Bergische Universität
Wuppertal

Real and Complex Staggered (Interval) Arithmetics with Wide Exponent Range (in German)

Frithjof Blomquist, Werner Hofschuster, Walter Krämer

Preprint 2008/1

Wissenschaftliches Rechnen/
Softwaretechnologie

Diese Arbeit entstand im Rahmen des gemeinsamen Projektes
Erweiterung des Funktionsumfanges der C-XSC Bibliothek

Impressum

Herausgeber:	Prof. Dr. W. Krämer, Dr. W. Hofschuster Wissenschaftliches Rechnen/Softwaretechnologie Fachbereich C (Mathematik und Naturwissenschaften) Bergische Universität Wuppertal Gaußstr. 20 D-42097 Wuppertal

Internet-Zugriff

Die Berichte sind in elektronischer Form erhältlich über die World Wide Web Seiten

`http://www.math.uni-wuppertal.de/wrswt/literatur.html`

Autoren-Kontaktadressen

Frithjof Blomquist
Adlerweg 6
D-66436 Püttlingen

E-mail: `blomquist@math.uni-wuppertal.de`

Werner Hofschuster
Bergische Universität Wuppertal
Gaußstr. 20
D-42097 Wuppertal

E-mail: `hofschuster@math.uni-wuppertal.de`

Walter Krämer
Bergische Universität Wuppertal
Gaußstr. 20
D-42097 Wuppertal

E-mail: `kraemer@math.uni-wuppertal.de`

Real and Complex Staggered (Interval) Arithmetics with Wide Exponent Range (in German)

Frithjof Blomquist, Werner Hofschuster, Walter Krämer

Januar, 2008

Inhaltsverzeichnis

1 Zusammenfassung

In der vorliegenden Arbeit wird bei der Staggered Intervall-Arithmetik die typische Über- und Unterlaufproblematik und der beim Unterlauf oft auftretende Genauigkeitsverlust angesprochen. Diese Probleme lassen sich weitgehend vermeiden, wenn neben den bereits implementierten Klassen `l_interval`, `l_cinterval` (gewöhnliche Staggered Arithmetik) in den neu entwickelten C-XSC Klassen `lx_interval` und `lx_cinterval` (erweiterete Staggered Arithmetik) ein zusätzlicher Faktor 2^n mitgeführt wird, durch den der staggered Zahlenbereich wegen $-2147483648 \leq n \leq +2147483647$ ganz erheblich erweitert wird. Für die reellen und komplexen erweiterten (Intervall-) Grundoperationen und für 42 reelle und komplexe elementare transzendente Funktionen werden Algorithmen angegeben, die einen zwischenzeitlichen Über- oder Unterlauf vermeiden und bei hinreichend schmalen Eingangsintervallen eine optimale Genauigkeit liefern. Anwendungsbeispiele zeigen, wie numerische Probleme, die mit den Klassen `l_interval` und `l_cinterval` gar nicht oder nur schlecht lösbar sind, mit den neuen Klassen erfolgreich und mit großer Leichtigkeit bearbeitet werden können.

Die C-XSC Module und Programme stehen im Netz unter

`http://www.math.uni-wuppertal.de/wrswt/`

zur Verfügung.

Keywords: C-XSC, erweiterte Staggered Intervall-Arithmetik, großer Exponentenbereich, reelle und komplexe Intervallfunktionen, Einschließungsmethoden.

MSC (2000): 65G40, 65L70, 33B10, 33B99, 26A09;

2 Einleitung

Für Algorithmen aus dem Bereich der Numerik mit Ergebnisverifikation [18, 3, 11, 23] werden neben den Grundoperationen auch die Intervall-Standardfunktionen benötigt, um den Wertebereich eines vorgegebenen Ausdrucks garantiert einschließen zu können. Die berechneten Einschließungsintervalle sollen dabei möglichst eng sein.

Bei vielen Aufgabenstellungen ist die im IEEE double-Format maximal erreichbare Genauigkeit von höchstens 16 Dezimalziffern unzureichend, [4]. In C-XSC kann diese Genauigkeit der Einschließungen mit Hilfe der implementierten Klassen `l_interval`, `l_cinterval` für reelle und komplexe Rechnungen mit Hilfe einer Intervall-Staggered-Correction-Arithmetik, kurz Staggered-Arithmetik, etwa um den Faktor 20 vergrößert werden, [2, 5, 19, 20, 25, 22].

In C-XSC lässt sich eine Variable $\boldsymbol{x}$ vom Typ `l_interval` interpretieren als ein Feld mit $p+1$ Komponenten x_i vom Typ `real`:

$$\boldsymbol{x} = (x_1, x_2, \ldots, x_{p+1}), \quad x_i \in S(2,53), \quad p = \texttt{StagPrec(x)} \geq 1, \tag{1}$$

das mathematisch die folgende Bedeutung hat:

$$\boldsymbol{x} = \sum_{i=1}^{p-1} x_i + \boldsymbol{z}, \quad \boldsymbol{z} = [x_p, x_{p+1}], \quad x_i \in S(2,53), \quad i = 1, 2, \ldots, p+1. \tag{2}$$

Für $p = 1$ gilt damit $\boldsymbol{x} = \boldsymbol{z}$, und für $p \geq 2$ ergibt sich:

$$\texttt{Inf}(\boldsymbol{x}) = \sum_{i=1}^{p} x_i, \qquad \texttt{Sup}(\boldsymbol{x}) = \sum_{i=1}^{p-1} x_i + x_{p+1}. \tag{3}$$

Mit dem absoluten Durchmesser $\Delta(\boldsymbol{x}) := x_{p+1} - x_p \geq 0$ ist der relative Durchmesser von $\boldsymbol{x}$ definiert durch, [10]:

$$\texttt{RelDiam}(\boldsymbol{x}) := \frac{\Delta(\boldsymbol{x})}{<|\boldsymbol{x}|>}, \quad <|\boldsymbol{x}|> := \min\left\{|r| \,\middle|\, r \in \boldsymbol{x}\right\}, \quad <|\boldsymbol{x}|> \neq 0.$$

Im Fall $0 \in \boldsymbol{x}$ definiert man: $\texttt{RelDiam}(\boldsymbol{x}) := \Delta(\boldsymbol{x})$.

Ist das Intervall $\boldsymbol{x}$ eine berechnete Einschließung für einen beliebigen exakten Wert $r \in \boldsymbol{x}$, so ist der relative Durchmesser von $\boldsymbol{x}$ eine Oberschranke für den relativen Fehler bez. dieser Einschließung. Zusätzlich liefert der negative Zehnerexponent des relativen Durchmessers auch noch die Maximalzahl der korrekten Dezimalziffern von $\boldsymbol{x}$. Gilt z.B. $\texttt{RelDiam}(\boldsymbol{x}) = 2.34\ldots \cdot 10^{-87}$, so stimmen $\texttt{Inf}(\boldsymbol{x})$ und $\texttt{Sup}(\boldsymbol{x})$ auf etwa 87 Dezimalziffern überein.

Das nächste Beispiel zeigt, dass eine Staggered-Intervallarithmetik nur dann sinnvoll ist, wenn bei einer vorgegebenen Präzision p der relative Durchmesser von $\boldsymbol{x}$ folgende Bedingung erfüllt:

$$\texttt{RelDiam}(\boldsymbol{x}) < 10^{-16*(p-1)}. \tag{4}$$

Um (4) zu verstehen, wählen wir mit $p = 3$ zunächst drei dicht aufeinander folgende `real`-Zahlen[1] $x_i,\ i = 1, 2, 3$, und setzen $x_4 = 2.46 \cdot 10^{-32}$.

$$x_1 = 1; \quad x_2 = 10^{-16}; \quad x_3 = 1 \cdot 10^{-32}; \quad x_4 = 2.46 \cdot 10^{-32};$$

Nach einer formalen Addition[2] $\boldsymbol{x} + 0$ erhält man das Ergebnis:

$$x_1 = 1; \quad x_2 = 10^{-16}; \quad x_3 = -2.325\ldots \cdot 10^{-33}; \quad x_4 = 1.227\ldots \cdot 10^{-32};$$

mit $\texttt{RelDiam}(\boldsymbol{x}) = 1.46 \cdot 10^{-32}$. Zur Darstellung von $\boldsymbol{x}$ werden also alle x_i benötigt.

Wir wählen jetzt $x_4 = 2.47 \cdot 10^{-32}$ etwas größer, und mit

$$x_1 = 1; \quad x_2 = 10^{-16}; \quad x_3 = 1 \cdot 10^{-32}; \quad x_4 = 2.47 \cdot 10^{-32};$$

erhält man jetzt nach der gleichen formalen Addition $\boldsymbol{x} + 0$ das Ergebnis[3]:

$$x_1 = 1; \quad x_2 = \mathbf{0}; \quad x_3 = 1.00\ldots \cdot 10^{-16}; \quad x_4 = 1.00\ldots \cdot 10^{-16};$$

mit $\texttt{RelDiam}(\boldsymbol{x}) = 3.697\ldots \cdot 10^{-32}$. Durch den etwas größeren relativen Durchmesser erhalten wir jetzt jedoch $x_2 = \mathbf{0}$, so dass $\boldsymbol{x}$ jetzt auch mit der kleineren Präzision $p = 2$ darstellbar wäre. Nach (4) sollte man daher eine Staggered-Arithmetik nur mit hinreichend schmalen Intervallen durchführen.

[1] Die für x_2, x_3, x_4 angegebenen Dezimalzahlen sind zur jeweils nächsten Rasterzahl zu runden.

[2] Bei allen vier Grundoperationen wird zur optimalen Einschließung der Akkumulator benutzt und die Ergebnisse x_i werden dann der Reihe nach aus dem Akku ausgelesen.

[3] Intern gilt: $x_3 < 10^{-16} < x_4$, mit: $x_4 - x_3 = 3.69\ldots \cdot 10^{-32}$.

Im Vergleich zu einer Intervall-Langzahlarithmetik auf integer-Basis werden jetzt noch die Vor- und Nachteile der beschriebenen Staggered-Arithmetik angegeben. Die Vorteile einer Staggered-Arithmetik sind [8],[12],[19], [21],[25],[30]:

1. eine einfache Implementierung nach (1),
2. ein effizientes Laufzeitverhalten, da $\texttt{Inf}(\boldsymbol{x})$ und $\texttt{Sup}(\boldsymbol{x})$ nach (3) mit identischen x_i-Werten berechnet werden, $i = 1, 2, \ldots, p-1$.

Die Nachteile einer Staggered-Arithmetik sind:

1. der eingeschränkte Exponentenbereich. Für das in C-XSC benutzte double-Format gilt: $-1073 \leq \texttt{expo}(x_i) \leq +1024, \quad i = 1, 2, \ldots p.$
2. der drastische Genauigkeitsverlust, wenn bei hoher Präzi dass $\boldsymbol{a}_s \diamond \boldsymbol{b}_s$ noch keinesion, also z.B. mit $p = 5$, der betragsmäßig größte Wert x_1 in die Nähe des Unterlaufbereichs fällt.

Die folgenden Beispiele beschreiben diesen Genauigkeitsverlust.

Im **1. Beispiel** sind zwei Punktintervalle[4] $\boldsymbol{a}, \boldsymbol{b}$ mit der Präzision $p = 10$ und den Werten

$$\boldsymbol{a} = \underbrace{1.098...981}_{144 \text{ Dez.-Ziffern}} \cdot 10^{-154}, \qquad \boldsymbol{b} = \underbrace{1.234...789}_{144 \text{ Dez.-Ziffern}} \cdot 10^{-154}$$

zu multiplizieren. Da zur Darstellung des exakten Produktes $\boldsymbol{a} \cdot \boldsymbol{b} = 1.343 \ldots \cdot 10^{-308}$ 288 Dezimalstellen benötigt werden und die kleinste positive normalisierte Zahl des IEEE double Formats durch $\texttt{MinReal} = 2^{-1022} = 2.225 \ldots \cdot 10^{-308}$ gegeben ist, kann im IEEE Format die Maschinen-Einschließung[5] $\boldsymbol{a} \diamond \boldsymbol{b}$ des exakten Produktes $\boldsymbol{a} \cdot \boldsymbol{b}$ nur mit etwa 16 korrekten Dezimalziffern angegeben werden, und bei vielen Anwendungen ist dies völlig unzureichend. Der Genauigkeitsverlust lässt sich jedoch vermeiden, wenn $\boldsymbol{a}, \boldsymbol{b}$ zunächst rundungsfehlerfrei so weit nach oben skaliert werden

$$\boldsymbol{a}_s := 2^{+1022} \cdot \boldsymbol{a}, \quad \boldsymbol{b}_s := 2^{+1022} \cdot \boldsymbol{b},$$

dass $\boldsymbol{a}_s \diamond \boldsymbol{b}_s$ noch keinen Überlauf erzeugt, aber z.B. mit $p = 19$ die gewünschte hohe Genauigkeit von 288 Dezimalstellen liefert. Es gilt dann

$$\boldsymbol{a} \cdot \boldsymbol{b} = 2^{-2044} \cdot (\boldsymbol{a}_s \diamond \boldsymbol{b}_s). \tag{5}$$

Speichert man den Exponenten -2044 und das Maschinenprodukt $\boldsymbol{a}_s \diamond \boldsymbol{b}_s$, so kann man diese Zwischenergebnisse in nachfolgenden Rechnungen ohne Genauigkeitsverlust weiter verarbeiten.

Im **2. Beispiel** soll mit der Präzision $p = 19$ und den Staggered-Intervallen[6]

$$\boldsymbol{a} = 2^{-1022} + 2^{-1074}, \qquad \boldsymbol{b} = \texttt{sqrt(l\_interval(2))};$$

[4] Bei einem Punktintervall a der Präzision p gilt $a_p = a_{p+1}$ und bei einem echten Intervall $a_p < a_{p+1}$.
[5] Diese Maschinen-Einschließung ist notwendigerweise ein echtes Intervall.
[6] $\boldsymbol{a}$ ist dabei ein Punktintervall und $\boldsymbol{b}$ notwendigerweise ein echtes Intervall.

eine optimale Einschließung für $u := \sqrt{(2^{-1022} + 2^{-1074}) \cdot \sqrt{2}} \in \mathbb{R}$ berechnet werden. Die naive Intervallauswertung $\boldsymbol{u} = \mathtt{sqrt}(\boldsymbol{a} \diamond \boldsymbol{b})$ liefert die nur sehr grobe Einschließung

$$\sqrt{(2^{-1022} + 2^{-1074}) \cdot \sqrt{2}} \in \boldsymbol{u} = \underbrace{1.773902372731576}_{16 \text{ Dez.-Ziffern}} \overset{79\ldots}{65\ldots} \cdot 10^{-154}$$

mit nur 16 korrekten Dezimalstellen. Der Grund hierfür ist die notwendigerweise sehr grobe Einschließung des Arguments

$$v := (2^{-1022} + 2^{-1074}) \cdot \sqrt{2} \in \boldsymbol{a} \diamond \boldsymbol{b} = \underbrace{3.14672962798271}_{15 \text{ Dez.-Ziffern}} \overset{80\ldots}{75\ldots} \cdot 10^{-308},$$

da v sehr dicht am Unterlaufbereich liegt.

Eine wesentlich bessere Einschließung von u erhält man, wenn in $\mathtt{sqrt}(\boldsymbol{a} \diamond \boldsymbol{b})$ die Quadratwurzel aus jedem Faktor einzeln gezogen wird:

$$\sqrt{(2^{-1022} + 2^{-1074}) \cdot \sqrt{2}} \in \boldsymbol{u} := \mathtt{sqrt}(\boldsymbol{a}) \diamond \mathtt{sqrt}(\boldsymbol{b}) = \underbrace{1.773\ldots6833}_{170 \text{ Dez.-Ziffern}} \overset{75\ldots}{68\ldots} \cdot 10^{-154}.$$

Wir erhalten damit eine zehnfache Genauigkeit, da jetzt im ersten Faktor $\mathtt{sqrt}(\boldsymbol{a})$ das Argument $\boldsymbol{a} = 2^{-1022} + 2^{-1074}$ ein Punktintervall ist, und zur Berechnung von $\mathtt{sqrt}(\boldsymbol{a})$ etwa 170 Dezimalziffern zur Verfügung stehen.

Man erreicht eine noch größere Genauigkeit, auch ohne doppeltes Radizieren, wenn man folgende rundungsfehlerfreie Skalierungen vornimmt: $\boldsymbol{a}_s = 2^{1533} \cdot \boldsymbol{a},\ \boldsymbol{b}_s = 2^{511} \cdot \boldsymbol{b}$. Es gilt dann:

$$\sqrt{\boldsymbol{a} \diamond \boldsymbol{b}} = \sqrt{2^{-2044} \cdot (\boldsymbol{a}_s \diamond \boldsymbol{b}_s)} = 2^{-1022} \cdot \sqrt{\boldsymbol{a}_s \diamond \boldsymbol{b}_s} \subset 2^{-1022} \cdot \mathtt{sqrt}(\boldsymbol{a}_s \diamond \boldsymbol{b}_s), \quad \text{mit:}$$
$$\mathtt{sqrt}(\boldsymbol{a}_s \diamond \boldsymbol{b}_s) = \underbrace{7.972330293\ldots446131994819}_{308 \text{ Dez.-Ziffern}} \overset{1\ldots}{0\ldots} \cdot 10^{+153}.$$

Mit der obigen Skalierung erreicht man, dass $\boldsymbol{a}_s \diamond \boldsymbol{b}_s$ in die Nähe des Überlaufbereichs fällt und damit zur Berechnung der Quadratwurzel eine maximale Stellenzahl zur Verfügung steht. Wie im 1. Beispiel muss man auch jetzt den Exponenten -1022 und die Einschließung $\mathtt{sqrt}(\boldsymbol{a}_s \diamond \boldsymbol{b}_s)$ abspeichern, wenn diese Werte in nachfolgenden Rechnungen benutzt werden sollen.

3 Zur Notation

$\mathbb{R}$	Menge der reellen Zahlen
$\mathbb{C}$	Menge der komplexen Zahlen
$I\!R$	Menge der reellen Intervalle
IR	Menge der reellen Maschinen-Intervalle
IC	Menge der komplexen Maschinen-Intervalle
$S(b,l,\underline{e},\overline{e}) = S(b,l)$	Gleitkommaraster mit Basis b, Mantissenlänge l und Exponent e, mit: $\underline{e} \le e \le \overline{e}, \;\; \underline{e},\overline{e},e \in \mathbb{Z}$
$S(2,53,-1022,+1023) = S(2,53)$	IEEE-Datenformat *double*
`l_real`	Klasse der reellen staggered Zahlen
`lx_real`	Klasse der erweiterten, reellen staggered Zahlen
`l_interval`	Klasse der reellen staggered Intervalle
`lx_interval`	Klasse der erweiterten, reellen staggered Intervalle
$(a.ex, a.li)$	Objekt a der Klasse `lx_interval` mit dem Zweier-Exponenten $a.ex$ und dem staggered Intervall $a.li$ vom Typ `l_interval`
`l_cinterval`	Klasse der komplexen staggered Intervalle
`lx_cinterval`	Klasse der erweiterten, komplexen staggered Intervalle
x, y	reelle Größen; $x, y \in I\!R$
$\boldsymbol{x}, \boldsymbol{y}$	reelle oder komplexe staggered Intervalle
$\{+,-,\cdot,/\}$	exakte reelle Operatoren
$\{+,-,*,/\}$	Operatoren der C-XSC Sprache
$\{\diamondsuit, \diamondsuit, \diamondsuit, \diamondsuit\}$	Gleitkomma-Intervalloperatoren; z.B. $\boldsymbol{x},\boldsymbol{y} \in IR \quad \Longrightarrow \quad \boldsymbol{x}\cdot\boldsymbol{y} \subseteq \boldsymbol{x}\diamondsuit\boldsymbol{y} = \boldsymbol{x} * \boldsymbol{y}$
$\texttt{Minreal} = 2^{-1022} = 2.225.. \cdot 10^{-308}$	kleinste positive, normalisierte *double* Zahl
$\texttt{minreal} = 2^{-1074} = 4.940.. \cdot 10^{-324}$	kleinste positive, denormalisierte *double* Zahl
$\texttt{Maxreal} < 2^{+1024} = 1.797.. \cdot 10^{+308}$	größte, normalisierte *double* Zahl
$U := (-\texttt{Minreal}, +\texttt{Minreal})$	denormalisierter Zahlenbereich
$\texttt{Minreal} \le \lvert\widetilde{x}\rvert \le \texttt{Maxreal}$	normalisierter Bereich im *double*-Format
$\ln(x),\ x > 0$	reelle Logarithmusfunktion zur Basis e
$\text{lnp1}(x) = \ln(1+x),\ x > -1$	reelle Logarithmusfunktion zur Basis e
`Ln2_lx_interval()`	Einschließung von $\ln(2)$ in hoher Genauigkeit, vgl. Tabelle 2 auf Seite 18

4 Aufgabenstellung

Mit den Beispielen ab Seite 8 entsteht sofort die Frage, wie sich die dortigen Skalierungen zur Genauigkeitsoptimierung so systematisieren lassen, dass beim Aufruf geeigneter arithmetischer Operatoren und Standardfunktionen diese Skalierungen automatisch erfolgen. Für eine erweiterte, reellen Staggered-Intervall-Arithmetik ist dazu eine neue Klasse `lx_interval` zu implementieren, wobei ein Objekt dieser Klasse aus den privaten Datenelementen ex vom Typ `int` und li vom Typ `l_interval` bestehen soll. Symbolisiert man ein Objekt dieser Klasse mit (ex, li), so soll das entsprechende Staggered Intervall definiert sein durch:

$$(ex, li) := 2^{ex} \cdot li; \quad -2^{31} = -2147483648 \leq ex \leq 2^{31} - 1 = +2147483647.$$

Mit dem obigen Exponentenbereich wird man in fast allen praktischen Fällen auf keine Überlaufprobleme stoßen. In der Klasse `lx_interval` sind die vier arithmetischen Operatoren $\{+, -, *, /\}$ und alle gängigen Standardfunktionen zu implementieren. Bei den arithmetischen Operatoren ist durch geeignete Skalierungen eine möglichst maximale Genauigkeit der Ergebnisse zu realisieren. Bei den Standardfunktionen ist darauf zu achten, dass über einem jeweils maximalen Definitionsbereich die Funktionswerte in möglichst hoher Genauigkeit und mit geringen Laufzeiten eingeschlossen werden.

Für eine erweiterte, komplexe Staggered-Intervall-Arithmetik ist entsprechend eine neue Klasse `lx_cinterval` zu implementieren, wobei ein Objekt dieser Klasse aus zwei privaten Datenelementen vom Typ `lx_interval` bestehen soll. Symbolisiert man ein Objekt dieser Klasse mit (re, im), wobei die privaten Datenelemente re, im vom Typ `lx_interval` sein sollen, so ist das entsprechende komplexe Staggered Intervall definiert durch:

$$(re, im) := re + i \cdot im, \quad i = \sqrt{-1}.$$

Wie bei der reellen Rechnung sind in der Klasse `lx_cinterval` neben den Operatoren $\{+, -, *, /\}$ möglichst viele Konstruktoren und Standardfunktionen zu implementieren. Nach den Bemerkungen von Seite 7 sind die Standardfunktionen so zu implementieren, dass insbesondere für hinreichend schmale Eingangsintervalle eine optimale Genauigkeit erreicht wird. Bei zu breiten Eingangsintervallen $[a, b]$ wird man jedoch eine gewisse Überschätzung der Funktionswerte $f([a, b])$ nicht vermeiden können.

5 Erweiterte reelle Staggered Intervall-Arithmetik

Zunächst wird die neue Klasse `lx_interval` implementiert, deren Objekte Staggered Intervalle der Form $2^{ex} \cdot li$ repräsentieren. In dieser Klasse werden dann neben den Konstruktoren die Intervall-Operatoren[7] $\{ \oplus, \ominus, \odot, \oslash \}$ bereitgestellt, mit denen auf der Maschine die Grundoperationen mit diesen Objekten in optimaler Genauigkeit durchgeführt werden können. Einfache Beispielprogramme zeigen die Leistungsfähigkeit dieser Operationen. Neben den 42 Standardfunktionen werden zusätzlich auch noch 22 häufig gebrauchte Intervallkonstanten vom Typ `lx_interval` implementiert.

[7] In der C-XSC Sprache werden die Operatoren $\{ \oplus, \ominus, \odot, \oslash \}$ mit $\{+, -, *, /\}$ bezeichnet.

5.1 Die Klasse `lx_interval`

Die Klasse `lx_interval` ist u.a. wie folgt implementiert:

```
#include <sstream>
#include <string>
#include <l_interval.hpp>
#include "lx_real.hpp"

namespace cxsc {
class lx_interval
{
private:
    // ------------- private Datenelemente --------------------
    int ex;
    l_interval li;
    // The mathematical value of an object of type lx_interval is
    // interpreted as:  2^(ex) * li;
public:
    // ----------------- Konstruktoren ------------------------
    lx_interval(void)  throw() {}
    lx_interval(int n, const l_interval& a)
                               throw() : ex(n), li(a) { }
    lx_interval(int n, const l_real& a)
                               throw() : ex(n), li(a) { }
    lx_interval(int n, const interval& a)
                               throw() : ex(n), li(a) { }
    lx_interval(int n, const real& a)
                               throw() : ex(n), li(a) { }
    explicit lx_interval(const l_interval& a)
                               throw() : ex(0), li(a) { }
    explicit lx_interval(const l_real& a)
                               throw() : ex(0), li(a) { }
    lx_interval(const l_real& a, const l_real& b)
                              throw() : ex(0), li(a,b) { }
    explicit lx_interval(const interval& a)
                               throw() : ex(0), li(a) { }
    explicit lx_interval(const real& a)
                               throw() : ex(0), li(a) { }
    lx_interval(const real& a, const real& b)
                              throw() : ex(0), li(a,b) { }
    lx_interval(const lx_real& a, const lx_real& b)  throw();
    explicit lx_interval(const lx_real& a)
                 throw() : ex(expo(a)), li(lr_part(a)) { }
    lx_interval(int, const string&) throw();
/*  -------- and other member funktions of this class -------- */

}; // class lx_interval
}  // end namespace cxsc
```

Analog zur Klasse `lx_interval` liefert die eingebundene Klasse `lx_real` Objekte der Struktur $(ex, lr) = 2^{ex} \cdot lr$, wobei die privaten Datenelemente ex vom Typ `int` und lr vom Typ `l_real` sind. Die Klasse `lx_real` ermöglicht u.a. den Vergleich der erweiterten `real`-Zahlen (ex, lr) und damit die Implementierung der Funktionen `Inf`$(...)$ und `Sup`$(...)$ für Objekte der Klasse `lx_interval`. Auf die Klasse `lx_real` wird hier nicht weiter eingegangen.

5.1.1 Konstruktoren

Die zahlreichen Konstruktoren der Klasse `lx_interval` ermöglichen ein einfaches Einbinden von Objekten der Klassen `l_interval`, `lx_real`, `l_real`, `interval` und `real`. Alle Konstruktoren sind auf Seite 12 zusammengestellt. Besonders wichtig für die Anwendung ist der letzte Konstruktor

```
lx_interval(int p, const string& s) throw();
```

mit dessen Hilfe über den String `s` Objekte der Klasse `lx_interval` in dezimaler Form generiert werden können, vgl. dazu Seite 33. `p` ist dabei der Zehnerexponent. Mit `s = "[2.1,2.1]"` und `p = 700` generiert der Konstruktor ein Objekt a, mit: $10^{+700} \cdot [2.1, 2.1] \subseteq a$.

Ebenfalls wichtig für die Anwendung ist der Konstruktor

```
lx_interval(int ex, const l_interval &a) throw();
```

Mit `ex = 12345` und `a = l_interval(0.5,0.5)` generiert der Konstruktor ein Objekt $\boldsymbol{x}$ vom Typ `lx_interval`, mit: $2^{12345} \cdot [0.5, 0.5] = \boldsymbol{x}$, wobei $\boldsymbol{x}$ ein Punktintervall ist. Mit `"[2.1,2.1]" >> a` generiert der Konstruktor ein Objekt $\boldsymbol{x}$ vom Typ `lx_interval`, mit: $2^{12345} \cdot [2.1, 2.1] \subseteq \boldsymbol{x}$, wobei jedoch $\boldsymbol{x}$ jetzt kein Punktintervall ist, weil 2.1 auch im IEEE Staggered-Format nicht exakt dargestellt werden kann.

5.1.2 Zuweisungsoperatoren

In der Klasse `lx_interval` sind die folgenden Zuweisungsoperatoren definiert:

```
lx_interval & operator = (const lx_interval &a) throw( );
lx_interval & operator = (const l_interval &a) throw( );
lx_interval & operator = (const l_real     &a) throw( );
lx_interval & operator = (const real       &a) throw( );
lx_interval & operator = (const interval   &a) throw( );
lx_interval & operator = (const lx_real     &a) throw( );
```

Der für die Anwendung wichtige Operator

```
l_interval & operator = (const lx_interval& a) throw();
```

der in `l_interval.hpp` deklariert und in `lx_interval.cpp` implementiert ist, liefert eine optimale Einschließung von `a` durch ein Intervall `x` vom Typ `l_interval`. Falls dies wegen Overflow nicht möglich ist, erfolgt eine entsprechende Fehlermeldung. Ganz analog wird der Zuweisungsoperator

```
interval & operator = (const lx_interval& a) throw();
```

in `interval.hpp` deklariert und in `lx_interval.cpp` implementiert.

5.1.3 Mengenvergleiche

In der Klasse `lx_interval` sind die folgenden Mengenvergleiche definiert:

```
bool operator <  (const lx_interval &a, const lx_interval &b);
bool operator <= (const lx_interval &a, const lx_interval &b);
bool operator >  (const lx_interval &a, const lx_interval &b);
bool operator >= (const lx_interval &a, const lx_interval &b);

bool operator <  (const lx_interval &a, const l_interval &b);
bool operator <= (const lx_interval &a, const l_interval &b);
bool operator <  (const l_interval &a, const lx_interval &b);
bool operator <= (const l_interval &a, const lx_interval &b);
bool operator >  (const lx_interval &a, const l_interval &b);
bool operator >= (const lx_interval &a, const l_interval &b);
bool operator >  (const l_interval &a, const lx_interval &b);
bool operator >= (const l_interval &a, const lx_interval &b);

bool operator <  (const lx_interval &a, const interval &b);
bool operator <= (const lx_interval &a, const interval &b);
bool operator <  (const interval &a, const lx_interval &b);
bool operator <= (const interval &a, const lx_interval &b);
bool operator >  (const lx_interval &a, const interval &b);
bool operator >= (const lx_interval &a, const interval &b);
bool operator >  (const interval &a, const lx_interval &b);
bool operator >= (const interval &a, const lx_interval &b);

bool operator <  (const real &a, const lx_interval &b);
bool operator <= (const real &a, const lx_interval &b);
bool operator >  (const lx_interval &a, const real &b);
bool operator >= (const lx_interval &a, const real &b);

bool operator <  (const l_real &a, const lx_interval &b);
bool operator <= (const l_real &a, const lx_interval &b);
bool operator >  (const lx_interval &a, const l_real &b);
bool operator >= (const lx_interval &a, const l_real &b);

bool operator <  (const lx_real &a, const lx_interval &b);
bool operator <= (const lx_real &a, const lx_interval &b);
bool operator >  (const lx_interval &a, const lx_real &b);
bool operator >= (const lx_interval &a, const lx_real &b);
```

`a < b` bedeutet, dass `a` im Innern von `b` liegt und `a <= b` bedeutet $a \in b$ bzw. $a \subseteq b$.
`a > b` bedeutet, dass `b` im Innern von `a` liegt und `a >= b` bedeutet $b \in a$ bzw. $b \subseteq a$.

5.1.4 Vergleichsoperatoren

In der Klasse `lx_interval` sind die Operatoren $==$ und $!=$ definiert für Operanden vom Typ

```
lx_interval, l_interval, interval, lx_real, l_real, real;
```

wobei wenigstens einer der Operanden vom Typ `lx_interval` sein muss.

5.1.5 Arithmetische Operatoren

In der C-XSC Klasse `lx_interval` sind neben den beiden unitären Operatoren $+, -$ mit nur jeweils einem Operanden vom Typ `lx_interval` zusätzlich die arithmetischen Operatoren $\{+, -, *, /\}$ definiert, wobei wenigstens einer der beiden Operanden vom Typ `lx_interval` sein muss. Die Zuweisungsoperatoren $\{+=, -=, *=, /=\}$ sind ebenfalls implementiert.

5.1.6 Der Negationsoperator !

In der Klasse `lx_interval` ist der Operator

```
bool operator ! (const lx_interval& a);
```

definiert. Er liefert 1, wenn $0 \in$ `a.li`, sonst wird 0 zurückgegeben.

5.1.7 Ein- und Ausgabefunktionen

Zur Eingabe von Objekten des Typs `lx_interval` stehen die folgenden Operatoren zur Verfügung:

```
string & operator >> (std::string &s, lx_interval &a);
void operator >> (const std::string &s, lx_interval &a);
void operator >> (const char *s, lx_interval &a);
istream & operator >> (std::istream &s, lx_interval &a);
```

Mit den drei ersten Operatoren wird ein String `s` der Form

```
{ -345678 , [0.1e-4,0.1e-4] }
```

in eine Variable vom Typ `lx_interval` kopiert, wobei -345678 als Exponent zur Basis **10** interpretiert wird. Der vierte Operator kopiert die Tastatureingabe mit dem gleichen Format ebenfalls in eine Variable `a` vom Typ `lx_interval`. Die Variablen `a` sind dabei stets Einschließungen der eingegebenen String-Intervalle.

Zur Ausgabe von Objekten des Typs `lx_interval` stehen folgende Operatoren zur Verfügung:

```
ostream& operator << (ostream& s, const lx_interval& a);
string & operator << (string &s,  const lx_interval& a);
```

Der erste ermöglicht die Ausgabe in dezimaler Form und der zweite schreibt ein Objekt a vom Typ `lx_interval` mit seinem Zweier-Exponenten $a.ex$ in eine Zeichenkette `s`. Weitere Informationen findet man im Abschnitt 5.2.5 ab Seite 32.

5.1.8 Standardfunktionen

Die Argumente x, y der 42 folgenden Standardfunktionen sind nicht zu breite Intervalle vom Typ `lx_interval`. Beispiele findet man bei den Seitenangaben der letzten Spalte.

Standardfunktionen vom Typ `lx_interval`

Funktionsterm	C-XSC Name	Informationen	Seite
$\|x\|$	abs(x)	Intervall der Beträge	20
x^2	sqr(x)	Intervall der Quadrate	40
$\sqrt{x}$	sqrt(x)	Inf$(x) \geq 0$	40
$\sqrt[n]{x}$	sqrt(x, n)	$2 \leq n \leq +2147483647$	59
$\sqrt{1+x} - 1$	sqrtp1m1(x)	Inf$(x) \geq -1$	119
$\sqrt{1-x^2}$	sqrt1mx2(x)	Sup$(\|x\|) \leq 1$	89
$\sqrt{1+x^2}$	sqrt1px2(x)	Intervall x beliebig	80
$\sqrt{x^2-1}$	sqrtx2m1(x)	Inf$(\|x\|) \geq 1$	92
$\sqrt{x^2+y^2}$	sqrtx2y2(x, y)	x, y fast beliebig	145
x^n	power(x, n)	x fast beliebig, $n \in \mathbb{Z}$	42
x^y	pow(x, y)	x, y fast beliebig	58
$(1+x)^y$	xp1_pow_y(x, y)	x, y fast beliebig	60
e^x	exp(x)	Sup$(\|x\|) \leq 1.488521882 \cdot 10^9$	44
2^x	exp2(x)	Sup$(\|x\|) \leq 1.488521882 \cdot 10^9$	44
10^x	exp10(x)	Sup$(\|x\|) \leq 1.488521882 \cdot 10^9$	44
$e^x - 1$	expm1(x)	Sup$(\|x\|) \leq 1.488521882 \cdot 10^9$	61
$\ln(x)$	ln(x)	Inf$(x) > 0$	48
$\ln(1+x)$	lnp1(x)	Inf$(x) > -1$	56
$\ln(\sqrt{x^2+y^2})$	ln_ sqrtx2y2(x, y)	Inf$(\|x\|)$ + Inf$(\|y\|) > 0$	150
$\sin(x)$	sin(x)	Sup$(\|x\|) < 10^{308}$	65
$\sin(n\pi + x)$	sin_n(x, n)	Sup$(\|x\|) < 10^{308}$, $n \in \mathbb{Z}$	74
$\cos(x)$	cos(x)	Sup$(\|x\|) < 10^{308}$	75
$\cos((n+1/2)\pi + x)$	cos_n(x, n)	Sup$(\|x\|) < 10^{308}$, $n \in \mathbb{Z}$	76
$\tan(x)$	tan(x)	Sup$(\|x\|) < 10^{308}$	77
$\cot(x)$	cot(x)	Sup$(\|x\|) < 10^{308}$	79

Standardfunktionen vom Typ `lx_interval` (Fortsetzung)

Funktionsterm	C-XSC Name	Informationen	Seite
$\arcsin(x)$	asin(x)	Sup$(\|x\|) \leq 1$	95
$\arccos(x)$	acos(x)	Sup$(\|x\|) \leq 1$	98
$\arctan(x)$	atan(x)	Intervall x beliebig	82
arccot (x)	acot(x)	Intervall x beliebig	100
$\sinh(x)$	sinh(x)	Sup$(\|x\|) \leq 1.488521882 \cdot 10^9$	105
$\cosh(x)$	cosh(x)	Sup$(\|x\|) \leq 1.488521882 \cdot 10^9$	109
$\tanh(x)$	tanh(x)	Intervall x beliebig	111
$\coth(x)$	coth(x)	Intervall x beliebig, $x \neq 0$	115
arsinh (x)	asinh(x)	Intervall x beliebig	122
arcosh (x)	acosh(x)	Inf$(x) \geq 1$	126
arcosh $(1 + x)$	acoshp1(x)	Inf$(x) \geq 0$	130
artanh (x)	atanh(x)	Inf$(x) > -1,$ Sup$(x) < +1$	133
artanh $(1 - x)$	atanh1m(x)	Inf$(x) > 0,$ Sup$(x) < 2$	136
artanh $(-1 + x)$	atanhm1p(x)	Inf$(x) > 0,$ Sup$(x) < 2$	138
arcoth (x)	acoth(x)	Intervall x beliebig, $x \neq 0$	139
arcoth $(+1 + x)$	acothp1(x)	Inf$(x) > 0$	142
arcoth $(-1 - x)$	acothm1m(x)	Inf$(x) > 0$	144

Tabelle 1: Standardfunktionen vom Typ `lx_interval`

Hinweise:

1. Der relative Durchmesser eines Intervalls x sollte bei einer sinnvollen Anwendung einer Intervall Staggered-Arithmetik nicht zu groß gewählt werden, vgl. dazu auch die Bedingung (4) auf Seite 7.

2. 'Intervall x beliebig' bedeutet nicht, dass x beliebig breit sein kann, sondern dass etwa folgendes gilt: $-2^{+2147483647} <$ Inf$(x) \leq$ Sup$(x) < 2^{+2147483647}$. Genauere Hinweise findet man bei den Beispielen auf den entsprechenden Seiten mit den in der Tabelle angegebenen Seitenzahlen.

3. 'Intervall x fast beliebig' bedeutet ebenfalls nicht, dass x beliebig breit sein kann, sondern dass etwa folgendes gilt: $-2^{+2147482625} <$ Inf$(x) \leq$ Sup$(x) < 2^{+2147482625}$.

5.1.9 Intervallkonstanten

Intervallkonstenten vom Typ `lx_interval`			
Wert	**C-XSC Name**	**Wert**	**C-XSC Name**
π	Pi_lx_interval()	$\sqrt{2}$	Sqrt2_lx_interval()
$\pi/3$	Pid3_lx_interval()	$\sqrt{3}$	Sqrt3_lx_interval()
$1/\pi$	Pir_lx_interval()	$\sqrt{5}$	Sqrt5_lx_interval()
$\sqrt{\pi}$	SqrtPi_lx_interval()	$\sqrt{7}$	Sqrt7_lx_interval()
$1/\sqrt{\pi}$	SqrtPir_lx_interval()	$\ln(2)$	Ln2_lx_interval()
$\sqrt{2\pi}$	Sqrt2Pi_lx_interval()	$\ln(10)$	Ln10_lx_interval()
$1/\sqrt{2\pi}$	Sqrt2Pir_lx_interval()	$1/\ln(2)$	Ln2r_lx_interval()
$\ln(\pi)$	LnPi_lx_interval()	EulerGamma $= 0.5772...$	EulerGamma_lx_interval()
$\ln(2\pi)$	Ln2Pi_lx_interval()	Catalan $= 0.9159...$	Catalan_lx_interval()
e	E_lx_interval()	$1 + 2^{-2097}$	One_p_lx_interval()
e^{π}	EpPi_lx_interval()	$1 - 2^{-2097}$	One_m_lx_interval()

Tabelle 2: Intervallkonstanten vom Typ `lx_interval`

Die angegebenen 22 Intervallkonstanten sind in der Datei `lx_imath.hpp` deklariert und in `lx_imath.cpp` mit der Präzision `stagprec` $= 39$ definiert, so dass etwa 624 korrekte Dezimalziffern zur Einschließung der Konstanten zur Verfügung stehen.

Hinweise:

- $\pi -$ `Inf(Pi_lx_interval())` $= +4.1060273917\ldots \cdot 10^{-631}$ gilt bei maximaler Präzision `stagprec` $= 39$.
- `Sup(Pi_lx_interval()/2)` $- \pi/2 = +1.4565141001\ldots \cdot 10^{-632}$, dies gilt bei maximaler Präzision `stagprec` $= 39$. In `Sup(Pi_lx_interval()/2)` wird die Division durch 2 realisiert durch `times2pown(u,-1)`, wobei in `u` vom Typ `lx_interval` die cxsc Konstante `Pi_lx_interval()` gespeichert ist.
- $\pi/2 -$ `Inf(Pi_lx_interval()/2)` $= +2.053013695\ldots \cdot 10^{-631}$, dies gilt bei maximaler Präzision `stagprec` $= 39$. In `Inf(Pi_lx_interval()/2)` wird die Division durch 2 realisiert durch `times2pown(u,-1)`, wobei in `u` vom Typ `lx_interval` die cxsc Konstante `Pi_lx_interval()` gespeichert ist.
- `Sup(Sqrt2_lx_interval())` $- \sqrt{2} = +5.8078220004\ldots \cdot 10^{-632}$, dies gilt bei maximaler Präzision `stagprec` $= 39$.

Weitere Hinweise:

- Die kleinste positive Zahl x vom Typ `lx_real` besitzt die Darstellung

$$x = \texttt{lx\_real}(2^{-2147483647}, \texttt{l\_real}(2^{-1074})) = 2^{-2147484721}.$$

Die punktförmige Intervallkonstante `One_p_lx_interval()` aus Tabelle 2 schließt für $1 \leq \texttt{stagprec} \leq 38$ die kleinste Maschinenzahl vom Typ `lx_real` ein, die größer 1.0 ist, d.h. für jede Maschinenzahl $x > 1$ gilt

(6) $$x \geq 1 + 2^{-2097}, \quad \text{falls } x > 1 \text{ und } x \text{ vom Typ lx\_real ist.}$$

Um (6) nachzuweisen, betrachten wir zunächst die Präzision $\texttt{stagprec} = 1$, und die kleinste Maschinenzahl $x > 1$ kann mit einem zulässigen Exponenten $ex \in \mathbb{Z}$ dargestellt werden als

$$x = \texttt{lx\_real}(2^{-ex}, \texttt{l\_real}(\texttt{succ}(2^{ex}))) = 2^{-ex} \cdot (2^{ex} + 2^{ex-52}) = 1 + 2^{-52},$$

womit (6) bestätigt ist. Für $\texttt{stagprec} \geq 2$ kann eine Maschinenzahl $x > 1$ dargestellt werden als

$$x = \texttt{lx\_real}(-1023, \texttt{l\_real}(2^{+1023}) + 2^{-1074}) = 1 + 2^{-1023-1074} = 1 + 2^{-2097}.$$

Dieser Wert kann nicht weiter verkleinert werden, da der Summand $\texttt{minreal} = 2^{-1074}$ schon minimal gewählt wurde. Eine Verkleinerung von x wäre dann nur noch möglich, wenn man die `l_real`-Mantisse z.B. durch $\texttt{l\_real}(2^{1024}) + 2^{-1074}$ weiter nach oben verbreitern könnte, aber wegen $\texttt{MaxReal} < 2^{1024}$ ist dies natürlich nicht möglich. Falls man jedoch

(7) $$x = \texttt{lx\_real}(-1024, \texttt{l\_real}(\texttt{MaxReal}) + 2^{-1074})$$

wählt, so wird wegen $2^{-1024} \cdot \texttt{MaxReal} < 1$ die Maschinenzahl x kleiner als der Dezimalwert 0.99999999999999988898 und bleibt damit deutlich kleiner als 1. Damit haben wir die Ungleichung in (6) bestätigt. Wir zeigen jetzt noch

(8) $$x \leq 1 - 2^{-2097}, \quad \text{falls } x < 1 \text{ und } x \text{ vom Typ lx\_real ist.}$$

Zunächst könnte man analog zu (7) mit $m := \texttt{mant}(\texttt{MaxReal})$ den `l_real`-Teil t wie folgt konstruieren

$$t[1] = \texttt{MaxReal};\ \ t[2] = m \cdot 2^{+971};\ \ t[3] = m \cdot 2^{+918};\ \ \ldots$$

und t dabei so wählen, dass gilt: $2^{-1024} \cdot (t + 2^{-1074}) = 1$. Mit $2^{-1024} \cdot t = 1 - 2^{-2098}$ hätte man dann die größte Maschinenzahl $x = 2^{-1024} \cdot t < 1$. Die so konstruierte `l_real`-Zahl t hat jedoch den Nachteil, dass mit ihr die Grundoperationen, also z.B. $t = t + 0.0$, nicht mehr alle durchfürbar sind, so dass die größte Maschinenzahl $x < 1$, mit der wirklich gerechnet werden kann, wie oben gegeben ist durch

$$x = \texttt{lx\_real}(-1023, \texttt{l\_real}(2^{+1023}) - 2^{-1074}) = 1 - 2^{-1023-1074} = 1 - 2^{-2097},$$

wobei dieses x durch die punktförmige Intervallkonstante `One_m_lx_interval()` aus Tabelle 2 eingeschlossen wird.

5.1.10 Friend-Funktionen

In der Klasse `lx_interval` sind die folgenden friend-Funktionen deklariert:

1. `int StagPrec(const lx_interval &a);`
   liefert die Präzision des Objekts `a`, die gegeben ist durch die Präzision von `a.li`.

2. `int expo(const lx_interval &a);`
   liefert den Zweier-Exponenten `a.ex` des Objekts `a`.

3. `l_interval li_part(const lx_interval &a);`
   liefert bzgl. des Objekts `a` das Intervall `a.li` vom Typ `l_interval`.

4. `void scale_down(lx_interval &a);`
   `scale_down(a);` liefert ohne Wertänderung ein neues Objekt `a`, wobei das neue `a.li` möglichst weit nach unten skaliert wird, ohne dass dabei binäre Stellen verloren gehen.

5. `void scale_up(lx_interval &a);`
   `scale_up(a);` liefert ein neues Objekt `a`, welches das alte Objekt `a` garantiert einschließt, wobei das alte `a.li` so weit nach oben skaliert wird, dass mit dem neuen `a.li` die Summe `a.li` $+$ `a.li` gerade noch keinen Overflow erzeugt.

6. `bool point_intv(const lx_interval &a);`
   liefert die Information, ob das Objekt `a` ein Punktintervall ist.

7. `void times2pown(lx_interval &a, int n) throw();`
   multipliziert `a` rundungsfehlerfrei mit 2^n; $\quad -2147483648 \leq n \leq +2147483647$.

8. `void times2pown_neg(lx_interval &a, int n);`
   Für Intervalle `a`, welche die Null nicht in ihrem Innern enthalten, wird eine Einschließung von `a` $\cdot\, 2^n$ ohne integer-Überlauf berechnet, falls $n \leq 0$.

9. `l_real RelDiam( const lx_interval &a );`
   `lr = RelDiam(a);` liefert den relativen Durchmesser von `a.li`.

10. `lx_real diam(const lx_interval&a);`
    liefert den absoluten Durchmesser von `a`.

11. `lx_real mid(const lx_interval&a);`
    liefert eine Approximation des Mittelpunkts von `a`.

12. `lx_real Inf(const lx_interval &a);`
    liefert das Infimum des Intervalls `a`.

13. `lx_real Sup(const lx_interval &a);`
    liefert das Supremum des Intervalls `a`.

14. `lx_interval abs(const lx_interval &a)`
    `b = abs(a);` liefert ein Intervall `b` vom Typ `lx_interval`, das die Absolutbeträge $|r|$ aller Elemente $r \in$ `a` optimal einschließt.

15. `lx_interval adjust(const lx_interval &a);`
    passt `a.li` an die aktuelle durch `stagprec` vorgegebene Präzision an.

16. `bool IsEmpty(const lx_interval &a);`
    liefert 1, wenn `Inf(a)` $>$ `Sup(a)`, sonst wird 0 zurückgegeben.

5.1.11 Außerhalb der Klasse `lx_interval` deklarierte Funktionen

Für Testrechnungen sind die folgenden vier in `lx_interval.hpp` deklarierten Funktionen nützlich, da sie Punktintervalle oder nur leicht aufgeblähte Intervalle vom Typ `l_interval` liefern, wobei nahezu alle binären Mantissen-Bits von `a.li` gesetzt sind.

1. `l_interval point_max(void);`
   `a = point_max();` liefert ein Punktintervall mit `a.ex` $= 1020$, wobei in `a.li` fast alle binären Mantissen-Bits gesetzt sind.

2. `l_interval point_any(int n);`
   `a = point_any(n);` liefert ein Punktintervall `a` mit dem Zweier-Exponenten $n,\ -1074 \leq n \leq 1020,$ wobei nahezu alle Mantissen-Bits von `a.li` gesetzt sind.

3. `l_interval wide_max(void);`
   `a = wide_max();` liefert ein leicht aufgeblähtes Intervall `a` mit `a.ex` $= 1020$, wobei in `a.li` fast alle binären Mantissen-Bits gesetzt sind.

4. `l_interval wide_any(int n);`
   `a = wide_any();` liefert ein leicht aufgeblähtes Intervall `a` mit dem Zweier-Exponenten $n, -1074 \leq n \leq 1020$, wobei nahezu alle Mantissen-Bits von `a.li` gesetzt sind.

Es folgen weitere Funktionen und Operatoren, die in `lx_interval.hpp` außerhalb der Klasse `lx_interval` deklariert sind. Zur Beschreibung dienen folgende Mengen:
$T :=$ {`l_interval, interval, lx_real, l_real, real`},
$R :=$ {`lx_real, l_real, real`}.

1. `lx_interval expo2zero(const lx_interval &a);`
   `b = expo2zero(a);` liefert `a` $\subseteq$ `b`, mit `b.ex` $= 0$, falls kein Overflow auftritt.

2. `int Disjoint(const lx_interval& a,const lx_interval&b);`
   `Disjoint(a,b)` liefert 1, wenn `a` $\cap$ `b` $= \emptyset$, sonst wird 0 zurückgegeben.

3. `lx_interval operator | (const lx_interval& a,`
   `const lx_interval& b);`
   `c = a | b;` liefert mit `c` die konvexe Hülle von `a,b`. Ist einer der Operanden vom Typ `lx_interval`, so kann der Typ des anderen Operanden aus T sein. Es können auch beide Operanden vom Typ `lx_real` gewählt werden. Zusätzlich stehen auch die Operatoren `|=` zur Verfügung, wobei der jeweils erste Operand vom Typ `lx_interval` sein muss.

4. `lx_interval operator & (const lx_interval& a,`
   `const lx_interval& b);`
   `c = a & b;` liefert mit `c` den Durchschnitt von `a,b`. Gilt `a` $\cap$ `b` $= \emptyset$, so erfolgt eine Fehlermeldung. Ist einer der Operanden vom Typ `lx_interval`, so kann der Typ des anderen Operanden aus T sein. Es stehen auch die Operatoren `&=` zur Verfügung, wobei der jeweils erste Operand vom Typ `lx_interval` sein muss.

5. `lx_interval & SetInf(lx_interval& a, const lx_real& r);`
   `c = SetInf(a,r);` liefert mit `c` vom Typ `lx_interval` das Intervall `a`, jedoch mit dem neuen Infimum `r`. Der Operand `r` kann alle Typen aus R annehmen. Gilt `Inf(c)` $=$ `r` $>$ `Sup(c)` $=$ `Sup(a)`, so erfolgt eine Fehlermeldung.

6. `lx_interval & SetSup(lx_interval& a, const lx_real& r);`
   `c = SetSup(a,r);` liefert mit `c` vom Typ `lx_interval` das Intervall `a`, jedoch mit dem neuen Supremum `r`. Der Operand `r` kann alle Typen aus R annehmen. Gilt `Inf(c)` $=$ `Inf(a)` $>$ `Sup(c)` $=$ `r`, so erfolgt eine Fehlermeldung.

7. `int in(const lx_interval &a, const lx_interval &b);`
   `int j = in(a,b);` liefert `j` $= 1$, wenn `a` ganz im Innern von `b` enthalten ist, sonst wird `j` $= 0$ zurückgegeben. Der Typ des ersten Arguments `a` kann ein Element der folgenden Menge sein:
   `{ lx_interval, l_interval, interval }.`

8. `int in (const lx_real &a, const lx_interval &b);`
   `int j = in(a,b);` liefert `j` $= 1$, wenn `a` in `b` enthalten ist, d.h. `a` $\in$ `b`, sonst wird `j` $= 0$ zurückgegeben. Der Typ des ersten Arguments `a` kann Element der folgenden Menge sein: `{ lx_real, l_real, real }.`

9. `lx_interval Blow(const lx_interval&x, const real& eps);`
   `y = Blow(x,eps);` liefert mit `y` eine ε-Aufblähung des ersten Arguments `x`.

10. `lx_real AbsMin (const lx_interval &x);`
    `R = AbsMin(x);` liefert das Minimum $\langle[x]\rangle$ aller Betragselemente aus $[x]$.

11. `lx_real AbsMax (const lx_interval &x);`
    `R = AbsMax(x);` liefert das Maximum $|[x]|$ aller Betragselemente aus $[x]$.

5.2 Algorithmen der Operatoren und Standardfunktionen

5.2.1 Der Additionsoperator $+$

Es sind zwei Objekte der Klasse `lx_interval` zu addieren:

$$(a.ex, a.li) + (b.ex, b.li) = 2^{a.ex} \cdot a.li + 2^{b.ex} \cdot b.li,$$

wobei $a.li$ und $b.li$ vom Typ `l_interval` sind.

Ist einer der Operanden gleich Null, so ist der jeweils andere Operand als Summe zurückzugeben. Wir nehmen daher an, dass jetzt beide Operanden von Null verschieden sind.

Algorithmus:

1. Bestimme zuerst den betragsmäßig größten Operanden; dieser sei: $(a.ex, a.li)$.

2. Beim Objekt $(a.ex, a.li)$ wird das Intervall a.li vom Typ `l_interval` so mit 2^p, $p = 1022 - \texttt{expo\_gr}(a.li)$, multipliziert, dass $\alpha := 2^p \cdot a.li$ dicht am Überlaufbereich liegt:

   $$(a.ex, a.li) = (a.ex - p, 2^p \cdot a.li) = (n, \alpha), \quad \alpha \text{ vom Typ } \texttt{l\_interval}.$$

   Mit dieser Skalierung erreicht man, dass bei der nachfolgenden Addition $\alpha + \beta$ möglichst viele Stellen zur Verfügung stehen. `expo_gr`$(a.li)$ ist dabei der Zweierexponent des betragsmäßig größten Summaden von $a.li$. Mit der obigen Wahl von 1022 anstelle von 1023 oder 1024 wird erreicht, dass $\alpha + \beta$ stets ohne Überlauf berechnet wird.

3. Die Addition $(n, \alpha) + (b.ex, b.li)$ ist nur möglich, wenn $b.li$ so mit 2^p multipliziert wird, dass $b.ex - p = n$ erfüllt wird, d.h. für $p = b.ex - n$ und $\beta := 2^p \cdot b.li$ gilt:

   $$(n, \alpha) + (b.ex, b.li) = (n, \alpha) + (b.ex - p, 2^p \cdot b.li) = (n, \alpha) + (n, \beta) = (n, \alpha + \beta).$$

Mit diesem Algorithmus erreicht man maximale Genauigkeit, wenn die Addition $\alpha + \beta$ mit der Präzision `stagprec` $= 39$ erfolgt, da dann etwa $16 \cdot 39 = 624$ Dezimalziffern zur Verfügung stehen.

Der $-$ Operator für zwei Objekte der Klasse `lx_interval` wird mit Hilfe des $+$ Operators über die Beziehung

$$(a.ex, a.li) - (b.ex, b.li) = (a.ex, a.li) + (b.ex, -b.li).$$

realisiert.

Im ersten Programm-Beispiel ist mit den Punktintervallen $\boldsymbol{a} = 1.12285\ldots \cdot 10^{+307}$ und $\boldsymbol{b} = 5.6689\ldots \cdot 10^{+278}$, jeweils vom Typ `l_interval`, folgender Ausdruck einzuschließen:

$$\boldsymbol{y} := (2^{1200} \cdot \boldsymbol{a} + \boldsymbol{b}) - 2^{1200} \cdot \boldsymbol{a}, \tag{9}$$

wobei $\boldsymbol{a}$ und $\mathbf{b}$ durch die Funktionen

`l_interval point_max(void)` und `l_interval point_any(int n)`

realisiert werden. Bei jeder Präzision gilt: $a_i, b_i \neq 0, \quad i = 1, 2, \ldots$ `stagprec` $+ 1$.

Das nachfolgende C-XSC Programm

```
// Programm  lx_test1.cpp  zum Test der -,+ Operatoren:

#include <iostream>       // Wegen cout
#include "lx_interval.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39; // Präzision: ca. 16*39 = 624 Dezimalziffern
    l_interval a(point_max()),b(point_any(926)),yi;
    lx_interval A,Y;

    A = lx_interval(1200,a);  // Konstruktor-Aufruf
    Y = (A + b) - A;
    yi = Y;  // Zuweisung an den Typ l_interval;

    cout << "RelDiam(yi) = " << RelDiam(yi) << endl;
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;

    cout << "yi = " << yi << endl;
}
```

liefert die folgenden Ergebnisse: `RelDiam(yi) = 1.8757937551E-0241` und

$$\boldsymbol{y} \in \texttt{yi} = \underbrace{5.668972\ldots 44608495}_{240\ \text{Dez.-Ziffern}}{}^{112\ldots}_{006\ldots} \cdot 10^{+278}.$$

Hinweise:

1. `point_any(926)` liefert ein Punktintervall $\boldsymbol{b}$ mit der Zweierpotenz 2^{+926}.
2. $\texttt{A} = 2^{+1200} \cdot \boldsymbol{a} = 2^{+1200} \cdot (1.1228\ldots \cdot 10^{+307})$ ist nicht in `l_interval` darstellbar, so dass der Ausdruck rechts in (9) nur mit der Klasse `lx_interval` ausgewertet werden kann.
3. Die garantierte Einschließung von $\boldsymbol{y}$ kann mit der Anweisung `yi = Y;` wieder in `l_interval` mit 240 korrekten Dezimalziffern dargestellt werden.
4. Verkleinert man $\texttt{stagprec} = 39$ auf z.B. 35, so wird die Einschließung von $\boldsymbol{y}$ nur noch mit 179 korrekten Dezimalziffern berechnet.
5. Die Genauigkeit der Einschließung nimmt weiter ab, wenn man für `A` anstelle des Punktintervalls ein nur leicht aufgeblähtes Intervall wählt.

5.2.2 Der Multiplikationsoperator $*$

Für zwei Objekte a, b der Klasse `lx_interval` ist der Operator * zu implementieren, so dass bei vorgegebener Präzision mit $(a.ex, a.li) * (b.ex, b.li)$ eine Einschließung von

$$(10) \qquad (a.ex, a.li) \cdot (b.ex, b.li) = (2^{a.ex} \cdot a.li) \cdot (2^{b.ex} \cdot b.li) = 2^{a.ex+b.ex} \cdot (a.li \cdot b.li),$$

in maximaler Genauigkeit berechnet wird. $a.li$ und $b.li$ sind dabei Staggered Intervalle vom Typ `l_interval`, die als von Null verschieden vorausgesetzt werden. Mit exa und exb werden die Zweier-Exponenten der betragsmäßig größten IEEE-Zahlen a_j, b_k der Staggered Intervalle $a.li$ bzw. $b.li$ bezeichnet.

Algorithmus:

1. Im ersten Schritt werden $a.li$ und $b.li$ jeweils möglichst weit nach unten skaliert, ohne dabei jedoch einen zusätzlichen Genauigkeitsverlust beider Operanden zu bewirken. Bei dieser Skalierung darf der mathematische Wert beider Operanden nicht verändert werden. Nach der Skalierung werden die Operanden wie in (10) bezeichnet. Für eine optimale Einschließung des Produktes ist diese Skalierung notwendig, um einen vorzeitigen Überlauf bei der Berechnung von $(a.li \cdot b.li)$ zu vermeiden.

2. Wir betrachten zunächst den Fall $exa + exb \leq 1022$, so dass ein Überlauf nicht auftreten kann.

   - Im Fall $exa < 0$ wird der erste Operand ohne Wertänderung so nach oben skaliert, dass $exa = 0$ gilt, und der zweite Operand wird ohne Wertänderung so nach oben skaliert, dass danach $exb = 1022$ gilt.
   - Im Fall $exa \geq 0$ bleibt der erste Operand unverändert und der zweite Operand wird so nach oben skaliert, dass danach sein Zweier-Exponent durch $exb = 1022 - exa$ gegeben ist.

   In beiden Fällen gilt nach der Skalierung: $exa + exb = 1022$, so dass für das Produkt $(a.li \cdot b.li)$ die maximale Stellenzahl zur Verfügung steht, aber ein Überlauf stets vermieden wird.

3. Wir betrachten jetzt den Fall $exa + exb > 1022$, so dass ein möglicher Überlauf durch geeignete Skalierungen der Operanden nach unten zu vermeiden ist. Die bei diesen Skalierungen auftretenden Aufblähungen der Operandenintervalle sind auf ein Minimum zu reduzieren, wenn maximale Genauigkeit erreicht werden soll. Wir betrachten zusätzlich $exa > exb$. Der analoge Fall $exa \leq exb$ bleibt dem Leser überlassen. Es gelte: $d := exa + exb - 1022 > 0$ und $D := exa - exb > 0$.

   - Im Fall $d \leq D$ wird der erste, betragsmäßig größere Operand um den Faktor 2^{-d} nach unten skaliert. Daraus ergibt sich: $exa = 1022 - exb$. Der zweite, betragsmäßig kleinere Operand bleibt unverändert. Aus $d \leq D$ folgt direkt $exb \leq 511$, und damit: $exa \geq 511$. Dadurch bestimmt der erste Operand mit seiner größeren Genauigkeit die maximale Genauigkeit des Produkts.

- Im Fall $d > D$ werden beide Operanden so nach unten skaliert, dass anschließend die Zweier-Exponenten der jeweils betragsmäßig größten IEEE-Zahlen beide den Wert 511 annehmen. Dadurch erhält auch das Produkt die gleiche maximale Genauigkeit der beiden nach unten skalierten Operanden.

