k(1+(k+1)E_\ell)\,\Big] \\
&\subseteq \; E_\ell \Big[\, r_0 + (1-r_0)\,(1+E_\ell) + \frac{2}{15}\,u^2\,(1+E_\ell)^2 + \underbrace{\frac{4}{3}\sum_{k=3}^{N}\frac{1}{2^k}\,\frac{k}{2k+1}u^k}_{A} \\
&\qquad\qquad + \frac{4}{3}E_\ell \underbrace{\sum_{k=3}^{N}\frac{1}{2^k}\,\frac{k}{2k+1}(k+1)u^k}_{B}\,\Big]
\end{aligned}$$

Für die Summen A und B gilt (zwischenzeitlich mit der Abkürzung $v := \frac{u}{2}$)

$$\begin{aligned}
A \;&=\; \sum_{k=3}^{N}\frac{k}{2k+1}\,v^k \;=\; \frac{1}{2}\sum_{k=3}^{N}\frac{2k+1}{2k+1}\,v^k - \frac{1}{2}\sum_{k=3}^{N}\frac{1}{2k+1}\,v^k \;=\; \frac{1}{2}\sum_{k=3}^{N}v^k - \frac{1}{2}\sum_{k=3}^{N}\frac{1}{2k+1}\,v^k \\
&\le\; \frac{1}{2}\cdot\frac{v^3}{1-v} - \frac{1}{2}\cdot\frac{v^3}{7} \;=\; \frac{u^3}{8}\left(\frac{1}{2-u} - \frac{1}{14}\right), \\
B \;&=\; \sum_{k=3}^{N}v^k\frac{k(k+1)}{2k+1} \;=\; \frac{1}{2}\sum_{k=3}^{N}(k+1)v^k - \frac{1}{2}\sum_{k=3}^{N}\frac{k+1}{2k+1}\,v^k \;\le\; \frac{u^3}{8}\left(\frac{8-3u}{(2-u)^2} - \frac{2}{9}\right),
\end{aligned}$$

wobei Letzteres gilt, da

$$\begin{aligned}
\sum_{k=3}^{N}(k+1)v^k \;&=\; \sum_{k=3}^{N}(v^{k+1})' = \Big(\sum_{k=3}^{N}v^{k+1}\Big)' = \Big(v^4\cdot\frac{1-v^{N-2}}{1-v}\Big)' \\
&=\; \frac{4v^3 - 3v^4 - (N+2)v^{N+1} + (N+1)v^{N+2}}{(1-v)^2} \;\le\; \frac{4v^3-3v^4}{(1-v)^2}.
\end{aligned}$$

Beim relativen Auswertefehler $\varepsilon_P = \frac{\Delta_P}{r_0}$ schätzen wir r_0 im Nenner nach unten durch die ersten vier Terme ab (es ist ja $N \ge 3$), also mit $n(x) = 1 - \frac{1}{3}u + \frac{2}{15}u^2 - \frac{2}{35}u^3$:

$$\begin{aligned}
\varepsilon_P \;\in\; E_\ell \Big[1 + \Big(\frac{1}{n(x)} - 1\Big)(1+E_\ell) + \frac{1}{n(x)}\cdot\frac{2}{15}\;u^2\,(1+E_\ell)^2 + \frac{4}{3}\cdot\frac{u^3}{n(x)}\left(\frac{1}{8}\cdot\frac{1}{2-u} - \frac{1}{112}\right) \\
+E_\ell\cdot\frac{1}{n(x)}\cdot\frac{u^3}{6}\left(\frac{8-3u}{(2-u)^2} - \frac{2}{9}\right)\Big].
\end{aligned}$$

Für $0 < u \le \frac{1}{8}$ sind die Faktoren der Fehlerterme monoton steigend. Also gilt mit $b = 10$, $\ell \ge 5$:

$$\begin{aligned}
\varepsilon_P \;&\in\; E_\ell\,[1 + 0.041339895 + 0.0021698835 + 1.5657486\cdot 10^{-4} + 6.5987423\cdot 10^{-8}] \\
&\subseteq\; 1.043667\cdot E_\ell.
\end{aligned}$$

Die Kontrolle auf dem Rechner liefert den Faktor 1.043505. (In [Steins] wird für das normale Horner-Verfahren fälschlicherweise 1.0319 berechnet, da zwischendurch $u = x-1$ und $u = x^2$ verwechselt und daher die rechte Grenze als $\frac{1}{64}$ statt als $\frac{1}{8}$ angenommen wird. Mit der Grenze $\frac{1}{8}$ würde man dort den Wert 1.2965 erhalten.)

Zusätzlich müssen wir natürlich die Fehler mit einbeziehen, die durch die Auswertung des Vorfaktors $\sqrt{x^2-1}$ und die Multiplikation damit (bzw. die Endrundung) entstehen.

Der relative Gesamtfehler der Approximation der Potenzreihe heiße wie immer ε_{ges}, der Fehler bei der Auswertung der Wurzel $\varepsilon_{\surd}$, der Fehler bei der gerundeten Subtraktion ε_-.

Dann gilt für den relativen Gesamtfehler vor der Endrundung:

$$\begin{aligned}|\varepsilon_{\text{arcosh}}| &= |\sqrt{1+\varepsilon_-}\cdot(1+\varepsilon_{\surd})\cdot(1+\varepsilon_{ges})-1| = \left|\left(1+\frac{\varepsilon_-}{2\sqrt{\xi}}\right)(1+\varepsilon_{\surd})(1+\varepsilon_{ges})-1\right| \\ &\leq |\varepsilon_{ges}|+(1+\varepsilon_{ges})|\varepsilon_{\surd}|+\frac{|\varepsilon_-|}{2\sqrt{1-|\varepsilon_-|}}(1+\varepsilon_{ges}+\varepsilon_{\surd}(1+\varepsilon_{ges}))\end{aligned}$$

(mit $\xi \in \text{conv}\{1, 1+\varepsilon_-\}$). Bei ℓ-stelliger Rechnung in allen Operationen erhält man also die Forderung

$$2\cdot 10^{-\ell+1}+1.0002\cdot 10^{-\ell+1}+0.5002\cdot 10^{-\ell+1} = 3.5004\cdot 10^{-\ell+1} \overset{!}{\leq} 10^{-p},$$

d.h. man kommt mit 2 Schutzziffern aus. Das Argument der Wurzel kann aber auch problemlos exakt berechnet werden: Da $1 < x < 1+\frac{1}{8}$ gilt, ist die Subtraktion ohne Mantissenverlängerung exakt durchführbar. Die Wurzelauswertung läuft dann aber mit etwa der doppelten Mantissenlänge.

Wenn 2 Schutzziffern verwendet werden, kann die entstehende Toleranz für die Auswertung der Potenzreihe verwendet werden:

$$|\varepsilon_{ges}| \leq 10^{-p}-1.0002\cdot 10^{-p-1}-0.5002\cdot 10^{-p-1} = 0.84996\cdot 10^{-p}.$$

Daraus erhalten wir eine leichte Abwandlung von (4.8):

$$\begin{aligned}|\varepsilon_{ges}| &\leq |\varepsilon_P|+|\varepsilon_{app}|(1+|\varepsilon_P|) \\ &\leq 1.043667\cdot 10^{-\ell+1}+|\varepsilon_{app}|(1+1.043667\cdot 10^{-\ell+1}) \\ &\overset{!}{\leq} 0.84996\cdot 10^{-p},\end{aligned}$$

bzw. schärfer

$$\begin{aligned}|\varepsilon_{app}| &\leq 10^{-p-1}\cdot\frac{8.4996-1.043667}{1+1.043667\cdot 10^{-4}}, \\ \text{bzw. } |\varepsilon_{app}| &\leq 7.4551\cdot 10^{-p-1}.\end{aligned}$$

4.13.4 Approximationsfehler

Auch diese Koeffizienten c_k haben alternierendes Vorzeichen und fallen betragsmäßig monoton. Wir erhalten also für den Approximationsfehler des Polynom-Teils:

$$|\varepsilon_{app}^N| = \left|\sum_{k=N+1}^{\infty} c_k(x-1)^k\right| \Big/ \sum_{k=0}^{\infty} c_k(x-1)^k \leq \frac{|c_{N+1}|\cdot(x-1)^{N+1}}{1-\frac{(x-1)}{3}} = \frac{3|c_{N+1}|(x-1)^{N+1}}{4-x}.$$

Unter Verwendung der Schranke aus Teil d) oben suchen wir also ein kleines N, so dass

$$\frac{3|c_{N+1}|(x-1)^{N+1}}{4-x} \leq 7.4551\cdot 10^{-p-1},$$

implementiert als

$$|c_{N+1}| \cdot u^{N+1} \le (3-u) \cdot 2.485 \cdot 10^{-p-1},$$

mit einem Startwert aus der folgenden Tabelle (beachte, dass die Spalten den Bereich von $u = 0$ bis $u = \frac{1}{8}$ abdecken):

p	$j=1$	2	3	4	5	6	7	8	9	10
3	3	3	3	3	3	3	3	3	3	3
10	4	5	5	6	6	6	7	7	7	8
33	14	17	18	20	21	22	24	25	26	27
100	45	52	57	62	66	69	73	76	79	82
330	149	173	190	205	218	230	242	253	263	273
1000	453	524	578	623	663	700	735	767	799	829

4.13.5 Berechnung mit Logarithmus und Quadratwurzel

Für $x > 1 + \frac{1}{8}$ wird die Beziehung

$$\operatorname{arcosh} x = \ln u, \qquad u = x + \sqrt{x^2-1},$$

verwendet. Es sei wieder $\tilde{u} = u(1+\varepsilon_u)$ das mit Rundungsfehlern behaftete Argument u. Die Beziehung für den relativen Gesamtfehler kann vom Areasinus übernommen werden (mit anderer Unterschranke von u), d.h. wir haben

$$|\varepsilon_{\text{arcosh}}| \;\le\; |\varepsilon_{\ln}| + \frac{1+\varepsilon_{\ln}}{1-|\varepsilon_u|} \cdot \frac{1}{\ln u} |\varepsilon_u| \;\le\; |\varepsilon_{\ln}| + 2.022\,|\varepsilon_u|,$$

also benötigen wir zwei Schutzziffern für den Logarithmus, womit wir für das Argument u fordern können

$$|\varepsilon_u| \;\overset{!}{\le}\; 0.3956 \cdot 10^{-p}.$$

Für große x wird es wieder eine gesonderte Abschätzung geben, so dass wir hier davon ausgehen können, dass im Radikanden x^2-1 der Teil x^2 zwischenzeitlich exakt berechnet werden kann und dass die äußere Addition exakt erfolgt. Ähnlich wie beim Areasinus erhalten wir also

$$\tilde{u} = \sqrt{(x^2-1)(1+\varepsilon_1)}(1+\varepsilon_2),$$

also (mit ε_1' wie bei arsinh):

$$\begin{aligned}
|\varepsilon_u| \;&=\; \left|\frac{\tilde{u}-u}{u}\right| = \left|\frac{\sqrt{(x^2-1)(1+\varepsilon_1)}(1+\varepsilon_2) - \sqrt{x^2-1}}{x+\sqrt{x^2-1}}\right| = \left|\frac{\sqrt{(x^2-1)}(1+\varepsilon_1')(1+\varepsilon_2) - \sqrt{x^2-1}}{x+\sqrt{x^2-1}}\right| \\
&=\; \left|\frac{(\varepsilon_1'+\varepsilon_2+\varepsilon_1'\varepsilon_2)\sqrt{x^2-1}}{x+\sqrt{x^2-1}}\right| \;\le\; \underbrace{\frac{\sqrt{x^2-1}}{x+\sqrt{x^2-1}}}_{<\frac{1}{2}} \cdot (\,|\varepsilon_1'|\,(1+\varepsilon_2) + |\varepsilon_2|\,) \\
&\le\; 0.250041\,|\varepsilon_1| \;+\; 0.5\,|\varepsilon_2| \qquad \overset{!}{\le} 0.3956 \cdot 10^{-p}.
\end{aligned}$$

Wenn ℓ die Stellenzahl für Argument- und Wurzelberechnung ist, ergibt sich daraus als Forderung $\ell \ge p + 1.278$, also zwei Schutzziffern. Damit erhält man schließlich

$$|\varepsilon_{\text{arcosh}}| \;\le\; 0.2 \cdot 10^{-p} + 2.022(0.250041 + 0.5) \cdot 0.1 \cdot 10^{-p} \;\le\; 0.3517 \cdot 10^{-p},$$

was uns nach der Berechnung einer Unterschranke mit gerichteten Rundungen eine gesicherte Oberschranke liefert.

4.13.6 Auswertung für sehr große Argumente

Bei großen x gilt $\operatorname{arcosh}(x) \lessapprox \ln(2x)$. Die Fehlerabschätzungen ähneln denen beim Areasinus:

$$\begin{aligned}
\ln(x+\sqrt{x^2-1}) &= \ln\Big(x + x\sqrt{1-\tfrac{1}{x^2}}\Big) \;\geq\; \ln(x + x\Big(1 - \Big(2x^2\sqrt{1-\tfrac{1}{x^2}}\Big)^{-1}\Big)) \\
&= \ln(2x - \frac{1}{2\sqrt{x^2-1}}) \;\geq\; \ln(2x) - \Big(\Big(2x - \frac{1}{2\sqrt{x^2-1}}\Big)2\sqrt{x^2-1}\Big)^{-1} \\
&= \ln(2x) - \frac{1}{4x\sqrt{x^2-1}-1}
\end{aligned}$$

und damit

$$\begin{aligned}
|\varepsilon_{\operatorname{arcosh}}| &\leq \left|\frac{\varepsilon_{\ln}\ln(2x) + \ln(2x) - \ln(x+\sqrt{x^2-1})}{\ln(x+\sqrt{x^2-1})}\right| \\
&\leq \frac{\ln(2x)}{\ln(x+\sqrt{x^2-1})}|\varepsilon_{\ln}| + \frac{\ln(2x) - \ln(x+\sqrt{x^2-1})}{\ln(x+\sqrt{x^2-1})} \\
&\leq \frac{\ln(2x)}{\ln(x+\sqrt{x^2-1})}|\varepsilon_{\ln}| + \frac{1}{(4x\sqrt{x^2-1}-1)\ln(x+\sqrt{x^2-1})} \;\overset{!}{\leq}\; b^{-p}.
\end{aligned}$$

Wir setzen $b = 10$, $x \geq 100$ voraus. Der erste Quotient ist monoton fallend und kleiner als 1.000005. Wenn also der Logarithmus mit zwei Schutzziffern berechnet wird und der zweite Quotient durch den Rest beschränkt sein soll, fordern wir (schärfer werdend):

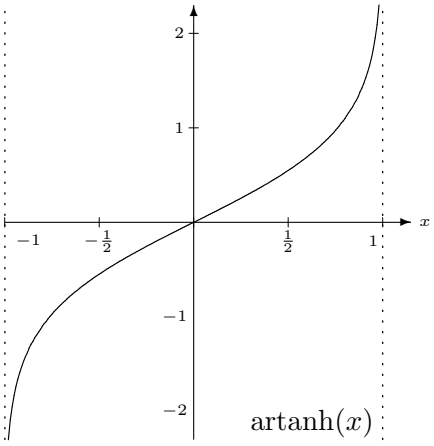
$$\begin{aligned}
(4x\sqrt{x^2-1}-1)\ln(x+\sqrt{x^2-1}) &\geq 1.2501\cdot 10^p, \\
4x^2\left(\sqrt{1-\tfrac{1}{x^2}} - \tfrac{1}{4x^2}\right)\cdot\ln(x + x\sqrt{1-\tfrac{1}{x^2}}) &\geq 1.2501\cdot 10^p, \\
4x^2\cdot 0.999924\cdot\ln(x\cdot 1.99994) &\geq 1.2501\cdot 10^p, \\
7.367x^2\cdot\log_{10}(1.99994x) &\geq 10^p, \\
2\operatorname{order}(x) + \operatorname{order}(\operatorname{order}(1.99994x)) &\geq p - 0.867,
\end{aligned}$$

wodurch sich momentan derselbe Schnelltest wie bei arsinh ergibt.

4.13.7 Intervallversion

Auf seinem Definitionsbereich ist der Areacosinus monoton steigend, so dass auch hier nichts zu tun ist.

4.14 Der Areatangens



Der Areatangens ist als Umkehrfunktion des hyperbolischen Tangens nur auf dem Intervall $]-1,1[$ definiert. Wegen $\operatorname{artanh}(-x) = -\operatorname{artanh}(x)$ brauchen wieder nur positive Argumente betrachtet zu werden.

Durch Auflösen von $\tanh y = \frac{e^{2y}-1}{e^{2y}+1} = x$ erhält man

$$\operatorname{artanh} x = \frac{1}{2} \ln \frac{1+x}{1-x}.$$

Die Potenzreihe ist die des Arcustangens ohne alternierendes Vorzeichen (konvergent gegen $\operatorname{artanh} x$ für $|x| < 1$).

Es ist außerdem (bis auf den Faktor 2) die, die sich beim natürlichen Logarithmus durch Addition der Reihen für $\ln(1+x)$ und $\ln(1-x)$ ergeben wird. Für Argumente $x \le \frac{1}{8}$ wird direkt diese Potenzreihe verwendet.

$$\operatorname{artanh} x \;=\; \sum_{k=0}^{\infty} \frac{x^{2k+1}}{2k+1} \;=\; x + \frac{x^3}{3} + \frac{x^5}{5} + \ldots.$$

Die Auswertung, die mit zwei Schutzziffern durchgeführt werden muss, wird im Abschnitt 4.17.4 beim natürlichen Logarithmus besprochen. Dort wird auch die Sonderbehandlung sehr kleiner Argumente berücksichtigt.

4.14.1 Berechnung über den Logarithmus

Für größere Argumente ziehen wir die obige Beziehung mit dem natürlichen Logarithmus heran. Für $x \in]0.125, 1[$ liegt dann das Argument des Logarithmus in $]\frac{9}{7}, +\infty[$; es muss also ggf. die vollständige dortige Argumentreduktion durchgeführt werden.

Bei der Argumenttransformation können der Zähler $1+x$ und der Nenner $1-x$ ohne Probleme exakt berechnet werden, ohne dass wesentliches Stellenwachstum zu erwarten ist (da ja $x \in]0.125, 1[$), ebenso die Division durch 2.

Wir rechnen nicht durchweg intervallmäßig, sondern berechnen eine Unterschranke des Arguments (Division nach unten gerundet), verwenden die Untergrenze der Einschließung des Logarithmus und erhalten eine Oberschranke durch eine Fehlerabschätzung. Die (negativen) relativen Fehler sollen $\varepsilon_/ \ge 0$ bzw. $\varepsilon_{\ln} \ge 0$ heißen. Wir haben dann also

$$\widetilde{\operatorname{artanh}}\, x = \frac{1}{2} \ln \left(\frac{1+x}{1-x}(1-\varepsilon_/) \right) (1-\varepsilon_{\ln}),$$

Es folgt

$$\widetilde{\operatorname{artanh}}\, x = \frac{1}{2} \left(\ln \left(\frac{1+x}{1-x} \right) - \frac{1}{\xi} \frac{1+x}{1-x} \varepsilon_/ \right) (1-\varepsilon_{\ln}), \qquad \frac{1+x}{1-x}(1-\varepsilon_/) < \xi < \frac{1+x}{1-x}.$$

Daraus erhalten wir

$$\begin{aligned}|\varepsilon_{\text{artanh}}| &= \left|\frac{\widetilde{\text{artanh}}\, x - \text{artanh}\, x}{\text{artanh}\, x}\right| = \left|\frac{-\varepsilon_{\ln}\ln(\frac{1+x}{1-x}) - \frac{1}{\xi}\varepsilon_{/}\frac{1+x}{1-x}(1-\varepsilon_{\ln})}{\ln\frac{1+x}{1-x}}\right| \\ &= \varepsilon_{\ln} + \frac{1+x}{1-x}\cdot\frac{1}{\xi}\cdot\frac{1}{\ln\frac{1+x}{1-x}}\cdot(1-\varepsilon_{\ln})\cdot\varepsilon_{/} \le \varepsilon_{\ln} + \frac{1-\varepsilon_{\ln}}{1-\varepsilon_{/}}\cdot\frac{1}{\ln\frac{1+x}{1-x}}\cdot\varepsilon_{/} \\ &\le \varepsilon_{\ln} + 3.9795\varepsilon_{/}.\end{aligned}$$

Offenbar sind Division und Logarithmus-Berechnung mit zwei Schutzziffern zu berechnen, und anschließend gilt (wenn das interne Ergebnis beim Logarithmus vor der Endrundung verwendet wird)

$$|\varepsilon_{\text{artanh}}| \le 0.399\cdot 10^{-p},$$

woraus man eine gesicherte Oberschranke erhält.

4.14.2 Intervallversion

Auch artanh ist monoton steigend; die Implementation ergibt sich kanonisch.

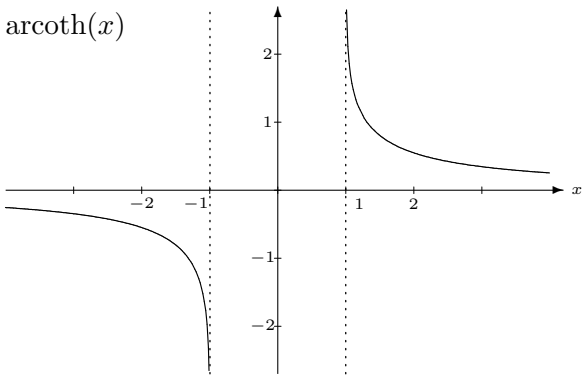
4.15 Der Areacotangens

Der Areacotangens ist die Umkehrfunktion des Cotangens hyperbolicus und daher nur auf $\mathbb{R}\setminus[-1,1]$ definiert. Durch Umstellen der Definition von coth erhält man die logarithmische Darstellung

$$\text{arcoth}\, x = \frac{1}{2}\ln\frac{x+1}{x-1},$$

bzw. den Zusammenhang

$$\text{arcoth}\, x = \text{artanh}\,\frac{1}{x}.$$



Daher ist der Areacotangens in völliger Analogie zum Areatangens implementiert. Für betragskleine Argumente x (also betragsgroße $\frac{1}{x}$) wird der Logarithmus verwendet, für betragsgroße Argumente x die Areatangens-Potenzreihe mit dem Argument $\frac{1}{x}$. Die Schranke für den Übergang wird entsprechend als 8 gewählt. Wegen $\text{arcoth}(-x) = -\,\text{arcoth}(x)$ betrachten wir hier wieder nur positive Argumente.

4.15.1 Berechnung über den Logarithmus

Für den logarithmischen Zusammenhang ($0 < x < 8$) ergibt sich fast dieselbe Fehlerabschätzung wie beim Areatangens:

$$\widetilde{\text{arcoth}}\, x = \frac{1}{2}\ln\left(\frac{x+1}{x-1}(1+\varepsilon_{/})\right)(1+\varepsilon_{\ln}) = \frac{1}{2}\left[\ln\left(\frac{x+1}{x-1}\right) + \frac{x+1}{\xi(x-1)}\,\varepsilon_{/}\right](1+\varepsilon_{\ln}),$$

$$\text{für ein } \xi \in \text{conv}\{\tfrac{x+1}{x-1}, \tfrac{x+1}{x-1}(1+\varepsilon_{/})\},$$

$$
\begin{aligned}
|\varepsilon_{\text{arcoth}}| &= \left|\frac{\widetilde{\text{arcoth}}\, x - \text{arcoth}\, x}{\text{arcoth}\, x}\right| = \left|\frac{\varepsilon_{\ln}\, \ln\frac{x+1}{x-1} \;+\; \frac{1}{\xi}\,\varepsilon_{/}\,(1+\varepsilon_{\ln})}{\ln\frac{x+1}{x-1}}\right| \\
&\le |\varepsilon_{\ln}| + \frac{x+1}{x-1}\cdot\frac{1}{\xi}\cdot\frac{1}{\ln\frac{x+1}{x-1}}\cdot(1+\varepsilon_{\ln})\,|\varepsilon_{/}| \;\le\; |\varepsilon_{\ln}| + \frac{1+\varepsilon_{\ln}}{1-|\varepsilon_{/}|}\,|\varepsilon_{/}| \\
&\le |\varepsilon_{\ln}| + 3.9795\,|\varepsilon_{/}|,
\end{aligned}
$$

(Der Term mit dem Logarithmus im Nenner ist monoton steigend und nimmt in $x = 8$ sein Maximum an.) Wie beim Areatangens ergibt sich also, dass wir bei beiden Operationen zwei Schutzziffern benötigen, und dann gilt

$$|\varepsilon_{\text{arcoth}}| \le 0.399 \cdot 10^{-p},$$

woraus sich aus einer Unterschranke (die mit gerichteten Rundungen bestimmt wird) eine Oberschranke bestimmen lässt.

4.15.2 Berechnung über die Areatangens-Reihe

Für $x \ge 8$ wird die Areatangens- (bzw. Logarithmus-) Reihe verwendet. Zusätzlich ist natürlich der Effekt der Inversion im Argument zu berücksichtigen. Es ist

$$
\widetilde{\text{arcoth}}\, x = \text{artanh}\Big(\frac{1}{x}\,(1+\varepsilon_{/})\Big) = \Big(\text{artanh}\Big(\frac{1}{x}\Big) + \frac{1}{1-\xi^2}\,\frac{1}{x}\,\varepsilon_{/}\Big)(1+\varepsilon_{\text{artanh}}),
$$

$$
\text{für ein } \xi \in \text{conv}\{\tfrac{1}{x}, \tfrac{1}{x}(1+\varepsilon_{/})\},
$$

$$
\begin{aligned}
|\varepsilon_{\text{arcoth}}| &= \left|\frac{\varepsilon_{\text{artanh}}\cdot\text{artanh}(\frac{1}{x}) \;+\; \frac{1}{x(1-\xi^2)}\,\varepsilon_{/}\,(1+\varepsilon_{\text{artanh}})}{\text{artanh}\,\frac{1}{x}}\right| \\
&\le |\varepsilon_{\text{artanh}}| + \frac{1}{x(1-\xi^2)\,\text{artanh}\,\frac{1}{x}}\,(1+\varepsilon_{\text{artanh}})\,|\varepsilon_{/}| \\
&\le |\varepsilon_{\text{artanh}}| + \frac{1+\varepsilon_{\text{artanh}}}{(x-\frac{1}{x}(1+|\varepsilon_{/}|)^2)\cdot\text{artanh}\,\frac{1}{x}}\,|\varepsilon_{/}| \quad \overset{!}{\le} b^{-p},
\end{aligned}
$$

Der Bruch ist für $x \ge 8$ monoton fallend, und man erhält für $b = 10$ die Forderung

$$|\varepsilon_{\text{artanh}}| + 1.0108 \cdot |\varepsilon_{/}| \quad \overset{!}{\le} 10^{-p},$$

d.h. $3.0108 \cdot 10^{-\ell+1} \le 10^{-p}$, bzw. $\ell \ge p + 1.479$. Für Division und Reihenberechnung werden also zwei Schutzziffern notwendig. Daraus ergibt sich

$$|\varepsilon_{\text{arcoth}}| \;\le\; 0.3011 \cdot 10^{-p},$$

was uns zur Berechnung einer Oberschranke dient, wenn wir zuvor mit gerichteten Rundungen eine Unterschranke bestimmt haben.

4.15.3 Berechnung für sehr große Argumente

Für sehr große x unterscheidet sich $\operatorname{arcoth} x$ praktisch nicht mehr von $\frac{1}{x}$, so dass wir noch einen möglichst schnellen (aber groben) entsprechenden Test suchen wollen.

Es gilt (für $x > 1$):

$$\begin{aligned}\operatorname{arcoth}(x) - \frac{1}{x} &= \operatorname{artanh}\left(\frac{1}{x}\right) - \frac{1}{x} = \frac{1}{3x^3} + \frac{1}{5x^5} + \frac{1}{7x^7} + \ldots \leq \frac{1}{3x^3}\left(1 + \frac{1}{x^2} + \frac{1}{x^4} + \ldots\right) \\ &= \frac{1}{3x^3} \cdot \frac{1}{1 - \frac{1}{x^2}} = \frac{1}{3x(x^2-1)}.\end{aligned}$$

Als Bedingung dafür, $\frac{1}{x}$ als Approximation verwenden zu können, erhält man, wenn man beim relativen Fehler das $\operatorname{arcoth} x$ im Nenner durch $\frac{1}{x}$ nach unten abschätzt,

$$3(x^2 - 1) \geq b^p.$$

Um den Test zu beschleunigen, verwenden wir $x^2 - 1 \geq 0.984375\, x^2$ für $x \geq 8$, wodurch sich die Bedingung (bei $b = 10$) vereinfacht zu $x \geq 0.582 \cdot 10^{p/2}$.

In der Implementation wird nun zuerst grob getestet, ob $\operatorname{order}(x) \geq \lfloor \frac{x}{2} \rfloor - 1$, und falls ja, bei geradem p die Bedingung $\operatorname{order}(1.71\, x) \geq \frac{p}{2}$, bei ungeradem p die Bedingung $\operatorname{order}(0.543\, x) \geq \lfloor \frac{p}{2} \rfloor$. Bei Erfüllen wird für arcosh nur eine Unterschranke von $\frac{1}{x}$ und aus der Abschätzung eine Oberschranke berechnet.

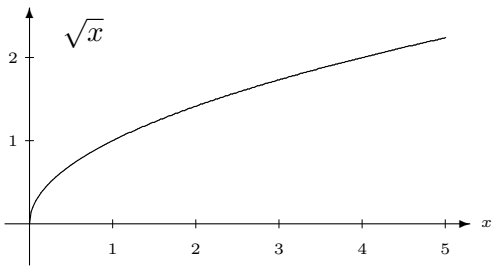
4.15.4 Intervallversion

Die Intervallversion muss natürlich testen, ob das Argument Elemente von $[-1, 1]$ enthält und ggf. eine Exception werfen. Für die anderen Fälle kann sie durch die fallende Monotonizität kanonisch implementiert werden.

4.16 Die Quadratwurzel

Die Ableitungen der Quadratwurzel $f(x) = \sqrt{x}$ lauten ($n \geq 1$):

$$\begin{aligned}f^{(n)}(x) &= \frac{1}{2}\left(-\frac{1}{2}\right)\left(-\frac{3}{2}\right)\left(-\frac{5}{2}\right)\ldots\left(\frac{3}{2} - n\right) x^{-\frac{2n-1}{2}} \\ &= n! \binom{1/2}{n} x^{-\frac{2n-1}{2}},\end{aligned}$$



wobei

$$\binom{1/2}{n} = (-1)^{n+1} \cdot \frac{1 \cdot 3 \cdot 5 \cdot 7 \cdot \ldots \cdot (2n-3)}{2 \cdot 4 \cdot 6 \cdot 8 \cdot \ldots \cdot (2n)},$$

und damit ergibt sich die Taylor-Entwicklung mit Restglied wie folgt:

$$\sqrt{1+x} = 1 + \frac{1}{2}x - \frac{1}{8}x^2 + \frac{3}{48}x^3 - \frac{15}{384}x^4 + \frac{105}{3840}x^5 - + \ldots,$$

$$\sqrt{1+x} = 1 + \binom{1/2}{1} x + \binom{1/2}{2} x^2 + \binom{1/2}{3} x^3 + \ldots + \binom{1/2}{N} x^N + \binom{1/2}{N+1} \xi^{-N-\frac{1}{2}} x^{N+1},$$

mit $\xi \in \mathrm{conv}\{1, 1+x\}$. Der Quotient aufeinander folgender Glieder geht damit gegen -1. Die Reihe konvergiert für $|x| < 1$, aber recht langsam. Als wesentlich schnellere Alternative werden wir daher (wie auch [Steins] und [Braune]) das Newton-Verfahren zur Nullstellenbestimmung bemühen.

Die Implementation zu [Steins] verwendet eine aus REDUCE übernommene Approximation der Quadratwurzel, die die genügende Genauigkeit selbst noch nicht garantieren kann, und benutzt diese nach geeigneter Aufblähung als Startintervall eines Intervall-Newton-Verfahrens.

Wir verfahren etwas anders und führen zunächst das normale Newton-Verfahren durch, um verifiziert eine Oberschranke der Quadratwurzel zu bestimmen. Als Abbruchbedingung verwenden wir, dass sich keine Verbesserung mehr erzielen lässt. Es stellt sich heraus, dass bei genügend erhöhter Stellenzahl diese Oberschranke garantiert nah genug am exakten Wert liegt und sich durch Quadrieren eine Einschließung finden lässt.

4.16.1 Argumentreduktion

Die Argumentreduktion ist für die Quadratwurzel besonders einfach, da die Fließkommazahlen ja bereits in einer logarithmischen Darstellung vorliegen:

$$x = m_x \cdot b^{e_x}, \quad m_x \in [\tfrac{1}{b}, 1[.$$

Für einen geraden Exponenten e_x erhält man direkt

$$\sqrt{x} = \sqrt{m_x} \cdot b^{e_x/2}, \quad \text{Argument } m_x \in [\tfrac{1}{b}, 1[,$$

d.h. die Hintransformation ist das Setzen des Exponenten auf 0, die Rücktransformation das Setzen des Exponenten auf $e_x/2$. Für ungerade e_x wird ein Faktor b^{-1} in die Mantisse abgespalten:

$$\sqrt{x} = \sqrt{m_x \cdot b^{-1}} \cdot b^{(e_x+1)/2}, \quad \text{Argument } m_x \cdot b^{-1} \in [\tfrac{1}{b^2}, \tfrac{1}{b}[,$$

d.h. die Hintransformation ist das Setzen des Exponenten auf -1, die Rücktransformation das Setzen des Exponenten auf $(e_x + 1)/2$. Alle Transformationen sind also Manipulationen des Exponenten und fehlerfrei durchführbar.

4.16.2 Erste Näherung

Das Newton-Verfahren konvergiert in unserem Fall (außer für den Startwert 0) global (genauer weiter unten), und zwar für positive Startwerte gegen $+\sqrt{x}$, für negative gegen $-\sqrt{x}$. Man kann sich aber ggf. viele Iterationsschritte sparen, wenn man den Startwert bereits geschickt wählt.

Die ersten Glieder der Potenzreihe liefern völlig unakzeptable Näherungen für kleine x. Lineare Interpolation auf $[\frac{1}{100}, \frac{1}{10}[$ bzw. $[\frac{1}{10}, 1[$ ist sinnvoll, das Intervall sollte aber in einige Teilintervalle zerlegt werden. So gehen die meisten Implementationen auch vor. Aufwändigere Interpolationen lohnen angesichts der schnellen Konvergenz der Iteration nicht.

Wenn allerdings bereits eine andere (nicht-dynamische) Arithmetik zur Verfügung steht, kann diese sehr gut verwendet werden (besonders wenn sie in Hardware vorliegt). In der vorliegenden Implementation wird daher einfach die Routine `double sqrt(double)` aus der `C`-Standardbibliothek verwendet, die zumindest bereits 15 korrekte Stellen liefern sollte.

Natürlich ist eine Umwandlung des Langzahlarguments nach `double` und zurück nötig, die sich bei unseren Tests zeitlich aber immer deutlich gelohnt hat. Gegenüber einer schnelleren Näherung mit beispielsweise nur 2 Stellen spart man sich bei quadratischer Konvergenz ca. drei Iterationsschritte.

4.16.3 Newton-Verfahren

Das Newton-Verfahren, angewandt auf $f(y) := y^2 - x$, liefert als exakte Iterationsvorschrift

$$y_{n+1} = \frac{1}{2}\Big(y_n + \frac{x}{y_n}\Big).$$

Wir schreiben $y := \sqrt{x}$ für den exakten gesuchten Wert. Für einen positiven Startwert y_0 sind alle Folgenglieder ab y_1 größer oder gleich y:

$$y_{n+1} - y = \frac{1}{2}\left(y_n + \frac{y^2}{y_n}\right) - y = \frac{1}{2}\left(\frac{y_n^2 + y^2}{y_n}\right) - y = \frac{y_n^2 + y^2 - 2yy_n}{2y_n} = \frac{(y_n - y)^2}{2y_n} \geq 0$$

Außerdem gilt für $n \geq 1$

$$y_n - y_{n+1} = \frac{1}{2}\Big(y_n - \frac{x}{y_n}\Big) = \frac{y_n^2 - y^2}{2y_n} = \frac{(y_n - y)(y_n + y)}{2y_n} \geq 0,$$

d.h. die Folgenglieder sind monoton fallend.

Wir werden das Newton-Verfahren daher naheliegenderweise dazu verwenden, die Obergrenze einer Einschließung zu bestimmen. Entsprechend werden wir bei der Division und am Ende eines Schritts nach oben auf ℓ Stellen runden (mit noch zu bestimmendem ℓ).