Mit dem obigen Algorithmus[8] erhält man mit `stagprec` $= 39$ die größtmögliche Genauigkeit für die Einschließung von $(a.ex, a.li) \cdot (b.ex, b.li) \subset a * b$.

Im zweiten Programmbeispiel ist mit den Punktintervallen $\boldsymbol{a} = 6.66384 \ldots \cdot 10^{+240}$ und $\boldsymbol{b} = 2.58063546 \ldots \cdot 10^{+120}$, jeweils vom Typ `l_interval`, folgender Ausdruck einzuschließen:

$$\boldsymbol{y} := (2^{-40000} \cdot \boldsymbol{a}) \cdot (2^{-60000} \cdot \boldsymbol{b}), \tag{11}$$

wobei $\boldsymbol{a}$ und $\mathbf{b}$ mit Hilfe der Funktion

```
l_interval point_any(int n);
```

realisiert werden. Bei jeder Präzision gilt dabei: $a_i, b_i \neq 0$, $i = 1, 2, \ldots$ `stagprec` $+ 1$. Das nachfolgende C-XSC Programm

```
// Programm  lx_test2.cpp  zum Test des * Operators:

#include <iostream>       // Wegen cout
#include "lx_interval.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;  // Präzision: ca. 16*39 = 624 Dezimalziffern
    l_interval a(point_any(800);),b(point_any(400));
    lx_interval A,B,Y;

    A = lx_interval(-40000,a);  // Konstruktor-Aufruf
    B = lx_interval(-60000,b);  // Konstruktor-Aufruf
    Y = A * B;
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "Y = " << Y << endl;
}
```

liefert für y die optimale Einschließung

$$\boldsymbol{y} \in 2^{-99822} \cdot \underbrace{4.488615046109 \ldots 499767875216}_{510 \text{ Dez.-Ziffern}} \begin{matrix} 532\ldots \\ 022\ldots \end{matrix} \cdot 10^{+307}.$$

mit maximal 511 korrekten Dezimalziffern. Es ist zu beachten, dass sowohl `A` und `B` als auch das Produkt `Y = A * B` nicht als Werte vom Typ `l_interval` darstellbar sind.

[8] Der Quelltext des * Operators befindet sich im Anhang ...

5.2.3 Der Divisionsoperator /

Für zwei Objekte a, b der Klasse `lx_interval` ist der Operator / zu implementieren, so dass bei vorgegebener Präzision mit $(a.ex, a.li)/(b.ex, b.li)$ eine Einschließung von

$$\frac{(a.ex, a.li)}{(b.ex, b.li)} = \frac{(2^{a.ex} \cdot a.li)}{(2^{b.ex} \cdot b.li)} = 2^{a.ex-b.ex} \cdot \frac{a.li}{b.li}, \tag{12}$$

in maximaler Genauigkeit berechnet wird. $a.li$ und $b.li$ sind dabei Staggered Intervalle vom Typ `l_interval`, die als von Null verschieden vorausgesetzt werden. Mit exa und exb werden die Zweier-Exponenten der betragsmäßig größten IEEE-Zahlen a_j, b_k der Staggered Intervalle $a.li$ bzw. $b.li$ bezeichnet.

Algorithmus:

1. Im ersten Schritt werden $a.li$ und $b.li$ jeweils möglichst weit nach unten skaliert, ohne dabei jedoch einen zusätzlichen Genauigkeitsverlust beider Operanden zu bewirken. Bei dieser Skalierung darf der mathematische Wert beider Operanden nicht verändert werden. Nach der Skalierung werden die Operanden wie in (12) bezeichnet. Wie bei der Multiplikation ist diese Skalierung auch für eine optimale Einschließung des Quotienten notwendig, um einen vorzeitigen Überlauf bei der Berechnung von $(a.li/b.li)$ zu vermeiden.

2. Im Zähler a wird ohne Wertänderung von a das Intervall $a.li$ so nach oben skaliert, dass danach gilt[9]: $exa = 1022$.

3. Ist $b.li$ ein Punktintervall, so wird wie folgt verfahren:

   - Im Fall $0 \leq exb \leq 511$ bleibt der Nenner b unverändert.
   - Im Fall $exb < 0$ wird ohne Wertänderung von b das Intervall $b.li$ so nach oben skaliert, dass danach gilt: $exb = 0$.
   - Im Fall $exb > 511$ wird das Intervall $b.li$ so nach unten skaliert, dass danach gilt: $exb = 511$. Bei dieser Skalierung wird $b.li$ natürlich mit einem Verlust an korrekten Ziffern leicht aufgebläht. Wegen $exb = 511$ bleibt für eine optimale Einschließung von a/b jedoch eine ausreichende Genauigkeit von $b.li$ erhalten.

4. Ist $b.li$ ein echtes Intervall, dann wird $b.li$ so skaliert, dass danach gilt: $exb = 511$.

5. Im letzten Schritt wird die Differenz $a.ex - b.ex$ und die Einschließung $a.li \diamondsuit b.li$ der skalierten Operanden berechnet.

Hinweise:

1. Ist einer der Operanden a, b ein echtes Intervall, so kann der relative Durchmesser des Quotienten a/b nicht kleiner werden als das Maximum der relativen Durchmesser beider Operanden. Ist z.B. der Zähler a ein Punktintervall und der Nenner b ein

[9] Im Fall $exa \geq 1023$ wird nach unten skaliert, wobei es nur zu einer minimalen Intervallaufblähung kommen kann.

echtes Intervall mit 600 korrekten Binärziffern, so kann die Einschließung von a/b höchstens mit 600 korrekten Binärziffern berechnet werden.

2. Der obige Algorithmus liefert für a/b optimale Genauigkeit, d.h. eine Maximalzahl korrekter Binärstellen. Betrachtet man z.B. den Fall, dass a, b Punktintervalle sind, mit $exb = 511 + k,\ k > 0$, so wird b mit etwa $(511 + k) + 1022 = 1533 + k$ Binärstellen dargestellt. Der Quotient $a.li/b.li$ besitzt dann den Zweier-Exponenten $1022 - (511 + k) = 511 - k$, so dass für den Quotienten nur maximal $(511 - k) + 1022 = 1533 - k$ Binärstellen Zur Verfügung stehen. Für $k = 0$ erhält man daher maximale Genauigkeit, d.h. $b.li$ muss auf $exb = 511$ runterskaliert werden.

3. Ist b ein echtes Staggered Intervall, so muss $b.li$ nach obigem Algorithmus auf $exb = 511$ skaliert werden, um optimale Genauigkeit für den Quotienten zu erreichen. Der Nachweis wird in den folgenden Abschnitt geführt.

Wir nehmen zunächst an, dass a ein Punktintervall und b ein echtes Staggered Intervall ist. Dann gilt nach Hinweis 1. auf Seite 28: a/b kann mit höchstens $\beta := exb + 1022$ korrekten Binärziffern berechnet werden.

Im Algorithmus wird $b.li$ auf $exb = 511$ skaliert, so dass für den Quotienten maximal $m := (1022 - 511) + 1022 = 1533$ Binärstellen zur Verfügung stehen. Für $\beta \leq m \iff exb \leq 511$ wird dann a/b optimal berechnet, wobei exb der Zweier-Exponent von $b.li$ vor der Skalierung ist. Bleibt der Fall $\beta > m \iff exb > 511$, d.h. $exb = 511 + k$, mit $k > 0$. Dann kann a/b nach Hinweis 1. auf Seite 28 mit höchstens $\beta := (511+k)+1022 = 1533 + k$ korrekten Binärziffern berechnet werden, und weil $a.li$ auf $exa = 1022$ skaliert wurde, stehen für den Quotienten a/b höchstens $m := 1022-(511+k)+1022 = 1533-k$ Binärstellen zur Verfügung. Optimale Genauigkeit für den Quotienten liefert daher die Forderung $\beta = m \iff 1533 + k = 1533 - k$, d.h. $k = 0$. Daher muss auch im Fall $exb > 511$ das Intervall $b.li$ auf $exb = 511$ skaliert werden.

Bleibt noch der Fall, dass auch a ein echtes Intervall ist: Im Fall $exa \geq exb$ kann dann nach Hinweis 1. a/b auch nur mit höchstens $\beta = exb+1022$ korrekten Binärziffern berechnet werden, so dass die obigen Überlegungen direkt übernommen werden können, und im Fall $exa < exb$ gelten diese Überlegungen erst recht!

Im dritten Programmbeispiel ist mit den Punktintervallen $\boldsymbol{a} = 10^{+300}$, $\boldsymbol{b} = 2^{-800}$ und $\boldsymbol{c} = 2^{-850}$ jeweils vom Typ `l_interval`, folgender Ausdruck einzuschließen:

$$\text{(13)} \qquad \begin{aligned} \boldsymbol{y} &:= \frac{10^{+300}}{2^{-800}} \cdot 2^{-850} = 10^{+300} \cdot 2^{-50} \\ &= 8.88178419700125232338905334472656 25 \cdot 10^{284}. \end{aligned}$$

Beachten Sie bitte, dass rechts in (13) die Auswertung des ersten Bruchs in der Klasse `l_interval` zum Overflow führt, obwohl das Endergebnis $\boldsymbol{y} = 10^{+300} \cdot 2^{-50}$ in einer Variablen vom Typ `l_interval` darstellbar ist. Zu beachten ist ferner, dass im Programm durch `string("[1e300,1e300]") >> a;` der Wert 10^{300} durch das Punktintervall $\boldsymbol{a}$ exakt eingeschlossen wird, so dass auch $\boldsymbol{y}$ in `l_interval` exakt darstellbar ist. Das nachfolgende C-XSC Programm

```
// Programm  lx_test3.cpp  zum Test des / Operators:

#include <iostream>       // Wegen cout
#include "lx_interval.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;  // Präzision: ca. 16*39 = 624 Dezimalziffern
    l_interval a,b,c,y;
    lx_interval A;

    string("[1e300,1e300]") >> a; // a: Punkt-Intervall
    b = comp(0.5,-799);  c = comp(0.5,-849);

    A = lx_interval(a);  // Konstruktor-Aufruf
    y = (A / b) * c;
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "y = " << y << endl;
}
```

liefert mit:

```
y = [8.88178419...7265625E+0284,8.88178419...7265625E+0284]
```

eine optimale Einschließung von $10^{+300} \cdot 2^{-50}$ durch das Punktintervall `y`.

5.2.4 Die komplexe Division

Mit den bisherigen C-XSC Programmen haben wir die vier Grundoperationen mit Operanden vom Typ `lx_interval` meist nur einzeln getestet. Die komplexe Division ist ein geeignetes Beispiel aus der Praxis, um die Leistungsfähigkeit der neuen Staggered Arithmetik mit allen vier Operanden vom Typ `lx_interval` zu testen. Mit den komplexen Zahlen

$$z = a + i \cdot b, \quad w = c + i \cdot d, \quad i = \sqrt{-1}$$

gilt für den Imaginärteil des komplexen Quotienten z/w

$$\Im(z/w) = \frac{b \cdot c - a \cdot d}{c^2 + d^2}. \tag{14}$$

Mit den Werten

$$a = b = 10^{300}, \quad c = 10^{155}, \quad d = 10^{155} - 1,$$

die sich bei der gegebenen Präzision `stagprec = 39` alle durch Punktintervalle vom Typ `l_interval` einschließen lassen, liefert das nachfolgende C-XSC Programm

```
// Programm  lx_test4.cpp,  Komplexe Division:
// Berechnung des Imaginärteils.

#include <iostream>       // Wegen cout
#include "lx_interval.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;  // Präzision: ca. 16*39 = 624 Dezimalziffern
    l_interval a,c,y;
    lx_interval Im,A,B,C,D;

    string("[1e300,1e300]") >> a; // a: Punkt-Intervall
    A = lx_interval(a);   B = A;
    string("[1e155,1e155]") >> c; // c: Punkt-Intervall
    C = lx_interval(c);
    D = C - 1;                    // D: Punkt-Intervall

    Im = (B*C - A*D) / (C*C + D*D);
    y = Im;
    cout << SetDotPrecision(16*stagprec,16*stagprec)
        << Scientific;
    cout << "y = " << y << endl;
}
```

mit: $\texttt{y} = \underbrace{5.000\ldots 5000\ldots 2}_{311\text{ Dez.-Ziffern}}{}^{\,545\ldots}_{\,446\ldots} \cdot 10^{-11}$ eine optimale Einschließung von $\Re(z/w)$ mit 311 korrekten Dezimalstellen.

Hinweise:

1. Nur mit der `l_interval`-Arithmetik erzeugen rechts in (14) schon alle auftretenden Produkte einen Überlauf, der bei Anwendung des Divisionsoperators in der Klasse `l_cinterval` durch geeignete Skalierungen vermieden wird. Benutzt man im obigen Beispiel jedoch die Operanden `A,B,C,D` für die `lx_interval`-Arithmetik, so braucht man sich um diese Skalierungen nicht mehr zu kümmern, da diese notwendigen Skalierungen bereits in den entsprechenden Operatoren der Klasse `lx_interval` implementiert sind.

2. Ersetzt man die Programmzeile `D = C - 1;` durch `D = C - 1e-300;` und gibt statt `y` die `lx_interval`-Variable `Im` aus, so wird $\Im(z/w)$ immer noch mit 176 korrekten Dezimalziffern eingeschlossen. Wird 10^{-300} jedoch durch ein echtes Intervall eingeschlossen, so reduziert sich die genannte Genauigkeit auf nur noch 15 korrekte Dezimalstellen.

5.2.5 Ein- und Ausgabe

Ein Objekt a der Klasse `lx_interval` wird symbolisiert durch

$$a = (a.ex, a.li) = 2^{a.ex} \cdot a.li,$$

wobei die Datenelemente $a.ex$ und $a.li$ jeweils vom Typ `int` bzw. `l_interval` sind. Mit dem Datentyp `lx_interval` steht daher dem Anwender wegen

$$-2147483648 \leq a.ex \leq +2147483647$$

im Vergleich zum Typ `l_interval` ein stark erweiterter Zahlenbereich zur Verfügung.

Bei den bisherigen Beispielen, vgl. z.B. (11) auf Seite 27, waren bei den vier Grundoperationen die Operanden und die Ergebnisse stets von der Form

$$a = 2^{a.ex} \cdot a.li, \quad \text{bzw.} \quad a = 2^0 \cdot a.li = a.li,$$

d.h. für den ausschließlich mit dem Dezimalsystem vertrauten Anwender sind Objekte vom Typ `lx_interval` nur lesbar, wenn $a.ex = 0$ gilt. Damit wäre dann aber der für uns lesbare Zahlenbereich wieder sehr stark reduziert. Wer überschaut z.B. spontan, dass die Zahl 12 durch das folgende Produkt

$$12 \approx 2^{-37577} \cdot 7.64413435082585335863175465285689440 17 \ldots \cdot 10^{+306}$$

mit 38 korrekten Dezimalziffern approximiert wird? Die erfolgreiche Anwendung der `lx_interval`-Arithmetik erfordert daher die Umwandlung eines Produkts $2^{a.ex} \cdot a.li$ in einen direkt lesbaren dezimalen Ausdruck und umgekehrt.

Für ein sinnvolles Arbeiten mit `lx_interval`-Objekten ist es daher erforderlich, für die folgenden Aufgabenstellungen

1. Optimale Einschließung von Dezimalzahlen, wie z.B. $x = 1.2345678 \ldots \cdot 10^{-30111}$ oder entsprechenden Intervallen in dezimaler Darstellung durch Intervalle vom Typ `lx_interval`.

2. Optimale Einschließung eines Ergebnisses vom Typ `lx_interval` durch ein Intervall z in dezimaler Darstellung, wobei z als Variable vom Typ `string` zur Verfügung gestellt werden soll.

3. Um die Genauigkeit einer Rechnung direkt kontrollieren zu können, soll ein Ergebnis a vom Typ `lx_interval` mit seinem Zweier-Exponenten $a.ex$ und seinem Intervall $a.li$ vom Typ `l_interval` in eine Zeichenkette geschrieben werden.

4. Falls möglich, ist eine Variable a vom Typ `lx_interval` optimal in eine Variable b vom Typ `l_interval` zu transformieren, wobei $a \subseteq b$ erfüllt sein muss. Die letzte Bedingung lässt sich nur dann nicht realisieren, wenn der Wert von a betragsmäßig größer ist als `MaxReal` $= 1.797 \ldots \cdot 10^{+308}$.

entsprechende Werkzeuge zur Verfügung zu stellen.

Die **1. Aufgabe** wird gelöst mit dem Konstruktor

```
lx_interval(int n, const string& s);
```

wobei `n` der Zehnerexponent und `s` eine Zeichenkette ist, die ein Intervall m in dezimaler Form repräsentiert. Ist a das vom Konstruktor erzeugte Objekt vom Typ `lx_interval`, so gilt

$$10^n \cdot m \subseteq a.$$

Zur besseren Lesbarkeit sollte $|m| \sim 1$ gelten, was jedoch nicht zwingend notwendig ist. `s` muss die eckigen Klammern enthalten, und Infimum und Supremum sind durch ein Komma zu trennen. Erlaubte Zeichenketten sind damit:

```
s = "[0,0]"; s = "[1.2,1.3]"; s = "[-1.51e-13,-1.51e-13]";
       s = "[0,0.12345]"; s = "[-1e-200,+1e+200]";
```

Nicht erlaubte Zeichenketten sind:

```
s = "1.234"; s = "[-1.51e-13,-1.51e-13";
```

`s = "[0,0.12345]"` ist zwar erlaubt, das Intervall besitzt jedoch für eine sinnvolle Staggered Intervallarithmetik einen viel zu großen relativen Durchmesser, vgl. dazu die Bedingung (4) auf Seite 7.

Für die Fälle $n = 0$ und $n \neq 0$ noch die folgenden Hinweise:

1. Im Fall $n = 0$ und z.B. `s = "[0.5,0.5]"` liefert der obige Konstruktor ein Punktintervall, da 0.5 in $S(2, 53)$ exakt darstellbar ist.

2. Im Fall $n = 0$ und `s = "[0.51,0.51]"` liefert der obige Konstruktor kein Punktintervall, da 0.51 in $S(2, 53)$ nicht exakt darstellbar ist.

3. Im Fall $n \neq 0$ generiert der Konstruktor für `s` $\neq 0$ kein Punktintervall a, da die Bedingung $10^n \cdot m \subseteq 2^{a.ex} \cdot a.li$ durch logarithmische Intervallrechnung realisiert wird, bei der minimale Überschätzungen unvermeidlich sind.

Im nachfolgenden Programm findet man entsprechende Anwendungen zur 1. Aufgabenstellung.

Die **2. Aufgabe** wird gelöst mit dem Operator

```
ostream & operator « (ostream& s,const lx_interval& a);
```

wobei das Objekt a in dezimaler Darstellung als da in den Ausgabekanal geschrieben wird. Auch hierbei gilt: $a \subseteq da$. Der in den Ausgabekanal geschriebene Wert da hat die folgende Struktur:

(15) `{-13, [2.1756...e+1,2.1756...e+1]}`

und beschreibt dabei das Intervall $10^{-13} \cdot [2.1756... \cdot 10^{+1}, 2.1756... \cdot 10^{+1}]$.

Die **3. Aufgabe** wird gelöst mit dem Operator

```
string & operator « (string &s, const lx_interval& a);
```

der ein Objekt a vom Typ `lx_intervall` mit seinem Zweier-Exponenten $a.ex$ und dem Intervall $a.li$ vom Typ `l_interval`in eine Zeichenkette schreibt. Da hierbei logarithmischen Rechnungen nicht nötig sind, kann jetzt die interne Genauigkeit von a direkt, d.h. ohne zusätzliche Überschätzungen, abgelesen werden, wenn die Zeichenkette `s` auf dem Bildschirm ausgegeben wird. `s` hat dabei die Struktur:

```
{(a.ex),[Inf(a.li), Sup(a.li)]} .
```

Zur notwendigen Unterscheidung von der dezimalen Darstellung in (15) wird also jetzt der Zweier-Exponent $a.ex$ in runde Klammern gesetzt!

Die **4. Aufgabe** wird gelöst mit dem Zuweisungsoperator

```
l_interval & operator = (const lx_interval & a);
```

der in `l_interval.hpp` deklariert und in `lx_interval.cpp` implementiert ist. Der Zuweisungsoperator `=` erzeugt eine Fehlermeldung, wenn wegen eines drohenden Überlaufs ein Objekt a nicht in `l_interval` darstellbar ist. Bei einem drohenden Unterlauf kann a natürlich nur mit großen Überschätzungen eingeschlossen werden.

Das folgende Programm `lx_In-Out_1.cpp` demonstriert die dezimale Ein- und Ausgabe entsprechend der Aufgabenstellungen 1. und 2.

```
// Programm  lx_In-Out_1.cpp;
// Test der dezimalen Ein- und Ausgabe:

#include <iostream>
#include "lx_interval.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 20;
    lx_interval Y;

    Y = lx_interval(646456684,"[1,1]") /
                              lx_interval(6684,"[3,3]");
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "Y = " << Y << endl;
}
```

Das Programm liefert auf dem Bildschirm für den Quotienten $10^{646450000}/3$ die folgende Einschließung mit 299 korrekten Dezimalziffern:

$$\frac{10^{646450000}}{3} \in \mathtt{Y} = 10^{646449998} \cdot \left(\underbrace{3.333333 \ldots 3.333333}_{299 \text{ korrekte Dez.-Ziffern}} {}^{435\ldots}_{235\ldots} \cdot 10^{1} \right).$$

Das nächste C-XSC Programm zeigt, wie Objekte `X1,X2` vom Typ `lx_interval` in der Form `(X1.ex,X1.li)` zu realisieren sind und wie ein Ergebnis `Y` vom gleichen Typ zur Kontrolle seiner Genauigkeit in der Form `(Y.ex,Y.li)` und zusätzlich zur besseren Lesbarkeit in dezimaler Form ausgegeben werden kann.

```
// Programm  lx_In-Out_2.cpp;
// Test der Ein- und Ausgabe:

#include <iostream>       // Wegen cout
#include "lx_interval.hpp"
using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 20;
    lx_interval X1,X2,Y;
    l_interval y;
    string s;

    X1 = lx_interval(12345,l_interval(10.5));
    X2 = lx_interval(12345,l_interval(10.5));
    Y = X1 / X2;
    if (point_intv(Y))
       cout << "Y ist ein Punktintervall" << endl;
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    s << Y;
    cout << "(2^Y.ex * Y.li) = " << s << endl;
    cout << "Y = " << Y << endl;
    y = Y;
    cout << "y = " << y << endl;
}
```

Das Programm liefert folgende Ergebnisse:

```
Y ist ein Punktintervall.

(2^Y.ex*Y.li) = {(-1022),[4.49...40...E+0307,4.49...04...E+0307]

Y = {-1, [9.99999...9999873...E+0000,1.00000...0000115...E+0001]

y = [1.000000000000...00000E+0000, 1.000000000000...00000E+0000]
```

Hinweise:

1. `X1, X2` sind Punktintervalle mit gleichen Werten:

   $$\texttt{X1} = \texttt{X2} = 2^{12345} \cdot [10.5, 10.5],$$

   so dass für den exakten Quotienten gilt: `X1/X2` $= 1$.
   Beachten Sie dabei bitte, dass `X1, X2` vom Typ `lx_interval` wegen ihrer großen Zweier-Exponenten in `l_interval` nicht darstellbar sind.

2. Wegen der Ausgabe `Y ist ein Punktintervall` wird `Y` intern auch als Punktintervall berechnet. Um `Y` in der Form `(Y.ex,Y.li)` ohne zusätzliche Überschätzungen auszugeben, wird `Y` zunächst durch `s « Y` in eine Zeichenkette `s` geschrieben. Dabei muss vorher mit `SetDotPrecision(...)` die Breite des Ausgabeformats für das Intervall `Y.li` vom Typ `l_interval` festgelegt werden. Zur Kontrolle der Genauigkeit erfolgt die Ausgabe der Zeichenkette `s` durch:

   ```
   cout << "(2^Y.ex * Y.li) = " << s << endl;
   ```

   wobei auf dem Bildschirm direkt abzulesen ist, dass `Y` ein Punktintervall ist.

3. Die Ausgabe des Punktintervalls `Y` in dezimaler Form erfolgt durch

   ```
   cout << "Y = " << Y << endl;
   ```

   Wegen der notwendigen internen Umrechnung auf die neue Basis 10 kann man wegen der dabei auftretenden unvermeidbaren aber nur minimalen Überschätzungen die Ausgabe des Punktintervals $[1, 1]$ nicht erwarten. Vielmehr erhält man für das exakte Intervall $[1, 1]$ eine nur leicht überschätzte Einschließung mit 301 korrekten Dezimalstellen.

4. Mit der Anweisung `y = Y;` wird das Punktintervall `Y` rundungsfehlerfrei in den Typ `l_interval` transformiert.

5. Wählt man im Programm den Konstruktor-Aufruf `l_interval(10.1)` anstelle von `l_interval(10.5)`, so erhält man für `Y` ebenfalls das Punktintervall $[1, 1]$. Jedoch gilt jetzt nicht `X1` $= 2^{12345} \cdot [10.1, 10.1]$, da im Gegensatz zu 10.5 die Dezimalzahl 10.1 nicht in $S(2, 53)$ darstellbar ist. Bedeutet α die zu 10.1 nächstgelegene Rasterzahl, so gilt vielmehr: `X1` $= 2^{12345} \cdot [\alpha, \alpha]$. Soll jedoch `X1` das Intervall $2^{12345} \cdot [10.1, 10.1]$ wirklich einschließen, so gelingt dies mit den Anweisungen:

   ```
   l_interval x;  "[10.1,10.1]" >> x;
       X1 = lx_interval(12345,x);
   ```

   Allerdings sind dann `X1` und somit auch `Y` und `y` keine Punktintervalle mehr, wovon sich der Leser mit einem entsprechenden Programm leicht überzeugen kann.

Unter Punkt 5. der vorhergehenden Seite 36 wurde gezeigt, wie man z.B. das Intervall

$$2^{12345} \cdot [10.1, 10.1]$$

durch ein Intervall `X1` vom Typ `lx_interval` bez. der vorgegebenen Präzision optimal, d.h. ohne wesentliche Überschätzungen, einschließen kann. Da der obige Intervallwert wegen der Basis 2 nicht unmittelbar abgelesen werden kann, wird jetzt noch gezeigt, wie ein in dezimaler Form gegebenes Intervall

$$\mathtt{u} := 10^{12345} \cdot [10.1 \cdot 10^{-1}, 10.1 \cdot 10^{-1}]$$

durch ein Intervall vom Typ `lx_interval` in ausreichender Genauigkeit eingeschlossen werden kann. Eine erste Lösung haben wir bereits auf Seite 33 mit Hilfe des Konstruktors

```
lx_interval(int n, const string& s);
```

kennengelernt. Im nachfolgenden Programm `lx_In-Out_3.cpp` wird das Intervall `u` durch `string("{(12345),[10.1e-1,10.1e-1]}") >> U;` in das Objekt `U` kopiert, wobei $\mathtt{u} \subseteq \mathtt{U}$ garantiert ist. Im zweiten Teil des Programms wird die Tastatureingabe mittels `cin` ebenfalls in die Variable `U` kopiert und anschließend auf dem Bildschirm ausgegeben.

```
// Programm  lx_In-Out_3.cpp,

#include <iostream>       // Wegen cout
#include "lx_interval.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 3;
    lx_interval U;
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    string("{12345,[10.1e-1,10.1e-1]}") >> U;
    cout << "U = " << U << endl;

    cout << "lx_interval u = ?" << endl;
    cin >> U;
    cout << "U = " << U << endl;
}
```

Das obige Programm liefert die erste Ausgabe

```
U = {12344,[1.00999...999159...E+0001,1.0100...000816...E+0001}}
```

mit 44 korrekten Dezimalziffern. Beachten Sie, dass bei der Ein- und Ausgabe interne Umrechnungen notwendig sind, wodurch nur 44 der möglichen $3 \cdot 16 = 48$ korrekten Dezimalziffern ausgegeben werden.

Hinweise:

1. In der obigen string-Anweisung ist `12345` der Exponent zur Basis 10;

2. Die Anweisung `string("{12345,[10.1e-1,10.1e-1]}") >> U;` wird realisiert durch den Operator

   ```
   void operator >> (const string &, lx_interval &) throw();
   ```

   vgl. dazu den entsprechenden Quelltext in `lx_interval.cpp`.

3. Wählt man die Präzisionsvariable `stagprec` ≥ 20, so erhält man bei der obigen Ein- und Ausgabe wegen der notwendigen interne Umrechnungen nicht mehr als etwa 300 korrekte Dezimalziffern, was in der numerischen Praxis meist völlig ausreichend ist. Wählt man bei der Eingabe jedoch Punktintervalle, die in $S(2, 53)$ exakt darstellbar sind, so liefert z.B. die Anweisung

   ```
   string("{0,[10.1e0001,10.1e0001]}") >> U;
   ```

   ein Punktintervall `U`, das $[101, 101]$ in der jeweils vorgegebenen Präzision optimal einschließt. Diese optimale Einschließung kann kontrolliert werden, wenn man das Intervall `U` über einen string `s` auf dem Bildschirm ausgibt, vergleiche dazu z.B. das Programm `lx_In-Out_2.cpp` auf Seite 35. Wird dieses Punktintervall `U` durch

   ```
   cout << "U = " << U << endl;
   ```

   wieder in dezimaler Form auf dem Bildschirm ausgegeben, so erhält man dabei wegen der schon genannten internen Umrechnungen leichte Überschätzungen und im Fall `stagprec` ≥ 20 maximal nur noch etwa 300 korrekte Dezimalziffern. Es ist also zu beachten, dass durch eine Überschätzung bei der Ergebnisausgabe in dezimaler Form die tatsächliche interne Genauigkeit einer Rechnung, insbesondere für `stagprec` ≥ 20, nicht mehr am Bildschirm direkt abgelesen werden kann!

4. Die Übergabe der Tastatureingabe an das Intervall `U` durch

   ```
   cout << "lx_interval u = ?" << endl;  cin >> U;
   ```

   wird realisiert durch den Operator

   ```
   std::istream & operator >> (std::istream &, lx_interval &);
   ```

   und garantiert `u` $\subseteq$ `U`. Wie bei der string-Eingabe unter Punkt 2. muss die Tastatureingabe das folgende Format besitzen:

   ```
   {12345,[10.1e-1,10.1e-1]}
   ```

   wobei z.B. vor und nach einem Komma oder einer Klammer beliebige Leerzeichen erlaubt sind.

5.2.6 Zu breite Eingangsintervalle

Auf Seite 7 hatten wir schon die Bedingung

(16) $$\texttt{RelDiam}(\boldsymbol{x}) < 10^{-16*(p-1)}$$

für den relativen Durchmesser eines Intervalls $\boldsymbol{x}$ angegeben, die erfüllt sein sollte, um mit einer vorgegebenen Präzision $\texttt{stagprec} = p$, d.h. also mit etwa $16 \cdot p$ Dezimalstellen, eine sinnvolle staggered Intervallrechnung realisieren zu können. Wir zeigen jetzt anhand der Exponentialfunktion, dass bei den Standardfunktionen die Algorithmen unter der Prämisse schmaler Eingangsintervalle entwickelt wurden. Es muss also damit gerechnet werden, dass bei zu breiten Eingangsintervallen $\boldsymbol{x}$ das einzuschließende Intervall der Funktionswerte $e^{\boldsymbol{x}} := \{y \mid y = e^t \wedge t \in \boldsymbol{x}\}$ deutlich überschätzt werden kann. Um dies zu zeigen, wählen wir $\boldsymbol{x} = [-600, +600]$ und erhalten mit den Anweisungen

```
stagprec = 3;
lx_interval X,Y;

cout << SetDotPrecision(16*stagprec,16*stagprec+3)
     << Scientific;
X = lx_interval(0,l_interval(-600,600));
Y = exp(X);
cout << "exp(X) = " << Y << endl;
```

die Einschließung

(17) $$e^{\boldsymbol{x}} \subset \texttt{Y} = 10^{+259} \cdot [0.000\ldots000 \cdot 10^0, 3.774367778\ldots \cdot 10^{+1}].$$

Mit `X = lx_interval(0,l_interval(-600,-600));` erhält man

$$e^{\boldsymbol{x}} \subset \texttt{Y} = 10^{-261} \cdot [2.65039\ldots270147\ldots \cdot 10^0, 2.65039\ldots270160\ldots \cdot 10^0],$$

und mit `X = lx_interval(0,l_interval(600,600));` erhält man

$$e^{\boldsymbol{x}} \subset \texttt{Y} = 10^{+259} \cdot [3.773020\ldots31350969\ldots \cdot 10^1, 3.773020\ldots31350974\ldots \cdot 10^1].$$

Aus diesen beiden letzten Einschließungen kann jetzt die deutliche Überschätzung in (17) direkt abgelesen werden.

Ganz analog kann man z.B. für das Eingangslintervall $\boldsymbol{x} = [1, 6]$ zeigen, dass auch das Intervall $\ln(\boldsymbol{x})$ deutlich überschätzt wird.

> Wenn der relative Durchmesser von $\boldsymbol{x}$ die Bedingung (16) annähernd erfüllt, so liefert die erweiterte staggered Intervall-Arithmetik über einen sehr großen Zahlenbereich Einschließungen in hoher Genauigkeit.

5.2.7 Die Funktionen x^2 und $\sqrt{x}$

Die Quadrat-Funktion `lx_interval sqr(const lx_interval &a);` wird mit Hilfe der Beziehung

$$\texttt{sqr}(2^{a.ex} \cdot a.li) = 2^{2 \cdot a.ex} \cdot \texttt{sqr}(a.li)$$

implementiert, wobei vorher $a.li$ für eine optimale Genauigkeit so zu skalieren ist, dass danach gilt: $exa = 511$. $exa =$ `expo_gr(a.li);`

Die Quadratwurzel `lx_interval sqrt(const lx_interval &a);` wird mit
Hilfe der Beziehung

$$\sqrt{(2^{a.ex} \cdot a.li)} = 2^{a.ex/2} \cdot \sqrt{a.li}$$

implementiert, wobei vorher $a.li$ für eine optimale Genauigkeit so zu skalieren ist, dass danach gilt: $exa = 1024$. Bei dieser Skalierung ist zusätzlich darauf zu achten, dass $a.ex$ durch 2 teilbar ist. Bei einer maximalen Präzision `stagprec` $= 30$ kann $\sqrt{a.li}$ mit höchstens $30 \cdot 16 = 480$ korrekten Dezimalstellen berechnet werden, so dass die Quadratwurzel auch in der Klasse `lx_interval` nicht genauer eingeschlossen werden kann. Ein Test beider Funktionen erfolgt mit Hilfe der bekannten Beziehung

$$\sqrt{x^2} = |x|, \quad \text{mit} \quad x = -4 \cdot 10^{-40000}.$$

Das folgende C-XSC Programm

```
// Programm  lx_test5.cpp,  Test der Funktionen:
// sqr(...) und sqrt(...).

#include <iostream>       // Wegen cout
#include "lx_interval.hpp"
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 19;
    lx_interval X,Y;

    X = lx_interval(-40000,"[-3.1,-3.1]");
    Y = sqrt( sqr(X) );

    cout << SetDotPrecision(16*stagprec,16*stagprec)
        << Scientific;
    cout << "Y = " << Y << endl;
}
```

liefert mit Y eine garantierte Einschließung von $|x| = 4 \cdot 10^{-40000}$ mit 301 korrekten Dezimalziffern, d.h. eine nahezu optimale Einschließung.

Die Quadratwurzelfunktion kann mit der maximalen Präzision `stagprec` $= 30$ angewandt werden. Dazu darf man jedoch nicht die dezimale Eingabe mit dem Konstruktor

```
lx_interval(int p, const string& s);
```

wählen, wobei `p` der Zehner-Exponent ist, da dann wegen der internen logarithmischen Umrechnung auf die Basis 2 Überschätzungen schon nach etwa 300 Dezimalstellen auftreten. Im nachfolgenden Programm muss man vielmehr den Konstruktor benutzen, bei dem der Zweier-Exponenten -7654321 einzugeben ist. Das Programm `lx_test5a.cpp` berechnet für $\boldsymbol{x} = 2^{-7654321} \cdot [4.1 \cdot 10^{+300}, 4.1 \cdot 10^{+300}]$ eine garantierte Einschließung von $\sqrt{\boldsymbol{x}^2}$ mit maximal 510 korrekten Dezimalstellen.

```
// Programm  lx_test5a.cpp,  Test der Funktionen:
// sqr(...) und sqrt(...).

#include <iostream>
#include "lx_interval.hpp"
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 30;
    l_interval x;
    string s;
    lx_interval X,Y;

    string("[4.1e300,4.1e300]") >> x;
    X = lx_interval(-7654321,x);
    Y = sqrt( sqr(X) );

    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "Y = " << Y << endl; // Dezimale Ausgabe
    s << Y;  // Zur Ausgabe in maximaler Genauigkeit
    cout << "string s = " << s << endl;
}
```

Die Ausgabe der Einschließung erfolgt in maximaler Genauigkeit über den String `s` und in wirklich lesbarer dezimaler Form in geringerer Genauigkeit mit dem Intervall `Y`:

$$\sqrt{\boldsymbol{x}^2} \in 2^{-7653834} \cdot [\underbrace{1.026067\ldots2499\ldots999}_{510 \text{ korrekte Dez.-Ziffern}}505, 1.026067\ldots2500\ldots000495] \cdot 10^{154},$$

$$\sqrt{\boldsymbol{x}^2} \in \mathtt{Y} = 10^{-2303880} \cdot [\underbrace{2.4850958065\ldots5059670}_{299 \text{ korrekte Dez.-Ziffern}}098, 2.4850958065\ldots5059670206].$$

Nur wenn der Eingabewert $\boldsymbol{x} = [4.1 \cdot 10^{+300}, 4.1 \cdot 10^{+300}]$ mit dem Zweier-Exponenten 0 vorliegt, liefert auch der string `s` eine lesbare Einschließung in maximaler Genauigkeit!

5.2.8 Die Funktion x^n

Die Potenzfunktion `lx_interval power(const lx_interval &a, int n);`
ist für $n \in \mathbb{Z}$ mit Hilfe des nahezu optimalen Algorithmus von A.M. Legendre (1798) implementiert. Da neben integer-Operationen nur Multiplikationen und Divisionen mit `lx_interval`-Operanden auftreten, kann mit der maximalen Präzision `stagprec` $= 39$ gerechnet werden. Als Anwendung ist der Ausdruck

$$y = \frac{(10^{300}+1)^{2000001}}{(10^{300}+1)^{2000000}} = 10^{300}+1 \tag{18}$$

rechts in (18) mit Hilfe der Potenz-Funktion einzuschließen. Das nachfolgende Programm

```
// Programm  lx_test6.cpp,  Test der Funktion:
// lx_interval power(const lx_interval &, int);
#include <iostream>       // Wegen cout
#include "lx_interval.hpp"
#include "lx_imath.hpp"
using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;  // Maximale Präzision
    l_interval x,y;
    lx_interval X;

    string("[1e300,1e300]") >> x; // x: Punkt-Intervall
    X = lx_interval(x) + 1;          // X: Punkt-Intervall

    y = power(X,2000001) / power(X,2000000);

    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "y = " << y << endl;
}
```

liefert mit `y` eine Einschließung von $10^{+300}+1$ mit 471 korrekten Dezimalstellen.

Hinweise:

1. Während in der Klasse `l_interval` bereits die Berechnung von $(10^{+300}+1)^2$ zum Überlauf führt, kann der Bruch in (18) mit der `lx_interval` power-Funktion in hoher Genauigkeit eingeschlossen werden.

2. Schreibt man das Programm `lx_test6.cpp` so um, dass y definiert wird durch

$$y = \frac{(10^{300}+1)^{2000001}}{(10^{300}+2)^{2000000}},$$

so liefert y eine Einschließung von y mit 470 korrekten Dezimalstellen.

5.2.9 Die Funktion e^x

Ein Nachteil der Exponentialfunktion für die Typen `interval` und `l_interval` ist der recht kleine Definitionsbereich $|x| < 709,78\ldots$, da sonst die Funktionswerte in den Überlaufbereich fallen. In der Klasse `lx_interval` kann dieser Definitionsbereich auf $|x| < 1488521882$ ganz erheblich erweitert werden, wobei auch die Genauigkeit der Funktionswerteinschließungen deutlich verbessert wurde.

Algorithmus

Der Algorithmus basiert auf der Identität

$$e^x = \left(e^{x\cdot 2^{-n}}\right)^{2^n}, \quad n = 0, 1, 2, 3, \ldots$$

wobei wie folgt vorgegangen wird:

1. Zunächst wird n so berechnet, dass gilt: $|x| \cdot 2^{-n} \sim \alpha := 10^{-9}$. Im Fall $n < 0$ wird $n = 0$ gesetzt. Mit dieser Argumentreduktion kann $e^{x\cdot 2^{-n}}$ auch bei einer vorgegebenen Präzision `stagprec` $= 40$ noch sehr effektiv mit einem abgebrochenen Taylorpolynom approximiert werden. Es ist zu beachten, dass die Multiplikation des Argumentintervalls $\boldsymbol{x}$ mit 2^{-n} in der Klasse `lx_interval` mit der Funktion `times2pown(x,-n)` rundungsfehlerfrei und sehr schnell erfolgt.

2. Zur Ergebnisanpassung ist dann noch $u := e^{x\cdot 2^{-n}}$ mit 2^n zu potenzieren. Mit der folgenden Schleife

```
for (int k=1; k<=n; k++)
    u = sqr(u);
```

   kann dies im Gegensatz zur `power`$(u, 2^n)$-Funktion sehr effektiv realisiert werden.

Der absolute Approximationsfehler δ wird mit Hilfe der geometrischen Reihe wie folgt abgeschätzt:

$$e^x = 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \ldots = P_\infty(x), \quad x \in \mathbb{R},$$
$$e^x \approx 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \ldots + \frac{x^N}{N!} = P_N(x).$$

$$\begin{aligned} |\delta| &= |P_\infty(x) - P_N(x)| = \left|\sum_{k=N+1}^{\infty} \frac{x^k}{k!}\right| = \frac{|x|^{N+1}}{(N+1)!} \cdot \left|\sum_{k=N+1}^{\infty} \frac{x^{k-N-1}\cdot(N+1)!}{k!}\right| \\ &= \frac{|x|^{N+1}}{(N+1)!} \cdot \left|\sum_{k=0}^{\infty} \frac{x^n\cdot(N+1)!}{(n+N+1)!}\right| \le \frac{|x|^{N+1}}{(N+1)!} \cdot \frac{1}{1-|x|}, \quad |x| < 1. \end{aligned}$$

Bedeutet jetzt $\boldsymbol{T} = \boldsymbol{x} \cdot 2^n$ das reduzierte Argumentintervall, mit $t \in \boldsymbol{T}$ und $|\boldsymbol{T}| = R < 1$, so gilt für den absoluten Approximationsfehler

$$|P_\infty(t) - P_N(t)| \le \frac{R^{N+1}}{(N+1)!} \cdot \frac{1}{1-R} \le S, \quad \forall t \in \boldsymbol{T}.$$

Im nächsten Schritt wird der Polynomgrad N bestimmt. Wegen $P_N(t) \approx 1$ sollte bei einer gewählten Präzision $p =$ `stagprec` der absolute Fehler betragsmäßig nicht größer sein als $1 \cdot 2^{-53 \cdot p}$. Wir verlangen daher:

$$\frac{R^{N+1}}{(N+1)!} \cdot \frac{1}{1-R} \leq 2^{-53p}.$$

Wegen $|\boldsymbol{T}| = R \ll 1$ kann dies noch vereinfacht werden zu

$$\frac{R^{N+1}}{(N+1)!} \leq 2^{-53p}, \quad \text{bzw.}$$

$$(N+1) \cdot \ln(R) - \ln(N+1)! \leq -53p \cdot \ln(2). \tag{19}$$

Um die Berechnung von $\ln(R)$ zu vermeiden, schreibt man $R = M \cdot 2^m,\ m \in \mathbb{Z}$, wobei für die Mantisse M gilt $0.5 \leq M < 1$. Die Ungleichung (19) kann damit weiter vereinfacht werden zu

$$(N+1) \cdot m \cdot \ln(2) - \ln(N+1)! \leq -53p \cdot \ln(2).$$

Mit $D(N) := (N+1) \cdot m \cdot \ln(2) - \ln(N+1)!$ und mit

$$\ln(N+2)! = \ln(N+1)! + \ln(N+2)$$

folgt die Rekursionsformel

$$D(N+1) = D(N) + m \cdot \ln(2) - \ln(N+2), \quad N = 0, 1, 2, \ldots$$

und (19) kann damit wie folgt geschrieben werden

$$D(N) \leq -53p \cdot \ln(2). \tag{20}$$

Man beginnt mit $N = 0$, vergrößert N schrittweise um 1 und berechnet $D(N+1)$ mit Hilfe der Rekursionsformel, bis die Ungleichung (20) erfüllt ist. Dadurch wird N ohne wesentliche Überschätzungen berechnet. Speichert man die `real`-Werte $\ln(N+2)$ für $N = 0, 1, 2, \ldots$ als Konstanten, so kann die Laufzeit noch wesentlich reduziert werden.

Die Auswertung des Polynoms $P_N(\boldsymbol{T})$ erfolgt in bekannter Weise nach dem Horner-Schema. Bezeichnet man das Maschinenergebnis mit $\mathsf{P}(\boldsymbol{T})$, so gilt $P_N(\boldsymbol{T}) \subseteq \mathsf{P}(\boldsymbol{T})$. Mit der Oberschranke S wird der absolute Approximationsfehler $P_\infty(t - P_N(t)$ eingeschlossen durch $P_\infty(t - P_N(t) \in [-S, +S]$ und es gilt:

$$e^t \in \boldsymbol{u} := \mathsf{P}_N(\boldsymbol{T}) \oplus [-S, +S], \quad \forall t \in \boldsymbol{T}.$$

Die garantierte Einschließung von $\boldsymbol{u}^{2^n} \subseteq \mathsf{u}$ und damit von $e^x \in \mathsf{u}\ \forall x \in \boldsymbol{x}$ erfolgt dann durch intervallmäßige Auswertung der `for`-Schleife von Seite 44, wobei u das Maschinenergebnis dieser Schleife ist.

Abschließend noch Anmerkungen zum Laufzeitverhalten. Mit wachsender Präzision wächst die Laufzeit etwa quadratisch. Verkleinert man $\alpha = 10^{-9}$ z.B. auf 10^{-11}, so wird die Laufzeit zwar etwas geringer, aber wegen der notwendigen Potenzierungen mit 2^n wird die Genauigkeit mit wachsendem n dabei etwas reduziert. Testrechnungen haben ergeben, dass $\alpha = 10^{-9}$ ein guter Kompromiss ist.

Als erste Anwendung wird der sehr große Funktionswert $y = e^{1488521882}$ mit folgendem Programm in hoher Genauigkeit eingeschlossen.

```
// Programm  lx_test7.cpp,  Test der Funktion:
// lx_interval exp(const lx_interval &x);

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 40;
    l_interval x,y;
    lx_interval X,Y;
    string s;

    x = l_interval(1488521882.0);
    X = lx_interval(0,x);
    X = adjust(X);

    Y = exp(X);

    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "exp(X) = " << Y << endl;
    s << Y;
    cout << "exp(X) = " << s << endl;
    // y = Y;
}
```

Y vom Typ `lx_interval` liefert die Einschließung in dezimaler Form

$$y = e^{1488521882} \in \mathsf{Y} = 10^{646456838} \cdot \underbrace{3.506253825318\ldots480773513492027 75}_{299 \text{ korrekte Dez.-Ziffern}} {}^{690\ldots}_{425\ldots} \cdot 10^{1}.$$

Natürlich kann dieses große Intervall nicht durch ein Intervall vom Typ `l_interval` eingeschlossen werden. Mit der obigen Anweisung `y = Y` erhält man daher eine entsprechende Fehlermeldung.

Bei der zweiten Ausgabe über den String `s` wird die volle Genauigkeit geliefert, mit der intern die Einschließung berechnet wurde:

$$y = e^{1488521882} \in \mathsf{Y} = 2^{2147482116} \cdot \underbrace{2.99837667210223\ldots77446182629586}_{457 \text{ korrekte Dez.-Ziffern}} {}^{538\ldots}_{303\ldots} \cdot 10^{307}.$$

Bei der ersten Ausgabe im lesbaren Dezimalformat muss logarithmisch auf die Basis 10 umgerechnet werden, was mit einem Genauigkeitsverlust, jedoch *nur* bei der Ausgabe, verbunden ist.

Das zweite Beispiel zeigt, dass z.B. die Funktion $y = e^{-x^2}$ nicht extra implementiert werden muss, sondern mit Hilfe der Exponentialfunktion über die Anweisungen

```
lx_interval X,Y;     Y = exp( -sqr(X) );
```

sehr einfach realisiert werden kann.

```
// Programm  lx_test8.cpp, 2. Test der Funktion:
// lx_interval exp(const lx_interval &x);

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 40;
    l_interval x;
    lx_interval X,Y;
    string s;

    x = l_interval(38581);
    X = lx_interval(0,x);

    Y = exp(-sqr(X));
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "Y = " << Y << endl;
    s << Y;
    cout << "Y = " << s << endl;
}
```

Y vom Typ `lx_interval` liefert die Einschließung in dezimaler Form:

$$y = e^{-(38581^2)} \in \mathsf{Y} = 10^{-646444540} \cdot \underbrace{1.285815419687065\ldots4980749307270}_{299 \text{ korrekte Dez.-Ziffern}} \,{}^{980\ldots}_{888\ldots} \cdot 10^{0}.$$

Bei der zweiten Ausgabe über den String `s` wird die volle Genauigkeit geliefert, mit der intern die Einschließung berechnet wurde:

$$y = e^{-(38581^2)} \in \mathsf{Y} = 2^{-2147443300} \cdot \underbrace{2.49358455077811\ldots65459119714718}_{457 \text{ korrekte Dez.-Ziffern}} \,{}^{852\ldots}_{327\ldots} \cdot 10^{307}.$$

Bei der ersten Ausgabe im lesbaren Dezimalformat muss logarithmisch auf die Basis 10 umgerechnet werden, was mit einem Genauigkeitsverlust, jedoch *nur* bei der Ausgabe, verbunden ist.

5.2.10 Die Funktion $\ln(x)$

Bei der Berechnung der Logarithmusfunktion mit Ergebnissen vom Typ `l_interval` tritt u.a. das Problem auf, dass in der Nähe der Nullstelle 1 die Funktionswerte so klein werden, dass sie nur noch mit wenigen korrekten Dezimalstellen eingeschlossen werden können. Zur Lösung dieses Problems wird zunächst die Hilfsfunktion

```
lx_interval Lnp1(const lx_interval &x)
```

implementiert, die Funktionswerte $\ln(1+x)$ vom Typ `lx_interval`, mit $x \approx 0$, in ausreichender Genauigkeit berechnet. Der Algorithmus basiert auf folgender Reihenentwicklung:

$$\ln(1+x) = \zeta \cdot \sum_{k=0}^{\infty} \frac{2}{2k+1} \cdot (\zeta^2)^k, \quad \zeta := \frac{x}{2+x}, \quad x > -1, \quad |\zeta| < 1. \tag{21}$$

Mit $P(\zeta) := \sum_{k=0}^{\infty} \frac{2}{2k+1} \cdot (\zeta^2)^k$ und $P_N(\zeta) := \sum_{k=0}^{N} \frac{2}{2k+1} \cdot (\zeta^2)^k,\ N \geq 0$ wird $\ln(1+x)$ approximiert durch

$$\ln(1+x) \approx \zeta \cdot P_N(\zeta),\ N \geq 0. \tag{22}$$

Der absolute Approximationsfehler δ wird definiert durch

$$\begin{aligned} \delta &:= |P(\zeta) - P_N(\zeta)| = \sum_{k=N+1}^{\infty} \frac{2}{2k+1} \cdot (\zeta^2)^k = \sum_{n=0}^{\infty} \frac{2}{2n+2N+3} \cdot (\zeta^2)^{n+N+1} \\ &= (\zeta^2)^{N+1} \cdot \sum_{n=0}^{\infty} \frac{2}{2n+2N+3} \cdot (\zeta^2)^n \leq (\zeta^2)^{N+1} \cdot \frac{2}{2N+3} \cdot \frac{1}{1-\zeta^2} \end{aligned}$$

und die letzte Oberschranke kann weiter abgeschätzt werden durch

$$\delta := |P(\zeta) - P_N(\zeta)| \leq \frac{(\zeta^2)^{N+1}}{N+1}, \tag{23}$$

denn es gilt

$$\frac{2}{2N+3} \cdot \frac{1}{1-\zeta^2} < \frac{1}{N+1} \quad \Longleftrightarrow \quad \zeta^2 < \frac{1}{2N+3}.$$

Die letzte Ungleichung ist bei allen praktischen Anwendungen sicher erfüllt, denn für z.B. $|x| \sim 10^{-7}$ gilt auch $|\zeta| \sim 10^{-7}$ und wie wir sehen werden, ist dann der maximale Polynomgrad $N_{max} = 42$, so dass die Oberschranke $1/(2N+3)$ nicht kleiner werden kann als $1/(2N_{max}+3) = 1.14\ldots \cdot 10^{-2}$, d.h. sie ist sicher größer als $\zeta^2 \sim 10^{-14}$.

Wir betrachten jetzt ein Intervall $\boldsymbol{x}$, mit[10] $|\boldsymbol{x}| \ll 1$. Mit $R := |\boldsymbol{x}/(2+\boldsymbol{x})| = |\boldsymbol{\zeta}|$ gilt dann für den absoluten Fehler die Abschätzung

$$\delta = |P(\zeta) - P_N(\zeta)| \leq \frac{R^{2N+2}}{N+1} =: \Delta, \quad \forall \zeta \in \boldsymbol{\zeta} := \frac{\boldsymbol{x}}{2+\boldsymbol{x}}. \tag{24}$$

[10] $|\boldsymbol{x}| := \max_{r \in \boldsymbol{x}} \{|r|\}$

Als nächstes wird für die Approximation in (22) der geeignete Polynomgrad N bestimmt. Für $|\zeta| \ll 1$ gilt: $P(\zeta) \approx P_N(\zeta) \approx 2$, und bei einer vorgegebenen Präzision $p :=$ `StagPrec(x)` sollte der absolute Approximationsfehler δ die Größenordnung $2^{1-53\cdot p}$ besitzen. Nach (24) verlangen wir daher

$$\frac{R^{2N+2}}{N+1} = 2^{1-53\cdot p} \quad \Longleftrightarrow \quad R^{2N+2} = (N+1)\cdot 2^{1-53\cdot p}. \tag{25}$$

Zur Berechnung von N muss die rechte Seite in (25) noch weiter vereinfacht werden. Mit `xi = li_part(x); ex = expo_gr(xi)` gilt zunächst

$$|\boldsymbol{x}| \approx 2^{ex+\mathtt{expo}(\boldsymbol{x})} = 2^m, \quad \text{d.h.} \quad m := ex + \mathtt{expo}(\boldsymbol{x}).$$

Weiter gilt: $R \approx |\boldsymbol{x}|/2 = 2^{m-1}$ und das ergibt

$$2^{(m-1)\cdot(2N+2)} = (N+1)\cdot 2^{1-53\cdot p},$$

und wegen $(N+1) = 2^{\log_2(N+1)}$ folgt weiter

$$\begin{aligned}(m-1)\cdot(2N+2) &= \log_2(N+1) + 1 - 53\cdot p \\ 2N+2 &= \frac{53\cdot p - 1 - \log_2(N+1)}{1-m}, \quad m \neq 1.\end{aligned}$$

Die Bedingung $m \neq 1$ ist dabei wegen der Forderung $|\boldsymbol{x}| \ll 1$ sicher erfüllt. Für den Bereich $0 \leq N \leq 42$ gilt $0 \leq \log_2(N+1) < 5.43$, so dass für $p \geq 2$

$$53\cdot p - 1 - \log_2(N+1) \approx 53\cdot p - 4$$

zur Berechnung von N eine gute Näherung ist. Wir erhalten damit

$$2N+2 = \frac{53\cdot p - 4}{1-m} \quad \Longleftrightarrow \quad N = \frac{53\cdot p - 4}{2\cdot(1-m)} - 1.$$

Um $N \geq 0$ zu gewährleisten, wird N noch um 1 vergrößert, und wir erhalten damit zur Berechnung des Polynomgrades N die sehr einfache Formel:

$$N = \frac{53\cdot p - 4}{2\cdot(1-m)}, \quad m := ex + \mathtt{expo}(\boldsymbol{x}), \quad N \geq 0. \tag{26}$$

Die C-XSC Funktion $\mathtt{expo}(\boldsymbol{x})$ liefert dabei den Zweier-Exponenten $\boldsymbol{x}.ex$ des Intervalls $\boldsymbol{x}$ vom Typ `lx_interval`.

Bei der Auswertung des Nenners $2\cdot(1-m)$ in (26) kann es zum integer-Overflow kommen, wenn $|\boldsymbol{x}|$ hinreichend klein bzw. $-m$ hinreichend groß ist. Diesen Overflow kann man vermeiden, wenn im Fall $2\cdot(1-m) > 53\cdot p - 4$ der Polynomgrad $N = 0$ gesetzt wird.

Da bei der Auswertung von $2\cdot(1-m)$ der besagte Overflow auftreten kann, muss die letzte Bedingung $2\cdot(1-m) > 53\cdot p - 4$ weiter umgeformt werden. Es gilt

$$\begin{aligned}2\cdot(1-m) > 53\cdot p - 4 \quad &\Longleftrightarrow \quad 2m < 6 - 53\cdot p \quad \text{bzw.} \\ 2\cdot(ex + \mathtt{expo}(\boldsymbol{x})) < 6 - 53\cdot p \quad &\Longleftrightarrow \quad \mathtt{expo}(\boldsymbol{x}) < 3 - \frac{53\cdot p}{2} - ex.\end{aligned}$$

Die letzte Ungleichung ist nun sicher erfüllt, wenn gilt:

$$\texttt{expo}(\boldsymbol{x}) < 3 - 27 \cdot p - ex. \tag{27}$$

Ist daher (27) erfüllt, so wird $N = 0$ gesetzt, sonst wird N nach (26) berechnet, wobei dann kein integer-Overflow mehr auftreten kann.

Wir kommen jetzt zur Auswertung der Schranke Δ des absoluten Approximationsfehlers in (24), wobei $N = 0$ gesondert zu behandeln ist. In diesem Fall gilt $\Delta = R^2$, und jetzt kann bei der Auswertung von R^2 ein integer-Overflow entstehen, den es zu vermeiden gilt. Wir formulieren zunächst den folgenden Satz zu Anwendung der Funktion

```
lx_interval sqr(const lx_interval& x);
```

Sei `lx_interval x; l_interval y(li_part(x)); ex = expo_gr(y);`
dann wird unter der Voraussetzung $ex + \texttt{expo}(\boldsymbol{x}) \geq -1073741313$ `sqr(x)` ohne integer-Overflow berechnet.

Es gilt: $R \approx |\boldsymbol{x}|/2 \approx 2^{(ex-1)+\texttt{expo}(\boldsymbol{x})}$ und nach obigem Satz wird bei der Berechnung von R^2 ein integer-Overflow vermieden, wenn gilt:

$$(ex-1) + \texttt{expo}(\boldsymbol{x}) \geq -1073741313 \quad \Longleftrightarrow \quad \texttt{expo}(\boldsymbol{x}) \geq -1073741312 - ex.$$

Die Schranke Δ des absoluten Approximationsfehlers $\delta = |P(\zeta) - P_N(\zeta)|$ wird daher wie folgt definiert:

$$\Delta = \begin{cases} R^2, & \texttt{expo}(\boldsymbol{x}) \geq -1073741312 - ex,\ N = 0, \\ R, & \texttt{expo}(\boldsymbol{x}) < -1073741312 - ex,\ N = 0, \\ \dfrac{R^{2N+2}}{N+1}, & N \geq 1. \end{cases} \tag{28}$$

In (28) können jetzt die rechten Seiten ohne integer-Overflow ausgewertet werden. In der zweiten Zeile ist die gewählte Schranke $\Delta = R$ sicher größer als der eigentliche Wert $\Delta = R^2$. Die Funktion `Lnp1(...)` ist implementiert durch:

```
lx_interval Lnp1(const lx_interval &x)
                    throw(ERROR_LINTERVAlx_STD_FKT_OUT_OF_DEF)
// Calculating an inclusion of ln(1+x);
{
    lx_interval res(0),z2,zeta,Ri;
    l_interval xli;
    int m,N,ex,p,expox;

    p = StagPrec(x);     xli = li_part(x);
    ex = expo_gr(xli);
    if (ex>-100000) // x <> 0
    {
        expox = expo(x);
        if (expox<0)
```

```
            if (expox+2147482626 < -ex) // zeta calculation
                                        // with int-Overflow!
                cxscthrow(ERROR_LINTERVAlx_STD_FKT_OUT_OF_DEF(
                     "lx_interval Lnp1(const lx_interval &)"));
        if (expox < 3-27*p-ex) N = 0;
        else
        {
            m = ex + expox;  // m = ex + expo(x);
            N = (53*p-4)/(2*(1-m));
        }
        // N: Polynomial degree, 0<=N<=42;

        zeta = x / (2+x); // int-Overflow impossible!
        Ri = lx_interval( Sup(abs(zeta)) );
        if (N==0)
        {
            res = lx_interval(2.0);
            if (expox >= -1073741312-ex)
                Ri = sqr(Ri); // Without int-Overflow!
        }
        else // N >= 1:
        {
            z2 = sqr(zeta); // z2 = zeta^2
            // Evaluating the polynomial:
            res = lx_interval(2.0) / (2*N+1);  // res = a_(N)
            for (int i=N-1; i>=0; --i)
                res = res*z2 + lx_interval(2)/(2*i+1);
            // Calculating the absolute approximation error:
            Ri = sqr(Ri);  Ri = power(Ri,N+1)/(N+1);
        }
        // Implementing the approximation error:
        res += lx_interval(lx_real(0),Sup(Ri));
        res *= zeta;
    } // x <> 0;

    return res;
} // Lnp1(...)
```

Hinweise:

1. Es ist zu beachten, dass nach Definition des abs. Approximationsfehlers δ in (23) die Addition seiner Oberschranke[11] Δ *vor* der Multiplikation mit ζ zu erfolgen hat.

2. Wegen $P_N(\zeta) < P(\zeta)$ wird der absolute Approximationsfehler nicht durch `lx_interval(-Sup(Ri),+Sup(Ri))`, sondern mit einer sehr viel geringeren Überschätzung durch `lx_interval(lx_real(0),Sup(Ri))` eingeschlossen.

[11] Im Quelltext gilt: $\Delta :=$ `Sup(Ri)`.

Im **1. Beispiel** wird für $x = 2^{-1074} \in S(2,53)$ mit Hilfe der Funktion `Lnp1(...)` eine garantierte Einschließung von

$$y = \ln(1+x) \approx 2^{-1074} = 4.9406\ldots \cdot 10^{-324}$$

in hoher Genauigkeit berechnet. Die Anweisungen:

```
stagprec = 39;
l_interval x(comp(0.5,-1073));  lx_interval X(0,x),Y;
Y = Lnp1( X );
```

liefern mit Y die gewünschte Einschließung

$$\ln(1+x) \in \mathtt{Y} = 2^{-2095} \cdot \underbrace{2.2471164185778\ldots 58984375000\ldots 000}_{477\text{ Dez.-Ziffern}}{}^{496\ldots}_{000\ldots} \cdot 10^{+307}.$$

mit 477 korrekten Dezimalziffern. Schließt man Y jedoch durch ein Intervall y vom Typ `l_interval` ein, so erhält man mit $\mathtt{y} = [0, 9.881312\ldots \cdot 10^{-324}]$ eine nur sehr grobe und damit praktisch unbrauchbare Einschließung des Funktionswertes $y = \ln(1+x)$.

Im **2. Beispiel** wird für $x = 10^{-623456789}/3$ mit der Funktion `Lnp1(...)` eine garantierte Einschließung von

$$y = \ln(1+x) \approx 10^{-623456789}/3$$

in hoher Genauigkeit berechnet. Die Anweisungen:

```
stagprec = 20;
lx_interval X,Y;
X = lx_interval(-623456789,"[1,1]")/3;
Y = Lnp1( X );
```

liefern mit Y die gewünschte Einschließung:

$$\ln(1+x) \in \mathtt{Y} = 10^{-623456790} \cdot \underbrace{3.333333333333\ldots 3333333333}_{300\text{ korrekten Dez.-Ziffern}}{}^{492\ldots}_{168\ldots} \cdot 10^{0}.$$

mit 300 korrekten Dezimalziffern. Schließt man jetzt den sehr kleinen Funktionswert $y \approx 10^{-623456789}/3$ mit einem Intervall y vom Typ `l_interval` ein, so erhält man mit $\mathtt{y} = [0, 4.94065\ldots \cdot 10^{-324}$ eine sehr grobe und völlig unbrauchbare Einschließung des Funktionswertes $y = \ln(1+x)$.

Mit Hilfe der `Lnp1`-Funktion kann jetzt die Logarithmus-Funktion

```
lx_interval ln(const lx_interval &x);
```

implementiert werden. Gilt zunächst $\mathtt{x} \approx 1$, d.h. $|\mathtt{x} - 1| \leq 10^{-7}$, so wird $\ln(\mathtt{x})$ eingeschlossen durch: $\ln(\mathtt{x}) \subseteq \mathtt{Lnp1}(\mathtt{x} \diamond 1)$. Im Fall $|\mathtt{x} - 1| > 10^{-7}$ geht man aus von der Identität

$$\begin{aligned} \ln(x) &= \ln\left((x \cdot 2^{-n}) \cdot 2^{n}\right), \quad x \in \mathbb{R},\ n \in \mathbb{Z}, \\ &= \ln(t) + n \cdot \ln(2); \quad t := x \cdot 2^{-n} \in \mathbb{R}. \end{aligned}$$

Beim ersten reduzierten Argument $t = x \cdot 2^{-n}$ wird n so gewählt, dass $t \approx 1$ möglichst gut erfüllt ist, um $\ln(t)$ mit Hilfe der `Lnp1()`-Funktion möglichst effektiv auswerten zu können. Im Fall $n \leq 0$ wird $n = 0$ und $t = x$ gesetzt. Zu beachten ist, dass $t = x \cdot 2^{-n}$ in der Klasse `lx_interval` intervallmäßig sehr schnell und rundungsfehlerfrei berechnet werden kann.

Um $t \approx 1$ weiter zu verbessern, benutzt man

$$\ln(t) = 2^k \cdot \ln\left(\sqrt[2^k]{t}\right), \quad \text{mit} \quad \lim_{k\to\infty} \sqrt[2^k]{t} = 1,$$

d.h. für hinreichend großes $k \in \mathbb{N}$ liegt dann das zweite reduzierte Argument $u := \sqrt[2^k]{t}$ hinreichend dicht bei 1. Für z.B. $k = 22$ wäre die Berechnung von u sehr aufwendig, wenn u mit Hilfe einer 2^k-ten Wurzel berechnet werden sollte. Sehr viel effektiver ist jedoch die folgende Schleife

```
u = t;
for (int j=1; j<=22; j++)
    u = sqrt(u);
```

die mit dem letzten `u` den Wert $\sqrt[2^k]{t} \approx$ `u` sehr gut approximiert.

Zur Berechnung von k wird zunächst der Zweier-Exponent m von $x = M \cdot 2^m$, mit $0.5 \leq M < 1$ berechnet. Wählt man dann $n := m$, so gilt $t = M$. Setzt man $t = 1 - \varepsilon$, so erhält man $0 < \varepsilon \leq 0.5$ und damit in guter Näherung

$$u = \sqrt[2^k]{t} = \sqrt[2^k]{1-\varepsilon} \approx 1 - \frac{\varepsilon}{2^k}.$$

Um mit $u = 1 + \delta$ die Funktion $\ln(u) = \ln(1 + \delta)$ effektiv auswerten zu können, verlangt man z.B.

$$\frac{\varepsilon}{2^k} \leq 10^{-6} \quad \Longleftrightarrow \quad \varepsilon \leq 2^k \cdot 10^{-6}$$
$$\Longleftrightarrow \quad k \geq \frac{\ln(\varepsilon) + 6 \cdot \ln(10)}{\ln(2)}.$$

Für $\varepsilon \ll 1$ wird der obige Bruch negativ und somit $k = 0$ gesetzt, d.h. die 2. Argumentreduktion $u = \sqrt[2^k]{t}$ entfällt. Im Fall $k \geq 1$ gilt

$$\ln(x) = n \cdot \ln(2) + 2^k \cdot \ln(u), \quad t = x \cdot 2^{-n},\ u = \sqrt[2^k]{t},\ k = 1, 2, 3, \ldots \tag{29}$$

und $\ln(u) = \ln(1 + (u - 1))$. Um $\ln(x)$ einzuschließen, muss die rechte Seite von (29) intervallmäßig ausgewertet werden:

$$\ln(x) \in n \diamond <\ln(2)> \oplus 2^k \diamond \texttt{Lnp1}(\texttt{u} \diamond 1), \ \forall x \in \texttt{x}.$$

$<\ln(2)>$ bedeutet dabei eine garantierte Einschließung von $\ln(2)$ in hoher Genauigkeit und wird realisiert durch die C-XSC Intervallkonstante `Ln2_lx_interval()`. `u` ist das Ergebnis der intervallmäßig ausgewerteten `for`-Schleife, wobei vor dieser Schleife $t \in \mathbb{R}$ durch $\texttt{x} \diamond 2^{-n}$ zu ersetzen ist. Die Multiplikation mit 2^k erfolgt rundungsfehlerfrei und sehr effektiv mit Hilfe der Funktion `times2pown(...)`.

Im **1. Beispiel** wird für $x = 1 + 2^{-1074}$ der Funktionswert $y = \ln(x)$ mit folgendem Programm in hoher Genauigkeit eingeschlossen:

```
// Programm  lx_test9.cpp, Test der Funktion:
// lx_interval ln(const lx_interval &x);

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    l_interval x,y;
    lx_interval X,Y;
    string s;

    x = l_interval(1) + minreal;
    X = lx_interval(0,x);

    Y = ln( X );

    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "Y = " << Y << endl;
    y = Y;
    cout << "y = " << y << endl;
    s << Y;
    cout << "Y = " << s << endl;
}
```

Das Programm liefert mit `Y` vom Typ `lx_interval` die gewünschte Einschließung

$$\ln(x) \in \mathtt{Y} = 10^{-324} \cdot \underbrace{4.94065645841246544 17 \ldots 2385655911710266 58}_{297 \text{ Dez.-Ziffern}} {}^{570\ldots}_{401\ldots} \cdot 10^{0}$$

mit 297 korrekten Dezimalziffern. Im Gegensatz dazu erhält man mit dem Intervall `y` vom Typ `l_interval` die sehr grobe Einschließung: $\ln(x) \in \mathtt{y} = [0, 9.881312 \ldots \cdot 10^{-324}]$. Wählt man `stagprec = 39` und gibt man `Y` mit der Anweisung `s << Y;` über den String `s` aus, so erkennt man, dass `Y` intern sogar mit 477 korrekten Dezimalstellen berechnet wird. Bei der obigen lesbaren dezimalen Ausgabe von `Y` können jedoch wegen notwendiger logarithmischer Umrechnungen nur 297 Dezimalstellen angegeben werden, was in der Praxis meist völlig ausreichend ist.

Im **2. Beispiel** liefert das gleiche Programm mit `stagprec` $= 39$ und der Anweisung `x = l_interval(2);` für $y = \ln(2)$ die Einschließung `Y` in dezimaler Form:

$$\ln(2) \in \mathtt{Y} = 10^{-1} \cdot (\underbrace{6.931471805599453\ldots39157814952043740 4}_{301 \text{ korrekte Dez.-Ziffern}}{}^{333\ldots}_{282\ldots} \cdot 10^{0})$$

mit jeweils 301 korrekten Dezimalziffern. Gibt man jedoch `Y` über den String `s` aus, so kann man eine interne Genauigkeit von 472 korrekten Dezimalstellen ablesen.

Im **3. Beispiel** wird für $x = 1 + 10^{-500}/3$ der Funktionswert $y = \ln(x)$ mit folgendem Programm eingeschlossen.

```
// Programm  lx_test10.cpp;
#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 19;
    lx_interval Y;

    Y = ln( 1+lx_interval(-500,"[1,1]")/3 );
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "Y = " << Y << endl;
}
```

Das Programm liefert mit `Y` vom Typ `lx_interval` die folgende Einschließung

$$\ln(x) \in \mathtt{Y} = 10^{-501} \cdot \underbrace{3.3333333333333333\ldots3333333333333}_{128 \text{ Dez.-Ziffern}}{}^{691\ldots}_{290\ldots} \cdot 10^{0}$$

mit nur 128 korrekten Dezimalziffern, da $\ln(x)$ über die Beziehung

$$\ln(x) = \ln(1 + (x - 1)) \in \mathtt{Lnp1}(\boldsymbol{x} \diamondsuit 1)$$

berechnet wird und bei der Auswertung des Arguments $\boldsymbol{x} \diamondsuit 1$ wegen $\boldsymbol{x} \approx [1, 1]$ mit entsprechender Auslöschung zu rechnen ist. Mit der neuen Anweisung

```
Y = ln( 1+lx_interval(-628;"[1,1]")/3 )
```

wird diese Auslöschung weiter verstärkt und man erhält

$$\ln(x) \in \mathtt{Y} = 10^{-629} \cdot [3.1001177993\ldots, 3.5178641694347] \cdot 10^{0}$$

mit nur noch einer einzigen korrekten Dezimalziffer. Liegt das nicht darstellbare x noch dichter bei 1, so sollte die folgende Funktion $\ln(1 + x)$ zur Anwendung kommen.

5.2.11 Die Funktion $\ln(1+x)$

Mit Hilfe der `Lnp1(...)`-Funktion, vgl. Seite 48, ist die Implementierung der Funktion

```
lx_interval lnp1(const lx_interval &x);
```

zur Einschließung der Funktionswerte $y = \ln(1 + \boldsymbol{x})$ denkbar einfach:

```
lx_interval lnp1(const lx_interval &x)
{
    const real r = 1e-7;
    int stagsave = stagprec,
       stagmax = 39;
    if (stagprec>stagmax) stagprec = stagmax;
    lx_interval res;

    res = (Inf(x)>-r && Sup(x)<r)? Lnp1(x) : ln(1.0+x);

    stagprec = stagsave;
    res = adjust(res);

    return res;
}
```

Im **1. Beispiel** wird mit $x = 10^{-630}/3$ für den Funktionswert $y = \ln(1+x)$ eine garantierte Einschließung in hoher Genauigkeit berechnet. Das nachfolgende Programm

```
// Programm  lx_test11.cpp: Zum Test der
// Funktion  ln(1+x).

#include <iostream>       // Wegen cout
#include "lx_interval.hpp"
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 19;
    lx_interval Y;

    Y = lnp1( lx_interval(-630,"[1,1]")/3 );

    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "Y = " << Y << endl;
}
```

liefert mit Y die Einschließung

$$\ln(1+x) \in \mathtt{Y} = 10^{-631} \cdot \underbrace{3.3333333333333333\ldots3333333333333}_{300 \text{ korrekte Dez.-Ziffern}} {}^{481\ldots}_{186\ldots} \cdot 10^{0}$$

mit 300 korrekten Dezimalziffern.

Im **2. Beispiel** wird eine garantierte Einschließung von $y = (1+10^{-600})^{10^{600}}$ berechnet. Benutzt wird die Identität

$$y = (1+10^{-600})^{10^{600}} \equiv e^{10^{600}\cdot\ln(1+10^{-600})},$$

und mit den neuen Anweisungen

```
Y = lx_interval(600,"[1,1]"); // 10^600;
Y = exp(Y * lnp1( lx_interval(-600,"[1,1]") ));
```

liefert das obige Programm die Einschließung

$$y = (1+10^{-600})^{10^{600}} \in \mathtt{Y} = 10^{-1} \cdot \underbrace{2.7182818284590452\ldots753907774}_{294 \text{ korrekte Dez.-Ziffern}} {}^{50019\ldots}_{498\ldots} \cdot 10^{1}$$

mit 294 korrekten Dezimalziffern. Es ist zu beachten, dass neben 10^{+600} auch 10^{-600} nicht exakt darstellbar ist und mit dem Konstruktor `lx_interval(-600,"[1,1]")` durch ein echtes Intervall eingeschlossen wird. Selbst wenn man daher die obige Einschließung von y mit der größeren Präzision `stagprec` $= 39$ berechnet, so kann man keine größere Genauigkeit erwarten. Tatsächlich erhält man bei dieser Präzision auch *nur* eine interne Genauigkeit von 299 korrekten Dezimalziffern. Zu beachten ist ferner, dass die Berechnung von y mit `Mathematica` wegen Overflow scheitert und daher nur gelingt, wenn das Argument a der Exponentialfunktion zunächst separat ausgewertet und erst danach `Exp[a]` berechnet wird, $\quad a := 10^{600} \cdot \ln(1+10^{-600}) \approx 1.$

5.2.12 Die Funktion x^y

Die C-XSC Funktion

```
lx_interval pow(const lx_interval &x, const lx_interval &y)
```

ist ganz analog zur gleichnamigen Funktion der Klasse `l_interval` implementiert, so dass der Algorithmus hier nicht ausführlich diskutiert werden muss.

Gilt $\mathtt{y} \in \mathbb{Z}$ und $|\mathtt{y}| < 2147483647$, so kommt die Funktion `power(x,n)` von Seite 42 zur Anwendung, andernfalls wird folgende Identität benutzt:

$$x^y = e^{y \cdot \ln(x)}, \quad \mathtt{Inf}(x) > 0.$$

Wegen des großen Definitionsbereiches der benutzten Exponentialfunktion $\mathtt{exp}(x)$, mit $|x| \leq 1488521882$, vgl. Seite 44, wird auch die Potenzfunktion `pow(x,y)` nur in ganz extremen Ausnahmefällen einen Overflow liefern.

Im **1. Beispiel** wird mit dem folgenden Programm eine garantierte Einschließung der Potenz $y = 1.1^{201000000}$ berechnet.

```
// Programm lx_test12.cpp zum Test der
// Potenzfunktion pow(x,y):

#include <iostream>
#include "lx_interval.hpp"
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
   stagprec = 20;
   lx_interval X,E,Y;

   X = lx_interval(0,"[1.1,1.1]");
   E = lx_interval(8,"[2.01,2.01]");
   Y = pow(X,E);

   cout << SetDotPrecision(16*stagprec,16*stagprec)
        << Scientific;
   cout << "Y = " << Y << endl;
}
```

Das obige Programm liefert mit dem Intervall Y die gewünschte Einschließung:

$$y = 1.1^{201000000} \in \mathtt{Y} = 10^{8319928} \cdot \underbrace{5.209586256931927\ldots5628903152655110}_{292 \text{ korrekte Dez.-Ziffern}} \,{}^{683\ldots}_{361\ldots} \cdot 10^{1}.$$

Im **2. Beispiel** wird mit dem folgenden Programm die Identität

$$\left[\left(\frac{5}{3}\right)^n\right]^{1/n} = \frac{5}{3} = 1.6666\ldots, \quad \text{mit} \quad n = 1071000000$$

überprüft.

```
// Programm lx_test13.cpp zum Test der
// Potenzfunktion pow(x,y):

#include <iostream>       // Wegen cout
#include "lx_interval.hpp"
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 20;
    lx_interval X,N,Y;

    X = lx_interval(0,"[5,5]")/3;
    N = lx_interval(0,"[1.071e9,1.071e9]");
    Y = pow(X,N);  // (5/3)^(1071000000) < Y
    Y = pow(Y,1/N); // 5/3 < Y;

    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "Y = " << Y << endl;
}
```

Das obige Programm liefert mit dem Intervall Y die folgende Einschließung

$$\frac{5}{3} \in \mathsf{Y} = 10^{-1} \cdot \underbrace{1.666666666666\ldots6666666666666}_{302 \text{ korrekte Dez.-Ziffern}} {}^{954\ldots}_{347\ldots} \cdot 10^{1}.$$

mit 302 korrekten Dezimalziffern.
In der folgenden **Übungsaufgabe** wird die bekannte Potenzregel benutzt:

$$(a \cdot b)^y = a^y \cdot b^y, \quad a, b > 0,\ y \in \mathbb{R}.$$

Wählen Sie $a = 2.1$ und $b = 1/2.1$ und schreiben Sie ein Programm, das für $(a \cdot b)^y$ und $a^y \cdot b^y$ mit $y = 50000.1$ Einschließungen berechnet und zeigen Sie, dass dies sehr enge Einschließungen von 1.0 sind. Überlegen Sie vorher, welches Intervall die 1 am engsten einschließen wird.

Beachten Sie, dass mit der Potenzfunktion garantierte Einschließungen von $\sqrt[y]{x},\ n = 3, 4, 5, \ldots$ z.B. für $x = 5.12 \cdot 10^{200000}$ problemlos in hoher Genauigkeit berechnet werden können.

5.2.13 Die Funktion $(1+x)^y$

Die C-XSC Funktion

```
lx_interval xp1_pow_y(const lx_interval&x, const lx_interval&y);
```

berechnet eine Einschließung der Funktionswerte

$$f(x) = (1+x)^y, \quad \texttt{Inf}(1+x) > 0.$$

Der Algorithmus benutzt die Identität: $f(x) = (1+x)^y = e^{y \cdot \ln(1+x)}$, wobei die Funktionswerte $\ln(1+x)$ mit Hilfe der C-XSC Funktion `lnp1(x)` eingeschlossen werden.

Im **1. Beispiel** wird eine garantierte Einschließung von

$$(1+10^{-600000000})^{10^{+600000000}} \approx e = 2.71828182845\ldots$$

berechnet. Das folgende Programm

```
// Programm  lx_test14.cpp;
// Zum Test der Funktion  (1+x)^y;

#include <iostream>       // Wegen cout
#include "lx_interval.hpp"
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 20;
    lx_interval X,Y,A;

    X = lx_interval(-600000000,"[1,1]");
    Y = lx_interval(+600000000,"[1,1]");
    A = xp1_pow_y(X,Y);

    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "A = " << A << endl;
}
```

liefert mit A die gewünschte garantierte Einschließung:

$$(1+10^{-600000000})^{10^{+600000000}} \in \mathsf{A} = 10^{-1} \cdot \underbrace{2.718281828459\ldots0777449920}_{299 \text{ korrekte Dez.-Ziffern}} {}^{736\ldots}_{660\ldots} \cdot 10^{1}.$$

Vergleichen Sie dazu auch das Beispiel von Seite 57. Versuchen Sie, die obige Potenz mit dem Algebra-System *Mathematica* zu berechnen!

5.2.14 Die Funktion $e^x - 1$

Die C-XSC Funktion

```
lx_interval expm1(const lx_interval &x);
```

berechnet eine Einschließung der Funktionswerte

$$f(x) = e^x - 1.$$

Für $x \to 0$ wird die Taylorreihe

$$e^x - 1 = x \cdot \sum_{k=0}^{\infty} \frac{x^k}{(k+1)!} = x \cdot P_\infty(x).$$

benutzt, die zunächst in der C-XSC Funktion `Expm1(...)` realisiert wird. Mit

$$P_N(x) := \sum_{k=0}^{N} \frac{x^k}{(k+1)!} = 1 + \frac{x}{2} + \frac{x^2}{6} + \frac{x^3}{24} + \ldots$$

wird der Betrag des absoluten Approximationsfehler $\delta(x, N)$ definiert und abgeschätzt durch

$$\begin{aligned} |\delta(x,N)| &:= |P_\infty(x) - P_N(x)| = \left| \sum_{k=N+1}^{\infty} \frac{x^k}{(k+1)!} \right| \\ &= \left| \frac{x^{N+1}}{(N+2)!} \cdot \left\{ 1 + \frac{(N+2)!}{(N+3)!} \cdot x^1 + \frac{(N+2)!}{(N+4)!} \cdot x^2 + \ldots \right\} \right| \\ &\leq \frac{|x|^{N+1}}{(N+2)!} \cdot \left\{ 1 + |x|^1 + |x|^2 + \ldots \right\} = \frac{|x|^{N+1}}{(N+2)!} \cdot \frac{1}{1-|x|}, \quad |x| < 1. \end{aligned}$$

Wir betrachten jetzt ein Intervall $\boldsymbol{x}$, mit[12] $|\boldsymbol{x}| \ll 1$. Mit $R := |\boldsymbol{x}|$ gilt dann für den Betrag des absoluten Fehlers $\delta(x, N)$ die Abschätzung

$$|\delta(x,N)| = |P_\infty(x) - P_N(x)| \leq \frac{R^{N+1}}{(N+2)!} \cdot \frac{1}{1-R} =: \Delta, \quad \forall x \in \boldsymbol{x}. \tag{30}$$

Wir kommen jetzt zur Berechnung des geeigneten Polynomgrades N, wenn das Intervall $\boldsymbol{x}$ und die mit p bezeichnete Präzision von $\boldsymbol{x}$ vorgegeben sind. Wegen $P_N(\boldsymbol{x}) \approx 1$ sollte bei gegebener Präzision p die Schranke Δ des absoluten Approximationsfehlers kleiner bleiben als $1 \cdot 2^{-53 \cdot p}$:

$$\frac{R^{N+1}}{(N+2)!} \cdot \frac{1}{1-R} \leq 2^{-53 \cdot p}, \quad R = |\boldsymbol{x}| < 1.$$

Wegen $|\boldsymbol{x}| \ll 1$ kann dies weiter vereinfacht werden zu

$$\frac{R^{N+1}}{(N+2)!} \leq 2^{-53 \cdot p}, \quad R = |\boldsymbol{x}| \ll 1, \tag{31}$$

[12] $|\boldsymbol{x}| := \max_{r \in \boldsymbol{x}} \{|r|\}$

wobei der linke Term in (31) monoton fallend in N ist. Da $2^{-53 \cdot p}$ schon für $p = 21$ in $S(2, 53)$ nicht mehr darstellbar ist, muss die Berechnung von N nach (31) logarithmisch erfolgen:

$$(N + 1) \cdot \ln(R) - \ln(N + 2)! \leq -53 \cdot p \cdot \ln(2).$$

Mit $R = |\boldsymbol{x}|,\ ex = \texttt{expo\_gr}(\boldsymbol{x}),\ \texttt{expo}(\boldsymbol{x}) = \boldsymbol{x}.ex$ gilt: $R \approx 2^{ex+\texttt{expo}(\boldsymbol{x})} = 2^m$ und damit

$$(N + 1) \cdot m \cdot \ln(2) - \ln(N + 2)! \leq -53 \cdot p \cdot \ln(2). \tag{32}$$

Mit der Bezeichnung $D(N) := (N + 1) \cdot m \cdot \ln(2) - \ln(N + 2)!$ und mit der Beziehung

$$\ln(N + 1)! = \ln N! + \ln(N + 1), \quad N = 0, 1, 2, \ldots$$

erhält man die Rekursionsformel

$$D(N + 1) = D(N) + m \cdot \ln(2) - \ln(N + 3), \quad N = 0, 1, 2, 3, \ldots \tag{33}$$

die sehr einfach auszuwerten ist, wenn die Werte $\ln(N+3)$ zusätzlich als einfache `real`-Konstanten gespeichert werden. Nach (32) lautet damit die Bedingung zur Bestimmung des Polynomgrads N:

$$D(N) \leq -53 \cdot p \cdot \ln(2). \tag{34}$$

Man startet mit $N = 0$ und vergrößert N schrittweise um 1, bis (34) erfüllt ist. Damit erhält man einen Polynomgrad, der praktisch ohne Überschätzungen berechnet wird und damit auch die Laufzeit bei der Horner-Auswertung von $P_N(x)$ optimiert.

Zur Berechnung einer garantierten Oberschranke des absoluten Approximationsfehlers ist nach (30) Δ intervallmäßig auszuwerten und vom entsprechenden Intervallausdruck die Oberschranke S zu bilden. Bezeichnen wir mit $P_N(\boldsymbol{x})$ den auf der Maschine intervallmäßig ausgewerteten Polynomausdruck $P_N(x)$, so gilt:

$$P_N(x) \in P_N(\boldsymbol{x}) \qquad \text{und} \qquad P_\infty(x) \in P_N(\boldsymbol{x}) \oplus [-S, +S] \quad \forall x \in \boldsymbol{x}.$$

Die Einschließung der Funktionswerte $f(x) = e^x - 1$ wird daher realisiert durch:

$$e^x - 1 \in \texttt{expm1}(\boldsymbol{x}) := \begin{cases} \boldsymbol{x} \diamond (P_N(\boldsymbol{x}) \oplus [-S, +S]), & \forall x \in \boldsymbol{x}, \quad |\boldsymbol{x}| < 10^{-7}, \\ \texttt{exp}(\boldsymbol{x}) \ominus 1, & \forall x \in \boldsymbol{x}, \quad |\boldsymbol{x}| \geq 10^{-7}. \end{cases}$$

Für $\boldsymbol{x} = [9.9999 \cdot 10^{-8}, 9.9999 \cdot 10^{-8}]$ und mit der Präzision $p = 40$ benötigt man den maximalen Polynomgrad $N_{max} = 75$. Dieser lässt sich zwar verkleinern, wenn die Grenze 10^{-7} ebenfalls, z.B. auf 10^{-8} mit $N = 68$, verkleinert wird, aber wegen wachsender Auslöschungseffekte geht bei der Auswertung von $\texttt{exp}(\boldsymbol{x}) \ominus 1$ zunehmend Genauigkeit verloren, so dass die Wahl der Grenze 10^{-7} einen guten Kompromiss darstellt. Beachten Sie bitte, dass wegen der intervallmäßigen Auswertung von $P_N(x)$ die Einschließung $P_N(x) \in P_N(\boldsymbol{x})$ automatisch erfüllt ist und damit die Berechnung eines Auswertefehlers überflüssig ist.

Im **1. Beispiel** wird für das Punktintervall $\boldsymbol{x} = [2^{-123456789}, 2^{-123456789}]$ mit der Präzision `stagprec` $= 40$ eine Einschließung von $y = e^x - 1$ berechnet. Das folgende Programm

```
// Programm  lx_test16.cpp;
// Zum Test der Funktion  e^x - 1;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 40;
    lx_interval X,Y;
    string s;

    X = lx_interval(-123456789,l_interval(1.0));
    Y = expm1(X);

    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    s << Y;
    cout << "expm1(X) = " << s << endl;
}
```

liefert mit der Zeichenkette `s` für den Funktionswert $e^x - 1$ die folgende Einschließung

$$e^x - 1 \in 2^{-123457809} \cdot [\underbrace{1.12355\ldots7599\ldots999}_{631 \text{ korrekte Dez.-Ziffern}}505, 1.12355\ldots7600\ldots000494] \cdot 10^{307}.$$

mit 631 von maximal $308 + 324 = 632$ möglichen korrekten Dezimalziffern. Das Beispiel ist zwar rein akademisch, zeigt jedoch die interne maximale Genauigkeit bei der Einschließung von $e^x - 1$. Zu beachten ist dabei, dass das obige Punktintervall $\boldsymbol{x} = [2^{-123456789}, 2^{-123456789}]$ in der Klasse `lx_interval` *exakt* darstellbar ist. Wählt man jedoch ein leicht aufgeblähtes Intervall $\boldsymbol{x}$, so kann die Genauigkeit der Einschließung von $e^{\boldsymbol{x}} - 1 \approx \boldsymbol{x}$ natürlich nicht größer sein als die von $\boldsymbol{x}$ selbst!

Im **2. Beispiel** wird die Identität

$$[(e^x - 1) + 1] \cdot e^{-x} \equiv 1 \tag{35}$$

mit dem sehr kleinen Wert $x = 10^{-234567890}$ getestet. Dabei wird die linke Seite in (35) intervallmäßig mit der Präzision `stagprec = 19` ausgewertet, so dass eine Einschließung von 1 berechnet wird, die möglichst eng sein sollte. Das folgende C-XSC Programm

```
// Programm  lx_test17.cpp;
// Zum Test der Funktion  e^x - 1;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 19;
    lx_interval X,Y;

    X = lx_interval(-234567890,"[1,1]");
    Y = (expm1(X) + 1.0)*exp(-X);

    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "1 enthalten in: " << Y << endl;
}
```

liefert mit Y die gewünschte Einschließung

$$1 \in \mathtt{Y} = 10^{-1} \cdot [\underbrace{9.999999\ldots9999999}_{302 \text{ korrekte Dez.-Ziffern}}038\ldots, 1.000000\ldots00000103\ldots \cdot 10^{1}]$$

mit 302 korrekten Dezimalziffern. Beachten Sie bitte, dass X jetzt kein Punktintervall ist, da $10^{-234567890}$ als Objekt der Klasse `lx_interval` nicht exakt darstellbar ist. Im Gegensatz zur Klasse `l_interval` fällt jetzt die Einschließung von $e^{\boldsymbol{x}} - 1$ *nicht* in den Unterlaufbereich und wird in der Klasse `lx_interval` mit Hilfe der C-XSC Funktion `expm1(X)` in hoher Genauigkeit berechnet.

5.2.15 Die Funktion `sin(x)`

Die C-XSC Funktion

```
lx_interval sin(const lx_interval &x);
```

berechnet mit `y = sin(x)` eine garantierte Einschließung aller Funktionswerte

$$f(x) = \sin(x) \in \mathsf{y}, \quad \forall x \in \mathsf{x}.$$

Für die Auswertung auf dem Rechner benutzt man die 2π-Periodizität

$$\sin(x) \equiv \sin(x - m \cdot 2\pi), \quad m \in \mathbb{Z}, \tag{36}$$

und mit[13] $m := \text{floor}(x/2\pi)$ gilt für das reduzierte Argument $t := x - m \cdot 2\pi$

$$\sin(x) \equiv \sin(t), \quad 0 \le t \le 2\pi. \tag{37}$$

Wir betrachten jetzt ein Intervall $\boldsymbol{x}$ und definieren damit das reduzierte Intervallargument $\boldsymbol{t} := \boldsymbol{x} - m \cdot 2\pi$, wobei noch zu klären ist, für welches spezielle $x \in \boldsymbol{x}$ die ganze Zahl $m := \text{floor}(x/2\pi)$ zu berechnen ist. Wählt man z.B. $x = \mathtt{Inf}(\boldsymbol{x})$, so muss insbesondere bei breiten Intervallen die Beziehung $|\boldsymbol{t}| \le 2\pi$ nicht mehr erfüllt sein, wenn m etwa um 1 zu groß oder zu klein ausfällt. Aber dies ist für die Berechnung einer Einschließung von $\sin(\boldsymbol{x})$ belanglos, da die Beziehung $\sin(\boldsymbol{x}) \equiv \sin(\boldsymbol{x} - m \cdot 2\pi)$ für *jedes* $m \in \mathbb{Z}$ erfüllt ist. Es gilt damit

$$\sin(x) \in \sin(\boldsymbol{x}) = \sin(\boldsymbol{t}), \quad \boldsymbol{t} := \boldsymbol{x} - m \cdot 2\pi, \quad \forall x \in \boldsymbol{x}.$$

Des weiteren muss geklärt werden, wie $\boldsymbol{t}$ auf der Maschine durch ein Intervall `T` vom Typ `lx_interval` einzuschließen ist. Um nicht zu große Werte von `|x|` zu benutzen, wird im ersten Schritt durch `xl = x;` das Eingangsintervall `x` vom Typ `lx_interval` durch ein Intervall `xl` vom Typ `l_interval` eingeschlossen. Die Argumentintervalle müssen daher $|\mathsf{x}| < 10^{+308}$ erfüllen, was in der Praxis jedoch keine wirkliche Einschränkung sein dürfte. Das Argument $(\mathtt{Inf(x)}/2\pi)$ der obigen floor-Funktion kann jetzt mit $\mathtt{Inf(x)}/2\pi \approx \mathsf{u} := \mathtt{Inf(x)} \oslash \mathtt{Inf(Pi2\_l\_interval())}$ sehr gut approximiert werden, wobei `Pi2_l_interval()` eine C-XSC Intervall-Konstante ist, die 2π optimal einschließt. Das Argument `u` liefert dann mit der C-XSC Funktion

```
l_real floor(const l_real &u);
```

den geeigneten Wert $m \in \mathbb{Z}$. Im nächsten Schritt wird dann $\boldsymbol{t} := \mathsf{x} - m \cdot 2\pi$ eingeschlossen durch `T` vom Typ `lx_interval`:

$$\boldsymbol{t} := \mathsf{x} - m \cdot 2\pi \subset \mathsf{T} := \mathsf{x} \diamondminus m \diamondsuit (2 \cdot \mathtt{Pi\_lx\_interval()})$$

wobei das Intervall $2 \cdot$ `Pi_lx_interval()` den Wert 2π in hoher Genauigkeit mit ca. 630 korrekten Dezimalstellen einschließt. Diese hohe Genauigkeit ist notwendig, um das reduzierte Argument `T` mit möglichst geringer Überschätzung berechnen zu können.

[13] floor(a) liefert die nächst ganze Zahl $k \in \mathbb{Z}$, mit $k \le a$.

Nach dieser ersten Argumentreduktion gilt dann

(38) $\sin(x) \in \sin(\boldsymbol{x}) \subset \sin(\mathsf{T}), \quad \mathsf{T} := \mathsf{x} \diamondsuit m \diamondsuit (2 \cdot \texttt{Pi\_lx\_interval()})$.

Um nun mit vertretbarem Aufwand eine Einschließung von $\sin(\mathsf{T})$ mit Hilfe einer abgebrochenen Taylorreihe berechnen zu können, ist wegen $|\mathsf{T}| \sim 2\pi$ eine zweite Argumentreduktion notwendig. Wir benutzen dazu

$$\begin{aligned} \sin(x) &= \sin\left(\frac{x}{3}\right) \cdot \left\{3 - 4 \cdot \sin^2\left(\frac{x}{3}\right)\right\}, \\ \sin\left(\frac{x}{3}\right) &= \sin\left(\frac{x}{9}\right) \cdot \left\{3 - 4 \cdot \sin^2\left(\frac{x}{9}\right)\right\}. \end{aligned}$$

Bei vorgegebenem $\sin(x/9)$ kann man also mit der letzten Gleichung zunächst $\sin(x/3)$ berechnen und anschließend mit der ersten Gleichung den Funktionswert $\sin(x)$ selbst. Dies kann verallgemeinert werden:

Gegeben sei $u = \sin(x/3^n)$, dann liefert die Schleife

```
for (int k=1; k<=n; k++)
    u = u * (3-4*u*u);
```

den Wert $u = \sin(x)$. Ersetzt man jetzt x durch unser reduziertes Intervallargument T, so erhält man nach der Inklusions-Monotonie der Intervallrechnung entsprechend:

Gilt $\sin(\mathsf{T}/3^n) \subset \mathsf{u}$, dann liefert die Schleife

```
for (int k=1; k<=n; k++)
    u = u * (3-4*sqr(u));
```

$\sin(\mathsf{T}) \subset u$. Bei der praktischen Anwendung wird also zunächst $n \in \mathbb{N}$ so gewählt, dass $|\mathsf{T}|/3^n \sim 10^{-8}$ erfüllt ist, damit der Polynomgrad N des auszuwertenden Taylor-Polynoms nicht zu groß wird, und nach der obigen Schleife erhält man schließlich die gewünschte garantierte Einschließung für $\sin(\mathsf{T}) \subset u$. In der obigen Schleife sind nur wenige Rechenoperationen notwendig, so dass selbst mit $n_{max} \sim 18$ keine größeren Überschätzungen auftreten, da wegen $|u| \ll 1$ die bekannten Auslöschungseffekte völlig vermieden werden.

Benutzt man bei dieser zweiten Argumentreduktion z.B. die Gleichung

$$\sin(x) = \sin\left(\frac{x}{5}\right) \cdot \left\{5 - 20\sin^2\left(\frac{x}{5}\right) + 16 \cdot \sin^4\left(\frac{x}{5}\right)\right\},$$

so erhält man für die gleiche Genauigkeit, d.h. $|\mathsf{T}|/5^n \sim 10^{-8}$, einen zwar kleineren Wert n_{max}, die Anzahl der notwendigen Grundoperationen ist jedoch in der entsprechenden `for`-Schleife deutlich größer als in der obigen `for`-Schleife bez. $|\mathsf{T}|/3^n \sim 10^{-8}$.

Im nächsten Schritt wird jetzt $n \in \mathbb{N}$ so bestimmt, dass $|\mathsf{T}|/3^n \sim 10^{-8}$ gilt. Dazu schließt man zunächst T durch ein Intervall `z` vom Typ `interval` ein und sucht dann mit $r := \texttt{Sup(abs(z))}$ für n eine optimale Lösung der Gleichung

$$\frac{r}{3^n} = 10^{-8} \quad \text{bzw.} \quad \ln(r) - n \cdot \ln(3) = -8 \cdot \ln(10).$$

Berechnet man mit der Anweisung `p = expo(r)` den Zweier-Exponenten von r, so gilt $1/2 \le r < 2^p$ und man erhält:

$$n = \frac{p \cdot \ln(2) + 8 \cdot \ln(10)}{\ln(3)}.$$

Speichert man $\ln(2), \ln(3), \ln(10)$ als `real`-Konstanten, so kann die obige rechte Seite schnell ausgewertet und zur nächsten ganzen Zahl $n \in \mathbb{Z}$ gerundet werden. Dabei signalisiert $n \le 0$, dass eine zweite Argumentreduktion nicht notwendig ist, da dann $|\mathsf{T}| \le 10^{-8}$ schon vorher erfüllt war. Testrechnungen haben $n \le 18$ ergeben. Um im Fall $n > 0$ das reduzierte Argument $\mathsf{T}/3^n$ effektiv berechnen zu können, werden die ganzzahligen Werte 3^n für $1 \le n \le 20$ als `real`-Konstanten `pot3_n`$[k] = 3^{k+1}, k = 0, 1, \ldots 19$ gespeichert.

Mit dem reduzierten Intervall-Argument $\mathsf{T}_3 := \mathsf{T} \diamond 3^n$ und $|\mathsf{T}_3| \le R \sim 10^{-8}$ wird jetzt mit Hilfe einer abgebrochenen Taylor-Reihe eine Einschließung für $\sin(\mathsf{T}_3)$ berechnet. Für $x \in \mathbb{R}$ gilt:

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

Approximiert man $P_\infty(x^2)$ durch $P_N(x^2) := \displaystyle\sum_{k=0}^{N} \frac{(-1)^k}{(2k+1)!}\, x^{2k}$, so gilt für den Betrag des absoluten Approximationsfehlers δ die Abschätzung:

$$\begin{aligned} |\delta| &= \left|P_\infty(x^{2k}) - P_N(x^{2k})\right| = \left| \sum_{k=N+1}^{\infty} \frac{(-1)^k}{(2k+1)!}\, x^{2k} \right| \\ &= \frac{x^{2N+2}}{(2N+3)!} \cdot \left| 1 - \frac{(2N+3)!}{(2N+5)!}\, x^2 + \frac{(2N+3)!}{(2N+7)!}\, x^4 - + \ldots \right|. \end{aligned}$$

Für $|x| < 1$ ist die letzte Reihe eine alternierende Leibniz-Reihe, deren Wert kleiner oder gleich 1 bleibt, daher gilt

$$|\delta| \le \frac{x^{2N+2}}{(2N+3)!}, \quad |x| < 1.$$

Für ein beliebiges $x \in \mathsf{T}_3$ mit $|\mathsf{T}_3| \le R$ gilt daher

$$|\delta| \le \frac{R^{2N+2}}{(2N+3)!} \le S, \quad \forall x \in \mathsf{T}_3, \qquad \text{und} \tag{39}$$

$$P_\infty(x^2) \in P_N(\mathsf{T}_3 \diamond \mathsf{T}_3) \oplus [-S, +S] \qquad \rightsquigarrow$$

$$\sin(x) \in \sin(\mathsf{T}_3) \subset \mathsf{T}_3 \diamond \left(P_N(\mathsf{T}_3 \diamond \mathsf{T}_3) \oplus [-S, +S]\right), \quad \forall x \in \mathsf{T}_3. \tag{40}$$

In (39) ist S eine Oberschranke des absoluten Approximationsfehlers für alle $x \in \mathsf{T}_3$, und in (40) haben wir die gewünschte Einschließung für $\sin(\mathsf{T}_3)$.

Wir kommen jetzt zur Berechnung des geeigneten Polynomgrades N, wenn T_3 und die mit p bezeichnete Präzision von T_3 vorgegeben sind. Wegen $P_N(\mathsf{T}_3) \approx 1$ sollte

bei gegebener Präzision p die Schranke S des absoluten Approximationsfehlers kleiner bleiben als $1 \cdot 2^{-53 \cdot p}$:

$$\frac{R^{2N+2}}{(2N+3)!} \leq 2^{-53 \cdot p}, \tag{41}$$

wobei der linke Term in (41) monoton fallend in N ist. Da $2^{-53 \cdot p}$ schon für $p = 21$ in $S(2, 53)$ nicht mehr darstellbar ist, muss die Berechnung von N nach (41) logarithmisch erfolgen:

$$(2N+2) \cdot \ln(R) - \ln(2N+3)! \leq -53 \cdot p \cdot \ln(2).$$

Mit $R = |\mathsf{T}_3|$, $ex = \texttt{expo\_gr}(\mathsf{T}_3)$, $\texttt{expo}(\mathsf{T}_3) = \mathsf{T}_3.ex$ gilt: $R \approx 2^{ex+\texttt{expo}(\mathsf{T}_3)} = 2^k$ und damit

$$(2N+2) \cdot k \cdot \ln(2) - \ln(2N+3)! \leq -53 \cdot p \cdot \ln(2).$$

Mit der Bezeichnung $D(N) := (2N+2) \cdot k \cdot \ln(2) - \ln(2N+3)!$ und mit der Beziehung

$$\ln(2N+3+2)! = \ln(2N+3)! + \ln(2N+4) + \ln(2N+5)$$

erhält man die Rekursionsformel

$$D(N+1) = D(N) + 2k \cdot \ln(2) - \ln(2N+4) - \ln(2N+5), \quad N = 0, 1, 2, 3, \ldots$$

die sehr einfach auszuwerten ist, wenn die Werte $\ln(2N+4), \ln(2N+5)$ zusätzlich als einfache `real`-Konstanten gespeichert werden. Die obige Bedingung zur Bestimmung des Polynomgrads N lautet damit:

$$D(N) \leq -53 \cdot p \cdot \ln(2). \tag{42}$$

Man startet die Iteration mit $N = 0$ und vergrößert N schrittweise um 1, bis (42) erfüllt ist. Damit erhält man einen Polynomgrad, der praktisch ohne Überschätzungen berechnet wird und damit auch die Laufzeit bei der Horner-Auswertung von $P_N(\mathsf{T}_3)$ optimiert.

Das in (40) auftretende Polynom $P_N(\mathsf{T}_3 \diamond \mathsf{T}_3)$ muss noch auf der Maschine nach dem Horner-Schema ausgewertet werden. Dazu werden die Polynomkoeffizienten a_k zunächst durch intervallmäßige Auswertung der Rekursionsformel

$$a_k = \frac{-a_{k-1}}{2k \cdot (2k+1)}, \quad a_0 = 1, \quad k = 1, 2, \ldots, N$$

eingeschlossen und in einem entsprechenden Feld bereitgestellt. Nach intervallmäßiger Anwendung des Horner-Schemas gilt dann mit dem Maschinenergebnis $\mathsf{P}_N(\mathsf{T}_3 \diamond \mathsf{T}_3)$

$$P_N(\mathsf{T}_3 \diamond \mathsf{T}_3) \subset \mathsf{P}_N(\mathsf{T}_3 \diamond \mathsf{T}_3) \quad \forall x \in \mathsf{T}_3$$

und nach (40) erhält man

$$\sin(x) \in \sin(\mathsf{T}_3) \subset \mathsf{u} := \mathsf{T}_3 \diamond \left(\mathsf{P}_N(\mathsf{T}_3 \diamond \mathsf{T}_3) \oplus [-S, +S]\right), \quad \forall x \in \mathsf{T}_3. \tag{43}$$

Im Fall $n > 0$ ist jetzt noch mit u aus (43) im letzten Schritt die zweite Argumentreduktion mit Hilfe der zweiten `for`-Schleife auf Seite 66 zu berücksichtigen, (Ergebnisanpassung). Bezeichnet man das Ergebnis dieser Schleife wieder mit u, so gilt $\sin(\mathsf{T}) \subset \mathsf{u}$ und nach (38) erhalten wir die gewünschte Einschließung:

$$\sin(x) \in \sin(\mathsf{x}) \subset \sin(\mathsf{T}) \subset \mathsf{u}, \quad \forall x \in \mathsf{x}.$$

Für ein Eingangsintervall $\boldsymbol{v}$, mit $|\boldsymbol{v}| = R_v$ und hinreichend kleinem $R_v \ll 1$ wird der Polynomgrad $N = 0$, und nach (39) muss dann für den absoluten Approximationsfehler der Ausdruck $R_v^2/3!$ ausgewertet werden. Dabei kommt es bei der Berechnung von R_v^2 für $R_v \to 0$ zu einem internen integer-Überlauf, der jedoch recht einfach vermieden werden kann. Es gilt

$$\frac{\sin(x)}{x} = 1 - \frac{x^2}{3!} + \frac{x^4}{5!} - \frac{x^6}{7!}, \quad x \in \mathbb{R}\backslash\{0\},$$

und da die Reihe oben rechts für $|x| < 1$ eine alternierende Leibniz-Reihe ist, folgt direkt

$$1 - \frac{x^2}{3!} < \frac{\sin(x)}{x} < 1, \quad 0 < |x| < 1, \quad \text{und damit:}$$

$$x \cdot \left(1 - \frac{x^2}{3!}\right) \leq \sin(x) \leq x, \quad \text{falls } 0 \leq x < 1; \tag{44}$$

$$x \leq \sin(x) \leq x \cdot \left(1 - \frac{x^2}{3!}\right), \quad \text{falls } -1 < x \leq 0. \tag{45}$$

Die Einschließungen (44) und (45) sind ideal für $x \to 0$ und kommen zur Anwendung, wenn $R_v \leq 2 \cdot 2^{-1080} = 2^{-1079}$. Bei einer beliebigen Wahl des Intervalls $\boldsymbol{v} = [v_1, v_2]$, mit $|\boldsymbol{v}| = R_v \leq 2^{-1079}$ sind drei Fälle zu unterscheiden:

1. $0 \in \boldsymbol{v}$, d.h. $v_1 \leq 0$ und $v_2 \geq 0$. Mit (44) und (45) folgt dann direkt:

$$\sin(x) \in [v_1, v_2] \ \ \forall x \in \boldsymbol{v}.$$

2. $v_1 > 0$. Nach (44) gilt für $0 < v_1 \leq v_2 < 1$ die Doppelungleichung

$$v_1 \cdot \left(1 - \frac{v_2^2}{3!}\right) \leq \sin(x) \leq v_2.$$

Um die Berechnung von $v_2^2/3!$ wegen drohenden integer-Überlaufs zu vermeiden, muss weiter vereinfacht werden. Mit $a := 1 - 2^{-2097} \in$ `One_m_lx_interval()` verlangen wir:

$$v_1 \cdot a \leq v_1 \cdot \left(1 - \frac{v_2^2}{3!}\right)$$

$$\overset{v_1>0}{\iff} \ 1 - 2^{-2097} \leq 1 - \frac{v_2^2}{3!} \quad \iff \quad \frac{v_2^2}{6} \leq 2^{-2097} \quad \iff \quad v_2^2 \leq 6 \cdot 2^{-2097},$$

und die letzte Ungleichung ist sicher erfüllt, wenn gilt:

$$v_2^2 \leq 2 \cdot 2^{-2097} = 2^{-2096} \quad \iff \quad v_2 < 2^{-1048}.$$

Zusammenfassung:

Im Fall $0 \leq v_1 \leq v_2 < 2^{-1048}$ gilt: $\sin(x) \in [a \cdot v_1, v_2] \ \ \forall x \in \boldsymbol{v} = [v_1, v_2]$.

Zu beachten ist, dass $v_2 < 2^{-1048}$ automatisch erfüllt ist, wenn $R_v \leq 2^{-1079}$ verlangt wird.

3. $v_2 < 0$. Nach (45) gilt für $-1 < v_1 \le v_2 < 0$ die Doppelungleichung

$$v_1 \le \sin(x) \le v_2 \cdot \left(1 - \frac{v_1^2}{3!}\right).$$

Um die Berechnung von $v_1^2/3!$ wegen drohenden integer-Überlaufs zu vermeiden, muss weiter vereinfacht werden. Mit $a := 1-2^{-2097} \in$ `One_m_lx_interval()` verlangen wir:

$$\begin{aligned} & v_2 \cdot \left(1 - \frac{v_1^2}{3!}\right) < v_2 \cdot a \\ \overset{v_2<0}{\Longleftrightarrow} \quad & 1 - \frac{v_1^2}{3!} > 1 - 2^{-2097} \quad \Longleftrightarrow \quad \frac{v_1^2}{3!} < 2^{-2097} \quad \Longleftrightarrow \quad v_1^2 < 6 \cdot 2^{-2097}, \end{aligned}$$

und die letzte Ungleichung ist sicher erfüllt, wenn gilt:

$$v_1^2 < 2 \cdot 2^{-2097} = 2^{-2096} \quad \Longleftrightarrow \quad |v_1| = -v_1 < 2^{-1048} \quad \Longleftrightarrow \quad v_1 > -2^{-1048}.$$

Zusammenfassung:

Im Fall $-2^{-1048} < v_1 \le v_2 \le 0$ gilt: $\sin(x) \in [v_1, a \cdot v_2] \ \forall x \in \boldsymbol{v} = [v_1, v_2]$.

Zu beachten ist, dass $-2^{-1048} < v_1$ automatisch erfüllt ist, wenn $R_v \le 2^{-1079}$ verlangt wird.

Im **1. Beispiel** liefert das Programm `lx_test27.cpp` von Seite 86 für $x = 2^{-2147482624}$ mit den neuen Anweisungen

```
X = lx_interval(-2147482624,l_interval(1));   Y = sin(X);
```

eine Einschließung des Funktionswertes $y = \sin(x)$. Das Programm `lx_test27.cpp` liefert mit Y die folgende optimale Einschließung

$$\sin(x) \in \mathsf{Y} = 2^{-2147483646} \cdot [\underbrace{4.4942328371557\ldots303999\ldots99}_{630 \text{ korrekte Dez.-Ziffern}}80237 \cdot 10^{+307},$$
$$4.4942328371557\ldots304000000000\ldots000 \cdot 10^{+307}]$$

mit 630 korrekten Dezimalziffern. Mit dem etwas kleineren Argument $x = 2^{-2147482625}$ erhält man einen vorzeitigen Programmabbruch wegen eines Überlaufs bei den internen integer-Berechnungen.

Zu beachten ist, dass das Algebra-System *Mathematica* schon mit $x = 2^{-2147482624}$ wegen eines internen Underflows einen entsprechenden Programmabbruch liefert.

Im **2. Beispiel** wird mit dem folgenden C-XSC Programm für das in `lx_interval` exakt darstellbare Argument $x = 10$ eine garantierte Einschließung von $y = \sin(x)$ berechnet.

```
// Programm  lx_test18.cpp;
// Zum Test der sin-Funktion;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;

    X = lx_interval(0,l_interval(10));
    Y = sin(X);

    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "sin(X) = " << Y << endl;
    s << Y;
    cout << "sin(X) = " << s << endl;
}
```

Die erste Ausgabe erfolgt in dezimaler Form und liefert

$$\sin(10) \in \mathsf{Y} = -10^{-1} \cdot \underbrace{5.4402111088936\ldots57964481785854175 8}_{300 \text{ korrekte Dez.-Ziffern}}{}^{893\ldots}_{999\ldots} \cdot 10^{0}.$$

Bei der zweiten Ausgabe wird `Y` vom Typ `lx_interval` zunächst als `(n,Y.li)` in eine Zeichenkette `s` geschrieben, wobei `n` der Zweier-Exponent von `Y` ist. Danach wird diese Zeichenkette in den Ausgabekanal geleitet und auf dem Bildschirm ausgegeben:

$$\sin(10) \in \mathsf{Y} = -2^{-1022} \cdot \underbrace{2.44495754066497701\ldots674445509359467}_{474 \text{ korrekte Dez.-Ziffern}}{}^{284\ldots}_{574\ldots} \cdot 10^{+307}.$$

Mit dem Punktintervall-Argument $\mathsf{X} = 10$ wird daher $\sin(10)$ mit 474 korrekten Dezimalstellen eingeschlossen. Da bei der ersten Ausgabe im Dezimalformat intern auf die Basis 10 logarithmisch umgerechnet werden muss, reduziert sich die Ausgabegenauigkeit dabei jedoch auf 300 Dezimalstellen.

Im **3. Beispiel** wird mit dem folgenden C-XSC Programm für das in `lx_interval` nicht exakt darstellbare Argument $x = 0.1$ eine garantierte Einschließung von $y = \sin(x)$ berechnet.

```
// Programm  lx_test19.cpp;
// Zum Test der sin-Funktion;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 19;
    lx_interval X,Y;

    X = lx_interval(0,"[0.1,0.1]");
    Y = sin(X);

    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "sin(X) = " << Y << endl;
}
```

Das obige C-XSC Programm liefert mit Y bei einer Präzision von `stagprec` $= 19$ die folgende Einschließung

$$\sin(0.1) \in \mathsf{Y} = 10^{-2} \cdot \underbrace{9.9833416646828152\ldots907329845348005814497 39}_{299 \text{ korrekte Dez.-Ziffern}} {}^{988\ldots}_{241\ldots} \cdot 10^{0}$$

mit 299 korrekten Dezimalstellen.

Im **4. Beispiel** wird mit dem folgenden C-XSC Programm für das in `lx_interval` nicht exakt darstellbare sehr kleine Argument $x = 10^{-123456789}/3$ eine garantierte Einschließung von $y = \sin(x)$ in hoher Genauigkeit berechnet.

```
// Programm  lx_test20.cpp;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 19;
    lx_interval X,Y;

    X = lx_interval(-123456789,"[0.1,0.1]");
    Y = sin(X/3);
```

```
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "sin(X) = " << Y << endl;
}
```

Das obige C-XSC Programm liefert mit Y bei einer Präzision von `stagprec` $= 19$ die folgende Einschließung

$$\sin(10^{-123456789}/3) \in \mathsf{Y} = 10^{-123456791} \cdot \underbrace{3.33333333333\ldots333333333333}_{298 \text{ korrekte Dez.-Ziffern}}{}^{448\ldots}_{205\ldots} \cdot 10^{0}$$

mit 298 korrekten Dezimalstellen. Im Gegensatz zur Klasse `l_interval` können wir jetzt mit der Klasse `lx_interval` auch für betragsmäßig sehr kleine Argumente die sin-Funktionswerte mit sehr hoher Genauigkeit einschließen.

Im **5. Beispiel** wählen wir im C-XSC Programm `lx_test18.cpp` von Seite 71 die Präzision `stagprec` $= 38$ und berechnen mit

```
X = lx_interval( Inf(Pi_lx_interval()) );
```

eine Einschließung von $\sin(\mathsf{X})$, wobei das Punktargument $\mathsf{X} < \pi$ sehr dicht bei π liegt. Das Programm liefert mit `Y = sin(X);` jedoch trotz der hohen Präzision eine nur sehr grobe Einschließung des exakten Funktionswertes, da das Intervall Y auch noch die Null mit einschließt, obwohl $\sin(\mathsf{X}) > 0$ gelten muss. Diese grobe Einschließung lässt sich jedoch nicht vermeiden, da π von der Intervall-Konstanten `Pi_l_interval( )` zwar sehr genau, aber eben nicht durch ein Punktintervall eingeschlossen wird.

Wählt man jedoch mit[14]

```
X = lx_interval(0,l_interval( Inf(Pi_l_interval()) ));
```

ein Punktintervall $\mathsf{X} < \pi$, das nicht mehr ganz so dicht bei π liegt, so erhält man mit

```
X = lx_interval(0,l_interval( Inf(Pi_l_interval()) ));
```

die folgende Einschließung mit nur 151 korrekten Dezimalziffern

$$\sin(\mathsf{X}) \subset \mathsf{Y} = 10^{-324} \cdot \underbrace{4.7470749589991443\ldots648736782019562}_{151 \text{ korrekte Dez.-Ziffern}}{}^{750\ldots}_{510\ldots} \cdot 10^{0}.$$

Abschließend noch ein **Hinweis**:
Soll z.B. für $x = \pi + 10^{-20000}/3$ mit

```
X = Pi_l_interval() + lx_interval(-20000,"[1,1]")/3;
```

eine Einschließung für $\sin(x)$ berechnet werden, so fällt diese wiederum sehr grob aus, da auch die Null mit eingeschlossen wird, obwohl $\sin(x) < 0$ gelten muss. Eine sehr gute Einschließung erhält man jedoch bei Anwendung der Beziehung $\sin(\pi + x) \equiv -\sin(x)$, siehe den folgenden Abschnitt.

[14] `Pi_lx_interval( )` und `Pi_l_interval( )` sind Intervallkonstanten, welche in den Klassen `lx_interval` und `l_interval` die Zahl π jeweils optimal einschließen.

5.2.16 Die Funktion $\sin(n\pi + x)$

Die C-XSC Funktion

```
lx_interval sin_n(const lx_interval &x, int n);
```

liefert Einschließungen für $\boldsymbol{y} = \sin(n \cdot \pi + \mathtt{x})$. Die Implementierung beruht auf der für $x \in \mathbb{R}, n \in \mathbb{Z}$ gültigen Beziehung:

$$\sin(n \cdot \pi + x) = \begin{cases} +\sin(x), & \text{falls } n \text{ gerade} \\ -\sin(x), & \text{falls } n \text{ ungerade,} \end{cases}$$

die sich auch direkt auf Intervalle `x` übertragen lässt.

Im **1. Beispiel** wird für $t := 3001\pi + 10^{-20000}/3$ mit dem folgenden Programm eine garantierte Einschließung für $y = \sin(t)$ berechnet.

```
// Programm  lx_test21.cpp;
// Zum Test der sin_n-Funktion;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 19;
    lx_interval X,Y;

    X = lx_interval(-20000,"[1,1]")/3;
    Y = sin_n(X,3001);

    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "sin_n(X) = " << Y << endl;
}
```

Das obige Programm liefert mit `Y` die sehr enge Einschließung

$$\sin(3001\pi + 10^{-20000}/3) \in \mathtt{Y} = -10^{-20001} \cdot \underbrace{3.33333333\ldots333333333}_{300 \text{ korrekte Dez.-Ziffern}}{}^{171\ldots}_{498\ldots} \cdot 10^{0}$$

mit 300 korrekten Dezimalziffern. Berechnet man jedoch eine Einschließung für $\sin(t)$ mit den Anweisungen

```
X = 3001*Pi_l_interval() + lx_interval(-20000,"[1,1]")/3;
                      Y = sin(X);
```

so enthält `Y` die Null und liefert daher eine nur grobe Einschließung für $\sin(t)$. Die sehr starke Überschätzung ergibt sich aus der Tatsache, dass `Pi_lx_interval()` zur Einschließung von π kein Punktintervall sein kann.

5.2.17 Die Funktion `cos(x)`

Die C-XSC Funktion

```
lx_interval cos(const lx_interval &x);
```

berechnet mit `y = cos(x)` eine garantierte Einschließung aller Funktionswerte

$$f(x) = \cos(x) \in \mathsf{y}, \quad \forall x \in \mathsf{x}.$$

Für die Auswertung auf dem Rechner benutzt man die Identität

$$\cos(x) \equiv \sin\left(x + \frac{\pi}{2}\right), \quad x \in \mathbb{R},$$

die sich auch direkt auf Intervalle übertragen lässt.

Im **1. Beispiel** wird mit dem C-XSC Programm `lx_test18.cpp` von Seite 71 für das in `lx_interval` exakt darstellbare Argument $x = 10$ eine garantierte Einschließung von $y = \cos(x)$ berechnet, wenn im Programm `Y = sin(X)` durch `Y = cos(X)` ersetzt wird. Die erste Programmausgabe erfolgt in dezimaler Form und liefert

$$\cos(10) \in \mathsf{Y} = -10^{-1} \cdot \underbrace{8.390715290764524\ldots2332529081478971258}_{301 \text{ korrekte Dez.-Ziffern}} \,{}^{767\ldots}_{827\ldots} \cdot 10^{0}.$$

Bei der zweiten Ausgabe wird `Y` vom Typ `lx_interval` zunächst als `(n,Y.li)` in eine Zeichenkette `s` geschrieben, wobei `n` der Zweier-Exponent von `Y` ist. Danach wird diese Zeichenkette in den Ausgabekanal geleitet und auf dem Bildschirm ausgegeben:

$$\cos(10) \in \mathsf{Y} = -2^{-1022} \cdot \underbrace{3.77098281869791\ldots3022477919027651}_{474 \text{ korrekte Dez.-Ziffern}} \,{}^{001\ldots}_{523\ldots} \cdot 10^{+307}.$$

Mit dem Punktintervall-Argument $\mathsf{X} = 10$ wird daher $\cos(10)$ mit 474 korrekten Dezimalstellen eingeschlossen. Da bei der ersten Ausgabe im Dezimalformat intern auf die Basis 10 logarithmisch umgerechnet werden muss, reduziert sich die Ausgabegenauigkeit dabei jedoch auf 301 Dezimalstellen.

Hinweise:

1. Überträgt man das 4. Beispiel von Seite 73 auf die cos-Funktion bez. $x = \pi/2$, so erhält man sinngemäß die gleichen unvermeidbaren groben Einschließungen.

2. Soll z.B. für $x = \pi/2 + 10^{-20000}/3$ mit

   ```
   X = Pi_l_interval();  times2pown(X,-1);
   X = X + lx_interval(-20000,"[1,1]")/3;
   ```

   eine Einschließung für $\cos(x)$ berechnet werden, so fällt diese wiederum sehr grob aus, da auch die Null mit eingeschlossen wird, obwohl $\cos(x) < 0$ gelten muss. Eine sehr gute Einschließung erhält man jedoch bei Anwendung der Beziehung $\cos(\pi/2 + x) \equiv -\sin(x)$, siehe den folgenden Abschnitt.

5.2.18 Die Funktion $\cos((n+1/2)\pi + x)$

Die C-XSC Funktion

```
lx_interval cos_n(const lx_interval &x, int n);
```

liefert Einschließungen für $\boldsymbol{y} = \cos((n+1/2)\cdot\pi + \mathtt{x})$. Der Algorithmus beruht auf der für $x \in \mathbb{R},\ n \in \mathbb{Z}$ gültigen Beziehung:

$$\cos((n+1/2)\cdot\pi + x) = \begin{cases} +\sin(x), & \text{falls } n \text{ ungerade} \\ -\sin(x), & \text{falls } n \text{ gerade,} \end{cases}$$

die sich auch direkt auf Intervalle `x` übertragen lässt.