Die inexakten Folgenglieder sollen $\tilde{y}_n$ heißen, der Fehler aus der Division $0 \leq \varepsilon_/ \leq b^{-\ell+1}$ und der aus der Endrundung $0 \leq \varepsilon_r \leq b^{-\ell+1}$ (Addition und Halbierung sind exakt). Der Gesamtfehler nach Schritt n sei ε_n, d.h. $\tilde{y}_n = y(1+\varepsilon_n)$. Es gilt:

$$\begin{aligned}
\tilde{y}_{n+1} &= \frac{1}{2}\Big(\tilde{y}_n + \frac{x}{\tilde{y}_n}(1+\varepsilon_/)\Big)(1+\varepsilon_r) & &= \frac{1}{2}\left(y(1+\varepsilon_n) + \frac{y^2(1+\varepsilon_/)}{y(1+\varepsilon_n)}\right)(1+\varepsilon_r) \\
&= \frac{y}{2}\left(\frac{1+\varepsilon_/}{1+\varepsilon_n} + (1+\varepsilon_n)\right)(1+\varepsilon_r) & &= y\left(1 + \frac{1+\varepsilon_/}{2(1+\varepsilon_n)} + \frac{\varepsilon_n - 1}{2}\right)(1+\varepsilon_r) \\
&= y\left(1 + \frac{\varepsilon_/ + \varepsilon_n^2}{2(1+\varepsilon_n)}\right)(1+\varepsilon_r) & &= y\left(1 + \varepsilon_r + (1+\varepsilon_r)\frac{\varepsilon_/ + \varepsilon_n^2}{2(1+\varepsilon_n)}\right) \\
\Rightarrow \varepsilon_{n+1} &= \varepsilon_r + (1+\varepsilon_r)\frac{\varepsilon_/ + \varepsilon_n^2}{2(1+\varepsilon_n)} & &= \varepsilon_r + \frac{1+\varepsilon_r}{2(1+\varepsilon_n)}\varepsilon_/ + \frac{1+\varepsilon_r}{2(1+\varepsilon_n)}\varepsilon_n^2,
\end{aligned}$$

Wegen der gerichteten Rundungen sind alle Fehler nicht negativ. Der Term ε_n^2 zeigt die quadratische Konvergenz auf (vgl. auch [Braune], Seite 120, dort mit zusätzlich inexakter

Addition). Der Quotient im letzten Ausdruck ist ≤ 0.50005 für $b = 10$ und Division und Rundung mit $\ell \geq 5$ Stellen, d.h. die ersten beiden Summanden sind durch $1.50005 \cdot 10^{-\ell+1}$ beschränkt. Wenn ε_n überhaupt kleiner als 10^{-p} werden können soll, benötigen wir also zwei Schutzziffern.

Solange $\varepsilon_n \gg \varepsilon_/, \varepsilon_r$, ist garantiert, dass auch die $\tilde{y}_n$ streng monoton fallen. Wenn $\tilde{y}_{n+1} \geq \tilde{y}_n$, ist ε_n in die Nähe der Rundungsfehler gefallen, und es ist keine Verbesserung mehr erreichbar. Es ist also $\tilde{y}_n$ unsere beste Näherung.

Aus $\varepsilon_{n+1} \geq \varepsilon_n$ und Einsetzen der obigen Beziehung (und unserer Fehlerschranken) folgt

$$\begin{aligned}
2(1+\varepsilon_n)\varepsilon_r + (1+\varepsilon_r)(\varepsilon_/ + \varepsilon_n^2) &\geq 2(1+\varepsilon_n)\varepsilon_n, \\
(1+\varepsilon_r)(\varepsilon_/ + \varepsilon_n^2) &\geq 2(1+\varepsilon_n)(\varepsilon_n - \varepsilon_r), \\
\varepsilon_n^2(\varepsilon_r - 1) + 2\varepsilon_n(\varepsilon_r - 1) &\geq -2\varepsilon_r - \varepsilon_/(1+\varepsilon_r), \\
\varepsilon_n^2 + 2\varepsilon_n &\leq \frac{2\varepsilon_r + \varepsilon_/(1+\varepsilon_r)}{1-\varepsilon_r},
\end{aligned}$$

$$\begin{aligned}
\varepsilon_n + 1 &\leq \sqrt{1 + \frac{2\varepsilon_r + \varepsilon_/ + \varepsilon_r\varepsilon_/}{1-\varepsilon_r}} \leq 1 + \frac{2\varepsilon_r + \varepsilon_/ + \varepsilon_r\varepsilon_/}{2(1-\varepsilon_r)}, \\
\varepsilon_n &\leq \frac{3 + b^{-4}}{2(1-b^{-4})} \cdot b^{-\ell+1} \leq 1.634 \cdot b^{-\ell+1},
\end{aligned}$$

bzw. besser $\leq 1.5003 \cdot 10^{-\ell+1}$ für $b = 10$. Dieses $\tilde{y}_n \geq y$ genügt also (mit $\ell \geq p+2$) unseren Genauigkeitsanforderungen für das p-stellige Raster.

Wir runden nun $\tilde{y}_n$ *nach unten* in das p-stellige Endraster:

$$\hat{y} := \bigtriangledown_p(\tilde{y}_n).$$

Falls nun $\hat{y}^2 \geq x$ (exakt durchführbar), so ist $\hat{y} \geq y$, und $\hat{y}$ wird die Obergrenze unserer Einschließung. Als Untergrenze verwenden wir den Vorgänger $\underline{y} = \mathrm{pred}_p(\hat{y})$ im p-stelligen Raster. Es gilt $\underline{y} < y$, da (mit $\hat{y} = y_n(1-\varepsilon')$, $\varepsilon' \leq 10^{-p+1}$, und $\underline{y} = \hat{y}(1-\varepsilon'')$, $\varepsilon'' \geq 10^{-p}$):

$$\frac{y - \underline{y}}{y} = 1 - (1+\varepsilon_n)(1-\varepsilon')(1-\varepsilon'') = \varepsilon''(1 - \varepsilon' + \varepsilon_n - \varepsilon_n\varepsilon') + \varepsilon'(1+\varepsilon_n) - \varepsilon_n \geq 0.82 \cdot 10^{-p} > 0.$$

Falls dagegen $\hat{y}^2 < x$, also $\hat{y} < y$, so verwenden wir $\hat{y}$ als die Untergrenze unserer Einschließung und als Obergrenze den Nachfolger $\bar{y} := \bigtriangleup_p \tilde{y}_n = \mathrm{succ}_p(\hat{y})$. Wir erhalten in beiden Fällen per Konstruktion maximale Genauigkeit.

Bemerkung: Die Iterationsvorschrift im obigen Verfahren leidet darunter, dass sie eine Division enthält, also die mit Abstand langsamste der Grundrechenarten bei unserer Langzahlarithmetik. Alternativ kann man zunächst $\frac{1}{\sqrt{x}}$ approximieren ($f(x) = \frac{1}{y^2} - x$) und anschließend *eine einzige* Division durchführen. Man erhält die (monoton steigende) Iteration

$$y_{n+1} := y_n - \frac{1/y^2 - x}{-2/y^3} = y_n + \frac{y_n - xy_n^3}{2} = y_n\left(\frac{3 - xy_n^2}{2}\right),$$

Leider sind nun hier drei Multiplikationen beteiligt. Es hängt also von der genauen Implementation von Multiplikation und Division auf der jeweiligen Maschine ab, ob sich diese Version lohnt. Zusätzlich braucht man eine Division am Schluss, die (analog zur Reduktion bei der Exponentialfunktion) zwei Schutzziffern erfordert, d.h. das Verfahren muss insgesamt mit vier Schutzziffern laufen.

Messungen auf den verwendeten Testrechnern ergaben, dass dieses geänderte Verfahren (im relevanten Stellenbereich) langsamer ist als das erste. Erst bei sehr hohen Stellenzahlen gewinnen die drei (rekursiv implementierten) Multiplikationen gegenüber der Division. Die vorliegende Implementation verwendet immer die erste angegebene Iteration.

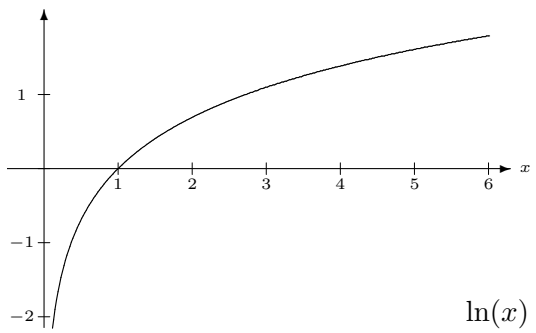
4.16.4 Intervallversion

Auch $\sqrt{x}$ ist auf seinem Definitionsbereich monoton und daher kanonisch intervallmäßig zu implementieren.

4.17 Der natürliche Logarithmus

Für den natürlichen Logarithmus boten sich zwei Implementationsmöglichkeiten an: über die Potenzreihe mit einer Argumentreduktion bzw. mit dem Newton-Verfahren.

Mit der Potenzreihe wird beispielsweise in [Krämer] gearbeitet; es ist eine recht aufwändige Argumentreduktion erforderlich. Vermutlich um diese zu umgehen, verwendet [Steins] ausschließlich ein (Intervall-)Newton-Verfahren, das allerdings ziemlich langsam ist und um 1 herum mit Auslöschungsproblemen kämpft. (Für den verwandten Areatangens wird bei [Steins] dagegen genau diese Potenzreihe benutzt.)



ln(x)

Im Rahmen dieser Arbeit wurden beide Möglichkeiten implementiert. Das Newton-Verfahren erfordert allerdings in jedem Schritt die Auswertung der Exponentialfunktion und erwies sich für größere Stellenzahlen als nicht mehr sinnvoll durchführbar. Die endgültige Version der Implementation arbeitet daher ausschließlich mit der Potenzreihe.

4.17.1 Die Potenzreihe

Die Taylor-Entwicklung des natürlichen Logarithmus ln um 1 lautet (konvergent für $|x| < 1$):

$$\ln(1+x) \;=\; \sum_{k=1}^{\infty} (-1)^{k-1} \frac{x^k}{k} \;=\; x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - + \ldots .$$

Zur Verbesserung der Konvergenzgeschwindigkeit kann man die Reihe für $1+x$ und $1-x$ voneinander subtrahieren:

$$\ln \frac{1+x}{1-x} = 2x \left(1 + \frac{x^2}{3} + \frac{x^4}{5} + \frac{x^6}{7} + \frac{x^8}{9} + \ldots \right).$$

Wir erhalten so bis auf den Faktor 2 die Reihe des Areatangens (d.h. die des Arcustangens ohne alternierendes Vorzeichen), was durch die entsprechende algebraische Beziehung der beiden Funktionen klar ist:

$$\tanh x = \frac{e^{2x}-1}{e^{2x}+1}, \qquad \ln\frac{1+x}{1-x} = 2\ \operatorname{artanh} x.$$

Da die Argumentreduktion aber stark logarithmusbezogen ist und der Logarithmus vermutlich die wichtigere der beiden Funktionen, wird die Auswertung dieser Reihe in diesem Kapitel besprochen.

In diesem Kapitel stehe ab jetzt x immer für das eigentliche Argument des Logarithmus, u für das sich ergebende Argument der Potenzreihe, also

$$x = \frac{1+u}{1-u}, \qquad u = \frac{x-1}{x+1}, \qquad \ln x = \ln\frac{1+u}{1-u} = 2\sum_{k=0}^{\infty}\frac{u^{2k+1}}{2k+1}.$$

Da die Potenzreihe beim Areatangens direkt (ohne die Argumenttransformation) verwendet wird, ist sie als eigene Routine implementiert.

Der Bereich, in dem wir die Potenzreihe direkt auswerten wollen, ist $u \in [-\frac{1}{8}, +\frac{1}{8}]$, woraus sich für das Logarithmusargument ergibt $x \in [0.\overline{7}, 1.\overline{285714}]$. Wir werden genauer $x \in [0.78, 1.25]$ verwenden.

4.17.2 Argumentreduktion

Argumente x außerhalb des Intervalls $[0.78, 1.25]$ müssen auf dieses reduziert werden.

Die Betrachtungen in diesem Abschnitt hängen teilweise von der verwendeten Basis b des Fließkommasystems ab. Beispielsweise wird in Schritt 2) ausgenutzt, dass in unserer Implementation mit $b = 10$ exakt durch 2 bzw. durch 5 dividiert werden kann. Wir setzen hier daher sinnvollerweise fest $b = 10$ voraus. Die Reduktionen lassen sich aber leicht auf allgemeine Basen übertragen. Man sollte aber beachten, dass beispielsweise bei $b = 2$ exakte Divisionen nur durch Zweierpotenzen möglich sind, wodurch man in 2) nie ohne zusätzliche Rundungsfehler auskommen wird.

1) Es wird die ohnehin halblogarithmische Darstellung der Fließkommazahlen ausgenutzt. Es ist $x = x_m \cdot 10^{c_{10}}$, wobei $x_m \in [1, 10[$ und $c_{10} \in \mathbb{Z}$. Dann verwendet man

$$\ln(x) = \ln(x_m \cdot 10^{c_{10}}) = \ln(x_m) + c_{10} \cdot \ln(10),$$

wobei die Konstante $\ln 10$ meist schon mit genügender Genauigkeit vorliegen sollte.

Für Argumente $x_m \in [7.8, 10[$ führen wir hier eine Korrektur durch:

$$x_m := x_m/10, \quad c_{10} := c_{10} + 1,$$

damit x_m direkt im Bereich der Potenzreihenauswertung zu liegen kommt.

Nach diesem Schritt gilt also $x_m \in [0.78, 7.8[$, weiter zu reduzieren ist noch $]1.25, 7.8[$.

2) Letzteres Intervall wird in einige Teilintervalle unterteilt, die jeweils durch Multiplikation mit einem festen Faktor in das Intervall der Potenzreihenauswertung hinein abgebildet werden. Man nutzt

$$\ln(x_m) = \ln(f \cdot x') = \ln(f) + \ln(x'), \qquad x' := \frac{x_m}{f}.$$

Um einen Rundungsfehler bei der Division zu vermeiden, verwenden wir nur rationale Zahlen mit Primfaktoren 2 und 5 im Nenner. Es werden die Konstanten $\ln 2$ und $\ln 5$ benötigt. Da wir nicht noch weitere Konstanten berechnen müssen wollen, wählen wir die Faktoren so, dass auch im Zähler nur 2 und 5 vorkommen. Es ist dann

$$f = 2^{c_2} \cdot 5^{c_5},\ c_2,\, c_5 \in \mathbb{Z}, \qquad \ln x_m = c_2 \cdot \ln 2 + c_5 \cdot \ln 5 + \ln x'.$$

In [Krämer] findet sich ein Vorschlag zur Konstruktion einer Unterteilung in Teilintervalle, die wir hier wegen der Beschränkung der Faktoren aber nicht verwenden können. Unsere per Hand konstruierte Unterteilung sieht wie folgt aus:

a)	$[6.25, 7.8[$	$\xrightarrow{/2^3}$	$[0.78125, 0.975[\,,$	$f = 8,$	$\frac{1}{f} = 0.125,$	$c_2 = 3,$	$c_5 = 0,$
b)	$[4, 6.25[$	$\xrightarrow{/5}$	$[0.8, 1.25[\,,$	$f = 5,$	$\frac{1}{f} = 0.2,$	$c_2 = 0,$	$c_5 = 1,$
c)	$[3.125, 4[$	$\xrightarrow{/2^2}$	$[0.78125, 1[\,,$	$f = 4,$	$\frac{1}{f} = 0.25,$	$c_2 = 2,$	$c_5 = 0,$
d)	$[1.95, 3.125[$	$\xrightarrow{/5\ *2}$	$[0.78, 1.25[\,,$	$f = 2.5,$	$\frac{1}{f} = 0.4,$	$c_2 = -1,$	$c_5 = 1,$
e)	$[1.375, 1.95[$	$\xrightarrow{/5^2\ *2^4}$	$[0.88, 1.248[\,,$	$f = 1.5625,$	$\frac{1}{f} = 0.64,$	$c_2 = -4,$	$c_5 = 2,$
f)	$]1.25, 1.375[$	$\xrightarrow{/5\ *2^2}$	$]1, 1.1[\,,$	$f = 1.25,$	$\frac{1}{f} = 0.8,$	$c_2 = -2,$	$c_5 = 1,$
g)	$[0.78, 1.25[$	$\longrightarrow$	$[0.78, 1.25]\,,$	$f = 1,$	$\frac{1}{f} = 1,$	$c_2 = 0,$	$c_5 = 0.$

Da $\ln 10 = \ln 2 + \ln 5$, kann die Reduktion aus Schritt 1) mit dieser verschmolzen werden:

$$\ln x = (c_2 + c_{10}) \cdot \ln 2 + (c_5 + c_{10}) \cdot \ln 5 + \ln x',$$

wobei nicht $c_2 = 0 = c_5$, falls $c_{10} = 0$ (dann liegt das Argument schon im gewünschten Bereich). Es ist $\ln f = c_2 \ln 2 + c_5 \ln 5$.

3) Alternativ können in z.B. $[1, 2]$ auch ein oder mehrere Schritte mit der Quadratwurzel durchgeführt werden, folgende Beziehung benutzend:

$$\ln(x) = 2 \cdot \ln(\sqrt{x}).$$

Die Quadratwurzel ist zwar relativ schnell implementiert, die Versionen aus Teil 2) sind aber deutlich schneller (sofern die Konstanten nicht neu mit höherer Genauigkeit berechnet werden müssen). Wir verwenden diese Reduktionen nicht.

4.17.3 Auswertung der Potenzreihe

Wegen der Symmetrie braucht die Potenzreihe nur für positive Argumente untersucht zu werden. Dort liefert sie, nach dem N-ten Glied abgebrochen, eine Unterschranke für den exakten Wert.

Analog zum Arcustangens ergibt sich das wie folgt geklammerte Approximationspolynom ($v := u^2$):

$$\begin{aligned} P_N(x) &= u\left(1 + \frac{v}{3} + \frac{v^2}{5} + \frac{v^3}{7} + \ldots + \frac{v^N}{2N+1}\right) \\ &= u\left(1 + v\left(\frac{1}{3} + v\left(\frac{1}{5} + v\left(\frac{1}{7} + \ldots + v \cdot \frac{1}{2N+1}\right)\right)\ldots\right)\right) \end{aligned}$$

Die Koeffizienten sind nicht fakultätsähnlich, so dass wir das normale Horner-Verfahren verwenden müssen:

$$\begin{aligned} &a) \quad r_N := \frac{1}{2N+1}, \\ &b) \quad r_k := \frac{1}{2k+1} + r_{k+1} \cdot v \qquad \text{für } k = N-1, \ldots, 1, \\ &c) \quad r_0 := 1 + v \cdot r_1 \\ &\quad \Rightarrow \quad P(u) = u \cdot r_0. \end{aligned}$$

4.17.4 Auswertefehler beim Horner-Verfahren

Zur Abschätzung des Auswertefehlers können wir Korollar 6 anwenden. Mit

$$r_k = \sum_{i=0}^{N-k} \frac{u^{2i}}{2k+2i+1}$$

liefert es folgende Aussage:

$$\begin{aligned} \Delta_P \in E_\ell \Big[& r_0 u + r_1 u^3 (1+E_\ell) + (2+E_\ell)(1+E_\ell) \overbrace{\sum_{k=1}^{N} \sum_{i=0}^{N-k} \frac{u^{2k+2i+1}}{2k+2i+1}}^{A} \\ & + 2E_\ell(2+E_\ell)(1+E_\ell) \underbrace{\sum_{k=1}^{N} \sum_{i=0}^{N-k} k \, \frac{u^{2k+2i+1}}{2k+2i+1}}_{B} \Big]. \end{aligned}$$

Für die beiden Summen A und B gilt (mit $N \geq 3$):

$$\begin{aligned} A &= \sum_{k=1}^{N} k \cdot \frac{u^{2k+1}}{2k+1} = \frac{1}{2} \sum_{k=1}^{N} (2k+1) \frac{u^{2k+1}}{2k+1} - \frac{1}{2} \sum_{k=1}^{N} \frac{u^{2k+1}}{2k+1} \\ &= \frac{u^3}{2} \sum_{k=0}^{N} u^{2k} - \frac{1}{2} \sum_{k=1}^{N} \frac{u^{2k+1}}{2k+1} \leq \frac{u^3}{2} \cdot \frac{1}{1-u^2} - \frac{u^3}{6} - \frac{u^5}{10}, \end{aligned}$$

$$
\begin{aligned}
B &= \sum_{k=1}^{N} \frac{k(k+1)}{2} \cdot \frac{u^{2k+1}}{2k+1} = \frac{1}{2}\sum_{k=1}^{N} k^2\,\frac{u^{2k+1}}{2k+1} + \frac{1}{2}\sum_{k=1}^{N} k\,\frac{u^{2k+1}}{2k+1} \\
&= \frac{1}{4}\sum_{k=1}^{N}(2k+1)k\,\frac{u^{2k+1}}{2k+1} + \frac{1}{4}\sum_{k=1}^{N} k\,\frac{u^{2k+1}}{2k+1} = \frac{u^2}{8}\sum_{k=1}^{N} 2k\,u^{2k-1} + \frac{1}{8}\sum_{k=1}^{N} 2k\,\frac{u^{2k+1}}{2k+1} \\
&= \frac{u^2}{8}\sum_{k=1}^{N}(u^{2k})' + \frac{1}{8}\sum_{k=1}^{N}(2k+1)\,\frac{u^{2k+1}}{2k+1} - \frac{1}{8}\sum_{k=1}^{N}\frac{u^{2k+1}}{2k+1} \\
&= \frac{u^2}{8}\Big(\frac{u^2-u^{2N+2}}{1-u^2}\Big)' + \frac{u^3}{8}\sum_{k=0}^{N-1} u^{2k} - \frac{1}{8}\sum_{k=1}^{N}\frac{u^{2k+1}}{2k+1} \\
&= \frac{u^2(2u-(2N+2)u^{2N+1}+2Nu^{2N+3})}{8(1-u^2)^2} + \frac{u^3}{8}\sum_{k=0}^{N-1} u^{2k} - \frac{1}{8}\sum_{k=1}^{N}\frac{u^{2k+1}}{2k+1} \\
&\le \frac{u^3}{4(1-u^2)^2} + \frac{u^3}{8(1-u^2)} - \frac{u^3}{24} - \frac{u^5}{40}.
\end{aligned}
$$

Mit $r_1 = \frac{r_0-1}{u^2}$ und der Abschätzung von r_0 nach unten über die ersten beiden Terme, $n(u) := 1 + \frac{u^2}{3}$, erhält man für den relativen Fehler:

$$
\begin{aligned}
\varepsilon_P &\in E_\ell\left[1 + (1 - \frac{1}{r_0}(1+E_\ell) + \frac{A}{ur_0}(2+E_\ell)(1+E_\ell) + \frac{2B}{ur_0}E_\ell(1+E_\ell)(2+E_\ell)\right] \\
&\subseteq E_\ell\Big[1 + \Big(1 - \frac{u}{\operatorname{artanh} u}\Big)(1+E_\ell) + \frac{1}{n(u)}\Big(2+E_\ell)(1+E_\ell)(\frac{u^2}{2(1-u^2)} - \frac{u^2}{6} - \frac{u^4}{10}\Big) \\
&\qquad + \frac{1}{n(u)}E_\ell(1+E_\ell)(2+E_\ell)\Big(\frac{u^2}{2(1-u^2)^2} + \frac{u^2}{4(1-u^2)} - \frac{u^2}{12} - \frac{u^4}{20}\Big)\Big].
\end{aligned}
$$

Die Faktoren vor den Fehlertermen sind auf $0 < u \le \frac{1}{8}$ monoton steigend. Mit $b = 10$, $\ell \ge 5$ haben wir nun insgesamt

$$
\begin{aligned}
\varepsilon_P &\in E_\ell\,[1 + 0.0052307372 + 0.010562435 + 2.1325043 \cdot 10^{-5}] \\
&\subseteq 1.015796 \cdot E_\ell.
\end{aligned}
$$

Die Plausibilitätskontrolle auf dem Rechner ergibt einen Faktor 1.015793. In [Steins] wird (beim Areatangens) 1.0264 errechnet. Jedenfalls brauchen wir zwei Schutzziffern.

Wir erhalten als Schranke für den Approximationsfehler

$$
\begin{aligned}
|\varepsilon_{app}| &\le 10^{-p} \cdot \frac{1 - 1.015796 \cdot 10^{-1}}{1 + 1.015796 \cdot 10^{-4}}, \\
\text{bzw.} \qquad |\varepsilon_{app}| &\le 8.9832 \cdot 10^{-p-1}.
\end{aligned}
$$

4.17.5 Approximationsfehler

Für den relativen Approximationsfehler gilt (sehr kurz in [Krämer], S. 74):

$$
\begin{aligned}
\varepsilon_{app} &= \sum_{k=N+1}^{\infty}\frac{u^{2k+1}}{2k+1} \Big/ \sum_{k=0}^{\infty}\frac{u^{2k+1}}{2k+1} = \sum_{k=N+1}^{\infty}\frac{u^{2k}}{2k+1} \Big/ \sum_{k=0}^{\infty}\frac{u^{2k}}{2k+1} \\
&\le \frac{1}{2N+3}\sum_{k=N+1}^{\infty} u^{2k}\Big/1 = \frac{u^{2N+2}}{2N+3}\cdot\frac{1}{1-u^2},
\end{aligned}
$$

d.h. wir suchen ein kleines N, so dass

$$\frac{u^{2N+2}}{2N+3} \cdot \frac{1}{1-u^2} \;\le\; 8.9832 \cdot 10^{-p-1},$$

implementiert als

$$u^{2N+2} \;\le\; (2N+3) \cdot [\,(1-u^2) \cdot 8.9832 \cdot 10^{-p-1}\,],$$

wobei ein Startwert für N aus folgender Tabelle interpoliert wird:

p	$j=1$	2	3	4	5	6	7	8	9	10
3	3	3	3	3	3	3	3	3	3	3
10	3	3	3	3	3	4	4	4	4	4
33	8	9	11	12	13	14	14	15	16	17
100	25	30	34	37	40	43	46	49	51	54
330	86	102	114	125	136	145	154	163	172	181
1000	262	311	349	383	414	443	471	498	525	551

Im Hinblick auf die noch erforderliche Argumenttransformation sollte die berechnete Einschließung hier nicht von ℓ auf $p = \ell - 2$ Stellen gerundet werden. Sie trägt also einen relativen Fehler $\le b^{-p}$.

4.17.6 Auswertung für sehr kleine Argumente

Für sehr kleine $x > 0$ gilt bei der Auswertung der Potenzreihe $\operatorname{artanh}(x) \gtrapprox x$, so dass man ggf. als Einschließung $[x\,,\, x(1+b^{-p})]$ verwenden kann. Genauer hat man für $0 < x \le 10^{-3}$:

$$\begin{aligned}\frac{\operatorname{artanh} x - x}{x} &= \frac{\frac{x^3}{3} + \frac{x^5}{5} + \frac{x^7}{7} + \ldots}{x} \;\le\; \frac{x^2}{3}\,(1 + x^2 + x^4 + \ldots) \\ &= \frac{1}{3(1-x^2)}\,x^2 \;\le\; 0.333334\,x^2 \quad \overset{!}{\le}\; b^{-p},\end{aligned}$$

bzw. gröber $x \le 1.732 \cdot b^{-p/2}$. Es kann also derselbe Test wie beim Sinus verwendet werden.

4.17.7 Fehler bei der Argumenttransformation

Wir müssen noch den Fehler untersuchen, den wir durch die Argumenttransformation $x \mapsto u = u(x) = \frac{x-1}{x+1}$ machen. Die Reihe ist für positive Argumente implementiert (Symmetrie), d.h. wir berechnen immer eine betragsmäßige Unterschranke (eine Unterschranke für $x > 1$, eine Oberschranke für $x < 1$). Entsprechend wird die Division bei u gerundet. Addition und Subtraktion bei u können ohne großes Mantissenwachstum exakt durchgeführt werden, da $x \in [0.78, 1.25]$, also:

$$\tilde{u} = \frac{x-1}{x+1}\,(1-\varepsilon_1).$$

Für $x \in]1, x_{max}]$ erhält man

$$\begin{aligned}\ln\Big(\frac{1+\tilde{u}}{1-\tilde{u}}\Big) &= \ln\Big(\frac{x+1+(x-1)(1-\varepsilon_1)}{x+1-(x-1)(1-\varepsilon_1)}\Big) \;=\; \ln\Big(x - \frac{x^2-1}{2-\varepsilon_1(1-x)}\,\varepsilon_1\Big) \\ &= \ln(x) - \frac{1}{\xi} \cdot \frac{x^2-1}{2-\varepsilon_1(1-x)} \cdot \varepsilon_1\end{aligned}$$

für eine Zwischenstelle $\xi = \xi_x$ mit

$$1 \;<\; x - \frac{x^2-1}{2-\varepsilon_1(1-x)}\varepsilon_1 \;<\; \xi_x \;<\; x,$$

und damit für den relativen Gesamtfehler (ps=„power series“, ε_2 sei der Auswertefehler der Reihe):

$$\begin{aligned}|\varepsilon_{ps}| &= \left|\frac{(1-\varepsilon_2)\ln(\frac{1+\tilde{u}}{1-\tilde{u}}) - \ln x}{\ln x}\right| = \frac{1}{\ln x}\left|-\varepsilon_2 \ln x - \frac{1}{\xi_x}\cdot\frac{x^2-1}{2+\varepsilon_1(x-1)}(1-\varepsilon_2)\varepsilon_1\right| \\ &\leq \varepsilon_2 + \frac{1}{\xi_x}\cdot\frac{x^2-1}{\ln x}\cdot\frac{1-\varepsilon_2}{2+\varepsilon_1(x-1)}\cdot\varepsilon_1 \;\leq\; \varepsilon_2 + 1.009\,\varepsilon_1,\end{aligned}$$

Letzteres für $b = 10$, $\ell \geq 5$ und $x_{max} = 1.25$.

Analog gilt für $x \in [x_{min}, 1[$:

$$\ln\left(\frac{1+\tilde{u}}{1-\tilde{u}}\right) = \ln(x) + \frac{1}{\xi}\cdot\frac{1-x^2}{2+\varepsilon_1(x-1)}\cdot\varepsilon_1, \qquad x \;<\; \xi \;<\; x + \frac{1-x^2}{2-\varepsilon_1(1-x)}\varepsilon_1 \;<\; 1,$$

und

$$|\varepsilon_{ps}| \;\leq\; \varepsilon_2 + \underbrace{\frac{1}{x}\cdot\frac{1-x^2}{|\ln x|}}_{\leq 2.021}\cdot\underbrace{\frac{1-\varepsilon_2}{2-\varepsilon_1(1-x)}}_{\leq 0.50002}\cdot\varepsilon_1 \;\leq\; \varepsilon_2 + 1.011\cdot\varepsilon_1$$

bei $b = 10$, $\ell \geq 5$, $x_{min} >= 0.8$.

In beiden Fällen benötigt man also zwei Schutzziffern bei Division und Potenzreihenberechnung (und dort intern noch einmal zwei, also insgesamt vier). Wir werden aber sehen, dass durch die Argumentreduktion nicht noch mehr Schutzziffern anfallen. Insgesamt wird $|\varepsilon_{ps}| \leq 0.12 \cdot 10^{-p}$.

4.17.8 Fehler bei der Argumentreduktion

Es sei ε_2 der relative Fehler, der bei der Berechnung der Konstanten $\ln 2$ gemacht wird, ε_5 der für $\ln 5$. ε_{ps} ist der Fehler der Potenzreihenberechnung aus Abschnitt 4.17.7. Dann ergibt sich der verrundete Logarithmus zu

$$\begin{aligned}\widetilde{\ln} x &= (c_2 + c_{10})\cdot\widetilde{\ln} 2 \;+\; (c_5 + c_{10})\cdot\widetilde{\ln} 5 \;+\; \widetilde{\ln} x' \\ &= (c_2 + c_{10})\cdot(1+\varepsilon_2)\cdot\ln 2 + (c_5 + c_{10})\cdot(1+\varepsilon_5)\cdot\ln 5 + (1+\varepsilon_{ps})\cdot\ln x'.\end{aligned}$$

Bei der vorliegenden Implementation können die Additionen problemlos exakt durchgeführt werden (bevor insgesamt auf p Stellen endgerundet wird), da $\ln 2, \ln 5 \in [0.69, 1.61]$, $\ln x' \in [-0.25, 0.23]$ und $|c_{10}| < 2.2 \cdot 10^9$.

Damit erhalten wir als Forderung für den relativen Gesamtfehler

$$\begin{aligned}|\varepsilon_{ges}| &= \left|\frac{\widetilde{\ln} x - \ln x}{\ln x}\right| = \left|\frac{(c_2 + c_{10})\varepsilon_2 \ln 2 + (c_5 + c_{10})\varepsilon_5 \ln 5 + \varepsilon_{ps}\ln x'}{\ln f + c_{10}\ln 10 + \ln x'}\right| \\ &\leq \frac{|c_2 + c_{10}||\varepsilon_2|\ln 2 + |c_5 + c_{10}||\varepsilon_5|\ln 5 + |\varepsilon_{ps}||\ln x'|}{|\ln f + c_{10}\ln 10 + \ln x'|} \quad \overset{!}{\leq} \quad 10^{-p}.\end{aligned}$$

Mit den Schutzziffern aus den Abschnitten 4.17.4 bis 4.17.7 ist $|\varepsilon_{ps}| \le 0.12 \cdot 10^{-p}$. Sei m die noch zu bestimmende Stellenzahl der Konstanten $\ln 2$ und $\ln 5$. Damit wird die Forderung zu

$$(|c_2 + c_{10}| \ln 2 + |c_5 + c_{10}| \ln 5) \cdot 10^{-m+1} \le (\underbrace{|\ln f + c_{10} \ln 10 + \ln x'| - 0.12|\ln x'|}_{R}) \cdot 10^{-p}.$$

Diese ist für genügend großes m immer zu erfüllen, sofern es ein $r \in \mathbb{R}$, $r > 0$ gibt, so dass für alle Parameter-Konstellationen gilt $R > r$ (wird weiter unten gezeigt). In den Einzelfällen ergibt sich als Forderung

$$Q \cdot 10^{-m+1} \overset{!}{\le} 10^{-p}, \qquad Q = \frac{|c_2| \ln 2 + |c_5| \ln 5 + |c_{10}| \ln 10}{|\ln f + c_{10} \ln 10 + \ln x'| - 0.12|\ln x'|},$$

woraus man eine jeweils ausreichende Stellenzahl als $m \ge p + 1 + \log_{10} Q$ erhält.

In den folgenden Betrachtungen wird sich ergeben, dass in jedem Fall $\log_{10} Q < 3$ gilt, d.h. es reichen stets $m = p+3$ Stellen aus. Durch eine feinere Fallunterscheidung als unten durchgeführt kann man die Zahl der Schutzziffern in einzelnen Fällen auf 2 oder 1 absenken. Die Implementation verwendet der Einfachheit halber *immer drei* Schutzziffern. (Die Konstanten brauchen ja nur relativ selten berechnet zu werden, und die zusätzlichen Stellen können bei späteren Aufrufen genutzt werden.)

Wir zeigen $R > 0.141$ und $Q \le 42.59$ über eine Fallunterscheidung über die möglichen c_{10} und die Fälle a) bis g) der Argumentreduktion (von Seite 157).

$c_{10} = 0$: Es entfällt Fall g), d.h. es ist $f \ge 1.25$. Wir haben

$$R = |\ln f + \ln x'| - 0.12|\ln x'| = \ln f + \ln x' - 0.12|\ln x'|,$$

d.h. R ist auf $[0.78, 1.25]$ monoton steigend in x'. Nachrechnen in den Fällen a) bis f) ergibt insgesamt $R > 0.223$, wobei der kleinste Wert gerade in Fall f) auftritt (dort ist $x \in]1, 1.1]$):

$$R \;\ge\; \ln 1.25 + 1.12 \ln 1 \;>\; 0.223.$$

Für Q gilt in den Fällen a)–e):

$$Q \;\le\; \frac{4 \ln 2 + 2 \ln 5}{|\ln f + \ln x'| - 0.12|\ln x'|} \;\le\; \frac{\ln 400}{\ln 1.5625 + 1.12 \ln 0.78} \;<\; 35.67,$$

und im Fall f) (mit $x' > 1$)

$$Q \;\le\; \frac{2 \ln 2 + \ln 5}{\ln 1.25} \;<\; 13.43.$$

Hier werden also immer drei Schutzziffern benötigt.

$c_{10} \geq 1$: Nun ist in der Definition von R das Argument des ersten Betrags immer positiv, d.h.

$$R = \ln f + c_{10} \ln 10 + \ln x' - 0.12|\ln x'|.$$

Für $x' < 1$ bedeutet das

$$R \;=\; \ln f + c_{10} \ln 10 + 1.12 \ln x' \;\geq\; \ln 10 + 1.12 \cdot \ln 0.78 \;>\; 2.024$$

und für $x' \geq 1$

$$R \;=\; \ln f + c_{10} \ln 10 + 0.88 \ln x' \;\geq\; \ln 10 \;>\; 2.302\,.$$

Für Q gilt

$$\begin{aligned} Q \;&\leq\; \frac{4 \ln 2 + 2 \ln 5 + c_{10} \ln 10}{|\ln f + c_{10} \ln 10 + \ln x'| - 0.12 |\ln x'|} \;=\; \frac{\ln 400 + c_{10} \ln 10}{\ln f + c_{10} \ln 10 + \ln x' - 0.12 |\ln x'|} \\ &\leq\; \frac{c_{10} \ln 10 + \ln 400}{c_{10} \ln 10 + 1.12 \ln 0.78} \;\leq\; \frac{\ln 10 + \ln 400}{\ln 10 + 1.12 \ln 0.78} \;<\; 4.16\,, \end{aligned}$$

d.h. hier kämen wir immer mit nur zwei Schutzziffern aus.