Im **1. Beispiel** wird für $t := (3001+1/2)\cdot\pi + 10^{-20000}/3$ mit dem folgenden Programm eine garantierte Einschließung für $y = \cos(t)$ berechnet.

```
// Programm  lx_test22.cpp;
// Zum Test der cos_n-Funktion;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 19;
    lx_interval X,Y;

    X = lx_interval(-20000,"[1,1]")/3;
    Y = cos_n(X,3001);

    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "cos_n(X) = " << Y << endl;
}
```

Das obige Programm liefert mit `Y` die sehr enge Einschließung

$$\cos((3001+1/2)\cdot\pi + 10^{-20000}/3) \in \mathtt{Y} = 10^{-20001}\cdot \underbrace{3.333333\ldots333333}_{300 \text{ korrekte Dez.-Ziffern}}{}^{498\ldots}_{171\ldots} \cdot 10^{0}$$

mit 300 korrekten Dezimalziffern. Berechnet man jedoch eine Einschließung für $\cos(t)$ mit den Anweisungen

```
X = (3001+0.5)*Pi_l_interval() + lx_interval(-20000,"[1,1]")/3;
                          Y = cos(X);
```

so enthält `Y` die Null und liefert daher eine nur grobe Einschließung für $\cos(t) > 0$. Die sehr starke Überschätzung ergibt sich aus der Tatsache, dass `Pi_lx_interval()` zur Einschließung von π kein Punktintervall sein kann.

5.2.19 Die Funktion `tan(x)`

Die C-XSC Funktion

```
lx_interval tan(const lx_interval &x);
```

liefert Einschließungen für $\boldsymbol{y} = \tan(\mathsf{x})$. Die Implementierung beruht auf der Identität:

$$\tan(x) = \frac{\sin(x)}{\cos(x)}, \quad x \in \mathbb{R}\backslash\{x \,|\, x = (k+1/2)\cdot\pi,\ k \in \mathbb{Z}\},$$

die sich auch direkt auf Intervalle `x` übertragen lässt.

Im **1. Beispiel** wird mit dem folgenden C-XSC Programm für das in `lx_interval` exakt darstellbare Argument $x = 10^{135}$ eine sehr enge Einschließung von $y = \tan(x)$ berechnet.

```
// Programm  lx_test23.cpp;
// Zum Test der tan-Funktion;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;

    X = lx_interval(0,"[1e135,1e135]");
    Y = tan(X);

    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "tan(X) = " << Y << endl;
    s << Y;
    cout << "tan(X) = " << s << endl;
}
```

Die erste Ausgabe erfolgt in dezimaler Form und liefert

$$\tan(10^{135}) \in \mathsf{Y} = 10^{-1} \cdot \underbrace{1.4300631917329\ldots926090179129699}_{302 \text{ korrekte Dez.-Ziffern}} {}^{967\ldots}_{800\ldots} \cdot 10^{1}.$$

Bei der zweiten Ausgabe wird `Y` vom Typ `lx_interval` zunächst als `(n,Y.li)` in eine Zeichenkette `s` geschrieben, wobei `n` der Zweier-Exponent von `Y` ist. Danach wird diese Zeichenkette in den Ausgabekanal geleitet und auf dem Bildschirm ausgegeben:

$$\tan(10^{135}) \in \mathsf{Y} = 2^{-511} \cdot \underbrace{9.587006301217887813\ldots7081680175123896}_{473 \text{ korrekte Dez.-Ziffern}} {}^{689\ldots}_{425\ldots} \cdot 10^{+153}.$$

Mit dem Punktintervall-Argument $\mathsf{X} = 10^{135}$ wird daher $\tan(\mathsf{X})$ sehr eng mit 473 korrekten Dezimalstellen eingeschlossen. Da bei der ersten Ausgabe im Dezimalformat intern auf die Basis 10 logarithmisch umgerechnet werden muss, reduziert sich die Ausgabegenauigkeit dabei jedoch auf nur noch 302 Dezimalstellen.

Zu beachten ist ferner, dass mit $|\mathsf{X}| \to \infty$ die Genauigkeit der berechneten Einschließungen Y abnimmt, da π mit der Intervall-Konstanten `Pi_lx_interval()` zwar mit etwa 630 Dezimalstellen sehr genau aber eben doch nur mit einer endlichen Stellenzahl eingeschlossen wird. Für $|\mathsf{X}| = 10^{+300}$ wird $\tan(\mathsf{X}) \subset \mathsf{Y}$ intern z.B. nur noch mit 331 korrekten Dezimalstellen eingeschlossen.

Im **2. Beispiel** wird mit dem folgenden Programm die Identität

$$\frac{\tan(2x) \cdot (1 - \tan^2(x))}{2\tan(x)} \equiv 1, \quad x \in \mathbb{R}\backslash\{x \mid x = k \cdot \pi,\ k \in \mathbb{Z}\}$$

überprüft, indem die obige linke Seite für $x = 10^{+100}$ intervallmäßig ausgewertet wird. Bei einer Präzision `stagprec` $= 39$ kann $x = 10^{+100}$ durch ein Punktintervall X vom Typ `lx_interval` rundungsfehlerfrei eingeschlossen werden.

```
// Programm  lx_test24.cpp;
// Zum Test der tan-Funktion;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y1,Y;
    string s;

    X = lx_interval(0,"[1e100,1e100]");
    Y1 = X;  times2pown(Y1,1);  // Y1 = 2*X;
    Y1 = tan(Y1);      // Y1 = tan(2*X)
    Y = tan(X);
    Y = ( Y1*(1-sqr(Y)) )/(2*Y);

    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "Y = " << Y << endl;
    s << Y;
    cout << "Y = " << s << endl;
}
```

Das obige C-XSC Programm `lx_test24.cpp` liefert mit der Ausgabe über den String `s` eine Einschließung der 1 mit einer internen Genauigkeit von 473 Dezimalstellen.

5.2.20 Die Funktion `cot(x)`

Die C-XSC Funktion

```
lx_interval cot(const lx_interval &x);
```

liefert Einschließungen für $\boldsymbol{y} = \cot(\mathtt{x})$. Die Implementierung beruht auf der Identität:

$$\tan(x) = \frac{\cos(x)}{\sin(x)}, \quad x \in \mathbb{R}\backslash\{x \,|\, x = k \cdot \pi,\ k \in \mathbb{Z}\},$$

die sich auch direkt auf Intervalle `x` übertragen lässt.

Im **1. Beispiel** liefert das folgende Programm für $x = 1.1 \cdot 10^{-200000}$ eine garantierte Einschließung des Funktionswertes $y = \cot(x)$.

```
// Programm  lx_test25.cpp;
// Zum Test der cot-Funktion;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 19;
    lx_interval X,Y;

    X = lx_interval(-200000,"[1.1,1.1]");
    Y = cot(X);

    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "cot(X) = " << Y << endl;
}
```

Das berechnete Intervall `Y` vom Typ `lx_interval` ist die gesuchte Einschließung

$$\cot(1.1 \cdot 10^{-200000}) \in \mathtt{Y} = 10^{199998} \cdot \underbrace{9.090909090909090\ldots909090909090}_{298 \text{ korrekte Dez.-Ziffern}}{}^{916\ldots}_{893\ldots} \cdot 10^{1}$$

mit 298 korrekten Dezimalziffern. Es ist noch zu beachten, dass der obige Konstruktoraufruf

```
X = lx_interval(-200000,"[1.1,1.1]");
```

formal auch ersetzt werden kann durch

```
X = lx_interval(-199700,"[1.1e-300,1.1e-300]");
```

Damit wird jedoch $[1.1 \cdot 10^{-300}, 1.1 \cdot 10^{-300}]$ beim Konstruktoraufruf in der Nähe des Unterlaufbereichs der Klasse `l_interval` intern nur noch mit 24 korrekten Dezimalstellen eingeschlossen, so dass auch die Einschließung von $\cot(x)$ nicht genauer ausfallen kann. Der letzte Konstruktoraufruf sollte daher stets vermieden werden!

5.2.21 Die Funktion $\sqrt{1+x^2}$

Die C-XSC Funktion

```
lx_interval sqrtx2p1(const lx_interval &x);
```

liefert Einschließungen für $\boldsymbol{y} = \sqrt{1+\mathtt{x}^2}$.

Für $x \in \mathbb{R}$ gilt ganz allgemein

$$\sqrt{1+x^2} \equiv \sqrt{1+|x|^2}, \quad \forall x \in \mathbb{R},$$

so dass man sich auf $x \geq 0$ beschränken kann. Im Fall $x \to 0$ beweist man elementar

$$1 \leq \sqrt{1+x^2} \leq 1+x, \quad \forall x \geq 0.$$

Ist nun $\boldsymbol{u}$ ein beliebiges Intervall[15] mit $\mathtt{Inf}(\boldsymbol{u}) \geq 0$, so gilt wegen der Monotonie der Terme in der oberen zweiten Ungleichung

$$\sqrt{1+x^2} \in [1, 1+\mathtt{Sup}(\boldsymbol{u})] = \boldsymbol{v} := [1,1] + [0, \mathtt{Sup}(\boldsymbol{u})], \quad \forall x \in \boldsymbol{u}.$$

Die Einschließung von $\sqrt{1+x^2} \in \boldsymbol{v}$ wird dabei um so enger, je kleiner die Intervallobergrenze $\mathtt{Sup}(\boldsymbol{u}) \geq 0$ gewählt wird. Der Vorteil von $\boldsymbol{v}$ ist nicht nur seine denkbar einfache Struktur, sondern auch die Tatsache, dass in

$$\boldsymbol{v} := [1,1] + [0, \mathtt{Sup}(\boldsymbol{u})] \subseteq [1,1] \diamondplus [0, \mathtt{Sup}(\boldsymbol{u})]$$

die rechte Seite schon für $\mathtt{Sup}(\boldsymbol{u}) < 2^{-3210}$ auch bei `stagprec=39` eine hinreichend gute Einschließung liefert.

Fast noch einfacher kann eine Einschließung von $\sqrt{1+x^2}$ für $\mathtt{Inf}(\boldsymbol{u}) \to +\infty$ angegeben werden. Wir gehen dazu aus von der einfach zu beweisenden Ungleichung

$$x \leq \sqrt{1+x^2} \leq 1+x, \quad \forall x \geq 0.$$

Wegen der Monotonie der oberen drei Terme in x gilt damit

$$\sqrt{1+x^2} \in [\mathtt{Inf}(\boldsymbol{u}), \mathtt{Sup}(\boldsymbol{u})+1] = \boldsymbol{w} := \boldsymbol{u} + [0,1], \quad \forall x \in \boldsymbol{u}.$$

Die Einschließung von $\sqrt{1+x^2} \in \boldsymbol{w}$ wird dabei um so enger, je größer die Intervalluntergrenze $\mathtt{Inf}(\boldsymbol{u}) \geq 0$ gewählt wird. Der Vorteil von $\boldsymbol{w}$ ist auch jetzt nicht nur seine denkbar einfache Struktur, sondern auch die Tatsache, dass in

$$\boldsymbol{w} := \boldsymbol{u} + [0,1] \subseteq \boldsymbol{u} \diamondplus [0,1]$$

die rechte Seite schon für $\mathtt{Inf}(\boldsymbol{u}) > 2^{+3210}$ auch bei `stagprec=39` eine hinreichend gute Einschließung liefert.

In den Extrembereichen $\mathtt{Sup}(\boldsymbol{u}) \to 0$ und $\mathtt{Inf}(\boldsymbol{u}) \to +\infty$ erhält man also mit $\mathbf{v}$ und $\mathbf{w}$ optimale Einschließungen für $\sqrt{1+x^2}$ ohne einen sonst möglichen integer-Overflow bei den Grundoperationen. Im restlichen Zwischenbereich liefert dann die intervallmäßige Auswertung von $\sqrt{1+x^2}$ die gewünschte Einschließung, ebenfalls ohne einen integer-Overflow.

[15] Zur besseren Lesbarkeit bezeichnen wir das Argumentintervall jetzt nicht mit $\mathtt{x}$ sondern mit $\boldsymbol{u}$.

Im **1. Beispiel** liefert das folgende Programm für $x = 1 \cdot 2^{-2147483647}$ eine garantierte Einschließung des Funktionswertes $y = \sqrt{1+x^2}$.

```
// Programm  lx_test26.cpp;
// Zum Test der sqrt(1+x^2)-Funktion;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;

    X = lx_interval(-2147483647,l_interval(1));
    Y = sqrt1px2(X);

    cout << SetDotPrecision(16*stagprec,16*stagprec+10)
         << Scientific;
    s << Y;
    cout << "sqrt1px2(X) = " << s << endl;
}
```

Das Programm liefert mit Y die folgende optimale Einschließung

$$\sqrt{1+x^2} \in \mathsf{Y} = 2^{-1021} \cdot \underbrace{2.247116418577\ldots41203028017152000\ldots000}_{631 \text{ korrekte Dez.-Ziffern}}{}^{494\ldots}_{000\ldots} \cdot 10^{307}$$

mit 631 korrekten Dezimalziffern. Mit dem Algebra-System *Mathematica* kann man z.B. leicht überprüfen, dass $2.247\ldots715200\ldots000 \cdot 10^{307}$ genau der Wert 2^{+1021} ist.

Im **2. Beispiel** liefert das gleiche Programm für $x = 1 \cdot 2^{+2147483646}$ mit der Anweisung

```
X = lx_interval(+2147483646,l_interval(1));
```

eine ebenfalls optimale Einschließung des Funktionswertes $y = \sqrt{1+x^2}$:

$$\sqrt{1+x^2} \in \mathsf{Y} = 2^{+2147482625} \cdot \underbrace{2.247116418577\ldots41203028017152000\ldots000}_{631 \text{ korrekte Dez.-Ziffern}}{}^{494\ldots}_{000\ldots} \cdot 10^{307}$$

mit ebenfalls 631 korrekten Dezimalziffern. Mit dem Algebra-System *Mathematica* kann wieder leicht überprüft werden, dass $2.247\ldots715200\ldots000 \cdot 10^{307}$ genau der Wert 2^{+1021} ist, wie es wegen des vorgegebenen Arguments $x = 2^{2147483646}$ sein muss.

5.2.22 Die Funktion $\arctan(x)$

Die C-XSC Funktion

```
lx_interval atan(const lx_interval &x);
```

liefert garantierte Einschließungen für das Intervall $\boldsymbol{y} = \arctan(\mathtt{x})$.

Es gelten die folgenden Reihenentwicklungen:

$$\begin{aligned} \arctan(x) &= x \cdot \left(1 - \frac{x^2}{3} + \frac{x^4}{5} - + \ldots\right) = x \cdot \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1} \cdot x^{2k} \\ &= x \cdot P_\infty(x^2), \quad |x| < 1; \end{aligned} \tag{46}$$

$$\arctan(x) = \frac{\pi}{2} - \frac{1}{x} + \frac{1}{3x^3} - \frac{1}{5x^5} + - \ldots, \quad x > 1. \tag{47}$$

Da die Reihe rechts in (47) eine alternierende Leibniz-Reihe ist, gilt die Ungleichung

$$\frac{\pi}{2} - \frac{1}{x} < \arctan(x) < \frac{\pi}{2}, \quad x > 1. \tag{48}$$

Es soll jetzt ein Intervall $\boldsymbol{u}$, mit $u_1 := \mathtt{Inf}(\boldsymbol{u}) > 0$ angegeben werden, so dass gilt

$$\arctan(x) \in \frac{1}{2} \cdot \mathtt{Pi\_lx\_interval()}, \quad \forall x \in \boldsymbol{u} = [u_1, u_2]. \tag{49}$$

`Pi_lx_interval()` ist dabei ein Maschinenintervall vom Typ `lx_interval`, das die Zahl π sehr eng einschließt:

$$\pi - \mathtt{Inf(Pi\_lx\_interval())} = +4.1060273917\ldots \cdot 10^{-631}.$$

Um (49) zu realisieren, muss nach (48) die folgende Ungleichung erfüllt sein:

$$\begin{gathered} \frac{1}{2} \cdot \mathtt{Inf(Pi\_lx\_interval())} < \frac{\pi}{2} - \frac{1}{u_1} \quad \Longleftrightarrow \\ u_1 > \frac{2}{\pi - \mathtt{Inf(Pi\_lx\_interval())}} = \alpha := 4.87088810\ldots \cdot 10^{+630}; \\ 4.87088810 \cdot 10^{+630} = 2^{2092} \cdot 8.5675613601\ldots. \end{gathered}$$

Für Intervalle $\boldsymbol{u} = [u_1, u_2]$, mit $u_1 > \alpha$ ist damit eine enge Einschließung für $\arctan(\boldsymbol{u})$ gefunden. Im Restbereich[16] $0 \leq u_1 \leq \alpha$ wird nach B. Rothmaier die folgende Argumentreduktion benutzt, [29].

$$\begin{aligned} t_0 := x \in \mathbb{R}, \qquad t_m &:= \frac{t_{m-1}}{1 + \sqrt{1 + t_{m-1}^2}}, \quad m = 1, 2, \ldots \\ \rightsquigarrow \quad \arctan(x) &= 2^m \cdot \arctan(t_m). \end{aligned} \tag{50}$$

Die obige Rekursionsformel basiert dabei auf der Identität

$$\arctan(x) = 2 \cdot \arctan\left(\frac{x}{1 + \sqrt{1 + x^2}}\right),$$

[16] Wegen der Punktsymmetrie kann man sich auf $u_1 \geq 0$ beschränken.

die lediglich m-mal sukzessiv anzuwenden ist. Neben dem Nachteil einer etwas aufwendigen Auswertung der Transformationsformel (50) für das reduzierte Argument t_m besitzt diese jedoch auch die folgenden Vorteile:

1. In (50) können keine Auslöschungseffekte auftreten.
2. Für $|t_0| = |x| \gg 1$ gilt schon nach nur einem Transformationsschritt: $|t_1| \approx 1$.
3. Für $|t_0| = |x| \leq 1$ wird das reduzierte Argument bei jedem Iterationsschritt etwa halbiert, so dass ein hinreichend kleines Approximationsintervall, z.B. $|t_m| \leq 10^{-4}$, mit nur wenigen Iterationsschritten erreicht werden kann.
4. Bei der Ergebnisanpassung ist lediglich mit der Zweierpotenz $2^m, m \geq 1$ zu multiplizieren, und wegen der Implementierung der Klasse `lx_interval` wird dies mit nur einer integer-Addition realisiert.

Nach 2. und 3. kann man für $|t_m|$ nach $m \geq 1$ Transformationen die folgende grobe Näherung angeben

$$|t_m| \sim \begin{cases} 2^{-m+1}, & \text{falls } |t_0| > 1, \\ |t_0| \cdot 2^{-m}, & \text{falls } |t_0| \leq 1, \end{cases}$$

mit der anschließend die notwendige Iterationszahl m für ein vorgegebenes Approximationsintervall, z.B. $|t_m| \leq 10^{-4}$, sehr einfach angegeben werden kann:

$$m = \begin{cases} \dfrac{\ln(2) + 4 \cdot \ln(10)}{\ln(2)} > 1, & \text{falls } |t_0| > 1, \\ \dfrac{ex \cdot \ln(2) + 4 \cdot \ln(10)}{\ln(2)}, & \text{falls } |t_0| \leq 1, \end{cases}$$

dabei ist ex der Zweierexponent des Startwertes $t_0 = x$, mit $|x| \sim 2^{ex}$.

Zur Approximation von $\arctan(x)$ wird nach (46) die Reihe

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

approximiert durch

$$P_N(x^2) := \sum_{k=0}^{N} \frac{(-1)^k}{2k+1} \cdot x^{2k} = 1 - \frac{(x^2)^1}{3} + \frac{(x^2)^2}{5} - \frac{(x^2)^3}{7} + - \ldots + (-1)^N \cdot \frac{(x^2)^N}{2N+1}.$$

Da $P_\infty(x^2)$ für $|x| < 1$ eine alternierende Leibniz-Reihe ist, gilt für den absoluten Fehler δ die Abschätzung:

$$|\delta| = \left| P_\infty(x^2) - P_N(x^2) \right| \leq \frac{x^{2 \cdot (N+1)}}{2N+3}.$$

Ist jetzt $\boldsymbol{u}$ ein vorgegebenes Argumentintervall, mit $|\boldsymbol{u}| = R$, und ist $\boldsymbol{u}$ ein Teilintervall des Approximationsintervalls, so gilt:

$$|\delta| = \left| P_\infty(x^2) - P_N(x^2) \right| \leq \frac{R^{2 \cdot (N+1)}}{2N+3}, \quad \forall x \in \boldsymbol{u}, \quad N = 0, 1, 2, \ldots \; . \tag{51}$$

Es muss jetzt noch zu einer vorgegebenen Präzision p von $\boldsymbol{u}$ der geeignete Polynomgrad N bestimmt werden. Wegen $P_N(x^2) \approx 1$ verlangen wir analog zu (41)

$$\frac{R^{2\cdot(N+1)}}{2N+3} \leq 2^{-53\cdot\mathsf{p}} \qquad \Longleftrightarrow$$
$$(2N+2)\cdot\ln(R) - \ln(2N+3) \leq -53\cdot\mathsf{p}\cdot\ln(2);$$

Mit $R \sim 2^{ex}$ ergibt sich dann aus der letzten Ungleichung die Forderung

$$(52) \qquad (2N+2)\cdot ex\cdot\ln(2) - \ln(2N+3) \leq -53\cdot\mathsf{p}\cdot\ln(2), \quad N = 0,1,2,\ldots$$

Man startet mit $N = 0$ und vergrößert N schrittweise um 1, bis obige Ungleichung erfüllt ist und erhält damit den geeigneten Polynomgrad N.

Die Iterationszahl m sollte wegen der aufwendigen Quadratwurzelauswertungen nicht zu groß sein, d.h. das Approximationsintervall sollte nicht unnötig klein gewählt werden. Im Programm wird für das reduzierte Argument t_m das Intervall $|t_m| \leq 10^{-4}$ verlangt, womit ein guter Kompromiss getroffen ist.

Die Maschinenauswertung von $P_N(x^2)$ nach dem Hornerschema und die Berechnung einer Schranke des Approximationsfehlers nach (51) muss wie im Abschnitt 5.2.15 intervallmäßig erfolgen, um garantierte Einschließungen von $\arctan(\boldsymbol{u})$ berechnen zu können.

Für ein Eingangsintervall $\boldsymbol{u}$, mit $|\boldsymbol{u}| = R$ und hinreichend kleinem $R \ll 1$ wird der Polynomgrad $N = 0$, und nach (51) muss dann für den absoluten Approximationsfehler der Ausdruck $R^2/3$ ausgewertet werden. Dabei kommt es bei der Berechnung von R^2 für $R \to 0$ zu einem internen integer-Überlauf, der jedoch recht einfach vermieden werden kann. Nach (46) gilt

$$\frac{\arctan(x)}{x} = 1 - \frac{x^2}{3} + \frac{x^4}{5} - \frac{x^6}{7}, \quad 0 < |x| < 1,$$

und da die Reihe oben rechts eine alternierende Leibniz-Reihe ist, folgt direkt

$$1 - \frac{x^2}{3} < \frac{\arctan(x)}{x} < 1, \quad \text{und damit:}$$

$$(53) \qquad x\cdot\left(1 - \frac{x^2}{3}\right) \leq \arctan(x) \leq x, \quad \text{falls } 0 \leq x < 1;$$

$$(54) \qquad x \leq \arctan(x) \leq x\cdot\left(1 - \frac{x^2}{3}\right), \quad \text{falls } -1 < x \leq 0.$$

Die Einschließungen (53) und (54) sind ideal für $x \to 0$ und kommen zur Anwendung, wenn $R \leq 2\cdot 2^{-1080} = 2^{-1079}$. Bei einer beliebigen Wahl des Intervalls $\boldsymbol{u} = [u_1, u_2]$, mit $|\boldsymbol{u}| = R \leq 2^{-1079}$ sind drei Fälle zu unterscheiden:

1. $0 \in \boldsymbol{u}$, d.h. $u_1 \leq 0$ und $u_2 \geq 0$. Mit (53) und (54) folgt dann direkt:

$$\arctan(x) \in [u_1, u_2] \;\; \forall x \in \boldsymbol{u}.$$

2. $u_1 > 0$. Nach (53) gilt für $0 < u_1 \leq u_2 < 1$ die Doppelungleichung

$$u_1 \cdot \left(1 - \frac{u_2^2}{3}\right) \leq \arctan(x) \leq u_2.$$

Um die Berechnung von $u_2^2/3$ wegen drohenden integer-Überlaufs zu vermeiden, muss weiter vereinfacht werden. Mit $a := 1-2^{-2097} \in$ `One_m_lx_interval()` verlangen wir:

$$\begin{aligned} & u_1 \cdot a \leq u_1 \cdot \left(1 - \frac{u_2^2}{3}\right) \\ \overset{u_1>0}{\Longleftrightarrow}\ & 1 - 2^{-2097} \leq 1 - \frac{u_2^2}{3} \iff \frac{u_2^2}{3} \leq 2^{-2097} \iff u_2^2 \leq 3 \cdot 2^{-2097}, \end{aligned}$$

und die letzte Ungleichung ist sicher erfüllt, wenn gilt:

$$u_2^2 \leq 2 \cdot 2^{-2097} = 2^{-2096} \iff u_2 < 2^{-1048}.$$

Zusammenfassung:

Im Fall $0 \leq u_1 \leq u_2 < 2^{-1048}$ gilt: $\arctan(x) \in [a \cdot u_1, u_2]\ \forall x \in \boldsymbol{u} = [u_1, u_2]$.

Zu beachten ist, dass $u_2 < 2^{-1048}$ automatisch erfüllt ist, wenn $R \leq 2^{-1079}$ verlangt wird.

3. $u_2 < 0$. Nach (54) gilt für $-1 < u_1 \leq u_2 < 0$ die Doppelungleichung

$$u_1 \leq \arctan(x) \leq u_2 \cdot \left(1 - \frac{u_1^2}{3}\right).$$

Um die Berechnung von $u_1^2/3$ wegen drohenden integer-Überlaufs zu vermeiden, muss weiter vereinfacht werden. Mit $a := 1-2^{-2097} \in$ `One_m_lx_interval()` verlangen wir:

$$\begin{aligned} & u_2 \cdot \left(1 - \frac{u_1^2}{3}\right) < u_2 \cdot a \\ \overset{u_2<0}{\Longleftrightarrow}\ & 1 - \frac{u_1^2}{3} > 1 - 2^{-2097} \iff \frac{u_1^2}{3} < 2^{-2097} \iff u_1^2 < 3 \cdot 2^{-2097}, \end{aligned}$$

und die letzte Ungleichung ist sicher erfüllt, wenn gilt:

$$u_1^2 < 2 \cdot 2^{-2097} = 2^{-2096} \iff |u_1| = -u_1 < 2^{-1048} \iff u_1 > -2^{-1048}.$$

Zusammenfassung:

Im Fall $-2^{-1048} < u_1 \leq u_2 \leq 0$ gilt: $\arctan(x) \in [u_1, a \cdot u_2]\ \forall x \in \boldsymbol{u} = [u_1, u_2]$.

Zu beachten ist, dass $-2^{-1048} < u_1$ automatisch erfüllt ist, wenn $R \leq 2^{-1079}$ verlangt wird.

Im **1. Beispiel** liefert das folgende Programm für $x = 2^{-2147482626}$ eine Einschließung des Funktionswertes $y = \arctan(x)$.

```
// Programm  lx_test27.cpp;
// Zum Test der atan-Funktion;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;

    X = lx_interval(-2147482626,l_interval(1));
    Y = atan(X);

    cout << SetDotPrecision(16*stagprec,16*stagprec+10)
         << Scientific;
    s << Y;
    cout << "atan(X) = " << s << endl;
}
```

Das Programm liefert mit Y die folgende optimale Einschließung

$$\arctan(x) \in \mathsf{Y} = 2^{-2147483648} \cdot [\underbrace{4.4942328371557\ldots303999\ldots999}_{630\ \text{korrekte Dez.-Ziffern}}80237 \cdot 10^{+307},$$
$$4.4942328371557\ldots30400000000\ldots000 \cdot 10^{+307}]$$

mit 630 korrekten Dezimalziffern. Mit dem etwas kleineren Argument $x = 2^{-2147482627}$ erhält man einen vorzeitigen Programmabbruch wegen eines Überlaufs bei den internen integer-Berechnungen. Zu beachten ist, dass das Algebra-System *Mathematica* schon mit $x = 2^{-1073741570}$ wegen eines internen Underflows einen entsprechenden Programmabbruch produziert.

Im **2. Beispiel** liefert das folgende Programm für $x = 1.3 \cdot 10^{646457298}$ eine Einschließung des Funktionswertes $y = \arctan(x)$.

```
// Programm  lx_test28.cpp;
// Zum Test der atan-Funktion;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
```

```
using namespace std;

int main()
{
    stagprec = 19;
    lx_interval X,Y;

    X = lx_interval(646457298,"[1.3,1.3]");
    Y = atan(X);

    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "atan(X) = " << Y << endl;
}
```

Das Programm liefert mit Y die folgende Einschließung

$$\arctan(1.3 \cdot 10^{646457298}) \in \mathsf{Y} = 10^{-1} \cdot \underbrace{1.570796326794\ldots8036301245706368}_{302 \text{ korrekte Dez.-Ziffern}} {}^{700\ldots}_{554\ldots} \cdot 10^{1}$$

mit 302 korrekten Dezimalziffern. Beachten Sie, dass x in der Klasse `lx_interval` nicht exakt darstellbar ist, so dass das Eingangsintervall $\mathsf{X} \ni x$ kein Punktintervall sein kann. Wählt man das nur um den Faktor 10 größere Argument $x = 1.3 \cdot 10^{646457299}$, so erhält man eine Fehlermeldung mit Programmabbruch. *Mathematica* liefert schon für das erste Argument x eine Overflow-Fehlermeldung.

Im **3. Beispiel** wird die folgende Gleichung benutzt

$$\arctan\left(\sqrt{2}-1\right) = \frac{\pi}{8}. \tag{55}$$

Zur Überprüfung wird die linke Seite durch ein Intervall $\boldsymbol{u} \ni \arctan\left(\sqrt{2}-1\right)$ eingeschlossen und $\pi/8$ durch ein Intervall $\boldsymbol{v} \ni \pi/8$. Dabei kann $\pi/8$ mit Hilfe der Intervall-Konstanten `Pi_lx_interval()` $\ni \pi$ vom Typ `lx_interval` sehr eng eingeschlossen werden. Nach der Berechnung von $\boldsymbol{u}$ und $\boldsymbol{v}$ sollte dann $\boldsymbol{v} \subset \boldsymbol{u}$ realisiert werden können. Das folgende Programm

```
// Programm  lx_test29.cpp;
// Zum Test der atan-Funktion;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 19;
```

```
    lx_interval X,u,v;

    X = Sqrt2_lx_interval()-1;
    u = atan(X);
    v = Pi_l_interval();
    times2pown(v,-3); // v: inclusion of pi/8;

    if (v<u) cout << "v ist in u enthalten." << endl;
    else cout << "v ist NICHT in u enthalten." << endl;
}
```

bestätigt die vermutete Einschließung $\boldsymbol{v} \subset \boldsymbol{u}$. Zu beachten ist, dass das Argument $\sqrt{2}-1$ nicht exakt darstellbar ist und somit das einschließende Intervall `X` kein Punktintervall sein kann. Aus diesem Grund muss der relative Durchmesser von $\boldsymbol{u}$ deutlich größer sein als der von $\boldsymbol{v}$.

Hinweis:

Zur Einschließung von $\pi/4$ kann man eine Einschließung von $\arctan(1)$ berechnen, die mit `stagprec` $= 39$ auf 475 korrekte Dezimalstellen geliefert wird. Laufzeitmäßig deutlich schneller und noch sehr viel enger kann $\pi/4$ jedoch mit Hilfe der Intervallkonstanten `Pi_l_interval()`$/4$ eingeschlossen werden, die eine maximale Genauigkeit von ca. 630 Dezimalstellen liefert.

5.2.23 Die Funktion $\sqrt{1-x^2}$

Die C-XSC Funktion

```
lx_interval sqrt1mx2(const lx_interval &x);
```

liefert garantierte Einschließungen für das Intervall $\boldsymbol{y} = \sqrt{1-\mathsf{x}^2},\ |\mathsf{x}| \leq 1$. Zur besseren Lesbarkeit bezeichnen wir das Eingangsargumentintervall nicht mit x sondern mit $\boldsymbol{u} = [u_1, u_2]$. Bei der Implementierung wird der Funktionsterm

$$y = \sqrt{1-x^2}, \quad x \in \mathbb{R}, \quad |x| \leq 1$$

intervallmäßig ausgewertet, so dass nach der Inklusionsmonotonie der Intervallrechnung die folgende Einschließung realisiert wird:

$$\sqrt{1-x^2} \in \texttt{sqrt(1}\diamondsuit\texttt{sqr(u))}, \quad \forall x \in \boldsymbol{u}. \tag{56}$$

Für Intervalle $\boldsymbol{u}$, mit $|\boldsymbol{u}| \ll 1$, also z.B. $|\boldsymbol{u}| = 2^{-2147483000}$ liefert die Funktion `sqr(u)` rechts in (56) wegen eines internen integer-Überlaufs einen vorzeitigen Programmabbruch, der jedoch sehr einfach verhindert werden kann. Ausgangspunkt ist die Doppelungleichung

$$1 - |x| \leq \sqrt{1-x^2} \leq 1, \quad \text{falls } |x| \leq 1, \tag{57}$$

wobei die obige Einschließung von $\sqrt{1-x^2}$ um so enger wird, je kleiner man $|x|$ wählt. Wir betrachten jetzt ein Eingangsintervall $\boldsymbol{u}$, mit $R := |\boldsymbol{u}| < 1$, dann folgt direkt mit (57)

$$\sqrt{1-x^2} \in [1-R, 1], \quad \forall x \in \boldsymbol{u}, \quad \text{falls: } 0 \leq |\boldsymbol{u}| = R < 1. \tag{58}$$

In der Klasse `lx_interval` ist eine sehr große darstellbare Zahl $\alpha < 1$ gegeben durch

$$\alpha := 2^{-1023} \cdot (2^{+1023} - 2^{-1074}) = 1 - 2^{-2097}$$

Für $2 \leq \texttt{stagprec} \leq 39$ wird mit den folgenden Anweisungen

```
lx_interval a;
l_interval li;
a = lx_interval(-1023,l_interval(comp(0.5,1024)));
li = li_part(a);
li += 0; // To get the actual precision;
li[StagPrec(li)] = -minreal;
a = lx_interval(-1023,li);
```

das Intervall $\mathsf{a} = [1 - 2^{-2097}, 1]$ vom Typ `lx_interval` realisiert. Das Eingangsintervall $\boldsymbol{u}$, mit $|\boldsymbol{u}| = R$, soll jetzt so gewählt werden, dass gilt:

$$1 - R \geq \alpha = 1 - 2^{-2097} \quad \Longleftrightarrow \quad R \leq 2^{-2097}.$$

Für alle $x \in \boldsymbol{u}$ gilt dann nach (58)

$$\sqrt{1-x^2} \in [1-R, 1] \subset [1 - 2^{-2097}, 1], \quad \text{falls } R \leq 2^{-2097}. \tag{59}$$

Wir brauchen jetzt noch ein einfaches Verfahren, um zu einem vorgegebenen Staggered Intervall $\boldsymbol{u}$ vom Typ `lx_interval` eine Oberschranke S für $R := |\boldsymbol{u}| \leq S$ zu bestimmen. Für ein Objekt $\boldsymbol{u}$ vom Typ `lx_interval` gilt:

$$\boldsymbol{u} = 2^{ex} \cdot \boldsymbol{z}, \quad \text{mit } ex = \texttt{expo}(\boldsymbol{u}), \ \boldsymbol{z} = \texttt{li\_part}(\boldsymbol{u});$$

$\boldsymbol{z}$ ist dabei ein Staggered Intervall vom Typ `l_interval`, und mit der Anweisung

$$exl = \texttt{expo\_gr}(\boldsymbol{z});$$

erhält man den Zweier-Exponenten der betragsmäßig größten Komponente z_i von $\boldsymbol{z}$. Es gilt dann $|z_i| < 2^{exl}$ und wenn $\boldsymbol{z}$ nach Voraussetzung im Staggered Format vorliegt, so gilt $|\boldsymbol{z}| \leq 2 \cdot 2^{exl} = 2^{exl+1}$. Der zusätzliche Faktor 2 berücksichtigt dabei die Beträge der restlichen Komponenten von $\boldsymbol{z}$. Man erhält schließlich für S den einfach auszuwertenden Ausdruck:

$$R := |\boldsymbol{u}| = 2^{ex} \cdot |\boldsymbol{z}| \leq 2^{ex+exl+1} =: S.$$

Verlangt man jetzt

$$R \leq S := 2^{ex+exl+1} \leq 2^{-2097} \quad \Longleftrightarrow \quad ex \leq -2098 - exl,$$

so gilt nach (59) für alle $x \in \boldsymbol{u}$ die Einschließung

$$\sqrt{1-x^2} \in [1 - 2^{-2097}, 1], \quad \text{falls } ex \leq -2098 - exl. \tag{60}$$

Beachten Sie bitte, dass der integer-Ausdruck $-2098-exl$ stets ohne Über- oder Unterlauf ausgewertet werden kann. Auch für sehr kleine Eingangsintervalle $\boldsymbol{u}$ in der unmittelbaren Umgebung des Ursprungs können jetzt die Funktionswerte $\sqrt{1-x^2}$ effektiv und sehr eng für alle $x \in \boldsymbol{u}$ eingeschlossen werden.

Im **1. Beispiel** liefert das folgende Programm für $x = 2^{-2147483647}$ eine Einschließung des Funktionswertes $y = \sqrt{1-x^2}$.

```
// Programm  lx_test30.cpp;
// Zum Test der sqrt(1-x^2)-Funktion;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;

    X = lx_interval(-2147483647,l_interval(1));
```

```
    Y = sqrt1mx2(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec+10)
         << Scientific;
    s << Y;
    cout << "sqrt1mx2(X) = " << s << endl;
}
```

Das Programm liefert mit Y die folgende optimale Einschließung

$$\sqrt{1-x^2} \in \mathsf{Y} = 2^{-1023} \cdot [\underbrace{8.9884656743\ldots8607999\ldots9995059}_{631 \text{ korrekte Dez.-Ziffern}} \cdot 10^{+307},$$
$$8.9884656743\ldots86080000000\ldots000 \cdot 10^{+307}]$$

mit 631 korrekten Dezimalziffern, dabei hat das Supremum des obigen Ausgabeintervalls Y genau den Wert 1.

Im **2. Beispiel** liefert das folgende Programm für $x = 0.57$ eine garantierte Einschließung des Funktionswertes $y = \sqrt{1-x^2}$.

```
// Programm  lx_test31.cpp;
// Zum Test der sqrt(1-x^2)-Funktion;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 19;
    lx_interval X,Y;

    X = lx_interval(0,"[0.57,0.57]");
    Y = sqrt1mx2(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "sqrt1mx2(X) = " << Y << endl;
}
```

Das Programm liefert mit Y die folgende garantierte Einschließung

$$\sqrt{1-0.57^2} \in \mathsf{Y} = 10^{-1} \cdot \underbrace{8.2164469206585884205\ldots3019161616334}_{302 \text{ korrekte Dez.-Ziffern}}{}^{570\ldots}_{129\ldots} \cdot 10^{0}$$

mit 302 korrekten Dezimalziffern. Es ist zu beachten, dass $x = 0.57$ im Binärsystem nicht exakt darstellbar ist, so dass X kein Punktintervall sein kann.

5.2.24 Die Funktion $\sqrt{x^2-1}$

Die C-XSC Funktion

```
lx_interval sqrtx2m1(const lx_interval &x);
```

liefert garantierte Einschließungen für das Intervall $\boldsymbol{y} = \sqrt{\mathtt{x}^2 - 1},\ |\mathtt{x}| \geq 1$. Zur besseren Lesbarkeit bezeichnen wir das Eingangsargumentintervall nicht mit `x` sondern mit $\boldsymbol{u} = [u_1, u_2]$. Bei der Implementierung wird der Funktionsterm

$$y = \sqrt{x^2 - 1}, \quad x \in \mathbb{R}, \quad |x| \geq 1$$

intervallmäßig ausgewertet, so dass nach der Inklusionsmonotonie der Intervallrechnung die folgende Einschließung realisiert wird:

$$\sqrt{x^2-1} \in \boldsymbol{y} := \mathtt{sqrt}((\boldsymbol{u} \diamondsuit 1) \diamondsuit (\boldsymbol{u} \diamondsuit 1)), \quad \forall x \in \boldsymbol{u}. \tag{61}$$

Für Intervalle $\boldsymbol{u}$, mit $|\boldsymbol{u}| \gg 1$, also z.B. $|\boldsymbol{u}| = 2^{+2147483000}$ liefert das Produkt rechts in (61) wegen eines internen integer-Überlaufs einen vorzeitigen Programmabbruch, der jedoch sehr einfach verhindert werden kann. Ausgangspunkt ist dazu die Doppelungleichung

$$|x| - 1 \leq \sqrt{x^2-1} \leq |x|, \quad \text{falls } |x| \geq 1, \tag{62}$$

wobei die obige Einschließung von $\sqrt{x^2-1}$ um so enger wird, je größer man $|x|$ wählt. Wegen $y(-x) \equiv y(+x)$ kann man sich auf $x \geq 1$ beschränken.

Wir betrachten jetzt ein Eingangsintervall $\boldsymbol{u} = [u_1, u_2]$, mit $u_1 \geq 1$, dann folgt direkt mit (62) die Einschließung

$$\sqrt{x^2-1} \in [u_1, u_2] \diamondsuit [-1, 0], \quad \forall x \in \boldsymbol{u} = [u_1, u_2],\ \ u_1 \geq 1, \tag{63}$$

und im Gegensatz zur `sqr(u)`-Funktion erfolgt jetzt in (63) die Addition des Intervalls $[-1, 0]$ auch für sehr große u_1-Werte ohne einen internen integer-Überlauf. Zu beachten ist außerdem, dass die Auswertung in (63) sehr effektiv ist, da weder eine Multiplikation noch die Quadratwurzel-Funktion benötigt wird. Aus diesem Grund sollte man in (63) u_1 möglichst klein wählen, ohne dabei jedoch unnötige Genauigkeitsverluste zu erkaufen. Testrechnungen haben ergeben, dass $u_1 \geq 2^{+1604}$ eine gute Wahl ist. Beachten Sie bitte, dass sowohl (61) als auch (63) für $u_1 \geq 1$ stets garantierte Einschließungen liefern, die aber teilweise mit stark unterschiedlicher Genauigkeit berechnet werden.

Mit Punktintervallen $\boldsymbol{u} = [u_1, u_1]$ und mit einer Präzision `stagprec` $= 39$ liefert (61) etwa 477 korrekte Dezimalziffern, während man mit (63) Einschließungen mit bis zu 615 korrekten Dezimalziffern erhält.

Schließt man $\sqrt{x^2-1}$ für $x \in \boldsymbol{u} = [u_1, u_2]$ und $u_1 < 2^{1604}$ nach (61) ein, so erhält man für $u_1 \to +1$ starke Überschätzungen. Mit $\boldsymbol{u} = [u_1, u_1]$ und der Maschinenzahl[17] $u_1 = 1 + 2^{-2097}$ liefert die Einschließung nach (61) die Unterschranke $\mathtt{Inf}(\boldsymbol{y}) = 0$,

[17] $u_1 = 1 + 2^{-2097}$ ist die kleinste Maschinenzahl, die größer 1 ist. u_1 wird mit dem Punktintervall `One_p_lx_interval()` eingeschlossen, vgl. Seite 18 und (6) auf Seite 19.

während $\sqrt{u_1^2 - 1} = 3.315\ldots \cdot 10^{-316}$ natürlich positiv ist. Um diese Überschätzung zu vermeiden, geht man im Fall $\texttt{Inf}(\boldsymbol{y}) <= 0 \wedge x > +1$ aus von $x \geq 1 + 2^{-2097}$ und erhält

$$\sqrt{x^2 - 1} = \sqrt{(x-1)(x+1)} \geq \sqrt{2^{-2097} \cdot (x+1)}, \quad \text{falls } x > 1.$$

Für ein gegebenes Maschinenintervall $\boldsymbol{u} = [u_1, u_2]$, mit $u_1 > 1$, gilt dann wegen $x \geq u_1$ die Ungleichung $x + 1 \geq u_1 + 1$, und damit

$$\sqrt{x^2 - 1} \geq \alpha := \sqrt{2^{-2097} \cdot (u_1 + 1)} \quad \text{für alle } x \in \boldsymbol{u} = [u_1, u_2], \quad \text{mit } u_1 > 1.$$

Wertet man daher α mit dem Punktintervall $\boldsymbol{u}_1 \ni u_1$ intervallmäßig aus, so erhält man mit $\texttt{Inf}(\boldsymbol{\alpha})$ eine positive, garantierte Unterschranke für $\sqrt{x^2 - 1}$ für alle $x \in \boldsymbol{u} = [u_1, u_2]$, mit $u_1 > 1$. Diese verbesserte Unterschranke $\texttt{Inf}(\boldsymbol{\alpha})$ kommt jedoch nur zur Anwendung, wenn bei der intervallmäßigen Auswertung von $\boldsymbol{y} = \sqrt{(\boldsymbol{u} - 1) \cdot (\boldsymbol{u} + 1)}$ im Fall $u_1 > 1$ $\texttt{Inf}(\boldsymbol{y})$ verschwindet.
Im **1. Beispiel** liefert das folgende Programm für $x = 2^{+2147483646}$ eine Einschließung des Funktionswertes $y = \sqrt{x^2 - 1}$.

```
// Programm  lx_test32.cpp;
// Zum Test der sqrt(x^2-1)-Funktion;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;

    X = lx_interval(2147483646,l_interval(1));
    Y = sqrtx2m1(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec+10)
        << Scientific;
    s << Y;
    cout << "sqrtx2m1(X) = " << s << endl;
}
```

Das Programm liefert mit Y die folgende garantierte und optimale Einschließung

$$\sqrt{x^2 - 1} \in \texttt{Y} = 2^{+2147482625} \cdot [\underbrace{2.24711641857789\ldots17151999\ldots9997774}_{615 \text{ korrekte Dez.-Ziffern}} \cdot 10^{+307},$$
$$2.24711641857789\ldots171520000000\ldots000 \cdot 10^{+307}]$$

mit 631 korrekten Dezimalziffern, dabei hat das Supremum von Y genau den optimalen Wert $2^{+2147483646}$. Die Berechnung von $\sqrt{x^2 - 1}$ z.B. mit *Mathematica* scheitert an einem internen Overflow.

Im **2. Beispiel** liefert das folgende Programm für $x = 1 + 2^{-2097}$ eine Einschließung des Funktionswertes $y = \sqrt{x^2 - 1} = 3.315618423383237992361\ldots \cdot 10^{-316}$, wobei diese Näherung mit *Mathematica* berechnet wurde.

```
// Programm  lx_test32a.cpp;
// Zum Test der sqrt(x^2-1)-Funktion;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;

    X = One_p_lx_interval(); // X = 1+2^(-2097)
    Y = sqrtx2m1(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec+10)
         << Scientific;
    cout << "sqrtx2m1(X) = " << Y << endl;
}
```

Das Programm liefert mit `Y` die folgende, immer noch etwas grobe Einschließung

$$\sqrt{x^2 - 1} \in \mathtt{Y} = 10^{-316} \cdot [3.31561842338_{219}\ldots, 6.6312368467693145746011\ldots],$$

wobei die Unterschranke mit 12 korrekten Dezimalziffern berechnet wurde, während die Oberschranke um den Faktor 2 zu groß ausfällt. Diese grobe Einschließung ist begründet durch das sehr dicht bei 1 liegende Argument $x = 1 + 2^{-2097}$.

Eine sehr viel bessere Einschließung liefert das Programm `lx_test31.cpp` mit den neuen Anweisungen

```
X = lx_interval(0,"[1.0001,1.0001]");   Y = sqrtx2m1(X);
```

Dabei ist zu beachten, dass $x = 1.0001$ im Binärsystem nicht exakt darstellbar ist, so dass das einschließende Intervall $\mathtt{X} \ni 1.0001$ kein Punktintervall sein kann. Außerdem ist im Vergleich zum 2. Beispiel $x = 1.0001$ jetzt sehr viel größer als 1, so dass Auslöschungseffekte kaum eine Rolle spielen.

5.2.25 Die Funktion $\arcsin(x)$

Für alle reellen $x \in [-1, +1]$ ist der Graph des Hauptzweiges der $\arcsin$-Funktion dargestellt in Abb. 1.

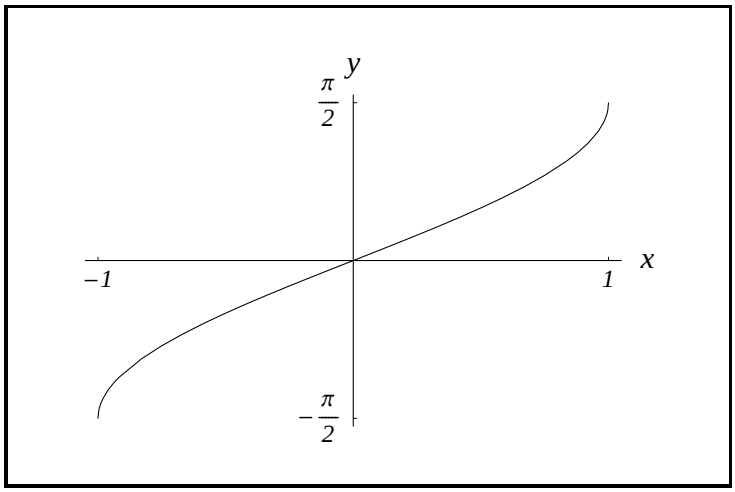


Abbildung 1: $y = \arcsin(x)$

Die C-XSC Funktion

```
lx_interval asin(const lx_interval &u);
```

liefert garantierte Einschließungen für das Intervall $\boldsymbol{y} = \arcsin(\boldsymbol{u})$, $\boldsymbol{u} = [u_1, u_2]$, $|\boldsymbol{u}| \leq 1$. Die Implementierung der `asin`-Funktion erfolgt mit Hilfe der bereits realisierten $\arctan$-Funktion:

$$\arcsin(x) = \arctan\left(\frac{x}{\sqrt{1-x^2}}\right), \quad |x| \leq 0.75; \tag{64}$$

$$\arcsin(x) = \frac{\pi}{2} - \arctan\left(\frac{\sqrt{1-x^2}}{x}\right), \quad 0.75 < x \leq 1; \tag{65}$$

$$\arcsin(x) = -\frac{\pi}{2} - \arctan\left(\frac{\sqrt{1-x^2}}{x}\right), \quad -1 \leq |x| < -0.75; \tag{66}$$

Um für alle $x \in \boldsymbol{u}$ Einschließungen für $\arcsin(x)$ zu erhalten, werden die rechten Seiten von (64), (65), (66) intervallmäßig ausgewertet, indem dort x durch $\boldsymbol{u}$ ersetzt wird. Wenn der Fall $0 \in \boldsymbol{u}$ erledigt ist, kann man sich wegen der Punktsymmetrie der $\arcsin$-Funktion beschränken auf $u_1 > 0$, so dass dann nur noch (64) und (65) zur Anwendung kommen. Der Vorteil von (65) besteht darin, dass man für $x \to 1$ im Vergleich zu (64) genauere Einschließungen erhält. Dabei ist jedoch zu beachten, dass wegen der vertikalen Tangente die Einschließungen für $x \to 1$ mit erheblichen Überschätzungen behaftet sind. So erhält man mit der Präzision `stagprec=39` für $x = 0.875$ eine Einschließung von $\arcsin(x)$ mit stolzen 476 korrekten Dezimalziffern, während für $x = 1 - 2^{-2097}$ die Einschließung mit nur noch 315 korrekten Dezimalziffern berechnet wird.

Die intervallmäßige Auswertung von $\sqrt{1-x^2}$ erfolgt mit Hilfe der schon implementierten C-XSC Funktion `sqrt1mx2(...)`, vgl. Seite 89.

Im **1. Beispiel** liefert das folgende C-XSC Programm für das betragsmäßig sehr kleine Argument $x = 2^{-2147482626}$ eine Einschließung des Funktionswertes $y = \arcsin(x)$.

```
// Programm  lx_test33.cpp;
// Zum Test der asin(x)-Funktion;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;

    X = lx_interval(-2147482626,l_interval(1));
    Y = asin(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec+10)
         << Scientific;
    s << Y;
    cout << "asin(X) = " << s << endl;
}
```

Das Programm liefert mit Y die folgende Einschließung

$$\arcsin(x) \in \mathsf{Y} = 2^{-2147483648} \cdot [4.4942328371557\ldots303999\ldots99980237 \cdot 10^{+307}, \underbrace{4.4942328371557\ldots30400000\ldots9936}_{477 \text{ korrekte Dez.-Ziffern}}\ldots \cdot 10^{+307}]$$

mit 477 korrekten Dezimalziffern. Mit dem etwas kleineren Argument $x = 2^{-2147482627}$ erhält man einen vorzeitigen Programmabbruch wegen eines Überlaufs bei den internen integer-Berechnungen.

Das Algebra-System *Mathematica* liefert jedoch schon mit $x = 2^{-1073741820}$ wegen eines internen Underflows einen entsprechenden Programmabbruch.

Im **2. Beispiel** liefert das folgende Programm für das im Binärsystem nicht darstellbare Argument $x = 0.901$ eine garantierte Einschließung des Funktionswertes $y = \arcsin(x)$.

```
// Programm  lx_test34.cpp;
// Zum Test der asin(x)-Funktion;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;
```

```
int main()
{
    stagprec = 19;
    lx_interval X,Y;
    string s;

    X = lx_interval(0,"[0.901,0.901]");
    Y = asin(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec+10)
         << Scientific;
    s << Y;
    cout << "asin(X) = " << s << endl;
    cout << "asin(X) = " << Y << endl;
}
```

Bei der ersten Ausgabe wird `Y` vom Typ `lx_interval` zunächst als `(n,Y.li)` in eine Zeichenkette `s` geschrieben, wobei `n` der Zweier-Exponent von `Y` ist. Danach wird diese Zeichenkette in den Ausgabekanal geleitet und auf dem Bildschirm ausgegeben:

$$\arcsin(0.901) \in \mathsf{Y} = 2^{-1021} \cdot \underbrace{2.521419973326051\ldots9365970841841322}_{303 \text{ korrekte Dez.-Ziffern}} \tfrac{46}{37} \cdot 10^{+307}.$$

Die zweite Ausgabe erfolgt in lesbarer, dezimaler Form und liefert

$$\arcsin(0.901) \in \mathsf{Y} = 10^{-1} \cdot \underbrace{1.12206913379314446\ldots30426875316715532}_{302 \text{ korrekte Dez.-Ziffern}} \tfrac{571}{408} \cdot 10^{1}.$$

Übungsaufgaben:

1. Überprüfen Sie die Beziehung

$$\arcsin\left(\frac{1+\sqrt{3}}{2\sqrt{2}}\right) = \frac{5\pi}{12},$$

   indem Sie bei maximaler Präzision mit `stagprec` $= 39$ zeigen, dass die Einschließung von $5\pi/12$ in der Einschließung des $\arcsin$-Funktionswertes enthalten ist. Benutzen Sie für $\pi, \sqrt{2}, \sqrt{3}$ die entsprechenden Intervallkonstanten aus Tabelle 2 auf Seite 18.

2. Überprüfen Sie die für $0 < |x| \leq 1$ geltende Identität

$$\frac{\arcsin\left(\dfrac{2x}{x^2+1}\right)}{2 \cdot \arctan(x)} \equiv 1,$$

   indem Sie zeigen, dass z.B. für die darstellbaren Werte $x = -0.25, +0.5, +0.75, 1$ die sehr engen Einschließungen der linken Seite stets auch die Zahl 1 enthalten.

5.2.26 Die Funktion $\arccos(x)$

Für alle reellen $x \in [-1, +1]$ ist der Graph des Hauptzweiges der $\arccos$-Funktion dargestellt in Abb. 2.

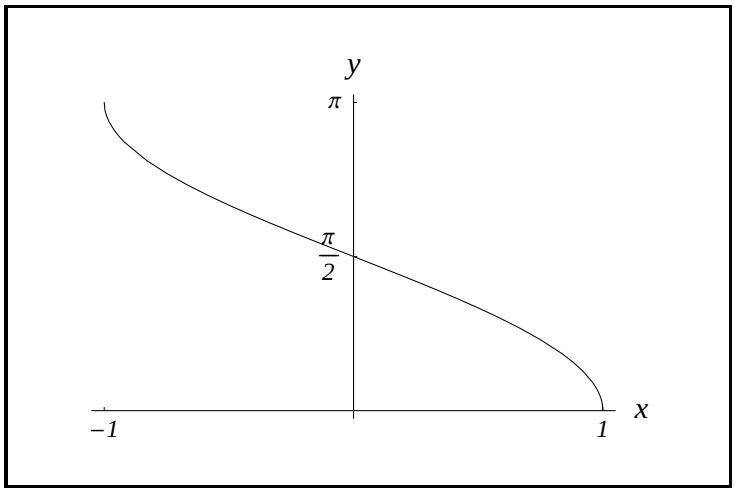


Abbildung 2: $y = \arccos(x)$

Die C-XSC Funktion

```
lx_interval acos(const lx_interval &u);
```

liefert garantierte Einschließungen für das Intervall $\boldsymbol{y} = \arccos(\boldsymbol{u})$, $\boldsymbol{u} = [u_1, u_2]$, $|\boldsymbol{u}| \leq 1$. Die Implementierung der `acos`-Funktion erfolgt mit Hilfe der bereits realisierten $\arctan$-Funktion:

$$\arccos(x) = \frac{\pi}{2} - \arctan\left(\frac{x}{\sqrt{1-x^2}}\right), \quad |x| < 0.25, \tag{67}$$

$$\arccos(x) = \arctan\left(\frac{\sqrt{1-x^2}}{x}\right), \quad 0.25 \leq x \leq 1, \tag{68}$$

$$\arccos(x) = \pi + \arctan\left(\frac{\sqrt{1-x^2}}{x}\right), \quad -1 \leq x \leq -0.25. \tag{69}$$

Um für alle $x \in \boldsymbol{u}$ Einschließungen für $\arccos(x)$ zu erhalten, werden die rechten Seiten von (67), (68), (69) intervallmäßig ausgewertet, indem dort x durch $\boldsymbol{u}$ ersetzt wird. Wenn der Fall $0 \in \boldsymbol{u}$ erledigt ist, kann man sich wegen der Punktsymmetrie zu $(0, \pi/2)$ bei der $\arccos$-Funktion beschränken auf $u_1 > 0$, so dass dann nur noch (67) und (68) zur Anwendung kommen. Der Vorteil von (68) besteht darin, dass man für $|x| \to 1$ im Vergleich zu (67) genauere Einschließungen erhält.

Es ist verblüffend, mit welch hoher Genauigkeit die Funktionswerte $\arccos(x)$ für z.B. $x \to -1$ trotz der vertikalen Tangente eingeschlossen werden können. So erhält man mit `stagprec=39` für $x = -0.875$ eine Einschließung von $\arccos(x)$ mit 476 korrekten Dezimalziffern, und für $x = -1 + 2^{-2095}$ wird die Einschließung sogar mit 629 korrekten Dezimalziffern berechnet. Erst mit $x = -1 + 2^{-2097}$ sinkt die Genauigkeit auf 315 Dezimalstellen, da dann $\sqrt{1-x^2}$ nur noch mit starken Überschätzungen eingeschlossen werden kann.

Im **1. Beispiel** liefert das folgende Programm für $x = -1 + 2^{-2095}$ eine Einschließung des Funktionswertes $y = \arccos(x)$.

```
// Programm  lx_test35.cpp;
// Zum Test der acos(x)-Funktion;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;

    X = -1 + lx_interval(-2095,l_interval(1));
    Y = acos(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec+8)
         << Scientific;
    s << Y;
    cout << "acos(X) = " << s << endl;
}
```

Das Programm liefert mit Y die Einschließung

$$\arccos(-1+2^{-2095}) \in \mathsf{Y} = 2^{-1020} \cdot \underbrace{3.529762216182\ldots778492245567}_{629\ \text{korrekte Dez.-Ziffern}}{}^{\,3261}_{1679} \cdot 10^{+307}$$

mit 629 korrekten Dezimalstellen.

Zeigen Sie, dass mit dem etwas kleineren Argument $x = -1 + 2^{-2097}$ die Einschließung von $\arccos(x)$ nur noch mit 315 korrekten Dezimalstellen berechnet werden kann. Der Grund für die geringere Genauigkeit ist die unvermeidbare, größere Überschätzung bei der Einschließung von

$$\sqrt{1-x^2} \equiv \sqrt{(1-x)\cdot(1+x)}.$$

Übungsaufgabe:
Überprüfen Sie die Identität

$$\frac{\arccos(x)}{2\cdot\arctan\left(\sqrt{\dfrac{1-x}{1+x}}\right)} \equiv 1, \quad -1 < x < +1,$$

indem Sie zeigen, dass z.B. für die darstellbaren Werte $x = -0.75, -0.25, +0.5, +0.75$ die sehr engen Einschließungen der linken Seite stets auch die Zahl 1 enthalten.

5.2.27 Die Funktion arccot(x)

Betrachtet man die Funktion $f(x) = \cot(x)$ im Bereich $0 < x < \pi$, so erhält man für die stetige Umkehrfunktion $y = \text{arccot}\,(x)$, mit $-\infty < x < +\infty$, den Graph in Abb. 3.

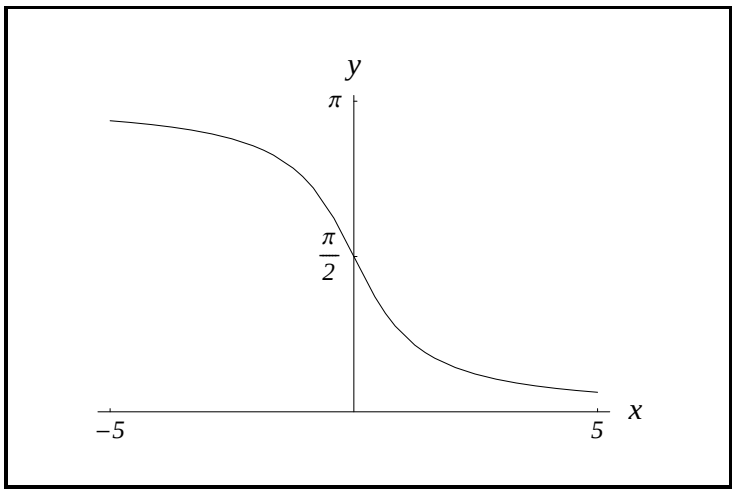


Abbildung 3: $y = \text{arccot}\,(x)$

Beachten Sie bitte, dass der Hauptzweig der arccot-Funktion, z.B. im Algebrasystem *Mathematica*, oft auch anders definiert wird, wobei der Ast für $x < 0$ um π nach unten verschoben wird.[18] Der Nachteil dieser Definition ist die Unstetigkeit bei $x = 0$, die bei der obigen Definition für die C-XSC Funktion mit dem Wertebereich $0 < y < \pi$ nicht auftritt. Die C-XSC Funktion

```
lx_interval acot(const lx_interval &u);
```

liefert garantierte Einschließungen für das Intervall $\boldsymbol{y} = \text{arccot}\,(\boldsymbol{u})$, $\boldsymbol{u} = [u_1, u_2]$, mit $-\infty < u_1 \leq u_2 < +\infty$. Die Implementierung der `acot`-Funktion erfolgt mit der bereits realisierten $\arctan$-Funktion:

$$\text{arccot}\,(x) = \frac{\pi}{2} - \arctan(x), \quad 0 \leq |x| < 1, \tag{70}$$

$$\text{arccot}\,(x) = \arctan\left(\frac{1}{x}\right), \quad x \geq 1, \tag{71}$$

$$\text{arccot}\,(x) = \pi + \arctan\left(\frac{1}{x}\right), \quad x \leq -1. \tag{72}$$

Um für alle $x \in \boldsymbol{u}$ Einschließungen für $\text{arccot}\,(x)$ zu erhalten, werden die rechten Seiten von (70), (71), (72) intervallmäßig ausgewertet, indem dort x durch $\boldsymbol{u}$ ersetzt wird. Wenn der Fall $0 \in \boldsymbol{u}$ erledigt ist, kann man sich wegen der Punktsymmetrie zu $(0, \pi/2)$ bei der arccot -Funktion beschränken auf $u_1 > 0$, so dass dann nur noch (70) und (71) zur Anwendung kommen. Der wesentliche Vorteil von (71) besteht nun darin, dass man für $x \to +\infty$ im Vergleich zu (70) viel genauere Einschließungen erhält.