$c_{10} \leq -1$: Hier ist das Argument im ersten Betrag von R immer negativ, d.h.

$$R = -\ln f - c_{10} \ln 10 - \ln x' - 0.12|\ln x'|.$$

Damit gilt für $x' < 1$

$$R \;=\; -\ln f - c_{10} \ln 10 - 0.88 |\ln x'| \;\geq\; -\ln 8 + \ln 10 - 0.88 \ln 1 \;>\; 0.223$$

bzw. für $x' \geq 1$ (das ist nur für $f \leq 5$ möglich)

$$R \;=\; -\ln f - c_{10} \ln 10 - 1.12 \ln x' \;\geq\; -\ln 5 + \ln 10 - 1.12 \ln 1.25 \;>\; 0.443.$$

Sei zunächst $c_{10} = -1$. Dann gilt in den Fällen b)–g) (wegen $f \leq 5$):

$$Q \;\leq\; \frac{4 \ln 2 + 2 \ln 5 + \ln 10}{|\ln f - \ln 10 + \ln x' - |0.12| \ln x'|} \;\leq\; \frac{\ln 4000}{\ln 2 - 1.12 \ln 1.25} \;<\; 18.72,$$

und im Fall a) (mit $f = 8$ und $x' \in [0.78125, 0.975[$):

$$Q \;\leq\; \frac{3 \ln 2 + \ln 10}{|\ln 8 - \ln 10 + \ln x'| - 0.12 |\ln x'|} \;\leq\; \frac{\ln 80}{\ln 1.25 - 0.88 \ln 0.78125} \;<\; 9.96$$

(hier würden sehr knapp zwei Schutzziffern ausreichen).

Sei schließlich $c_{10} \leq -2$. Dann haben wir

$$\begin{aligned} Q \;&\leq\; \frac{\ln 400 + |c_{10}| \ln 10}{|\ln f + c_{10} \ln 10 + \ln x'| - 0.12 |\ln x'|} \;=\; \frac{\ln 400 + |c_{10}| \ln 10}{-\ln f + |c_{10}| \ln 10 - \ln x' - 0.12 |\ln x'|} \\ &=\; 1 + \frac{\ln 3200 + 1.12 \ln 1.25}{|c_{10}| \ln 10 - \ln 8 - 1.12 \ln 1.25} \;\leq\; 1 + \frac{\ln 3200 + 1.12 \ln 1.25}{2 \ln 10 - \ln 8 - 1.12 \ln 1.25} \;<\; 4.66, \end{aligned}$$

d.h. auch hier würden zwei Schutzziffern ausreichen.

Es bleibt noch festzuhalten, dass wir (mit $Q \leq 35.67$ und drei Schutzziffern) vor der Endrundung nun erhalten:

$$|\varepsilon_{ges}| \leq 0.3567 \cdot 10^{-p}.$$

4.17.9 Berechnung mit dem Newton-Verfahren

Wie bereits erwähnt, findet in der endgültigen Implementation das Newton-Verfahren keine Verwendung. Da aber die Version zu [Steins] beim Logarithmus mit einem Intervall-Newton-Verfahren arbeitet, wurde auch für die vorliegende Arbeit zumindest eine entsprechende Testversion entwickelt (wiederum keine echte Intervall-Variante), die hier kurz skizziert werden soll.

Es wird in jedem Schritt die Auswertung der Exponentialfunktion benötigt, die bei hohen Stellenforderungen auch ein entsprechend langes Argument übergeben bekommt, weswegen das Verfahren dann unerträglich langsam wird. Außerdem hat es prinzipiell Probleme mit Auslöschung um das Argument $x = 1$ herum (siehe weiter unten), weswegen unsere Testversion in $[0.8, 1.2]$ die Potenzreihe (ohne Argumentreduktion) verwendet, so dass diese ohnehin implementiert werden müsste.

Die Implementation zu [Steins] bestimmt einen Startwert mit Hilfe einer aus den Interna des Computer-Algebrasystems REDUCE übernommenen Routine, die mit der Potenzreihe und Argumentreduktion arbeitet. Die Näherung hat bereits die gewünschte Stellenzahl; es kann allerdings nicht die Genauigkeit bis zur letzten Stelle garantiert werden, was dann durch das anschließende Intervall-Newton-Verfahren geschieht.

Wir greifen dagegen wiederum auf die schon im System vorhandene (auf den Testrechnern einigermaßen IEEE-kompatible) Arithmetik zurück. Weiter unten gehen wir davon aus, dass der erste relative Fehler $\leq 10^{-15}$ ist, was für das Funktionieren des Verfahrens nicht essentiell ist, unsere Betrachtungen aber deutlich abkürzt. Außerdem ist in den Grenzen der implementierten Arithmetik (BCD mit 32-Bit-Exponenten) $\ln x < 5 \cdot 10^9$, was wir aus denselben Gründen unten voraussetzen.

4.17.9.1 Das exakte Verfahren

Das Argument heiße wieder x, der exakte gesuchte Werte $y := \ln x$. Das Newton-Verfahren zur Nullstellenbestimmung wird angewandt auf die Funktion $f(y) = e^y - x$. Als *exakte* Iterationsvorschrift (insbesondere Auslöschung nicht beachtend) erhalten wir

$$y_{n+1} \;=\; y_n - \frac{e^{y_n} - x}{e^{y_n}} \;=\; y_n + \frac{x}{e^{y_n}} - 1\,.$$

Wenn der Startwert y_0 nur größer oder gleich y gewählt wird, sind induktiv alle Folgenglieder ebenfalls größer oder gleich y (wir verwenden die Potenzreihe von exp):

$$y_{n+1} - y \;=\; y_n - y - \frac{e^{y_n} - e^y}{e^{y_n}} \;=\; e^{y-y_n} - 1 - (y - y_n) \;\geq\; 0.$$

Außerdem ist die (exakte) Reihe monoton fallend, da $e^{y_n} - x = e^{y_n} - e^y \geq 0$. Sinnvollerweise sollte also mit gerichteten Rundungen eine Folge garantierter Oberschranken $\tilde{y}_n$ berechnet werden.

Der relative Gesamtfehler des exakten Verfahrens heiße ε_n. Dann ergibt sich (für $x > 1$):

$$y_{n+1} = y_n - \frac{e^{y_n} - x}{e^{y_n}} = y\left(1 + \varepsilon_n + \frac{e^{-y\varepsilon_n} - 1}{y}\right),$$

also

$$\varepsilon_{n+1} \;=\; \varepsilon_n + \frac{\varepsilon^{-y\varepsilon_n} - 1}{y} \;=\; \varepsilon_n^2 \left(\frac{y}{2!} - \frac{y^2}{3!}\,\varepsilon_n + \frac{y^3}{4!}\,\varepsilon_n^2 - + \ldots \right).$$

Der Klammerterm ist kleiner als $\frac{y}{2}$, falls $y < 4\varepsilon_n^{-1}$, d.h. bei unserem $\varepsilon_0 \leq 10^{-15}$, falls $\ln x < 4 \cdot 10^{15}$. In den Grenzen unserer Arithmetik ist das exakte Verfahren also immer von Beginn an quadratisch konvergent.

4.17.9.2 Das inexakte Verfahren

Bei der praktischen Implementation bietet die Iterationsvorschrift das Problem, dass sich der Quotient $\frac{x}{\exp(y_n)}$ schnell von unten 1 annähert und Auslöschung mit dem letzten Term -1 eintritt. Die Änderung $y_n - y_{n+1}$ hat also eine wesentlich schlechtere relative Genauigkeit, als die Stellenzahl von exp und der Division vermuten lassen würden.

Falls nun y betragsmäßig klein ist (für x um 1), reicht diese Genauigkeit sehr schnell nicht mehr aus, die Näherung zu verbessern, und das Verfahren stagniert, lange bevor die gewünschte Genauigkeit erreicht wird. Die Implementation zu [Steins] erhöht in diesem Fall (allerdings in einem Intervall-Newton-Verfahren) schrittweise die Stellenzahl, die letztlich ein Vielfaches der gewünschten betragen kann (Tests ergaben typischerweise mehr als das Doppelte).

Falls aber y betragsmäßig groß genug ist, kann man das Verfahren erfolgversprechend anwenden. Addition und Subtraktion können sogar exakt angewandt werden, da die Stellenzahl der Änderung nicht unkontrolliert anwachsen kann.

Genauer gilt Folgendes. Die verrundeten Näherungen seien $\tilde{y}_n$, und (unter Missbrauch der Bezeichnungen) sei nun ε_n der relative Gesamtfehler inklusive Rundungen, $\tilde{y}_n = y(1+\varepsilon_n)$. (Die Vorzeichen der relativen Fehler sind hier für den Fall $y > 0$ gewählt, die Beziehungen gelten aber entsprechend für $y < 0$.) Dann haben wir:

$$\begin{aligned}
\tilde{y}_{n+1} &= \tilde{y}_n + \frac{x}{\exp(y + y\varepsilon_n)} \cdot \frac{1+\varepsilon_/}{1-\varepsilon_e} - 1 = y(1+\varepsilon_n) + \frac{1}{\exp(y\varepsilon_n)} \cdot \frac{1+\varepsilon_/}{1-\varepsilon_e} - 1 \\
&= y(1+\varepsilon_n) + \frac{1+\varepsilon_/}{1-\varepsilon_e} \left(1 - y\varepsilon_n + \frac{y^2\varepsilon_n^2}{2} - + \ldots \right) - 1. \\
&= y(1+\varepsilon_n) + \frac{1}{1-\varepsilon_e} \left(\varepsilon_/ + \varepsilon_e - (1+\varepsilon_/) \left(y\varepsilon_n - \frac{y^2\varepsilon_n^2}{2} + \ldots \right) \right) \\
&= y \left[1 + \varepsilon_n + \frac{\varepsilon_/ + \varepsilon_e}{y(1-\varepsilon_e)} - \frac{1+\varepsilon_/}{1-\varepsilon_e} \left(\varepsilon_n - \frac{y}{2}\varepsilon_n^2 + \frac{y^2}{3!}\varepsilon_n^3 - + \ldots \right) \right],
\end{aligned}$$

also

$$\varepsilon_{n+1} \;\leq\; \frac{\varepsilon_/ + \varepsilon_e}{y(1-\varepsilon_e)} + \frac{1+\varepsilon_/}{1-\varepsilon_e} \cdot \frac{y}{2} \cdot \varepsilon_n^2.$$

Am y im Nenner des ersten Terms erkennt man die Probleme in der Nähe von $x = 1$.

Wenn z.B. bei uns $|x-1| \geq 0.2$, so ist $y > 0.182$, und der Faktor $(y(1-\varepsilon_e))^{-1}$ im ersten Term wird immerhin ≤ 5.485. Wenn wir also mit dem Gesamtfehler überhaupt unter 10^{-p}

gelangen wollen, muss für die verwendete Stellenzahl ℓ gelten $5.484 \cdot (1+2) \cdot 10^{-\ell+1} < 10^{-p}$, also $\ell > p + 2.199$, d.h. wir müssen mit mindestens 3 Schutzziffern arbeiten.

Wir führen nun also das Newton-Verfahren (nicht intervallmäßig) mit einer um 3 erhöhten Stellenzahl durch, unter Verwendung der zweiten Darstellung der Iterationsvorschrift. Die Division wird dabei nach oben, die Exponentialfunktion nach unten gerundet (bzw. eingeschlossen und die Untergrenze verwendet). Das Verfahren bricht ab, wenn sich die Näherung nicht verbessert hat. $\tilde{y}_n$ sei die beste erhaltene Näherung. Aus $\varepsilon_{n+1} \geq \varepsilon_n$ folgt mit der obigen Beziehung

$$\varepsilon_n \leq \frac{\varepsilon_/ + \varepsilon_e}{y(1-\varepsilon_e)} \;+\; \frac{1+\varepsilon_/}{1-\varepsilon_e} \cdot \frac{y}{2} \cdot 10^{-15} \cdot \varepsilon_n,$$

bzw. umgeformt

$$\begin{aligned} \varepsilon_n &\leq \frac{2(\varepsilon_/ + \varepsilon_e)}{y(2(1-\varepsilon_e) - (1+\varepsilon_/) \cdot 10^{-15} \cdot y)} \\ &\leq \frac{2(1+2)}{y(2(1-2\cdot 10^{-5}) - (1+10^{-5}) \cdot 10^{-15} \cdot y)}\, 10^{-p-2} \end{aligned}$$

Der Term ist in unserem Bereich monoton fallend in y. Mit $y \geq 0.182$ ergibt sich:

$$\varepsilon_n \leq 0.16484 \cdot 10^{-p}.$$

Danach erhalten wir durch einen abschließenden leicht veränderten Newton-Schritt die Untergrenze einer garantierten Einschließung $[\underline{y}, \tilde{y}_n]$ von y:

$$\underline{y} \;:=\; \tilde{y}_n - \frac{e^{\tilde{y}_n} - x}{x} \;=\; \tilde{y}_n - \frac{e^{\tilde{y}_{n-1}}}{x} + 1\,.$$

Die Division wird nach oben gerundet. Die Exponentialfunktion braucht nicht neu ausgewertet zu werden – es wird die Obergrenze der Intervall-Auswertung aus dem letzten Newton-Schritt verwendet.

$\underline{y}$ ist eine Unterschranke, da

$$y - \underline{y} \;=\; y - \tilde{y}_n + \frac{e^{\tilde{y}_n} - e^y}{e^y} \;=\; e^{\tilde{y}_n - y} - 1 - (\tilde{y}_n - y) \;\geq\; 0.$$

Für die Breite des entstehenden (ungerundeten) Intervalls gilt nun

$$\begin{aligned} \tilde{y}_n - \underline{y} &= \frac{\exp(\tilde{y}_n)(1+\varepsilon_{e'})}{x}(1-\varepsilon_{/'}) = (\exp(y\varepsilon_n)(1+\varepsilon_{e'}) - 1)(1-\varepsilon_{/'}) \\ &= \left[\left(1 + y\varepsilon_n + \frac{y^2\varepsilon_n^2}{2!} + \frac{y^3\varepsilon_n^3}{3!} + \ldots\right)(1+\varepsilon_{e'}) - 1\right](1-\varepsilon_{/'}) \\ &= (1-\varepsilon_{/'})\varepsilon_e + (1+\varepsilon_{e'})(1-\varepsilon_{/'}) \cdot y\varepsilon_n \cdot \left(1 + \frac{y\varepsilon_n}{2!} + \frac{y^2\varepsilon_n^2}{3!} + \ldots\right) \\ &\leq (1-\varepsilon_{/'})\varepsilon_{e'} + (1+\varepsilon_{e'})(1-\varepsilon_{/'}) \cdot y\varepsilon_n \cdot \left(1 + \frac{y\varepsilon_n}{2} \cdot \frac{1}{1-y\varepsilon_n}\right). \end{aligned}$$

Der Klammerterm ist (im Rahmen unserer Argumente) kleiner gleich 1.000003. Zusammen mit der Abschätzung für ε_n folgt

$$\begin{aligned}\tilde{y}_n - \underline{y} &\leq 2 \cdot 10^{-p-2} + 1.000014 \cdot \frac{2(1+2)}{2(1-2\cdot 10^{-5}) - (1+10^{-5}) \cdot 10^{-15} \cdot y} \cdot 10^{-p-2}, \\ &\leq 0.051 \cdot 10^{-p}.\end{aligned}$$

Die Einschließung genügt also unseren Genauigkeitsanforderungen.

Im Hinblick auf mögliche große Stellenforderungen wurde dem normalen Verfahren eine Startphase vorgeschaltet, in der mit der Stellenzahl 16 begonnen und diese in jedem Schritt verdoppelt wird. Der Gedanke dabei ist, dass die Anfangsnäherung etwa 16 Stellen genau sein dürfte und sich bei quadratischer Konvergenz in jedem Schritt die Anzahl gültiger Stellen etwa verdoppelt. So wird vermieden, dass von Anfang an unnötig große Mantissen entstehen und insbesondere unnötig genaue Aufrufe der Exponentialfunktion stattfinden. Meist sind am Schluss nur noch ein oder zwei Durchläufe mit der vollen Stellenzahl notwendig.

4.17.10 Der dekadische Logarithmus

Zusätzlich ist der Logarithmus zur Basis 10 implementiert, der über $\log_{10}(x) = \frac{\ln(x)}{\ln(10)}$ berechnet wird. Wenn ℓ die Stellenzahl sowohl bei der Berechnung von $\ln x$, der Konstanten $\ln 10$ und bei der Division bezeichnet, ergibt sich als relativer Fehler

$$\begin{aligned}|\varepsilon_{\log_{10}}| &= \left|\frac{\widetilde{\ln}\, x \,\tilde{/}\, \widetilde{\ln}\, x - \ln x / \ln 10}{\log_{10} x}\right| = \left|\frac{\frac{(\ln x)(1+\varepsilon_{\ln})}{(\ln 10)(1+\varepsilon_{\ln 10})}(1+\varepsilon_{/}) - \frac{\ln x}{\ln 10}}{\log_{10} x}\right| = \left|\frac{1+\varepsilon_{\ln}}{1+\varepsilon_{\ln 10}}(1+\varepsilon_{/}) - 1\right| \\ &= \left|(1+\varepsilon_{\ln})(1-\varepsilon_{\ln 10}(1-\frac{\varepsilon_{\ln 10}}{1+\varepsilon_{\ln 10}}))(1+\varepsilon_{/}) - 1\right| \\ &\leq |\varepsilon_{\ln}| + |\varepsilon_{/}| \cdot (1+|\varepsilon_{\ln}|) + |\varepsilon_{\ln 10}| \cdot |1 + \varepsilon_{\ln} + \varepsilon_{/} + \varepsilon_{\ln}\varepsilon_{/}|(1 - \tfrac{\varepsilon_{\ln 10}}{1+\varepsilon_{\ln 10}}) \\ &\overset{b=10}{\leq} |\varepsilon_{\ln}| + 1.0002 \cdot |\varepsilon_{/}| + 1.00041 \cdot |\varepsilon_{\ln 10}| \\ &\leq 2 \cdot 10^{-\ell+1} + 1.0002 \cdot 10^{-\ell+1} + 1.00041 \cdot 10^{-\ell+1} = 4.00061 \cdot 10^{-\ell+1} \overset{!}{\leq} 10^{-p},\end{aligned}$$

woraus sich die Forderung $\ell \geq p + 1 + \log_{10}(4.00061) \approx p + 1.603$ ergibt, d.h. wir arbeiten jeweils mit zwei Schutzziffern.

Wenn die Identität $\log_{10}(x_m \cdot 10^{x_e}) = \log_{10}(x_m) + x_e$ verwendet wird, kann sich die notwendige Stellenzahl für die Berechnung von $\log_{10}(x_m)$ verringern. Man erhält in der Abschätzung folgende Änderung:

$$|\varepsilon_{\log 10}| = \left|\frac{\log_{10}(x_m)}{\log_{10}(x_m) + x_e}\right| \cdot \left|\frac{1+\varepsilon_{\ln}}{1+\varepsilon_{\ln 10}}(1+\varepsilon_{/}) - 1\right|,$$

und für $x_e \neq -1$ gilt für den Vorfaktor

$$\left|\frac{\log_{10}(x_m)}{\log_{10}(x_m) + x_e}\right| \leq \frac{1}{|1+x_e|} \leq 1.$$

Damit ergibt sich für die notwendige Stellenzahl ℓ:

$$\ell \;\geq\; p + 1 - \operatorname{order} \frac{|1 + x_e|}{4.00061}.$$

Falls das so berechnete ℓ negativ wird (bei großen Exponenten und kleinen Stellenforderungen p), kann als Einschließung $[x_e, \operatorname{succ}_p(x_e)]$ verwendet werden.

Für $x_e \neq -1$ wird immer Verwendung der obigen Identität gemacht, da sich die interne Argumentreduktion bei ln vereinfacht. Da aber bei der Stellenabschätzung eine Division notwendig wird, wird diese nur für Exponenten verwendet, die betragsmäßig größer als 100 sind.

4.17.11 Intervallversionen

Wiederum ist wegen der Monotonizität von $\ln(x)$ und $\log_{10}(x)$ zur intervallmäßigen Implementation nichts zu sagen.

4.18 Die Potenzfunktion

Die Potenzfunktion x^y, $x, y \in \mathbb{R}$, wird mit Hilfe der Exponentialfunktion und des natürlichen Logarithmus berechnet. Vorab werden (in der aufgeführten Reihenfolge) folgende Spezialfälle getestet:

a) Für $y = 0$ wird 1 berechnet (auch für $x = 0$, in Kompatibilität mit anderen Programmiersprachen, aber Vorsicht in der Intervallversion).

b) Für $x = 0$ ist ansonsten $x^y = 0$.

c) Für $x < 0$ und im Fall $y \in \mathbb{Z}$ ist $x^y = (-1)^{|y|} \cdot |x|^y$ (was an Fall d) weitergereicht wird). Falls $y \notin \mathbb{Z}$, wird eine Exception geworfen – auch wenn $\frac{1}{y} \in \mathbb{Z}_{\text{odd}}$ gelten sollte, da Letzteres bei unserer Fließkommaarithmetik nur auftreten kann, falls $y \in \mathbb{Q}$ eine Potenz von 5 als Nenner besitzt. Wenn man gezielt eine n-te Wurzel berechnen möchte, sollte man direkt die dafür vorgesehene Funktion `root`(x, n) benutzen.

d) Ansonsten ist $x > 0$, und es wird $x^y = e^{y \cdot \ln x}$ verwendet.

Bei ganzzahligem Exponenten kann auch immer das bereits implementierte *exakte* `pow` mit anschließender Rundung verwendet werden. Wegen der möglicherweise stark wachsenden Stellenzahl wird so in der vorliegenden Implementation aber nur für $|y| \leq 4$ verfahren. Außerdem wird für $y = \frac{1}{2}$ die Funktion zur Quadratwurzel verwendet.

4.18.1 Fehleranalyse

Mit den offensichtlichen Bezeichnungen sei die verrundete Potenzfunktion dargestellt als

$$\widetilde{\text{pow}}(x, y) = \widetilde{\exp}\,(\widetilde{\ln}\,(x) \mathbin{\tilde{\cdot}} y) = \exp(\ln(x)(1 + \varepsilon_{\ln}) \cdot y(1 + \varepsilon_*)) \cdot (1 + \varepsilon_{\exp}),$$

womit sich für den relativen Fehler Folgendes ergibt:

$$|\varepsilon_{\text{pow}}| = \left|\frac{\widetilde{\text{pow}}(x,y) - \text{pow}(x,y)}{\text{pow}(x,y)}\right| = \left|\frac{\exp(y\ln x \cdot (1+\varepsilon_{\ln})(1+\varepsilon_*))(1+\varepsilon_{\exp}) - \exp(y\ln x)}{\exp(y\ln x)}\right|,$$

bzw. mit $z := y\ln x$ und $1+\varepsilon_z := (1+\varepsilon_{\ln})(1+\varepsilon_*)$

$$\begin{aligned}
|\varepsilon_{\text{pow}}| &= \left|\frac{\exp(z(1+\varepsilon_z))(1+\varepsilon_{\exp}) - \exp z}{\exp z}\right| = \left|\frac{(\exp(z) + \exp(\xi)\varepsilon_z z)(1+\varepsilon_{\exp}) - \exp z}{\exp z}\right| \\
&= \left|\frac{\varepsilon_{\exp}\cdot\exp(z) + \exp(\xi)\varepsilon_z z(1+\varepsilon_{\exp})}{\exp z}\right| \le |\varepsilon_{\exp}| + \frac{\exp\xi}{\exp z}\,|\varepsilon_z z|(1+\varepsilon_{\exp}) \\
&\le |\varepsilon_{\exp}| + \exp(|\varepsilon_z z|)\,|\varepsilon_z z|\,(1+\varepsilon_{\exp}) \quad \overset{!}{\le} b^{-p}.
\end{aligned}$$

Hieraus ergibt sich sofort, dass zwei Schutzziffern für die Berechnung der Exponentialfunktion notwendig sind, und damit ist $|\varepsilon_{\exp}| \le 2\cdot b^{-p-1}$. Die verbleibende Forderung für das Argument ist dann

$$\exp(|\varepsilon_z z|)\,|\varepsilon_z z| \;\overset{!}{\le}\; b^{-p}\,\frac{1-2b^{-1}}{1+2b^{-4}},$$

bzw. für $b = 10$ etwas schärfer

$$\exp(|\varepsilon_z z|)\,|\varepsilon_z z| \;\overset{!}{\le}\; 0.7798\cdot 10^{-p}.$$

Leider ist der Term $|\varepsilon_z z|$ unbeschränkt, so dass man keine global gültige erforderliche Stellenzahl für die Berechnungen angeben kann. Daher werden zunächst erste Einschließungen $[z_1]$ von $y\ln x$ und $[w_1]$ von $\exp([z_1])$ berechnet, woraus sich – analog zum späteren Vorgehen beim Wiederberechnungsverfahren – hinreichende Bedingungen für die Stellenzahlen bei ln, der Multiplikation und exp ergeben.

Man berechnet nun ersten Einschließungen mit einigen Schutzziffern gegenüber der Gesamtstellenforderung – im typischen Argumentbereich sind 3 bis 4 genug. Bei großen Argumenten, z.B. wenn die Summe der Charakteristiken größer als 10 ist, ist fast immer eine Neuberechnung erforderlich. Dann brauchen die ersten Einschließungen nur Aussagen über die Größenordnungen der Terme zu liefern, wofür schon wenige (z.B. 8) dezimale Stellen ausreichen.

In jedem Fall erhält man für die neuen Einschließungen $[z_2]$, $[w_2]$ die Forderung

$$\frac{[w_2]_{\text{sup}}}{[w_2]_{\text{inf}}}\cdot\text{diam}[z_2] \;\overset{!}{\le}\; b^{-p}\,\frac{1-2b^{-1}}{1+2b^{-4}}.$$

Bei Berechnung von ln und der Multiplikation mit m Stellen wird $|\varepsilon_z| = |\varepsilon_{\ln} + \varepsilon_*(1+\varepsilon_{\ln})| \le b^{-m+1}\,(3+2b^{-4})$, woraus sich a priori insgesamt ergibt

$$b^{-m+1} \;\overset{!}{\le}\; b^{-p}\cdot\frac{(1-2b^{-1})\cdot[w_1]_{\text{inf}}}{(1+2b^{-4})\,(3+2b^{-4})\cdot[w_1]_{\text{sup}}\,|[z_1]|_{\text{sup}}},$$

und damit für $b = 10$ schärfer

$$m := p + 1 - \text{order}\;\frac{[w_1]_{\text{inf}}}{3.752\,[w_1]_{\text{sup}}\,[|z_1|]_{\text{sup}}}.$$

4.18.2 Intervallversion

Zunächst sei $x > 0$ vorausgesetzt. Es ist nicht notwendig, die Potenzfunktion in allen vier Eckpunkten des durch die beiden Argumente beschriebenen Rechtecks auszuwerten. Wie in [Krämer], ab Seite 47, ausführlich behandelt, ändert die Funktion in $x = 1$ ihr Monotonieverhalten bezüglich y und in $y = 0$ bezüglich x. Außer in dem Fall, dass beide kritischen Stellen in den Argument-Intervallen enthalten sind, kommt man mit zwei Auswertungen aus. Im kritischen Fall muss zwar der Ausdruck $y \ln x$ in den vier Eckpunkten ausgewertet werden, die Exponentialfunktion jedoch danach nur zweimal.

Wenn die Argumente $x = [\underline{x}, \overline{x}]$, $y = [\underline{y}, \overline{y}]$ sind, ergibt sich folgende Tabelle (analog zu [Krämer], Seite 48, und [Steins], Seite 74):

			$\mathrm{pow}(x, y)$
$x \leq 1$	$y \leq 0$		$[\underline{\mathrm{pow}}(\overline{x}, \overline{y}), \overline{\mathrm{pow}}(\underline{x}, \underline{y})]$
	$y \geq 0$		$[\underline{\mathrm{pow}}(\underline{x}, \overline{y}), \overline{\mathrm{pow}}(\overline{x}, \underline{y})]$
	$0 \in y^{\mathrm{o}}$		$[\underline{\mathrm{pow}}(\underline{x}, \overline{y}), \overline{\mathrm{pow}}(\underline{x}, \underline{y})]$
$x \geq 1$	$y \leq 0$		$[\underline{\mathrm{pow}}(\overline{x}, \underline{y}), \overline{\mathrm{pow}}(\underline{x}, \overline{y})]$
	$y \geq 0$		$[\underline{\mathrm{pow}}(\underline{x}, \underline{y}), \overline{\mathrm{pow}}(\overline{x}, \overline{y})]$
	$0 \in y^{\mathrm{o}}$		$[\underline{\mathrm{pow}}(\overline{x}, \underline{y}), \overline{\mathrm{pow}}(\overline{x}, \overline{y})]$
$1 \in x^{\mathrm{o}}$	$y \leq 0$		$[\underline{\mathrm{pow}}(\overline{x}, \underline{y}), \overline{\mathrm{pow}}(\underline{x}, \underline{y})]$
	$y \geq 0$		$[\underline{\mathrm{pow}}(\underline{x}, \overline{y}), \overline{\mathrm{pow}}(\overline{x}, \overline{y})]$
	$0 \in y^{\mathrm{o}}$		s.u.

Im letzten Fall gilt

$$\mathrm{pow}(x, y) = [\ \underline{\exp}(\ \min\{\ \overline{y}\,\underline{\ln}(\underline{x}), \underline{y}\,\overline{\ln}(\overline{x})\ \}\ )\ ,\ \overline{\exp}(\ \min\{\ \underline{y}\,\underline{\ln}(\underline{x}), \overline{y}\,\overline{\ln}(\overline{x})\ \}\ )\ ]$$

Wenn die Intervallversion von pow (so wie die zu [Steins]) allerdings so arbeitet, den Ausdruck $y \ln x$ und nach der Bestimmung von Minimum und Maximum die Exponentialfunktion mit einer gewissen Zahl von Schutzziffern auszuwerten, kann sie nicht garantieren, dass die Intervallgrenzen auf p Stellen exakt sind. Außerdem wird so nie eine Ganzzahligkeit einer Grenze des Exponentenintervalls ausgenutzt.

Daher wird in der vorliegenden Implementation zunächst vorab mit sehr geringer Stellenzahl (z.B. $p/4$) bestimmt, in welchen beiden Eckpunkten pow punktweise auszuwerten ist. Falls sich die Einschließungen der Terme $y \ln x$ dabei noch schneiden, muss der Test mit erhöhter Genauigkeit durchgeführt werden. Anschließend wird für die beiden ermittelten Eckpunkte die punktweise pow-Routine aufgerufen, die die gewünschte Genauigkeit auf jeden Fall garantiert, dafür aber intern den Term $y \ln x$ erneut auswerten muss.

Die Potenzfunktion ist zusätzlich auch für negative Werte und 0 im Intervall definiert, wenn y eine ganze Zahl $y = z$ (als Punktintervall) ist. Wenn x die 0 enthält, darf z zusätzlich nicht negativ sein. Das Ergebnis wird abhängig vom Vorzeichen von x und Parität von z

berechnet.

$$x^y = \begin{cases} [\ x_{\text{inf}}^z\ ,\ x_{\text{sup}}^z\ ] & \text{für } 0 \in x,\ z > 0 \text{ ungerade,} \\ & \quad \text{oder } x < 0, \text{ und } z < 0 \text{ gerade oder } z > 0 \text{ ungerade,} \\ [\ x_{\text{sup}}^z\ ,\ x_{\text{inf}}^z\ ] & \text{für } x < 0, \text{ und } z < 0 \text{ ungerade oder } z > 0 \text{ gerade,} \\ [\ 0\ ,\ |x|_{\text{sup}}^z\ ] & \text{für } 0 \in x,\ z > 0 \text{ gerade.} \end{cases}$$

Zusätzlich ist eine Version `BFInterval`×`BigFloat` implementiert, bei der sich die Fallunterscheidung entsprechend vereinfacht.

4.18.3 *n*-te Wurzel

Es wurden spezielle Versionen `root`(x, n) zur Berechnung der n-ten Wurzel (mit Argumenten aus `BigFloat` bzw. `BFInterval`) implementiert, über

$$\sqrt[n]{x} = \begin{cases} \exp(\ln(x)/n) & \text{für } x \geq 0, \\ -\exp(\ln(|x|)/n) & \text{für } x < 0 \text{ und } n \text{ ungerade.} \end{cases}$$

Die Abschätzungen aus dem letzten Abschnitt bleiben unverändert gültig, aber die Fallunterscheidungen vereinfachen sich erheblich. Außerdem muss gegenüber einem Aufruf pow$(x, \frac{1}{n})$ nicht zunächst eine Division und intern anschließend eine Multiplikation durchgeführt werden. Die Intervallversion vereinfacht sich dadurch drastisch, dass negative Werte in x nur für ungerade n erlaubt sind und daher die Wurzel immer monoton steigend ist.

4.19 Weitere Funktionen

1. Der komplexe Betrag

Da der komplexe Betrag $|x + iy| = \sqrt{x^2 + y^2}|$ auch in anderem Zusammenhang (z.B. in einer $\mathbb{R}^2$-Implementation) Verwendung finden könnte, wurde er als Funktion `abs` von zwei `BigFloat`- bzw. `BFInterval`-Argumenten implementiert.

Falls eines der Argumente 0 ist, kann der Betrag des anderen zurückgegeben werden. Ansonsten werden die Quadrate exakt berechnet, die Addition und die Wurzel nur gerundet. Für den relativen Fehler gilt:

$$\begin{aligned} |\varepsilon_{\text{abs}}| &= \left| \frac{\widetilde{\text{sqrt}}(x^2 \tilde{+} y^2) - \text{sqrt}(x^2 + y^2)}{\text{sqrt}(x^2 + y^2)} \right| \\ &= \left| \frac{\sqrt{(x^2 + y^2)(1 + \varepsilon_+)}(1 + \varepsilon_{\surd}) - \sqrt{x^2 + y^2}}{\sqrt{x^2 + y^2}} \right| = |\sqrt{1 + \varepsilon_+} \cdot (1 + \varepsilon_{\surd}) - 1| \\ &\leq |\varepsilon_{\surd}| + (1 + \varepsilon_{\surd}) \frac{|\varepsilon_+|}{2\sqrt{1 - |\varepsilon_+|}} \leq b^{-m+1} \left(1 + \frac{1 + b^{-4}}{2\sqrt{1 - b^{-4}}} \right) \overset{!}{\leq} b^{-p}, \end{aligned}$$

Letzteres, falls Addition und Quadratwurzel mit jeweils m Stellen berechnet werden. Für $b = 10$ ist die Forderung $1.5001 \cdot 10^{-m+1} \leq 10^{-p}$, bzw. $m \geq p + 1.1762$, also werden zwei Schutzziffern benötigt.

Wegen der vergleichsweise hohen Geschwindigkeit von Addition und Wurzel werden die Operationen echt intervallweise ausgewertet. Zusätzlich gibt es eine Version `abs_up`, die schneller nur die Obergrenze der Einschließung berechnet. Die Version mit Intervallargumenten benutzt, dass die Funktion das Minimum in $(|x|_{\inf}, |y|_{\inf})$ annimmt, das Maximum in $(|x|_{\sup}, |y|_{\sup})$.

2. Das komplexe Argument

Auch der Polarwinkel $\in]-\pi, +\pi]$ einer komplexen Zahl ist für andere Einsatzmöglichkeiten interessant, weswegen er als Funktion `arg` von zwei Argumenten des Typs `BigFloat` realisiert wurde.

Im Innern des ersten Quadranten gilt $\arg(x, y) = \arctan(\frac{y}{x})$, die anderen Quadranten können hierauf zurückgeführt werden.

Es ist geschickter, den Arcustangens nur für Argumente aufzurufen, die betragsmäßig kleiner gleich 1 sind, d.h. für $|y| \leq |x|$. Ansonsten würde intern in der Arcustangens-Routine eine Argumentreduktion mit Inversion und dem Summanden $\frac{\pi}{2}$ durchgeführt. Da an dieser Stelle ggf. weitere solche Summanden anfallen, führen wir die Argumentreduktion *hier* durch (das Argument des Arcustangens wird $\frac{x}{y}$) und fassen Summanden zusammen.

Insgesamt erhält man für die Berechnung der Punkt-Routine:

$$\arg(x,y) = \begin{cases} \text{(Fehler)} & \text{für} \quad x=0,\ y=0, \\ -\frac{\pi}{2} & \text{für} \quad x=0,\ y<0, \\ 0 & \text{für} \quad y=0,\ x>0, \\ \frac{\pi}{2} & \text{für} \quad x=0,\ y>0, \\ \pi & \text{für} \quad y=0,\ x<0, \\ \arctan(\frac{y}{x}) & \text{für} \quad x>0,\ |y|\leq x, \\ -\arctan(\frac{x}{y})+\frac{\pi}{2} & \text{für} \quad x>0,\ |y|>x,\ y>0, \\ \arctan(\frac{x}{y})-\frac{\pi}{2} & \text{für} \quad x>0,\ |y|>x,\ y<0, \\ \arctan(\frac{y}{x})+\pi & \text{für} \quad x<0,\ y\geq 0,\ y\leq |x|, \\ -\arctan(\frac{x}{y})+\frac{\pi}{2} & \text{für} \quad x<0,\ y\geq 0,\ y>|x|, \\ \arctan(\frac{y}{x})-\pi & \text{für} \quad x<0,\ y\leq 0,\ |y|\leq |x|, \\ -\arctan(\frac{x}{y})-\frac{\pi}{2} & \text{für} \quad x<0,\ y\leq 0,\ |y|>|x|. \end{cases}$$

Es sind in allen Fällen jeweils zwei Schutzziffern für die Division, die Arcustangens-Berechnung und den π-Term notwendig. Für den ersten Arcustangens-Term (beachte $|\frac{y}{x}| \leq 1$) gilt beispielsweise:

$$\begin{aligned} |\varepsilon_{\arg}| &= \left| \frac{\widetilde{\arctan}\,(\tilde{y/x}) - \arctan(\frac{y}{x})}{\arctan(\frac{y}{x})} \right| = \left| \frac{\arctan(\frac{y}{x}(1+\varepsilon_/))(1+\varepsilon_a) - \arctan(\frac{y}{x})}{\arctan(\frac{y}{x})} \right| \\ &= \left| \varepsilon_a + \frac{y}{x} \cdot \frac{1}{1+\xi^2} \cdot \frac{1}{\arctan(\frac{y}{x})} \cdot (1+\varepsilon_a) \cdot \varepsilon_/ \right| \\ &\leq |\varepsilon_a| + \frac{1+|\varepsilon_a|}{1+|\frac{y}{x}|(1-|\varepsilon_/|)} \cdot \frac{|\frac{y}{x}|}{|\arctan(\frac{y}{x})|} \cdot |\varepsilon_/| \\ &\leq |\varepsilon_a| + 1.274 \cdot |\varepsilon_/| \ \leq\ 3.274 \cdot 10^{-m+1} \qquad \overset{!}{\leq}\ 10^{-p}, \end{aligned}$$

mit $\xi \in \mathrm{conv}\{\frac{y}{x}, \tilde{y/x}\}$, bei $b = 10$ und m-stelliger Berechnung von Division und Arcustangens, woraus sich schärfer $m \geq p + 1.5151$, also die Notwendigkeit von zwei Schutzziffern ergibt. Bei Hinzunahme eines π-Summanden bei exakter Addition gilt

$$\begin{aligned}\left|\frac{\tilde{\pi} + \widetilde{\arctan}\,(\tilde{y/x}) - \pi - \arctan(y/x)}{\pi + \arctan(y/x)}\right| &\leq \left|\frac{\pi}{\pi + \arctan(\frac{y}{x})}\right| \cdot |\varepsilon_\pi| + \left|\frac{\arctan(\frac{y}{x})}{\pi + \arctan(\frac{y}{x})}\right| \cdot |\varepsilon_{\arg}| \\ &\leq \tfrac{4}{3}\,|\varepsilon_\pi| + \tfrac{1}{3}\,|\varepsilon_a| + 0.4247 \cdot |\varepsilon_/| \;\leq\; 2.4247 \cdot 10^{-m+1},\end{aligned}$$

woraus sich auch hier überall die Forderung nach zwei Schutzziffern ergibt.

Die Punktfunktion nimmt auf Rechtecken der Ebene Minimum und Maximum in zwei Eckpunkten an, die allein von den Quadranten der Eckpunkte bestimmt sind (vgl. dazu die Behandlung in [Krämer], Seite 95f). Daher kann die Intervallversion von arg jeweils unter Benutzung zweier Aufrufe der Punktfunktion implementiert werden.

Es gibt den kleinen Problemfall, dass das Rechteck die negative x-Achse in seinem Innern schneidet. Es werden dann (wie bei [Krämer]) für die Punkte im dritten Quadranten Werte die um 2π erhöhten Werte der dortigen Punktfunktion verwendet (in der komplexen Interpretation wird auf das nächste Blatt der Riemannschen Fläche gewechselt).

5 Das Wiederberechnungsverfahren

In diesem Kapitel stellen wir die Grundlagen und die Implementation des zentralen Algorithmus vor, der die beliebig genaue Auswertung arithmetischer Ausdrücke ermöglicht, d.h. mit einer hoch genauen Einschließung des Werts des Ausdrucks bei vorgegebener Stellenzahl. Das Verfahren entspricht im Wesentlichen dem, das in [Steins], Abschnitt 1.3, entwickelt wird. Neuerungen ergeben sich bei den Details des Wiederberechnungsschritts, sowie natürlich in der bereits beschriebenen unterliegenden Arithmetik.

5.1 Codelisten

Wir wollen unter einem (reellen) arithmetischen Ausdruck in m (reellen) Variablen $x_1, \ldots, x_m$ eine Abbildung $F : D \to \mathbb{R}$, $D \subseteq \mathbb{R}^m$, verstehen, so dass es zur Auswertung eine „Codeliste“ folgender Art gibt:

$$\begin{aligned} x_{m+1} &= f_1(x_1, \ldots, x_m), \\ x_{m+2} &= f_2(x_1, \ldots, x_{m+1}), \\ &\vdots \\ x_{m+n} &= f_n(x_1, \ldots, x_{m+n-1}), \end{aligned}$$

so dass $y := F(x_1, \ldots, x_m) = x_{m+n}$.

Dabei sollen die $f_i : D_i \to \mathbb{R}$ genau für die vier Grundrechenarten und die im letzten Kapitel behandelten Standardfunktionen stehen, d.h. es sollen in ihrer Definition jeweils nur *maximal zwei* der möglichen Variablen tatsächlich vorkommen. Außerdem soll jedes der $x_{m+1}, \ldots, x_{m+n-1}$ *mindestens einmal* als Variable eines der f_i erscheinen, d.h. keines soll überflüssig sein. Die *mehrfache* Verwendung von Teilausdrücken ist damit erlaubt und kann im Programm durch mehrfaches Einsetzen eines Ausdrucks-Objekts in ein anderes erzeugt werden (der Compiler erkennt ein Mehrfach-Vorkommen so natürlich nicht automatisch). In der Theorie brauchen aber Mehrfachvorkommen nicht gesondert behandelt zu werden (vergleiche Bemerkung iii) auf Seite 177).

Unter den $x_1, \ldots, x_m$ sollen auch bestimmte Konstante vorkommen können, in unserer Implementation e und π, die im Verlauf der tatsächlichen Auswertung je nach Bedarf automatisch genau genug berechnet werden.

Codelisten dieser Art wurden z.B. bereits in [Rall] bei der Behandlung von Methoden der automatischen Differentiation eingeführt. Offensichtlich ist die Codeliste nicht eindeutig durch die reine Funktion F gegeben. Im einfachsten Fall sind beispielsweise unabhängige Berechnungen gegeneinander verschoben. Außerdem können selbstverständlich beliebige einfache oder komplexe mathematische Zusammenhänge äquivalente Darstellungen derselben Funktion F liefern.

Die Codeliste kann als gerichteter Graph dargestellt werden, wobei sich genau dann ein (Binär-) Baum ergibt, wenn es keine Mehrfachvorkommen von Teilausdrücken gibt. Die Knoten bilden die x_i, $i = 1, \ldots, m+n$, mit x_{m+n} als Wurzel und den Variablen $x_1, \ldots, x_m$ als die Blätter. Der Knoten x_{m+j}, $j = 1, \ldots, n$, hat als Nachfolger die eine oder die beiden in f_j tatsächlich vorkommenden x_i. In der Schreibweise werden wir die Knoten mit den x_i

identifizieren. Bei Mehrfachvorkommen hat der Graph Zyklen, aber keine gerichteten. Für die Theorie ergibt sich kein Unterschied zum Binärbaum, so dass wir der Einfachheit halber im Folgenden üblicherweise von einem Baum sprechen werden.

Der Baum wird von der Implementation explizit als Datenstruktur gebildet. Ein Knoten wird dabei dargestellt durch ein Objekt der Klasse `RealNode` bzw. davon abgeleiteter Klassen (etwa `RN_add` für einen Knoten zur Addition, `RN_exp` zur Exponentialfunktion, siehe dazu genauer Abschnitt 5.8.2). Während des Aufbaus des Baums findet noch keine Auswertung statt, da zu diesem Zeitpunkt noch nicht die letztlich gewünschte Genauigkeit bekannt zu sein braucht und sich eine Vorberechnung mit einer geschätzten Zahl oder Mindestzahl von Stellen bei einer später dann viel höheren Stellenforderung als bei den Abschätzungen zeitlich insgesamt als ungünstig erweisen könnte.

5.2 Spezifikationen

Zu einem wie im letzten Abschnitt beschrieben gegebenen arithmetischen Ausdruck F mit exaktem Wert y und zu einer gegebenen Stellenzahl $p \geq 1$ soll der Algorithmus eine hoch genaue Einschließung $[y]$ im folgenden Sinn berechnen (siehe Seite 5, vgl. [Steins], Seite 23):

Spezifikation A (für das Resultat $[y]$ des gesamten Algorithmus):

1. $[y] \in IS(b, p)$,
2. falls $0 \notin [y]$, enthält $[y]$ maximal 3 Elemente von $S(b, p)$,
3. falls $0 \in [y]$, gelte dagegen $\mathrm{diam}([y]) \leq b^{-p}$.

Dabei ist zu beachten, dass nicht garantiert wird, dass der Algorithmus auch ein Ergebnis liefern wird – beispielsweise wenn es zu Überläufen oder Überschreitungen von Definitionsbereichen kommt, auch ggf. nur zwischenzeitlich (in diesen Fällen wird eine entsprechende Exception geworfen). Wir werden aber soweit wie möglich versuchen, dies zu verhindern. Wenn aber ein Ergebnis berechnet wird, entspricht es obiger Spezifikation.

Wie bereits eingangs erwähnt, wird eine einfache intervallmäßige Auswertung mit p-stelliger Arithmetik nur in den allerseltensten Fällen eine p-stellige hoch genaue Einschließung liefern. Da beliebige Argumente erlaubt sind, kann auch a priori kein $p' > p$ gefunden werden, das als Stellenzahl für eine einfache Auswertung mit hoch genauem Endergebnis ausreichen würde. Eine schrittweise Erhöhung der Stellenzahl und komplette Neuberechnung wäre denkbar, aber unnötig langsam. Die letztlich für den ganzen Ausdruck verwendete Stellenzahl wird möglicherweise durch einen einzelnen, numerisch problematischen Teilausdruck bestimmt. Außerdem werden keinerlei Ergebnisse aus dem jeweils vorangegangenen Durchgang verwendet.

Wir entwickeln den von uns verwendeten Algorithmus zunächst informell. Er führt zuerst eine normale intervallmäßige Auswertung mit q Stellen durch (z.B. $p = q$) und speichert alle Zwischenergebnisse $[x_j]$ mit. Danach wird von der Wurzel ausgehend rekursiv in den Binärbaum des Ausdrucks hinabgestiegen. In jedem Knoten wird getestet, ob die geforderte Genauigkeit bereits vorliegt. Falls nicht, wird mit Hilfe der Einschließungen aus dem ersten

Durchgang abgeschätzt, mit wie vielen Stellen Operanden möglicherweise neu berechnet werden müssen (mit der so ermittelten Stellenzahl wird dann rekursiv weiter abgestiegen), und mit wie vielen Stellen die Operation dieses Knotens neu berechnet werden muss.

Die Stellenabschätzungen sind so angelegt, dass nur ein einziger Abstieg, also jeweils maximal eine Neuberechnung notwendig ist. Die einzige Ausnahme hiervon ergibt sich, wenn in der Wurzel zunächst die 0 mit eingeschlossen wurde, das nach dem Abstieg aber nicht mehr der Fall ist. Durch die unterschiedlichen Forderungen in der Definition von Hochgenauigkeit für diese beiden Fälle braucht die erste Stellenschätzung dann nicht notwendigerweise ausreichend gewesen zu sein. Es wird erneut abgestiegen, wobei der Abstieg aber normalerweise nach einem Schritt oder wenigen Schritten wieder enden wird. Es wird *nicht* bei jedem Teilergebnis (in jedem Knoten) erneut abgestiegen, nur von der Wurzel aus.

Damit ergibt sich eine etwas unterschiedliche Spezifikation für das Ergebnis eines Wiederberechnungsschritts mit ℓ Stellen in den Knoten (die Spezifikation gilt auch in der Wurzel, nur wird dort ggf. *anschließend* die Berechnung wiederholt). (A steht für Algorithmus, R für *recompute*).

Spezifikation R (für das Wiederberechnungs-Ergebnis $[x_i]$):

1. Falls beim Aufruf $0 \notin [x_i]$, gilt nach der Berechnung $[x_i] \in IS(b, \ell)$, und $[x_i]$ enthält maximal 3 Elemente von $S(b, \ell)$,
2. falls beim Aufruf $0 \in [x_i]$, gilt nach der Berechnung $\mathrm{diam}([x_i]) \leq b^{-\ell}$.

Bemerkungen:

i) Die Spezifikation ist gegenüber [Steins] (Seite 24) leicht geändert, da die dortigen Forderungen nicht immer erfüllbar sind. Es soll dort in *beiden* Fällen nach der Berechnung $[x_i] \in IS(b, \ell)$ gelten. Wenn die Einschließung aus dem ersten Durchgang die 0 enthält, die aus dem zweiten jedoch nicht mehr, kann zwar intern tatsächlich eine Einschließung mit Durchmesser $\leq b^{-\ell}$ berechnet werden, deren Grenzen höhere Stellenzahlen besitzen. Wenn diese nicht zufällig exakt ist, hat sie nach einer Rundung in das ℓ-stellige Raster aber mindestens einen Durchmesser von $b^{-\ell+c}$, wobei c die Charakteristik von $|[x_i]|_{\sup}$ ist.

   Daher muss, wenn die 0 ausgeschlossen wurde, auf die Endrundung verzichtet werden. Wenn doch gerundet wird und bei $c > 0$ obige Situation eintritt, sind die Voraussetzungen für die nachfolgenden Wiederberechnungsschritte nicht erfüllt, für die dann teilweise völlig sinnlose Stellenzahlen berechnet werden. Dieser Effekt ist anscheinend bei der Steins'schen Arithmetik auch zu beobachten, die dann zwar keine falschen, aber eventuell extrem ungenaue Einschließungen liefert.

   Für den Fall, dass im zweiten Schritt 0 eingeschlossen *bleibt*, darf endgerundet werden; die bei dieser Rundung möglicherweise auftretenden Fehler müssen aber berücksichtigt werden, d.h. es reicht nicht, für das Ergebnis vor der Rundung $\mathrm{diam}([x_i]) \leq b^{-\ell}$ zu fordern wie in [Steins] (siehe dazu Seite 184).

ii) Man beachte hier die Design-Entscheidung, die Genauigkeitsansprüche auch in den unteren Knoten in Form von Stellenforderungen zu definieren und damit auch während

der Rekursion als solche weiterzugeben, nicht etwa in Form von nicht zu überschreitenden absoluten oder relativen Toleranzen. Dieser Mechanismus wird von der Zahldarstellung auf dem Rechner und die Durchführung der Grundrechenarten praktisch erzwungen.

Bei anderer Handhabung wären leichte Geschwindigkeitsgewinne höchstens bei der Ermittlung der Anzahl von Gliedern der Potenzreihe bei den Standardfunktionen zu erwarten, was aber praktisch nicht ins Gewicht fällt, da sich (in steigendem N) aufeinander folgende Restglieder typischerweise gleich um Größenordnungen unterscheiden. Bei unserer Handhabung ergeben sich allerdings in jedem Schritt der Rekursion leichte Überschätzungen, d.h. die Anforderungen können manchmal deutlich übererfüllt werden.

iii) Der Fall, dass ein Teilausdruck *mehrfach* in einem größeren Ausdruck vorkommt, braucht in der Theorie nicht besonders behandelt zu werden, wohl aber in der Implementation. Ein Teilausdruck, der bereits Hochgenauigkeit garantiert (nach einem Durchlauf von *recompute*), wird in der Implementation entsprechend *markiert*. Wenn er von einer anderen Stelle aus erneut angelaufen wird, braucht er bei einer *geringeren* Stellenforderung als der alten nicht neu berechnet zu werden; das genauere Ergebnis kann einfach auf die niedrigere Stellenzahl gerundet werden (außer im Fall i) oben). Bei einer *höheren* Forderung wird die alte Einschließung seines Werts als Basis für den zweiten Durchgang verwendet; für die Abschätzung der Stellenzahl ist die Herkunft der ersten Einschließung gleichgültig.

5.3 *compute* und *recompute*

Die beiden Berechnungsphasen werden durch zwei Familien von Funktionen dargestellt, die jeweils die Berechnung des Wertes in einem Knoten N des Baums und durch rekursiven Aufruf möglicherweise in anhängenden Teilbäumen vornehmen.

- Die erste Phase mit normaler Intervallarithmetik wird durch die Funktionen $N.compute(p)$ realisiert (p ist die gewünschte Stellenzahl).
- Die Wiederberechnungsphase wird durch die Funktionen $N.recompute(p)$ realisiert.

Die Punkt-Notation ist hier an die objektorientierte Implementation angelehnt. *compute* und *recompute* sind dort virtuelle Funktionen der abstrakten Knoten-Klasse `RealNode`, die also erst in den Unterklassen definiert werden, abhängig vom jeweiligen genauen Knoten-Typ.

Die Funktionen speichern die jeweils berechnete Einschließung innerhalb des Knotens und liefern sie außerdem als Rückgabewert zurück.

5.3.1 Erstberechnungsphase

Die Funktionen *compute* brauchen nicht ausführlich beschrieben zu werden. Sie liefern, abhängig vom Typ der arithmetischen Operation, den jeweiligen Wert unter Verwendung normaler Intervallarithmetik zurück (und speichern ihn im Knoten).

Man kann diesen ersten Durchgang statt mit p Stellen mit einer leicht erhöhten Stellenzahl durchführen. Damit erhöht sich die Wahrscheinlichkeit, dass einige Zwischenergebnisse bereits jetzt genau genug werden und im nächsten Schritt von hier aus nicht weiter abgestiegen zu werden braucht. Andererseits verlangsamt sich dieser Durchgang dadurch eventuell deutlich (besonders bei Standardfunktionen), ohne dass ein Vorteil garantiert wäre, und ohne dass die alten Teilergebnisse zur Ermittlung der verbesserten Einschließungen verwendet werden könnten. Wiederum andererseits kann eine zu kleine erste Stellenzahl bei hohen Bäumen durch grobe erste Einschließungen zu deutlichen Überschätzungen der notwendigen endgültigen Stellenzahl führen. (Die Implementation erlaubt es dem Benutzer, eine andere Stellenzahl als p für den ersten Durchgang zu wählen.)

5.3.2 Bereichsüberschreitungen und eingeschlossene 0

a) Ein Problem in dieser ersten Phase ist der Fall, wenn eine Überschreitung eines Definitionsbereichs beispielsweise einer Standardfunktion auftritt. Die Implementation zu [Steins] bricht dann das Programm mit einem Fehler ab. Falls die Einschließung des problematischen Operanden aber nicht *völlig* außerhalb des jeweiligen Definitionsbereichs liegt, könnte eine Neuberechnung einer feineren Einschließung das Problem eventuell beseitigen.

   Unsere Implementation sieht daher eine Möglichkeit vor, bereits zu diesem Zeitpunkt eine *Wiederberechnung des Operanden* anzustoßen. Der Benutzer kann zur Laufzeit eine entsprechende Einstellung vornehmen. Es ist nicht sinnvoll, einen solchen Versuch zwingend durchzuführen, da sein Erfolg nicht garantiert werden und die Durchführung viel Zeit kosten kann. Vor allen Dingen könnte aber das Auslösen einer solchen Exception ein Hinweis an den *Benutzer* sein, dass sein Verfahren noch einen Schwachpunkt haben könnte. Möglicherweise kann er durch eine leichte Änderung der Berechnung von vornherein verhindern, dass Bereichsprobleme auftreten, während der Wiederberechnungsalgorithmus diese nachträglich aufwändig auflösen müsste.

   Es ist leider allgemein nicht möglich, eine Aussage darüber zu machen, ob ein solcher Wiederberechnungsversuch gelingen kann, zumindest nicht ohne eine („symbolische“) Analyse des Operanden-Ausdrucks. Auch kann allgemein a priori keine Aussage über eine ausreichende Stellenzahl gemacht werden.

   Beispielsweise soll beim Ausdruck $\tan(2\arctan(1))$ der Tangens in einer Polstelle ausgewertet werden, und jede noch so genaue Neuberechnung des Operanden $2\arctan(1)$ wird nichts daran ändern. Im Normalfall kann ein Verfeinerungsversuch aber erfolgversprechend sein. Wir werden das genaue Vorgehen im jeweiligen Fall in den passenden Abschnitten angeben.

   Falls dies eingestellt ist, werden einige Versuche gemacht, Definitionslücken auszuschließen, bevor ggf. durch das Werfen einer Exception nach außen aufgegeben wird; auch die Maximalzahl von Versuchen kann zur Laufzeit eingestellt werden. Wenn in der Konfigurationsdatei `XArithConfig.h` solche Versuche ganz verboten werden (`allow_retry_if_domain_error=false`), wird gar kein entsprechender Code generiert.

b) Eine ganz ähnliche Situation liegt immer vor, wenn der Operand eine Stelle einschließt, an der eine Standardfunktion einen Nulldurchgang hat. Das normale Vorgehen nach

der Spezifikation der Algorithmen liefert dann ein sehr kleines Intervall um die Null herum als Ergebnis, was nicht befriedigend ist, wenn man beispielsweise am Vorzeichen des Ergebnisses interessiert ist.

Für diesen Fall kann man mit ähnlichen Überlegungen wie oben versuchen, die Operandeneinschließung $[x]$ zu verfeinern. Natürlich ist dies wieder zur Laufzeit einstellbar und standardmäßig abgeschaltet (immerhin ist das normale Vorgehen ja von der Spezifikation abgedeckt, und die Wiederberechnung führt zu Laufzeitverlängerung). Man kann festlegen, dass nie wiederberechnet werden soll („`never`“), immer („`always`“) oder nur beim Endresultat („`result`“), also in der Wurzel des Baums. In der Konfigurationsdatei können solche Versuche ganz verboten werden (`allow_retry_if_zero_included=false`).

In den häufigen Fällen, dass gerade 0 diese Nullstelle ist (sin, tan, sinh, tanh, arcsin, arctan, arsinh, artanh), ergibt sich die beim Neuversuch verwendete Stellenzahl m aus der Überlegung, den Intervalldurchmesser auf unter die Hälfte von $d = \min\{-x_{\inf}, x_{\sup}\}$ zu drücken; man erhält $m := -\operatorname{order} \frac{d}{2}$. Ähnlich geht man bei ln und $\log_{10}$ vor, aufwändiger wird es bei transzendenten Nullstellen (sin, cos, tan, cot).

Es ist noch etwas zu bedenken, falls 0 so aus einem Intervall ausgeschlossen wird. Danach ist üblicherweise der relative Fehler der (nun gerade positiv oder negativ gewordenen) Einschließung ziemlich groß, was fast immer automatisch in *recompute* zu einer Neuberechnung führen wird.

An einer einzigen Stelle wird dies vom Prinzip her problematisch, da der große relative Fehler zu einer ganz extrem hohen Stellenschätzung für die Wiederberechnung führen kann, nämlich bei der Potenzierung x^y (besonders bei großen y). Dort wird, falls eine Heuristik ergibt, dass die Stellenzahl unverhältnismäßig groß geworden ist, explizit eine Neuberechnung des problematischen Operanden durchgeführt und die eigentliche Stellenzahl danach erneut geschätzt (siehe den entsprechenden Abschnitt).

5.3.3 Wiederberechnungsphase

Die Beschreibung von *recompute* ergibt sich wie folgt (wobei die Speicher- und Markierungsvorgänge nicht gesondert aufgeführt sind):

***N.recompute*(ℓ)**:

- Falls N ein Knoten zu einer Variablen ist, gib ihren auf ℓ Stellen gerundeten Wert zurück.
- Falls N ein Knoten zu einer Konstanten (π, e) ist, liefere eine ℓ-stellige Einschließung dieser Konstanten unter Verwendung der in Kapitel 3 beschriebenen Funktionen zurück.

Anderenfalls liegt ein unärer oder binärer Operator vor.

- Falls die Argumente (eines oder zwei) bereits genau genug vorliegen, bestimme die notwendige Stellenzahl der Neuauswertung für den vorliegenden Operator, berechne den neuen Wert und gib ihn auf ℓ Stellen gerundet zurück.

- Falls andererseits die Argumente noch nicht genau genug vorliegen, bestimme die notwendigen neuen Stellenzahlen für ihre Berechnung und rufe rekursiv *recompute* für die entsprechenden Knoten und die berechneten Stellenzahlen auf.
  Bestimme anschließend die notwendige Stellenzahl für die Neuberechnung des vorliegenden Operators (ggf. abhängig von den neuen Operanden), berechne den neuen Wert und gib ihn auf ℓ Stellen gerundet zurück.

Die hier benutzten Kriterien für genügende Genauigkeit der Operatoren bzw. für die notwendigen Stellenzahlen werden in den nächsten Abschnitten für die unterschiedlichen Operatoren entwickelt.

Vom Prinzip her ist die Vorgehensweise die folgende. Es ist durch die Genauigkeitsforderung an die aktuelle Operation ein maximal erlaubter Fehler vorgegeben, der auf den oder die Operanden und die eigentliche Operation verteilt wird. Wenn ein Operand die ihm zugeteilte Fehlertoleranz überschreitet, muss er neu berechnet werden. Wenn ein Operand neu berechnet wurde und die Toleranz danach unterschreitet, kann der verbleibende Maximalfehler neu verteilt werden (je nachdem, ob es sich vom Berechnungsaufwand bei der Verteilung her lohnt oder nicht).

Meistens wird so gearbeitet, dass eine Neuberechnung der Operanden (also ggf. eines großen Teilbaums) vermieden wird. Wenn in den jeweiligen Fehlerabschätzungen die zu den Operanden gehörigen Terme die Fehlertoleranz unterschreiten, wird kein Operand neu berechnet, was dann natürlich zu einer sehr genauen erneuten Auswertung der Operation führen kann.

Wenn die Operanden nicht genau genug vorliegen, wird meistens mit einer Gleichverteilung gearbeitet, also bei unären Operatoren mit einer Halbierung und Verteilung auf Operand und Operation, bei binären Operatoren einer Drittelung oder Aufteilung $\frac{1}{5} + \frac{2}{5} + \frac{2}{5}$, wenn die Operation (z.B. Grundrechenart) besonders schnell durchführbar ist.

Diese Verteilung muss immer heuristisch bleiben. Es wäre beispielsweise auch denkbar, sie proportional zum Beitrag zum Gesamtfehler aus dem ersten Durchgang vorzunehmen oder ähnlich. Letztlich führt jede Aufteilung zum Endziel, die gewünschte Genauigkeit im Knoten zu erreichen. Interessiert ist man schließlich hauptsächlich an einer möglichst großen Ausführungsgeschwindigkeit, und es kann leider den Operanden nicht angesehen werden, *wie schnell* sie neu berechnet werden können (also welchem schneller zu berechnenden Knoten man eher eine höhere Genauigkeitsforderung zumuten könnte).

5.3.4 Gesamtalgorithmus mit *compute* und *recompute*

Der Gesamtalgorithmus zur Berechnung des arithmetischen Ausdrucks mit hoch genauem p-stelligen Ergebnis (nach Spezifikation A) lautet nun, unter Verwendung von *compute* und *recompute* (vgl. [Steins], Seite 24):

Algorithmus A:

- Baue den internen Binärbaum auf, R sei dessen Wurzel.
- Erste Berechnungsphase, $[y] := R.compute(q)$.

- Falls $0 \in [y]$ und $\mathrm{diam}[y] > b^{-p}$,
  Wiederberechnung mit $[y] := R.recompute(p)$.

- Falls $0 \notin [y]$, und falls $[y]$ nicht p-stellig hoch genau,
  Wiederberechnung mit $[y] := R.recompute(p)$.

Typischerweise ist $q = p$. Die zwei angegebenen Schritte zur Wiederberechnung spiegeln nur den Fall wieder, dass zunächst 0 eingeschlossen war, nach der ersten Wiederberechnung aber nicht mehr.

5.4 Wiederberechnung bei unären Operatoren

Der einfachste unäre Operator, der in unseren Ausdrücken vorkommt, ist das unäre Plus, das aus technischen Gründen zwar als eigener Knoten im Binärbaum vorkommt, bei der Wiederberechnung aber einfach überlesen werden kann. Ebenso können natürlich Stellenforderungen beim unären Minus einfach an den Operanden weitergegeben werden. Die verbleibenden vorkommenden unären Operatoren sind die Standardfunktionen aus Kapitel 4.

Bemerkung: Wir schicken noch voraus, dass die verwendeten Intervallversionen der Standardfunktionen für den Fall, dass 0 im Ergebnis eingeschlossen ist, *nicht* dieselbe Spezifikation erfüllen wie die Zwischenergebnisse bei der Wiederberechnung, also bei m-stelliger Rechnung den absoluten Fehler betragsmäßig durch b^{-m} zu beschränken!

Die Gesamteinschließungen entstehen vielmehr durch Bestimmung eines $\underline{x}$, in dem die Funktion ihr Minimum auf dem Argumentintervall annimmt, und eines $\overline{x}$, in dem sie das Maximum annimmt, anschließende Einschließung $[f(\underline{x})]$ und $[f(\overline{x})]$ und Bildung der konvexen Hülle. (Bei den meisten Funktionen fallen wegen Monotonizität $\underline{x}$, $\overline{x}$ mit den beiden Intervallgrenzen zusammen.) Wenn die beiden Extremums-Einschließungen nicht 0 sind, garantieren sie Hochgenauigkeit der berechneten Grenzen über den *relativen Fehler an der jeweiligen Grenze*.

Die Art der Berechnung der Potenzreihen garantiert außerdem, dass 0 bei den Einschließungen an den (als exakt angenommenen) Grenzen nie eingeschlossen wird, es sei denn, der Wert ist dort exakt 0. Das liegt beispielsweise an der genauen Kenntnis der Nullstellen dieser Funktionen (und zusätzlich beispielsweise der entsprechend fein durchgeführten Argumentreduktion bei den trigonometrischen Funktionen).

Der absolute Fehler, der maximal bei der intervallmäßigen Auswertung gemacht wird, ergibt sich also, auch wenn 0 enthalten ist, aus dem maximalen relativen Fehler an den beiden Grenzen. Zu einem Intervalldurchmesser kann er also ungünstigstenfalls als $(|[y]|_{\inf} + |[y]|_{\sup}) \cdot 2 \cdot b^{-m+1}$, bzw. etwas überschätzt als $|[y]|_{\sup} \cdot 4 \cdot b^{-m+1}$ beitragen. Hierdurch ergeben sich Differenzen zu den in [Steins] verwendeten Beziehungen.

Um die Bezeichnungen hier abzukürzen, sei die Operation im betrachteten Knoten

$$y = f(x).$$

Die Einschließung des Operanden aus dem ersten Durchgang sei $[x_1]$ und das Resultat $[y_1]$, die Einschließungen, die aus dem nun durchzuführenden Wiederberechnungsschritt resultieren,

mögen $[x_2] \subseteq [x_1]$, bzw. $[f([x_2])] = [y_2] \subseteq [y_1]$ heißen. In Beziehungen, die für beide gelten können, verwenden wir $[x]$ und $[y]$. Die Bezeichnungen sind nicht mit denen für die Knoten des Binärbaums der vorangegangenen Abschnitte zu verwechseln.

5.4.1 Betrachtung mit absoluten Fehlern

Wir betrachten zunächst nur absolute Fehler, um den Sonderfall zu umgehen, dass 0 in einer der Einschließungen enthalten ist. Die Übersetzung in relative Fehler bzw. Stellenforderungen erfolgt ausführlich im Anschluss in Abschnitt 5.4.2.

Die Genauigkeitsforderung für diesen Schritt laute

$$\operatorname{diam}[y] \leq \Delta,$$

wobei sich Δ aus einer Forderung an den relativen Fehler ergeben wird.

Für den absoluten Fehler bei der verrundeten Berechnung von f mit einem gestörten Argument, $\tilde{y} = \tilde{f}(\tilde{x})$, mit $\Delta x = \tilde{x} - x$, gilt zunächst (vergleiche Abschnitt 4.1.3)

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

Der erste Summand bezieht sich auf den Fehler bei der Auswertung der Funktion (bei als exakt angenommenen Argument) und ist direkt mit einer Stellenforderung an die Funktionsauswertung verbunden, der zweite Summand auf das gestörte Argument und liefert eine Stellenforderung für eine Neuberechnung des entsprechenden Teilausdrucks.

Falls die Funktion auf dem Operandenintervall stetig differenzierbar ist, können wir den Mittelwertsatz auf den zweiten Term anwenden und erhalten

$$|\Delta y| \leq |\Delta f(\tilde{x})| + |f(\tilde{x}) - f(x)| \leq |\Delta f(\tilde{x})| + |f'(\xi)|\,|\Delta x| \overset{!}{\leq} \Delta \tag{5.1}$$

für ein $\xi \in \operatorname{conv}\{x, \tilde{x}\}$.

Wenn keine stetige Differenzierbarkeit vorliegt (etwa bei Quadratwurzel, Arcussinus, Arcuscosinus und Areacosinus am Rand ihres Definitionsbereichs), muss der Term $|f(\tilde{x}) - f(x)|$ anders als über den Mittelwertsatz (und in jedem der Fälle unterschiedlich) abgeschätzt werden, was in den Abschnitten zu diesen Funktionen behandelt wird. An dieser Stelle werden wir im Weiteren die stetige Differenzierbarkeit voraussetzen.

Auf unsere Situation bezogen gilt $x, \tilde{x}, \xi \in [x_i]$ und damit $\Delta x \leq \operatorname{diam}[x_i]$. Der Term $|f'(\xi)|$ kann mit Hilfe einer entsprechenden intervallmäßigen Funktionsauswertung $[f'([x_1])]$ durch $|[f'([x_1])]|_{\sup}$ nach oben abgeschätzt werden.

a) Falls nun für den zweiten Summanden in (5.1) gilt

$$|[f'([x_1])]|_{\sup} \cdot \operatorname{diam}[x_1] < \Delta,$$

so war das Operandenintervall $[x_1]$ bereits genau genug, und mit genügend hoher Stellenzahl bei der Funktionsauswertung kann insgesamt die Forderung bereits erfüllt werden. Es erfolgt also kein rekursiver Aufruf für den Teilbaum, der dem Operanden

entspricht. Die Differenz ist nun die erlaubte Toleranz bei der neuen Funktionsauswertung:

$$|\Delta f(\tilde{x})| \;\leq\; \Delta - |[f'([x_1])]|_{\sup} \cdot \operatorname{diam}[x_1].$$

Aus der Garantie der Standardfunktionen, hoch genaue Ergebnisse zu liefern, ergibt sich hieraus direkt eine Stellenforderung.

Wenn die Differenz auf der rechten Seite extrem klein ist, erzwingt dies hier eine hohe Stellenzahl. Da die Schranke Δ aber üblicherweise durch eine gewisse Überschätzung bei der Übersetzung von anderen Schranken in Stellenforderungen entstanden ist, tritt dies in der Praxis sehr selten auf und ist fast immer der Neuberechnung eines ganzen Teilbaums vorzuziehen. (Alternativ könnte man eine Neuberechnung des Operanden beispielsweise bereits ab der Schranke 0.95Δ o.ä. fordern.)

b) Anderenfalls (d.h. wenn der zweite Summand größer gleich Δ ist) muss der Operand durch rekursiven Abstieg neu berechnet werden, bevor die Funktion mit ihm als Argument neu ausgewertet werden kann. In diesem Fall verteilen wir die Fehlertoleranz Δ gleichmäßig auf beide Fehlerquellen. Für den Operanden soll also gelten

$$|[f'([x_2])]|_{\sup} \cdot \operatorname{diam}[x_2] \;\leq\; \frac{\Delta}{2}.$$

Da die Einschließung von f' für das neue x noch nicht bekannt ist, ergibt sich mit $[f'([x_1])]_{\sup} \geq [f'([x_2])]_{\sup}$ etwas schärfer

$$\operatorname{diam}[x_2] \;\leq\; \frac{\Delta}{2|[f'([x_1])]|_{\sup}},$$

woraus sich direkt eine Stellenforderung ableiten lässt.

An die Funktionsauswertung kann man nun eine der folgenden Forderungen stellen:

$$|\Delta f(\tilde{x})| \;\leq\; \frac{\Delta}{2} \qquad \text{oder} \qquad |\Delta f(\tilde{x})| \;\leq\; \Delta - |[f'([x_i])]|_{\sup} \cdot \operatorname{diam}[x_2].$$

Ersteres hat den Vorteil, schnell berechenbar zu sein, zweiteres kann zu niedrigeren Stellenforderungen führen. Dort kann bei der Ableitung die alte Einschließung $[f'([x_1])]$ verwendet werden oder eine neue $[f'([x_2])]$, falls eine solche schnell berechenbar ist.

5.4.2 Betrachtung mit relativen Fehlern

Es gilt nun noch, die Beziehungen des letzten Abschnitts exakt an die Spezifikation R des Wiederberechnungsschritts anzupassen.

i) Falls $0 \notin [y_1]$, d.h. auch $0 \notin [y_2]$, wird gefordert, dass das Ergebnis auf ℓ Stellen hoch genau ist. Nach Lemma 1 reicht es daher aus, zu gewährleisten, dass die intern berechnete Einschließung $[y_2]$ vor der Endrundung auf ℓ Stellen einen relativen Fehler von maximal $b^{-\ell}$ trägt, d.h. die Forderung ist

$$\frac{\operatorname{diam}[y_2]}{|[y_2]|_{\inf}} \leq b^{-\ell}.$$

Da a priori natürlich nur $[y_1]$ bekannt ist (mit $|[y_1]|_{\inf} \leq |[y_2]|_{\inf}$), muss schärfer

$$\frac{\operatorname{diam}[y_2]}{|[y_1]|_{\inf}} \leq b^{-\ell}$$

gefordert werden, d.h. das Δ aus dem letzten Abschnitt ergibt sich als

$$\Delta := b^{-\ell} \cdot |[y_1]|_{\inf}.$$

ii) Falls $0 \in [y_1]$ (also möglicherweise auch $0 \in [y_2]$), fordert die Spezifikation, dass das Ergebnis einen Maximaldurchmesser von $b^{-\ell}$ hat. Falls auch $0 \in [y_2]$, kann die Endrundung in das ℓ-stellige Raster durchgeführt werden (vergleiche die Bemerkung i) auf Seite 176). Dann reicht es *nicht* aus zu fordern, dass für das Intervall vor der Rundung $\operatorname{diam}[y_2] \leq b^{-\ell}$ gilt. Beispielsweise würde bei $\ell = 3$ das Intervall $[-4.999 \cdot 10^{-4}, +5.001 \cdot 10^{-4}]$ mit ausreichendem Durchmesser 10^{-3} gerundet auf das Intervall $[-5 \cdot 10^{-4}, +5.01 \cdot 10^{-4}]$ mit nicht mehr zulässigem Durchmesser $1.001 \cdot 10^{-3}$.