[18] $f(x) = \cot(x)$ wird dabei betrachtet in den Teilbereichen $-\pi/2 < x < 0$ und $0 < x < +\pi/2$.

Der Ausdruck rechts in (70) liefert genaue Einschließungen für $|x| \to 0$, besitzt jedoch einen Schönheitsfehler, da die $\arctan$-Funktion nach Seite 86 nur bis $x_0 = 2^{-2147482626}$ ohne Fehlermeldung ausgewertet werden kann. Damit liefert auch (70) für $0 < x < x_0$ eine entsprechende Fehlermeldung, die jedoch durch die folgenden Überlegungen leicht vermieden werden kann. Nach (53) auf Seite 84 gilt:

$$x \cdot \left(1 - \frac{x^2}{3}\right) \leq \arctan(x) \leq x, \quad \text{falls } \; 0 \leq x < 1.$$

Wegen $\operatorname{arccot}(x) = \pi/2 - \arctan(x)$ folgt daraus direkt

$$\frac{\pi}{2} - x \leq \operatorname{arccot}(x) \leq \frac{\pi}{2} - x \cdot \left(1 - \frac{x^2}{3}\right), \quad \text{falls } \; 0 \leq x < 1,$$

und mit einer nur etwas größeren Oberschranke erhält man schließlich:

$$\frac{\pi}{2} - x \leq \operatorname{arccot}(x) \leq \frac{\pi}{2}, \quad \text{falls } \; 0 \leq x < 1. \tag{73}$$

Für ein vorgegebenes Argumentintervall $\boldsymbol{u} = [u_1, u_2]$, mit $0 < u_1 \leq u2 < 1$, gilt dann die Einschließung

$$\operatorname{arccot}(x) \in \left[\frac{\pi}{2} - u_2, \frac{\pi}{2}\right] \quad \forall x \in \boldsymbol{u} = [u_1, u_2], \;\; 0 < u_1 \leq u2 < 1. \tag{74}$$

Bei maximaler Präzision $\texttt{stagprec} = 39$ wird $\pi/2$ mit Hilfe der Intervallkonstanten $\texttt{Pi\_lx\_interval()}/2$ mit ca. 630 korrekten Dezimalstellen in hoher Genauigkeit eingeschlossen.

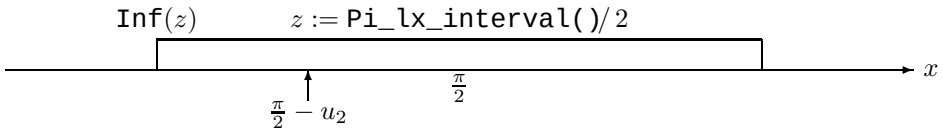


Abbildung 4: Einschließung von $\pi/2$

Dabei gilt: $\frac{\pi}{2} - \texttt{Inf}(z) = 2.0530136\ldots \cdot 10^{-631} = 2^{-1507.769\ldots} < 2^{-1507}$. Nach Abb. 4 verlangen wir

$$\frac{\pi}{2} - u_2 \geq \texttt{Inf}(z) \quad \Longleftrightarrow \quad u_2 \leq \frac{\pi}{2} - \texttt{Inf}(z) < 2^{-1507}.$$

Wählt man daher $u_2 < 2^{-1507}$, so gilt die folgende Einschließung:

$$\operatorname{arccot}(x) \in \texttt{Pi\_lx\_interval()}/2 \quad \forall x \in \boldsymbol{u} = [u_1, u_2], \;\; u_2 < 2^{-1507}. \tag{75}$$

Mit $ex = \texttt{expo}(\boldsymbol{u})$ und $exl = \texttt{expo\_gr}(\texttt{li\_part}(\boldsymbol{u}))$ gilt nach den Überlegungen von Seite 90 die Abschätzung $u_2 < 2^{ex+exl+1}$. Verlangt man daher $2^{ex+exl+1} < 2^{-1507}$ bzw. $ex < -exl - 1508$, so gilt die Einschließung

$$\operatorname{arccot}(x) \in \texttt{Pi\_lx\_interval()}/2 \quad \forall x \in \boldsymbol{u} = [u_1, u_2], \;\; ex < -exl - 1508. \tag{76}$$

Damit können jetzt auch für Argumente x vom Typ `lx_interval`, mit $0 < x < x_0$, garantierte Einschließungen für $\operatorname{arccot}(x)$ sehr effektiv mit nur einer zusätzlichen integer-Abfrage berechnet werden.

Im **1. Beispiel** liefert das folgende Programm für $x = 2^{-2147483647}$ eine Einschließung des Funktionswertes $y = \text{arccot}(x)$.

```
// Programm  lx_test36.cpp;
// Zum Test der acot(x)-Funktion;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;

    X = lx_interval(-2147483647,l_interval(1));
    Y = acot(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec+9)
         << Scientific;
    s << Y;
    cout << "acot(X) = " << s << endl;
    cout << "acot(X) = " << Y << endl;
}
```

Bei der ersten Ausgabe wird `Y` vom Typ `lx_interval` zunächst als `(n,Y.li)` in eine Zeichenkette `s` geschrieben, wobei `n` der Zweier-Exponent von `Y` ist. Danach wird diese Zeichenkette in den Ausgabekanal geleitet und auf dem Bildschirm ausgegeben:

$$\text{arccot}(2^{-2147483647}) \in \mathsf{Y} = 2^{-1021} \cdot \underbrace{3.5297622161826\ldots7849224556730}_{631 \text{ korrekte Dez.-Ziffern}}{}^{628}_{133} \cdot 10^{+307}.$$

Die zweite Ausgabe erfolgt in lesbarer, dezimaler Form und liefert

$$\text{arccot}(2^{-2147483647}) \in \mathsf{Y} = 10^{-1} \cdot \underbrace{1.570796326794896\ldots36301245706368}_{302 \text{ korrekte Dez.-Ziffern}}{}^{698\ldots}_{555\ldots} \cdot 10^{1}.$$

Bei dieser letzten dezimalen Ausgabe erhält man wegen der internen Umrechnungen mit der Logarithmus-Funktion nur noch 302 korrekte Dezimalziffern, die jedoch über die interne Genauigkeit von 631 Dezimalziffern keine Auskunft liefert!

Im **2. Beispiel** liefert das folgende Programm für $x = 2^{2147482625}$ eine Einschließung des Funktionswertes $y = \text{arccot}(x)$.

```
// Programm  lx_test37.cpp;
// Zum Test der acot(x)-Funktion;
```

```
#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;

    X = lx_interval(2147482625,l_interval(1));
    Y = acot(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec+9)
         << Scientific;
    s << Y;
    cout << "acot(X) = " << s << endl;
}
```

Das Programm liefert mit Y die folgende optimale Einschließung

$$\operatorname{arccot}(x) \in \mathsf{Y} = 2^{-2147483647} \cdot [\underbrace{4.4942328371557\ldots303999\ldots9998}_{630 \text{ korrekte Dez.-Ziffern}}023 \cdot 10^{+307},$$
$$4.4942328371557\ldots3040000000\ldots000 \cdot 10^{+307}]$$

mit 630 korrekten Dezimalziffern. Mit dem etwas größeren Argument $x = 2^{2147482626}$ erhält man einen vorzeitigen Programmabbruch wegen eines Überlaufs bei den internen integer-Berechnungen. Zu beachten ist, dass das Algebra-System *Mathematica* schon mit dem sehr viel kleineren Argument $x = 2^{+1073741437}$ wegen eines internen Underflows einen entsprechenden Programmabbruch produziert.

Im **3. Beispiel** liefert das folgende Programm für das im Binärsystem nicht darstellbare Argument $x = 0.7501$ eine Einschließung des Funktionswertes $y = \operatorname{arccot}(x)$.

```
// Programm  lx_test38.cpp;
// Zum Test der acot(x)-Funktion;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 19;
    lx_interval X,Y;
```

```
    string s;

    X = lx_interval(0,"[0.7501,0.7501]");
    Y = acot(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec+2)
         << Scientific;
    s << Y;
    cout << "acot(X) = " << s << endl;
    cout << "acot(X) = " << Y << endl;
}
```

Bei der ersten Ausgabe wird `Y` vom Typ `lx_interval` zunächst als `(n,Y.li)` in eine Zeichenkette `s` geschrieben, wobei `n` der Zweier-Exponent von `Y` ist. Danach wird diese Zeichenkette in den Ausgabekanal geleitet und auf dem Bildschirm ausgegeben:

$$\operatorname{arccot}(0.7501) \in \mathtt{Y} = 2^{-1021} \cdot \underbrace{2.0835965006924088\ldots168107571747687}_{304 \text{ korrekte Dez.-Ziffern}}{}^{819}_{599} \cdot 10^{+307}.$$

Die zweite Ausgabe erfolgt in lesbarer, dezimaler Form und liefert

$$\operatorname{arccot}(0.7501) \in \mathtt{Y} = 10^{-1} \cdot \underbrace{9.272312210735521\ldots2271105068065357}_{301 \text{ korrekte Dez.-Ziffern}}{}^{772\ldots}_{698\ldots} \cdot 10^{0}.$$

Bei dieser letzten dezimalen Ausgabe erhält man wegen der internen Umrechnungen mit der Logarithmus-Funktion nur noch 301 korrekte Dezimalziffern, die jedoch über die interne Genauigkeit von 304 Dezimalziffern keine Auskunft liefert!

Übungsaufgabe:

Überprüfen Sie die speziellen Funktionswerte

$$\operatorname{arccot}(2+\sqrt{3}) = \frac{\pi}{12}; \quad \operatorname{arccot}\left(\sqrt{5+2\cdot\sqrt{5}}\right) = \frac{\pi}{10}; \quad \operatorname{arccot}(\sqrt{3}) = \frac{\pi}{6};$$

$$\operatorname{arccot}\left(\frac{1}{\sqrt{3}}\right) = \frac{\pi}{3}; \quad \operatorname{arccot}\left(\sqrt{1+\frac{2}{\sqrt{5}}}\right) = \frac{\pi}{5}; \quad \operatorname{arccot}(2-\sqrt{3}) = \frac{5\pi}{12};$$

indem Sie bei maximaler Präzision `stagprec` $= 39$ zeigen, dass die Einschließungen der rechten Seiten in den Einschließungen der jeweils linken Funktionswerte enthalten sind. Benutzen Sie für $\pi, \sqrt{3}, \sqrt{5}$ die entsprechenden Intervallkonstanten aus Tabelle 2 auf Seite 18.

5.2.28 Die Funktion $\sinh(x)$

Für $x \in [-4, +4]$ ist der Graph der hyperbolischen $\sin$-Funktion dargestellt in Abb. 5.

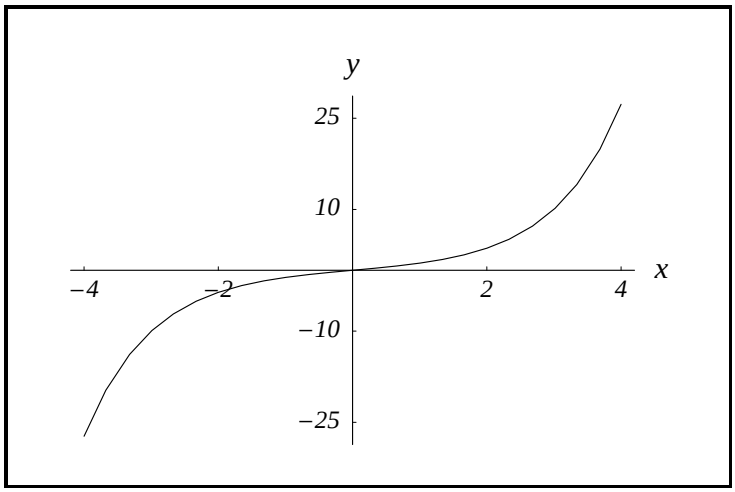


Abbildung 5: $y = \sinh(x)$

Die Funktion $f(x) = \sinh(x)$ ist für alle $x \in \mathbb{R}$ definiert durch

$$\sinh(x) = \frac{e^x - e^{-x}}{2}, \quad x \in \mathbb{R}. \tag{77}$$

Die C-XSC Funktion

```
lx_interval sinh(const lx_interval &u);
```

liefert garantierte Einschließungen für das reelle Intervall $\boldsymbol{y} = \sinh(\boldsymbol{u})$, $\boldsymbol{u} = [u_1, u_2]$, mit $-\infty < u_1 \leq u_2 < +\infty$. Bei der Implementierung ist dabei zu beachten, dass nach (77) Auslöschung für $x \to 0$ vermieden wird. Dies kann z.B. mit Hilfe der Taylorreihe realisiert werden, wobei dann ein ganz ähnlicher Algorithmus wie bei der $\sin$-Funktion von Seite 65 zur Anwendung kommt. Um die Auslöschung für $x \to 0$ zu vermeiden, kann man anstelle der Taylor-Reihe mit $t(x) := e^{-x} - 1$ auch die folgende sehr einfach auszuwertende Identität benutzen

$$\sinh(x) \equiv -\frac{t}{2} \cdot \left(1 + \frac{1}{1+t}\right), \quad |x| < 1, \quad t(x) := e^x - 1 > -1, \tag{78}$$

die insbesondere für $x \to 0$ zur optimalen Einschließung von $\sinh(x)$ geeignet ist. Der Vorteil von (78) besteht darin, dass $t(x)$ durch die bereits vorhandene C-XSC Funktion `expm1`$(-x)$ eingeschlossen wird, deren Laufzeit im Vergleich zur $\sin$-Funktion mit der sehr ähnlichen Taylorreihe sogar etwas geringer ist.

Für $|x| \geq 1$ wird die rechte Seite von (77) benutzt. Im Fall $|x| \to \infty$ tritt dabei jedoch das Problem auf, das die Auswertung des Quotienten nur bis $|x| \leq 744261295.0$ gelingt, während die exp-Funktion selbst in dem sehr viel größeren Bereich $|x| \leq 1488521882$ ohne internen integer-Überlauf berechnet werden kann. Um diesen Bereich auch für die

sinh-Funktion zu realisieren, gehen wir aus von der für alle $x \geq 0$ gültigen Doppelungleichung[19]

$$\frac{e^x}{2} - \frac{1}{2} \leq \sinh(x) < \frac{e^x}{2}, \quad x \geq 0, \tag{79}$$

wobei die obige Einschließung von $\sinh(x)$ für $x \to \infty$ beliebig eng wird. Betrachtet man jetzt ein Eingangsintervall $\boldsymbol{u} = [u_1, u_2]$ vom Typ `lx_interval`, mit $0 < u_1 \leq u_2 < \infty$, so gilt

$$\frac{e^{u_1}}{2} - \frac{1}{2} < \sinh(x) < \frac{e^{u_2}}{2} \quad \forall x \in \boldsymbol{u} = [u_1, u_2], \;\; u_1 > 0, \quad \text{und damit:}$$

$$\sinh(x) \in \texttt{exp}(\boldsymbol{u}) \diamond 2 \oplus [-0.5, 1] \quad \forall x \in \boldsymbol{u} = [u_1, u_2], \;\; u_1 > 0. \tag{80}$$

Die notwendige Intervall-Addition rechts in (80) kann jetzt bis $u_1 = u_2 \leq 1488521882$ ohne vorzeitigen integer-Überlauf ausgeführt werden. Die zusätzliche Division durch 2 wird dabei mit `times2pown(exp(u),-1)` problemlos und sehr effektiv realisiert. Bei maximaler Präzision mit `stagprec` $= 39$ erhält man schon für $u_1 \geq 4096.0$ mit (80) optimale Einschließungen für $\sinh(x)$ $\forall x \in \boldsymbol{u}$.

Im **1. Beispiel** liefert das folgende Programm `lx_test39` für das maximale Argument $x = 1488521882$ eine garantierte Einschließung des Funktionswertes $y = \sinh(x)$.

```
// Programm  lx_test39.cpp;
// Zum Test der sinh(x)-Funktion;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;

    X = lx_interval(0,l_interval(1488521882));
    Y = sinh(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec+2)
         << Scientific;
    s << Y;
    cout << "sinh(X) = " << s << endl;
    cout << "sinh(X) = " << Y << endl;
}
```

[19] Wegen der Punktsymmetrie $\sinh(-x) \equiv \sinh(-x)$ kann man sich auf $x \geq 0$ beschränken.

Bei der ersten Ausgabe wird Y vom Typ `lx_interval` zunächst als `(n,Y.li)` in eine Zeichenkette `s` geschrieben, wobei `n` der Zweier-Exponent von Y ist. Danach wird diese Zeichenkette in den Ausgabekanal geleitet und auf dem Bildschirm ausgegeben:

$$\sinh(1488521882) \in \mathsf{Y} = 2^{2147482115} \cdot \underbrace{2.998376672102\ldots446182629586}_{457 \text{ korrekte Dez.-Ziffern}} {}^{538\ldots}_{303\ldots} \cdot 10^{+307}.$$

Die zweite Ausgabe erfolgt in lesbarer, dezimaler Form und liefert

$$\sinh(1488521882) \in \mathsf{Y} = 10^{646456838} \cdot \underbrace{1.753126912659\ldots5674601387776}_{302 \text{ korrekte Dez.-Ziffern}} {}^{927\ldots}_{806\ldots} \cdot 10^{1}.$$

Bei dieser letzten dezimalen Ausgabe erhält man wegen der internen Umrechnungen mit der Logarithmus-Funktion nur noch 302 korrekte Dezimalziffern, die jedoch über die interne Genauigkeit von 457 Dezimalziffern keine Auskunft liefert!

Im **2. Beispiel** liefert das folgende Programm für $x = 2^{-2147482625}$ eine Einschließung des Funktionswertes $y = \sinh(x)$.

```
// Programm  lx_test40.cpp;
// Zum Test der sinh(x)-Funktion;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;

    X = lx_interval(-2147482625,l_interval(1));
    Y = sinh(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec+2)
         << Scientific;
    s << Y;
    cout << "sinh(X) = " << s << endl;
    cout << "sinh(X) = " << Y << endl;
}
```

Wie beim 1. Beispiel wird bei der ersten Ausgabe Y mit dem Faktor $2^n = 2^{-2147483645}$ in maximaler Genauigkeit mit 477 korrekten Dezimalstellen angegeben.

$$\sinh(2^{-2147482625}) \in \mathsf{Y} = 2^{-2147483645} \cdot [\underbrace{1.123558209\ldots575999\ldots999}_{477 \text{ korrekte Dez.-Ziffern}}834\ldots \cdot 10^{+307},$$
$$1.123558209\ldots576000\ldots000331\ldots \cdot 10^{+307}]$$

Die zweite Ausgabe erfolgt wieder in lesbarer, dezimaler Form und liefert

$$\sinh(2^{-2147482625}) \in \mathsf{Y} = 10^{-646456686} \cdot \underbrace{5.1024063801\ldots8709337887001}_{301 \text{ korrekte Dez.-Ziffern}} {}^{414\ldots}_{286\ldots} \cdot 10^{0}.$$

Bei der letzten dezimalen Ausgabe werden wegen der internen Umrechnungen nur 301 korrekte Dezimalziffern ausgegeben, die jedoch über die tatsächliche interne Genauigkeit von 477 Dezimalziffern keine Auskunft liefern!

Übungsaufgabe:

Überprüfen Sie die Identität

$$\sinh(3x) \equiv \sinh(x) \cdot \left\{3 + 4 \cdot \sinh^2(x)\right\}, \quad x \in \mathbb{R}.$$

Zeigen Sie bei maximaler Präzision mit `stagprec` $= 39$, dass für im Staggered-Format darstellbare Argumente x die Einschließungen der linken Seiten in den Einschließungen der jeweiligen rechten Seite enthalten sind.

5.2.29 Die Funktion $\cosh(x)$

Für $x \in [-4, +4]$ ist der Graph der hyperbolischen cos-Funktion dargestellt in Abb. 6.

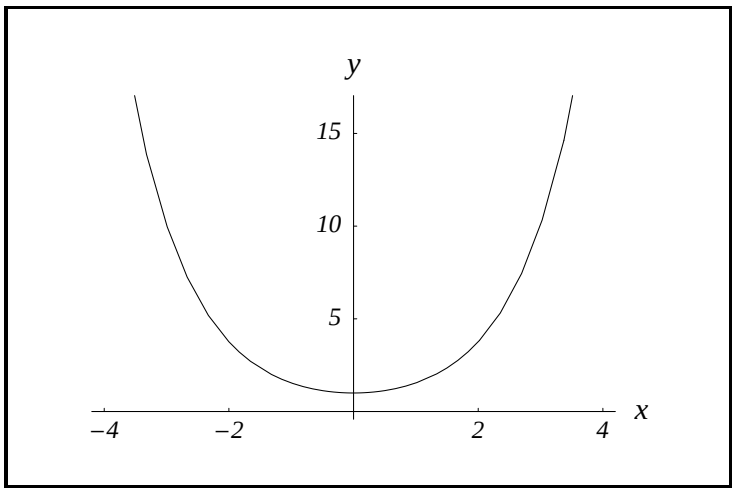


Abbildung 6: $y = \cosh(x)$

Die Funktion $f(x) = \cosh(x)$ ist für alle $x \in \mathbb{R}$ definiert durch

$$\cosh(x) = \frac{e^x + e^{-x}}{2}, \quad x \in \mathbb{R}. \tag{81}$$

Die C-XSC Funktion

```
lx_interval cosh(const lx_interval &u);
```

liefert garantierte Einschließungen für das reelle Intervall $\boldsymbol{y} = \cosh(\boldsymbol{u})$, $\boldsymbol{u} = [u_1, u_2]$, mit $-\infty < u_1 \leq u_2 < +\infty$. Da im Gegensatz zur $\sinh(x)$-Funktion jetzt für $x \to 0$ keine Auslöschung auftritt, wird bei der Implementierung die rechte Seite der Definitionsgleichung (81) benutzt, wobei man sich wegen der Achsensymmetrie auf $x \geq 0$ beschränken kann. Wie bei der $\sinh(x)$-Funktion muss man sich auch jetzt wieder Gedanken machen, wie man den Quotienten rechts in (81) auch noch im Bereich

$$744261295 < x \leq 1488521882$$

ohne internen integer-Überlauf auswerten kann[20]. Wir betrachten dazu die sehr einfach zu beweisende Doppelungleichung

$$\frac{e^x}{2} < \cosh(x) < \frac{e^x}{2} + \frac{1}{2}, \quad x > 0,$$

wobei die obige Einschließung von $\cosh(x)$ für $x \to \infty$ beliebig eng wird. Betrachtet man jetzt ein Eingangsintervall $\boldsymbol{u} = [u_1, u_2]$ vom Typ `lx_interval`, mit $0 < u_1 \leq u_2 < \infty$, so gilt

$$\frac{e^{u_1}}{2} < \cosh(x) < \frac{e^{u_2}}{2} + \frac{1}{2} \quad \forall x \in \boldsymbol{u} = [u_1, u_2], \;\; u_1 > 0, \quad \text{und damit:}$$

$$\cosh(x) \in \mathsf{exp}(\boldsymbol{u}) \diamond 2 \oplus [0, 0.5] \quad \forall x \in \boldsymbol{u} = [u_1, u_2], \;\; u_1 > 0. \tag{82}$$

[20] Die Exponentialfunktion selbst kann bis $x = 1488521882$ ohne integer-Überlauf berechnet werden.

Die notwendige Intervall-Addition rechts in (82) kann jetzt bis $u_1 = u_2 \leq 1488521882$ ohne vorzeitigen integer-Überlauf ausgeführt werden. Die zusätzliche Division durch 2 wird dabei mit `times2pown(exp(u),-1)` problemlos und sehr effektiv realisiert. Bei maximaler Präzision mit `stagprec` $= 39$ erhält man schon für $u_1 \geq 4096.0$ mit (82) optimale Einschließungen aller Funktionswerte $\cosh(x),\ \forall x \in \boldsymbol{u}$.

Im **1. Beispiel** liefert das folgende Programm `lx_test41` für das maximale Argument $x = 1488521882$ eine garantierte Einschließung des Funktionswertes $y = \cosh(x)$.

```
// Programm  lx_test41.cpp;
// Zum Test der cosh(x)-Funktion;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;

    X = lx_interval(0,l_interval(1488521882));
    Y = cosh(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec+2)
         << Scientific;
    s << Y;
    cout << "sinh(X) = " << s << endl;
    cout << "sinh(X) = " << Y << endl;
}
```

Bei der ersten Ausgabe wird `Y` vom Typ `lx_interval` zunächst als `(n,Y.li)` in eine Zeichenkette `s` geschrieben, wobei `n` der Zweier-Exponent von `Y` ist. Danach wird diese Zeichenkette in den Ausgabekanal geleitet und auf dem Bildschirm ausgegeben:

$$\cosh(1488521882) \in \mathsf{Y} = 2^{2147482115} \cdot \underbrace{2.998376672102\ldots446182629586}_{457\text{ korrekte Dez.-Ziffern}}{}^{538\ldots}_{303\ldots} \cdot 10^{+307}.$$

Die zweite Ausgabe erfolgt in lesbarer, dezimaler Form und liefert

$$\cosh(1488521882) \in \mathsf{Y} = 10^{646456838} \cdot \underbrace{1.753126912659\ldots5674601387776}_{302\text{ korrekte Dez.-Ziffern}}{}^{927\ldots}_{806\ldots} \cdot 10^{1}.$$

Bei dieser letzten dezimalen Ausgabe erhält man wegen der internen Umrechnungen mit der Logarithmus-Funktion nur noch 302 korrekte Dezimalziffern, die jedoch über die interne Genauigkeit von 457 Dezimalziffern keine Auskunft liefert!

5.2.30 Die Funktion $\tanh(x)$

Für $x \in [-4, +4]$ ist der Graph der hyperbolischen $\tan$-Funktion dargestellt in Abb. 7.

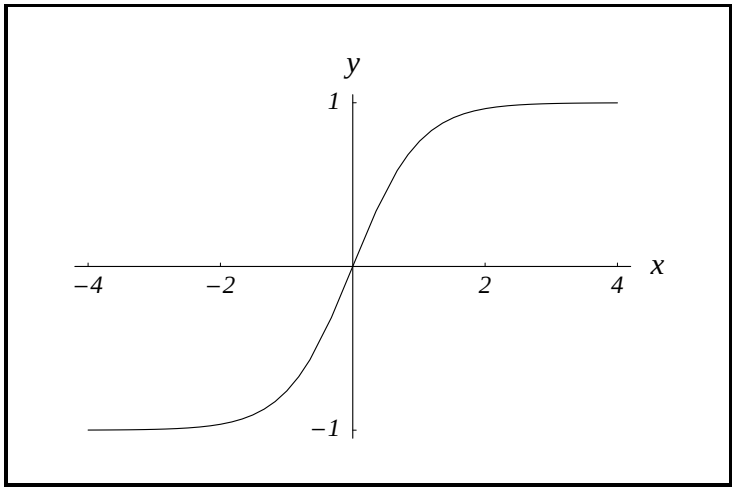


Abbildung 7: $y = \tanh(x)$

Die Funktion $f(x) = \tanh(x)$ ist für alle $x \in \mathbb{R}$ definiert durch

$$\begin{aligned} \tanh(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}}, \quad x \in \mathbb{R}, \\ &= \frac{1 - e^{-2x}}{1 + e^{-2x}}, \quad x \in \mathbb{R}, \qquad (83) \\ &= \frac{-t(x)}{2 + t(x)}, \quad t(x) := e^{-2x} - 1, \quad x \in \mathbb{R}. \qquad (84) \end{aligned}$$

Dabei ist (83) geeignet für $x \to +\infty$, und (84) ist geeignet für $x \to 0$, wobei $t(x)$ mit der schon vorhandenen Funktion `expm1(...)` sehr eng und auch effektiv eingeschlossen werden kann. Die C-XSC Funktion

```
lx_interval tanh(const lx_interval &u);
```

liefert garantierte Einschließungen für das reelle Intervall $\boldsymbol{y} = \tanh(\boldsymbol{u})$, $\boldsymbol{u} = [u_1, u_2]$, mit $-\infty < u_1 \leq u_2 < +\infty$. Bei der Implementierung kommt im Fall $0 \in \boldsymbol{u}$ zunächst (84) zur Anwendung. Danach sind zwei Fälle zu unterscheiden[21]:

1. Für alle $x \in \boldsymbol{u} = [u_1, u_2]$, mit $0 < u_1 \leq u_2 < 1$ gilt:

   $$\tanh(x) = \frac{-t(x)}{2 + t(x)}, \quad t(x) := e^{-2x} - 1; \qquad (85)$$

2. Für alle $x \in \boldsymbol{u} = [u_1, u_2]$, mit $0 < u_1 \leq u_2$ und $u_2 > 1$ gilt:

   $$\tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}. \qquad (86)$$

[21] Wegen der Punktsymmetrie zum Ursprung kann man sich jetzt auf $u_1 > 0$ beschränken.

Nach 1. und 2. sind auch sehr breite Intervalle $\boldsymbol{u}$ möglich, die jedoch in der Praxis einer Staggered-Arithmetik keine Rolle spielen. Um Einschließungen für $\tanh(x)$ zu erhalten, sind die rechten Seiten von (85) und (86) intervallmäßig auszuwerten, wobei x jeweils durch das Eingangsintervall $\boldsymbol{u} = [u_1, u_2]$ zu ersetzen ist.

Der Quotient rechts in (86) kann wegen $\exp(-2 \cdot x)$ ohne integer-Überlauf nur bis $2u_2 \leq +1488521882$ ausgewertet werden, obwohl man $\tanh(\boldsymbol{u})$ für hinreichend große Werte von u_1 durch das Intervall $\alpha := [1 - 2^{-2097}, 1]$ einschließen könnte[22]. Um dies zu erreichen, muss wegen der einfach zu beweisenden Doppelungleichung

$$1 - e^{-x} \leq \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} < 1, \quad x \in \mathbb{R}$$

folgende Bedingung erfüllt sein:

$$1 - e^{-u_1} > 1 - 2^{-2097} \iff e^{-u_1} < 2^{-2097} \iff -u_1 < -2097 \cdot \ln(2)$$
$$\iff u_1 > 2097 \cdot \ln(2) = 1453.5296\ldots ,$$

d.h. die Bedingung ist erfüllt für $u_1 > 1454$. Damit erhält man die Einschließung:

$$\tanh(x) \in [1 - 2^{-2097}, 1] \quad \forall x \in \boldsymbol{u} = [u_1, u_2], \text{ falls } u_1 > 1454.$$

Da die Überprüfung der Bedingung $u_1 > 1454$ für sehr große $u_1 \gg 1$ zu einem integer-Überlauf führen kann, wird jetzt gezeigt, wie dieser Überlauf vermieden werden kann. Mit dem staggered Wert u_1 vom Typ `lx_real` bestimmt man zunächst die beiden Zweier-Exponenten `ex` und `exa` mit den Anweisungen

```
ex = expo(u1);    exa = expo_gr( lr_part(u1) );
```

Bezeichnet man `lr_part(u1)` mit a, dann ist `exa` der Zweier-Exponent der betragsmäßig größten Komponente $a[j]$, d.h. es gilt wegen $u_1 > 0$

$$a[j] = m \cdot 2^{exa}, \quad \rightsquigarrow \quad a[j] \geq 0.5 \cdot 2^{exa} = 2^{exa-1};$$

Da einige Komponenten $a[i]$ auch negativ sein können, gilt: $a \geq 0.5 \cdot 2^{exa-1} = 2^{exa-2}$, und damit $u_1 \geq 2^{ex} \cdot 2^{exa-2} = 2^{ex+exa-2}$. Die Bedingung $u_1 > 1454$ ist also erfüllt, wenn

$$2^{ex+exa-2} > 1454 = 2^{10.50581\ldots}, \text{ d.h. wenn } \quad ex + exa \geq 13.$$

Die Überprüfung $ex \geq -exa + 13$ funktioniert jetzt auch noch mit dem maximalen Wert $u_1 = 2^{+2147483647} \cdot$ `MaxReal` $= 2^{+2147483647} \cdot 1.79769\ldots \cdot 10^{+308}$ und ist außerdem sehr viel schneller als die obige Abfrage $u_1 > 1454$, bei der vergleichsweise sehr aufwendige `idotprecision`-Ausdrücke auszuwerten sind.

[22] $\alpha := [1 - 2^{-2097}, 1]$ kann mit der Punktintervall-Konstanten `One_m_lx_interval()` sehr einfach realisiert werden.

Im **1. Beispiel** liefert das nachfolgende Programm für das in der Klasse `lx_interval` maximale Argument $x = 2^{+2147483647} \cdot \texttt{MaxReal} = 2^{+2147483647} \cdot 1.79769\ldots \cdot 10^{+308}$ eine Einschließung des Funktionswertes $y = \tanh(x)$.

```
// Programm  lx_test42.cpp;
// Zum Test der tanh(x)-Funktion;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;

    X = lx_interval(2147483647,l_interval(MaxReal,MaxReal));
    Y = tanh(X);

    cout << SetDotPrecision(16*stagprec,16*stagprec+10)
         << Scientific;

    s << Y;
    cout << "tanh(X) = " << s << endl;
    cout << "tanh(X) = " << Y << endl;
}
```

Das obige Programm `lx_test42.cpp` liefert bei der 1. Ausgabe mit `Y` die folgende optimale Einschließung

$$\tanh(x) \in \mathsf{Y} = 2^{-1022} \cdot [\underbrace{4.4942328371557\ldots303999\ldots9995}_{\text{631 korrekte Dez.-Ziffern}}059 \cdot 10^{+307}, \\ 4.4942328371557\ldots3040000000\ldots000 \cdot 10^{+307}]$$

mit 631 korrekten Dezimalziffern. Die zweite Ausgabe erfolgt in lesbarer, dezimaler Form und liefert

$$\tanh(x) \in \mathsf{Y} = 10^{-1} \cdot [\underbrace{9.999999\ldots9999999}_{\text{301 korrekte Dez.-Ziffern}}873\ldots, 1.000000\ldots00000115\ldots \cdot 10^{1}]$$

mit nur 301 korrekten Dezimalziffern. Bei dieser letzten dezimalen Ausgabe erhält man wegen der internen Umrechnungen mit der Logarithmus-Funktion nur noch 301 korrekte Dezimalziffern, die jedoch über die interne Genauigkeit von 631 Dezimalziffern keine Auskunft liefert!

Im **2. Beispiel** liefert das folgende Programm für $x = 2^{-2147482626}$ eine Einschließung des Funktionswertes $y = \tanh(x)$.

```
// Programm  lx_test43.cpp;
// Zum Test der tanh(x)-Funktion;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;

    X = lx_interval(-2147482626,l_interval(1));
    Y = tanh(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec+10)
         << Scientific;
    s << Y;
    cout << "tanh(X) = " << s << endl;
}
```

Die folgende Ausgabe von Y über die Zeichenkette s liefert mit dem Faktor $2^{-2147483137}$ die Einschließung von $\tanh(x)$ mit der intern berechneten Genauigkeit von 477 korrekten Dezimalstellen:

$$\tanh(x) \in \mathsf{Y} = 2^{-2147483137} \cdot [\underbrace{6.70390396497129\ldots047999\ldots9995059\ldots}_{477\ \text{korrekte Dez.-Ziffern}} \cdot 10^{+153},$$
$$6.70390396497129\ldots048000\ldots0001482\ldots \cdot 10^{+153}]$$

Wegen der Division durch $(2 + t(x))$ in (85) erhalten wir jetzt eine Genauigkeit von *nur* 477 Dezimalstellen, während z.B. bei der $\arctan(x)$-Funktion mit $x = 2^{-2147482626}$ eine Genauigkeit von 630 Dezimalstellen erreicht wird, vgl. Seite 86.

Übungsaufgabe:
Überprüfen Sie die folgenden Identitäten

$$\frac{2}{e^{-2x}+1} - \tanh(x) \equiv 1; \quad \frac{\sinh^2(x)}{1+\sinh^2(x)} \equiv \tanh^2(x);$$
$$\tanh(x) \cdot \cosh(x) \equiv \sinh(x);$$

indem Sie für darstellbare Punktargumente, wie z.B. $x = -0.5, +0.75, +2.0, \ldots$, zeigen, dass die Einschließungen der rechten Seiten in den Einschließungen der entsprechenden linken Seiten enthalten sind.

5.2.31 Die Funktion $\coth(x)$

Für $x \in [-2, +2]$ ist der Graph der hyperbolischen $\cot$-Funktion dargestellt in Abb. 8.

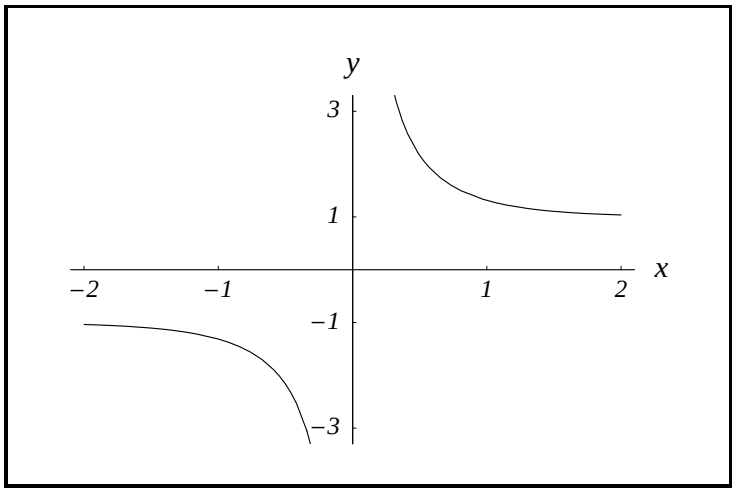


Abbildung 8: $y = \coth(x)$

Die Funktion $f(x) = \coth(x)$ ist für alle $x \in \mathbb{R}\backslash\{0\}$ definiert durch

$$\coth(x) = \frac{e^x + e^{-x}}{e^x - e^{-x}}, \quad x \in \mathbb{R}\backslash\{0\},$$

$$= \frac{1 + e^{-2x}}{1 - e^{-2x}}, \quad x \in \mathbb{R}\backslash\{0\}, \tag{87}$$

$$= \frac{2 + t(x)}{-t(x)}, \quad t(x) := e^{-2x} - 1, \quad x \in \mathbb{R}\backslash\{0\}. \tag{88}$$

Dabei ist (87) geeignet für $x \to +\infty$, und (88) ist geeignet für $x \to 0$, wobei $t(x)$ mit der schon vorhandenen Funktion `expm1(...)` sehr eng und auch effektiv eingeschlossen werden kann. Die C-XSC Funktion

```
lx_interval coth(const lx_interval &u);
```

liefert garantierte Einschließungen für das Intervall $\boldsymbol{y} = \coth(\boldsymbol{u})$, wobei $\boldsymbol{u} = [u_1, u_2]$ die Null nicht enthalten darf, und wegen der Punktsymmetrie zum Ursprung kann man sich auf $u_1 > 0$ beschränken. Bei der Implementierung sind für $u_1 > 0$ zwei Fälle zu unterscheiden:

1. Für alle $x \in \boldsymbol{u} = [u_1, u_2]$, mit $0 < u_1 \leq u_2 < 1$ gilt:

$$\coth(x) = \frac{2 + t(x)}{-t(x)}, \quad t(x) := e^{-2x} - 1; \tag{89}$$

2. Für alle $x \in \boldsymbol{u} = [u_1, u_2]$, mit $0 < u_1 \leq u_2$ und $u_2 > 1$ gilt:

$$\coth(x) = \frac{1 + e^{-2x}}{1 - e^{-2x}}. \tag{90}$$

Nach 1. und 2. sind auch sehr breite Intervalle $\boldsymbol{u}$ möglich, die jedoch in der Praxis einer Staggered-Arithmetik keine Rolle spielen. Um Einschließungen für $\coth(x)$ zu erhalten, sind die rechten Seiten von (89) und (90) intervallmäßig auszuwerten, wobei x jeweils durch das Eingangsintervall $\boldsymbol{u} = [u_1, u_2]$ zu ersetzen ist.

Der Quotient rechts in (90) kann wegen $\exp(-2 \cdot x)$ ohne integer-Überlauf nur bis $2u_2 \leq +1488521882$ ausgewertet werden, obwohl man $\coth(\boldsymbol{u})$ für hinreichend große Werte von u_1 durch das Intervall $\alpha := [1, 1 + 2^{-2097}]$ einschließen könnte[23]. Um dies zu erreichen, muss wegen der Doppelungleichung

$$1 < \coth(x) = \frac{e^x + e^{-x}}{e^x - e^{-x}} < 1 + e^{-x}, \quad x > 1 \tag{91}$$

und wegen der Monotonie der $\coth$-Funktion folgende Bedingung erfüllt sein:

$$1 + e^{-u_1} < 1 + 2^{-2097} \iff e^{-u_1} < 2^{-2097} \iff -u_1 < -2097 \cdot \ln(2)$$
$$\iff u_1 > 2097 \cdot \ln(2) = 1453.5296\ldots ,$$

d.h. die Bedingung ist erfüllt für $u_1 > 1454$. Damit erhält man die Einschließung:

$$\coth(x) \in [1, 1 + 2^{-2097}, 1] \quad \forall x \in \boldsymbol{u} = [u_1, u_2], \text{ falls } u_1 > 1454.$$

Da die Überprüfung der Bedingung $u_1 > 1454$ für sehr große $u_1 \gg 1$ zu einem interger-Überlauf führen kann, wird jetzt gezeigt, wie dieser Überlauf vermieden werden kann. Mit dem staggered Wert u_1 vom Typ `lx_real` bestimmt man zunächst die beiden Zweier-Exponenten `ex` und `exa` mit den Anweisungen

```
ex = expo(u1);    exa = expo_gr( lr_part(u1) );
```

Bezeichnet man `lr_part(u1)` mit a, dann ist `exa` der Zweier-Exponent der betragsmäßig größten Komponente $a[j]$, d.h. es gilt wegen $u_1 > 0$

$$a[j] = m \cdot 2^{exa}, \quad \rightsquigarrow \quad a[j] \geq 0.5 \cdot 2^{exa} = 2^{exa-1};$$

Da einige Komponenten $a[i]$ auch negativ sein können, gilt: $a \geq 0.5 \cdot 2^{exa-1} = 2^{exa-2}$, und damit $u_1 \geq 2^{ex} \cdot 2^{exa-2} = 2^{ex+exa-2}$. Die Bedingung $u_1 > 1454$ ist also erfüllt, wenn

$$2^{ex+exa-2} > 1454 = 2^{10.50581\ldots}, \text{ d.h. wenn } \quad ex + exa \geq 13.$$

Die Überprüfung $ex \geq -exa + 13$ funktioniert jetzt auch noch mit dem maximalen Wert $u_1 = 2^{+2147483647} \cdot$ `MaxReal` $= 2^{+2147483647} \cdot 1.79769\ldots \cdot 10^{+308}$ und ist außerdem sehr viel schneller als die obige Abfrage $u_1 > 1454$, bei der vergleichsweise sehr aufwendige `idotprecision`-Ausdrücke auszuwerten sind. Der aufmerksame Leser wird bemerkt haben, dass die Herleitung der Bedingung $ex \geq -exa + 13$ die gleiche ist wie bei der $\tanh$-Funktion. Der Beweis der rechten Ungleichung in (91) bleibt dem Leser überlassen.

[23] $\alpha := [1, 1 + 2^{-2097}, 1]$ kann mit der Punktintervall-Konstanten `One_p_lx_interval()` sehr einfach realisiert werden.

Im **1. Beispiel** liefert das nachfolgende Programm für das in der Klasse `lx_interval` maximale Argument $x = 2^{+2147483647} \cdot \texttt{MaxReal} = 2^{+2147483647} \cdot 1.79769\ldots \cdot 10^{+308}$ eine Einschließung des Funktionswertes $y = \coth(x)$.

```
// Programm  lx_test44.cpp;
// Zum Test der coth(x)-Funktion;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;

    X = lx_interval(2147483647,l_interval(MaxReal,MaxReal));
    Y = coth(X);

    cout << SetDotPrecision(16*stagprec,16*stagprec+10)
         << Scientific;

    s << Y;
    cout << "coth(X) = " << s << endl;
    cout << "coth(X) = " << Y << endl;
}
```

Das obige Programm `lx_test44.cpp` liefert bei der 1. Ausgabe mit `Y` die folgende optimale Einschließung

$$\coth(x) \in \texttt{Y} = 2^{-1022} \cdot [\underbrace{4.4942328371557\ldots3040000\ldots000}_{631 \text{ korrekte Dez.-Ziffern}}0000 \cdot 10^{+307}, \\ 4.4942328371557\ldots3040000\ldots0004941 \cdot 10^{+307}]$$

mit 631 korrekten Dezimalziffern. Die zweite Ausgabe erfolgt in lesbarer, dezimaler Form und liefert

$$\coth(x) \in \texttt{Y} = 10^{-1} \cdot [\underbrace{9.999999\ldots9999999}_{302 \text{ korrekte Dez.-Ziffern}}873\ldots, 1.000000\ldots00000115\ldots \cdot 10^{1}]$$

mit nur 302 korrekten Dezimalziffern. Bei dieser letzten dezimalen Ausgabe erhält man wegen der internen Umrechnungen mit der Logarithmus-Funktion nur noch 302 korrekte Dezimalziffern, die jedoch über die interne Genauigkeit von 631 Dezimalziffern keine Auskunft liefert! Beachten Sie bitte, dass das Intervall `X` ein Punktintervall ist.

Im **2. Beispiel** liefert das folgende Programm für $x = 2^{-2147482626}$ eine Einschließung des Funktionswertes $y = \coth(x)$.

```
// Programm  lx_test45.cpp;
// Zum Test der coth(x)-Funktion;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;

    X = lx_interval(-2147482626,l_interval(1));
    Y = coth(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec+10)
         << Scientific;
    s << Y;
    cout << "coth(X) = " << s << endl;
}
```

Die folgende Ausgabe von `Y` über die Zeichenkette `s` liefert mit dem Faktor $2^{+2147482115}$ die Einschließung von $\coth(x)$ mit der intern berechneten Genauigkeit von 476 korrekten Dezimalstellen:

$$\coth(x) \in \mathtt{Y} = 2^{+2147482115} \cdot [\underbrace{6.70390396497129\ldots047999\ldots9998517\ldots}_{476\text{ korrekte Dez.-Ziffern}} \cdot 10^{+153},$$
$$6.70390396497129\ldots048000\ldots0001482\ldots \cdot 10^{+153}]$$

Wegen des Zählers $(2 + t(x))$ in (89) und wegen der notwendigen Division durch $-t(x)$ erhalten wir jetzt eine Genauigkeit von *nur* 476 Dezimalstellen, während z.B. bei der $\arctan(x)$-Funktion mit $x = 2^{-2147482626}$ eine Genauigkeit von 630 Dezimalstellen erreicht wird, vgl. Seite 86.

5.2.32 Die Funktion $\sqrt{1+x}-1$

Für $x \in [-1,+4]$ ist der Graph der Funktion $\sqrt{1+x}-1$ dargestellt in Abb. 9.

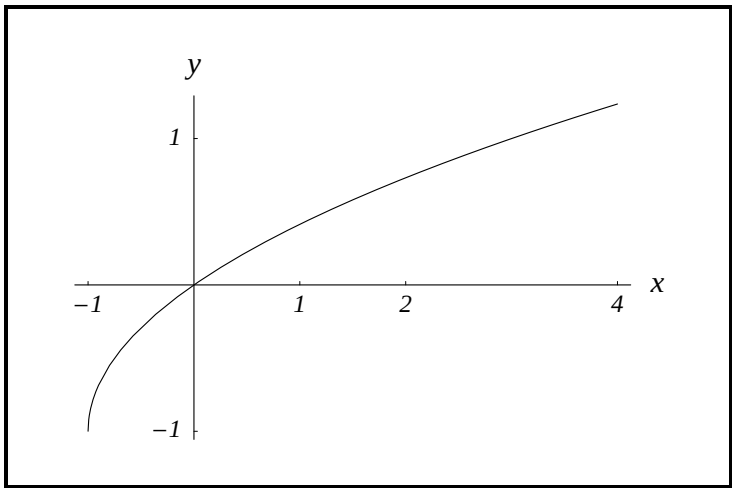


Abbildung 9: $y = \sqrt{1+x}-1$

Die Funktion $f(x) = \sqrt{1+x}-1 \equiv \dfrac{x}{\sqrt{1+x}+1}$ kann für alle $x \geq -1$ definiert werden durch

$$f(x) := \begin{cases} \dfrac{x}{\sqrt{1+x}+1}, & \text{falls } -1 \leq x < 0.1, \\ \sqrt{1+x}-1, & \text{falls } x \geq 0.1; \end{cases}$$

Ist dann $\boldsymbol{u} = [u_1, u_2]$, mit $u_1 \geq -1$ ein Eingangsintervall, so wird die C-XSC Funktion

```
lx_interval sqrtp1m1(const lx_interval& u)
```

implementiert durch

$$f(x) \in \begin{cases} \dfrac{\boldsymbol{u}}{\sqrt{1+\boldsymbol{u}}+1}, & \forall x \in \boldsymbol{u} \subseteq [-0.1,+0.1], \\ \sqrt{1+\boldsymbol{u}}-1, & \text{sonst.} \end{cases}$$

Um auf der Maschine garantierte Einschließungen für $f(x),\ x \in \boldsymbol{u}$, zu erhalten, werden in `sqrtp1m1(...)` die obigen von $\boldsymbol{u}$ abhängigen Terme intervallmäßig ausgewertet.

Im **1. Beispiel** liefert das nachfolgende Programm `lx_test46.cpp` für das maximale Argument $x = 2^{+2147483646}$ eine Einschließung des Funktionswertes $f(x) = \sqrt{1+x}-1$.

```
// Programm  lx_test46.cpp;
// Zum Test von sqrt(1+x)-1;

#include <iostream>
#include "lx_imath.hpp"
```

```
using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 30;
    lx_interval X,Y;
    string s;

    X = lx_interval(+2147483646,l_interval(1));
    Y = sqrtp1m1(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    s << Y;
    cout << "sqrtp1m1(X) = " << s << endl;
}
```

Die folgende Ausgabe von Y über die Zeichenkette s liefert mit dem Faktor $2^{+1073740802}$ die Einschließung von $f(x) = \sqrt{1+x} - 1$ mit einer intern berechneten Genauigkeit von 477 korrekten Dezimalstellen:

$$\sqrt{1+x} - 1 \in \mathsf{Y} = 2^{+1073740802} \cdot [\underbrace{2.247116418577\ldots151999\ldots9998}_{477 \text{ korrekte Dez.-Ziffern}}343 \cdot 10^{+307},$$
$$2.247116418577\ldots152000\ldots0001657 \cdot 10^{+307}]$$

Im **2. Beispiel** liefert das nachfolgende Programm `lx_test47.cpp` für das minimale Argument $x = 2^{-2147482627}$ eine Einschließung des Funktionswertes $f(x) = \sqrt{1+x} - 1$.

```
// Programm  lx_test47.cpp;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 30;
    lx_interval X,Y;
    string s;

    X = lx_interval(-2147482627,l_interval(1));
    Y = sqrtp1m1(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    s << Y;
    cout << "sqrtp1m1(X) = " << s << endl;
}
```

Die folgende Ausgabe von Y über die Zeichenkette s liefert mit dem Faktor $2^{-2147483139}$ die Einschließung von $f(x) = \sqrt{1+x} - 1$ mit einer intern berechneten Genauigkeit von 476 korrekten Dezimalstellen:

$$\sqrt{1+x} - 1 \in \mathsf{Y} = 2^{-2147483139} \cdot [\underbrace{6.703903964971\ldots047999\ldots99985178}_{476 \text{ korrekte Dez.-Ziffern}} \cdot 10^{+153},$$
$$6.703903964971\ldots048000\ldots00014822 \cdot 10^{+153}]$$

Im **3. Beispiel** liefert das nachfolgende Programm `lx_test48.cpp` für das Argument $x = 10^{-3000} \cdot \sqrt{2}$ eine Einschließung des Funktionswertes $f(x) = \sqrt{1+x} - 1$.

```
// Programm  lx_test48.cpp;
// Test von sqrt(1+x)-1;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 30;
    lx_interval X,Y;

    X = Sqrt2_lx_interval() * lx_interval(-3000,"[1,1]");
    Y = sqrtp1m1(X);

    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "sqrtp1m1(X) = " << Y << endl;
}
```

Das obige Programm liefert mit Y die folgende garantierte Einschließung

$$\sqrt{1+x}-1 \in \mathsf{Y} = 10^{-3001} \cdot \underbrace{7.071067811865475244008\ldots30181399762570399}_{299 \text{ korrekte Dez.-Ziffern}} {}^{515\ldots}_{452\ldots} \cdot 10^{0}$$

mit 299 korrekten Dezimalziffern. Es ist zu beachten, dass sowohl $x = 10^{-3000}$ als auch $\sqrt{2}$ im Binärsystem nicht exakt darstellbar sind, so dass X und damit auch Y keine Punktintervalle sein können. Es ist klar, dass $f(x)$ wegen des Faktors 10^{-3000} in der Klasse `l_interval` nicht eingeschlossen werden kann und dass dieser Faktor in der Klasse `lx_interval` auch noch sehr viel kleiner hätte gewählt werden können.

Mit *Mathematica* lassen sich die Beispiele 1. und 2. wegen entsprechender Overflow-Meldungen nicht nachrechnen. Die mit *Mathematica* berechnete Approximation für den Funktionswert $f(x)$ aus Beispiel 3. wird durch das Intervall Y jedoch eingeschlossen.

5.2.33 Die Funktion arsinh (x)

Für $x \in [-4, +4]$ ist der Graph der Funktion arsinh (x) dargestellt in Abb. 10.

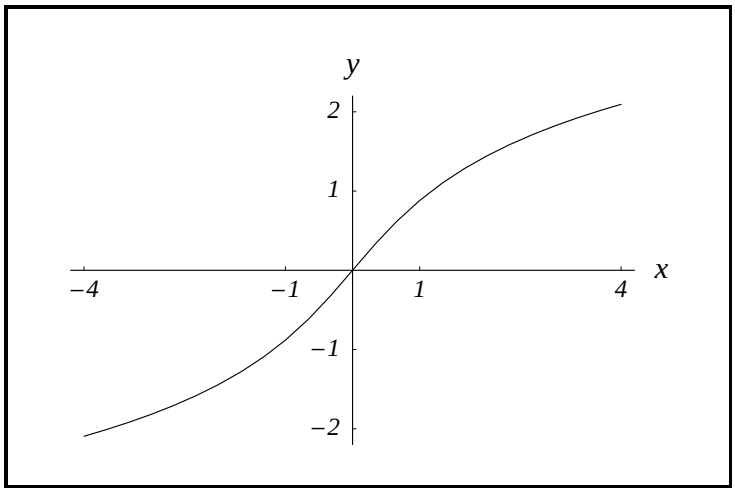


Abbildung 10: $y = \text{arsinh}\,(x)$

Zur hyperbolischen sin-Funktion kann deren Umkehrfunktion $f(x) = \text{arsinh}\,(x)$ für alle $x \in \mathbb{R}$ definiert werden durch

$$\text{arsinh}\,(x) = \ln(x + \sqrt{1 + x^2}), \quad x \in \mathbb{R}. \tag{92}$$

Die C-XSC Funktion

```
lx_interval asinh(const lx_interval &u);
```

liefert garantierte Einschließungen für das reelle Intervall $\boldsymbol{y} = \text{arsinh}\,(\boldsymbol{u})$, $\boldsymbol{u} = [u_1, u_2]$, mit $-\infty < u_1 \leq u_2 < +\infty$. Bei der Implementierung wird zunächst der Fall $\boldsymbol{u} = 0$ behandelt und danach der Fall $0 \in \boldsymbol{u}$. Wegen der Punktsymmetrie zum Ursprung kann man sich dann auf $u_1 > 0$ beschränken.

Für $u_2 \to 0$ ist (92) wegen starker Auslöschung nicht geeignet. Wir betrachten daher zunächst die Taylorreihe

$$\text{arsinh}\,(x) = x + \sum_{k=1}^{\infty} \frac{1 \cdot 3 \cdot 5 \cdot \ldots \cdot (2k-1) \cdot (-1)^k x^{2k+1}}{2 \cdot 4 \cdot 6 \cdot \ldots \cdot (2k)(2k+1)}, \quad |x| < 1, \tag{93}$$

$$= x \cdot \left\{ 1 - \frac{x^2}{6} + \frac{3x^4}{40} - \frac{15x^6}{336} + - \ldots \right\}. \tag{94}$$

Da in (94) die Reihe rechts in der Klammer eine alternierende Leibniz-Reihe ist, gilt für $0 \leq x < 1$ die Doppelungleichung

$$x \cdot \left(1 - \frac{x^2}{6} \right) \leq \text{arsinh}\,(x) \leq x, \quad 0 \leq x < 1. \tag{95}$$

Um in (95) den bei der Auswertung von x^2 auftretenden integer-Überlauf für $x \to 0$ zu vermeiden, benutzt man die für $x \geq 0$ allgemeingültige, etwas gröbere Abschätzung

$$x \cdot \left(1 - \frac{x}{2}\right) \cdot \left(1 + \frac{x}{2}\right) \equiv x \cdot \left(1 - \frac{x^2}{4}\right) \leq x \cdot \left(1 - \frac{x^2}{6}\right)$$

und erhält damit

$$x \cdot \left(1 - \frac{x}{2}\right) \cdot \left(1 + \frac{x}{2}\right) \leq \text{arsinh}\,(x) \leq x, \quad 0 \leq x < 1. \tag{96}$$

In (96) wird dabei die Einschließung von arsinh (x) umso besser, je kleiner man $x \geq 0$ wählt. Für das Eingangsintervall $\boldsymbol{u} = [u_1, u_2]$, mit $0 \leq u_1 \leq u_2 < 1$ gilt dann

$$\text{arsinh}\,(x) \in [\texttt{Inf}(\boldsymbol{u} \diamond (1 \diamond \boldsymbol{u} \diamond 2) \diamond (1 \diamond \boldsymbol{u} \diamond 2)), u_2] \quad \forall x \in \boldsymbol{u}. \tag{97}$$

Der Vorteil von (97) besteht nun darin, dass die Unterschranke jetzt sehr effektiv und vor allem für $u_2 \to 0$ ohne vorzeitigen integer-Überlauf berechnet werden kann. Testrechnungen haben dabei ergeben, dass man mit `stagprec` $= 39$ und Punktintervallen $\boldsymbol{u}$, mit $u_2 < 2^{-790}$, eine maximale Genauigkeit der Einschließungen von ca. 474 korrekten Dezimalziffern erhält.

Im Fall $2^{-790} \leq u_2 < 1$ kann mit (92) immer noch starke Auslöschung auftreten, die man jedoch mit Hilfe der Identität

$$\ln(x + \sqrt{1 + x^2}) \equiv \ln(1 + \{x + (\sqrt{1 + x^2} - 1)\}) \tag{98}$$

vermeiden kann, wenn die innere Klammer $(\sqrt{1 + x^2} - 1)$ mit Hilfe der C-XSC Funktion `sqrtp1m1(sqr(u))` intervallmäßig auswertet und der ganze Term rechts in (98) mit Hilfe der C-XSC Funktion `lnp1( u+sqrtp1m1(sqr(u)) )` eingeschlossen wird. Dabei ist zu beachten, dass jetzt wegen $u_2 \geq 2^{-790}$ bei nicht zu breiten Intervallen $\boldsymbol{u}$ die Auswertung von `sqr(u)` keinen integer-Überlauf mehr verursachen kann.

Nur im letzten Fall $u_2 > 1$ kommt schließlich (92) direkt zur Anwendung, wobei der Ausdruck $\sqrt{1 + x^2}$ mit Hilfe der C-XSC Funktion `sqrt1px2(u)` ohne vorzeitigen integer-Überlauf eingeschlossen werden kann.

Im **1. Beispiel** liefert das nachfolgende Programm `lx_test49.cpp` für das maximale Argument $x = 2^{+2147483645}$ eine Einschließung des Funktionswertes $f(x) = \text{arsinh}\,(x)$.

```
// Programm  lx_test49.cpp;
// Zum Test von asinh(x);

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;
```

```
int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;

    X = lx_interval(+2147483645,l_interval(1));
    Y = sqrtp1m1(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    s << Y;
    cout << "asinh(X) = " << s << endl;
}
```

Die folgende Ausgabe von Y über die Zeichenkette s liefert mit dem Faktor $2^{+1073740801}$ die Einschließung von $f(x) = \text{arsinh}\,(x)$ mit einer intern berechneten Genauigkeit von 477 korrekten Dezimalstellen:

$$\text{arsinh}\,(x) \in \mathsf{Y} = 2^{+1073740801} \cdot \underbrace{3.17790251538411\ldots160455478}_{477\ \text{korrekte Dez.-Ziffern}}{}^{422\ldots}_{257\ldots} \cdot 10^{+307}$$

Im **2. Beispiel** liefert das nachfolgende Programm `lx_test50.cpp` für das minimale Argument $x = 2^{-2147482626}$ eine Einschließung des Funktionswertes $f(x) = \text{arsinh}\,(x)$.

```
// Programm  lx_test49.cpp;
// Zum Test von asinh(x);

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;

    X = lx_interval(-2147482626,l_interval(1));
    Y = sqrtp1m1(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    s << Y;
    cout << "asinh(X) = " << s << endl;
}
```

Die folgende Ausgabe von Y über die Zeichenkette S liefert mit dem Faktor $2^{-2147483138}$ die Einschließung von $f(x) = \text{arsinh}\,(x)$ mit einer intern berechneten Genauigkeit von 476 korrekten Dezimalstellen:

$$\text{arsinh}\,(x) \in \mathsf{Y} = 2^{-2147483138} \cdot [\underbrace{6.703903964971\ldots047999\ldots999}_{476\text{ korrekte Dez.-Ziffern}}851\ldots\cdot 10^{+153},$$
$$6.703903964971\ldots048000\ldots000148\ldots\cdot 10^{+153}].$$

Der Versuch, die Funktionswerte der beiden letzten Beispiele mit *Mathematica* nachzurechnen, scheitert an entsprechenden Overflow-Meldungen.