Unter Berücksichtigung des maximalen relativen Fehlers $1 + b^{-\ell+1}$ bei der Rundung ergibt sich die Forderung

$$\operatorname{diam}[y_2] \;\leq\; \frac{b^{-\ell}}{1 + b^{-\ell+1}},$$

bzw. bei unserer Implementation für $b = 10$ und $\ell \geq 3$ etwas schärfer

$$\operatorname{diam}[y_2] \;\leq\; 0.99 \cdot 10^{-\ell}.$$

In der Implementation zu [Steins] wird der Rundungseffekt nicht berücksichtigt. Die Differenzen sind aber so klein und die Toleranzen (bei der Fortpflanzung der Stellenforderungen) so groß, dass es schwer ist, ein Gegenbeispiel zu konstruieren. Insbesondere sind in diesem Fall in der Praxis die entstehenden Mantissen tendentiell eher *kürzer* als ℓ, so dass bei der Rundung gar kein Fehler entsteht.

Eine alternative Möglichkeit ist es, im Fall $0 \in [y_1]$ *nie* endzurunden (wir hatten unsere Spezifikation mit Absicht so formuliert, dass sie dabei erfüllt bleibt). So entsteht zwar an dieser Stelle möglicherweise ein längeres Zwischenergebnis; der Effekt pflanzt sich jedoch nicht nach oben fort. Das Rechnen und Vergleiche mit der Fehlertoleranz $10^{-\ell}$ sind aber deutlich schneller. Die augenblickliche Version der Implementation arbeitet standardmäßig so (es kann zur Compilationszeit durch eine Präprozessor-Definition zwischen beiden Möglichkeiten gewählt werden).

Nun übersetzen wir entsprechend die Fallunterscheidung des letzten Abschnitts.

i) Es sei zunächst $0 \notin [y_1]$, d.h. wir verwenden

$$\Delta := b^{-\ell} \cdot |[y_1]|_{\inf}.$$

a) Hinreichend für die genügende Genauigkeit des Operanden ist nun

$$|[f'([x_1])]|_{\sup} \cdot \operatorname{diam}[x_1] \;<\; \Delta \;=\; b^{-\ell} \cdot |[y_1]|_{\inf}.$$

Falls dies erfüllt ist ($[x_2] := [x_1]$), kann die Resttoleranz für die neue Funktionsauswertung ausgenutzt werden. Intervallmäßige m-stellige Berechnung garantiert, dass der entsprechende absolute Beitrag beschränkt ist durch

$$|\Delta y| \;\leq\; 4 \cdot b^{-m+1} \cdot |[y_1]|_{\sup},$$

woraus sich direkt die Forderung

$$b^{-m+1} \;\leq\; \frac{b^{-\ell} \cdot |[y_1]|_{\inf} - |[f'([x_1])]|_{\sup} \cdot \operatorname{diam}[x_1]}{4\,|[y_1]|_{\sup}} =: Q$$

ergibt, d.h. $m \geq 1 - \log_b Q$, was durch $m := 1 - \operatorname{order} Q$ garantiert wird. (Außerdem muss natürlich in der Implementation bei der Berechnung von Q geeignet gerichtet gerundet werden).

Es ist noch zu beachten, dass bei uns Quadrat und Quadratwurzel (analog später die Grundrechenarten) maximale Genauigkeit garantieren und sich dann in obiger Beziehung nur ein Faktor 2 im Nenner ergibt.

b) Falls der Operand die Bedingung unter a) nicht erfüllt, muss er neu berechnet werden, mit dem Ziel Folgendes zu erreichen:

$$\operatorname{diam}[x_2] \;\leq\; \frac{b^{-\ell}}{2} \frac{|[y_1]|_{\inf}}{|[f'([x_1])]|_{\sup}}.$$

Der rekursive Aufruf garantiert nun unterschiedliche Dinge, je nachdem, ob $[x_1]$ die Null enthält oder nicht.

α) Für $0 \in [x_1]$ garantiert der *recompute*-Aufruf mit Stellenforderung n, dass $\operatorname{diam}[x_2] \leq b^{-n}$. Zusammen mit obigem Ziel ergibt sich daraus für n die Bedingung

$$n \;\geq\; -\log_b \frac{b^{-\ell}|[y_1]|_{\inf}}{2|[f'([x_1])]|_{\sup}},$$

wobei schärfer wieder $\log_b$ durch order ersetzt werden kann, also wahlweise

$$n := -\operatorname{order} \frac{b^{-\ell}|[y_1]|_{\inf}}{2|[f'([x_1])]|_{\sup}}, \qquad n := \ell - \operatorname{order} \frac{|[y_1]|_{\inf}}{2|[f'([x_1])]|_{\sup}}.$$

Die zweite Berechnungsweise ist suggestiver, da die Anzahl *zusätzlicher* Stellen ablesbar ist. Meistens wird dennoch die erste verwendet, da $\Delta = b^{-\ell}|[y_1]|_{\inf}$ an mehreren Stellen der Fallunterscheidung vorkommt und daher ohnehin schon vorberechnet wurde. Gleiches gilt für die noch folgenden ähnlichen Beziehungen.

β) Ist dagegen $0 \notin [x_1]$, garantiert die Stellenforderung n bei *recompute*, dass $\operatorname{diam}[x_2] \leq 2 \cdot b^{-n+1} \cdot |[x_2]|_{\inf}$, d.h. wir fordern für n:

$$2 \cdot b^{-n+1} \cdot |[x_2]|_{\inf} \ \leq \ \frac{b^{-\ell} \cdot |[y_1]|_{\inf}}{2 \cdot |[f'([x_1])]|_{\sup}},$$

bzw.

$$n \ := \ 1 - \operatorname{order} \frac{b^{-\ell} \cdot |[y_1]|_{\inf}}{4 \cdot |[f'([x_1])]|_{\sup} \cdot |[x_1]|_{\sup}}.$$

Anschließend muss die Funktion mit dem neuen Operanden ausgewertet werden. Bei m-stelliger Berechnung und Gleichverteilung der Toleranz ergibt sich analog zu a):

$$m \ := \ 1 - \operatorname{order} \frac{b^{-\ell} \cdot |[y_1]|_{\inf}}{8 \cdot |[y_1]|_{\sup}},$$

bzw. bei Ausnutzung der neuen Einschließung des Operanden

$$m \ := \ 1 - \operatorname{order} \frac{b^{-\ell} \cdot |[y_1]|_{\inf} - |[f'([x_i])]|_{\sup} \cdot \operatorname{diam}[x_2]}{4 \cdot |[y_1]|_{\sup}}.$$

ii) Nun gelte $0 \in [y_1]$. Man beachte, dass dieser Fall bei einigen Funktionen nie eintritt (exp, cosh). Falls $[y_1] = 0$, braucht ohnehin nichts getan zu werden. Ansonsten ist hier die Toleranzgrenze, ja nachdem, ob mit Endrundung gearbeitet wird oder nicht,

$$\Delta := 0.99 \cdot b^{-\ell}, \quad \text{bzw.} \quad \Delta := b^{-\ell}.$$

a) Der Operand ist sicher genau genug, falls

$$|[f'([x_1])]|_{\sup} \cdot \operatorname{diam}[x_1] \ < \Delta.$$

Falls mit Endrundung, also dem Faktor 0.99, gearbeitet wird, braucht man nur „$\leq$“ zu fordern, da dieser Faktor bereits nach unten gerundet ist.

Ist dies erfüllt, erhält man als Forderung an die Funktionsauswertung mit m Stellen:

$$b^{-m+1} \ \leq \ \frac{\Delta - |[f'([x_1])]|_{\sup} \cdot \operatorname{diam}[x_1]}{4 \cdot |[y_1]|_{\sup}} =: Q,$$

also $m \geq 1 - \log_b Q$, bzw. $m := 1 - \operatorname{order} Q$.

b) Wenn keine genügende Genauigkeit des Operanden garantiert ist, muss er neu berechnet werden. Wir fordern analog zu i):

$$\operatorname{diam}[x_2] \ \leq \ \frac{\Delta}{2} \cdot \frac{1}{|[f'([x_1])]|_{\sup}}.$$

α) Falls $0 \in [x_1]$, ergibt sich für die notwendige Mindeststellenzahl n bei der Neuberechnung

$$n \ \geq \ -\log_b \frac{\Delta}{2 \cdot |[f'([x_1])]|_{\sup}},$$

was für $\Delta = b^{-\ell}$ (sonst müssen die Faktoren angepasst werden) auch bestimmt werden kann durch

$$n := \ell - \text{order}\ \frac{1}{2\,|[f'([x_1])]|_{\sup}} \quad \text{oder} \quad n := \ell + 1 + \text{order}(2\,|[f'([x_1])]|_{\sup}).$$

β) Falls $0 \notin [x_1]$, so garantiert der rekursive Aufruf mit n Stellen dagegen $\text{diam}[x_2] \leq |[x_2]|_{\inf} \cdot 2 \cdot b^{-n+1}$, und man erhält

$$n \ \geq\ 1 - \log_b \frac{\Delta}{4 \cdot |[f'([x_1])]|_{\sup} \cdot |[x_1]|_{\sup}},$$

bestimmbar beispielsweise als $n := \ell + 2 + \text{order}(4 \cdot |[f'([x_1])]|_{\sup} \cdot |[x_1]|_{\sup})$ (falls $\Delta = b^{-\ell}$).

Anschließend muss die Funktion mit dem neuen Operanden berechnet werden. Eine Gleichverteilung der Toleranz liefert (bei m-stelliger Rechnung) die Forderung

$$b^{-m+1} \ \leq\ \frac{\Delta}{8 \cdot |[y_1]|_{\sup}},$$

bestimmbar (bei $\Delta = b^{-\ell}$) auch über $m := \ell + 2 + \text{order}(8|[y_1]|_{\sup})$.

Die neue Operandeneinschließung verwendend, kann man natürlich auch fordern

$$b^{-m+1} \ \leq\ \frac{\Delta - |[f'([x_i])]|_{\sup} \cdot \text{diam}[x_2]}{4 \cdot |[y_1]|_{\sup}}\,.$$

Zu den Genauigkeiten bei der Berechnung der Stellenforderungen sei angemerkt, dass die Subtraktionen nicht exakt durchgeführt werden sollten, sondern beispielsweise passend gerichtet gerundet mit der für das Endergebnis geforderten Stellenzahl ℓ. (Bei großen Unterschieden in den Größenordnungen der vorkommenden Terme kann es bei exakter Rechnung wieder überlange Mantissen geben, wie in [Steins]). Die Divisionen brauchen dagegen nicht mit der Genauigkeit ℓ stattzufinden (wie in [Steins]). Da es immer nur auf die Größenordnungen der Terme ankommt, reicht sogar eine Division mit *einer* Stelle völlig aus. (Es ist von der resultierenden Stellenzahl aber meist günstiger, eine solche Division $\text{order}(z/n)$ zu benutzen als eine Differenz $\text{order}(z) - \text{order}(n) + 1$!)

Im Folgenden gehen wir kurz auf die Besonderheiten der implementierten (Standard-)Funktionen ein.

5.4.3 $\text{sqr}(x)$

Diese Funktion ist bei [Steins] gar nicht implementiert. Durch die einfache Ableitung gegenüber dem allgemeinen `pow` und der höheren Genauigkeit und Geschwindigkeit mit der Multiplikation schien es aber angebracht, sie bei uns aufzunehmen.

Die Quadrierung wird von der Arithmetik momentan intern immer exakt durchgeführt und anschließend gerundet, d.h. das Ergebnis einer Quadrierung ist immer maximal genau. Es ist immer $[y_1] \geq 0$, und wenn $0 \in [y_1]$, so ist es untere Grenze und $0 \in [x_1]$, was die entsprechende Fallunterscheidung vereinfacht.

Durch die Ableitung $f'(x) = 2x$ vereinfachen sich die Beziehungen zusätzlich. Beispielsweise muss der Operand für $0 \notin [x_1]$ bzw. $0 \in [x_1]$ mit folgenden Stellenzahlen berechnet werden:

$$m := 1 - \operatorname{order} \frac{\Delta}{4[y_1]_{\sup}}, \quad \text{bzw.} \quad m := -\operatorname{order} \frac{\Delta}{2|[x_1]|_{\sup}}.$$

5.4.4 $\exp(x)$

Es gilt stets $0 \notin [y_i]$, d.h. es ist nur der Fall i) der Fallunterscheidung zu behandeln.

Außerdem ist $f'(x) = f(x)$, d.h. für den Ableitungsterm $|[f'([x_1])]|_{\sup}$ kann denkbar einfach $[y_1]_{\sup}$ aus dem ersten Schritt verwendet werden.

5.4.5 $\sin(x)$, $\cos(x)$

Falls Wiederberechnungsversuche bei Einschließung von 0 eingestellt wurden (siehe Abschnitt 5.3.2), wird bei den transzendenten Nullstellen so verfahren wie bei den transzendenten Definitionslücken beim Tangens (siehe dort).

Es ist zeitlich günstiger, die Einschließung der jeweiligen Ableitung über $\sin^2 x + \cos^2 x = 1$ zu bestimmen, also $|\cos x| = \sqrt{1 - \sin^2 x}$, $|\sin x| = \sqrt{1 - \cos^2 x}$, da die Quadratwurzel wesentlich schneller implementiert ist als die trigonometrischen Funktionen.

Es sei noch angemerkt, dass die hergeleiteten Stellenforderungen auch in den Fällen korrekte Ergebnisse liefern, in denen im ersten Berechnungsschritt als Einschließung $[-1, +1]$ berechnet wurde, so dass es nicht nötig ist, dann das Argument zwangsweise feiner zu berechnen.

5.4.6 $\tan(x)$, $\cot(x)$

Diese beiden Funktionen haben unendlich viele Definitionslücken (Polstellen), nämlich $\frac{\pi}{2} + \pi\mathbb{Z}$ bzw. $\pi\mathbb{Z}$. Alle (außer 0) sind transzendent und können bei genügend hoher Genauigkeit bei der Berechnung des Operanden aus dem Operanden-Intervall ausgeschlossen werden – wenn beispielsweise nur rationale Arithmetik im Operanden betrieben wird. Wenn allerdings Funktionen vorkommen, die Vielfache von π beliebig genau berechnen können, muss jeder Wiederberechnungsversuch fehlschlagen. Entsprechend ungünstige Ausdrücke sind also

$$\tan(\pi/2), \quad \tan(2\arctan(1)), \quad \tan(\arcsin(1)), \quad \tan(\arccos(0)).$$

Diese Beispiele sind natürlich eher pathologisch, typischerweise ist ein Verfeinerungsversuch erfolgversprechend. Wenn dies eingestellt ist, werden in unserer Implementation Verfeinerungen berechnet, bis Polstellen ausgeschlossen wurden oder bis die Maximalzahl von Versuchen erreicht ist.

Das Problem, das bereits in *compute* auftritt, braucht nicht dort explizit abgetestet zu werden, vielmehr werfen die Intervallversionen von Tangens bzw. des Cotangens im Problemfall eine Exception, auf die *compute* reagieren muss.

Da jedes Mal ggf. ein ganzer Teilbaum neu berechnet werden muss, sollte man nicht einfach nur die Stellenzahl z.B. um 2 erhöhen und damit einen neuen Berechnungsversuch wagen. Bei

der Abschätzung ist es hilfreich, dass die getroffene Polstelle beliebig genau eingeschlossen werden kann.

Die Polstellen haben den Abstand π, so dass es Sinn macht, zunächst einmal sicherzustellen, dass das Operandenintervall einen Durchmesser kleiner als zum Beispiel $0.75 < \frac{\pi}{4}$ hat (was bei sehr großen Argumenten nicht notwendigerweise bereits erfüllt sein muss). Das jeweils betrachtete Argument heiße $[x]$. Falls $0 \notin [x]$, ergibt sich für die notwendige Stellenzahl m:

$$2 \cdot b^{-m+1} \cdot |[x]|_{\sup} \leq \frac{3}{4}, \qquad \text{bzw.} \qquad m := 1 - \operatorname{order} \frac{3}{8|[x]|_{\sup}}.$$

Wenn danach noch immer eine Polstelle eingeschlossen sein sollte, kann das leicht über die Durchführung der Argumentreduktion (für die trigonometrischen Funktionen) an den beiden Intervallgrenzen festgestellt werden. Sie liefert unter anderem die ganzzahligen Anteile i_1, i_2 bei Division durch $\pi/4$, und im Positiven ist genau dann noch eine Polstelle eingeschlossen, wenn $(i_1 + 2)/4 \neq (i_2 + 2)/4$ (Tangens) bzw. $i_1/4 \neq i_2/4$ (Cotangens). Diese Polstelle ist dann $i_2 \cdot \frac{\pi}{4}$.

Mit einer geeignet feinen Einschließung der Polstelle wird nun der Abstand der Unter- und der Obergrenze des Operandenintervalls von ihr näherungsweise bestimmt, d sei das Minimum der beiden. Dann soll die nächste Wiederberechnung den Intervalldurchmesser auf unter $\frac{d}{2}$ verkleinern, was durch

$$m := 1 - \operatorname{order} \frac{d}{4|[x]|_{\sup}}$$

Stellen garantiert wird.

So werden neue Versuche durchgeführt, bis der Tangens oder Cotangens gefahrlos aufgerufen werden kann oder bis die eingestellte Maximalzahl von Versuchen erreicht ist (danach wird eine Exception nach außen geworfen). Falls Wiederberechnungsversuche bei Einschließung von 0 eingestellt sind, wird völlig analog verfahren.

Die eigentliche Wiederberechnung *recompute* wird nur erreicht, wenn *compute* zuvor erfolgreich durchgeführt werden konnte, d.h. diese Funktion kann davon ausgehen, dass keine Polstelle eingeschlossen wird. Insbesondere verschwindet $\cos x$ (beim Tangens) bzw. $\sin x$ (beim Cotangens) auf dem Operandenintervall nicht.

Die Ableitungen der beiden Funktionen sind $1/\cos^2 x$ bzw. $-1/\sin^2 x$. Die jeweilige trigonometrische Funktion muss leider mit einer echten Intervallversion berechnet werden. Statt $|[f'([x_i])]|_{\sup}$ sollte aber offenbar besser $|[\frac{1}{f'}([x_i])]|_{\inf}$ berechnet werden. Die Beziehungen für die Stellenforderungen werden entsprechend angepasst, beispielsweise Vergleiche mit diesem Wert multipliziert oder Brüche auf den Hauptnenner gebracht, um nicht zwei langsame Divisionen zu benötigen.

Falls beim Tangens beispielsweise $0 \notin [y_1]$, ist das Argument bereits genau genug, falls

$$\operatorname{diam}[x_1] \;\leq\; b^{-\ell} \cdot |[y_1]|_{\inf} \cdot (\cos[x_1])^2_{\inf},$$

und als Stellenforderung für die Neuberechnung des Tangens ergibt sich

$$m := 1 - \operatorname{order} \frac{b^{-\ell} \cdot |[y_1]|_{\inf} \cdot (\cos[x_1])^2_{\inf} - \operatorname{diam}[x_1]}{4 \cdot |[y_1]|_{\sup} \cdot (\cos[x_1])^2_{\inf}}.$$

Beim Cotangens vereinfacht sich noch die Implementation, da 0 als Definitionslücke nie im Operandenintervall enthalten ist.

5.4.7 $\sinh(x)$

Bei dieser Funktion kann man einstellen, dass für $0 \in [x_1]$ Wiederberechnungsversuche gemacht werden.

Die Fallunterscheidung vereinfacht sich dadurch, dass $0 \in [y_1] \Leftrightarrow 0 \in [x_1]$. Die Ableitung ist $\cosh(x)$, und wegen $\cosh^2 x - \sinh^2 x = 1$ kann für den vorkommenden Ableitungsterm $[\cosh[x_1]]_{\sup}$ der Term $[\sqrt{1 + [y_1]^2_{\sup}}]_{\sup}$ verwendet werden, da die Quadratwurzel wiederum von der Geschwindigkeit her zu bevorzugen ist.

5.4.8 $\cosh(x)$

Wegen $\cosh(x) \geq 1$ entfällt hier der Teil $0 \in [y_1]$ der Fallunterscheidung. Für den Ableitungsterm $|[\sinh([x_1])]|_{\sup}$ kann günstigerweise $[\sqrt{[y_1]^2_{\sup} - 1}]_{\sup}$ verwendet werden.

5.4.9 $\tanh(x)$

Hier können für $0 \in [x_1]$ Wiederberechnungsversuche eingestellt werden. Die Ableitung von $f(x) = \tanh(x)$ ist $f'(x) = \frac{1}{\cosh^2 x}$, weswegen die Abschätzungen, in denen der Ableitungsterm vorkommt, durch Multiplikationen günstig umgeformt werden können. Bei der Berechnung von $\cosh^2$ sind allerdings Exceptions wegen eines Overflows abzufangen; der Ableitungsausdruck kann in diesem Fall z.B. durch $b^{-\ell}$ ersetzt werden. Die Berechnung von tanh ist dann trotzdem erfolgversprechend, weil dort große Argumente ja separat (als $\approx \pm 1$) berechnet werden.

5.4.10 $\coth(x)$

Die Funktion *compute* versucht ggf. durch Wiederberechnungen zu erreichen, dass die Polstelle 0 nicht in der Einschließung des Arguments vorkommt, d.h. *recompute* kann daher immer davon ausgehen, dass $0 \notin [x_1]$. Außerdem gilt immer $|f(x)| > 1$, also $0 \notin [y_1]$.

Wegen der Form der Ableitung, $f'(x) = \frac{1}{\sinh^2 x}$, können die entsprechenden Abschätzungen wieder umgeformt werden, um Divisionen zu umgehen. Bei Overflows bei der Berechnung von $\sinh^2$ wird wie bei tanh verfahren.

5.4.11 $\arcsin(x)$, $\arccos(x)$

Hier gibt es im ersten Berechnungsschritt Probleme, wenn die Operandeneinschließung Werte kleiner als -1 oder größer als $+1$ enthält. Es werden dann Wiederberechnungsversuche gemacht, die den Intervalldurchmesser unter die Hälfte des Abstands der Grenzen zu ± 1 drücken. Außerdem kann der Benutzer bei arcsin Wiederberechnungsversuche für den Fall einstellen, dass 0 eingeschlossen wird.

Im eigentlichen Wiederberechnungsschritt kann nun für den Operanden eine Untergrenze -1 bzw. eine Obergrenze $+1$ ankommen. An diesem Rand existiert die Ableitung nicht, so

dass der Term $|f(\tilde{x}) - f(x)|$ aus (5.1) anders abgeschätzt wird. In leichter Abwandlung zu [Steins], S. 22, gilt für arcsin bei $\tilde{x} = -1$:

$$\begin{aligned} |\arcsin(\tilde{x}) - \arcsin(x)| &= |-\tfrac{\pi}{2} - \arcsin(x)| = \tfrac{\pi}{2} + \arcsin(x) = \arccos(-x) \\ &= \arccos(\tilde{x} - x + 1) = \arccos(1 - |\Delta x|). \end{aligned}$$

Die Beziehung ist offensichtlich ebenso gültig, wenn man x und $\tilde{x}$ vertauscht, und aus Symmetriegründen bei arcsin auch am Rand $+1$.

In diesem Ausnahmefall sollte auf jeden Fall eine Wiederberechnung des Operanden durchgeführt werden. Wenn in (5.1) die Fehlertoleranz auf die beiden Terme gleich verteilt wird, ist die Forderung an den Operanden also

$$\arccos(1 - |\Delta x|) \stackrel{!}{\leq} \frac{\Delta}{2},$$

bzw. nach Anwendung des Cosinus (der auf $[0, \pi]$ monoton fällt)

$$1 - |\Delta x| \geq \cos\frac{\Delta}{2}, \qquad |\Delta x| \leq 1 - \cos\frac{\Delta}{2} = 2\sin^2\frac{\Delta}{4}.$$

Für kleine Argumente (und im Normalfall wird die Fehlertoleranz sehr klein sein) kann der Sinus sehr gut (und schnell) durch die ersten beiden Glieder seiner Potenzreihe approximiert werden, d.h. wir verwenden die Beziehung $|\sin u| \geq |u(1 - \frac{u^2}{6})|$, so dass

$$2\sin^2\frac{\Delta}{4} \geq 2\Big[\frac{\Delta}{4}\Big(1 - \frac{(\Delta/4)^2}{6}\Big)\Big]^2 = \frac{[\Delta(96 - \Delta^2)]^2}{73728} \stackrel{*}{\geq} \frac{(95.999999\,\Delta)^2}{73728} \geq 0.124999\,\Delta^2.$$

Dabei gilt die mit * bezeichnete Ungleichung für $\Delta \leq 10^{-3}$, was nach *compute* und $\ell \geq 3$ erfüllt ist.

Aus der Forderung $\mathrm{diam}[x_2] \leq 0.124999\,\Delta^2$ erhält man nun als Stellenforderung für die Operandenberechnung (falls $0 \notin [x_1]$):

$$b^{-n+1} \leq 0.06249 \cdot b^{-2\ell} \cdot |[y_1]|^2_{\inf}, \qquad n := 2\ell + 1 - \mathrm{order}(0.06249\,|[y_1]|^2_{\inf}).$$

Im Normalfall, wenn also nicht -1 oder $+1$ im Operanden vorkommen, kann normal mit der Ableitung gearbeitet werden. Es ist $f'(x) = \frac{1}{\sqrt{1-x^2}}$, und der Ableitungsterm in den Formeln ist $1/[\sqrt{1 - |[x_i]|^2_{\sup}}]_{\inf}$.

Die Behandlung des Arcuscosinus unterscheidet sich nicht von der des Arcussinus, was wegen $\arccos(x) = \frac{\pi}{2} - \arcsin(x)$ nicht verwundern sollte.

Es macht also zunächst *compute* Wiederberechnungsversuche, um Werte außerhalb des Definitionsbereichs $[-1, 1]$ aus den Operanden auszuschließen, und in *recompute* müssen ankommende Grenzen ± 1 ohne Mittelwertsatz behandelt werden. Beispielsweise ergibt sich bei $\tilde{x} = -1$ für den entsprechenden Term aus (5.1)

$$\begin{aligned} |\arccos(\tilde{x}) - \arccos(x)| &= |\pi - \arccos(x)| = \pi - \arccos(x) = \arccos(-x) \\ &= \arccos(\tilde{x} - x + 1) = \arccos(1 - |\Delta x|), \end{aligned}$$

also dasselbe wie für arcsin. Im Innern des Definitionsbereichs kann die Ableitung $f'(x) = -\frac{1}{\sqrt{1-x^2}}$ verwendet werden, die natürlich auf dieselben Beziehung und Stellenforderung führt wie bei arcsin. Dort kann allerdings der Funktionswert nicht 0 werden, was die Fallunterscheidung vereinfacht.

5.4.12 $\arctan(x)$, $\operatorname{arccot}(x)$

Bei diesen Funktionen ist $|f'(x)| = \frac{1}{1+x^2}$, so dass der Ableitungsterm 1 wird, wenn das Operandenintervall 0 enthält. Außerdem ist der Term so einfach zu berechnen, dass er nach einer Operandenneuberechnung ebenfalls (in x_2) neu berechnet wird. Es ist nur zu beachten, dass die Addition mit 1 nur z.B. mit ℓ Stellen und nicht exakt durchgeführt werden sollte, da sich bei exakter Rechnung und sehr kleinen Argumenten extrem lange Mantissen ergeben würden (die Implementation zu [Steins] stürzt deswegen ab oder braucht extrem viel Speicher und Rechenzeit).

Bei arctan vereinfacht sich die Fallunterscheidung durch $0 \in [y_1] \Leftrightarrow 0 \in [x_1]$, bei arccot gilt ohnehin immer $[y_1] > 0$. Bei arctan kann der Benutzer für $0 \in [x_1]$ Wiederberechnungsversuche einstellen.

5.4.13 $\operatorname{arsinh}(x)$

Falls der Benutzer dies eingestellt hat, werden hier für $0 \in [x_1]$ Wiederberechnungsversuche gemacht. Bei *recompute* ist genau dann $0 \in [y_1]$, wenn $0 \in [x_1]$. Die Ableitung ist als $f'(x) = \frac{1}{\sqrt{1+x^2}}$ auf ganz $\mathbb{R}$ problemlos. Als entsprechender Term wird $[\sqrt{1+|[x_1]|^2_{\inf}}]_{\inf}$ verwendet; die Beziehungen können nach Multiplikation schneller implementiert werden.

5.4.14 $\operatorname{arcosh}(x)$

Hier muss *compute* ggf. durch Wiederberechnungsversuche Werte kleiner als 1 aus dem Operandenintervall ausschließen (oder eine Exception werfen).

recompute kann für eine Untergrenze 1 des Operandenintervalls nicht die Formel aus dem Mittelwertsatz verwenden. Für den entsprechenden Term aus (5.1) gilt aber bei $\tilde{x} = 1$ (und Gleichverteilung der Toleranzen):

$$|\operatorname{arcosh}(\tilde{x}) - \operatorname{arcosh}(x)| = \operatorname{arcosh}(x) = \operatorname{arcosh}(x - \tilde{x} + 1) = \operatorname{arcosh}(1 + |\Delta x|) \overset{!}{\leq} \frac{\Delta}{2}.$$

Anwendung des Cosinus hyperbolicus (für $x \geq 0$ streng monoton steigend) liefert nun:

$$1 + |\Delta x| \leq \cosh\frac{\Delta}{2}, \qquad \Delta x \leq \cosh\Big(\frac{\Delta}{2}\Big) - 1 = 2\sinh^2\frac{\Delta}{4}.$$

Um den Aufruf des Sinus hyperbolicus zu umgehen, verwenden wir (analog zu arcsin) die Beziehung $\sinh u \geq u(1 + \frac{u^2}{6})$ für $u \geq 0$:

$$2\sinh^2\frac{\Delta}{4} \;\geq\; 2\Big[\frac{\Delta}{4}\Big(1+\frac{(\Delta/4)^2}{6}\Big)\Big]^2 \;=\; \frac{[\Delta(96+\Delta^2)]^2}{73728} \;\geq\; \frac{(96\,\Delta)^2}{73728} = \frac{\Delta^2}{8} = 0.125\,\Delta^2,$$

d.h. wir fordern für den Operanden schärfer $|\Delta x| \leq 0.125\,\Delta^2$.
Ansonsten ist die Ableitung $f'(x) = \frac{1}{\sqrt{x^2-1}}$ verwendbar, der vorkommende Ableitungsterm ist $[\sqrt{[x_i]^2_{\inf} - 1}]$. Da nie $0 \in [x_1]$ und nie $0 \in [y_1]$, vereinfacht sich die Fallunterscheidung entsprechend.

5.4.15 artanh(x)

Bei dieser Funktion müssen in *compute* ggf. wieder Werte ≤ -1 oder $\geq +1$ ausgeschlossen werden. Außerdem kann der Benutzer festlegen, dass Wiederberechnung stattfinden soll, wenn $0 \in [y_1]$.

In *recompute* gibt es dann keine Besonderheiten mehr. Es ist $f'(x) = \frac{1}{1-x^2}$, und für den (inversen) Ableitungsterm wird $1 - |[x_i]|^2_{\sup}$ verwendet. Da dieser so einfach zu berechnen ist, wird bei der Bestimmung der Stellenzahl für die neue artanh-Berechnung dieser Term in $x_{\underline{2}}$ verwendet.

5.4.16 arcoth(x)

Wenn hier Elemente von $[-1, 1]$ im Operandenintervall liegen, werden Wiederberechnungsversuche gemacht, bis der Operand vollständig in $[-1, 1]$ liegt (Abbruch mit Exception), bis der Operand ganz unterhalb von -1 oder oberhalb von $+1$ liegt, oder bis die eingestellte Maximalzahl an Versuchen erreicht ist.

Die Ableitung von arcoth ist dieselbe ist wie die von artanh (die Definitionsbereiche sind disjunkt). Allerdings enthalten hier Operanden und Funktionswerteinschließungen nie die 0, so dass sich die Fallunterscheidung stark vereinfacht.

5.4.17 sqrt(x)

Bei der Quadratwurzel gibt es im ersten Schritt Probleme, falls der Operand negative Werte enthält. Es werden dann, wie in 5.3.2 Teil b) beschrieben, Wiederberechnungsversuche gemacht, die den Intervalldurchmesser auf unter die Hälfte der Abstände der Grenzen von 0 drücken. Wenn ein dabei entstehendes Intervall ganz im Negativen zu liegen kommt, werfen wir (berechtigterweise) eine Exception. Wenn der Benutzer dies eingestellt hat, wird ggf. eine Wiederberechnung versucht, bis der Operand ganz im Positiven liegt.

Bei der eigentlichen Wiederberechnung des Wurzelausdrucks mit *recompute* kann nun noch ein Intervall ankommen, das 0 als Untergrenze hat. Da die Wurzel in 0 nicht differenzierbar ist, kann nicht Beziehung (5.1) für die Genauigkeitsabschätzungen herangezogen werden.

Es gilt aber (nicht nur für $\tilde{x} = 0$) $|\sqrt{\tilde{x}} - \sqrt{x}| \leq \sqrt{|\tilde{x} - x|}$, d.h. dieser Term ist bei uns durch $\sqrt{\mathrm{diam}[x]}$ beschränkt. Wenn die Fehlertoleranz Δ wie üblich gleich verteilt wird, erhält man für den neu berechneten Operanden die Forderung

$$\sqrt{\mathrm{diam}[x_2]} \leq \frac{\Delta}{2}, \quad \text{bzw.} \quad \mathrm{diam}[x_2] \leq \frac{\Delta^2}{4}$$

und damit die Stellenforderung $b^{-n} \leq \frac{b^{-2\ell}}{4}$, bzw. bei $b = 10$ schärfer $n := 2\ell + 1$.

Wenn der Operand danach immer noch 0 enthält, braucht die neue Wurzelberechnung nur punktweise auf die Obergrenze angewendet zu werden. Jedenfalls ergibt sich die Stellenforderung

$$b^{-m+1} \leq \frac{b^{-\ell}}{2[y_1]_{\sup}}, \quad \text{berechnet als} \quad m := \ell + 2 + \mathrm{order}(2[y_1]_{\sup}).$$

Falls der Operand nicht die Null enthält, können die normalen Beziehungen, die die Ableitung $f'(x) = \frac{1}{2\sqrt{x}}$ verwenden, benutzt werden. Für den Ableitungsterm $|[f([x_i])]|_{\sup}$ wird $\frac{1}{2[y_1]_{\inf}}$ eingesetzt. Als Bedingung für genügende Genauigkeit des Operanden und die notwendige neue Stellenzahl bei der Wurzel ergibt sich dann

$$\operatorname{diam}[x_1] < 2b^{-\ell}[y_1]_{\inf}^2, \qquad m := 1 - \operatorname{order} \frac{2b^{-\ell}[y_1]_{\inf}^2 - \operatorname{diam}[x_1]}{4[y_1]_{\inf}[y_1]_{\sup}}.$$

Ansonsten ergeben sich folgende Stellenzahlen für Operand und Wurzel:

$$n := \ell + 1 - \operatorname{order} \frac{[y_1]_{\inf}^2}{2[x_1]_{\sup}}, \qquad m := 1 - \operatorname{order} \frac{b^{-\ell}[y_1]_{\inf} - \frac{\operatorname{diam}[x_2]}{2[y_1]_{\inf}}}{2[y_1]_{\sup}}.$$

Es ist dabei zu beachten, dass die Implementation der Quadratwurzel maximale Genauigkeit garantiert und man in den entsprechenden Nennern den Faktor 2 statt 4 erhält.

5.4.18 $\ln(x)$, $\log_{10}(x)$

Auch bei diesen Funktionen muss in *compute* der Fall behandelt werden, dass das Operanden-Intervall negative oder verschwindende Zahlen enthält. Man kann dabei genauso verfahren wie bei der Quadratwurzel. Falls Wiederberechnungsversuche bei Einschließung von 0 eingestellt wurden, wird wiederum versucht, den Intervalldurchmesser auf unter die Hälfte des Abstands der Grenzen zu 1 zu drücken.

Bei *recompute* kommen nun nur noch positive Argumente an. Die Ableitungen ergeben sich zu $\frac{1}{x}$ bzw. $\frac{1}{\ln 10} \cdot \frac{1}{x}$, so dass für den Term $|[f'([x])]|_{\sup}$ im ersten Fall $\frac{1}{[x]_{\inf}}$, im zweiten Fall $\frac{1}{[\ln 10]_{\inf}[x]_{\inf}}$ verwendet werden können. Die Abschätzungen und Stellenforderungen können wieder durch Multiplikationen günstig umgeformt werden.

5.4.19 $\texttt{root}(x, n) = \sqrt[n]{x}$

Falls n gerade ist und der Operand ganz im Negativen liegt, wirft *compute* eine Exception; falls der Operand auch positive Werte hat, versucht die Funktion, das Problem mit einer Wiederberechnung des Operanden zu lösen. Außerdem kann der Benutzer einstellen, dass für $0 \in [x_1]$ immer eine Ausschließung der 0 versucht wird.