Im **3. Beispiel** wird die Identität

$$\text{arsinh}\,(\,\sinh(x)\,) \equiv x, \quad x \in \mathbb{R}$$

für $x = \sqrt{2}$ benutzt. Dabei wird die nicht darstellbare Zahl $\sqrt{2}$ mit Hilfe der Intervall-Konstanten `Sqrt2_lx_interval()` $\ni \sqrt{2}$ in hoher Genauigkeit eingeschlossen.

```
// Programm  lx_test51.cpp;
// Zum Test von asinh(x);

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;

    X = Sqrt2_lx_interval();
    Y = asinh( sinh(X) );
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "asinh(X) = " << Y << endl;
    if (X<Y) cout << "X in Y enthalten" << endl;
    else cout << "X NICHT in Y enthalten" << endl;
}
```

Die Programmausgabe liefert mit Y eine Einschließung für $\sqrt{2}$ mit *nur* noch 302 korrekten Dezimalziffern

$$\sqrt{2} \in \mathsf{Y} = 10^{-1} \cdot \underbrace{1.41421356237309504880\ldots6279952514079896}_{302\text{ korrekte Dez.-Ziffern}}{}^{919\ldots}_{833\ldots} \cdot 10^{+1}$$

und die Meldung: `X in Y enthalten`, da bei der Berechnung von Y die unvermeidbaren Rundungsfehler zu einer etwas gröberen Einschließung von $\sqrt{2}$ führen.

5.2.34 Die Funktion arcosh (x)

Für $x \in [1,4]$ ist der Graph der Funktion arcosh (x) dargestellt in Abb. 11.

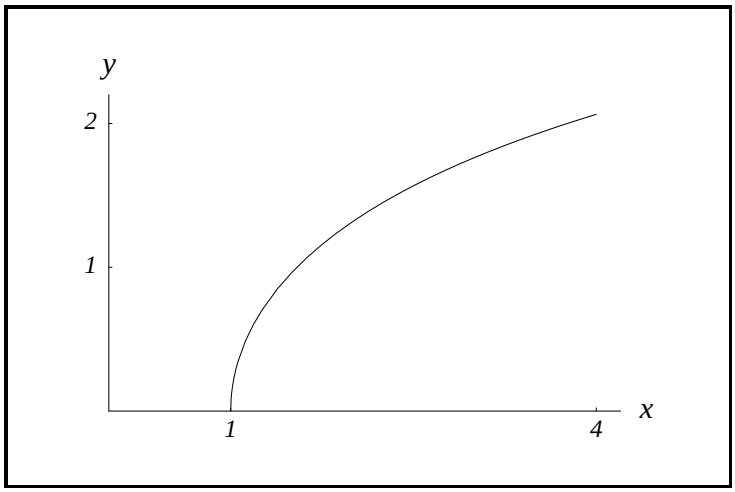


Abbildung 11: $y = \text{arcosh}\,(x)$

Zur hyperbolischen cos-Funktion wird deren Umkehrfunktion $f(x) = \text{arcosh}\,(x)$ für alle $x \geq 1$ definiert durch:

(99) $$\text{arcosh}\,(x) = \ln(x + \sqrt{x^2 - 1}), \quad x \geq 1.$$

Die C-XSC Funktion

```
lx_interval acosh(const lx_interval &u);
```

liefert garantierte Einschließungen für das reelle Intervall $\boldsymbol{y} = \text{arcosh}\,(\boldsymbol{u})$, $\boldsymbol{u} = [u_1, u_2]$, mit $1 \leq u_1 \leq u_2 < +\infty$, wobei das Intervall $\boldsymbol{u}$ bei einer sinnvollen Anwendung der Staggered Arithmetik nicht zu breit gewählt werden darf.

Für $x \to +1$ wird der Ausdruck rechts in (99) in der Nähe der Nullstelle der $\ln$-Funktion ausgewertet, so dass starke Auslöschung zu erwarten ist. Für $x \to +1$ betrachten wir daher mit der Pochhammer-Symbolik

$$(z)_0 := 1; \quad (z)_0 := z \cdot (z+1) \cdot (z+2) \cdot \ldots \cdot (z+k-1), \quad k = 1, 2, 3, \ldots$$

$$\text{d.h. mit} \quad \left(\frac{1}{2}\right)_k = \frac{1 \cdot 3 \cdot 5 \cdot \ldots \cdot (2k-1)}{2^k}, \quad k = 1, 2, 3, \ldots$$

die Taylorreihe um den Entwicklungspunkt $x_0 = 1$

(100)
$$\begin{aligned} \text{arcosh}\,(x) &= \sqrt{2 \cdot (x-1)} \cdot \sum_{k=0}^{\infty} \frac{\left(\frac{1}{2}\right)_k \cdot (-1)^k \cdot (x-1)^k}{2^k \cdot (2k+1) \cdot k!}, \quad 0 \leq x - 1 < 2; \\ &= \sqrt{2 \cdot (x-1)} \cdot \left(1 - \frac{x-1}{12} + \frac{3}{160} \cdot (x-1)^2 - + \ldots\right) \end{aligned}$$

Da die Reihe rechts in (100) für $0 \leq x-1 < 2$ eine alternierende Leibniz-Reihe ist, gilt die Doppelungleichung

$$\sqrt{2x-2}\cdot\left(1-\frac{x-1}{12}\right) \leq \operatorname{arcosh}(x) \leq \sqrt{2x-2}, \quad 0 \leq x-1 < 2,$$

wobei die linke Seite wegen $1-(x-1)/12 \geq 2-x$ für eine effektive Auswertung noch weiter vereinfacht werden kann:

$$\sqrt{2x-2}\cdot(2-x) \leq \operatorname{arcosh}(x) \leq \sqrt{2x-2}, \quad 0 \leq x-1 < 2. \tag{101}$$

Wir betrachten jetzt das Eingangsintervall $\boldsymbol{u} = [u_1, u_2]$, mit $1 \leq u_1 \leq u_2 < 2$. Da die obige Einschließung von $\operatorname{arcosh}(x)$ nur hinreichend eng ist für $x \to +1$, kommt (101) nur im Fall $u_2 - 1 < 2^{-1600}$ zur Anwendung. Dabei werden die Terme $\sqrt{2x-2}\cdot(2-x)$ und $\sqrt{2x-2}$ intervallmäßig ausgewertet, indem x durch $\boldsymbol{u}$ ersetzt wird. Die entsprechenden Unter- und Oberschranken dieser Einschließungen liefern dann die garantierte Einschließung von $\operatorname{arcosh}(x)\ \forall x \in \boldsymbol{u} = [u_1, u_2]$.

Danach wird im Fall $u_2 - 1 < 1$ zur Vermeidung möglicher Auslöschungen in (99) mit $t := x-1$ die folgende Identität benutzt:

$$\operatorname{arcosh}(x) := \ln(x+\sqrt{x^2-1}) \equiv \ln\left(1+\left\{t+\sqrt{t\cdot(2+t)}\right\}\right), \quad t = x-1 \geq 0.$$

Zur Einschließung von $\operatorname{arcosh}(x)$ wird der obige rechte Term wieder intervallmäßig mit Hilfe der C-XSC Funktion `lnp1(...)` ausgewertet, wobei x durch $\boldsymbol{u}$ zu ersetzen ist. Erst im Fall $u_2 - 1 \geq 1$ kommt schließlich der einfachere Term in (99) zur Anwendung, wobei $\sqrt{\boldsymbol{u}^2-1}$ mit Hilfe der C-XSC Funktion `sqrtx2m1(u)` intervallmäßig auszuwerten ist, um einen vorzeitigen integer-Überlauf bei der Berechnung von $\boldsymbol{u}^2$ zu vermeiden.

Im **1. Beispiel** liefert das nachfolgende Programm `lx_test52.cpp` für das Argument $x = 1 + 2^{-2095}$ eine Einschließung des Funktionswertes $f(x) = \operatorname{arcosh}(x)$.

```
// Programm  lx_test52.cpp;
// Zum Test von acosh(x);

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;

    X = lx_interval(0,l_interval(1));
```

```
    X += lx_interval(-2095,l_interval(1));

    Y = acosh(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    s << Y;
    cout << "acosh(X) = " << s << endl;
}
```

Die folgende Ausgabe von Y über die Zeichenkette s liefert mit dem Faktor 2^{-2069} die Einschließung von $f(x) = \operatorname{arcosh}(x)$ mit einer intern berechneten Genauigkeit von 476 korrekten Dezimalstellen:

$$\operatorname{arcosh}(x) \in \mathsf{Y} = 2^{-2069} \cdot [\underbrace{4.4942328371557\ldots303999\ldots999}_{476 \text{ korrekte Dez.-Ziffern}}876\ldots \cdot 10^{+307},$$
$$4.4942328371557\ldots304000\ldots000331\ldots \cdot 10^{+307}].$$

Der mit *Mathematica* auf 490 Dezimalstellen berechnete Funktionswert wird durch das Intervall Y eingeschlossen.

Zeigen Sie mit dem obigen Programm `lx_test52.cpp`, dass mit $x = 1 + 2^{-2060}$ die entsprechende Intervall-Addition

```
X += lx_interval(-2060,l_interval(1));
```

kein punktförmiges Ergebnisintervall X mehr liefert, so dass mit Y nur noch eine sehr grobe Einschließung für $\operatorname{arcosh}(x)$ berechnet werden kann. Sollen Funktionswerte auch mit Argumenten $x = 1 + 2^t,\ t < -2059$ in hoher Genauigkeit eingeschlossen werde, so muss die Funktion `acoshp1(...)` benutzt werden, vgl. dazu den nächsten Abschnitt.

Im **2. Beispiel** liefert das nachfolgende Programm `lx_test53.cpp` für das maximale Argument $x = 2^{+2147483646}$ eine Einschließung des Funktionswertes $f(x) = \operatorname{arcosh}(x)$.

```
// Programm  lx_test53.cpp;
// Zum Test von acosh(x);

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;

    X = lx_interval(+2147483646,l_interval(1));
    Y = acosh(X);
```

```
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    s << Y;
    cout << "acosh(X) = " << s << endl;
}
```

Die folgende Ausgabe von Y über die Zeichenkette s liefert mit dem Faktor $2^{1073740802}$ die Einschließung von $f(x) = \operatorname{arcosh}(x)$ mit einer intern berechneten Genauigkeit von 477 korrekten Dezimalstellen:

$$\operatorname{arcosh}(x) \in \mathsf{Y} = 2^{1073740802} \cdot [\underbrace{2.2471164185778\ldots151999\ldots999}_{477\text{ korrekte Dez.-Ziffern}}834\ldots \cdot 10^{+307},$$
$$2.2471164185778\ldots152000\ldots000165\ldots \cdot 10^{+307}].$$

Die Berechnung einer Näherung für $\operatorname{arcosh}(2^{+2147483646})$ scheitert mit *Mathematica* an einer Overflow-Meldung.

Im **3. Beispiel** liefert das folgende Programm für das im Binärsystem nicht darstellbare Argument $x = 2.001$ eine garantierte Einschließung für $\operatorname{arcosh}(2.001)$.

```
// Programm  lx_test54.cpp;
// Zum Test von acosh(x);

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;

    X = lx_interval(0,"[2.001,2.001]");
    Y = sqrtp1m1(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "acosh(X) = " << Y << endl;
}
```

Das obige Programm liefert mit Y die folgende garantierte Einschließung

$$\operatorname{arcosh}(2.001) \in \mathsf{Y} = 10^{-1} \cdot \underbrace{1.3175350548400828\ldots6977472596835510829}_{301\text{ korrekte Dez.-Ziffern}}{}^{544\ldots}_{414\ldots} \cdot 10^{1}$$

mit 301 korrekten Dezimalziffern. Eine mit dem Algebra-System *Mathematica* auf 308 Dezimalstellen berechnete Näherung für $\operatorname{arcosh}(2.001)$ wird durch das obige Intervall Y eingeschlossen.

5.2.35 Die Funktion arcosh $(1 + x)$

Für $x \in [0, 4]$ ist der Graph der Funktion arcosh $(1 + x)$ dargestellt in Abb. 12.

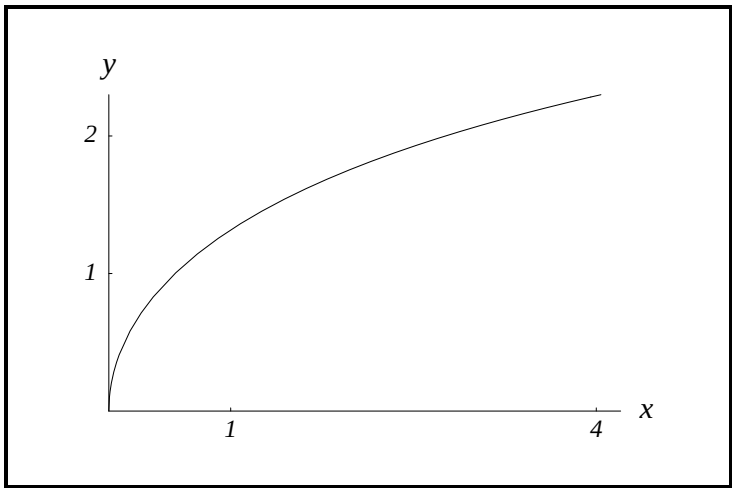


Abbildung 12: $y = \text{arcosh}\,(1 + x)$

Die Funktion $f(x) = \text{arcosh}\,(1 + x)$ ist für alle $x \geq 0$ definiert durch:

(102) $$\text{arcosh}\,(1 + x) = \ln(1 + \{x + \sqrt{x \cdot (2 + x)}\}), \quad x \geq 0,$$

dabei erhält man (102), indem man in (99) x durch $1 + x$ ersetzt. Die C-XSC Funktion

```
lx_interval acoshp1(const lx_interval &u);
```

liefert garantierte Einschließungen für das reelle Intervall $\boldsymbol{y} = \text{arcosh}\,(1+\boldsymbol{u}), \boldsymbol{u} = [u_1, u_2]$, mit $0 \leq u_1 \leq u_2 < +\infty$, wobei das Intervall $\boldsymbol{u}$ bei einer sinnvollen Anwendung der Staggered Intervall-Arithmetik nicht zu breit gewählt werden darf.

Für $x \to 0$ kann die rechte Seite von (102) mit Hilfe der C-XSC Funktion `lnp1`$\{...\}$ in hoher Genauigkeit eingeschlossen werden, wenn im Argument $\{x + \sqrt{x \cdot (2 + x)}\}$ die Variable x durch das Intervall $\boldsymbol{u}$ ersetzt wird. Führt man in (101) die Transformation $x \to 1 + x$ durch, so erhält man

(103) $$\sqrt{2x} \cdot (1 - x) \leq \text{arcosh}\,(1 + x) \leq \sqrt{2x}, \quad 0 \leq x < 2;$$

und für $u_2 < 2^{-1600}$ liefert dies eine sehr effektive Einschließung für arcosh $(1+x)$, wenn man die Terme $\sqrt{2x} \cdot (1-x)$ und $\sqrt{2x}$ intervallmäßig auswertet und mit den entsprechenden Unter- und Oberschranken die Einschließung für arcosh $(1 + x)$ $\forall x \in \boldsymbol{u}$ realisiert. Im Fall $u_2 < 1$ kommt dann die rechte Seite von (102) zur Anwendung, sonst wird zur Einschließung von arcosh $(1 + x)$ die rechte Seite von

$$\text{arcosh}\,(1 + x) = \ln(t + \sqrt{t^2 - 1}), \quad t := 1 + x,\ x \geq 1,$$

intervallmäßig ausgewertet, indem wieder x durch das Eingangsintervall $\boldsymbol{u}$ ersetzt wird.

Im **1. Beispiel** liefert das nachfolgende Programm `lx_test55.cpp` für das minimale Argument $x = 2^{-2147482626}$ eine Einschließung für $f(x) = \operatorname{arcosh}(1+x)$.

```
// Programm  lx_test55.cpp;
// Zum Test von acosh(1+x);

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;

    X = lx_interval(-2147482626,l_interval(1));
    Y = acoshp1(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    s << Y;
    cout << "acoshp1(X) = " << s << endl;
}
```

Die folgende Ausgabe von Y über die Zeichenkette s liefert mit dem Faktor $2^{-1073742335}$ die Einschließung von $f(x) = \operatorname{arcosh}(1+x)$ mit einer intern berechneten Genauigkeit von 475 korrekten Dezimalstellen:

$$\operatorname{arcosh}(1+x) \in \mathsf{Y} = 2^{-1073742335} \cdot \underbrace{6.35580503076823\ldots1231832091095}_{475 \text{ korrekte Dez.-Ziffern}}{}^{684\ldots}_{524\ldots} \cdot 10^{+307}$$

Im **2. Beispiel** liefert das folgende Programm für das im Binärsystem nicht darstellbare Argument $x = e^{-50000\cdot\sqrt{\pi}}$ eine garantierte Einschließung für $\operatorname{arcosh}(1+x)$.

```
// Programm  lx_test56.cpp;
// Zum Test von acosh(1+x);

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
```

```
    lx_interval X,Y;

    X = exp(-50000 * SqrtPi_lx_interval());
    Y = acoshp1(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "acoshp1(X) = " << Y << endl;
}
```

Das obige Programm liefert mit Y die folgende garantierte Einschließung

$$\operatorname{arcosh}(1+x) \in \mathsf{Y} = 10^{-19245} \cdot \underbrace{9.491679975163393\ldots896840259235745}_{301 \text{ korrekte Dez.-Ziffern}} \,{}^{195\ldots}_{093\ldots} \cdot 10^{0}$$

mit 301 korrekten Dezimalziffern. Eine mit dem Algebra-System *Mathematica* auf 308 Dezimalstellen berechnete Näherung für $\operatorname{arcosh}\left(1 + e^{-50000 \cdot \sqrt{\pi}}\right)$ wird durch das obige Intervall Y eingeschlossen.

5.2.36 Die Funktion artanh (x)

Für $x \in (-1, +1)$ ist der Graph der Funktion artanh (x) dargestellt in Abb. 13.

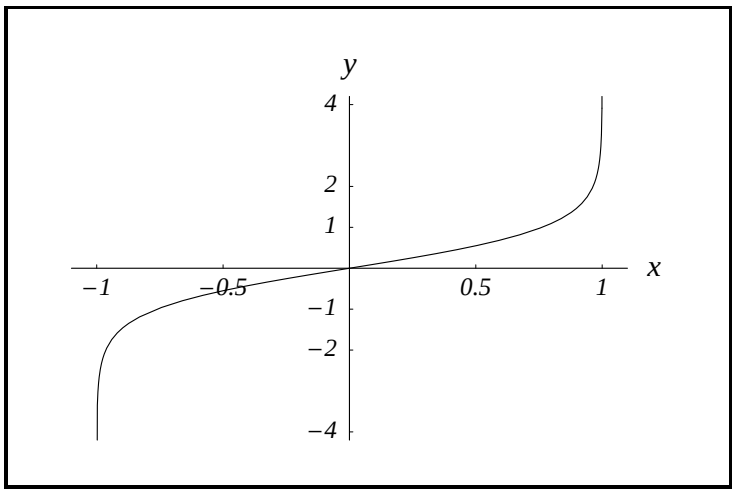


Abbildung 13: $y = \text{artanh}\,(x)$

Zur hyperbolischen tan-Funktion wird deren Umkehrfunktion $f(x) = \text{artanh}\,(x)$ für alle $x \in (-1, +1)$ definiert durch:

$$\text{artanh}\,(x) = \frac{1}{2} \cdot \ln\left(\frac{1+x}{1-x}\right), \quad -1 < x < +1. \tag{104}$$

Die C-XSC Funktion

```
lx_interval atanh(const lx_interval &u);
```

liefert garantierte Einschließungen für das reelle Intervall $\boldsymbol{y} = \text{artanh}\,(\boldsymbol{u})$, $\boldsymbol{u} = [u_1, u_2]$, mit $-1 < u_1 \leq u_2 < +1$, wobei das Intervall $\boldsymbol{u}$ bei einer sinnvollen Anwendung der Staggered Intervall-Arithmetik nicht zu breit gewählt werden darf.

Die Logarithmusfunktion rechts in (104) wird für $x \to 0$ in der Nähe ihrer Nullstelle $x_0 = 1$ ausgewertet. Um die dabei auftretenden starken Auslöschungen zu vermeiden, kommt im Fall $|u_2| < 0.25$ folgende Identität zur Anwendung:

$$\text{artanh}\,(x) = \frac{1}{2} \cdot \ln\left(1 + \frac{2x}{1-x}\right), \quad -1 < x < +1, \tag{105}$$

dabei wird $2x/(1-x)$ intervallmäßig ausgewertet und als Argument der bereits implementierten C-XSC Funktion `lnp1(...)` verwendet. Im Fall $0.25 \leq |u_2| < 1$ wird (104) ebenfalls intervallmäßig ausgewertet, indem x durch das Eingangsintervall $\boldsymbol{u}$ ersetzt wird.

Liegt x z.B. zu dicht bei der Singularität $+1$, so wird bei der intervallmäßigen Auswertung von $1-x$ rechts in (104) die Null mit eingeschlossen, so dass die Fehlermeldung *DIV_BY_ZERO* entsteht.

Wird das Eingangsargument $\boldsymbol{u}$ als Differenz $\boldsymbol{u} = 1 \diamondsuit 2^{-p}$ berechnet, so erhält man mit mit $p \geq 2096$ eine Fehlermeldung, da in diesem Fall das Maschinenintervall $\boldsymbol{u}$ die Eins mit einschließt, wodurch eine entsprechende Fehlermeldung erzeugt wird, vgl. dazu das 1. Beispiel auf der nächsten Seite.

Im **1. Beispiel** liefert das nachfolgende Programm `lx_test57.cpp` für das Intervall-Argument $x = 1 - 2^{-2095}$ eine sehr enge Einschließung für $f(x) = \text{artanh}\,(x)$.

```
// Programm  lx_test57.cpp;
// Zum Test von atanh(x);

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;

    X = 1 - lx_interval(-2095,l_interval(1));
    Y = atanh(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    s << Y;
    cout << "atanh(X) = " << s << endl;
}
```

Die folgende Ausgabe von `Y` über die Zeichenkette `s` liefert mit dem Faktor 2^{-1012} die Einschließung von $f(x) = \text{artanh}\,(x)$ mit einer intern berechneten Genauigkeit von 473 korrekten Dezimalstellen:

$$\text{artanh}\,(x) \in \texttt{Y} = 2^{-1012} \cdot \underbrace{3.18817649531979839\ldots9508193890254331}_{473\ \text{korrekte Dez.-Ziffern}}{}^{200\ldots}_{195\ldots} \cdot 10^{+307}$$

Wenn für Argumente x, die noch dichter bei +1 liegen, eine Einschließung für $\text{artanh}\,(x)$ in gleich hoher Genauigkeit zu berechnen ist, so muss dies mit Hilfe der C-XSC Funktion `atanh1m(...)` erfolgen, vgl. den nächsten Abschnitt.

Im **2. Beispiel** liefert das nachfolgende Programm `lx_test58.cpp` für das minimale positive Argument $x = 2^{-2147482627}$ eine Einschließung für $f(x) = \text{artanh}\,(x)$.

```
// Programm  lx_test58.cpp;
// Zum Test von atanh(x);

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;
```

```
int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;

    X = lx_interval(-2147482627,l_interval(1));
    Y = atanh(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    s << Y;
    cout << "atanh(X) = " << s << endl;
}
```

Die folgende Ausgabe von Y über die Zeichenkette s liefert mit dem Faktor $2^{-2147483647}$ die Einschließung von $f(x) = \text{artanh}\,(x)$ mit einer intern berechneten Genauigkeit von 477 korrekten Dezimalstellen:

$$\text{artanh}\,(x) \in \mathsf{Y} = 2^{-2147483647} \cdot [\underbrace{1.1235582092889\ldots575999\ldots999}_{477\text{ korrekte Dez.-Ziffern}}834\ldots\cdot 10^{+307},$$
$$1.1235582092889\ldots576000\ldots000496\ldots\cdot 10^{+307}].$$

Die Berechnung einer Näherung für artanh $(2^{-2147482627})$ scheitert mit *Mathematica* an einer entsprechenden Overflow-Meldung.

Im **3. Beispiel** liefert das folgende Programm für das im Binärsystem nicht darstellbare Argument $x = 0.51$ eine garantierte Einschließung für artanh (0.51).

```
// Programm  lx_test59.cpp;
#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;

    X = lx_interval(0,"[0.51,0.51]");
    Y = atanh(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific << "atanh(X) = " << Y << endl;
}
```

Das obige Programm liefert mit Y die folgende garantierte Einschließung

$$\text{artanh}\,(0.51) \in \mathsf{Y} = 10^{-1} \cdot \underbrace{5.6272976935214885\ldots97088180874251963}_{302\text{ korrekte Dez.-Ziffern}}1\,{}^{785\ldots}_{332\ldots} \cdot 10^{0}$$

mit 302 korrekten Dezimalziffern.

5.2.37 Die Funktion artanh $(1-x)$

Für $x \in (0, +2)$ ist der Graph der Funktion artanh $(1-x)$ dargestellt in Abb. 14.

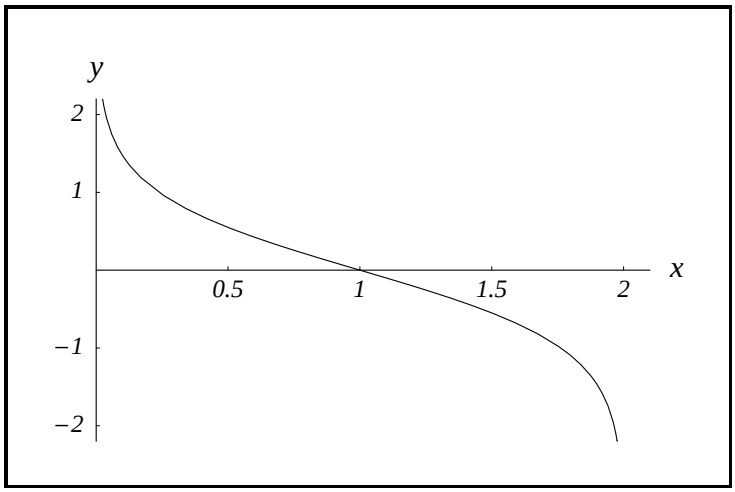


Abbildung 14: $y =$ artanh $(1-x)$

artanh (x) ist dabei die im letzten Abschnitt behandelte Umkehrfunktion der hyperbolischen Tangens-Funktion. Die C-XSC Funktion

```
lx_interval atanh1m(const lx_interval &u);
```

liefert Einschließungen für das reelle Intervall $\boldsymbol{y} =$ artanh $(1-\boldsymbol{u})$, $\boldsymbol{u} = [u_1, u_2]$, mit $0 < u_1 \leq u_2 < +2$, wobei das Intervall $\boldsymbol{u}$ bei einer sinnvollen Anwendung der Staggered Intervall-Arithmetik nicht zu breit gewählt werden darf. Wegen $2^{-2147482627} \leq u_1$ kann mit Hilfe dieser Funktion die inverse hyperbolische Tangens-Funktion ganz in der Nähe ihrer Singularität $+1$ ausgewertet werden.

Mit Hilfe der Transformation $x \to 1-x$ erhält man aus (104) direkt den Ausdruck

$$\text{artanh}\,(1-x) = \frac{1}{2} \cdot \ln\left(-1 + \frac{2}{x}\right), \quad 0 < x < 2, \tag{106}$$

der für $x \to 0$ intervallmäßig sehr effektiv ausgewertet werden kann.

Im **1. Beispiel** liefert das nachfolgende Programm `lx_test60.cpp` für das Intervall-Argument $x = 2^{-2147482627}$ eine sehr enge Einschließung für $f(x) =$ artanh $(1-x)$.

```
// Programm  lx_test60.cpp;
// Zum Test von atanh(1-x);

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;
```

```
int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;

    X = lx_interval(-2147482627,l_interval(1));
    Y = atanh1m(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    s << Y;
    cout << "atanh(1-X) = " << s << endl;
}
```

Die folgende Ausgabe von Y über die Zeichenkette s liefert mit dem Faktor 2^{-992} die Einschließung von $f(x) = \text{artanh}\,(1-x)$ mit einer intern berechneten Genauigkeit von 481 korrekten Dezimalstellen:

$$\text{artanh}\,(1-x) \in \mathsf{Y} = 2^{-992} \cdot \underbrace{3.1151633402305661\ldots4593969135356039}_{481 \text{ korrekte Dez.-Ziffern}}{}^{870\ldots}_{365\ldots} \cdot 10^{+307}$$

Damit erhalten wir für den Funktionswert $\text{artanh}\,(1-2^{-2147482627})$ eine garantierte Einschließung mit einer internen Genauigkeit von 481 Dezimalziffern. Die Berechnung einer Näherung für $\text{artanh}\,(1-2^{-2147482627})$ scheitert mit dem Algebra-System *Mathematica* an einer entsprechenden Overflow-Meldung.

Hinweis:

Die intervallmäßige Auswertung der rechten Seite von (106) ist besonders günstig, da die Variable x nur einmal vorkommt und für die Berechnung des Arguments der $\ln$-Funktion nur zwei Rechenoperationen erforderlich sind.

5.2.38 Die Funktion artanh $(-1+x)$

Für $x \in (0, +2)$ ist der Graph der Funktion artanh $(-1+x)$ dargestellt in Abb. 15.

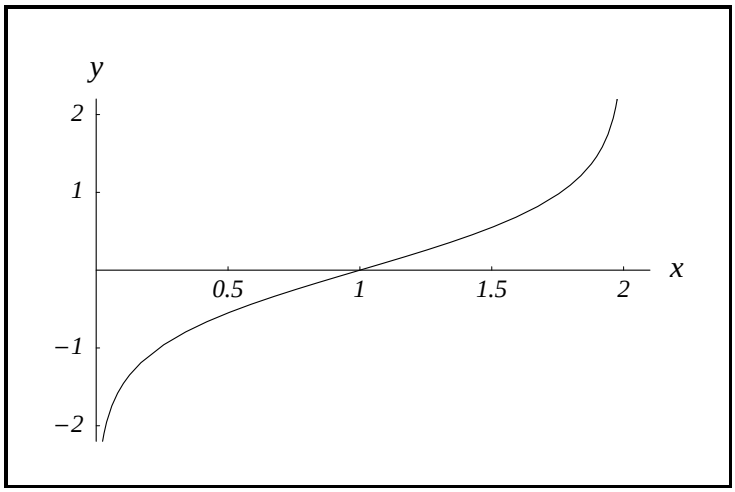


Abbildung 15: $y =$ artanh $(-1+x)$

artanh (x) ist dabei die im vorletzten Abschnitt behandelte Umkehrfunktion der hyperbolischen Tangens-Funktion. Die C-XSC Funktion

```
lx_interval atanhm1p(const lx_interval &u);
```

liefert Einschließungen für das reelle Intervall $\boldsymbol{y} =$ artanh $(-1+\boldsymbol{u})$, $\boldsymbol{u} = [u_1, u_2]$, mit $0 < u_1 \leq u_2 < +2$, wobei das Intervall $\boldsymbol{u}$ bei einer sinnvollen Anwendung der Staggered Intervall-Arithmetik nicht zu breit gewählt werden darf. Wegen $2^{-2147482627} \leq u_1$ kann mit Hilfe dieser Funktion die inverse hyperbolische Tangens-Funktion ganz in der Nähe ihrer Singularität -1 ausgewertet werden.

Da die artanh-Funktion punktsymmetrisch zum Ursprung ist, gilt

$$\text{artanh}\,(-1+x) \equiv -\text{artanh}\,(+1-x), \quad 0 < x < 2,$$

so dass die Funktion `atanhm1p(...)` direkt mit Hilfe der schon im letzten Abschnitt implementierten Funktion `atanh1m(...)` realisiert werden kann.

5.2.39 Die Funktion arcoth (x)

Für $x \notin [-1, +1]$ ist der Graph der reellen Funktion arcoth (x) dargestellt in Abb. 16.

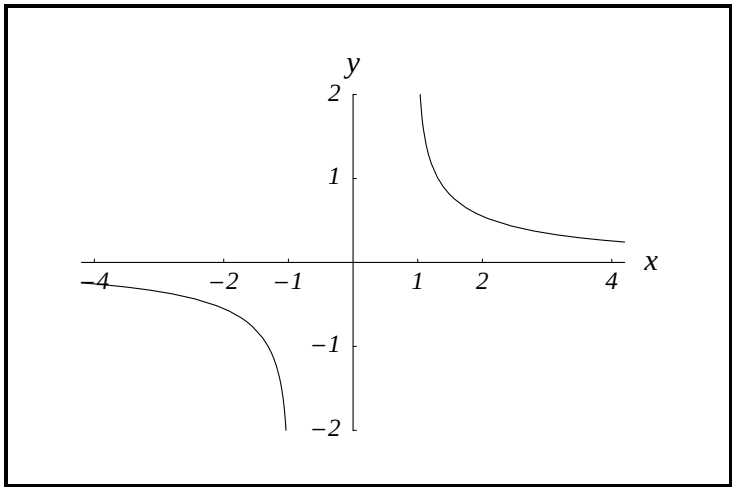


Abbildung 16: $y =$ arcoth (x)

arcoth (x) ist die Umkehrfunktion der hyperbolischen Cotangens-Funktion. Die C-XSC Funktion

```
lx_interval acoth(const lx_interval &u);
```

liefert Einschließungen für das reelle Intervall $\boldsymbol{y} =$ arcoth $(\boldsymbol{u})$, $\boldsymbol{u} = [u_1, u_2]$, wobei weder u_1 noch u_2 in $[-1, +1]$ enthalten sein darf. Bei einer sinnvollen Anwendung der Staggered Intervall-Arithmetik darf das Intervall $\boldsymbol{u}$ nicht zu breit gewählt werden.

Für $|x| > 1$ gilt die folgende Darstellung

$$\text{arcoth}\,(x) = \text{artanh}\,(1/x) = \frac{1}{2} \cdot \ln\left(1 + \frac{2}{x-1}\right), \quad |x| > 1. \tag{107}$$

Wegen der zusätzlichen Division $1/x$ wird artanh $(1/x)$ nicht ausgewertet, und wegen der Punktsymmetrie arcoth $(-x) \equiv -$arcoth (x) kann man sich auf $x > 1$ beschränken. Das Argument $1 + 2/(x-1)$ rechts in (107) kann übrigens für $x = 1 + \varepsilon,\ \varepsilon > 0$ sehr viel genauer eingeschlossen werden als für $x = -1 - \varepsilon$, daher wird die rechte Seite in (107) nur für $x > 1$ ausgewertet.

Im **1. Beispiel** liefert das nachfolgende Programm `lx_test61.cpp` für das Intervall-Argument $x = 1 + 2^{-2095}$ eine sehr enge Einschließung für $f(x) =$ arcoth (x).

```
// Programm  lx_test61.cpp;
// Zum Test von acoth(x);

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
```

```
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;

    X = 1 + lx_interval(-2095,l_interval(1));
    Y = acoth(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    s << Y;
    cout << "acoth(X) = " << s << endl;
}
```

Die folgende Ausgabe von `Y` über die Zeichenkette `s` liefert mit dem Faktor 2^{-1012} die Einschließung von $f(x) = \text{arcoth}\,(x)$ mit einer intern berechneten Genauigkeit von 473 korrekten Dezimalstellen:

$$\text{arcoth}\,(1+2^{-2095}) \in \mathsf{Y} = 2^{-1012} \cdot \underbrace{3.1881764953197\ldots08193890254331}_{473\ \text{korrekte Dez.-Ziffern}}{}^{200\ldots}_{195\ldots} \cdot 10^{+307}$$

Damit erhalten wir für den Funktionswert $\text{arcoth}\,(1+2^{-2095})$ eine garantierte Einschließung mit einer internen Genauigkeit von 473 Dezimalziffern. Bei der intervallmäßigen Auswertung des etwas kleineren Arguments $x = 1 + 2^{-2096} \in \boldsymbol{u} := 1 \diamondsuit [2^{-2096}, 2^{-2096}]$ wird die Singularität $x_0 = 1$ mit eingeschlossen, so dass beim Aufruf von $\text{arcoth}\,(\boldsymbol{u})$ eine entsprechende Fehlermeldung erfolgt. Wenn $\text{arcoth}\,(x)$ für Argumente x einzuschließen ist, die noch dichter bei $x_0 = +1$ liegen, so muss die C-XSC Funktion `acothp1(...)` benutzt werden, vgl. dazu den nächsten Abschnitt.

Im **2. Beispiel** liefert das nachfolgende Programm `lx_test62.cpp` für das maximale Argument $x = 2^{+2147482627}$ eine sehr enge Einschließung für $f(x) = \text{arcoth}\,(x)$.

```
// Programm  lx_test62.cpp;
// Zum Test von acoth(x);

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;
```

```
    X = lx_interval(+2147482627,l_interval(1));
    Y = acoth(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    s << Y;
    cout << "acoth(X) = " << s << endl;
}
```

Die folgende Ausgabe von Y über die Zeichenkette s liefert mit dem Faktor $2^{-2147483647}$ die Einschließung von $f(x) = \operatorname{arcoth}(x)$ mit einer intern berechneten Genauigkeit von 477 korrekten Dezimalstellen:

$$\operatorname{arcoth}(x) \in \mathsf{Y} = 2^{-2147483647} \cdot [\underbrace{1.1235582092889474\ldots575999\ldots999}_{477 \text{ korrekte Dez.-Ziffern}}834\ldots \cdot 10^{+307},$$
$$1.1235582092889474\ldots576000\ldots000496\ldots \cdot 10^{+307}].$$

Die Berechnung einer Näherung für $\operatorname{arcoth}(2^{+2147482627})$ scheitert mit *Mathematica* an einer Overflow-Meldung.

Im **3. Beispiel** liefert das nachfolgende Programm `lx_test63.cpp` für das nicht darstellbare Argument $x = -3.01$ eine garantierte Einschließung für $\operatorname{arcoth}(-3.01)$.

```
// Programm  lx_test63.cpp;
// Zum Test von acoth(x);

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y;

    X = lx_interval(0,"[-3.01,-3.01]");
    Y = acoth(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "acoth(X) = " << Y << endl;
}
```

Das obige Programm liefert mit Y die folgende garantierte Einschließung

$$\operatorname{arcoth}(-3.01) \in \mathsf{Y} = 10^{-1} \cdot \underbrace{-3.4532825962374671739\ldots979645203397590}_{302 \text{ korrekte Dez.-Ziffern}}{}^{484\ldots}_{813\ldots} \cdot 10^{0}$$

mit 302 korrekten Dezimalziffern. Eine mit *Mathematica* auf 310 Dezimalstellen berechnete Näherung für $\operatorname{arcoth}(-3.01)$ ist in Y enthalten.

5.2.40 Die Funktion arcoth $(1 + x)$

Für $x > 0$ ist der Graph der Funktion arcoth $(1 + x)$ dargestellt in Abb. 17.

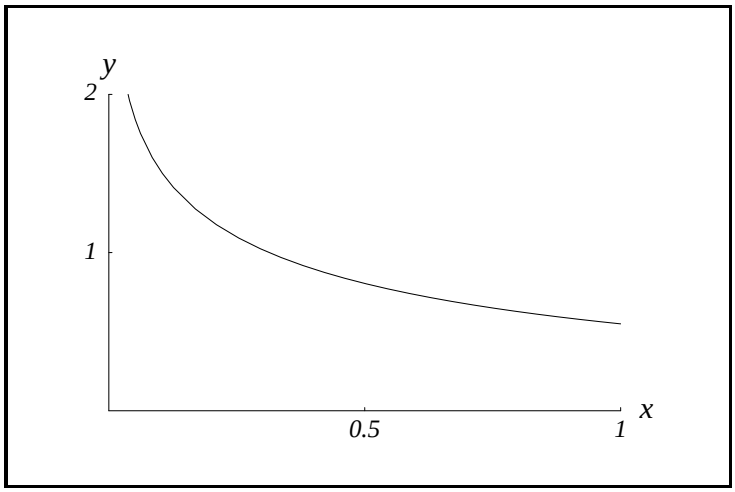


Abbildung 17: $y = \text{arcoth}\,(1 + x)$

arcoth (x) ist dabei die im letzten Abschnitt behandelte Umkehrfunktion der hyperbolischen Cotangens-Funktion. Die C-XSC Funktion

```
lx_interval acothp1(const lx_interval &u);
```

liefert Einschließungen für das reelle Intervall $\boldsymbol{y} = \text{arcoth}\,(1 + \boldsymbol{u})$, $\boldsymbol{u} = [u_1, u_2]$, mit $0 < u_1 \leq u_2$, wobei das Intervall $\boldsymbol{u}$ bei einer sinnvollen Anwendung der Staggered Intervall-Arithmetik nicht zu breit gewählt werden darf. Wegen $2^{-2147483645} \leq u_1$ kann mit dieser Funktion die inverse hyperbolische Cotangens-Funktion in der unmittelbaren Nähe ihrer Singularität $x_0 = +1$ ausgewertet werden.

Mit Hilfe der Transformation $x \to 1 + x$ erhält man aus (107) direkt den Ausdruck

$$\text{arcoth}\,(1 + x) = \frac{1}{2} \cdot \ln\left(1 + \frac{2}{x}\right), \quad x > 0, \tag{108}$$

der für $x \to 0$ intervallmäßig sehr effektiv mit Hilfe der C-XSC Funktion `lnp1(...)` ausgewertet werden kann.

Im **1. Beispiel** liefert das nachfolgende Programm `lx_test64.cpp` für das minimale Intervall-Argument $x = 2^{-2147483645}$ eine enge Einschließung für $f(x) = \text{arcoth}\,(1 + x)$.

```
// Programm  lx_test64.cpp;
// Zum Test von acoth(x);

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;
```

```
int main()
{
    stagprec = 39;
    lx_interval X,Y;
    string s;

    X = lx_interval(-2147483645,l_interval(1));
    Y = acothp1(X);
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    s << Y;
    cout << "acothp1(X) = " << s << endl;
}
```

Die folgende Ausgabe von Y über die Zeichenkette s liefert mit dem Faktor 2^{-992} die Einschließung von $f(x) = \operatorname{arcoth}(1+x)$ mit einer intern berechneten Genauigkeit von 481 korrekten Dezimalstellen:

$$\operatorname{arcoth}(1+x) \in \mathsf{Y} = 2^{-992} \cdot \underbrace{3.1151648169532361\ldots5965825903019065}_{481 \text{ korrekte Dez.-Ziffern}} {}^{544\ldots}_{039\ldots} \cdot 10^{+307}$$

Damit erhalten wir für den Funktionswert $\operatorname{arcoth}(1+2^{-2147483645})$ eine garantierte Einschließung mit einer internen Genauigkeit von 481 Dezimalziffern. Die Berechnung einer Näherung für $\operatorname{arcoth}(1+2^{-2147483645})$ scheitert mit dem Algebra-System *Mathematica* an einer entsprechenden Overflow-Meldung.

5.2.41 Die Funktion arcoth $(-1-x)$

Für $x > 0$ ist der Graph der Funktion arcoth $(-1-x)$ dargestellt in Abb. 18.

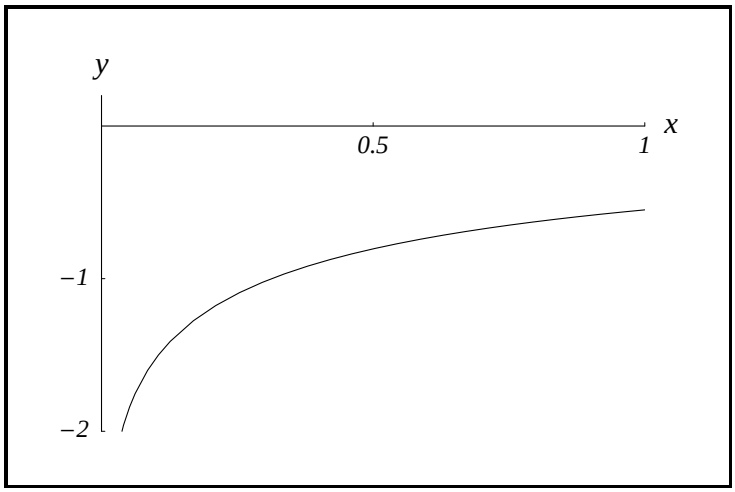


Abbildung 18: $y = \text{arcoth}\,(-1-x)$

arcoth (x) ist dabei die im vorletzten Abschnitt behandelte Umkehrfunktion der hyperbolischen Cotangens-Funktion. Die C-XSC Funktion

```
lx_interval acothm1m(const lx_interval &u);
```

liefert Einschließungen für das reelle Intervall $\boldsymbol{y} = \text{arcoth}\,(-1-\boldsymbol{u})$, $\boldsymbol{u} = [u_1, u_2]$, mit $0 < u_1 \leq u_2$, wobei das Intervall $\boldsymbol{u}$ bei einer sinnvollen Anwendung der Staggered Intervall-Arithmetik nicht zu breit gewählt werden darf. Wegen $2^{-2147483645} \leq u_1$ kann mit Hilfe dieser Funktion die inverse hyperbolische Cotangens-Funktion unmittelbar in der Nähe ihrer Singularität -1 ausgewertet werden.

Da die arcoth-Funktion punktsymmetrisch zum Ursprung ist, gilt

$$\text{arcoth}\,(-1-x) \equiv -\text{arcoth}\,(1+x), \quad x > 0,$$

so dass die Funktion `acothm1m(...)` direkt mit Hilfe der schon im letzten Abschnitt implementierten Funktion `acothp1(...)` realisiert werden kann.

5.2.42 Die Funktion $\sqrt{x^2+y^2}$

Für $x \in [-2,+2]$ und $y \in [-2,+2]$ ist der Graph der Funktion $f(x,y) = \sqrt{x^2+y^2}$ dargestellt in Abb. 19.

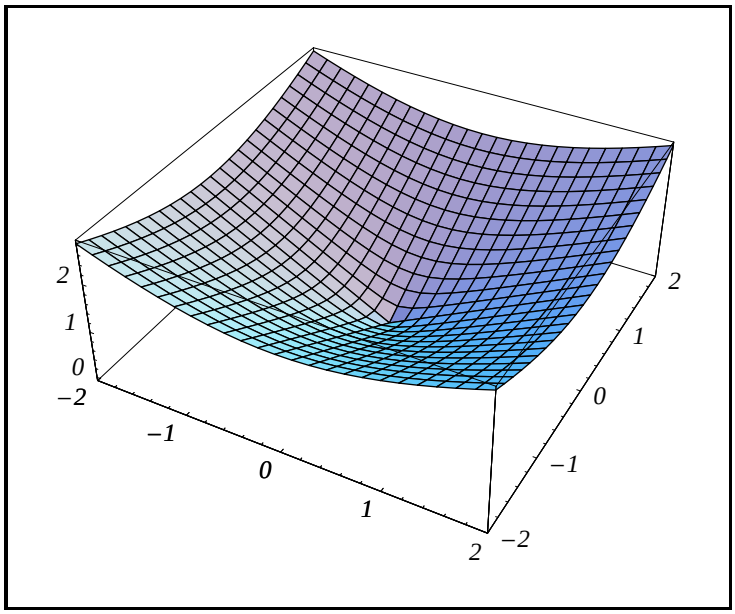


Abbildung 19: $f(x,y) = \sqrt{x^2+y^2}$

Die C-XSC Funktion

```
lx_interval sqrtx2y2(const lx_interval& x, const lx_interval& y);
```

liefert garantierte Einschließungen für das reelle Intervall $\boldsymbol{z} = \sqrt{\boldsymbol{x}^2+\boldsymbol{y}^2}$, $\boldsymbol{x} = [x_1,x_2]$, $\boldsymbol{y} = [y_1,y_2]$, mit $x_i, y_i \in \mathbb{R}$, wobei die Intervalle $\boldsymbol{x},\boldsymbol{y}$ bei einer sinnvollen Anwendung der Staggered Intervall-Arithmetik nicht zu breit gewählt werden sollten.

Bei der Implementierung wird zunächst vorausgesetzt, dass $\boldsymbol{x},\boldsymbol{y}$ die Intervalle der jeweiligen Beträge $|x|,|y|$ sind, d.h. es gilt $x_1, y_1 \geq 0$. Im Fall $y_2 = 0$ erhält man $\boldsymbol{z} = \boldsymbol{x}$ und im Fall $x_2 = 0$ erhält man entsprechend $\boldsymbol{z} = \boldsymbol{y}$. Im folgenden kann also $x_2 > 0$ und $y_2 > 0$ vorausgesetzt werden. O.E.d.A. wird zusätzlich $x_2 \geq y_2$ vorausgesetzt, und für die Implementierung benutzt man die folgenden Darstellungen:

$$\boldsymbol{z} = \sqrt{\boldsymbol{x}^2+\boldsymbol{y}^2}, \qquad \text{falls } 0 \in \boldsymbol{x}, \tag{109}$$

$$= \boldsymbol{x} \cdot \sqrt{1+\left(\frac{\boldsymbol{y}}{\boldsymbol{x}}\right)^2}, \quad \text{falls } 0 \notin \boldsymbol{x}. \tag{110}$$

Da in (109) die Null in $\boldsymbol{x}$ enthalten ist und vorher $x_2 > 0$ vorausgesetzt wurde, ist zumindest $\boldsymbol{x} = [0,x_2]$ ein sehr breites Eingangsintervall, so dass nach dem Quadrieren beim Radizieren sehr häufig eine Fehlermeldung nicht zu vermeiden ist.

In (110) ist rechts der Quadratwurzelausdruck intervallmäßig mit Hilfe der C-XSC Funktion `sqrt1px2(...)` mit dem Argument $\boldsymbol{y} \diamond \boldsymbol{x}$ auszuwerten. Wegen

$$\frac{\boldsymbol{y}}{\boldsymbol{x}} = \frac{[y_1,y_2]}{[x_1,x_2]} = \left[\frac{y_1}{x_2}, \frac{y_2}{x_1}\right] \tag{111}$$

kann bei der Auswertung von y_2/x_1 ein integer-Überlauf entstehen, wenn $y_2 \to \infty$ und $x_1 \to 0$. In diesem Fall wird dann aber wegen $y_2 \leq x_2$ das Intervall $\boldsymbol{x} = [x_1, x_2]$ sehr breit sein müssen, so dass eine sinnvolle Anwendung der Staggered-Intervallarithmetik nicht mehr möglich ist. Dieser integer-Überlauf muss daher nicht weiter diskutiert werden.

Bei der Auswertung von y_1/x_2 und y_2/x_1 in (111) kann es jedoch auch bei sehr engen Intervallen $\boldsymbol{x}, \boldsymbol{y}$ im Fall $y_2 \to 0$ und $x_1 \to \infty$ zu einem integer-Überlauf kommen, der nur verhindert werden kann, wenn man die Intervalldivision $\boldsymbol{y} \diamond \boldsymbol{x}$ vermeidet und durch die Differenz entsprechender Zweier-Exponenten ersetzt.

Nach (111) gilt zunächst

$$\sqrt{1 + \left(\frac{\boldsymbol{y}}{\boldsymbol{x}}\right)^2} \subseteq \left[1, \sqrt{1 + \left(\frac{y_2}{x_1}\right)^2}\right]. \tag{112}$$

Im nächsten Schritt verlangen wir, dass das Intervall rechts in (112) enthalten sein soll im Maschinenintervall $\boldsymbol{\alpha} := [1, 1 + 2^{-2097}]$, wobei $\boldsymbol{\alpha}$ mit Hilfe der Intervallkonstanten[24] `One_p_lx_interval()` einfach zu realisieren ist, d.h. es soll gelten:

$$\left[1, \sqrt{1 + \left(\frac{y_2}{x_1}\right)^2}\right] \subseteq [1, 1 + 2^{-2097}]. \tag{113}$$

Wir verlangen daher

$$\sqrt{1 + \left(\frac{y_2}{x_1}\right)^2} \leq 1 + 2^{-2097} \quad \Longleftrightarrow \quad \left(\frac{y_2}{x_1}\right)^2 \leq 2^{-2096} + 2^{-4194},$$

und die letzte Ungleichung ist sicher erfüllt, wenn gilt

$$\left(\frac{y_2}{x_1}\right)^2 \leq 2^{-2096} \quad \Longleftrightarrow \quad \frac{y_2}{x_1} \leq 2^{-1048}. \tag{114}$$

Wenn die letzte Ungleichung in (114) erfüllt ist, dann gilt folglich

$$\sqrt{1 + \left(\frac{\boldsymbol{y}}{\boldsymbol{x}}\right)^2} \subset [1, 1 + 2^{-2097}].$$

Im nächsten Schritt wird jetzt y_2/x_1 mit den Zweier-Exponenten von Zähler und Nenner abgeschätzt, wobei y_2 und x_1 Werte vom Typ `lx_real` sind.

Zunächst wird eine Oberschranke für $y_2 > 0$ berechnet[25]. Bezeichnet man die erste und damit auch größte Komponente von y_2 mit $y_2[1]$, so gilt mit

$$\texttt{exy = expo(y2); exyl = expo\_gr(lr\_part(y2))};$$
$$y_2[1] = 2^{exy} \cdot m \cdot 2^{exyl} < 2^{exy+exyl}, \quad 0.5 \leq m < 1, \quad \rightsquigarrow$$
$$y_2 < 2 \cdot 2^{exy+exyl} = 2^{exy+exyl+1}. \tag{115}$$

[24] Das C-XSC Punktintervall `One_p_lx_interval()` schließt den Wert $1 + 2^{-2097}$ ein.

[25] Der Fall $y_2 = 0$ liefert direkt $\boldsymbol{z} = \boldsymbol{x}$.

Der zusätzliche Faktor 2 in (115) berücksichtigt, dass alle nachfolgenden Komponenten $y_2[j],\ j = 2, 3, \ldots$ sehr viel kleiner sind und dabei auch alle positiv ausfallen können.

Jetzt wird noch eine Unterschranke für $x_1 > 0$ ganz analog berechnet. Bezeichnet man die erste und damit auch größte Komponente von x_1 mit $x_1[1]$, so gilt mit

$$\texttt{exx = expo(x1); exxl = expo\_gr(lr\_part(x1));}$$
$$x_1[1] = 2^{exx} \cdot m \cdot 2^{exxl} \geq 2^{exx+exxl-1}, \quad 0.5 \leq m < 1, \quad \rightsquigarrow$$
$$x_1 > 0.5 \cdot 2^{exx+exxl-1} = 2^{exx+exxl-2}. \tag{116}$$

Der zusätzliche Faktor 0.5 in (116) berücksichtigt, dass alle nachfolgenden Komponenten $x_1[j],\ j = 2, 3, 4, \ldots$ sehr viel kleiner sind und dabei auch alle negativ ausfallen können. Aus (115) und (115) ergibt sich damit die folgende Abschätzung

$$\frac{y_2}{x_1} < 2^{exy-exx+exyl-exxl+3}.$$

Nach (114) ist damit die Einschließung

$$\sqrt{1 + \left(\frac{\boldsymbol{y}}{\boldsymbol{x}}\right)^2} \subset [1, 1 + 2^{-2097}] \tag{117}$$

garantiert, wenn die folgende Bedingung erfüllt ist:

$$2^{exy-exx+exyl-exxl+3} \leq 2^{-1048} \quad \Longleftrightarrow \quad exy - exx + exyl - exxl \leq 1051$$
$$\Longleftrightarrow \quad exy \leq exx + \{(exxl - exyl) - 1051\}. \tag{118}$$

Bei der letzten Ungleichung wird $\{\ldots\}$ ohne integer-Überlauf und die ganze rechte Seite nur in wirklichen Extremfällen mit einem integer-Überlauf berechnet, was gegebenenfalls einen Programmabbruch zur Folge hat.

Wenn jedoch die Bedingung in (118) nicht erfüllt ist, erfolgt die Auswertung nach (110), da dann $\boldsymbol{y} \diamondsuit \boldsymbol{x}$ stets ohne einen integer-Überlauf berechnet werden kann.

Damit kann jetzt nach (109) bzw. (110) das Intervall $\boldsymbol{z} = \sqrt{\boldsymbol{x}^2 + \boldsymbol{y}^2}$ für hinreichend enge Eingangsintervalle $\boldsymbol{x}, \boldsymbol{y}$ in einem sehr umfangreichen Bereich der (x, y)-Ebene in hoher Genauigkeit eingeschlossen werden. Die relativen Durchmesser von $\boldsymbol{x}, \boldsymbol{y}$ sollten dabei die Bedingung (4) auf Seite 7 erfüllen. Die Beispiele auf der folgenden Seite zeigen z.B., dass für die Punktintervalle $\boldsymbol{x} = 2^{+2147482625}$ und $\boldsymbol{y} = 2^{-2147482625}$ eine garantierte Einschließung für $\boldsymbol{z} = \sqrt{\boldsymbol{x}^2 + \boldsymbol{y}^2}$ berechnet werden kann.

Mit *Mathematica* erhält man eine Overflow-Meldung, wenn man für $x = 2^{+2147482625}$ und $y = 2^{-2147482625}$ eine Näherung für $\sqrt{x^2 + y^2}$ berechnen will.

Im **1. Beispiel** liefert das nachfolgende Programm `lx_test65.cpp` für die Argumente $x = 2^{+2147482625}$ und $y = 2^{-2147482625}$ eine enge Einschließung für den Funktionswert $f(x,y) = \sqrt{x^2+y^2} \approx 2^{+2147482625}$.

```
// Programm  lx_test65.cpp;
// Zum Test von sqrt(x^2+y^2);

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 39;
    lx_interval X,Y,Z;
    string s;

    X = lx_interval(+2147482625,l_interval(1));
    Y = lx_interval(-2147482625,l_interval(1));
    Z = sqrtx2y2(X,Y);
    cout << SetDotPrecision(16*stagprec,16*stagprec+10)
         << Scientific;
    cout << "Z = " << Z << endl;
    s << Z;
    cout << "Z = " << s << endl;
}
```

Die erste Ausgabe von `Z` erfolgt in lesbarer dezimaler Form und liefert mit `Z` eine Einschließung für $\sqrt{x^2+y^2}$ mit 302 korrekten Dezimalstellen.

$$\sqrt{x^2+y^2} \in \mathtt{Z} = 10^{+646456684} \cdot \underbrace{1.95985957505354115\ldots939491187747718}_{302 \text{ korrekte Dez.-Ziffern}} \begin{smallmatrix}938\ldots\\828\ldots\end{smallmatrix} \cdot 10^{1}.$$

Die zweite Ausgabe über die Zeichenkette `s` liefert mit `Z` ebenfalls eine Einschließung für $\sqrt{x^2+y^2}$, wobei man jetzt die interne Genauigkeit der Rechnung mit 631 korrekten Dezimalstellen direkt ablesen kann.

$$\sqrt{x^2+y^2} \in \mathtt{Z} = 2^{+2147481605} \cdot \underbrace{1.1235582092889\ldots514008576000\ldots000}_{631 \text{ korrekte Dez.-Ziffern}} \begin{smallmatrix}494\ldots\\000\ldots\end{smallmatrix} \cdot 10^{307}.$$

Zeigen Sie mit dem gleichen Programm, dass für $x = y = 2^{+2147482625}$ und für $x = y = 2^{-2147482598}$ die Funktionswerte $f(x,y)$ in hoher Genauigkeit eingeschlossen werden.

Versucht man für die obigen Beispiele mit dem Algebra-System *Mathematica* Näherungen der Funktionswerte $f(x,y)$ zu berechnen, so erhält man stets eine entsprechende Overflow-Meldung.

Im **2. Beispiel** wählen wir $x = 0.6,\ y = 0.8$ und benutzen die Beziehung $x^2 + y^2 = 1$. Um dies auf dem Rechner zu bestätigen, muss man jedoch beachten, dass die Dezimalwerte 0.6 und 0.8 auch im Staggered Format nicht exakt darstellbar sind und daher durch möglichst enge Intervalle einzuschließen sind. Mit dem folgenden Programm

```
// Programm  lx_test66.cpp;
// Zum Test von sqrt(x^2+y^2);

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 30;
    lx_interval X,Y,Z;

    X = lx_interval(0,"[0.6,0.6]");
    Y = lx_interval(0,"[0.8,0.8]");
    Z = sqrtx2y2(X,Y);
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "Z = " << Z << endl;
}
```

erhält man die sehr enge Einschließung der Eins mit 302 korrekten Dezimalstellen.

$$\coth(x) \in \mathtt{Z} = 10^{-1} \cdot [\underbrace{9.999999\ldots9999999}_{302\ \text{korrekte Dez.-Ziffern}}021\ldots, 1.000000\ldots00000102\ldots \cdot 10^{1}].$$

Zu beachten ist, dass mit den Zeichenketten `"[0.6,0.6]"` und `"[0.8,0.8]"` die Intervalle `X` und `Y` die nicht darstellbaren Dezimalzahlen 0.6 und 0.8 wirklich einschließen, weshalb diese Intervalle auch keine Punktintervalle sein können.

5.2.43 Die Funktion $\ln(\sqrt{x^2+y^2})$

Für $x \in [0.1, +1]$ und $y \in [0.1, +1]$ ist der Graph der Funktion $f(x,y) = \ln(\sqrt{x^2+y^2})$ dargestellt in Abb. 20.

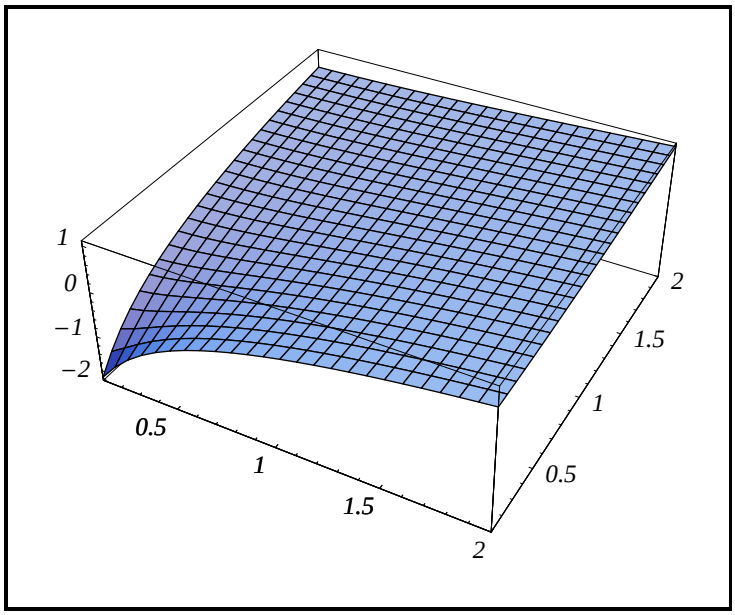


Abbildung 20: $f(x,y) = \ln(\sqrt{x^2+y^2})$

Die C-XSC Funktion

```
lx_interval ln_sqrtx2y2(const lx_interval &x, const lx_interval &y);
```

liefert Einschließungen für das reelle Intervall $\boldsymbol{z} = \ln(\sqrt{\boldsymbol{x}^2+\boldsymbol{y}^2})$, $\boldsymbol{x} = [x_1, x_2]$, $\boldsymbol{y} = [y_1, y_2]$, mit $x_i, y_i \in \mathbb{R}$, wobei die Intervalle $\boldsymbol{x}, \boldsymbol{y}$ bei einer sinnvollen Anwendung der Staggered Intervall-Arithmetik nicht zu breit gewählt werden sollten.

Bei der Implementierung der obigen C-XSC Funktion wird zunächst vorausgesetzt, dass $\boldsymbol{x}, \boldsymbol{y}$ die Intervalle der jeweiligen Beträge $|x|, |y|$ sind, d.h. es gilt $x_1, y_1 \geq 0$, mit $|x| + |y| > 0$, und wegen der Quadrate kann man sich auf den 1. Quadranten der im Ursprung gelochten (x,y)-Ebene beschränken. Zusätzlich kann o.E.d.A. $x_2 \geq y_2$ vorausgesetzt werden, wobei folgende Darstellungen benutzt werden:

$$f(x,y) = \ln(\sqrt{x^2+y^2}), \tag{119}$$

$$= \ln(1 + \{(x-1)\cdot(x+1) + y^2\}); \tag{120}$$

Im Fall $x_2 > 100$ oder $x_2 < 0.25$ kommt (119) zur Anwendung, wobei die auftretende Quadratwurzel direkt mit der bereits implementierten `sqrtx2y2()`-Funktion intervallmäßig ausgewertet wird. Daher braucht man sich um mögliche integer-Überläufe bei den Auftretenden Quadraten nicht zu kümmern. Dieser Vorteil wird allerdings mit einer etwas größeren Laufzeit erkauft, da in (119) zwei Funktionen auszuwerten sind.

Im verbleibenden Fall können die Intervalle $\boldsymbol{x}, \boldsymbol{y}$ in der unmittelbaren Nähe des Einheitskreises liegen, so dass nach (119) bei der Auswertung der $\ln$-Funktion in der Nähe

der Nullstelle 1 starke Auslöschungen auftreten können. Diese lassen sich vermeiden, wenn bei der intervallmäßigen Auswertung von (120) die C-XSC Funktion `lnp1(...)` mit dem Argument $\boldsymbol{\alpha} := \{(\boldsymbol{x}-1)\cdot(\boldsymbol{x}+1)+\boldsymbol{y}^2\}$ zur Anwendung kommt.

Bei der Auswertung von $\boldsymbol{y}^2$ kann es jedoch bei zu kleinem $y_1 > 0$ zu einem integer-Überlauf kommen. Im Fall $\boldsymbol{x} = 1$ ist dies wegen $\ln(\sqrt{\boldsymbol{x}^2+\boldsymbol{y}^2}) = 0.5\cdot\ln(1+\boldsymbol{y}^2) \approx 0.5\cdot\boldsymbol{y}^2$ kein Nachteil. Aber im Fall $\boldsymbol{x} \neq 1$ nimmt das Argument $\boldsymbol{\alpha}$ auch dann noch endliche Werte an, wenn $\boldsymbol{y}^2$ im Unterlaufbereich liegt und dadurch das Programm abgebrochen wird. Um dies zu vermeiden, wird die folgende Skalierung durchgeführt

$$\begin{aligned} \boldsymbol{\alpha} &:= \{(\boldsymbol{x}-1)\cdot(\boldsymbol{x}+1)+\boldsymbol{y}^2\} \\ &\equiv \frac{[2^N\cdot(\boldsymbol{x}-1)]\cdot[2^N\cdot(\boldsymbol{x}+1)]+(2^N\cdot\boldsymbol{y})\cdot(2^N\cdot\boldsymbol{y})}{2^{2N}}, \quad N\in\mathbb{N}, \end{aligned} \tag{121}$$

wobei $N > 0$ so zu wählen ist, dass $(2^N\cdot\boldsymbol{y})\cdot(2^N\cdot\boldsymbol{y})$ gerade noch keinen integer-Überlauf erzeugt. Damit können wir jetzt in sehr großen Bereichen der (x,y)-Ebene die Intervalle $f(\boldsymbol{x},\boldsymbol{y})$ in hoher Genauigkeit einschließen.

Im **1. Beispiel** liefert das nachfolgende Programm `lx_test67.cpp` für die Argumente $x = 2^{+2147482625}$ und $y = 2^{+2147482625}$ eine enge Einschließung für den Funktionswert $f(x,y) = \ln(\sqrt{x^2+y^2})$.

```
// Programm  lx_test67.cpp;

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;
int main()
{
    stagprec = 30;
    lx_interval X,Y,Z;
    string s;
    X = lx_interval(+2147482625,l_interval(1));
    Y = lx_interval(+2147482625,l_interval(1));
    Z = ln_sqrtx2y2(X,Y);
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    s << Z;
    cout << "Z = " << s << endl;
}
```

Die folgende Ausgabe von `Z` über die Zeichenkette `s` liefert mit dem Faktor 2^{-991} die Einschließung von $f(x,y)$ mit einer intern berechneten Genauigkeit von 481 korrekten Dezimalstellen:

$$\ln(\sqrt{x^2+y^2}) \in \mathsf{Z} = 2^{-991}\cdot\underbrace{3.115163336604036\ldots32630875184551366}_{481\text{ korrekte Dez.-Ziffern}}{}^{458\ldots}_{003\ldots}\cdot 10^{+307}.$$

Zeigen Sie mit dem gleichen Programm, dass mit den Argumenten

$$x = 2^{+2147482625}, y = 2^{-2147483647} \quad \text{und mit} \quad x = 2^{-2147482627}, y = 2^{-2147482598}$$

ebenfalls sehr enge Einschließungen der Funktionswerte $f(x, y)$ berechnet werden. Das Algebra-System *Mathematica* liefert entsprechende Overflow-Meldungen, wenn Näherungen zu den $f(x, y)$ zu berechnen sind.

Im **2. Beispiel** liefert das folgende Programm `lx_test68.cpp` für die Argumente $x = 1$ und den minimalen Wert $y = 2^{-1073741313}$ eine enge Einschließung für den Funktionswert $f(x, y) = \ln(\sqrt{1 + 2^{-2147482626}})$.

```
// Programm  lx_test68.cpp;
// Zum Test von ln( sqrt(x^2+y^2) );

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 30;
    lx_interval X,Y,Z;
    string s;

    X = lx_interval(0,l_interval(1));
    Y = lx_interval(-1073741313,l_interval(1));
    Z = ln_sqrtx2y2(X,Y);
    cout << SetDotPrecision(16*stagprec,16*stagprec+7)
         << Scientific;
    s << Z;
    cout << "Z = " << s << endl;
}
```

Die folgende Ausgabe von `Z` über die Zeichenkette `s` liefert mit dem Faktor $2^{-2147483648}$ die Einschließung von $\ln(\sqrt{1 + 2^{-2147482626}})$ mit einer internen Genauigkeit von 470 korrekten Dezimalstellen:

$$f(x, y) \in \mathtt{Z} = 2^{-2147483648} \cdot [\underbrace{2.2471164185778\ldots151999\ldots999}_{470 \text{ korrekte Dez.-Ziffern}}583\ldots \cdot 10^{+307},$$
$$2.2471164185778\ldots152000\ldots000394\ldots \cdot 10^{+307}].$$

Die Berechnung einer Näherung für $\ln(\sqrt{1 + 2^{-2147482626}}) = 0.5 \cdot \ln(1 + 2^{-2147482626})$ scheitert mit *Mathematica* jeweils an einer Overflow-Meldung.

Im **3. Beispiel** liefert das folgende Programm `lx_test69.cpp` für die Argumente $x = 1$ und $y = 3.1 \cdot 10^{-20000}$ eine garantierte Einschließung für den Funktionswert $f(x,y) = \ln(\sqrt{1 + (3.1 \cdot 10^{-20000})^2})$.

```
// Programm  lx_test69.cpp;
// Zum Test von ln( sqrt(x^2+y^2) );

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 30;
    lx_interval X,Y,Z;

    X = lx_interval(0,l_interval(1));
    Y = lx_interval(-20000,"[3.1,3.1]");
    Z = ln_sqrtx2y2(X,Y);
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "Z = " << Z << endl;
}
```

Die Ausgabe von `Z` erfolgt jetzt in lesbarer dezimaler Form und liefert mit dem Faktor 10^{-40000} die Einschließung von $f(x,y) = \ln(\sqrt{1 + (3.1 \cdot 10^{-20000})^2})$ mit einer internen Genauigkeit von 300 korrekten Dezimalstellen:

$$f(x,y) \in \mathtt{Z} = 10^{-40000} \cdot [\underbrace{4.8049999\ldots999}_{\text{300 korrekte Dez.-Ziffern}}582\ldots, 4.805000\ldots000417\ldots].$$

Beachten Sie, dass $y = 3.1 \cdot 10^{-20000}$ nicht im Binärsystem darstellbar ist und daher durch

```
Y = lx_interval(-20000,"[3.1,3.1]");
```

eingeschlossen werden muss, wobei `Y` $\ni y$ kein Punktintervall sein kann.

Im **4. Beispiel** wird ein Punkt $P_1(x_1, y_1)$ mit den Koordinaten x_1, y_1 möglichst dicht am Einheitskreis gewählt. Danach soll bewiesen werden, dass dieser Punkt P_1 jedoch nicht auf dem Einheitskreis liegt. Die Koordinaten x_1, y_1 sind dabei vom Typ `lx_real` zu wählen. Ausgangspunkt ist die bekannte Beziehung $0.6^2 + 0.8^2 = 1$. Im ersten Schritt wird im folgenden Programm `lx_test70.cpp` die nicht darstellbare Dezimalzahl 0.6 durch `Incl_6_10 = lx_interval(0,l_interval(6))/10;` mit der Präzision `stagprec = 29` in hoher Genauigkeit eingeschlossen. Ganz analog wird auch 0.8 durch das Intervall `Incl_8_10` eingeschlossen. Die Koordinate x_1 vom Typ `lx_real` wird dann durch `Inf(Incl_6_10)` bestimmt und durch das Punktintervall `x1 = lx_interval(Inf(Incl_6_10))` eingeschlossen. Ganz entsprechend wird

dann y_1 durch das Punktintervall `y1 = lx_interval(Sup(Incl_8_10));` eingeschlossen. Es gilt daher $x_1 \approx 0.6$ und $y_1 \approx 0.8$, und das folgende Programm soll zeigen, dass $P_1(x_1, y_1)$ nicht auf dem Einheitskreis liegt.

```
// Programm  lx_test70.cpp;
// Zum Test von ln( sqrt(x^2+y^2) );

#include <iostream>
#include "lx_imath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 29;
    lx_interval Incl_6_10,x1,Incl_8_10,y1,Z;

    Incl_6_10 = lx_interval(0,l_interval(6))/10;
    x1 = lx_interval(Inf(Incl_6_10));
    if (point_intv(x1))
       cout << "x1 ist ein Punktintervall" << endl;
    Incl_8_10 = lx_interval(0,l_interval(8))/10;
    y1 = lx_interval(Sup(Incl_8_10));
    if (point_intv(y1))
       cout << "y1 ist ein Punktintervall" << endl;
    stagprec = 30;
    Z = ln_sqrtx2y2(x1,y1);
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    cout << "Z = " << Z << endl;
}
```

Das Programm liefert die Ausgabe

```
x1 ist ein Punktintervall
y1 ist ein Punktintervall
```

$$\texttt{Z} = -10^{-472} \cdot 3.0911264920^{116...}_{422...}$$

Die Null ist also in der Einschließung `Z` des Funktionswertes $\ln(\sqrt{x_1^2 + y_1^2})$ nicht enthalten, so dass wegen $x_1^2 + y_1^2 \neq 1$ der Punkt $P_1(x_1, y_1)$ sicher nicht auf dem Einheitskreis liegen kann. Überlegen Sie, warum der obige Nachweis nicht mehr geführt werden kann, wenn die Punktintervalle `x1,y1` mit der höheren Präzision `stagprec` $= 30$ bestimmt werden. Natürlich lässt sich der obige Beweis auch ohne die $\ln$-Funktion führen, wenn $x_1^2 + y_1^2 \neq 1$ direkt bestätigt wird.

6 Erweiterte komplexe Staggered Intervall-Arithmetik

Es wird eine neue Klasse `lx_cinterval` mit Objekten der Form

$$(re, im) = (2^{ex_r} \cdot li_r, 2^{ex_i} \cdot li_i) = re + i \cdot im, \quad i := \sqrt{-1}$$

implementiert. In dieser Klasse werden neben den Konstruktoren die Intervall-Operatoren $\{\diamondsuit, \diamondsuit, \diamondsuit, \diamondsuit\}$ bereitgestellt, mit denen auf der Maschine die vier Grundoperationen in optimaler Genauigkeit durchgeführt werden können. Es stehen die gängigsten Standardfunktionen zum Datentyp `lx_cinterval` zur Verfügung, und einfache C-XSC Programme demonstrieren die Leistungsfähigkeit der erweiterten Klasse `lx_cinterval`.

6.1 Die Klasse `lx_cinterval`

Die Klasse `lx_interval` ist u.a. wie folgt implementiert:

```
#include <iostream>
#include <string>
#include <except.hpp>
#include <l_cinterval.hpp>
#include "lx_interval.hpp"
#include "lx_imath.hpp"
#include "lx_complex.hpp"

namespace cxsc {
class lx_cinterval
{
private:
   // ------------- private data elements ---------------------------
   lx_interval re, im;
   // (re,im) is a complex number:  re + i*im, i = sqrt(-1).
public:
   // ------------- Constructors ------------------------------------
   inline lx_cinterval(void)  throw() { }
   inline lx_cinterval(const lx_interval &,const lx_interval&) throw();
   inline lx_cinterval(const l_interval &, const l_interval &) throw();
   inline lx_cinterval(const interval &,   const interval &)   throw();
   inline lx_cinterval(const l_real &,     const l_real &)     throw();
   inline lx_cinterval(const lx_real &,    const lx_real &)    throw();
   inline lx_cinterval(const real &,       const real &)       throw();
   inline lx_cinterval(const l_cinterval &) throw();
   inline lx_cinterval(const cinterval &) throw();
   inline lx_cinterval(const complex &) throw();
   inline lx_cinterval(const l_complex &) throw();
   inline lx_cinterval(const lx_complex &) throw();
   inline lx_cinterval(const lx_complex&, const lx_complex&)
       throw(ERROR_CINTERVAlx_EMPTY_INTERVAL);
   inline lx_cinterval(const l_complex&, const l_complex&)
       throw(ERROR_CINTERVAlx_EMPTY_INTERVAL);
   inline lx_cinterval(const complex&, const complex&)
       throw(ERROR_CINTERVAlx_EMPTY_INTERVAL);
   inline lx_cinterval(int, const l_interval&, int, const l_interval&)
       throw();
```

```
   inline lx_cinterval(int, const l_interval&) throw();
   inline lx_cinterval(int, const l_interval&,const lx_interval&)
       throw();
   inline lx_cinterval(const lx_interval&, int, const l_interval&)
       throw();
   inline lx_cinterval(int, const string&, int, const string&) throw();
   explicit inline lx_cinterval(const lx_interval &) throw();
   explicit inline lx_cinterval(const l_interval &) throw();
   inline lx_cinterval(const interval &) throw();
   explicit inline lx_cinterval(const lx_real &) throw();
   explicit inline lx_cinterval(const l_real &) throw();
   explicit inline lx_cinterval(const real &) throw();
/*  ----------- and other member funktions of this class ----------- */
}; // class lx_cinterval                        }  // end namespace cxsc
```

Analog zur Klasse `lx_real`, vgl. Seite 13, liefert die Klasse `lx_complex` Objekte der Struktur $(re, im) = re + i \cdot im, i = \sqrt{-1}$, wobei die privaten Datenelemente re, im beide vom Typ `lx_real` sind. Die Klasse `lx_complex` ermöglicht u.a. die Implementierung der Funktionen `Inf`$(...)$, `Sup`$(...)$, `mid`$(...)$ und `diam`$(...)$ für Objekte der Klasse `lx_cinterval`. Auf die Klasse `lx_complex` wird hier nicht weiter eingegangen.

6.1.1 Konstruktoren

Die verschiedenen Konstruktoren der Klasse `lx_cinterval` ermöglichen ein einfaches Einbinden von Objekten der Klassen `lx_real`, `lx_complex`, ... Alle Konstruktoren sind ab Seite 155 zusammengestellt. Wichtig für die Anwendung ist der Konstruktor

```
lx_cinterval(int pr, const string &sr, int pi, const string &si);
```

mit dem über die beiden Zeichenketten `sr,si` für Real- und Imaginärteil Objekte der Klasse `lx_interval` in dezimaler Form generiert werden können. `pr,pi` bedeuten dabei die entsprechenden Zehnerexponenten. Mit `sr = "[2.1e-1,2.1e-1]"` und `pr = -700` generiert der Konstruktor z.B. für den Realteil ein Objekt a vom Typ `lx_interval`, mit: $10^{-700} \cdot [2.1 \cdot 10^{-1}, 2.1 \cdot 10^{-1}] \subseteq a$. Zu beachten ist, dass wegen notwendiger Umrechnungen die Einschließung durch a leicht überschätzt wird, wobei insbesondere für `stagprec` > 19 nur ca. 300 korrekte Dezimalziffern geliefert werden. Ebenfalls wichtig für die Anwendung ist der Konstruktor

```
lx_cinterval(int p,const l_interval&a,int q,const l_interval&b);
```

wobei `p,q` jetzt die Exponenten zur Basis 2 sind. Mit den Anweisungen

```
string("[0.1,0.1]") >> a;      string("[0.3,0.3]") >> a;
```

und `p=q=-8000;` liefert dann der Aufruf `z = lx_interval(p,a,q,b);` mit `z`
eine garantierte Einschließung des komplexen Intervalls

$$2^{-8000} \cdot [0.1, 0.1] + i \cdot 2^{-8000} \cdot [0.3, 0.3] \subseteq \mathtt{z}, \quad i = \sqrt{-1},$$

wobei diese Einschließung durch `z` bei jeder durch `stagprec` vorgegebenen Präzision nahezu optimal erfolgt, weil jetzt Umrechnungen der Zehnerpotenzen $10^{\mathtt{pr}}$ oder $10^{\mathtt{pi}}$ wie beim ersten Konstruktor-Beispiel nicht erforderlich sind.

6.1.2 Zuweisungsoperatoren

In der Klasse `lx_cinterval` sind die folgenden Zuweisungsoperatoren definiert:

```
lx_cinterval & operator = (const lx_cinterval &a) throw( );
lx_cinterval & operator = (const l_cinterval &a) throw( );
lx_cinterval & operator = (const cinterval   &a) throw( );
lx_cinterval & operator = (const lx_interval  &a) throw( );
lx_cinterval & operator = (const l_interval  &a) throw( );
lx_cinterval & operator = (const interval    &a) throw( );
lx_cinterval & operator = (const lx_real      &a) throw( );
lx_cinterval & operator = (const l_real      &a) throw( );
lx_cinterval & operator = (const real        &a) throw( );
lx_cinterval & operator = (const lx_complex   &a) throw( );
lx_cinterval & operator = (const l_complex   &a) throw( );
lx_cinterval & operator = (const complex     &a) throw( );
```

Der für die Anwendung wichtige Operator

```
l_cinterval & operator = (const lx_cinterval& a) throw();
```

der in `l_cinterval.hpp` deklariert und in `lx_cinterval.cpp` implementiert ist, liefert eine optimale Einschließung von `a` durch ein Intervall `x` vom Typ `l_cinterval`. Falls dies wegen eines Overflows nicht möglich ist, erfolgt eine entsprechende Fehlermeldung. Ganz analog wird der Zuweisungsoperator

```
cinterval & operator = (const lx_cinterval& a) throw();
```

in `cinterval.hpp` deklariert und in `lx_cinterval.cpp` implementiert.

6.1.3 Mengenvergleiche

In der Klasse `lx_cinterval` sind die folgenden Mengenvergleiche definiert:

```
<, <=, >, >=
```

wobei wenigstens ein Operand vom Typ `lx_cinterval` sein muss. Der Typ des zweiten Operanden muss ein Element der folgenden Menge sein:

```
{ lx_cinterval, l_cinterval, cinterval, lx_interval, l_interval, interval,
          lx_real, l_real, real, lx_complex, l_complex, complex }
```

6.1.4 Vergleichsoperatoren

In der Klasse `lx_vinterval` sind die Operatoren $==$ und $!=$ definiert für Operanden vom Typ

```
lx_cinterval, l_cinterval, cinterval, lx_interval, l_interval, interval,
          lx_real, l_real, real, lx_complex, l_complex, complex
```

wobei wenigstens einer der Operanden vom Typ `lx_cinterval` sein muss.

6.1.5 Arithmetische Operatoren

In der C-XSC Klasse `lx_cinterval` sind neben den beiden unitären Operatoren $+, -$ mit nur jeweils einem Operanden vom Typ `lx_cinterval` zusätzlich die arithmetischen Operatoren $\{+, -, *, /\}$ definiert, wobei wenigstens einer der beiden Operanden vom Typ `lx_cinterval` sein muss. Die Zuweisungsoperatoren $\{+=, -=, *=, /=\}$ sind ebenfalls implementiert.

6.1.6 Der Negationsoperator !

In der Klasse `lx_cinterval` ist der Operator

```
bool operator ! (const lx_cinterval& a);
```

definiert. Er liefert 1, wenn $0 \in$ `a.re` $\wedge\, 0 \in$ `a.im`, sonst wird 0 zurückgegeben.

6.1.7 Ein- und Ausgabefunktionen

Zur Eingabe von Objekten des Typs `lx_cinterval` stehen die folgenden Operatoren zur Verfügung:

```
string & operator >> (std::string &s, lx_cinterval &a);
void operator >> (const std::string &s, lx_cinterval &a);
void operator >> (const char *s, lx_cinterval &a);
istream & operator >> (std::istream &s, lx_cinterval &a);
```

Mit den drei ersten Operatoren wird ein String `s` der Form

```
( {-345678, [0.1e-4,0.1e-4]} , {-345678, [-0.3e-4,-0.3e-4]} )
```

in eine Variable vom Typ `lx_cinterval` kopiert, wobei -345678 als Exponent zur Basis **10** interpretiert wird. Der vierte Operator kopiert die Tastatureingabe mit dem gleichen Format ebenfalls in eine Variable `a` vom Typ `lx_cinterval`. Die Variablen `a` sind dabei stets Einschließungen der eingegebenen String-Intervalle.

Zur Erzeugung von Objekten des Typs `lx_ciinterval` können innerhalb eines Programms auch die auf Seite 156 beschriebenen Konstruktoren verwendet werden.

Zur Ausgabe von Objekten des Typs `lx_cinterval` stehen folgende Operatoren zur Verfügung:

```
ostream& operator << (ostream& s, const lx_cinterval& a);
string & operator << (string &s,  const lx_cinterval& a);
```

Der erste ermöglicht die Ausgabe von `a` in dezimaler Form in den Ausgabekanal und der zweite schreibt ein Objekt `a` vom Typ `lx_cinterval` mit den beiden geklammerten Zweier-Exponenten in eine Zeichenkette `s`.

6.1.8 Weitere Funktionen und Operatoren

Innerhalb und außerhalb der Klasse `lx_cinteval` sind die folgende Funktionen implementiert:

1. `lx_interval abs(const lx_cinterval &a);`
   liefert eine Einschließung des Absolutbetrags des komplexen Intervalls `a`.

2. `lx_interval Re(const lx_cinterval &a);`
   liefert den Realteil des komplexen Intervalls `a`.

3. `lx_interval Im(const lx_cinterval &a);`
   liefert den Imaginärteil des komplexen Intervalls `a`.

4. `lx_complex Inf(const lx_cinterval &a);`
   liefert das komplexwertige Infimum des komplexen Intervalls `a`.

5. `lx_complex Sup(const lx_cinterval &a);`
   liefert das komplexwertige Supremum des komplexen Intervalls `a`.

6. ```
   lx_cinterval & SetRe(lx_cinterval &a,
                        const lx_interval &x);
   ```
   liefert das komplexe Intervall `a` mit dem neuen Realteil `x`. Der Typ des Realteils `x` kann ein Element der folgenden Menge sein:
   `{lx_interval,l_interval,interval,lx_real,l_real,real}`.

7. ```
   lx_cinterval & SetIm(lx_cinterval &a,
                        const lx_interval &x);
   ```
   liefert das komplexe Intervall `a` mit dem neuen Imaginärteil `x`. Der Typ des Realteils `x` kann ein Element der folgenden Menge sein:
   `{lx_interval,l_interval,interval,lx_real,l_real,real}`.

8. `lx_real InfRe(const lx_cinterval &a);`
   liefert das reelle Infimum des Realteils des komplexen Intervalls `a`.

9. `lx_real InfIm(const lx_cinterval &a);`
   liefert das reelle Infimum des Imaginärteils des komplexen Intervalls `a`.

10. `lx_real SupRe(const lx_cinterval &a);`
    liefert das reelle Supremum des Realteils des komplexen Intervalls `a`.

11. `lx_real SupIm(const lx_cinterval &a);`
    liefert das reelle Supremum des Imaginärteils des komplexen Intervalls `a`.

12. `lx_complex mid(const lx_cinterval &a);`
    liefert eine Approximation der komplexen Mitte des komplexen Intervalls `a`.

13. `lx_complex diam(const lx_cinterval &a);`
    liefert den Durchmesser für Real- und Imaginärteil des komplexen Intervalls `a`.

14. `int expo_Re(const lx_cinterval &a);`
    liefert den Zweier-Exponenten des Realteils des komplexen Intervalls `a`.

15. `int expo_Im(const lx_cinterval &a);`
    liefert den Zweier-Exponenten des Imaginärteils des komplexen Intervalls `a`.

16. `l_interval li_part_Re(const lx_cinterval &a);`
    liefert von $\Re(\mathtt{a})$ den Intervallteil vom Typ `l_interval`.

17. `l_interval li_part_Im(const lx_cinterval &a);`
    liefert von $\Im(\mathtt{a})$ den Intervallteil vom Typ `l_interval`.

18. `lx_cinterval adjust(const lx_cinterval &a);`
    Anpassung des komplexen Intervalls `a` an die aktuelle Präzision `stagprec`.

19. `lx_cinterval conj(const lx_cinterval &a);`
    liefert das konjugiert komplexe Interval `a`.

20. `void times2pown(lx_cinterval &a , int n);`
    Das erste Argument liefert das komplexe Intervall: $2^n \cdot \mathtt{a}$.

21. `bool IsEmpty(const lx_cinterval &a);`
    liefert 1, wenn $\Re(\mathtt{a})$ und $\Im(\mathtt{a})$ leere Mengen sind, sonst wird 0 zurückgegeben.

22. `lx_cinterval operator | (const lx_cinterval &a,`
    `const lx_cinterval &b);`
    liefert die konvexe Hülle der beiden Operanden `a,b`. Die möglichen Typkombinationen der Operanden findet man in `lx_cinterval.hpp`.

23. `lx_cinterval operator & (const lx_cinterval &a,`
    `const lx_cinterval &b);`
    liefert den Durchschnitt der beiden Operanden `a,b`. Die möglichen Typkombinationen der Operanden findet man in `lx_cinterval.hpp`.

blabla

6.1.9 Standardfunktionen

Die Argumente z der folgenden Standardfunktionen sind nicht zu breite Intervalle vom Typ `lx_cinterval`. Beispiele findet man bei den Seitenangaben der letzten Spalte.

Standardfunktionen vom Typ `lx_cinterval`

Funktionsterm	C-XSC Name	Informationen	Seite
$\|z\| = \sqrt{(z.re)^2 + (z.im)^2}$	abs(z)	$\|z\| \subseteq$ abs(z)	
z^2	sqr(z)	Einschließung des Quadrats	
$\sqrt{z}$	sqrt(z)	Berechnung des Hauptzweigs	163
$\sqrt[n]{z}$	sqrt(z, n)	$2 \leq n \leq +2147483647$	163
arg(z)	Arg(z)	Analytische Argumentfunktion	164
arg(z)	arg(z)	Nicht analytische Funktion	165
$\ln(z)$	Ln(z)	Analytischer Logarithmus	166
$\ln(z)$	ln(z)	Nicht analytische Funktion	167
z^n	power_fast(z, n)	Schnelle Potenzfunktion	167
z^n	power(z, n)	Langsamere Potenzfunktion	42
z^p	pow(z, p)	lx_cinterval z; lx_interval p;	169
z^w	pow(z, w)	lx_cinterval z, w	169
e^z	exp(z)	lx_cinterval z	
$\sin(z)$	sin(z)	lx_cinterval z	
$\cos(z)$	cos(z)	lx_cinterval z	
$\tan(z)$	tan(z)	lx_cinterval z	
$\cot(z)$	cot(z)	lx_cinterval z	
$\arcsin(z)$	asin(z)	lx_cinterval z	172
$\arccos(z)$	acos(z)	lx_cinterval z	188
$\arctan(z)$	atan(z)	lx_cinterval z	200
arccot(z)	acot(z)	lx_cinterval z	
$\sinh(z)$	sinh(z)	lx_cinterval z	
$\cosh(z)$	cosh(z)	lx_cinterval z	
$\tanh(z)$	tanh(z)	lx_cinterval z	
$\coth(z)$	coth(z)	lx_cinterval z	

Standardfunktionen vom Typ `lx_cinterval`			
Funktionsterm	**C-XSC Name**	**Informationen**	**Seite**
arsinh (z)	asinh(z)	lx_cinterval z	
arcosh (z)	acosh(z)	lx_cinterval z	
artanh (z)	atanh(z)	lx_cinterval z	
arcoth (z)	acoth(z)	lx_cinterval z	

6.2 Algorithmen der komplexen Standardfunktionen

Die Einschließung komplexer Funktionswerte der Standardfunktionen mit rechteckigen Intervallargumenten $\boldsymbol{z} = \boldsymbol{x} + i \cdot \boldsymbol{y},\ \boldsymbol{x} = [x_1, x_2], \boldsymbol{y} = [y_1, y_2], \in I\!I\!R,\ x_j, y_j \in \mathbb{R},\ i = \sqrt{-1}$ ist für das einfache IEEE- und für das Staggered Correction-Format beschrieben in [7], [6], [8], [9], [16], [28]. Für das Staggered Correction-Format der Klasse `l_cinterval` findet man die Implementierung der komplexen Standardfunktionen in `l_cimath.cpp`.

Die komplexen Standardfunktionen der erweiterten C-XSC Klasse `lx_cinterval` sind ganz analog zu den entsprechenden Funktionen der C-XSC Klasse `l_cinterval` aufgebaut. In sehr einfachen Fällen, z.B. bei der Logarithmus-Funktion, mussten lediglich die Klassen-Namen `l_real, l_interval, l_cinterval` geändert werden in `lx_real, lx_interval, lx_cinterval`. Bei der Implementierung kamen außerdem die Standardfunktionen der erweiterten Klasse `lx_interval` zur Anwendung, die in den Abschnitten 5.2.7 bis 5.2.43 ausführlich beschrieben sind.

Bei den Standardfunktionen der Klasse `l_cinterval` mussten in verschiedenen Fällen geeignete Skalierungen vorgenommen werden, um in bestimmten Teilbereichen der komplexen Ebene eine hinreichende Genauigkeit zu erreichen. Solche Skalierungen sind bei Anwendung der Klasse `lx_cinterval` jedoch nicht mehr nötig, da diese bei den vier Grundoperationen bereits eingearbeitet wurden. Wenn sich die Algorithmen durch den Wegfall solcher Skalierungen wesentlich vereinfachen, werden die neuen Algorithmen gesondert angegeben. Dies geschieht auch dann, wenn die Algorithmen wegen des erweiterten Zahlenbereichs, z.B. zur Vermeidung eines vorzeitigen integer-Überlaufs, neu zu überarbeiten sind. So musste z.B. die Funktion $\arctan(y/x)$ im Abschnitt 6.2.11 für $y/x \to +\infty$ und $y/x \to 0$ grundlegend neu implementiert werden. Dadurch können jetzt Einschließungen dieser Funktionswerte für alle Punktintervalle $\boldsymbol{y}, \boldsymbol{x} \neq 0$ der Klasse `lx_interval` berechnet werden.

6.2.1 Die Funktion $\sqrt{z}$

Der Verzweigungsschnitt der komplexen Quadratwurzel ist wie üblich die negative reelle Achse. Ein rechteckiges, komplexes Eingabeintervall $\boldsymbol{z}$ darf diesen Verzeigungsschnitt nur von oben berühren aber nicht in seinem Innern enthalten. Erlaubte und nicht erlaublte Intervalle $\boldsymbol{z}$ sind in der folgenden Abbildung angegeben, wobei die nicht erlaubten Intervalle gestrichelt dargestellt sind. Es ist auch hier zu beachten, dass die erlaubten Intervalle $\boldsymbol{z}$ nicht zu breit gewählt werden sollten.

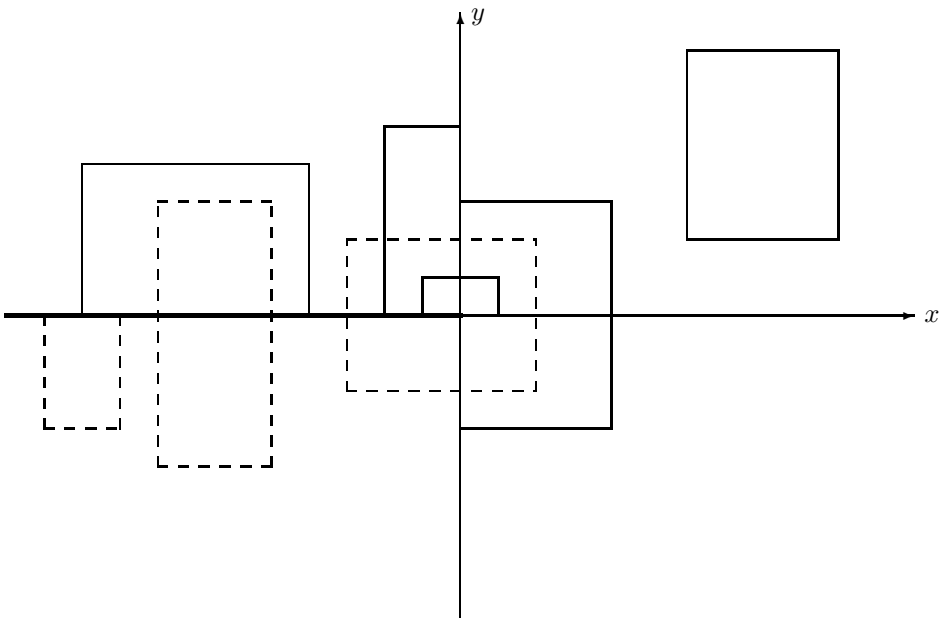


Abbildung 21: Erlaubte und nicht erlaubte Intervalle $\boldsymbol{z}$ für $\sqrt{\boldsymbol{z}}$

6.2.2 Die Funktion $\sqrt[n]{z}$

Der Verzweigungsschnitt bei der n-ten komplexen Wurzel ist wie bei der Quadratwurzel die negative reelle Achse. Ein rechteckiges, komplexes Eingabeintervall $\boldsymbol{z}$ darf keinen Punkt dieses Verzweigungsschnitts enthalten. Es gilt $2 \leq n \leq +2147483647$.

Als einfache Anwendung soll der Ausdruck

$$t := \sqrt[301]{\ln(2) + i \cdot \sqrt{2}}, \quad i := \sqrt{-1},$$

mit der Präzision `stagprec = 3` eingeschlossen werden. Mit den Anweisungen

```
lx_cinterval Z,T;   stagprec = 3;
Z = lx_cinterval( Ln2_lx_interval(), Sqrt2_lx_interval());
T = sqrt(Z,301);
cout << SetDotPrecision(16*stagprec,16*stagprec);
cout << "T = " << T << endl;
```

erhält man mit `T` eine Einschließung von $t \in$ `T` mit etwa 45 korrekten Dezimalstellen.

6.2.3 Die Funktion Arg(z)

Der Verzweigungsschnitt bei dieser Argumentfunktion ist die negative reelle Achse. Das rechteckige Eingangsintervall $\boldsymbol{z}$ darf keinen Punkt dieses Verzweigungsschnitts enthalten. Erlaubte und nicht erlaublte Intervalle $\boldsymbol{z}$ sind in der folgenden Abbildung angegeben, wobei nicht erlaubte Intervalle gestrichelt dargestellt sind.

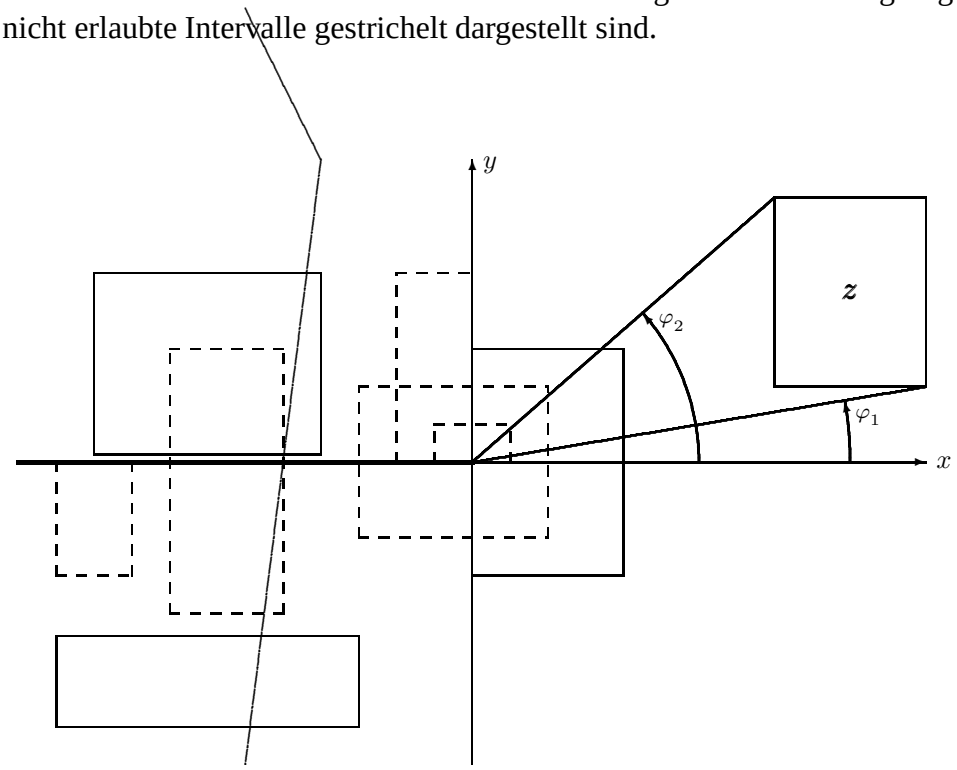


Abbildung 22: Erlaubte und nicht erlaubte Intervalle $\boldsymbol{z}$ für Arg$(\boldsymbol{z})$

Nach obiger Abbildung gilt zunächst:

1. $\texttt{Arg}(\boldsymbol{z}) := [\varphi_1, \varphi_2] \subset (-\pi, +\pi)$. Außerdem legt man fest:
2. $\texttt{Arg}([0 + 0 \cdot i) := [0, 0]$,
3. sonst wird $\texttt{Arg}(\boldsymbol{z})$ definiert durch die konvexe Hülle der folgenden Menge

$$\{\texttt{arg}(c) \,\big|\, c \in \mathbb{C} \wedge c \in \boldsymbol{z} \wedge c \neq 0\}, \quad \texttt{arg}(c) \in \mathbb{R}.$$

Danach gilt z.B.

$$\texttt{Arg}([0,0] + i \cdot [0,2]) = \texttt{Arg}([0,0] + i \cdot [1,2]) = [\pi/2],$$

wobei das Intervall $[\pi/2]$ zwar den Wert $\pi/2$ selbst, nicht aber die Null einschließt.

6.2.4 Die Funktion arg(z)

Der Verzweigungsschnitt bei dieser Argumentfunktion ist die negative reelle Achse. Das rechteckige Intervall $\boldsymbol{z}$ darf jetzt, im Gegensatz zur Arg(z)-Funktion, beliebige Punkte dieses Verzweigungsschnitts enthalten, so dass jedes Intervall $\boldsymbol{z} \in IC$ zulässig ist. In der folgenden Abbildung sind einige Eingangsintervalle $\boldsymbol{z}$ angegeben, wobei die Pfeile festlegen, in welcher Richtung die analytische Fortsetzung zum Verzweigungsschnitt vorzunehmen ist.

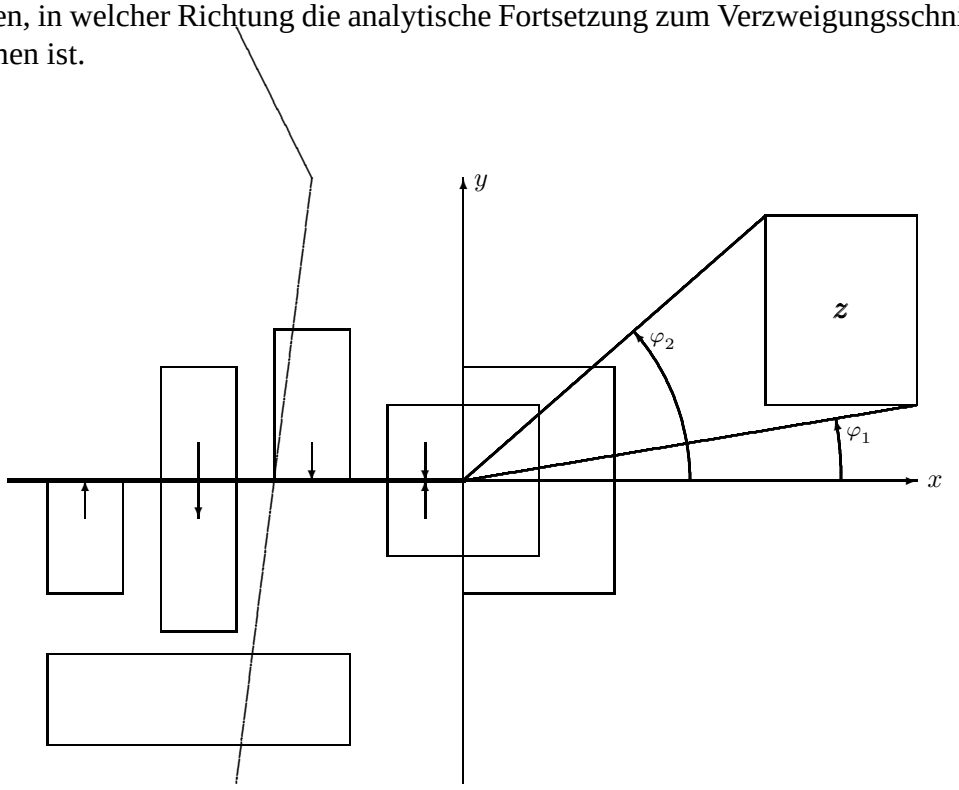


Abbildung 23: Eingangsintervalle $\boldsymbol{z}$ für arg$(\boldsymbol{z})$

Nach obiger Abbildung gilt zunächst:

1. $\mathtt{arg}(\boldsymbol{z}) := [\varphi_1, \varphi_2] \subset (-\pi, +3\pi/2)$. Außerdem legt man fest:
2. $\mathtt{arg}([0 + 0 \cdot i) := [0, 0]$,
3. $\mathtt{arg}(\boldsymbol{z}) = \mathtt{Arg}(\boldsymbol{z})$, falls $\mathtt{Arg}(\boldsymbol{z})$ existiert.

Danach gilt z.B.

$$\begin{aligned}
\mathtt{arg}([0,0] + i \cdot [0,2]) = \mathtt{arg}([0,0] + i \cdot [1,2]) &= [\pi/2, \pi/2], \\
\mathtt{arg}([-2,-1] + i \cdot [-1,0]) &= [-\pi, -3\pi/4], \\
\mathtt{arg}([-2,-1] + i \cdot [0,+1]) &= [3\pi/4, +\pi], \\
\mathtt{arg}([-2,-1] + i \cdot [-1,1]) &= [3\pi/4, 5\pi/4], \\
\mathtt{arg}([-1,0] + i \cdot [-1,+1]) &= [\pi/2, 3\pi/2], \\
\mathtt{arg}([-1,+1] + i \cdot [-1,+1]) &= [-\pi, +\pi].
\end{aligned}$$

6.2.5 Die Funktion Ln(z)

Der Verzweigungsschnitt der komplexwertigen Logarithmusfunktion ist wie üblich die negative reelle Achse. Ein rechteckiges, komplexes Eingangsintervall $\boldsymbol{z}$ darf den Ursprung nicht enthalten, und der Verzeigungsschnitt darf nur von oben her berührt werden. Erlaubte und nicht erlaublte Intervalle $\boldsymbol{z}$ sind in der folgenden Abbildung angegeben, wobei die nicht erlaubten Intervalle gestrichelt dargestellt sind. Es ist auch hier zu beachten, dass die erlaubten Intervalle $\boldsymbol{z}$ nicht zu breit gewählt werden sollten.

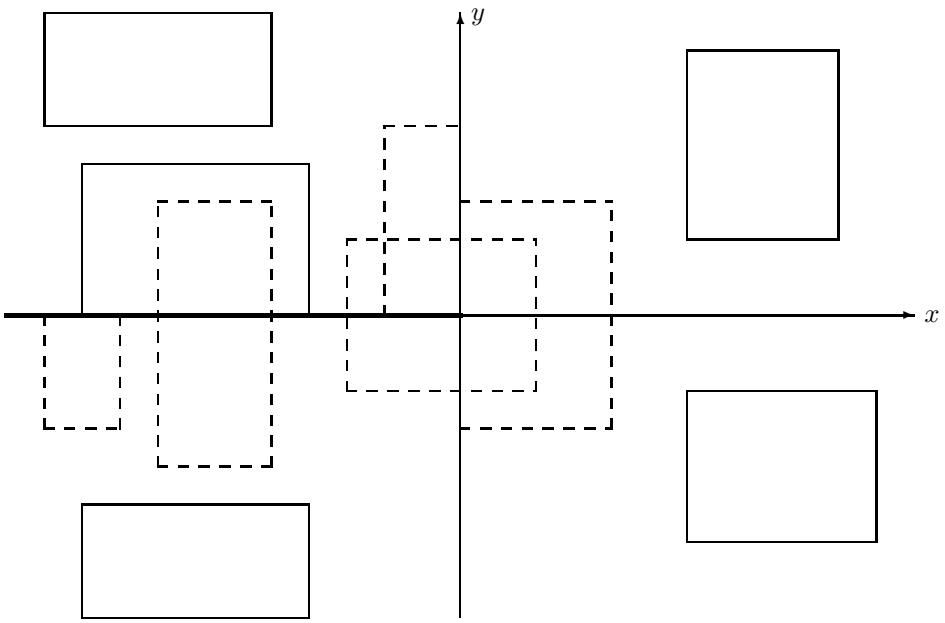


Abbildung 24: Erlaubte und nicht erlaubte Intervalle $\boldsymbol{z}$ für Ln$(\boldsymbol{z})$

Die Anweisungen

```
stagprec = 3;
lx_cinterval Z,W;
Z = lx_cinterval( 0,l_interval(-1,-1),
                  -1073741313,l_interval(0,1) );
W = Ln(Z);
cout << SetDotPrecision(16*stagprec,16*stagprec+3)
     << Scientific;
s << W;
cout << "Ln(Z) = " << s << endl;
```

liefern eine garantierte und enge Einschließung für: $\ln(-1 + i \cdot [0, 2^{-1073741313}])$. Mit dem Imaginärteil $[0, 2^{-1073741314}]$ erhält man wegen eines integer-Überlaufs eine Fehlermeldung.

6.2.6 Die Funktion ln(z)

Der Verzweigungsschnitt dieser komplexwertigen Logarithmusfunktion ist ebenfalls die negative reelle Achse. Das komplexe Eingangsintervall $\boldsymbol{z}$ kann jetzt aber beliebig gewählt werden, es darf lediglich den Ursprung nicht enthalten. Für Intervalle $\boldsymbol{z}$, mit denen Ln$(\boldsymbol{z})$ berechnet werden kann, gilt:

$$\ln(\boldsymbol{z}) \equiv \mathrm{Ln}(\boldsymbol{z}), \quad 0 \notin \boldsymbol{z}.$$

Erlaubte und nicht erlaublte Intervalle $\boldsymbol{z}$ sind in der folgenden Abbildung angegeben, wobei die nicht erlaubten Intervalle gestrichelt dargestellt sind. Es ist auch hier zu beachten, dass die erlaubten Intervalle $\boldsymbol{z}$ nicht zu breit gewählt werden sollten.

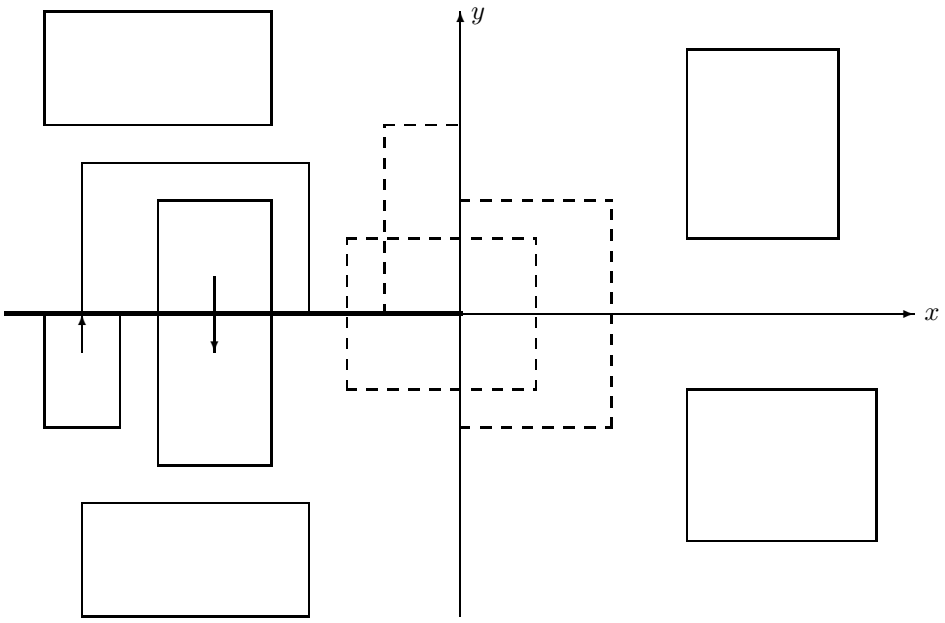


Abbildung 25: Erlaubte und nicht erlaubte Intervalle $\boldsymbol{z}$ für ln$(\boldsymbol{z})$

Die Anweisungen

```
stagprec = 3;
lx_cinterval Z,W;
Z = lx_cinterval( 0,l_interval(-1,-1),
                  -1073741313,l_interval(-1,1) );
W = ln(Z);
cout << SetDotPrecision(16*stagprec,16*stagprec+3)
     << Scientific;
s << W;
cout << "ln(Z) = " << s << endl;
```

liefern eine garantierte und enge Einschließung für: $\ln(-1 + i \cdot 2^{-1073741313} \cdot [-1, +1])$.

6.2.7 Die Funktion power_fast(z, n)

Die Funktion power_fast$(\boldsymbol{z}, n)$ schließt die komplexen Intervallpotenzen $\boldsymbol{z}^n, n \in \mathbb{Z}$ ein, mit

$$\boldsymbol{z}^n := \begin{cases} 0, & \boldsymbol{z} = 0 \wedge n > 0, \\ 1, & \boldsymbol{z} = 0 \wedge n = 0, \\ \infty, & \boldsymbol{z} = 0 \wedge n < 0, \\ \infty, & \boldsymbol{z} \neq 0 \wedge 0 \in \boldsymbol{z}, \text{ Abb. 25}, \\ e^{n \cdot \ln(\boldsymbol{z})}, & \boldsymbol{z} \neq 0 \wedge 0 \notin \boldsymbol{z}, \text{ Abb. 25}. \end{cases}$$

$\ln(\boldsymbol{z})$ ist dabei die Logarithmus-Funktion von Seite 167, wobei der Verzweigungsschnitt wieder die negative reele Achse ist. Für den Fall $\boldsymbol{z} \neq 0$ findet man erlaubte und nicht erlaubte Intervalle $\boldsymbol{z}$ in Abbildung 25 auf Seite 167. $\boldsymbol{z}^n = \infty$ bedeutet Programmabbruch mit einer entsprechenden Fehlermeldung.

Es ist auch hier wieder zu beachten, dass die Intervalle $\boldsymbol{z}$ nicht zu breit gewählt werden sollten, da man sonst, insbesondere für $|n| \gg 1$, größere Überschätzungen erhält.

6.2.8 Die Funktion power(z, n)

Die Funktion power$(\boldsymbol{z}, n)$ schließt die komplexen Intervallpotenzen $\boldsymbol{z}^n, n \in \mathbb{Z}$ ein, mit

$$\boldsymbol{z}^n := \begin{cases} \infty, & 0 \in \boldsymbol{z} \wedge n < 0, \\ 1, & n = 0, \\ w := \boldsymbol{z} \cdot \boldsymbol{z} \cdot \ldots \cdot \boldsymbol{z}, & n > 0, \\ 1/w, & n < 0. \end{cases}$$

$\boldsymbol{z}^n$ ist also nur dann nicht definiert, wenn $0 \in \boldsymbol{z} \wedge n < 0$. Der Algorithmus von Neher [27] bringt für die Einschließung von $\Re(\boldsymbol{z}^n)$ und $\Im(\boldsymbol{z}^n)$ die entsprechenden Extremalkurven zum Schnitt mit der Berandung von $\boldsymbol{z}$. Dadurch wird zunächst eine Überschätzung der Einschließungen minimiert. Es ist jedoch zu beachten, dass bei zu breiten Intervallen $\boldsymbol{z}$ und $|n| \gg 1$ die Intervalle $\boldsymbol{z}^n$ selbst große Durchmesser besitzen und daher mit Hilfe einer Staggered Intervall-Arithmetik nur mit geringer Genauigkeit eingeschlossen werden können. Zudem wächst mit zunehmendem $|n|$ die Laufzeit erheblich, so dass die Funktion power_fast(z, n) fast immer vorzuziehen ist. Mit den Anweisungen

```
stagprec = 30;
lx_cinterval Z,W;
cout << SetDotPrecision(16*stagprec,16*stagprec+3)
     << Scientific;
Z = lx_cinterval(Ln2_lx_interval(),Sqrt2_lx_interval());
W = power_fast(Z,1234567);
cout << "power_fast = " << W << endl;
```

erhält man für $(\ln(2) + i \cdot \sqrt{2})^{1234567}$ eine garantierte Einschließung bez. Real- und Imaginärteil mit mindestens 301 korrekten Dezimalstellen. Etwa die gleiche Genauigkeit liefert die deutlich langsamere Funktion power$(Z, 1234567)$.

6.2.9 Die Funktion pow(z, p)

Die Funktion pow$(\boldsymbol{z}, p)$ schließt die komplexen Intervallpotenzen $\boldsymbol{z}^p$ mit Intervallen vom Typ `lx_cinterval` ein, wobei p der reelle Intervallexponent vom Typ `lx_interval` ist und $\boldsymbol{z}^p$ definiert wird durch

$$\boldsymbol{z}^p := e^{p \cdot \mathsf{Ln}(\boldsymbol{z})}.$$

$\mathsf{Ln}(\boldsymbol{z})$ ist dabei die auf Seite 166 definierte analytische Logarithmusfunktion, so dass erlaubte und nicht erlaubte Eingangsintervalle $\boldsymbol{z}$ vom Typ `lx_cinterval` der Abbildung 24 auf Seite 166 entnommen werden können.

6.2.10 Die Funktion pow(z, w)

Die Funktion pow$(\boldsymbol{z}, w)$ schließt komplexwertige Intervallpotenzen $\boldsymbol{z}^w$ mit Intervallen vom Typ `lx_cinterval` ein, wobei w der komplexe Intervallexponent vom Typ `lx_cinterval` ist und $\boldsymbol{z}^p$ definiert wird durch

$$\boldsymbol{z}^w := e^{w \cdot \mathsf{Ln}(\boldsymbol{z})}.$$

$\mathsf{Ln}(\boldsymbol{z})$ ist dabei die auf Seite 166 definierte analytische Logarithmusfunktion, so dass erlaubte und nicht erlaubte Eingangsintervalle $\boldsymbol{z}$ vom Typ `lx_cinterval` der Abbildung 24 auf Seite 166 entnommen werden können.

6.2.11 Die Funktion $\arctan(y/x)$

Die C-XSC Funktion

```
lx_interval Atan(const lx_interval& y, const lx_interval& x);
```

berechnet garantierte Einschließungen der Funktionswerte

$$f(x,y) = \arctan(y/x), \quad x,y \in \mathbb{R},\ x \neq 0;\ \ x,y \in IR;$$

Die Funktion `Atan(...)` ist nur für den internen Gebrauch bestimmt und darf nur für **Punktintervalle** $\boldsymbol{y}, \boldsymbol{x}$ aufgerufen werden, wobei $\boldsymbol{x} \neq [0,0]$ vorausgesetzt wird. Außerdem sollen die Funktionswerte $f(x,y)$ für **alle** möglichen Punktintervalle $\boldsymbol{x}, \boldsymbol{y}$ des Datentyps `lx_interval` ohne einen vorzeitigen integer-Überlauf eingeschlossen werden. Daher müssen die beiden Fälle $y/x \to \pm\infty$ und $y/x \to 0$ gesondert behandelt werden. Wegen der Punktsymmetrie der $\arctan$-Funktion kann man sich dabei auf $y, x > 0$ beschränken.

Wir betrachten zunächst den Fall $y/x \to +\infty$.

Für hinreichend große Quotienten y/x soll $f(x,y) = \arctan(y/x)$ eingeschlossen werden durch die Intervallkonstante `Pi_lx_interval()`$/2$, vgl. dazu Tabelle 2 auf Seite 18. Um diese Einschließung zu garantieren, muss nach (48) auf Seite 82 folgende Bedingung erfüllt sein: $y/x > 2^{+2092} \cdot 8.5675\ldots$, und wegen $2^{+2092} \cdot 8.5675\ldots < 2^{+2096}$ gilt

$$\arctan(y/x) \in \texttt{Pi\_lx\_interval()}/2, \quad \text{falls } \ y/x > 2^{+2096}. \tag{1}$$

Im nächsten Schritt müssen wir jetzt noch die Bedingung $y/x > 2^{2096}$ garantieren, ohne die Intervalldivision $\boldsymbol{y} \diamond \boldsymbol{x}$ wegen drohenden integer-Überlaufs wirklich auszuführen. Für y und x benötigen wir dazu eine geeignete Unter- bzw. Oberschranke. In der Darstellung $\boldsymbol{y} = 2^{ex\_y} \cdot \boldsymbol{yl}$ ist $\boldsymbol{yl}$ das staggered Intervall vom Typ `l_interval`, und $yl[1]$ sei die betragsmäßig größte Komponente von $\boldsymbol{yl}$, mit $yl[1] = m \cdot 2^{ex\_yl}$ und $0.5 \leq m < 1$. Damit gilt dann $yl > 0.5 \cdot 2^{ex\_yl} \cdot 1/2$, wobei der letzte Faktor $1/2$ berücksichtigt, dass die nächste Komponente $yl[2]$ negativ sein kann. Für y ergibt sich schließlich die Unterschranke

$$y > 2^{ex\_y + ex\_yl - 2}, \quad ex\_y, ex\_yl \in \mathbb{Z}.$$

Um jetzt mit den entsprechenden Bezeichnungen eine Oberschranke für x zu berechnen, gilt zunächst $xl[1] < 1 \cdot 2^{ex\_xl}$ und damit $xl < 1 \cdot 2^{ex\_xl} \cdot 2$, wobei der letzte Faktor 2 berücksichtigt, dass $xl[2]$ positiv sein kann. Für x erhält man damit die Abschätzung

$$x < 2^{ex\_x + ex\_xl + 1}, \quad ex\_x, ex\_xl \in \mathbb{Z}, \quad \text{und folglich}$$
$$\frac{y}{x} > 2^{ex\_y + ex\_yl - ex\_x - ex\_xl - 3}.$$

Die Bedingung $y/x > 2^{+2096}$ ist also erfüllt, wenn gilt:

$$\begin{aligned} & ex\_y + ex\_yl - ex\_x - ex\_xl > 2099, \\ \iff \quad & ex\_y > ex\_x + (ex\_xl - ex\_yl) + 2099. \end{aligned} \tag{2}$$

Mit einer zusätzlichen Oberschranke für $(ex\_xl - ex\_yl)$ kann noch eine etwas gröbere Bedingung angeben werden, die (2) zur Folge hat. Mit $ex\_xl \leq 1024$ und $ex\_y \geq -1073$ ist diese Oberschranke gegeben durch $(ex\_xl - ex\_yl) \leq 2097$, und (2) ist erfüllt, falls

$$ex\_y > ex\_x + 4196. \tag{3}$$

Damit gilt dann

$$\arctan(y/x) \in \texttt{Pi\_lx\_interval()}/2, \quad \text{falls } ex\_y > ex\_x + 4196. \tag{4}$$

Im Programm kann man die rechte Seite von (3) ohne integer-Überlauf berechnen, falls $ex\_x \leq +2147479451$ erfüllt ist.

Wir betrachten jetzt den Fall $y/x \to 0$.

Wenn man $\arctan(y/x)$ mit der C-XSC Funktion `atan(...)` von Seite 82 einschließen will, so muss nach dem Programm von Seite 86 für das Argument die folgende Bedingung erfüllt sein:

$$\frac{y}{x} > 2^{-2147482626}. \tag{5}$$

Im Fall

$$\frac{y}{x} \leq 2^{-2147482625} \tag{6}$$

muss man daher überlegen, wie $\arctan(y/x)$ ohne Intervalldivision $\boldsymbol{y} \diamond \boldsymbol{x}$ eingeschlossen werden kann. Zunächst wird geklärt, wie die Bedingung (6) im Programm zu realisieren ist. Mit den gleichen Überlegungen der vorhergehenden Seite findet man für y/x die Oberschranke

$$\frac{y}{x} < 2^{ex\_y+ex\_yl-ex\_x-ex\_xl+3},$$

und (6) ist erfüllt, wenn gilt

$$\begin{aligned} & ex\_y + ex\_yl - ex\_x - ex\_xl + 3 \leq -2147482625 \\ \iff \quad & ex\_y + 2147482628 \leq ex\_x + (ex\_xl - ex\_yl). \end{aligned} \tag{7}$$

Um die rechte und linke Seite in (7) ohne integer-Überlauf auswerten zu können, sollte man die entsprechenden Rechnungen im `real`-Format ausführen, indem man ex_y und ex_x in den `real`-Variablen `r1` und `r2` speichert. Es ist jetzt noch zu klären, wie im Fall $y/x \leq 2^{-2147482625}$ der Funktionswert $\arctan(y/x)$ möglichst eng eingeschlossen werden kann. Nach (53) von Seite 84 kann diese Einschließung direkt angegeben werden:

$$0 \leq \arctan(y/x) < 2^{-2147482625}, \tag{8}$$

wobei $2^{-2147482625}$ einfach durch `lx_real(-2147482625,l_real(1))` zu realisieren ist. Damit ist auch der Fall $y/x \to 0$ abgeschlossen, und im restlichen Bereich wird dann $\arctan(y/x)$ mit Hilfe der C-XSC Funktion `atan(...)` von Seite 82 in sehr hoher Genauigkeit eingeschlossen.