In *recompute* gibt es analog zu $\sqrt{x}$ Probleme, falls der Operand 0 enthält, da die Ableitung der Funktion dort nicht existiert (und auch keine Lipschitz-Konstante als Ersatzfaktor vor $\operatorname{diam}[x_i]$). Falls nur eine Grenze des Operandenintervalls 0 ist, ist für den entsprechenden Term in (5.1) der Fall zu betrachten, dass $\tilde{x} = 0$ oder $x = 0$, d.h. (o.B.d.A. $\tilde{x} = 0$):

$$|\sqrt[n]{\tilde{x}} - \sqrt[n]{x}| \;=\; |0 - \sqrt[n]{x}| \;=\; \sqrt[n]{|x|},$$

d.h. der Term kann durch $\sqrt[n]{\operatorname{diam}[x_i]}$ nach oben abgeschätzt werden.

Falls dagegen 0 im Innern des Intervalls liegt, ist o.B.d.A. $\tilde{x} < 0$ und $x > 0$, und damit

$$|\sqrt[n]{\tilde{x}} - \sqrt[n]{x}| \;=\; \sqrt[n]{|\tilde{x}|} + \sqrt[n]{|x|} \;\leq\; 2\sqrt[n]{|\tilde{x} - x|},$$

hier kann also mit $2\sqrt[n]{\operatorname{diam}[x_i]}$ abgeschätzt werden. In beiden Fällen ist zu beachten, dass sich das n so *multiplikativ* in den Stellenforderungen bemerkbar machen wird!

Wenn 0 im Operanden enthalten ist, wird immer eine Wiederberechnung angestrengt. Wenn 0 eine der beiden Grenzen ist, ergibt sich als Genauigkeitsforderung

$$\sqrt[n]{\operatorname{diam}[x_2]} \;\leq\; \frac{b^{-\ell}}{2},$$

woraus man die Stellenforderung $m \geq \ell \cdot n + \log_{10}(2^n)$ erhält, gröber berechnet als $m := \ell \cdot n + 0.31 \cdot n$. Wenn 0 im Innern liegt, ergibt sich analog $m := \ell \cdot n + 0.61 \cdot n$. Bei großen n können sich daher sehr große Stellenzahlen für die Wiederberechnung ergeben.

Im Normalfall $0 \notin [x_1]$ kann $f'(x) = \frac{1}{n} \cdot x^{(\frac{1}{n}-1)} = \frac{\sqrt[n]{x}}{n \cdot x}$ verwendet werden. Es ergibt sich dann als Genauigkeitsbedingung für den Operanden bzw. als Stellenforderung an die Neuberechnung der Wurzel

$$|[y_1]|_{\sup} \operatorname{diam}[x_1] \;\leq\; n\, b^{-\ell}\, |[y_1]|_{\inf}\, |[x_1]|_{\inf}\,, \qquad q := 1 - \operatorname{order} \frac{b^{-\ell}\, |[y_1]|_{\inf} - \frac{|[y_1]|_{\sup}}{n\, |[x_1]|_{\inf}} \operatorname{diam}[x_1]}{4\, |[y_1]|_{\sup}}.$$

Ansonsten erhält man als Stellenzahlen für Operand und Wurzel

$$m := \ell + 1 - \operatorname{order} \frac{n\, |[y_1]|_{\inf}\, |[x_1]|_{\inf}}{4\, |[y_1]|_{\sup}\, |[x_1]|_{\sup}}\,, \qquad q := \ell + 1 - \operatorname{order} \frac{|[y_1]|_{\inf}}{8\, |[y_1]|_{\sup}}.$$

Zusätzlich zu den angegebenen Standardfunktionen sind folgende Funktionen implementiert, die in vielen Programmiersprachen zur Verfügung stehen:

5.4.20 $\operatorname{abs}(x) = |x|$

Der Absolutbetrag ist zwar in 0 nicht differenzierbar, es gilt aber für den Term aus (5.1) $||\tilde{x}| - |x|| \leq |\tilde{x} - x|$ (Lipschitz-Konstante 1), und bei der Auswertung der Funktion wird natürlich kein Fehler gemacht. Dadurch können Stellenforderungen an die Funktion einfach an den Operanden weitergegeben werden. Der Benutzer kann einstellen, dass Wiederberechnungsversuche angestellt werden, wenn der Operand 0 einschließt (um eine Untergrenze 0 des Ergebnisses zu vermeiden).

5.4.21 `sign`(x)

Diese Funktion berechnet das Vorzeichen von x, $\in \{-1, 0, +1\}$. In *compute* muss gegebenenfalls der Operand wiederberechnet werden, bis eindeutig über seine Lage entschieden werden kann. Dabei wird so wie bei den Funktionen vorgegangen, die 0 aus dem Definitionsbereich ausschließen müssen (z.B. `sqrt`). In *recompute* ist dann gar nichts mehr zu tun.

Bei Benutzung dieser Funktion sollten immer Exceptions abgefangen werden, für den Fall, dass die angegebene Maximalzahl von Wiederberechnungsversuchen erreicht wird.

5.4.22 `floor`$(x) = \lfloor x \rfloor$, `ceil`$(x) = \lceil x \rceil$

Hierbei muss der Operand ggf. so lange wiederberechnet werden, bis Obergrenze und Untergrenze unter der jeweiligen Punktfunktion denselben Wert liefern. (Falls der Durchmesser

des Intervalls größer als 1 ist, wird er zunächst unter $\frac{1}{2}$ gedrückt. Anschließend wird ähnlich wie beim Ausschließen einer Definitionslücke gearbeitet.)

Der Benutzer sollte hier auf jeden Fall Exceptions abfangen! Besonders bei Ausdrücken der Art $\exp(\log(n))$, $n \in \mathbb{N}$, ist Vorsicht geboten, da n immer echt eingeschlossen werden wird.

5.4.23 `round`(x)

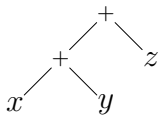
Dies liefert die „nächstliegende“ ganze Zahl, wobei für ein Argument exakt zwischen zwei benachbarten ganzen Zahlen nach außen gerundet wird. Der Operand wird wiederberechnet, bis Ober- und Untergrenze zur selben ganzen Zahl am nächsten liegen. Probleme gibt es hier, falls gerade eine Zahl $z + \frac{1}{2}$, $z \in \mathbb{Z}$, eingeschlossen wird. Es wird dann eine Exception geworfen, wenn die eingestellte Maximalzahl von Versuchen erreicht ist.

5.5 Wiederberechnung bei binären Operatoren

Bei uns sind neben der allgemeinen Potenzfunktion x^y (in Form der Funktion $\mathrm{pow}(x, y)$) die vier Grundrechenarten als binäre Operatoren implementiert. Es ergeben sich gegenüber den unären Operatoren keine neuen Ideen oder Vorgehensweisen. Die erlaubte Fehlertoleranz wird nun lediglich ggf. auf drei Terme verteilt werden. Bei den Grundrechenarten kommt man zudem immer ohne Verwendung des Mittelwertsatzes mit klassischer Fehlerrechnung aus.

Es sei vorab darauf hingewiesen, dass in dem zum auszuwertenden Ausdruck gehörigen Baum *keine* n-äre Struktur verwendet wird. Letzteres hätte insofern Vorteile, als Maximalfehler gleichmäßiger auf alle (eigentlich gleichgeordneten) Operanden verteilt werden könnten.

Im rechts stehenden Baum wird die im oberen Knoten zur Verfügung stehende Fehlertoleranz auf die beiden Nachfolgerknoten gleichmäßig verteilt und im linken Knoten erneut halbiert. Damit werden x und y doppelt so genau berechnet wie z.



Der Verzicht auf n-äre Strukturen liegt allein darin begründet, dass die Operatoren in `C++` binär sind und daher automatisch durch den vom Compiler erzeugten Code implizit eine Binärbaumstruktur entsteht. Es wäre umständlich und zeitraubend, diese Struktur nachträglich wieder auseinanderzureißen und neu zu ordnen.

In einer möglichen Erweiterung unserer Arithmetik könnte man eine Knotenart „Addition“ mit n vorzeichenbehafteten Nachfolgern und eine Knotenart „Multiplikation“ mit n Nachfolgern, die mit „invers“ markiert sein können, verwenden. Ein solches Vorgehen wäre dann sinnvoll, wenn Ausdrücke mehrfach (beispielsweise mit zwischenzeitlichen Änderungen von Variablenwerten) ausgewertet werden sollen und das Umordnen nur auf ausdrücklichen Wunsch des Benutzers hin ausgelöst würde.

Im Normalfall wird sich unsere momentane Vorgehensweise nicht sonderlich nachteilig auswirken. Wenn man Summen oder Produkte mit sehr vielen Operanden verwendet, könnte es (aus Geschwindigkeitsgründen) manchmal hilfreich sein, besonders zeitaufwändig zu berechnende Terme eher ans Ende zu setzen – da die vom Compiler erzeugten Binärbäume

vermutlich eher linkslastig sind wie der oben dargestellte – oder gleichmäßig zu klammern, etwa $((x_1 - x_2) + (x_3 + x_4))$.

Wir verwenden in diesem Abschnitt als Namen der Operanden immer $z := x \circ y$, um mit dem Index die Herkunft aus dem ersten oder zweiten Berechnungsschritt bezeichnen zu können. Die Einschließung aus *compute* ist also $[z_1] = [[x_1] \circ [y_1]]$. Wenn ein Operand als schon genau genug akzeptiert wird, ist $[x_2] := [x_1]$ bzw. $[y_2] := [y_1]$.

Die im jeweiligen Schritt geforderte Stellengenauigkeit heiße wieder ℓ. Analog zum Fall der unären Operatoren sei die Fehlertoleranz im Wiederberechnungsschritt

$$\Delta := \begin{cases} b^{-\ell} \cdot |[z_1]|_{\inf} & \text{für } 0 \notin [z_1], \\ b^{-\ell} & \text{für } 0 \in [z_1] \end{cases}$$

(bzw. $0.99 \cdot b^{-\ell}$ im zweiten Fall bei durchgeführter Endrundung).

Bei den Stellenforderungen an die eigentliche Operation ist noch zu beachten, dass die Grundrechenarten (nicht das Potenzieren!) maximal genau implementiert sind.

5.5.1 Die Addition und Subtraktion

Wir betreiben kurz klassische Fehleranalyse für $z = x \pm y$ bei gestörten Eingabedaten $\tilde{x}$, $\tilde{y}$ und inexakter Operation; die absoluten Fehler heißen wie üblich Δx, Δy.

$$\begin{aligned} |\Delta z| &= |\tilde{z} - z| = |(1+\varepsilon)(\tilde{x} \pm \tilde{y}) - (x \pm y)| \\ &= |\varepsilon(\tilde{x} \pm \tilde{y}) + (\tilde{x} - x) \pm (\tilde{y} - y)| \\ &= |\varepsilon(\tilde{x} \pm \tilde{y}) + \Delta x \pm \Delta y| \\ &\leq |\varepsilon(\tilde{x} \pm \tilde{y})| + |\Delta x| + |\Delta y| \quad \overset{!}{\leq} \Delta \end{aligned}$$

(mit $|\varepsilon| \leq \varepsilon_\pm$), d.h. insbesondere können Addition und Subtraktion gleich behandelt werden.

Wir gehen also wie folgt vor. Für $|[z_1]|_{\sup} = 0$ ist nichts zu tun. Wenn ansonsten $\operatorname{diam}[x_1] + \operatorname{diam}[y_1] < \Delta$, sind beide Operanden bereits exakt genug vorberechnet. Die Resttoleranz wird für die Operation verwendet:

$$m := 1 - \operatorname{order} \frac{\Delta - \operatorname{diam}[x_1] - \operatorname{diam}[y_1]}{2|[z_1]|_{\sup}}.$$

Ansonsten wird die Fehlertoleranz auf beide Operanden und die Operation aufgeteilt. Da eine Grundrechenart wesentlich schneller berechenbar sein dürfte als ein ganzer Teilbaum, erfolgt die Teilung aber nicht in gleiche Teile, vielmehr als $\frac{1}{5} + \frac{2}{5} + \frac{2}{5}$ (davon die kleinste Toleranz für die Operation). Sicherlich wären noch linkslastigere Teilungen denkbar, aber die Division durch 5 ist exakt darstellbar (genauer wird natürlich $\frac{1}{5} = \frac{2}{10}$, $\frac{2}{5} = \frac{4}{10}$ benutzt, also mit 2 bzw. mit 4 multipliziert und der Exponent erniedrigt). Die nächstgrößere ungerade Zahl, für die dies gilt, ist 25.

Für die Neuberechnung von x (analog für y) bedeutet das für $0 \notin [x_1]$ bzw. $0 \in [x_1]$ folgende Stellenzahl:

$$n_x := 1 - \operatorname{order} \frac{\Delta}{5|[x_1]|_{\sup}}, \qquad \text{bzw.} \qquad n_x := -\operatorname{order} \frac{2\,\Delta}{5}.$$

Falls nach der Berechnung des ersten Operanden die Genauigkeitsbedingung schon erfüllt sein sollte, braucht der zweite nicht neu berechnet zu werden.

Die bei der Neuberechnung des ersten Operanden möglicherweise entstandene zusätzliche Toleranz wird der Neuberechnung des zweiten Operanden zugute kommen gelassen. Leider kann man den Operanden auf dieser Ebene nicht ansehen, welcher der beiden zuerst berechnet werden sollte, damit die Neuberechnung des anderen möglichst überflüssig wird. Es wird immer zuerst der linke, dann der rechte berechnet.

Für die abschließende Addition bzw. Subtraktion erhält man durch das Gewicht $\frac{1}{5}$ als Stellenzahl

$$m := 1 - \text{order } \frac{\Delta}{10|[z_1]|_{\sup}}.$$

Durch die Geschwindigkeit der Operation lohnt es nicht, $\Delta - \text{diam}[x_1] - \text{diam}[x_2]$ zu verwenden und dabei zwei weitere Subtraktionen durchzuführen!

Auf jeden Fall sollte die *inexakte* Version der Operation verwendet werden. Bei [Steins] wird wieder exakt mit voller Mantissenlänge der Operanden gerechnet und anschließend gerundet, wodurch bei Summanden sehr unterschiedlicher Größenordnung die schon mehrfach besprochenen Probleme mit überlangen Mantissen auftreten.

5.5.2 Die Multiplikation

Mit klassischer Fehleranalyse erhalten wir ($|\varepsilon| \leq \varepsilon_*$):

$$\begin{aligned} \Delta z = \tilde{z} - z &= (1+\varepsilon)(\tilde{x} \cdot \tilde{y}) - x \cdot y &&= \varepsilon\tilde{x}\tilde{y} + \tilde{x}\tilde{y} - xy \\ &= \varepsilon\tilde{x}\tilde{y} + (x + \Delta x)\tilde{y} - xy &&= \varepsilon\tilde{x}\tilde{y} + \tilde{y}\Delta x + x(\tilde{y} - y) \\ &= \varepsilon\tilde{x}\tilde{y} + \tilde{y}\Delta x + x\Delta y \\ \text{(bzw. symmetrisch} \quad \Delta z &= \varepsilon\tilde{x}\tilde{y} + y\Delta x + \tilde{x}\Delta y \text{);} \end{aligned}$$

da wir sowohl $|x|$ wie auch $|\tilde{x}|$ nur durch $|[x_i]|_{\sup}$ abschätzen können (analog für y), ist es gleichgültig, welche der beiden letzten Formen verwendet wird.

Unser Vorgehen ist nun völlig analog zu dem bei der Addition. Falls

$$|[y_1]|_{\sup} \text{ diam}[x_1] \; + \; |[x_1]|_{\sup} \text{ diam}[y_1] \; \overset{!}{<} \; \Delta,$$

sind beide Operanden sicher genau genug vorberechnet. Aus der verbleibenden Toleranz erhält man für die Stellenzahl der Multiplikation

$$m := 1 - \text{order } \frac{\Delta - |[y_1]|_{\sup} \text{diam}[x_1] - |[x_1]|_{\sup} \text{diam}[y_1]}{2|[z_1]|_{\sup}}.$$

Ansonsten werden die einzelnen Summanden durch Wiederberechnung wieder auf maximal $\frac{2\Delta}{5}$ gedrückt. Für x bedeutet das, für $0 \notin [x_1]$ bzw. $0 \in [x_1]$, die minimale Stellenzahl

$$n_x := 1 - \text{order } \frac{\Delta}{5|[x_1]|_{\sup}|[y_1]|_{\sup}}, \qquad \text{bzw.} \qquad n_x := -\text{order } \frac{2\Delta}{5|[y_1]|_{\sup}},$$

(analog für y). Die durch die erste Neuberechnung erzielte zusätzliche Toleranz wird außerdem der zweiten Berechnung zugeschlagen.

Wenn die Gesamtbedingung nach Berechnung eines Operanden schon erfüllt sein sollte, braucht der zweite natürlich nicht neu berechnet zu werden. Aus diesem Grunde wird, falls genau eine Einschließung noch 0 enthält, *dieser* Operand zuerst berechnet, da möglicherweise durch ihn das Vorzeichen des Produkts festgelegt werden könnte. (Ansonsten wird immer der linke Operand zuerst berechnet.)

Für die neue Multiplikation erhält man als Stellenzahl

$$m := 1 - \text{order}\ \frac{\Delta}{10|[z_1]|_{\text{sup}}}.$$

Im Hinblick auf die Geschwindigkeit der Multiplikation lohnt es nicht, die im ersten Fall angegebene Beziehung zu verwenden, da dort zu viele weitere Operationen (insbesondere wieder Multiplikationen) vorkommen.

5.5.3 Die Division

Falls nach dem ersten Durchgang 0 in der Einschließung des Nenners enthalten sein sollte (aber der Nenner nicht exakt 0 ist), versucht bereits *compute* dies mit Wiederberechnungsschritten zu korrigieren (analog zu log und sqrt, etc.). Erst bei Erreichen der eingestellten Maximalzahl wird eine Exception geworfen.

Die Fehleranalyse der Division liefert ($|\varepsilon| \leq \varepsilon_/$):

$$\begin{aligned}
\Delta z = \tilde{z} - z &= (1+\varepsilon)\frac{\tilde{x}}{\tilde{y}} - \frac{x}{y} &&= \varepsilon\frac{\tilde{x}}{\tilde{y}} + \frac{\tilde{x}}{\tilde{y}} - \frac{x}{y} \\
&= \varepsilon\frac{\tilde{x}}{\tilde{y}} + \frac{x+\Delta x}{\tilde{y}} - \frac{x}{y} &&= \varepsilon\frac{\tilde{x}}{\tilde{y}} + \frac{1}{\tilde{y}}\Delta x + \frac{x}{\tilde{y}} - \frac{x}{y} \\
&= \varepsilon\frac{\tilde{x}}{\tilde{y}} + \frac{1}{\tilde{y}}\Delta x + \frac{y-\tilde{y}}{y\tilde{y}}x &&= \varepsilon\frac{\tilde{x}}{\tilde{y}} + \frac{1}{\tilde{y}}\Delta x - \frac{\Delta y}{y\tilde{y}}x \\
&= \varepsilon\frac{\tilde{x}}{\tilde{y}} + \frac{1}{\tilde{y}}\Delta x - \frac{x}{y\tilde{y}}\Delta y &&= \varepsilon\frac{\tilde{x}}{\tilde{y}} + \frac{1}{\tilde{y}}\Delta x - \frac{z}{\tilde{y}}\Delta y.
\end{aligned}$$

Falls $|[z_1]|_{\text{sup}} = 0$, ist nichts zu tun. Falls sonst

$$\text{diam}[x_1] \ + \ |[z_1]|_{\text{sup}} \cdot \text{diam}[y_1] \ \leq \ \Delta \cdot |[y_1]|_{\text{inf}},$$

sind beide Operanden bereits genau genug. Durch die Resttoleranz erhält man als Stellenzahl der neu durchzuführenden Division

$$m := 1 - \text{order}\ \frac{\Delta\,|[y_1]|_{\text{inf}} - \text{diam}[x_1] - |[z_1]|_{\text{sup}}\ \text{diam}[y_1]}{2\,|[x_1]|_{\text{sup}}}.$$

Im anderen Fall wird zuerst x und ggf. auch y neu berechnet. Für x ergeben sich die notwendigen Stellenzahlen für $0 \notin [x_1]$ bzw. $0 \in [x_1]$:

$$n_x := 1 - \text{order}\ \frac{\Delta\,|[y_1]|_{\text{inf}}}{5\,|[x_1]|_{\text{sup}}}, \quad \text{bzw.} \quad n_x := -\,\text{order}\ \frac{2\,\Delta\,|[y_1]|_{\text{inf}}}{5},$$

und für y (es gilt nach *compute* immer $0 \notin [y_1]$):

$$n_y := 1 - \text{order } \frac{2\Delta |[y_1]|_{\inf}}{5\,|[y_1]|_{\sup}\,|[z_1]|_{\sup}}\,.$$

Schließlich wird die Division mit

$$m := 1 - \text{order } \frac{\Delta}{10\,|[z_1]|_{\sup}}$$

Stellen durchgeführt.

5.5.4 Das Potenzieren $x^y = \text{pow}(x, y)$

Beim Potenzieren wird schon durch *compute* sichergestellt, dass nur Kombinationen von Operanden beim Wiederberechnungsschritt *recompute* ankommen, für die das Potenzieren definiert ist. Beispielsweise wird Folgendes durchgeführt:

Falls x negative Punkte enthält, ist (in unserer Arithmetik) x^y nur definiert, wenn $y \in \mathbb{Z}$ (als Punktintervall) ist. Sollte das nicht der Fall sein, ist es äußerst unwahrscheinlich, dass es durch einen Wiederberechnungsschritt noch erreicht werden kann. Dagegen kann, wenn x auch positive Werte enthält, durch Wiederberechnung versucht werden, die negativen Werte aus x zu eliminieren. Wenn $y \in \mathbb{Z}$ ist, aber x die 0 enthält, wird in einem Wiederberechnungsschritt versucht, diese aus x zu entfernen.

Für den eigentlichen Wiederberechnungsschritt in *recompute* betrachten wir zunächst den „Normalfall“ $x > 0$. Falls y ein Punktintervall sein sollte, wird dies getrennt günstiger behandelt. Es gilt allgemein

$$\begin{aligned}
|\Delta z| &= |\widetilde{\text{pow}}\,(\tilde{x}, \tilde{y}) - \text{pow}(x, y)| \\
&= |\widetilde{\text{pow}}\,(\tilde{x}, \tilde{y}) - \text{pow}(\tilde{x}, \tilde{y}) + \text{pow}(\tilde{x}, \tilde{y}) - \text{pow}(x, \tilde{y}) + \text{pow}(x, \tilde{y}) - \text{pow}(x, y)| \\
&= |\Delta\,\text{pow}(\tilde{x}, \tilde{y}) + \tilde{y}\,\xi_1^{\tilde{y}-1}\,\Delta x + x^{\xi_2}\,\ln(x)\,\Delta y| \\
&\leq |\Delta\,\text{pow}(\tilde{x}, \tilde{y})| + |\tilde{y}\,\xi_1^{\tilde{y}-1}\,\Delta x| + |x^{\xi_2}\,\ln(x)\,\Delta y| \quad \overset{!}{\leq} \Delta
\end{aligned}$$

für $\xi_1 \in \text{conv}\{x, \tilde{x}\}$, $\xi_2 \in \text{conv}\{y, \tilde{y}\}$. Entsprechend sei nun

$$A := |[y_1]|_{\sup}\,\frac{[z_1]_{\sup}}{[x_1]_{\inf}}, \qquad B := [z_1]_{\sup}\,|[\ln([x_1])]|_{\sup}$$

(wobei bei Letzterem der Logarithmus nur an einer Intervallgrenze berechnet zu werden braucht: falls $[x_1] \geq 1$ oder $[x_1]_{\inf}[x_1]_{\sup} \geq 1$ an der oberen, ansonsten an der unteren Grenze).

a) Falls

$$A \cdot \text{diam}[x_1] \;+\; B \cdot \text{diam}[x_2] \;<\; b^{-\ell} \cdot [z_1]_{\inf},$$

so liegen die beiden Operanden bereits genau genug vor, und die Stellenzahl für die Neuberechnung der Potenzierung ergibt sich als

$$q := 1 - \text{order } \frac{b^{-\ell}\,[z_1]_{\inf} - A\;\text{diam}[x_1] - B\;\text{diam}[y_1]}{4 \cdot [z_1]_{\sup}}.$$

b) Ansonsten wird zunächst getestet, ob $[x_1]$ bereits genau genug ist, d.h. bei Drittelung der Toleranz:

$$3\,A\ \mathrm{diam}[x_1] < b^{-\ell}\,[z_1]_{\inf}.$$

Ansonsten wird $[x_1]$ mit folgender Stellenzahl neu berechnet:

$$m := 1 - \mathrm{order}\,\frac{b^{-\ell}\,[z_1]_{\inf}}{6.001\,A\,[x_1]_{\sup}}.$$

Es ist noch Folgendes zu beachten: Falls die Einschließung des linken Operanden in *compute* noch die 0 enthält, kann der Exponent es erzwingen, dass die 0 durch einen frühen Wiederberechnungsschritt ausgeschlossen wird. Danach ist der relative Fehler für die (nun gerade positiv gewordene) Einschließung verhältnismäßig groß.

Wenn dann zusätzlich noch der Exponent betragsmäßig sehr groß ist, kommt es in diesem Schritt ggf. zu extrem großen Stellenforderungen (z.B. wird $[z_1]_{\inf}$ wesentlich zu klein geschätzt). In einem Testlauf ergaben sich z.B. einmal 2773 Stellen, obwohl nur 16 Stellen des Endergebnisses gefordert waren!

Wenn sich daher eine Stellenzahl größer als 3ℓ ergibt, wird zunächst $[x_2]$ nur mit 3ℓ Stellen berechnet, und daraus ein $[z_2]$ und ein neues B. Danach wird Schritt b) erneut durchgeführt, wobei dann typischerweise die geforderte Genauigkeit bereits erreicht ist.

c) Falls der rechte Operand nun nicht genau genug ist (Test wie in a)), wird er neu berechnet, und zwar für $0 \notin [y_1]$ mit der Stellenzahl

$$n := \ell + 1 - \mathrm{order}\,\frac{b^{-\ell}\,[z_2]_{\inf} - A\ \mathrm{diam}[x_2]}{2.001\,B\,|[y_2]|_{\sup}},$$

bzw. für $0 \in [y_1]$ mit der Stellenzahl

$$m := \ell - \mathrm{order}\,\frac{b^{-\ell}\,[z_2]_{\inf} - 4\ \mathrm{diam}[x_2]}{1.001\,B}.$$

d) Schließlich wird die eigentliche Potenzierung mit folgender Stellenzahl neu durchgeführt:

$$q := 1 - \mathrm{order}\,\frac{b^{-\ell}[z_2]_{\inf} - A\ \mathrm{diam}[x_2] - B\ \mathrm{diam}[y_2]}{4\,[z_2]_{\sup}}.$$

Für den Fall, dass y ein Punktintervall ist, vereinfachen sich die Betrachtungen von oben entsprechend, da nur eine Funktion in einer Variablen $f(x) = x^y$ mit $f'(x) = y\,x^{y-1}$ zu betrachten ist. Außerdem können so die Fälle behandelt werden, in denen x negative Punkte enthält (und $y \in \mathbb{Z}$). Der Ausdruck ξ_1^{y-1} von oben kann dabei durch $|[z_1]|_{\sup}/|[x_1]|_{\sup}$ ersetzt werden (für $0 \in x$, also $y > 0$), bzw. durch $|[z_1]_{\sup}/|[x_1]_{\inf}|$ (für unterschiedliches Vorzeichen von x und y), bzw. durch $|[z_1]|_{\sup}/|[x_1]_{\sup}|$ (für gleiches Vorzeichen).

5.6 Intervall-Konstanten

Es sei noch einmal ausdrücklich betont, dass der Wiederauswertungs-Algorithmus in dieser Form die Hochgenauigkeit nur gewährleistet, wenn die vorkommenden Konstanten bzw. Variablen als exakt angesehen werden können. Es sind also keine Intervalle (mit nicht verschwindendem Durchmesser) als „Atome“ zulässig! Solange der Intervalldurchmesser klein ist und mit relativ kleinen Genauigkeiten gerechnet wird, kann es sein, dass die sich ergebenden Genauigkeitsforderungen an diese Atome erfüllbar bleiben würden. Ansonsten wäre die Voraussetzung für den aufrufenden Wiederberechnungsschritt nicht erfüllt. Wenn der Algorithmus dennoch weitergeführt würde, ergäben sich nicht voraussagbare Ergebnisse.

Bei Intervallauswertungen treten selbst bei theoretisch exakter Rechnung Überschätzungen auf, sobald eine Variable mehr als einmal vorkommt. Daher ist es auch fraglich, ob es wirklich sinnvoll wäre, ein Verfahren zu konstruieren, das Ausdrücke mit Intervall-Konstanten auswertet und für die berechneten Grenzen Hochgenauigkeit garantiert.

Die Implementation zu [Steins] lässt aber (kommentarlos) auch Intervall-Atome zu. Natürlich wird so auf jeden Fall eine garantierte Einschließung berechnet, die mindestens so eng ist wie die mit normaler Intervall-Arithmetik derselben Stellenzahl (durch den ersten Durchgang, nach dem in allen Knoten die Genauigkeit jedenfalls nicht verringert wird). Welche Genauigkeiten die Grenzen (in Bezug auf die exakten Grenzen bei theoretischer exakter Intervall-Auswertung haben), ist aber unklar und von vielen „Zufällen“ abhängig.

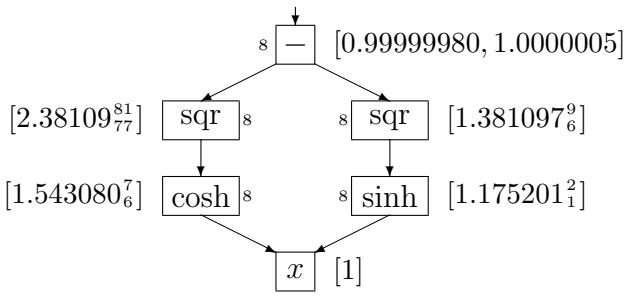
Die Steins'schen Formeln verteilen beispielsweise die Fehlertoleranz bei den Standardfunktionen *immer* zur Hälfte auf den Operanden und zur Hälfte auf die Funktion. Wir lassen dagegen teilweise nach der Neuberechnung des Operanden die zusätzlich entstandene Toleranz der Funktionsauswertung zugute kommen. Wenn der Operand nun aber durch die Neuberechnung nicht so genau geworden ist, wie der Algorithmus es annimmt, sind die entsprechenden Stellenberechnungen nicht anwendbar (die Differenz der Fehler wird unerwartet negativ).

- Für unsere Implementation legen wir daher fest, dass zur Compilationszeit durch eine Einstellung in `XArithConfig.h` Intervalle (`BFInterval`) als Atome *zugelassen* oder *verboten* werden (die Voreinstellung ist „verboten“).

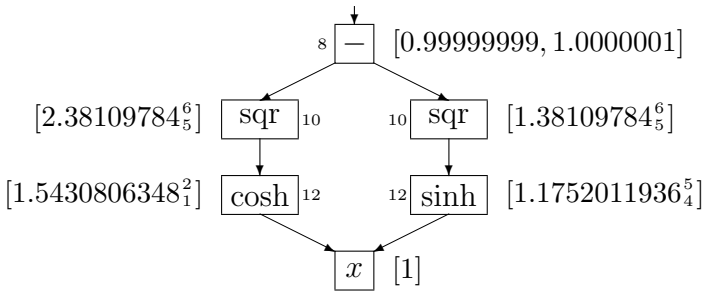
Die entsprechende Konstante heißt `allow_intervals`. Falls Intervalle zugelassen werden, wird im oben beschriebenen Fall, dass der Operand nicht genau genug geworden ist, die neue Stellenzahl nach der Gleichverteilungs-Methode bestimmt. Es kann dann aber keine Genauigkeitsaussage für das Ergebnis gemacht werden.

5.7 Beispiele

1. Wir werten $\cosh^2(x) - \sinh^2(x)$ in 1 mit der dezimalen Genauigkeit von 8 Stellen aus. Im ersten Durchgang werde in allen Operationen mit 8 Stellen gearbeitet:



Nach diesem Schritt reicht die erzielte Genauigkeit in der Wurzel nicht aus. Für den Wiederberechnungsschritt ergeben sich hier Stellenzahlen von 10 bzw. 12:



- $\cosh^2$ und $\sinh^2$ steigen sehr rasch (mit wachsendem $|x|$), während die Differenz weiterhin genau 1 bleibt. Die entstehende Auslöschung wird automatisch durch höhere Stellenforderungen ausgeglichen. Wenn man $x = 10$ wählt, sind die Stellenzahlen bei den Funktionen 18 und 20, bei $x = 100$ bereits 96 und 98. Ab $x = 29$ wird nach dem ersten Durchgang sogar 0 in der Wurzel eingeschlossen; die Einschließung nach dem zweiten Durchgang wird dennoch genau genug, so dass kein dritter Durchgang notwendig wird.
- Es ist zu erkennen, dass es nicht unbedingt sinnvoll ist, bereits im ersten Durchgang mit einer festen Zahl von zusätzlichen Ziffern zu arbeiten, wenn die meisten Zwischenergebnisse später doch mit höherer Genauigkeit berechnet werden müssen. Die engeren Einschließungen bringen keine Vorteile für die Stellenabschätzungen des zweiten Schritts. Wenn überhaupt, dann sollte diese Schutzziffer-Heuristik von der jeweiligen Funktion abhängig gemacht werden und auch grob die Größe des Arguments berücksichtigen. In vielen Fällen erscheint es sogar sinnvoller, im ersten Schritt *weniger* Ziffern zu verwenden als endgültig gewünscht. (Man vergleiche hierzu an diesem Beispiel die Zeitmessungen im Anhang.)

  In der Implementation kann der Benutzer unterschiedliche Stellenzahlen für die beiden Durchgänge angeben; wenn nur eine angegeben wird, werden beide Schritte mit dieser durchgeführt.
- Man beachte weiterhin, dass wir durch die Mehrfachverwendung von x *keinen Baum* erzeugt haben; es gibt einen (ungerichteten) Zyklus. Da x als Konstante ohnehin als exakt (intern mit Genauigkeit *maxprecision*) angenommen wird, können sich keine Probleme ergeben. Wenn ganze Teilausdrücke mehrfach vorkommen, muss sichergestellt werden, dass sie, wenn sie zum zweiten Mal angelaufen

werden, nicht mit einer niedrigeren Genauigkeit erneut berechnet werden. Das wird in der Implementation sichergestellt (siehe dazu den nächsten Abschnitt).

2. Die Frage aus [Steins], Seite 25, lässt sich mit unserer Arithmetik automatisch, bei beliebiger Vorab-Wahl der Stellenzahl beantworten.

   Es soll bestimmt werden, ob $e^{\pi\sqrt{163}}$ nur sehr nahe bei einer ganzen Zahl liegt oder tatsächlich eine ganze Zahl ist. Wir verwenden dafür unsere Funktion `floor` zur Rundung auf die nächstkleinere ganze Zahl und `sign` für das Vorzeichen:

   ```
   sign( exp(pi*sqrt(163)) - floor(exp(pi*sqrt(163))) )
   ```

   (Im Programm kann der doppelt vorkommende Teilausdruck natürlich als eigene Variable mehrfach verwendet werden und wird dann ggf. weniger oft ausgewertet).

   Der obige Ausdruck hat, bei beliebiger Wahl der Stellenzahl, den Wert $[1, 1]$, d.h. es liegt *nicht* exakt eine ganze Zahl vor. Genau dann, wenn man eine ganze Zahl hätte, wäre der Ausdruck 0 (das Argument von `sign` muss dafür exakt 0 sein, und das genau bei identischen Punktintervallen der Fall). Könnte keine genügende Genauigkeit erreicht werden, würde eine Exception geworfen. (Letzteres kann allerdings tatsächlich passieren, wenn der Benutzer die Maximalanzahl von Wiederholungen bei 1-stelliger Berechnung auf unter 7 einstellt ; bei 16-stelliger Rechnung reichen bei der momentanen Wiederberechnungsstrategie 4 Stück). Man beachte aber, dass sich die Arithmetik, gleichgültig wie viele Wiederholungen man einstellt, beim Feststellen der Ganzzahligkeit von Ausdrücken wie $\frac{1}{3} + \frac{2}{3}$ die Zähne ausbeißen wird!

   Ohne `floor` und `sign` müssen auch wir (genau wie die Arithmetik zu [Steins]) vorab mindestens 31 Stellen vorgeben, damit die resultierende Einschließung (von $e^{\pi\sqrt{163}}$) keine ganze Zahl enthält. (Das Ergebnis ist $[2.6253741264076874399999999999_{2}^{3}\texttt{e}{+}17]$.)

Weitere Beispiele (mit Realisierung in unserer Arithmetik) finden sich in Abschnitt 5.8.6, nachdem die Implementation vorgestellt wurde.

5.8 Implementation

5.8.1 Benutzersicht

Die Klasse, die nach außen arithmetische Ausdrücke repräsentiert, wurde `RealExpression` genannt (im Hinblick auf eine mögliche zukünftige komplexe Version), aus schreibtechnischen Gründen mit `typedef` als `REx` abgekürzt.

Für sie sind Konstruktoren aus allen sinnvollen arithmetischen Typen, alle Operatoren der Grundrechenarten und alle Standardfunktionen implementiert, so dass der Benutzer den Typ wie einen normalen arithmetischen Typ behandeln kann. Der Mechanismus zum internen Aufbau der Bäume und die ggf. ausgelösten Berechnungs- und Wiederberechnungsschritte sind für ihn völlig transparent. Es reicht, die vorhandenen Konstanten bzw. Variablen in einem Ausdruck explizit nach `RealExpression` zu wandeln, bei den vorangegangenen Beispielen etwa:

```
RealExpression X("-1.283891273");
RealExpression e1 = sqr(cosh(X)) - sqr(sinh(X));
RealExpression e2 = exp( REx::pi() * sqrt(REx(163L)) );
```

Wenn es keine anderweitigen Überladungen der Operatoren und Standardfunktionen auf String-Argumente gibt, kann man numerische Konstanten auch als String-Konstanten angeben:

```
RealExpression e2 = exp( REx::pi() * sqrt("163") );
```

Es ist aber zu beachten, dass diese dann *zur Laufzeit* gelesen und umgewandelt werden, so dass Integer-Argumente jedenfalls schneller sind.

Die Konstruktion der Ausdrücke ist völlig unabhängig von irgendwelchen Genauigkeitsforderungen; diese kommen immer erst bei der expliziten Anforderung einer numerischen Auswertung ins Spiel.

Bei der Erzeugung konkreter numerischer Werte unterscheidet sich die Klasse von den normalen Typen und vom Analogon `ADE` aus [Steins]. `ADE` kann nicht in andere arithmetische Typen umgewandelt werden, die dann im weiteren Programm verwendet werden könnten. Die einzige Möglichkeit, Werte zu erzeugen, ist über eine Ausgabe. Dabei wird dann der Ausgabe-Operator des aktuell eingestellten Arithmetikbereichs aufgerufen.

Bei uns erzeugt die *Ausgabe* des Ausdrucks keinen Wert, sondern eine Symbolkette der verwendeten Formel (provisorisch mit vielen Sicherheits-Klammern):

```
cout << e1 << endl << e2 << endl;
```

liefert die Ausgabe

```
(((cosh(-1.283891273))^2)-((sinh(-1.283891273))^2))
exp(((pi)*(sqrt(163))))
```

Die Berechnung numerischer Werte muss explizit durch Aufruf einer entsprechenden Methode oder durch Umwandlung (typecast) in einen arithmetischen Typ wie `BFInterval`, `BigFloat` oder `double` angestoßen werden.

- Wenn *nur der erste Pass* durchgeführt werden soll – der eine garantierte Einschließung berechnet, die aber nicht notwendigerweise hoch genau ist –, verwendet man die Methoden `BFInterval_approx`, `BigFloat_approx`, `double_approx`.
- Wenn *beide Passes* durchgeführt werden sollen, wenn also eine garantierte hoch genaue Einschließung des Werts des Gesamtausdrucks berechnet werden soll, verwendet man `BFInterval_value`, `BigFloat_value`, `double_value`.

Im letzten Fall wird der erste Durchgang natürlich nur dann durchgeführt, wenn der Ausdruck nicht bereits (durch eine frühere Anweisung) schon einmal durchgerechnet wurde (gleichgültig mit welcher Stellenzahl).

Alle genannten Methoden können als Parameter die gewünschte Stellenzahl übergeben bekommen. Fehlt diese Angabe, wird eine global mit der Methode `RealExpression::set_precision(long)` festlegbare Stellenzahl verwendet. Bei einem Typecast in `double` wird immer mit 16 Stellen gerechnet.

```
RealExpression::set_precision(20);
cout.precision(10);

cout << "double     (10 approx)    : " << e1.double_approx(10) << endl;
cout << "BFInterval (20 approx)    : " << e1.BFInterval_approx() << endl;
cout << "BFInterval (20 guaranteed): " << e1.BFInterval_value() << endl;
cout << "BigFloat   (20 guaranteed): " << e1.BigFloat_value() << endl;
cout << "double     (16 guaranteed): " << (double)e1 << endl;
```

Die Anweisung `cout.precision` hat nichts mit unserer Arithmetik zu tun; sie legt nur die Stellenzahl für die *Ausgabe* bei `double`-Zahlen fest (standardmäßig wären nur 6 Stellen sichtbar).

Die Ausgabe ist:

```
double     (10 approx)    : 0.9999999975
BFInterval (20 approx)    : [ 0.99999999999999999950 , 1.00000000000000000004 ]
BFInterval (20 guaranteed): [ 0.99999999999999999999 , 1.00000000000000000001 ]
BigFloat   (20 guaranteed): 1
double     (16 guaranteed): 1
```

Bei `double` und `BigFloat` wird hierbei immer zur nächsten Zahl gerundet; wenn man Ober- oder Unterschranken haben möchte, kann man aber den Umweg über `BFInterval` gehen. Es ist zu beachten, dass, falls mit `allow_intervals` Intervall-Konstanten erlaubt werden, ausschließlich `BFInterval`-Werte erzeugt werden sollten, da bei Wandlungen nach `BigFloat` oder `double` ggf. Exceptions geworfen werden, wenn das intern berechnete Intervall nicht eng genug ist!

Für `BigFloat` und `BFInterval` wurden keine Umwandlungen als Typecasts implementiert, da es sonst leider zu mehrdeutigen Überladungen käme. Diese wären nur durch Änderungen in diesen Klassen aufhebbar (Konstruktoren von `RealExpression` oder `explicit`-Deklaration). Diese sollen aber vom Auswertungsalgorithmus, also vom Typ `RealExpression` unabhängig bleiben.

Bei `BFInterval_value` können *zwei* Stellenzahlen angegeben werden: eine für den ersten und eine für den zweiten Berechnungsdurchgang. Es wird natürlich immer Genauigkeit mit der zweiten Stellenzahl garantiert; die erste Angabe kann aber die Gesamtgeschwindigkeit beeinflussen.

```
cout << "BFInterval (4/20): " << e1.BFInterval_value(4,20) << endl;
```

(Wenn der Ausdruck bereits einmal durchgerechnet wurde, wird natürlich kein erster Pass durchgeführt, d.h. die erste Angabe wird ignoriert.)

Die Version zu [Steins] liefert für den obigen Ausdruck (möglicherweise aus den bei der Spezifikation, Seite 176, beschriebenen Gründen) meist kein hoch genaues Ergebnis, etwa $[0.999999998, 1.00000001]$ bei 9 Stellen, in $x = 2$ und bei Verwendung der Multiplikation für das Quadrat, bzw. $[0.999999995, 1.00000001]$ bei Verwendung von `pow` (es ist dort keine Funktion `sqr` definiert).

Beispiel: Das Beispiel aus [Steins], Seite 95, wiederum zur Einschließung von $e^{\pi\sqrt{163}}$, zeigt, dass die Steins'sche Arithmetik von der Genauigkeit abhängt, die zum Zeitpunkt der Erzeugung der Ausdrücke eingestellt ist (da mit ihr der erste Durchgang durchgeführt wird).

```
SetActualDomain(ACCURATE);
precision(32);
ADE expr=exp(4*atan("1")*sqrt("163"));
precision(31);
cout << expr << endl;
```

Wenn vor der Konstruktion des Ausdrucks 31 Stellen festgelegt werden, erhält man die Einschließung

$$[262537412640768743.999999999999_1^3].$$

Wenn vorher 32 Stellen festgelegt werden, vor der eigentlichen Auswertung dann aber wieder nur 31, erhält man dagegen enger

$$[262537412640768743.999999999999_2^3].$$

Wenn nicht bereits eine Auswertung stattgefunden hat, führt unsere Arithmetik den ersten Berechnungsvorgang erst bei der angeforderten Auswertung durch, d.h. sie rechnet bei den folgenden Zeilen in beiden Durchgängen mit nur 31 Stellen (und liefert interessanterweise dennoch die engere Einschließung):

```
REx e1=exp( REx::pi() * sqrt("163") );
cout << "BFInterval(31) : " << e1.BFInterval_value(31) << endl;
```

Aus den in der Bemerkung auf Seite 10 erläuterten Gründen sind nur dann `BFInterval`-Werte (die keine Punktintervalle darstellen) innerhalb der Ausdrücke erlaubt, wenn bei der Compilation der Arithmetik in der Konfigurationsdatei `XArithConfig.h` der Wert `allow_intervals` als ungleich 0 definiert wurde. Ansonsten wirft der Versuch, ein solches Intervall zu verwenden, eine Exception.

5.8.2 Interne Klassen

Es sind zwei Oberklassen am Aufbau der Ausdrücke beteiligt.

- Die abstrakte Klasse `RealNode` stellt einen Knoten im Binärbaum dar; sie realisiert die allgemeinsten Daten-Member (z.B. den Wert in `BFInterval`-Darstellung) und sehr allgemeine Methoden. Unterklassen von ihr implementieren die speziellen Funktionalitäten der einzelnen Binär-Operatoren und Standard-Funktionen (insbesondere die entsprechenden Versionen von *compute* und *recompute*).

- `RealExpression` ist zum einen die Schnittstelle nach außen, und es macht Sinn, dort Methoden anzusiedeln, die nicht in jedem Knoten zur Verfügung zu stehen brauchen. Außerdem muss die Wurzel eines Ausdrucks ohnehin gesondert behandelt werden: Im Fall, dass durch den ersten Wiederberechnungsschritt 0 aus der Einschließung in der Wurzel ausgeschlossen wurde, muss ja insgesamt ein weiterer Wiederberechnungsschritt durchgeführt werden.

Jedes `RealExpression`-Objekt besitzt (als einzigen Daten-Member) einen Pointer auf die Wurzel des Binärbaums, beschrieben durch ein `RealNode`-Objekt. Ausdrücke werden vom Benutzer über `RealExpression`-Objekte zusammengesetzt. Intern werden dabei aber die

`RealNode`-Objekte zusammengehängt, so dass keine überflüssigen Zweifach-Indirektionen entstehen. Temporär entstandene `RealExpression`-Objekte werden vom Compiler automatisch entsorgt.

- Es ist eine weitere Design-Entscheidung, dass Ausdrücke ineinander eingesetzt werden können sollen, ohne dass der neue Teilausdruck *als Kopie* angelegt wird. Es ist also denkbar, dass ein Teilausdruck mehrfach innerhalb eines Ausdrucks oder in mehreren Ausdrücken erscheint.

Für die Verwaltung hiervon kommt wieder ein ähnlicher Mechanismus wie bei den BCD-Strings in `BigInt` zum Einsatz. In jedem Knoten wird ein Zähler mitgespeichert, der Buch darüber führt, wie oft er momentan noch in Gebrauch ist. Wenn der Zähler wieder 0 erreicht, kann der Knoten und der daran hängende Teilbaum (vom zugreifenden Objekt) entsorgt werden.

Nun wird ein Knoten eventuell von unterschiedlichen Ausdrücken aus mit ganz unterschiedlichen Stellenforderungen angelaufen. Es wird daher zusätzlich die zuletzt garantierte Genauigkeit mitgespeichert. Falls eine Neuberechnung mit einer kleineren Stellenzahl angefordert wird, wird lediglich der schon gespeicherte Wert entsprechend gerundet zurückgeliefert.

Diese Genauigkeits-Markierung darf zwei spezielle Werte annehmen. Noch nie betretene Knoten werden (beim Anlegen im Konstruktor) mit -1 markiert. In der ersten Phase werden alle berechneten Knoten (die ja aber noch nicht die entsprechende Gesamtgenauigkeit gewährleisten können) durch den Wert 0 gekennzeichnet. Erst wenn eine Wiederberechnung mit einer Genauigkeit $p \geq 1$ stattgefunden hat, erhält die Markierung diesen Wert p.

Knoten, die mit exakten Werten initialisiert werden (also von einer `BigFloat`- bzw. `BFInterval`-Variable oder -Konstante aus), erhalten von vornherein die Markierung *maxprecision*, wobei dieser Wert größer als alle zulässigen Genauigkeiten gewählt ist.

Es sei bereits erwähnt, dass die Markierungen in einem speziellen Zusammenhang (bei Ausdrücken, die aus einem String geparst wurden, in dem die symbolischen Variablen a bis z auftreten) wieder auf -1 zurückgesetzt werden können.

5.8.3 `RealNode`

Die Klasse `RealNode` ist abstrakt, sie definiert jedoch bereits einige allgemeine Datenmember und Methoden. Da beispielsweise die Methoden *compute* und *recompute* rein virtuell sind, können von ihr aber keine Objekte angelegt werden. Die Datenmember sind folgende:

```
class RealNode
 {
   protected:
      long precision;
      BFInterval value;
      long refCount;
   ...
 };
```

`precision` ist die bereits besprochene Markierung (-1, 0 oder die bereits garantierte Genauigkeit $p \geq 1$). `value` ist der in vorangegangenen Schritten mit *compute* oder *recompute* bereits berechnete oder bei Konstanten bei der Konstruktion eingetragene Wert. Er ist nicht gültig (Wert 0), falls der Knoten noch mit -1 markiert ist. `refCount` ist der Zähler, an dem abzulesen ist, wie oft dieser Knoten momentan in Gebrauch ist.

Allgemeine Methoden in `RealNode` sind `exclude_zero` (Ausschluss von 0), `exclude(BigFloat)` (Ausschluss eines beliebigen Werts) und `exclude_mulpi4` (Ausschluss von Vielfachen von $\frac{\pi}{4}$), die bei Problemen mit Definitionsbereichen oder zum ausdrücklich gewünschten Ausschluss von 0 verwendet werden.

Von `RealNode` wird eine Vielzahl von Unterklassen abgeleitet:

- Die Blätter des Baums bestehen aus Knoten, deren Werte von Konstanten der zugelassenen arithmetischen Typen festgelegt werden. Die Knotenklassen heißen entsprechend `BigFloatAtom`, `BFIntervalAtom`, `BigRationalAtom` und `DoubleAtom`. Dabei werden die ersten beiden immer als bereits maximal genau markiert, während die beiden anderen ja nach aktueller Genauigkeitsforderung neu in ein `BFInterval` eingeschlossen werden müssen.

- Zusätzlich als Blätter gibt es die beiden Knotenklassen `PiAtom` und `EAtom`, die die Konstanten π und e repräsentieren. Sie rufen zur Berechnung jeweils die beiden entsprechenden Methoden aus `BFInterval` auf. Da die jeweils aktuellen Näherungen bereits innerhalb dieser Methoden fest verwaltet werden, brauchen diese beiden Klassen keine Kopie als Datenmember.

- Außerdem gibt es noch die Klasse `VariableAtom` für die symbolischen Konstanten der Parser-Komponente. Sie wird in Abschnitt 5.8.5 beschrieben.

- Für die einstelligen Operatoren (unäres Plus und Minus, sowie die Standardfunktionen mit einem Argument) wurde die abstrakte Klasse `RN_oneop` implementiert, die die Behandlung eines Nachfolgeknotens implementiert (Ansprechen des Zählers und ggf. Löschen, etc.). Von ihr abgeleitet sind als konkrete Klassen (mit sprechenden Namen):

  `RN_abs`, `RN_acos`, `RN_acosh`, `RN_acot`, `RN_acoth`, `RN_asin`, `RN_asinh`, `RN_atan`, `RN_atanh`, `RN_ceil`, `RN_cos`, `RN_cosh`, `RN_cot`, `RN_coth`, `RN_exp`, `RN_floor`, `RN_log10`, `RN_log`, `RN_minus`, `RN_plus`, `RN_root`, `RN_round`, `RN_sign`, `RN_sin`, `RN_sinh`, `RN_sqr`, `RN_sqrt`, `RN_tan`, `RN_tanh`.

  Die Unterklassen implementieren *compute* und *recompute* wie in den vorangegangenen Abschnitten beschrieben, sowie die Ausgaberoutine `print` (beispielsweise gibt sich `RN_sqr` als `x^2` aus und nicht in Funktionsschreibweise, etc.). `RN_root` für die n-te Wurzel hat als zusätzlichen Datenmember n.

- Die zweistelligen Operatoren realisiert die abstrakte Klasse `RN_twoop`, die analog zwei Nachfolgeknoten verwaltet. Sie hat als Datenmember das Ausgabezeichen für den Operator ('`+`', '`^`', etc.) und implementiert eine Ausgaberoutine, sicherheitshalber mit Klammern um die Argumente, also „(x)+(y)“ etc. Eine verfeinerte Ausgaberoutine, die

sich jeweils den Typ ihrer Argumente anschaut und Klammerungen vermeidet, wäre natürlich denkbar. Von `RN_twoop` sind folgende konkreten Klassen abgeleitet:

`RN_add`, `RN_sub`, `RN_mul`, `RN_div`, `RN_pow`.

5.8.4 `RealExpression`

`RealExpression` stellt die meisten Operatoren eines typischen arithmetischen Typs zur Verfügung. Ausgelassen wurden z.B. die Prä- und Post-In- und -Dekrement-Operatoren `++` und `--`, die in diesem Zusammenhang eher für Verwirrung sorgen könnten.

Von arithmetischen Grundtypen aus gibt es Konstruktoren von `BigFloat`, `BFInterval`, `BigRational`, `double` und `long`, wobei ein jeweils passendes Atom erzeugt als Wurzel erzeugt wird.

Ein String als Konstruktor-Argument wird immer als Konstante eines der einfachen arithmetischen Typen, nicht als Formeldarstellung, angesehen. Wenn der String, abgesehen von Leerräumen, mit einer eckigen Klammer beginnt, wird ein `BFInterval`-Atom erzeugt (das aber nur gültig ist, wenn `allow_intervals` definiert wurde oder es sich um ein Punktintervall handelt). Ansonsten wird ein `BigFloat`-Atom erzeugt, es sei denn, es folgt der ersten Zahl ein "`/`", was ein `BigRational`-Atom erzwingt. Links und rechts dieses Bruchzeichens dürfen durchaus Dezimalbrüche stehen, die dann passend erweitert werden; „1/1.2“ liefert also $\frac{5}{6}$. (Letzteres liegt darin begründet, dass zu Beginn des Strings noch nicht zwischen einer Fließkommazahl und einer rationalen Zahl unterschieden werden kann und daher ohnehin die Fließkomma-Leseroutine aufgesetzt wird.) Falls die Zeichenkette dagegen den Ausdruck symbolisch als mathematische Formel darstellen soll, muss die Methode `parse` verwendet werden, die im nächsten Abschnitt beschrieben wird.

`RealExpression` verwaltet außerdem die Einstellungen, wie oft bei Problemen mit überschrittenen Definitionsbereichen oder auch einfach bei echt eingeschlossener Null Verfeinerungsversuche stattfinden sollen.

- Der Benutzer kann die Anzahl mit `set_max_retry(long)` einstellen und mit `get_max_retry` auslesen.
- Mit der Methode `set_retry_if_zero_included` wird festgelegt, wann eine echt eingeschlossenen Null ausgeschlossen werden soll. Die zulässigen Parameter sind `RealExpression::never` (nie), `always` (immer, also in allen Knoten des Baums) und `result` (nur in der Wurzel des Baums). Die Voreinstellung ist `never`. Analoges gilt für Bereichsüberschreitungen und die Funktion `set_retry_if_domain_error`.

5.8.5 `RealExpressionParser`

Die bisher besprochenen Mechanismen erlauben den Zusammenbau von Ausdrücken, deren Aufbau im Wesentlichen zur Compilationszeit feststeht. Daher wurde zusätzlich eine Klasse realisiert, die eine Formel als (ASCII-)String oder aus einer Datei erhält, geeignet interpretiert (parst), den internen Baum aufbaut und ein entsprechendes `RealExpression`-Objekt erzeugt.

Die Klasse ist nicht für den Benutzer gedacht. Ihm stehen vielmehr zwei Methoden aus `RealExpression` zur Verfügung, `parse(const char *)` und `parse(istream &)`.

Die String-Repräsentationen entsprechen denen gängiger Programmiersprachen. Insbesondere muss immer ein Stern `*` als Multiplikationszeichen gesetzt werden. Zusätzlich gibt es allerdings den (rechtsassoziativen) Potenz-Operator `^`.

Intern wird ein einfacher Recursive-Descent-Parser verwendet. Die Grammatik entspricht im Wesentlichen der zum Parser mit Parsebaum-Aufbau aus [Rogat] (Seite 111ff und Anhang A, Seiten 221–233), wobei auf Grammatikebene zusätzlich der Potenzoperator eingefügt wurde. Es wird aber natürlich *kein* „Constant Folding“ betrieben wie dort, also Zusammenfassen von Teilausdrücken, die nur Konstanten enthalten, zu einer Konstanten. Schließlich müssen auch diese Teilausdrücke später mit beliebiger noch festzulegender Genauigkeit berechenbar sein.

Natürlich wäre ein so geparster Ausdruck vergleichsweise wertlos, wenn nur Konstanten als Atome erlaubt wären. Daher sind als symbolische Variable `a` bis `z` (Groß- oder Kleinbuchstaben) bzw. gleichbedeutend `x0` bis `x25` erlaubt. Für diesen Zweck verwaltet `RealExpression` intern ein Array von 26 speziellen `RealNode`-Objekten der Unterklasse `VariableAtom`. Objekte dieser Klasse speichern nur die Nummer der Variablen und geben sich immer in der Kleinbuchstabenform aus. Die Variablen haben den Initialisierungswert 0. Zum Setzen bzw. Lesen dienen die `RealExpression`-Methoden `set_variable` bzw. `BigFloat get_variable` (über Nummer oder Namen).

Es sei betont, dass diese Variablen „klassenglobal“ sind. Es wurde entschieden, nicht jedem Ausdruck einen eigenen Satz Variabler zu geben, da dies erfordern würde, jeden Knoten mit der Wurzel (oder einer passenden Hilfsstruktur) rückzuverketten, was die Größe jedes Objekts und den Verwaltungsaufbau stark erhöhen würde. Außerdem müssten beim Einsetzen von Ausdrücken ineinander diejenigen Knoten *kopiert* werden, die momentan mehrfach verwendet werden können. In Hinblick auf normale Anwendungsgebiete wäre dieser Mechanismus aber auch stark übertrieben. Wenn nur einfach *mehr* Variable notwendig sind, kann das zur Compilationszeit durch Ändern einer Konstanten leicht geändert werden (die Variablen oberhalb der Nummer 26 sind dann aber nicht über einzelne Buchstaben ansprechbar, nur als „`x26`“ etc.).

Die Funktionen `parse` geben ein Bitmuster zurück, das Auskunft darüber gibt, welche Variablen im String angesprochen wurden (also 0 bei einem konstanten Ausdruck, Bit 24 gesetzt, wenn `x` angesprochen wurde, etc.).

Zu beachten ist, dass, wenn sich der Wert einer solchen Variable geändert hat, der Ausdruck beim nächsten Durchgang vollständig neu berechnet werden muss. Daher muss der Benutzer anschließend (ggf. nach Verändern *mehrerer* Variabler) die Methode `reset` auf den Ausdruck anwenden, der die Markierungen in allen nicht exakten Knoten auf -1 zurücksetzt. Das Ergebnis von weiteren Auswertungen ist nicht definiert, falls dieser Aufruf fehlt!

In den Strings sind zusätzlich die Konstanten π („`pi`“) und e („`eu`“, zur Unterscheidung von der Variablen „`e`“) erlaubt.

Beispiel: Wir lesen eine Funktion in einer Variablen x in einen String ein und werten sie danach in mehreren Punkten aus. (Wenn in der eingegebenen Formel andere Variable vor-

kommen, haben sie für die Auswertung den Wert 0.) Wenn nichts als Wert angegeben wird (nur Return-Taste), kann eine neue Formel eingegeben werden. Wenn nichts als Formel angegeben wird, wird das Programm beendet. Der Einfachheit halber arbeitet dieses Programm immer mit einer Genauigkeit von 32 Dezimalstellen.

```
RealExpression e;
RealExpression::set_precision(32);

for (;;)
 {
   char s[1024];
   cout << "f(x)=";
   cin.getline(s,1024,'\n');
   if (s[0]==0) break;

   e.parse(s);
   cout << "f(x)=" << e << endl;

   for (;;)
    {
      cout << "x=";
      cin.getline(s,1024,'\n');
      if (s[0]==0) break;

      BigFloat x;
      x.from_string(s);
      RealExpression::set_variable('x'-'a',x);

      e.reset();
      BFInterval val=e.BFInterval_value();
      cout << "f(" << x << ") in " << val << endl;
    }
 }
```

Es wäre sicher denkbar, dass bei Verändern einer Variablen automatisch alle Ausdrücke, die sie enthalten, als nicht ausgewertet markiert werden. Dazu müsste allerdings erstens in den Knoten des Typs `VariableAtom` eine Liste aller momentanen Vater-Knoten geführt werden, und zweitens müsste in allen Knoten eine zusätzliche Rückverkettung nach oben stattfinden.

Der Vorteil wäre, dass das Benutzerprogramm den Aufruf von `reset` nicht „vergessen“ kann, und dass ggf. Teilausdrücke, die von der Änderung nicht betroffen sind, nicht neu ausgewertet werden müssen. Letzteres betrifft aber dann höchstens Äste, in denen nur Konstante vorkommen, so dass normalerweise nicht viel Zeit damit gespart werden dürfte. Dagegen würde der zusätzliche Aufwand (die Rückverkettung) den „Normalbetrieb“ (ohne Parser) unnötig verlangsamen, so dass auf diesen Mechanismus in der momentanen Implementation verzichtet wurde.

5.8.6 Weitere Beispiele

Beispiel 1: In [HaHoKuRa], der „C++ Toolbox for Verified Computing“, Kapitel 8, wird ein Auswertungsalgorithmus analog zu [Fischer] unter Zuhilfenahme einer eigenen Staggered-Correction-Arithmetik (nur für Grundrechenarten und Potenzierung) vorgestellt. Als erstes

Beispiel wird die rationale Funktion

$$f(x,y) = \frac{1}{y^6 - 3xy^5 + 5x^3y^3 - 3x^5y - x^6}$$

jeweils in zwei aufeinander folgenden Fibonacci-Zahlen a_i, a_{i+1} ausgewertet ($a_0 := 0$, $a_1 := 1$, $a_i := a_{i-1} + a_{i-2}$ für $i \geq 2$); das exakte Ergebnis ist $(-1)^i$.

Die normale IEEE-Arithmetik liefert für das in der Toolbox ausgewählte Paar $(a_{66}, a_{67}) = (27777890035288, 44945570212853)$ einen Phantasiewert von ca. $2.374 \cdot 10^{-66}$. Für große Argumente ist die Funktion mit normaler Intervall-Arithmetik nicht mehr auswertbar, da das Nenner-Intervall 0 enthält – bei 16-stelliger Arithmetik ab $(a_{15}, a_{16}) = (610, 987)$. Für (a_{66}, a_{67}) erhalten wir für den Nenner die Einschließung $[-2.5 \cdot 10^{61}, +1.2 \cdot 10^{61}]$.

Für die direkte Verwendung unseres Wiederberechnungs-Algorithmus ist es hier essentiell, dass die automatische Wiederberechnung von Operanden bei Bereichsüberschreitungen eingeschaltet bleibt, da sonst nach dem ersten Durchgang wegen „Division durch 0“ abgebrochen würde!

Bei (a_{66}, a_{67}) mit (in Anlehnung an die Toolbox) 21 Stellen erhalten wir im ersten Durchgang für den Nenner die oben angegebene Einschließung, die 0 enthält. Dadurch wird bereits jetzt der Wiederberechnungsschritt für den Nenner mit 21 Stellen eingeleitet, der für ihn $[1, 1]$ berechnet – exakt, da bis zur achten Potenz per Multiplikation potenziert wird, ohne Verwendung von exp und log. Da sich nun auch für den Quotienten ein Punktintervall ergibt, wird der reguläre Wiederberechnungsschritt gar nicht mehr ausgelöst. Die insgesamt höchste zwischendurch vorkommende Genauigkeitsforderung ist 120 Stellen bei y^5 und beim Argument xy von $(xy)^3$, also fast das sechsfache der eigentlichen Stellenforderung. Das Verfahren in der Toolbox benötigt auch gerade 6 Schritte, nämlich einen Erstberechnungsschritt und 5 Korrekturschritte.

Wenn der Quotient verwendet wird, bricht die Arithmetik zu [Steins] das Programm erwartungsgemäß mit „`BIGFLOATINT: denominator interval including zero`“ ab. Wenn nur der Nenner mit 16 Stellen eingeschlossen werden soll, erhält man „`[-7482e58,9735e58]`“ als Ergebnis. Erst wenn man mindestens 83 (!) Stellen fordert, erhält man ein Intervall, das nicht 0 enthält, bei 83 Stellen $[0.2186, 1.9453]$, bei 90 Stellen $[0.99999990164, 1.00000007441]$. Die seltsamen Resultate sind vermutlich auf den in Abschnitt 5.2 beschriebenen Effekt zurückzuführen.

Folgendes Programmstück berechnet den Quotienten in einer Endlosschleife aufsteigend für alle Paare aufeinander folgender Fibonacci-Zahlen:

```
RealExpression e;
e.parse("1/(y^6-3*x*y^5+5*(x*y)^3-3*x^5*y-x^6)");

BigFloat x("0");
BigFloat y("1");

for (;;)
 {
   RealExpression::set_variable(x,'x');
   RealExpression::set_variable(y,'y');
   e.reset();
```

```
    cout << x << "  " << y << "  " << e.BFInterval_value(16) << endl;
    BigFloat z=x+y; x=y; y=z;
}
```

Die Ausgabe ist:

```
0  1  [ 1 , 1 ]
1  1  [ -1 , -1 ]
1  2  [ 1 , 1 ]
2  3  [ -1 , -1 ]
3  5  [ 1 , 1 ]
5  8  [ -1 , -1 ]
...
2.7777890035288e+13  4.4945570212853e+13  [ 1 , 1 ]
...
1.038810421957299147085105183827754016801420367758 41e+50
  1.6808305705945300883541229581164851348244958539 9521e+50  [ -1 , -1 ]
```

Beispiel 2: Direkt anschließend in der Toolbox wird ein Operator zur näherungsweisen Bestimmung der zweiten Ableitung reeller, zweimal differenzierbarer Funktionen über den Differenzenquotienten

$$D_f(x,h) = \frac{f(x-h) - 2f(x) + f(x+h)}{h^2}, \quad h \neq 0,$$

betrachtet. Normalerweise kann er wegen Auslöschungseffekten im Zähler nicht verwendet werden (für sehr kleine h, außer ggf. mit ergänzender Extrapolation). Die Auswertungsalgorithmen gleichen die Auslöschung aber automatisch durch höhere Stellenzahlen aus.

Wie in der Toolbox wenden wir den Operator zunächst auf die Funktion

$$f(x) = 540 \cdot \frac{x^4 - 23x^3 + 159x^2 + 2x + 45}{x^3 + 18x^2 + 501x + 20}$$

an und werten aus im Punkt $x = 1$; der exakte Wert ist $f''(x) = 36$.

Das Ergebnis lässt sich hier natürlich exakt mit `BigRational` berechnen; für $h = 10^{-8}$ erhält man einen 56/54-stelligen Bruch, in Dezimalbruchdarstellung 36.000000000000008052... Die punktweise IEEE-Arithmetik liefert unterschiedliche Werte, je nachdem, wie der Ausdruck geschrieben wird: Bei Verwendung von `pow` erhält man etwa 49.94, bei Ersetzen von `pow` durch Multiplikationen 26.78! Intervall-Arithmetik fest mit 16 Stellen berechnet $[-2000, +2000]$, mit 20 Stellen $[35.8, 36.2]$.

Folgendes Codestück (der Einfachheit halber mit festen Werten) übernimmt die obige Aufgabe und garantiert hoch genauen Einschluss:

```
RealExpression f(const RealExpression &x)
 {
   return 540 * ( pow(x,4) - 23*pow(x,3) + 159*sqr(x) - 2*x + 45 )
              / ( pow(x,3) + 18*sqr(x) + 501*x + 20 );
 }
```

```
RealExpression D2( RealExpression (*f)(const RealExpression &x),
   const BigFloat &x, const BigFloat &h)
 {
   RealExpression X(x);
   return ( f(X-h) - 2*f(X) + f(X+h) ) / sqr(h);
 }

int main()
 {
   BigFloat x("1");
   BigFloat h("1e-8");
   cout << D2(f,x,h).BFInterval_value(16) << endl;
 }
```

Die Ausgabe ist $[36, 36.00000000000001]$, wobei die intern höchste benötigte Stellenzahl 54 ist (bei der Potenzierung in $(x-h)^3$). Die in der Toolbox berechnete Einschließung hat in der Obergrenze die letzte Ziffer 2, braucht aber nur eine Nachkorrektur. Die Steins'sche Arithmetik liefert bei 16 Stellen $[-30, 100]$, bei 17 Stellen $[29, 43]$ und bei 18 Stellen $[35.3, 36.7]$.

Die Toolbox-Beispiele können wegen der Einschränkung der Staggered-Correction-Arithmetik keine Standardfunktionen enthalten. Wir testen noch zusätzlich mit $f(x) = \sin(x)$. Wenn man $D_f(x,h)/f(x)$ berechnen lässt ($f''(x)/f(x) = -1$, falls der Nenner nicht verschwindet), liefert die Arithmetik zu [Steins] bei $x = 1$ und $h = 10^{-8}$ das Intervall $[-0.92, -0.8]$, bzw. bei $x = 0.5$ das Intervall $[-0.5, -0.46]$, also keine korrekten Einschließungen. Mit punktweiser IEEE-Rechnung erhält man -1.11. Unsere Arithmetik erzeugt mit der Zeile

```
cout << ( D2(f,x,h)/f(x) ).BFInterval_value(16) << endl;
```

(bei praktisch gleichgültigem Argument $x \neq 0$) die Ausgabe $[-1, -0.9999999999999999]$. Bei $x = 1$ ist dabei die höchste auftretende Stellenzahl 42 (bei der Auswertung von $\sin(1)$); bei $x \approx 20000\,\pi$ (auf 50 Stellen genau) ist die Höchstforderung 93 Stellen (bei der Subtraktion in D_f).

6 Performance-Tests

Wir wollen noch anhand einiger Messreihen den Erfolg unserer Bemühungen, die Arithmetik auf allen Ebenen zu beschleunigen, dokumentieren und außerdem einige der getroffenen Design-Entscheidungen rechtfertigen.

Die Messungen, die unterschiedliche Arithmetiken vergleichen, wurden immer auf demselben Rechner, einem L*i*nux-PC (400 MHz Pentium-II-Prozessor, 256 MB Hauptspeicher) durchgeführt. Es wurde jeweils derselbe Compiler (GNU 2.95.2) mit denselben Einstellungen (z.B. Optimierungsstufe `-O3`) verwendet. Zu Beginn wird ein kurzer Vergleich mit dem anderen Testrechner, einer Sun UltraSparc 10 (UltraSparc IIi-CPU mit 333 MHz, 256 MB Hauptspeicher) angestellt.

Als Abkürzung für unsere Arithmetik (gegenüber „[Steins]“, bzw. „[St]“) wurde „`XAri`“ oder „[Ro]“ gewählt. Die Hardware-Arithmetik wird mit „IEEE“ bezeichnet.

6.1 Unterschiedliche Rechner, Basen, Arithmetiken

Vorab sollen nur kurz einige ausgewählte Laufzeiten auf den beiden Testrechnern gegenübergestellt werden.

Bei dem Sparc-Prozessor handelt es sich um eine 64-Bit-CPU, so dass hiermit auch sinnvoll eine BCD-Arithmetik zur Basis 100 000 000 übersetzbar ist. Der GNU-Compiler lässt in der verwendeten Version allerdings nicht die Erzeugung entsprechenden Codes zu. Der SUN-eigene Compiler hat leider größte Schwierigkeiten beim Überladen von Operatoren und mit Namespaces, so dass die endgültige Version mit ihm leider nur nach Einbau vieler Notlösungen übersetzbar gewesen wäre. Es wurden daher bislang keine entsprechenden ausführlichen Messreihen durchgeführt. Anfängliche Tests nur der Grundoperationen zeigten, dass Additionen tatsächlich um fast einen Faktor 2 gegenüber 32-Bit beschleunigen ließen; andere Operationen wurden dagegen langsamer. Wenn auf anderen Systemen zwischen 32 und 64 Bit entschieden werden soll, sollten also auf jeden Fall einige kurze Geschwindigkeitstests durchgeführt werden.

	XAri PC 32	[St] PC 32	XAri PC 64*	XAri SUN 32	[St] SUN 32
Float-Add. 16+16, $\Delta e = 0$	2.00 e-6	6.20 e-6	1.90 e-6	7.60 e-6	2.60 e-5
Float-Add. 16+16, $\Delta e = 8$	3.00 e-6	1.14 e-5	2.50 e-6	9.80 e-6	4.80 e-5
Float-Add. 16+16, $\Delta e = 16$	3.00 e-6	1.14 e-5	2.50 e-6	1.00 e-5	5.20 e-5
Float-Add. 32+32, $\Delta e = 0$	2.00 e-6	7.40 e-6	2.00 e-6	9.00 e-6	3.00 e-5
Float-Add. 32+32, $\Delta e = 16$	2.80 e-6	1.28 e-5	2.60 e-6	1.10 e-5	5.50 e-5
Float-Add. 32+32, $\Delta e = 32$	2.80 e-6	1.32 e-5	2.70 e-6	1.00 e-5	5.50 e-5
Float-Add. 64+64, $\Delta e = 0$	2.20 e-6	7.60 e-6	2.00 e-6	9.00 e-6	3.50 e-5
Float-Add. 64+64, $\Delta e = 32$	3.00 e-6	1.36 e-5	3.00 e-6	1.10 e-5	5.50 e-5
Float-Add. 64+64, $\Delta e = 64$	3.00 e-6	1.36 e-5	3.00 e-6	1.10 e-5	5.50 e-5
Float-Add. 128+128, $\Delta e = 0$	2.60 e-6	7.40 e-6	2.30 e-6	1.00 e-5	3.50 e-5
Float-Add. 128+128, $\Delta e = 64$	3.70 e-6	1.48 e-5	3.10 e-6	1.20 e-5	6.00 e-5
Float-Add. 128+128, $\Delta e = 128$	3.60 e-6	1.48 e-5	3.00 e-6	1.20 e-5	6.00 e-5

	XAri PC 32	[St] PC 32	XAri PC 64*	XAri SUN 32	[St] SUN 32
Int-Mult. 16*16	3.00 e-6	7.00 e-6	4.00 e-6	1.10 e-5	4.00 e-5
Int-Mult. 32*32	6.00 e-6	1.90 e-5	1.00 e-5	3.10 e-5	1.05 e-4
Int-Mult. 64*64	1.80 e-5	6.00 e-5	3.60 e-5	4.90 e-5	3.60 e-4
Int-Mult. 128*128	3.60 e-5	2.40 e-4	1.30 e-4	1.00 e-4	1.38 e-3
exp(2), 16-st.	4.90 e-4	1.50 e-3	8.00 e-4	1.60 e-3	6.80 e-3
exp(2), 32-st.	9.60 e-4	6.00 e-3	1.50 e-3	3.60 e-3	2.40 e-2
exp(2), 64-st.	2.78 e-3	2.68 e-2	4.40 e-3	1.08 e-2	9.84 e-2
exp(2), 128-st.	1.14 e-2	1.22 e-1	1.95 e-2	4.30 e-2	4.49 e-1
sqrt(2), 16-st.	1.80 e-4	1.60 e-3	2.00 e-4	5.00 e-4	6.05 e-3
sqrt(2), 32-st.	2.40 e-4	2.60 e-3	3.00 e-4	7.00 e-4	9.10 e-3
sqrt(2), 64-st.	4.00 e-4	5.20 e-3	6.00 e-4	1.50 e-3	1.60 e-2
sqrt(2), 128-st.	9.50 e-4	1.40 e-2	1.40 e-3	3.00 e-3	4.20 e-2
log(10), 16-st.	4.40 e-4	8.00 e-3	6.50 e-4	1.30 e-3	3.12 e-2
log(10), 32-st.	6.00 e-4	2.30 e-2	1.20 e-3	2.00 e-3	8.70 e-2
log(10), 64-st.	1.20 e-3	1.15 e-1	3.20 e-3	4.40 e-3	4.80 e-1
log(10), 128-st.	7.15 e-3	1.24 e+0	1.45 e-2	1.50 e-2	4.62 e+0
$\cosh^2(5)-\sinh^2(5)$, 16-st.	4.20 e-3	1.10 e-1	6.00 e-3	1.45 e-2	3.80 e-1
$\cosh^2(5)-\sinh^2(5)$, 32-st.	7.00 e-3	3.10 e-1	1.00 e-2	2.55 e-2	1.07 e+0
$\cosh^2(5)-\sinh^2(5)$, 64-st.	1.64 e-2	1.30 e+0	2.60 e-2	6.10 e-2	4.47 e+0
$\cosh^2(5)-\sinh^2(5)$, 128-st.	5.80 e-2	7.40 e+0	9.20 e-2	2.10 e-1	3.56 e+1
$\exp(\pi \cdot \sqrt{163})$, 16-st.	3.20 e-3	1.30 e-2	4.30 e-3	1.10 e-2	5.20 e-2
$\exp(\pi \cdot \sqrt{163})$, 32-st.	5.40 e-3	2.80 e-2	8.00 e-3	1.95 e-2	1.15 e-1
$\exp(\pi \cdot \sqrt{163})$, 64-st.	1.40 e-2	1.06 e-1	2.20 e-2	5.40 e-2	4.45 e-1
$\exp(\pi \cdot \sqrt{163})$, 128-st.	5.06 e-2	8.60 e-1	8.40 e-2	1.90 e-1	2.43 e+0

Zur Illustration wurden auf dem L*i*nux-PC auch Tests mit 64-Bit-Typen durchgeführt, die aber Compiler-intern auf 32-Bit-Typen und -Operationen zurückgeführt werden. Wie nicht anders zu erwarten, lohnt sich aber die Verwendung solcher emulierten 64-Bit-Typen nicht. Bei der Addition kann dabei sogar noch ein wenig gewonnen werden, da weniger Überträge auftreten. Bei der Multiplikation verringert sich deren Anzahl zwar auch, dafür steigt der Aufwand ihrer Behandlung deutlich (emulierte Division).

Bei den Messungen der Grundoperationen wurde jeweils über eine kleine Anzahl verschiedener Argumente gemittelt. Die Angaben sind jeweils in Sekunden für eine durchgeführte Operation.

Bei den direkt vergleichbaren Versionen L*i*nux-PC und SUN, 32 Bit mit GNU-Compiler beträgt der Faktor ohne besondere Abweichungen durchschnittlich 3.5. Bei der Steins'schen Arithmetik ergeben sich unterschiedliche Faktoren: etwa 4.6 bei den Grundoperationen und 3.75 bei Standardfunktionen und arithmetischen Ausdrücken.

6.2 Integer-Multiplikation

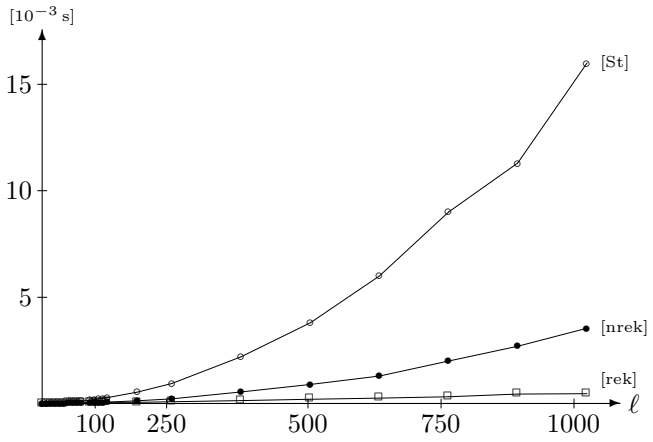
Wie in Abschnitt 2.4.4 bereits erwähnt, lohnt es sich erst bei relativ großen dezimalen Stellenzahlen, die rekursive Variante der Multiplikation zu verwenden. Gelegentlich treten aber

auch bei gemäßigten Stellenforderungen für die arithmetischen Ausdrücke bei Zwischenergebnissen oder benötigten Konstanten wesentlich höhere Längen auf, so dass sich die Rekursion früher positiv bemerkbar macht als gedacht.

Es sind Zeiten für die Stackverwaltung bei den rekursiven Unterprogrammaufrufen und ggf. für einen Aufruf zur Freigabe und zum Neuanlegen von Temporärspeicher beteiligt. Daher müssen die Konstanten auf jedem System durch Messungen neu bestimmt werden, die festlegen, ab wann die Rekursion angestoßen und wie tief sie hinabsteigt. Verbesserungen in den aufgerufenen Routinen (z.B. Addition und Subtraktion) können diese Werte immer wieder beeinflussen.

In der aktuellen Version auf dem L*i*nux-Rechner wird erst ab 48 Stellen diese Variante verwendet und weiter abgestiegen, wenn die Operanden mindestens 16 Stellen lang sind. Bei etwa 100 Stellen ergibt sich ein Faktor von nur 1.2, bei 1000 Stellen von 13.3.

Im Diagramm sind die rekursive und die nicht rekursive Variante, sowie die Version aus [Steins] für eine Multiplikation zweier unterschiedlicher Operanden der gleichen Länge ℓ dargestellt.



6.3 Normalisierung

Die verschiedenen Möglichkeiten zur Normalisierung von Fließkommazahlen wurden in Abschnitt 2.6.2 besprochen. Das Problem liegt darin, dass bei der BCD-Arithmetik (standardmäßig) zur Basis 10000 Nullen am Ende nicht einfach zu erkennen und nur durch aufwändige Shifts zu entfernen sind. Die Einführung einer Tabelle, die die Anzahlen der End-Nullen für jede BCD-Ziffer enthält, sorgt aber für deutliche Beschleunigung.

Es macht wenig Sinn, einzelne Grundoperationen in den drei Normalisierungsmodi auszumessen. Vielmehr sollten die Auswirkungen auf die komplexeren Stufen deutlich werden. Daher wurden beispielhaft Messungen bei der Berechnung der Exponentialfunktion und bei der Auswertung eines einfachen arithmetischen Ausdrucks mit dem Wiederberechnungsverfahren gemacht (mit variabler Stellenzahl p). Sie zeigen, dass es von der Geschwindigkeit her nicht mehr problematisch ist, mit der Übersetzungs-Option zu arbeiten, nach jeder Operation zu normalisieren. Allerdings sieht man auch, dass es bei völliger Abschaltung der Normalisierung nicht zu ungünstigen Zeiteffekten durch unnötig lange Mantissen kommt; diese Einstellung ist immer noch die schnellste.

Normalisierung: Auswirkungen bei exp					
p	Fak. 0	Mod. 0	Mod. 1	Mod. 2	Fak. 2
16	0.93	4.00 e-4	4.30 e-4	4.30 e-4	1
20	0.92	4.50 e-4	4.90 e-4	4.90 e-4	1
24	0.91	5.00 e-4	5.50 e-4	5.60 e-4	1.02
28	0.94	6.00 e-4	6.40 e-4	6.40 e-4	1
32	0.94	6.50 e-4	6.90 e-4	7.00 e-4	1.01
36	0.94	7.50 e-4	8.00 e-4	8.00 e-4	1
40	0.97	9.00 e-4	9.25 e-4	9.40 e-4	1.02
44	0.95	1.00 e-3	1.05 e-3	1.05 e-3	1
48	0.92	1.10 e-3	1.20 e-3	1.25 e-3	1.04
52	0.98	1.30 e-3	1.33 e-3	1.34 e-3	1.01
56	0.92	1.40 e-3	1.53 e-3	1.60 e-3	1.05
60	0.97	1.70 e-3	1.75 e-3	1.76 e-3	1.01
64	0.98	1.90 e-3	1.93 e-3	1.96 e-3	1.02

Bei der Berechnung der Exponentialfunktion (im Bereich, in dem keine Argumentreduktion notwendig ist), ergaben sich die in der vorangegangenen Tabelle dargestellten Werte für eine Auswertung (in Sekunden; hier lohnt keine Darstellung als Diagramm). Die Faktoren bei den Einstellungen „0“ (gar keine Normalisierung) und „2“ (eigentliche Normalisierung) sind bezogen sich auf die Normaleinstellung „1“ (nur BCD-Nullen entfernen).

Folgende Ergebnisse erhielt man bei der Auswertung des vielfach herangezogenen Ausdrucks $\cosh^2(x) - \sinh^2(x)$ bei kleinem Argument $x \approx 0.1$ mit p Stellen. Es geht die Zeit für den Aufbau des Baums und beide Berechnungsdurchgänge ein.

Normalisierung: Auswirkungen bei $\cosh^2(x) - \sinh^2(x)$					
p	Fak. 0	Mod. 0	Mod. 1	Mod. 2	Fak. 2
16	1	2.00 e-3	2.00 e-3	2.10 e-3	1.05
24	0.96	2.30 e-3	2.40 e-3	2.50 e-3	1.04
32	0.94	2.90 e-3	3.10 e-3	3.10 e-3	1
40	0.95	3.60 e-3	3.80 e-3	3.90 e-3	1.03
48	0.98	4.50 e-1	4.60 e-3	4.70 e-3	1.02
56	0.96	5.40 e-3	5.60 e-3	5.80 e-3	1.04
64	1	7.00 e-3	7.00 e-3	7.20 e-3	1.03

Da die Arithmetik zu [Steins] immer ohne Normalisierung arbeitet, wurde in den entsprechenden Tests auch bei unserer Version immer mit dieser Einstellung übersetzt, um eine direkte Vergleichbarkeit zu ermöglichen.

6.4 Intervallmultiplikation

Die Geschwindigkeitsgewinne, die sich durch die Reduktion von vier auf drei notwendige `BigFloat`-Multiplikationen (im Fall, dass 0 in beiden Intervallen liegt) ergeben, wurden ja bereits in Abschnitt 2.7.2 erwähnt. Hier sollen nun nur noch die tatsächlich gemessenen Zeiten vorgestellt werden.

Gemessen wurde jeweils ein Repräsentant der 8 nicht kritischen Fälle und der 16 Teilfälle des kritischen, jeweils mit 16-stelligen Grenzen.

Es sind die endgültige implementierte Fassung (mit 3 oder auch nur 2 Multiplikationen), der Heindl-Operator (mit 3), die Standard-Fassung mit 4 und die mit 3 Multiplikationen (mit Vorab-Vorzeichentests, aber ohne die zusätzlichen Vergleiche), sowie die Faktoren gegenüber der Endfassung aufgeführt (Werte für eine Intervall-Multiplikation in Sekunden):

Intervall-Multiplikation	XAri	Heindl	Fak.	4 Mul	Fak.	3 Mul	Fak.
nicht kritische Fälle	7.00 e-6	1.60 e-5	2.29	7.00 e-6	1.00	7.00 e-6	1.00
kritisch, 3 Multiplikationen	1.19 e-5	1.88 e-5	1.58	1.47 e-5	1.24	1.14 e-5	0.96
kritisch, 2 Multiplikationen	8.25 e-6	1.88 e-5	2.27	1.47 e-5	1.78	1.14 e-5	1.38
kritisch, gewichtet gemittelt	1.10 e-5	1.88 e-5	1.71	1.47 e-5	1.34	1.14 e-5	1.04

6.5 Standardfunktionen

Die folgende Tabelle dient zunächst dazu, den Aufwand der unterschiedlichen Funktionen im Vergleich miteinander grob zu veranschaulichen, und auch dazu, die Stellen zu kennzeichnen, an denen gegenüber der Version zu [Steins] die deutlichsten Verbesserungen erzielt werden konnten. Die Funktionen wurden der Einfachheit halber im festen Argument $x \approx 0.5$ (mit 8-stelliger Mantisse ohne BCD-Nullen) ausgewertet. Ausnahmen sind arcosh und arcoth mit $x \approx 1.5$; bei pow, abs und arg wird als zweites Argument $y \approx 1.5$ verwendet, bei root die dritte Wurzel. Es ist zu beachten, dass so bei einigen Funktionen Argumentreduktionen notwendig sind (exp), bei anderen nicht (sin, cos). Einige Funktionen sind in [Steins] nicht implementiert. Die Stellenforderung ist jeweils 32; die Angaben sind die benötigte Zeit (in Sekunden) für eine Auswertung.

Funktion	XAri	[Steins]	Faktor
exp	7.00 e-4	1.50 e-2	21.43
sin	6.00 e-4	4.00 e-3	6.67
cos	7.00 e-4	3.50 e-3	5.00
tan	1.30 e-3	8.00 e-3	6.15
cot	1.40 e-3	8.00 e-3	5.71
sinh	6.00 e-4	1.85 e-2	30.83
cosh	8.00 e-4	1.85 e-2	23.13
tanh	8.00 e-4	3.55 e-2	44.38
coth	8.00 e-4	–	–
arcsin	1.70 e-3	1.55 e-2	9.12
arccos	1.90 e-3	1.70 e-2	8.95
arctan	1.20 e-3	6.50 e-3	5.42
arccot	1.40 e-3	–	–

Funktion	XAri	[Steins]	Faktor
arsinh	9.00 e-4	1.40 e-1	155.56
arcosh	1.00 e-3	6.40 e-2	64.00
artanh	8.00 e-4	4.60 e-2	57.50
arcoth	7.00 e-4	–	–
sqrt	2.00 e-4	5.50 e-3	27.50
log	5.60 e-4	1.11 e-1	198.21
log10	6.80 e-4	2.05 e-1	301.47
pow	3.16 e-3	2.20 e-1	69.62
root	2.60 e-3	–	–
abs	5.20 e-4	–	–
arg	2.50 e-3	–	–
sincos	9.00 e-4	–	–
sinhcosh	1.30 e-3	–	–

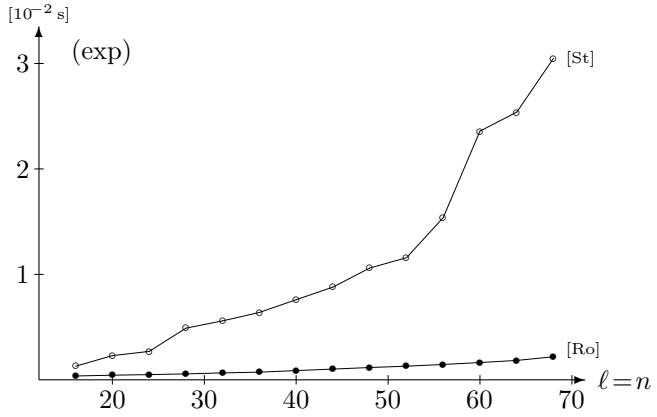
Die deutlichsten Beschleunigungen zeigen sich beim Logarithmus, der in [Steins] mit einem Intervall-Newton-Verfahren (ein Aufruf der Exponentialfunktion in jedem Schritt) und bei uns über die Potenzreihe bestimmt wird. Die geforderte Genauigkeit schlägt sich hier stärker nieder; bei 16 Stellen ergibt sich beispielsweise ein normaler Faktor von 24 (siehe

eine spätere Tabelle). Da die Areafunktionen bei [Steins] immer über eine algebraische Identität bestimmt werden, die den Logarithmus enthält, ergeben sich bei ihnen ähnliche Effekte (ebenso natürlich bei pow).

Die kombinierte Variante sincos ist um einen Faktor 1.44 schneller als ein nacheinander berechneter Sinus und Cosinus (die eingesparte Argumentreduktion geht hier noch nicht ein). Bei sinhcosh ergibt sich analog nur ein Faktor 1.08, was daran liegt, dass im Argument $x \approx 0.5$ für sinh die Potenzreihe mit Argumentreduktion herangezogen wird, also praktisch doch sinh und cosh nacheinander berechnet werden (für $x \approx 2$ erhält man den Faktor 1.8).

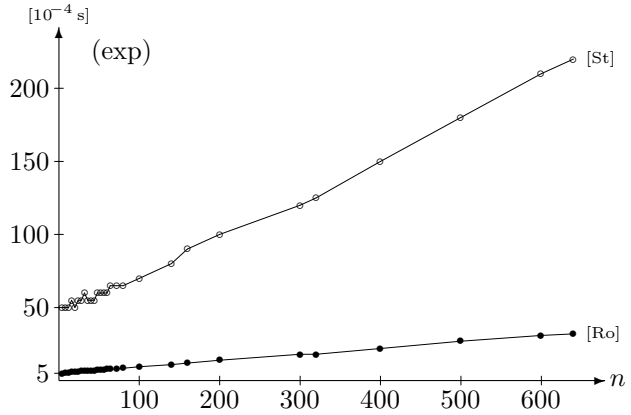
Wir wollen außerdem noch das Verhalten der beiden Implementationen bei steigenden Stellenzahlen betrachten.

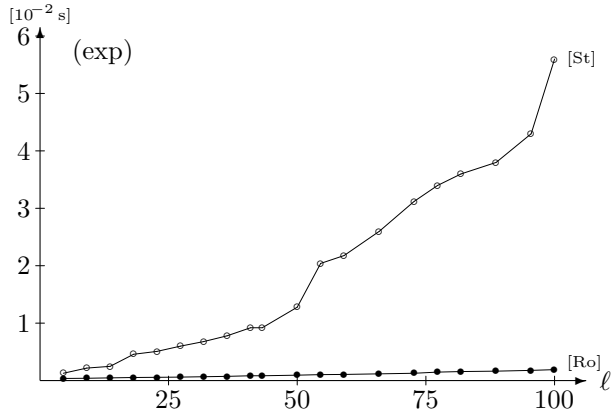
Im Diagramm auf der rechten Seite ist die benötigte Zeit für die Auswertung der Exponentialfunktion in $x \approx 0.1234$ mit ℓ-stelligem Argument und n Stellen geforderter Genauigkeit dargestellt. Die Implementation zu [Steins] bricht bei höheren Stellenzahlen noch deutlicher ein.



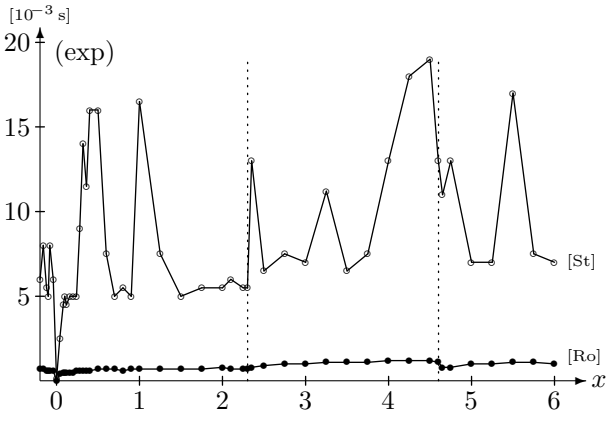
Es ist bei [Ro] kein deutlicher Effekt der rekursiv implementierten Multiplikation zu erkennen, die auf dem Testrechner ab ca. 48 Stellen der Operanden der Multiplikation, also schon bei etwas geringeren Stellenforderungen bei exp, verwendet wird.

In den beiden unteren Diagrammen ist das Verhalten bei steigender (auch einmal extrem groß gewählter) Mantissenlänge n des Arguments bei gleicher Genauigkeitsforderung 32 (links) und bei steigender Genauigkeitsforderung ℓ bei gleich bleibender Mantissenlänge 32 dargestellt. Die Knickstellen im oberen und unten rechts stehenden Graphen zu [Steins] sind bisher unerklärt.

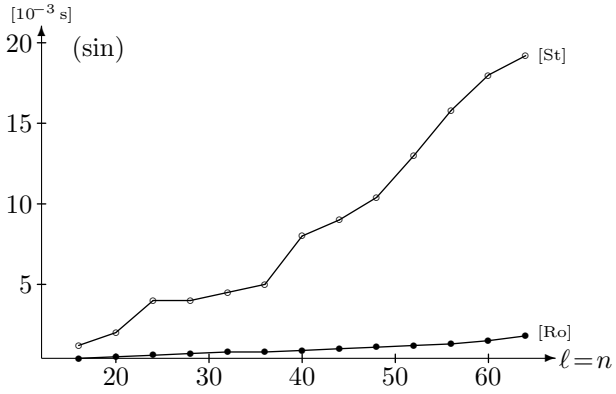


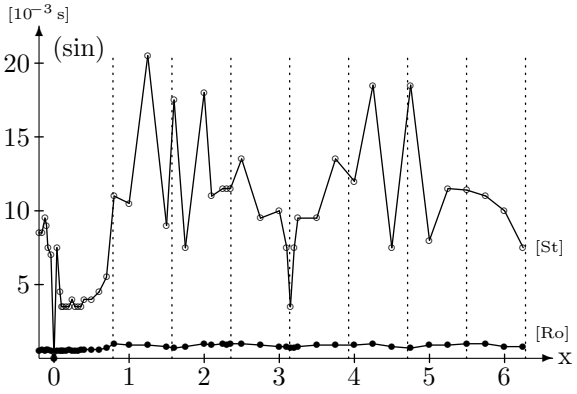


Zuletzt betrachten wir die Situation bei variablem Parameter x (der festen Mantissenlänge 8) und fester Genauigkeit 32 (siehe rechts). Die Grenzen der ersten Argumentreduktion sind durch gestrichelte Linien markiert. In unserer Version ist direkt rechts von diesen Grenzen ein leichtes Einknicken zu beobachten, das darauf zurückzuführen ist, dass das Argument auf einen Wert nahe 0 reduziert wird. Die Gründe für die Sprünge im Graphen zu [Steins] sind bislang noch nicht identifiziert.



Analog stellen wir noch das Verhalten der Sinusberechnung für variable Stellenzahlen/Genauigkeiten und für variables Argument dar. Auch hier sind rechts die Gebiete einheitlicher Argumentreduktion durch gestrichelte Linien voneinander getrennt. Der Sinusverlauf reproduziert sich andeutungsweise im Graphen zu unserer Arithmetik.





Zuletzt muss natürlich darauf hingewiesen werden, dass die reine Performance der in Software realisierten Routinen um viele Größenordnungen niedriger ist als die in Hardware realisierter Funktionen, etwa durch Fließkommaprozessoren. Zusätzlich zur Hardwareunterstützung können diese von einem Format fester Länge und kleineren Exponentenbereichen ausgehen, so dass intern schnellere Approximationsmethoden verwendet werden können. Entsprechend ergeben sich Faktoren um die 1000 gegenüber unseren Versionen.

In der folgenden Tabelle sind Messungen der in der `C++`-Bibliothek verfügbaren Standardfunktionen (die auf dem L*i*nux-Rechner in Aufrufe der IEEE-kompatiblen Fließkomma-Einheit umgesetzt werden) festgehalten, im Vergleich zu unserer Arithmetik und der zu [Steins] (mit denselben Argumenten wie zur ersten Tabelle des Abschnitts).

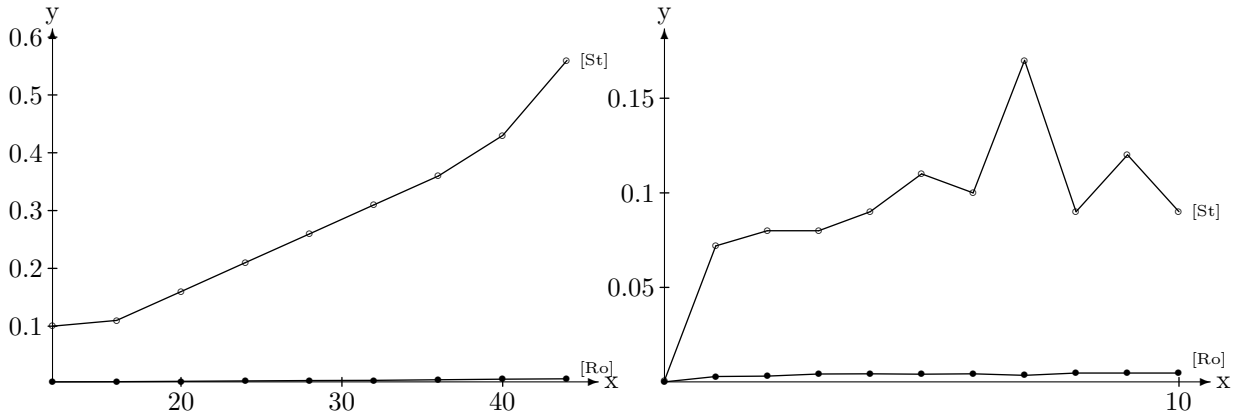
Die Messungen (Werte in Sekunden für einen Aufruf) bei den beiden Software-Versionen wurden der Vergleichbarkeit wegen mit 16-stelligen Argumenten und einer Genauigkeitsforderung von 16 Stellen durchgeführt. Die IEEE-Ergebnisse lagen immer innerhalb der von uns berechneten Einschließung, außer bei arcsin (das IEEE-Ergebnis war aber noch immer hoch genau, mit einer Abweichung von 2 in der letzten Stelle).

	IEEE	XAri	Faktor	[St]	Faktor	[St]/XAri
exp	6.40 e-7	4.80 e-4	750	3.40 e-3	5313	7.08
sin	3.20 e-7	4.50 e-4	1406	1.30 e-3	4063	2.89
cos	3.80 e-7	4.50 e-4	1184	1.90 e-3	5000	4.22
tan	4.60 e-7	1.00 e-3	2174	6.50 e-3	14130	6.50
sinh	8.80 e-7	4.00 e-4	455	4.40 e-3	5000	11.00
cosh	7.80 e-7	6.00 e-4	769	4.40 e-3	5641	7.33
tanh	7.20 e-7	7.00 e-4	972	1.42 e-2	19722	20.29
arcsin	7.40 e-7	1.10 e-3	1486	8.20 e-3	11081	7.45
arccos	7.80 e-7	1.20 e-3	1538	9.20 e-3	11795	7.67
arctan	3.40 e-7	8.00 e-4	2353	3.40 e-3	10000	4.25
arsinh	1.00 e-6	7.00 e-4	700	2.06 e-2	20600	29.43
arcosh	9.60 e-7	8.00 e-4	833	1.90 e-2	19792	23.75
artanh	8.20 e-7	5.00 e-4	610	1.28 e-2	15610	25.60
sqrt	2.20 e-7	1.60 e-4	727	2.80 e-3	12626	17.50
log	4.80 e-7	4.00 e-4	833	9.50 e-3	19792	23.75
log 10	5.00 e-7	5.00 e-4	1000	1.85 e-2	37000	37.00
pow	1.06 e-6	1.95 e-3	1840	2.40 e-2	22642	12.31

6.6 Auswertung arithmetischer Ausdrücke

Wir vergleichen hier die Laufzeiten für die Auswertung des nun klassischen $\cosh^2(x) - \sinh^2(x)$. Die Werte beinhalten die Zeit für den Aufbau des Ausdrucks als Baum, sowie die beider Durchläufe der Auswertung.

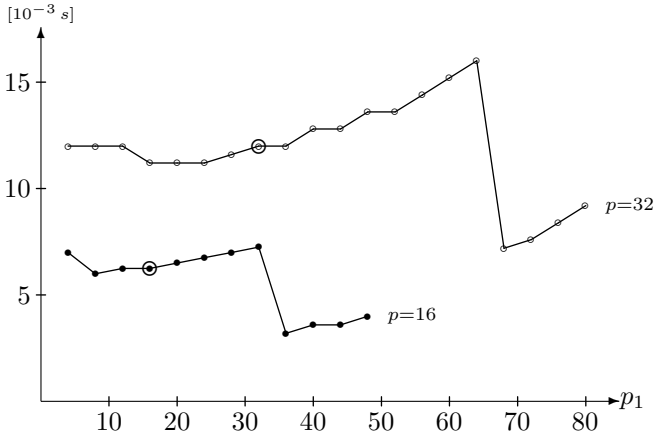
Im Diagramm links ist die Auswertung in festem $x = 5.0$ mit variabler Genauigkeitsforderung p dargestellt, die Gewinnfaktoren bewegen sich hier im Bereich 22 bis 62. Rechts wird mit fester Genauigkeit von 16 Stellen ausgewertet und das Argument x variiert; es ergeben sich Faktoren von 13 (in $x = 0$) bis 47. (Die Arithmetik zu [Steins] liefert hier meistens keine hoch genauen Ergebnisse, vermutlich aufgrund der in 5.2 beschriebenen Effekte).



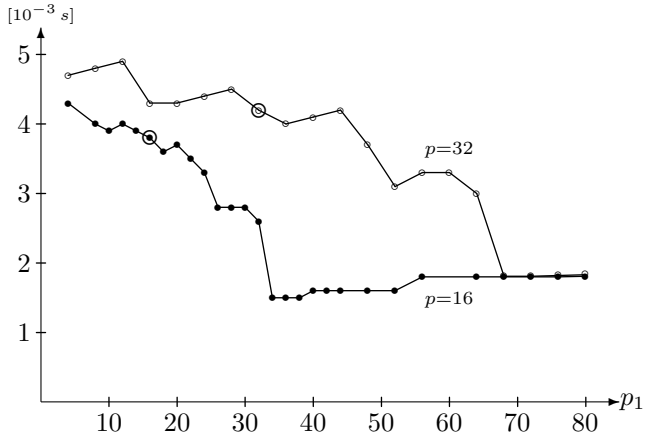
Wir schauen uns die Effekte an, die bei unterschiedlicher Wahl der Stellenzahl p_1 im ersten Durchgang auftreten können (nur bei unserer Arithmetik). Wir verwenden dazu das Beispiel

2 aus Abschnitt 5.8.6, also die Näherung für die zweite Ableitung über den Differenzenquotienten. Gemessen wurde für $p = 16$ und $p = 32$ (mit $h = 10^{-8}$ bzw. $h = 10^{-16}$). Die Stelle $p_1 = p$ ist jeweils mit einem Kreis markiert.

Unten ist das Ergebnis für $f(x) = \sin(x)$ dargestellt. Zu erkennen ist deutlich die Stelle, ab der bereits der erste Pass Hochgenauigkeit erreicht, so dass der zweite Pass gar nicht durchgeführt zu werden braucht. Diese Stelle ist aber vorab nicht zu „erraten“, und wesentlich zu große Stellenzahlen produzieren natürlich wieder wesentlich höhere Laufzeiten. Wahlen für p_1 etwas kleiner als p sind hier günstiger als $p_1 = p$. Sehr kleine p_1, also zu grobe erste Einschließungen, können dagegen zu hohe Stellenzahlen für den zweiten Pass bewirken.



Wenn dagegen für $f(x)$ die in 5.8.6 angegebene rationale Funktion verwendet wird, ergibt sich ein etwas kurioses Bild (unten). Die Buckel sind auf die vielen vorkommenden Einzelterme zurückzuführen, die nicht immer alle ein zweites Mal berechnet werden müssen. Die Funktionsberechnung enthält nur Grundrechenarten und ist daher gegenüber dem Algorithmus so schnell, dass sich eine Berechnung mit hohen Stellenzahlen als günstiger erweist.



Ausblick

Natürlich ist die Arbeit an einer Klassenbibliothek wie der nun vorliegenden nie wirklich abgeschlossen. Von der üblichen Pflege und Verschleifung einiger Kanten abgesehen, bieten sich aber auch einige konkrete Ansatzpunkte für Erweiterungen.

Gegenüber der Implementation zu [Steins] konnten auf allen Ebenen viele Bestandteile deutlich beschleunigt werden. Angesichts dessen, dass der Abstand zur Hardware-Arithmetik noch drei Zehnerpotenzen beträgt, sind Beschleunigungen der unterliegenden Grundoperationen natürlich dennoch höchst willkommen. Es sind Umsetzungen der innersten Kern-Routinen in (systemabhängigen) Assemblercode denkbar. Die gerichtet gerundete Fließkomma-Addition kann noch in einigen Fällen verbessert werden, in denen momentan noch intern unnötig viele Stellen berechnet werden. Das Design der langen Integer-Zahlen sollte dahingehend geändert werden, dass das Vorzeichen nicht zusammen mit der eigentlichen Ziffernfolge abgespeichert wird, da sonst bei Vorzeichenwechseln komplette Kopien notwendig werden. Einige Feinheiten, z.B. eine *Intervall*routine für `sincos` mit nur einer Argumentreduktion und verzahnter Fallunterscheidung, stehen noch aus.

Die momentane Stand-Alone-Implementation sollte problemlos mit anderen Werkzeugen zur Verifikationsnumerik zusammenarbeiten. Eine direktere Anbindung an die XSC-Sprachfamilie (insbesondere C-XSC) sollte dann aber beispielsweise noch Umwandlungsroutinen der verschiedenen Fließkommaformate, insbesondere der Langzahlformate beinhalten (Akkumulator-Format und Staggered-Correction) und die Ein- und Ausgabe sowie das Exception-Handling den XSC-Gepflogenheiten genau anpassen.

Die erst begonnene Erweiterung der Standardfunktionen auf komplexe Argumente sollte, unter Verwendung der umfangreichen Ergebnisse aus [Braune] und [Krämer], vervollständigt werden. Letztlich ist dies natürlich hauptsächlich gedacht als Grundlage für die entsprechende komplexe Erweiterung des Auswertungsalgorithmus für arithmetische Ausdrücke.

> Der Gebildete treibt die Genauigkeit nicht weiter, als es der Natur der Sache entspricht.
>
> *Aristoteles (384-322 v.Chr.)*

Literatur

[AlHe] Alefeld, Götz, und Herzberger, Jürgen, *Einführung in die Intervallrechnung*, Bibliographisches Institut AG, Mannheim, 1974.

[Atlas] *dtv-Atlas zur Mathematik, Band 2: Analysis und angewandte Mathematik*, Deutscher Taschenbuch Verlag, 1977

[Braune] Braune, Klaus Dieter, *Hochgenaue Standardfunktionen für reelle und komplexe Punkte und Intervalle in beliebigen Gleitpunktrastern*, Dissertation, Universität Karlsruhe, 1987

[HaHoKuRa] Hammer, Hocks, Kulisch, Ratz, *C++ Toolbox for Verified Computing, Volume I*, Springer, 1995

[HaPa] Habile, Bruno & Papanikolaou, Thomas, *Fast multiprecision evaluation of series of rational numbers*, Technical Report No. TI-7/97, Technische Universität Darmstadt University 1997, `www.informatik.tu-darmstadt.de/TI/Mitarbeiter/papanik`

[Heindl] Heindl, Gerhard, *An improved algorithm for computing the product of two machine intervals*, interner Bericht der integrierten Arbeitsgruppe Mathematische Probleme aus dem Ingenieurbereich, Fachbereich Mathematik der Bergischen Universität Gesamthochschule Wuppertal, IAGMPI–9304, 1993

[Knuth] Knuth, Donald E., *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, Addison-Wesley, Reading, Massachusetts, 1981

[Krämer] Krämer, Walter, *Inverse Standardfunktionen für reelle und komplexe Intervallargumente mit a priori Fehlerabschätzungen für beliebige Datenformate*, Dissertation, Universität Karlsruhe, 1987

[Muller] Muller, Jean-Michel, *Elementary Functions, Algorithms and Implementation*, Birkhäuser, 1997

[Rall] Rall, L.B., *Automatic Differentiation, Techniques and Applications*, Lecture Notes in Computer Science 120, Springer-Verlag, Berlin, 1981

[Rogat] Rogat, Axel, *Aufbau und Arbeitsweise von Compilern*, Bericht der integrierten Arbeitsgruppe Mathematische Probleme aus dem Ingenieursbereich, Fachbereich Mathematik der Bergischen Universität Gesamthochschule Wuppertal, IAGMPI–2000/1, 2000

[Steins] Steins, Andreas, *Verifizierte Formelauswertung in Computer-Algebra-Systemen und objektorientierten Programmierumgebungen*, Dissertation, Universität Wuppertal, 1995

[Wall] H.S. Wall, *Analytic Theory of Continued Fractions*, Chelsea Publishing Company New York, 1948

Diese Arbeit im `dvi`- und PDF-Format, sowie die Quellen der implementierten Arithmetik, sind im Internet zu finden unter `www.math.uni-wuppertal.de`.

Index