6.2.12 Die Funktion $\arcsin(z)$

Die C-XSC Funktion

```
lx_cinterval asin(const lx_cinterval& z);
```

berechnet für ein gegebenes rechteckiges komplexes Eingangsintervall $\boldsymbol{z}$ eine garantierte Einschließungen des Intervalls

$$f(\boldsymbol{z}) = \arcsin(\boldsymbol{z}), \quad \boldsymbol{z} = \boldsymbol{x} + i \cdot \boldsymbol{y},\ \ \boldsymbol{x}, \boldsymbol{y} \in IR,\ \ i = \sqrt{-1}.$$

Nach Walter Krämer gelten für $w = \arcsin(z) = \Re(w) + i \cdot \Im(w)$, mit $z = x + i \cdot y \in \mathbb{C}$ und $x, y \in \mathbb{R}$ die folgenden Beziehungen, [15]:

(1)
$$\alpha := \frac{1}{2} \cdot \left\{ \sqrt{(x+1)^2 + y^2} + \sqrt{(x-1)^2 + y^2} \right\}$$

(2)
$$\beta := \frac{1}{2} \cdot \left\{ \sqrt{(x+1)^2 + y^2} - \sqrt{(x-1)^2 + y^2} \right\}$$

(3)
$$\beta = \frac{x}{\frac{1}{2} \cdot \left\{ \sqrt{(x+1)^2 + y^2} + \sqrt{(x-1)^2 + y^2} \right\}} = \frac{x}{\alpha}$$

(4)
$$\Re(w) := \arcsin(\beta)$$

(5)
$$\Im(w) = \begin{cases} +\ln(\alpha + \sqrt{\alpha^2 - 1}), & \text{falls } \ y > 0 \\ +\ln(\alpha + \sqrt{\alpha^2 - 1}), & \text{falls } \ y = 0 \ \text{ und } \ x \leq -1 \\ \ 0 \qquad\quad , & \text{falls } \ y = 0 \ \text{ und } \ -1 \leq x \leq +1 \\ -\ln(\alpha + \sqrt{\alpha^2 - 1}), & \text{falls } \ y = 0 \ \text{ und } \ x \geq +1 \\ -\ln(\alpha + \sqrt{\alpha^2 - 1}), & \text{falls } \ y < 0 \end{cases}$$

Mit der Identität $\ln(\alpha + \sqrt{\alpha^2 - 1}) \equiv \text{arcosh}\,(\alpha),\ \alpha \geq 1$, gilt außerdem:

(6)
$$\Im(w) = \begin{cases} +\text{arcosh}\,(\alpha), & \text{falls } \ y > 0 \\ +\text{arcosh}\,(\alpha), & \text{falls } \ y = 0 \ \text{ und } \ x \leq -1 \\ \ 0 \qquad\quad , & \text{falls } \ y = 0 \ \text{ und } \ -1 \leq x \leq +1 \\ -\text{arcosh}\,(\alpha), & \text{falls } \ y = 0 \ \text{ und } \ x \geq +1 \\ -\text{arcosh}\,(\alpha), & \text{falls } \ y < 0 \end{cases}$$

Wegen $\alpha(x, y) \geq \alpha(x, 0) = (|x+1| + |x-1|)/2$ folgt mit einfachen Fallunterscheidungen $(|x| \leq 1,\ x < -1,\ x > +1)$:

(7)
$$\alpha(-x, y) \equiv \alpha(x, y) \equiv \alpha(x, -y) \geq \text{Max}(1, |x|),$$

und unter den Voraussetzungen $y = 0$ und $-1 \leq x \leq +1$ gilt:

(8)
$$\alpha(x, 0) \equiv 1, \quad \text{d.h. } \Im(w) = \pm\text{arcosh}\,(1) = 0, \quad \Re(w) = \arcsin(x);$$

Es gilt außerdem

$$|\beta(x, y)| \leq |\beta(x, 0)| \leq 1$$

Zum Beweis gilt nach (3):

$$|\beta(x,y)| \le |\beta(x,0)| = \frac{2\cdot|x|}{|x+1|+|x-1|} =: R(x),$$

und wegen $R(-x) \equiv R(x)$ kann man sich auf $x \ge 0$ beschränken:

Sei $0 \le x \le 1$, d.h. $|x-1| = -(x-1) \rightsquigarrow R(x) = \frac{2x}{(x+1)-(x-1)} = x \le 1$;

Sei $x > 1$, d.h. $|x-1| = +(x-1) \rightsquigarrow R(x) = \frac{2x}{(x+1)+(x-1)} = 1$ ∎

Im Algorithmus von Markus Neher wird für das rechteckige, komplexe Intervallargument, [7]

$$\boldsymbol{z} = \boldsymbol{x} + i\cdot\boldsymbol{y} = [x_1,x_2] + i\cdot[y_1,y_2]$$

eine garantierte Einschließung für $\arcsin(\boldsymbol{z})$ berechnet, wobei die beiden reellen Funktionen $\arcsin(\beta)$ und arcosh $(\alpha) \equiv \ln(\alpha+\sqrt{\alpha^2-1})$ für **Intervallargumente $\boldsymbol{\alpha}$** und **$\boldsymbol{\beta}$** auszuwerten sind. In (1) und (3) sind die reellen Werte x, y durch entsprechende Punktintervalle $\boldsymbol{x} = [x_1,x_1],\ \boldsymbol{y} = [y_1,y_2],\ x_i, y_i \in S(2,53)$ zu ersetzen, da nach W. Krämer die Extremalpunkte des Real- und Imaginärteils von $\boldsymbol{w} = \arcsin(\boldsymbol{z})$ auf den Eckpunkten des rechteckigen, komplexen Eingabeintervalls $\boldsymbol{z}$ liegen, [15].

In den folgenden Unterabschnitten werden die notwendigen Formeln zur Berechnung einer garantierten Einschließung des Real- und Imaginärteils von $\arcsin(\boldsymbol{z})$ zusammengestellt. Um einen vorzeitigen integer-Überlauf zu verhindern, kommen in verschiedenen Teilbereichen der komplexen Ebene entsprechende Termumformungen zur Anwendung.

6.2.12.1 Der Realteil des inversen Sinus Nach Seite 172 gelten für den Realteil $\Re(w)$ die folgenden Darstellungen

$$\beta := \frac{1}{2}\cdot\left\{\sqrt{(x+1)^2+y^2} - \sqrt{(x-1)^2+y^2}\right\}, \tag{9}$$

$$\beta = \frac{x}{\frac{1}{2}\cdot\left\{\sqrt{(x+1)^2+y^2} + \sqrt{(x-1)^2+y^2}\right\}}, \tag{10}$$

$$\Re(w) := \arcsin(\beta), \qquad |\beta(x,y)| \le 1. \tag{11}$$

Wegen $\beta(-x,y) \equiv -\beta(x,y)$ kann man sich auf die rechte komplexe Halbebene, d.h. auf $x \ge 0$ beschränken.

Wir betrachten zunächst den Fall: $\boldsymbol{x \to 0,\ |y| \to \infty}$.

Wegen $x = 0 \rightsquigarrow \beta = 0$ setzen wir im folgenden $x \ne 0$ voraus. Da nach (9) starke Auslöschung zu erwarten ist, benutzen wir zur Einschließung von β die Darstellung (10), aus der direkt abgelesen werden kann, dass β für $x \to 0,\ |y| \to \infty$ beliebig kleine wird. Um einen vorzeitigen integer-Überlauf bei der Auswertung von β zu vermeiden, sollen für $\beta \to 0$ die Funktionswerte $\arcsin(\beta)$ durch ein Intervall $[0,\varepsilon]$ grob eingeschlossen werden, wobei ε möglichst klein zu wählen ist. Aus (10) ergibt sich für β zunächst die Oberschranke

$$\beta < \frac{x}{|y|}, \quad y \ne 0.$$

Testrechnungen haben ergeben, dass im Fall $x/|y| \geq 2^{-2147482618}$ die Funktionswerte $\arcsin(\beta)$ noch in hoher Genauigkeit eingeschlossen werden können. Für

$$\beta < \frac{x}{|y|} \leq 2^{-2147482618} \tag{12}$$

soll daher $\arcsin(\beta)$ durch $[0, \varepsilon]$ grob eingeschlossen werden. Im nächsten Schritt suchen wir dazu eine möglichst einfach auszuwertende Bedingung, die (12) garantiert, ohne den Quotienten $x/|y|$ wegen drohenden integer-Überlaufs wirklich auswerten zu müssen. Mit den Bezeichnungen von Seite 146, 147 gelten für die staggered Zahlen x, y folgende Abschätzungen

$$x < 2^{exx+exxl+1}, \quad |y| > 2^{exy+exyl-2}$$

und (12) ist erfüllt, falls gilt:

$$exx + exxl - exy - exyl + 3 \leq 2^{-2147482618} \quad \Longleftrightarrow$$
$$exx \leq exy + exyl - exxl - 2147482621. \tag{13}$$

Damit die rechte Seite in (13) ohne integer-Überlauf berechnet werden kann, verlangen wir zusätzlich

$$exy + exyl - exxl - 2147482621 \geq -2147483647 \quad \Longleftrightarrow$$
$$exy + exyl - exxl \geq -1026. \tag{14}$$

Im Fall $y = 0$ kann β stets ohne integer-Überlauf berechnet werden, so dass $y \neq 0$ und damit $-1073 \leq exyl \leq 1024$ vorausgesetzt werden kann. Schreibt man dann (14) in der Form

$$exy \geq exxl - exyl - 1026 \tag{15}$$

so macht die Auswertung der rechten Seite jetzt keine Probleme mehr. Wenn also $y \neq 0$ und (13), (15) erfüllt sind, so gilt auch (12) und $\arcsin(\beta)$ kann wie folgt grob durch $[0, \varepsilon]$ eingeschlossen werden. Da nach (12) der Quotient $x/|y|$ sehr klein ist, gilt

$$\arcsin(\beta) \leq \arcsin(x/|y|) < \frac{2 \cdot x}{|y|},$$

und eine grobe Einschließung für $\arcsin(\beta)$ ist unter den Voraussetzungen $y \neq 0$ und (13), (15) gegeben durch

$$\arcsin(\beta) \in [0, 2^{-2147482617}].$$

Damit kann $\arcsin(\beta)$ auch für $x/|y| \leq 2^{-2147482618}$ zwar nur grob, aber ohne integer-Überlauf eingeschlossen werden.

Die Auswertung von $\beta(x, y)$ mit Hilfe von (10) liefert ausreichende Genauigkeit, wenn man den Bereich $\beta(x, y) \to +1$ vermeidet. Für $\beta(x, y) \to +1$ wird die reelle $\arcsin$-Funktion in der Nähe ihrer Nullstelle $\beta = 1$ ausgewertet, und wegen der dort senkrechten Tangente erzeugen unvermeidbare Rundungsfehler bei der Auswertung von $\beta(x, y)$ starke Überschätzungen bei der Einschließung von $\arcsin(\beta)$. Zur Vermeidung dieser Überschätzungen benutzt man im Bereich $0.75 \leq \beta(x, y) \leq +1$ mit $\beta \equiv 1 - \delta$ die folgende Darstellung:

$$\Re(w) := \arcsin(\beta) \equiv \arcsin(1 - \delta) = \frac{\pi}{2} - \arcsin\left(\sqrt{\delta \cdot (2 - \delta)}\right). \tag{16}$$

Wenn man das Argument $\sqrt{\delta \cdot (2-\delta)}$ für $\delta \to 0$ geschickt auswertet, so kann $\Re(w)$ nach (16) mit nahezu optimaler Genauigkeit eingeschlossen werden. Da die Extrema von $\Re(w)$ und $\Im(w)$ stets auf den Eckpunkten[26] des rechteckigen Eingabeintervalls liegen, sind $\beta(x,y)$ und $\delta(x,y)$ nur für Punktintervalle $\boldsymbol{x},\boldsymbol{y}$ auszuwerten.

Bevor wir für die Fälle: $0 \le x < 1,\ x = 1,\ x > 1$ zur Auswertung von $\delta(x,y)$ bzw. $\sqrt{\delta(x,y)}$ die jeweils geeigneten Terme angeben, soll zunächst in der komplexen Ebene der Bereich bestimmt werden, in dem $\beta \ge 0.75$ gilt. Aus (10) folgt unmittelbar, dass $\beta(x,y)$ für festes $x > 0$ sein relatives Maximum bez. y auf der reellen Achse, d.h. für $y = 0$ annimmt. Aus (10) erhält man für diese Maximumwerte direkt:

$$\beta(x,0) = \begin{cases} x & \text{falls } 0 \le x \le 1 \\ 1 & \text{falls } x > 1 \quad \rightsquigarrow \quad \delta = 0; \end{cases} \tag{17}$$

Dass für $x > 0$ die obigen Werte tatsächlich relative Extrema bez. y sind, ergibt sich auch direkt aus der nachfolgenden partiellen Ableitung $\partial\beta(x,y)/\partial y$, die für $y = 0$ verschwindet.

$$\frac{\partial\beta(x,y)}{\partial y} = -2xy \cdot \frac{\dfrac{1}{\sqrt{(x+1)^2+y^2}} + \dfrac{1}{\sqrt{(x-1)^2+y^2}}}{\left(\sqrt{(x+1)^2+y^2} + \sqrt{(x-1)^2+y^2}\right)^2}.$$

Da die partielle Ableitung in der rechten Halbebene für $y > 0$ negativ und für $y < 0$ positiv ist, liegt in der komplexen Ebene der gesuchte Bereich, in dem $\beta(x,y) \ge 0.75$ gilt, innerhalb der Höhenlinie

$$\beta(x,y) = \frac{3}{4} \quad \Longleftrightarrow \quad |y| = \frac{\sqrt{112\,x^2 - 63}}{12},$$

wobei die positive reelle Achse als Symmetrieachse durch diesen Bereich läuft. Beachten Sie bitte, dass durch die Funktionsgleichungen

$$y = \pm\frac{\sqrt{112\,x^2 - 63}}{12}$$

eine Hyperbel mit ihrem rechten Hyperbelast beschrieben wird, dessen Scheitelpunkt durch $x = 3/4$ und $y = 0$ gegeben ist. Der gesuchte Bereich $\beta(x,y) \ge 0.75$ ist in der Abbildung auf der folgenden Seite innerhalb dieses Hyperbelastes dargestellt.

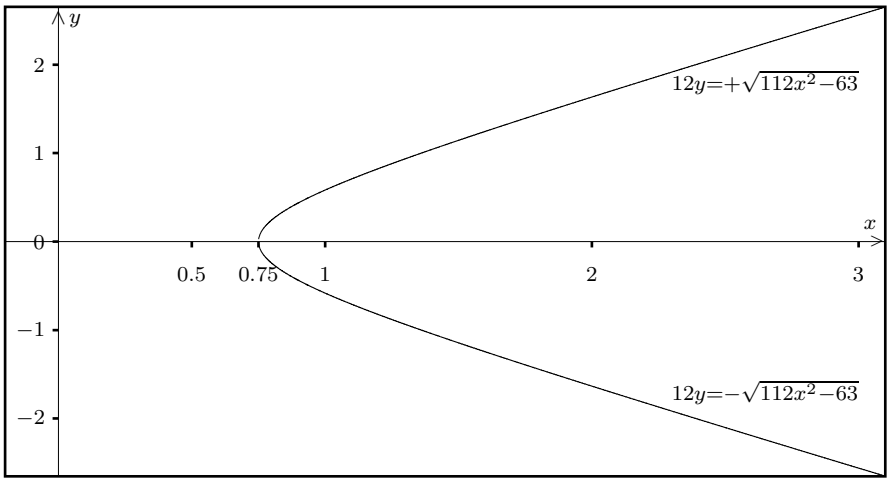
Nach diesen Vorüberlegungen wird jetzt $\beta(x,y) \ge 0.75$ vorausgesetzt, und wir betrachten den ersten Fall: $\boldsymbol{0.75 \le x < 1}$:

Zunächst gilt nach (9)

$$2 \cdot \delta = 2 - 2 \cdot \beta = 2 + \sqrt{(1-x)^2+y^2} - \sqrt{(1+x)^2+y^2}. \tag{18}$$

[26] Das Eingabeintervall darf dabei die Singularitäten $z_{1,2} = \pm 1 + 0 \cdot i$ nicht enthalten.

Bereich in der komplexen Ebene bez. $\beta \geq 0.75$



$y = 0$ liefert $\delta = (1 - x) > 0$, und wegen $\delta \to 0$ erweitert man zur Vermeidung von Auslöschung die rechte Seite in (18) mit

$$N := \left\{2 + \sqrt{(1-x)^2 + y^2}\right\} + \sqrt{(1+x)^2 + y^2} \tag{19}$$

und erhält nach einfachen Rechnungen

$$\delta = \frac{2 \cdot \left(\sqrt{(1-x)^2 + y^2} + (1-x)\right)}{2 + \sqrt{(1-x)^2 + y^2} + \sqrt{(1+x)^2 + y^2}}. \tag{20}$$

Im Bereich $0.75 \leq x < 1$ kann jetzt die rechte Seite von (20) für fast alle y-Werte der erweiterten Klasse `lx_interval` ohne vorzeitigen integer-Überlauf ausgewertet werden, wenn man die obigen Wurzelwerte mit Hilfe der C-XSC Funktion `sqrtx2y2(...)` einschließt.

Im Fall $\mathbf{0.75 \leq \beta \leq 1}$ und $\boldsymbol{x = 1}$ gilt nach (20)

$$\delta = \frac{2 \cdot |y|}{2 + |y| + \sqrt{2^2 + y^2}}, \tag{21}$$

und $\sqrt{\delta}$ lässt sich mit Hilfe dieses Ausdrucks auch jetzt wieder mit ausreichender Genauigkeit und ohne vorzeitigen integer-Überlauf einschließen, wenn rechts in (21) die Wurzel wieder mit Hilfe der `sqrtx2y2(...)`-Funktion ausgewertet wird.

Jetzt bleibt noch der Fall: $\mathbf{0.75 \leq \beta \leq 1}$ und $\boldsymbol{x > 1}$.

Mit (9) und

$$N := \left\{2 + \sqrt{(x-1)^2 + y^2}\right\} + \sqrt{(x+1)^2 + y^2}$$

folgt ganz analog

$$\delta(x,y) = \frac{(x-1) \cdot \left\{\sqrt{1 + \left(\dfrac{y}{x-1}\right)^2} - 1\right\}}{N/2}, \quad \text{falls } x > 1, \tag{22}$$

wobei $\{...\}$ mit Hilfe von `sqrt1pm1(...)` einzuschließen ist:

$$\sqrt{1+\left(\frac{y}{x-1}\right)^2}-1 \in \texttt{sqrt1pm1}\left((\frac{y}{x-1})^2\right).$$

Dabei kann das obige Argument $(y/(x-1))^2$ für $x \to 1$ und $y \to \infty$ keinen integer-Überlauf erzeugen, denn nach der Abbildung auf Seite 176 gilt

$$|y| \le \frac{1}{12}\sqrt{112x^2-63}, \quad \text{und wegen } x-1>0 \text{ folgt}$$

$$\frac{|y|}{x-1} \le \frac{\frac{1}{12}\sqrt{112x^2-63}}{x-1}. \tag{23}$$

Da in der Klasse `lx_interval` $x-1 > 10^{-632}$ sicher erfüllt ist, bleibt die rechte Seite von (23) beschränkt, so dass auch für $(y/(x-1))^2$ kein integer-Überlauf auftreten kann.

Für $y \to 0$ und $x \to \infty$, d.h. für $\delta(x,y) \to 0$, erzeugt das Argument $(y/(x-1))^2$ jedoch einen integer-Überlauf, der wie folgt vermieden werden kann. Nach (16) ist der Ausdruck $\pi/2 - \arcsin(\sqrt{\delta\cdot(2-\delta)})$ für $\delta \to 0$ auszuwerten, und die Idee ist dabei, δ so klein zu wählen, dass $\pi/2 - \arcsin(\sqrt{\delta\cdot(2-\delta)})$ mit Hilfe der Intervallkonstanten $Ph := \texttt{Pi\_lx\_interval()}/2$ durch das Intervall $[\texttt{Inf}(Ph), \pi/2]$ eingeschlossen werden kann. Mit[27]

$$\varepsilon := \pi/2 - \texttt{Inf}(Ph) = 2.053013695\ldots\cdot 10^{-631}$$

verlangen wir daher

$$\frac{\pi}{2} - \arcsin(\sqrt{\delta\cdot(2-\delta)}) > \frac{\pi}{2}-\varepsilon \quad \Longleftrightarrow$$

$$\arcsin(\sqrt{\delta\cdot(2-\delta)}) < \varepsilon, \tag{24}$$

und (24) ist erfüllt, wenn gilt:

$$\arcsin(\sqrt{2\delta}) < \varepsilon. \tag{25}$$

Im nächsten Schritt bestimmen wir eine Oberschranke von $\sqrt{\delta(x,y)}$, und mit (22) erhält man durch Erweitern mit $\sqrt{1+(...)^2}+1$ zunächst

$$\sqrt{\delta} = \frac{\sqrt{2}\cdot|y|}{\sqrt{N}\cdot\sqrt{(x-1)\cdot\left\{\sqrt{1+\left(\frac{y}{x-1}\right)^2}+1\right\}}}.$$

Mit den folgenden Abschätzungen

$$\sqrt{N} \ge 2 \quad \text{und} \quad \sqrt{(x-1)\cdot\left\{\sqrt{1+\left(\frac{y}{x-1}\right)^2}+1\right\}} > \sqrt{x-1}$$

[27] Siehe Tabelle 2 auf Seite 18

erhält man

$$\sqrt{2\delta} \leq \frac{|y| \cdot \sqrt{2}}{\sqrt{x-1}},$$

und (25) ist erfüllt, wenn

$$\arcsin\left(\frac{|y| \cdot \sqrt{2}}{\sqrt{x-1}}\right) < \varepsilon. \tag{26}$$

Da $\varepsilon \ll 1$ erfüllt ist, kann auf (26) die streng monotone $\sin$-Funktion angewandt werden:

$$\frac{|y| \cdot \sqrt{2}}{\sqrt{x-1}} < \sin(\varepsilon), \quad \text{und dies ist erfüllt, falls}$$

$$\frac{|y| \cdot \sqrt{2}}{\sqrt{x-1}} < \frac{\varepsilon}{2} \quad \Longleftrightarrow \quad 2\sqrt{2} \cdot |y| < \sqrt{x-1} \cdot \varepsilon, \tag{27}$$

und (27) ist erfüllt, wenn gilt:

$$4 \cdot |y| < \varepsilon \cdot \sqrt{x-1}, \tag{28}$$

und beide Seiten in (28) können jetzt problemlos, d.h. ohne vorzeitigen integer-Überlauf ausgewertet werden.

Zusammenfassung:

Im Fall $4 \cdot |y| < \varepsilon \cdot \sqrt{x-1}$ und $x > 1$ gilt:
$\frac{\pi}{2} - \arcsin(\sqrt{\delta \cdot (2-\delta)}) \in$ `Pi_lx_interval()/2`,
sonst wird $\delta(x, y)$ nach (22) von Seite 176 berechnet.

Anmerkung:

Mit $\varepsilon = 2.053013695\ldots \cdot 10^{-631}$ ist $4 \cdot |y| < \varepsilon \cdot \sqrt{x-1}$ erfüllt, falls gilt

$$|y| < 5.1325 \cdot 10^{-632} \cdot \sqrt{x-1} = 7.4699\ldots \cdot 2^{-2100} \cdot \sqrt{x-1},$$

und die obige Bedingung $4 \cdot |y| < \varepsilon \cdot \sqrt{x-1}$ ist damit erfüllt, wenn gilt:

$$|y| < 7.4699 \cdot 2^{-2100} \cdot \sqrt{x-1}. \tag{29}$$

Auch jetzt kann die rechte Seite von (29) ohne vorzeitigen integer-Überlauf ausgewertet werden.

6.2.12.2 Der Imaginärteil des inversen Sinus Mit der schon bekannten Definition

$$\alpha := \frac{1}{2} \cdot \left\{ \sqrt{(x+1)^2 + y^2} + \sqrt{(x-1)^2 + y^2} \right\} \tag{30}$$

gilt nach (5) von Seite 172 für den Imaginärteil $\Im(w)$ von $w = \arcsin(z),\ z = x + i \cdot y \in \mathbb{C}$

$$\Im(w) = \begin{cases} +\ln(\alpha + \sqrt{\alpha^2 - 1}), & \text{falls } y > 0 \\ +\ln(\alpha + \sqrt{\alpha^2 - 1}), & \text{falls } y = 0 \text{ und } x \leq -1 \\ 0 \qquad\qquad\quad , & \text{falls } y = 0 \text{ und } -1 \leq x \leq +1 \\ -\ln(\alpha + \sqrt{\alpha^2 - 1}), & \text{falls } y = 0 \text{ und } x \geq +1 \\ -\ln(\alpha + \sqrt{\alpha^2 - 1}), & \text{falls } y < 0. \end{cases} \tag{31}$$

Wie bei der Berechnung des Realteils kann man sich auch jetzt auf $x \geq 0$ beschränken, wobei $\alpha = \alpha(x, y)$ wieder nur für Punktintervalle $\boldsymbol{x}, \boldsymbol{y}$ auszuwerten ist. Berechnet man die Quadratwurzeln in (30) mit Hilfe der C-XSC Funktion `sqrtx2y2(...)` und den Ausdruck $\sqrt{\alpha^2 - 1}$ in (31) mit der Funktion `sqrtx2m1(...)`, so werden vorzeitige integer-Überläufe weitgehend vermieden.

Bei der intervallmäßigen Auswertung des Terms $\ln(\alpha + \sqrt{\alpha^2 - 1})$ treten jedoch für $\mathtt{Sup}(\alpha) \to +1$ erhebliche Überschätzungen auf, da dann die obige $\ln$-Funktion in der Nähe ihrer Nullstelle 1 zu berechnen ist. Man kann diese Überschätzungen vermeiden, wenn man mit

$$\alpha = 1 + (\alpha - 1) = 1 + \delta, \quad \delta(x, y) := \alpha(x, y) - 1$$

die folgende Identität benutzt:

$$\ln(\alpha + \sqrt{\alpha^2 - 1}) = \ln\left(1 + \sqrt{\delta} \cdot \{\sqrt{\delta} + \sqrt{2 + \delta}\}\right). \tag{32}$$

Bevor wir auf die Berechnung von $\delta := \alpha - 1$ näher eingehen, bestimmen wir zunächst in der komplexen Ebene diejenigen Bereiche, in denen $\ln(\alpha + \sqrt{\alpha^2 - 1})$ direkt oder mit Hilfe der rechten Seite von (32) ausgewertet wird. Dazu beweisen wir mit $z = x + i \cdot y$

$$|x| \geq 2 \ \vee \ |y| \geq 2 \quad \Longrightarrow \quad \alpha(x, y) \geq 2; \tag{33}$$

Zum Beweis benutzen wir: $\alpha(x, y) \geq \alpha(x, 0) = (|x + 1| + |x - 1|)/2 =: r(x)$. Sei zunächst $|x| \geq 2$, dann folgt mit einfachen Fallunterscheidungen: $r(x) \geq 2$. Außerdem gilt: $\alpha(x, y) \geq \alpha(0, y) = |y|$, und mit der Voraussetzung $|y| \geq 2$ folgt der zweite Teil ∎

Außerhalb eines Quadrats mit der Seitenlänge 4 und dem Mittelpunkt im Ursprung der komplexen Ebene benutzen wir daher wegen (33) die linke Seite von (32) direkt und innerhalb dieses Quadrats gilt: $\ln(\alpha + \sqrt{\alpha^2 - 1}) \equiv \ln(1 + \sqrt{\delta} \cdot \{\sqrt{\delta} + \sqrt{2 + \delta}\})$, mit $\delta(x, y) := \alpha(x, y) - 1$.

In den nachfolgenden Abschnitten wird der Bereich $0 \leq x \leq 2 \ \wedge \ |y| \leq 2$ in drei Teilbereiche zerlegt. In jedem Teilbereich ist dann ein geeigneter Ausdruck für $\delta(x, y)$ anzugeben, der ohne vorzeitigen integer-Überlauf ausgewertet werden kann.

Sei $\mathbf{0 \le x < 1 \ \wedge \ |y| \le 2}$.

Nach (30) gilt zunächst mit $\delta := \alpha - 1$

$$\begin{aligned} 2\delta &= \sqrt{(x+1)^2+y^2} + \sqrt{(x-1)^2+y^2} - 2 \\ &= |x+1| \cdot \left\{ \sqrt{1+\left(\frac{y}{x+1}\right)^2} - 1 \right\} + |x-1| \cdot \left\{ \sqrt{1+\left(\frac{y}{x-1}\right)^2} - 1 \right\} \\ &= (x+1) \cdot \left\{ \sqrt{1+\left(\frac{y}{1+x}\right)^2} - 1 \right\} + (1-x) \cdot \left\{ \sqrt{1+\left(\frac{y}{1-x}\right)^2} - 1 \right\} \end{aligned}$$

und mit der für $t \ge -1$ gültigen Identität

$$\sqrt{1+t} - 1 \equiv \frac{t}{\sqrt{1+t}+1}$$

erhält man schließlich

$$\delta = y^2 \cdot \left\{ \frac{1}{\sqrt{(x+1)^2+y^2}+(x+1)} + \frac{1}{\sqrt{(1-x)^2+y^2}+(1-x)} \right\} /2, \quad \text{mit} \tag{34}$$

$$\sqrt{\delta} = |y| \cdot \sqrt{\{\ldots\}/2}, \tag{35}$$

wobei jetzt die rechte Seite von (35) ohne integer-Überlauf ausgewertet werden kann, wenn die Quadratwurzeln in (34) wieder mit der C-XSC Funktion `sqrtx2y2(...)` berechnet werden.

Allerdings wird ein solcher integer-Überlauf nach (34) wegen des Faktors y^2 immer noch auftreten, wenn in (32) die Summe $2+\delta$ ausgewertet wird. Um dies zu vermeiden, betrachten wir jetzt den Fall $t := \sqrt{\delta} \to 0$ und es gilt:

$$t + \sqrt{2+t^2} \to \sqrt{2}, \quad \text{mit} \quad t + \sqrt{2+t^2} \ge \sqrt{2}.$$

Wir wollen jetzt für $t \to 0$ die Einschließung des Funktionsterms rechts in (32) so vereinfachen, dass die Wurzel $\sqrt{2+\delta(x,y)}$ nicht mehr direkt berechnet werden muss. Mit $\sqrt{2} \in$ `Sqrt2_lx_interval()` und `Sup(Sqrt2_lx_interval())` $= \sqrt{2}+\varepsilon$ verlangen wir daher[28]

$$t + \sqrt{2+t^2} < \sqrt{2} + \varepsilon, \quad \text{d.h.} \tag{36}$$

$$t + \sqrt{2+t^2} \in \texttt{Sqrt2\_lx\_interval()}. \tag{37}$$

Bedeutet dann T die berechnete Einschließung von $\sqrt{\delta} \in T$, so gilt

$$\ln\left(1 + \sqrt{\delta} \cdot \{\sqrt{\delta} + \sqrt{2+\delta}\}\right) \in \texttt{lnp1}(T \diamond \texttt{Sqrt2\_lx\_interval()}). \tag{38}$$

[28] $\varepsilon = 5.8078220004\ldots \cdot 10^{-632} = 4.226435019\ldots \cdot 2^{-2099}$, vgl. Seite 18.

Wir benötigen jetzt noch eine Bedingung für t, so dass die Forderung (36) und damit auch (37) erfüllt wird. Für $t < 1$ ist (36) erfüllt, wenn gilt

$$t + \sqrt{2+t} < \sqrt{2} + \varepsilon. \tag{39}$$

Für $t < \varepsilon$ gilt $\sqrt{2+t} < \sqrt{2+\varepsilon}$, und (39) ist erfüllt, falls

$$t + \sqrt{2+\varepsilon} < \sqrt{2} + \varepsilon \quad \Longleftrightarrow \quad t < \sqrt{2} + \varepsilon - \sqrt{2+\varepsilon}.$$

Wegen $\sqrt{2} + \varepsilon - \sqrt{2+\varepsilon} > \varepsilon/2 \ \ \forall \varepsilon > 0$ erhalten wir schließlich

$$t + \sqrt{2+t^2} < \sqrt{2} + \varepsilon, \quad \text{falls } \ t = \sqrt{\delta} < \frac{\varepsilon}{2}, \tag{40}$$

sonst wird mit der Einschließung T von $t = \sqrt{\delta} \in T$ der Funktionsterm

$$\ln\left(1 + \sqrt{\delta} \cdot \{\sqrt{\delta} + \sqrt{2+\delta}\}\right)$$

intervallmäßig ausgewertet, und erst jetzt kann $2 + \delta$ mit $\delta = \texttt{sqr}(\sqrt{\delta})$ ohne vorzeitigen integer-Überlauf berechnet werden.

Sei $\boldsymbol{x = 1 \ \wedge \ |y| \leq 2}$.

Aus (30) von Seite 179 folgt mit $x = 1$ für $\delta = \alpha - 1$ direkt

$$\delta(1, y) = \left\{ \sqrt{1 + \left(\frac{y}{2}\right)^2} - 1 \right\} + \frac{|y|}{2}\,, \tag{41}$$

und mit der Identität

$$\sqrt{1+t} - 1 \equiv \frac{t}{\sqrt{1+t}+1}, \quad t \geq -1,$$

erhält man sofort

$$\delta(1, y) = |y| \cdot \left\{ \frac{1}{2} + \frac{|y|/4}{\sqrt{1 + \left(\frac{y}{2}\right)^2} + 1} \right\}. \tag{42}$$

Mit $\delta(1, y)$ kann $\ln(1 + \{\delta + \sqrt{\delta(2+\delta)}\})$ ohne vorzeitigen integer-Überlauf ausgewertet werden, wenn in (42) die Wurzel mit Hilfe von $\texttt{sqrt1px2}(y/2)$ ausgewertet wird und $\texttt{lnp1}(\{\delta + \sqrt{\delta(2+\delta)}\})$ bei der Einschließung des Imaginärteils zur Anwendung kommt. Beachten Sie bitte, dass in (42) keine Auslöschungseffekte auftreten können, während in (41) bei Anwendung der C-XSC Funktion $\texttt{sqrtp1m1}(y^2/4)$ wegen y^2 ein vorzeitiger integer-Überlauf eintreten würde.

Sei $\mathbf{1 < x \leq 2 \ \wedge \ |y| \leq 2}$.

Im Fall $y = 0$ gilt $\delta(x) = x - 1$, und der Funktionsterm $\ln(1 + \{\delta + \sqrt{\delta(2+\delta)}\})$ kann problemlos berechnet werden.

Sei also $\mathbf{1 < x \leq 2 \ \wedge \ 0 < |y| \leq 2}$.

Es gilt zunächst:

$$\begin{aligned} 2\delta &= \sqrt{(x+1)^2 + y^2} + \sqrt{(x-1)^2 + y^2} - 2 \\ &= (x+1) \cdot \sqrt{1 + \left(\frac{y}{x+1}\right)^2} + (x-1) \cdot \sqrt{1 + \left(\frac{y}{x-1}\right)^2} - 2 \\ &= (x+1) \cdot \left\{ \sqrt{1 + \left(\frac{y}{x+1}\right)^2} - 1 \right\} + (x-1) \cdot \left\{ \sqrt{1 + \left(\frac{y}{x-1}\right)^2} - 1 \right\} \\ &\quad +2 \cdot (x-1). \end{aligned}$$

Mit der Identität

$$\sqrt{1+t} - 1 \equiv \frac{t}{\sqrt{1+t}+1}, \quad t \geq -1,$$

und mit den Abkürzungen

$$u := \frac{|y|}{\sqrt{1 + \left(\frac{1+x}{y}\right)^2} + \frac{1+x}{|y|}}, \tag{43}$$

$$v := \frac{|y|}{\sqrt{1 + \left(\frac{x-1}{y}\right)^2} + \frac{x-1}{|y|}}, \tag{44}$$

$$w := 2 \cdot (x-1); \tag{45}$$

erhält man schließlich:

$$\delta(x,y) = \frac{u+v+w}{2}. \tag{46}$$

Da u und v und damit auch $\delta(x,y)$ für $|y| \to 0$ einen vorzeitigen integer-Überlauf erzeugen, während $w > 0$ auf der Maschine die Schranke 2^{-2100} nicht unterschreitet, kann die Einschließung von $\delta(x,y)$ für $|y| \to 0$ durch die folgenden Überlegungen so vereinfacht werden, dass der vorzeitige integer-Überlauf vermieden wird. Wir multiplizieren dazu $w = 2 \cdot (x-1)$ mit dem Intervall $[1, 1 + 2^{-2097}]$, das mit dem Punktintervall `One_p_lx_interval()` $= 1 + 2^{-2097}$ leicht zu realisieren ist. Es gilt dann

$$w = 2 \cdot (x-1) \in 2 \cdot (x-1) \cdot [1, 1 + 2^{-2097}],$$

und wir stellen die Forderung:

$$2 \cdot (x-1) \cdot (1 + 2^{-2097}) > 2 \cdot (x-1) + \frac{2y^2}{x-1}, \tag{47}$$

wobei $2y^2/(x-1)$ eine Oberschranke von $u+v$ ist. Zum Beweis gehen wir aus von

$$u \le \frac{y^2}{x+1}, \quad v \le \frac{y^2}{x-1}, \quad \rightsquigarrow$$

$$\begin{aligned} u+v &\le y^2 \cdot \frac{2x}{(x-1)(x+1)}, \quad \text{und für } x \le 2 \text{ gilt} \\ u+v &\le \frac{4y^2}{(x-1)(x+1)}, \quad \text{und wegen } x+1 > 2 \text{ folgt} \\ u+v &\le \frac{2y^2}{x-1}, \quad \text{falls } 1 < x \le 2 \end{aligned}$$ ∎

Wenn also (47) erfüllt ist, so gilt $u+v+w \in 2 \cdot (x-1) \cdot [1, 1+2^{-2097}]$. Die Forderung (47) ist äquivalent mit

$$(x-1) \cdot 2^{-2097} > \frac{y^2}{x-1} \quad \Longleftrightarrow \quad (x-1)^2 \cdot 2^{-2097} > y^2,$$

und die letzte Ungleichung ist erfüllt, wenn gilt

$$\begin{aligned} & (x-1)^2 \cdot 2^{-2098} > y^2 \\ \Longleftrightarrow \quad & (x-1) \cdot 2^{-1049} > |y| \quad \Longleftrightarrow \quad (x-1) > 2^{+1049} \cdot |y|. \end{aligned} \tag{48}$$

Wenn also (48) erfüllt ist, so gilt

$$\delta(x,y) = \frac{u+v+w}{2} \in (x-1) \cdot [1, 1+2^{-2097}],$$

sonst werden u, v, w nach (43),(44),(45) ohne vorzeitigen integer-Überlauf berechnet. Die Einschließung des Imaginärteils kann damit problemlos durchgeführt werden.

6.2.12.3 Testrechnungen Nach Angabe der Algorithmen für den Real- und Imaginärteil der komplexen $\arcsin(z)$-Funktion erhält man mit der folgenden Tabelle einen groben Überblick über den großen Definitionsbereich der genannten Funktion. Die Angaben in der Tabelle beziehen sich auf die Einschließung der $\arcsin(z)$-Funktion nur für komplexe Punktintervalle $\boldsymbol{z} = \boldsymbol{x} + i \cdot \boldsymbol{y}$, wobei die reellen Punktintervalle $\boldsymbol{x}$ und $\boldsymbol{y}$ vom Typ `lx_interval` sind. Beachten Sie bitte, dass bei der $\arcsin(z)$-Funktion die komplexe Ebene längs der reellen Achse aufgeschnitten ist von $-\infty$ bis -1 und von $+1$ bis $+\infty$. Die Definition der Funktionswerte auf den Verzweigungsschnitten kann in der Abbildung auf Seite 189 anhand der fettgedruckten Pfeile abgelesen werden.

Auswertebereiche von $\arcsin(z)$, $z = x + i \cdot y$		
$\boldsymbol{x}$	$\boldsymbol{y}$	**Anmerkung**
0	$2^{-2147482626} \leq y \leq 2^{+2147482625}$	optimale Einschließung
$2^{-2147482625}$	$2^{-2147482614} \leq y \leq 2^{-5}$	optimale Einschließung
$2^{-2147482625}$	$2^{-5} < y \leq 2^{+2147483647}$	grobe Einschlg. des Realteils
0.5	$2^{-2147482626} \leq y \leq 2^{+2147482619}$	optimale Einschließung
0.5	$2^{+2147482619} < y \leq 2^{+2147482625}$	grobe Einschlg. des Realteils
$1 - 2^{-2095}$	$2^{-2147483645} \leq y \leq 2^{+2147482620}$	optimale Einschließung
$1 - 2^{-2095}$	$2^{+2147482620} < y \leq 2^{+2147483647}$	grobe Einschlg. des Realteils
1	$2^{-2147482625} \leq y \leq 2^{+2147482620}$	optimale Einschließung
1	$2^{+2147482620} < y \leq 2^{+2147483647}$	grobe Einschlg. des Realteils
$1 + 2^{-2095}$	$2^{-2147483645} \leq y \leq 2^{+2147482620}$	optimale Einschließung
$1 + 2^{-2095}$	$2^{+2147482620} < y \leq 2^{+2147483647}$	grobe Einschlg. des Realteils
1.5	$2^{-2147483645} \leq y \leq 2^{+2147482620}$	optimale Einschließung
1.5	$2^{+2147482620} < y \leq 2^{+2147483647}$	grobe Einschlg. des Realteils
$2^{+2147482625}$	$2^{-2147483645} \leq y \leq 2^{+2147482625}$	optimale Einschließung

Tabelle 3: $\arcsin(z)$, mögliche Auswertebereiche

Anmerkungen:

1. Die Anmerkung *grobe Einschlg. des Realteils* bezieht sich auf den Algorithmus von Seite 174, wobei $\arcsin(\beta)$ grob durch $[0, 2^{-2147482617}]$ eingeschlossen wird. Die Einschließung des Imaginärteils erfolgt aber optimal bezüglich der gewählten Präzision.

2. Bei maximaler Präzision `stagprec` $= 30$ erhält man für Punktintervalle Einschließungen für den Real- und Imaginärteil mit ca. 470 korrekten Dezimalstellen.

Im **1. Beispiel** liefert das folgende Programm `lx_test71` für das komplexe Argument $z = 2^{-2147482625} + i \cdot 2^{-2147482614}$ eine garantierte und sehr enge Einschließung für den Funktionswert $f(z) = \arcsin(z)$.

```
// Programm  lx_test71.cpp;
// Zum Test von asin(z);

#include <iostream>
#include "lx_cimath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 30;
    lx_cinterval Z,W;
    lx_interval X,Y;
    string s;

    X = lx_interval(-2147482625,l_interval(1));
    Y = lx_interval(-2147482614,l_interval(1));
    Z = lx_cinterval(X,Y);
    W = asin(Z);
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific;
    s << W;
    cout << "asin(Z) = " << s << endl;
}
```

Die folgende Ausgabe von `W` über die Zeichenkette `s` liefert die folgenden Einschließung des Real- und Imaginärteils von $\arcsin(z)$ in hoher Genauigkeit:

$$\Re(f(z)) \in 2^{-2147483647} \cdot [\underbrace{4.4942328371557\ldots303999\ldots999}_{469 \text{ korrekte Dez.-Ziffern}}777\ldots \cdot 10^{+307},$$
$$4.4942328371557\ldots304000\ldots000555\ldots \cdot 10^{+307}].$$

$$\Im(f(z)) \in 2^{-2147483636} \cdot [\underbrace{4.4942328371557\ldots303999\ldots999}_{476 \text{ korrekte Dez.-Ziffern}}791\ldots \cdot 10^{+307},$$
$$4.4942328371557\ldots304000\ldots000198\ldots \cdot 10^{+307}].$$

Zeigen Sie mit dem gleichen Programm, dass mit $z = 2^{-2147483645} + i \cdot 2^{+2147482625}$ der Imaginärteil von $f(z) = \arcsin(z)$ in sehr hoher Genauigkeit und der kleine Realteil $\Re(f(z)) \ll 1$ nur noch grob eingeschlossen wird, da $\Re(f(z))$ selbst in der erweiterten Klasse `lx_interval` im Unterlaufbereich liegt.

Im **2. Beispiel** liefert das folgende Programm `lx_test72` für das komplexe Argument $z = 2^{+2147482625} + i \cdot 2^{+2147482625}$ eine garantierte und sehr enge Einschließung für den Funktionswert $f(z) = \arcsin(z)$.

```
// Programm  lx_test72.cpp;
// Zum Test von asin(z);

#include <iostream>
#include "lx_cimath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 30;
    lx_cinterval Z,W;
    lx_interval X,Y;
    string s;

    X = lx_interval(+2147482625,l_interval(1));
    Y = lx_interval(+2147482625,l_interval(1));
    Z = lx_cinterval(X,Y);
    W = asin(Z);
    cout << SetDotPrecision(16*stagprec,16*stagprec+5)
         << Scientific;
    s << W;
    cout << "asin(Z) = " << s << endl;
}
```

Die folgende Ausgabe von `W` über die Zeichenkette `s` liefert die folgende Einschließung des Real- und Imaginärteils von $\arcsin(z)$ in hoher Genauigkeit:

$$\Re(f(z)) \in 2^{-1023} \cdot \underbrace{7.059524432365321347574\ldots47876180124119895}_{469 \text{ korrekte Dez.-Ziffern}} {}^{284\ldots}_{171\ldots} \cdot 10^{+307}.$$

$$\Im(f(z)) \in 2^{-992} \cdot \underbrace{6.2303266761092972251925\ldots23548322255297464}_{479 \text{ korrekte Dez.-Ziffern}} {}^{716\ldots}_{512\ldots} \cdot 10^{+307}.$$

Zeigen Sie mit dem gleichen Programm, dass mit $z = 2^{+2147482625} + i \cdot 2^{-2147483645}$ Real- und Imaginärteil von $f(z) = \arcsin(z)$ in nahezu gleicher Genauigkeit eingeschlossen werden.

Versucht man mit *Mathematica* Näherungen für die in den Beispielen 1,2 angegebenen Funktionswerte $\arcsin(z)$ zu berechnen, so wird das Programm jeweils wegen eines Overflows abgebrochen.

Im **3. Beispiel** liefert das folgende Programm `lx_test73` für das komplexe Argument $z = 2.501 + i \cdot 1.001 \cdot 10^{+300000}$ eine garantierte und sehr enge Einschließung für den Funktionswert $f(z) = \arcsin(z)$, dabei ist zu beachten, dass Real- und Imaginärteil von z nicht im Staggered Format darstellbar sind und daher nicht wie in den vorhergehenden Beispielen durch Punktintervalle eingeschlossen werden können. Die Einschließung von $\Re(z)$ und $\Im(z)$ erfolgt durch das folgende Programm mit 324 bzw. 300 korrekten Dezimalstellen, so dass die Einschließung von $\arcsin(z)$ nicht mit höherer Genauigkeit erfolgen kann.

```
// Programm  lx_test73.cpp;
// Zum Test von asin(z);

#include <iostream>
#include "lx_cimath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 30;
    lx_cinterval Z,W;
    lx_interval X,Y;

    X = lx_interval(0,"[2.501,2.501]");
    Y = lx_interval(300000,"[1.001,1.001]");
    Z = lx_cinterval(X,Y);
    W = asin(Z);
    cout << SetDotPrecision(16*stagprec,16*stagprec+5)
         << Scientific;
    cout << "asin(Z) = " << W << endl;
}
```

Die folgende Ausgabe von `W` in lesbarer dezimaler Form liefert die Einschließung des Real- und Imaginärteils von $f(z) = \arcsin(z)$ in hoher Genauigkeit:

$$\Re(f(z)) \in 10^{-300000} \cdot \underbrace{2.49850149850149850149850\ldots149850149850}_{\text{300 korrekte Dez.-Ziffern}} \begin{matrix}282\ldots\\019\ldots\end{matrix} \cdot 10^{0}.$$

$$\Im(f(z)) \in 10^{4} \cdot \underbrace{6.90776222044894598234240\ldots2496809307736096}_{\text{301 korrekte Dez.-Ziffern}} \begin{matrix}420\ldots\\348\ldots\end{matrix} \cdot 10^{1}.$$

Eine mit *Mathematica* unter hohem laufzeitmäßigem Aufwand berechnete Näherung für $f(z)$ wird durch das obige Intervall W eingeschlossen.

6.2.13 Die Funktion $\arccos(z)$

Die C-XSC Funktion

```
lx_cinterval acos(const lx_cinterval& z);
```

berechnet für ein gegebenes rechteckiges, komplexes Eingangsintervall $\boldsymbol{z}$ eine garantierte Einschließungen des Intervalls

$$f(\boldsymbol{z}) = \arccos(\boldsymbol{z}), \quad \boldsymbol{z} = \boldsymbol{x} + i \cdot \boldsymbol{y},\ \boldsymbol{x}, \boldsymbol{y} \in IR,\ i = \sqrt{-1},$$

wobei mit $\arccos(z), z \in \mathbb{C}$, der Hauptzweig bezeichnet wird. Zur Berechnung des inversen Cosinus steht folgende Identität zur Verfügung:

$$\arccos(z) \equiv \frac{\pi}{2} - \arcsin(z), \quad z = x + i \cdot y, \quad x, y \in \mathbb{R}. \tag{1}$$

Mit $w := \arccos(z) = \Re(w) + i \cdot \Im(w)$ kann daher $\Im(w)$ über den inversen Sinus optimal berechnet werden. Im Falle $\Re(w) \sim 0$, d.h. $\Re(\arcsin(z)) \sim \pi/2$, wird jedoch $\Re(w)$ nach (1) wegen starker Auslöschung sehr fehlerhaft berechnet, so dass $\Re(w)$ nach (1) nur grob eingeschlossen werden kann. Zur Berechnung von $\Re(w)$ muss also ein anderer Algorithmus angegeben werden.

Zunächst wird skizziert, wie zu einem vorgegebenen, komplexen Argumentintervall

$$\boldsymbol{Z} = \boldsymbol{X} + i \cdot \boldsymbol{Y} = [x_1, x_2] + i \cdot [y_1, y_2], \quad x_i, y_i \in S(2, 53)$$

die Einschließung von $\Re(\arccos(\boldsymbol{Z}))$ zu berechnen ist, [15]. Mit

$$\begin{aligned} \beta &:= \frac{1}{2} \cdot \left\{ \sqrt{(x+1)^2 + y^2} - \sqrt{(x-1)^2 + y^2} \right\}, \quad |\beta(x,y)| \le 1 \\ &= \frac{2x}{\sqrt{(x+1)^2 + y^2} + \sqrt{(x-1)^2 + y^2}}, \quad \beta(-x,y) = -\beta(x,y) \quad \text{gilt:} \end{aligned} \tag{2}$$

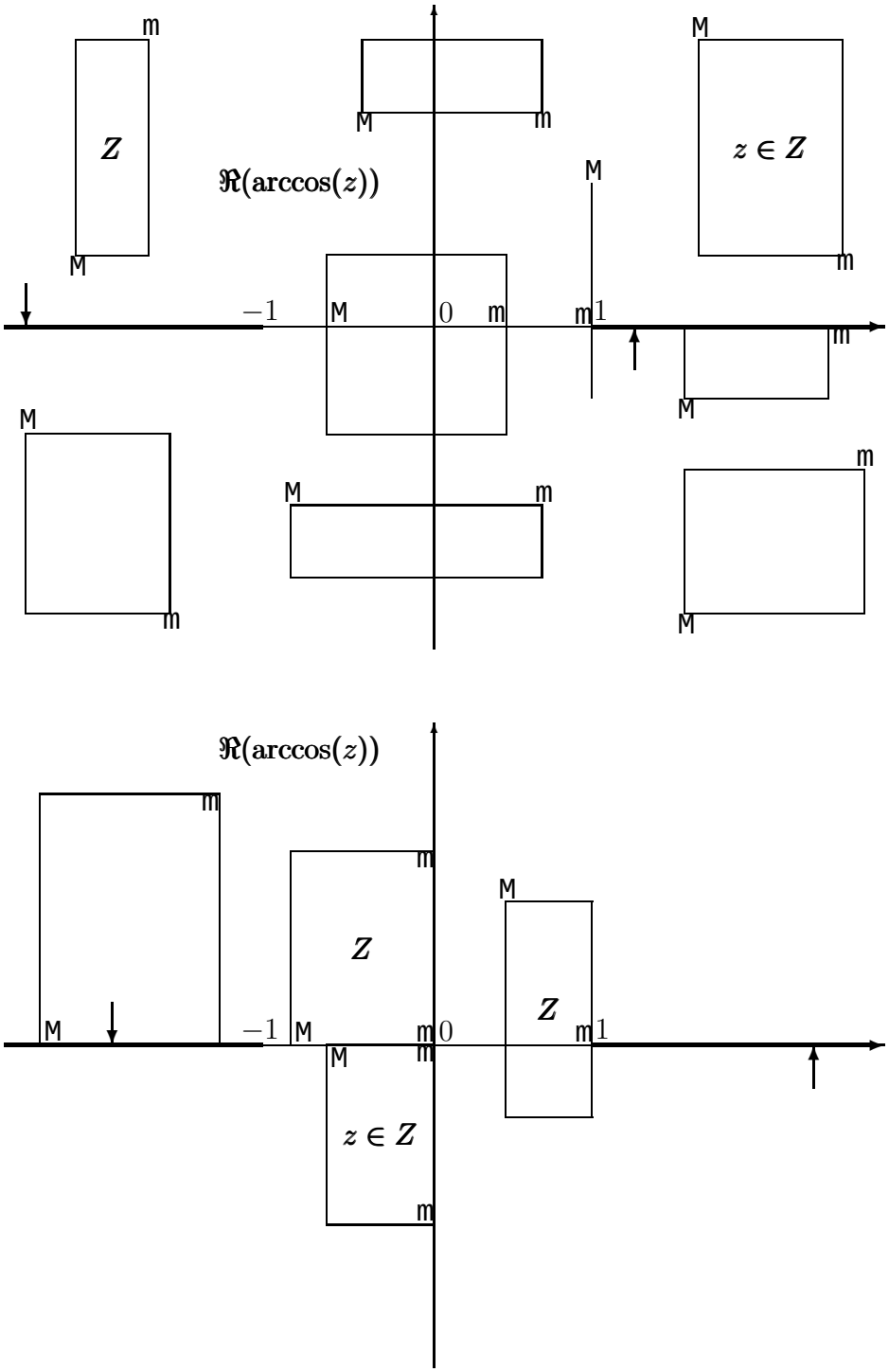
$$\Re(w) = \Re(\arccos(z)) = \arccos(\beta).$$

Die zur Einschließung von $\Re(w)$ benötigten Extrema von $\arccos(\beta)$ über dem Argumentintervall $\boldsymbol{Z}$ liegen auf seinem Rand, falls sich in seinem Innern keine Punkte eines Verzweigungsschnitts befinden. $\arccos(\beta)$ ist also auf den Extremalpunkten intervallmäßig auszuwerten, wobei für x, y entsprechende Punktintervalle $\boldsymbol{x}, \boldsymbol{y}$ einzusetzen sind.

In den folgenden Abbildungen sind zu einigen Intervallen $\boldsymbol{Z}$, die in der komplexen Ebene als Rechtecke dargestellt werden, die Extremalpunkte angegeben. Dabei sind die Koordinaten von `m` die Koordinaten des Extremalpunktes, an dem das Minimum von $\Re(w)$ angenommen wird. Entsprechend sind die Koordinaten von `M` die Koordinaten des Extremalpunktes, an dem das Maximum von $\Re(w)$ angenommen wird. Man erkennt, dass `m` und `M` fast immer die Eckpunkte von $\boldsymbol{Z}$ sind. Nur wenn der Koordinatenursprung im Innern von $\boldsymbol{Z}$ enthalten ist, liegen die Extremalpunkte `m,M` auf der reellen Achse, d.h. also nicht auf den Eckpunkten von $\boldsymbol{Z}$.

Einige Besonderheiten liegen vor, wenn $\boldsymbol{Z}$ z.B. in der rechten Halbebene liegt und die reelle Achse durch das Innere von $\boldsymbol{Z}$ läuft. Dann liegt `M` auf demjenigen linken Eckpunkt von $\boldsymbol{Z}$, der die betragsmäßig maximale y-Koordinate besitzt. Liegt jedoch $\boldsymbol{Z}$ in der linken Halbebene und läuft die reelle Achse wieder durch das Innere von $\boldsymbol{Z}$, dann liegt `m` auf demjenigen rechten Eckpunkt von $\boldsymbol{Z}$, der die betragsmäßig maximale y-Koordinate besitzt.

Zum Verständnis der Abbildung ist weiter zu beachten, dass im Fall zweier Punkte m auf dem Rand von Z auch alle Zwischenpunkte Extremalpunkte sind, auf denen $\Re(w)$ sein Minimum über Z annimmt. Entsprechendes gilt im Falle zweier verschiedener Punkte M auf dem Rand von Z.



In den beiden Abbildungen auf der vorhergehenden Seite sind einige mögliche Lagen der Intervalle $\boldsymbol{Z}$ angegeben. Die Pfeile auf die Verzweigungsschnitte beschreiben, aus welcher Richtung die Funktionswerte im Hauptzweig auf den Schnitten zu approximieren sind.

Mit einem Algorithmus von Markus Neher werden die Extrema von $\arccos(\beta(x,y))$ an den Stellen `M(x,y)` bzw. `m(x,y)` eingeschlossen, indem $\arccos(\beta(x,y))$ intervallmäßig ausgewertet wird, dabei sind die Koordinaten x, y durch ihre Punktintervalle $\boldsymbol{x}, \boldsymbol{y}$ zu ersetzen, [7]. Bei dieser Vorgehensweise sind Fehlerschranken für den Real- und Imaginärteil von $\arccos(z)$ nicht nötig, da $\arccos(\beta(x,y))$ nur mit den in C-XSC schon existierenden Intervallfunktionen eingeschlossen wird.

Da $\arccos(\beta)$ für $\beta \to \pm 1$ senkrechte Tangenten besitzt, treten für $0.75 \leq |\beta| \leq 1$ bei der intervallmäßigen Auswertung von $\arccos(\beta(x,y))$ starke Überschätzungen auf[29].

Zur Vermeidung dieser Überschätzungen wählt man im Fall $\mathbf{0.75 \leq \beta \leq 1}$ die Transformation

$\beta = 1 - \delta$ bzw. $\delta(x,y) = 1 - \beta(x,y) \geq 0$ und benutzt die folgende Identität

$$\arccos(\beta) \equiv \arccos(1-\delta) \equiv \arcsin\left(\sqrt{\delta \cdot (2-\delta)}\right). \tag{3}$$

Im Fall $\mathbf{-1 \leq \beta \leq -0.75}$ verfährt man analog, wählt $\beta(x,y) = -1 + \delta(x,y)$ bzw. $\delta = 1 + \beta = 1 - (-\beta) \geq 0$ und benutzt folgende Identität

$$\begin{aligned} \arccos(\beta) &\equiv \arccos(-1+\delta) \equiv \pi - \arccos(1-\delta) \\ &\equiv \pi - \arcsin\left(\sqrt{\delta \cdot (2-\delta)}\right). \end{aligned} \tag{4}$$

Wählt man dabei rechts in (3) und (4) für $\delta(x,y) \ll 1$ einen geeignet Term, so dass $\sqrt{\delta(x,y)}$ in maximaler Genauigkeit eingeschlossen werden kann, so kann auch der Funktionsterm $\arccos(\beta(x,y))$ für $\beta \to \pm 1$ ebenfalls in hoher Genauigkeit eingeschlossen werden. In (3) und (4) ist $\arcsin(\sqrt{\delta \cdot (2-\delta)})$ einzuschließen, wobei δ in beiden Fällen definiert ist durch

$$\delta(x,y) := 1 - |\beta(x,y)|. \tag{5}$$

Um zu gegebenen Punktintervallen $\boldsymbol{x}, \boldsymbol{y}$ eine Einschließung von $\Re(w) = \arccos(\beta(x,y))$ berechnen zu können, braucht man zunächst eine Einschließung von β, wobei β außerdem noch für die notwendige Entscheidung $|\beta| \leq 0.75$ zur Verfügung stehen muss. Wenn man in (2) die Quadratwurzeln mit Hilfe von `sqrtx2y2(...)` berechnet, so erhält man nur in Extremsituationen einen integer-Überlauf. Lediglich für $x \to 0$ und $|y| \to \infty$ liefert der Quotient einen vorzeitigen integer-Überlauf, den man bei der Berechnung von $\arccos(\beta(x,y))$ wie folgt umgehen kann.

Zunächst ergibt sich aus (2) für $|\beta(x,y)|$ die folgende Oberschranke

$$|\beta(x,y)| \leq \left|\frac{2 \cdot |x|}{2 \cdot \sqrt{y^2}}\right| = \left|\frac{x}{y}\right|, \quad y \neq 0, \tag{6}$$

[29] Auf Seite 175 wird in der komplexen Ebene derjenige Bereich angegeben, der durch $\beta(x,y) \geq 0.75$ charakterisiert ist.

dabei ist die Oberschranke $|x/y|$ umso besser, d.h. umso kleiner, je besser $|y| \gg |x|$ erfüllt ist. Die Idee zur Vermeidung des integer-Überlaufs besteht nun darin, bei hinreichend kleiner Oberschranke $|x/y|$ den Wert $\arccos(\beta(x,y))$ mit der Intervallkonstanten `Pi_lx_interval()` einzuschließen und sonst das Argument β nach (2) dann ohne integer-Überlauf auszuwerten.

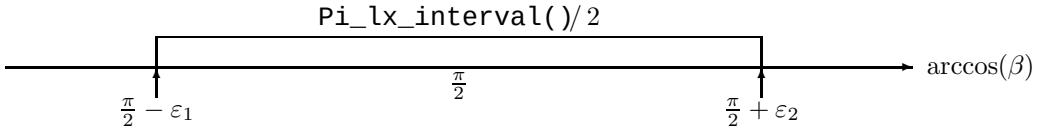


Abbildung 26: Einschließung von $\arccos(\beta)$ für $\beta(x,y) \to 0$.

Wir verlangen also für hinreichend kleines $|x/y|$

$$\arccos(\beta) \in \texttt{Pi\_lx\_interval()}/2, \quad \text{falls } |\beta| \leq \left|\frac{x}{y}\right| \ll +1.$$

Um eine Bedingung für $|x/y|$ angeben zu können, benötigen wir noch eine zusätzliche Abschätzung:

(7) $$\arccos(\beta) = \frac{\pi}{2} - \arcsin(\beta),$$

(8) $$\arcsin(\beta) = \beta \cdot \left(1 + \frac{\beta^2}{6} + \frac{3}{40}\beta^4 + \ldots\right), \quad |\beta| < 1,$$

(9) $$\left(1 + \frac{\beta^2}{6} + \frac{3}{40}\beta^4 + \ldots\right) \leq \frac{1}{1-\beta^2} \leq \frac{1}{1-\beta}, \quad \text{falls } 0 \leq |\beta| < 1.$$

Im Fall $\beta \geq 0$ verlangen wir jetzt nach Abb. 26

(10) $$\frac{\pi}{2} - \varepsilon_1 < \arccos(\beta).$$

Aus (7), (8), (9) folgt dann

$$\arccos(\beta) = \frac{\pi}{2} - \arcsin(\beta) \geq \frac{\pi}{2} - \frac{\beta}{1-\beta} > \frac{\pi}{2} - 2 \cdot \beta, \quad \text{falls } 0 \leq \beta < \frac{1}{2}.$$

Die Forderung (10) ist also erfüllt, wenn gilt

$$\frac{\pi}{2} - \varepsilon_1 < \frac{\pi}{2} - 2 \cdot \beta \quad \Longleftrightarrow \quad \beta < \frac{\varepsilon_1}{2},$$

und wegen $|\beta| = \beta \leq |x/y|$ ist (10) erfüllt, falls

(11) $$\left|\frac{x}{y}\right| < \frac{\varepsilon_1}{2} \quad \Longleftrightarrow \quad |x| < \frac{\varepsilon_1}{2} \cdot |y|.$$

Zusammen mit (11) gilt daher im Fall $0 \leq \beta < \frac{1}{2}$ die Ungleichung: $\pi/2 - \varepsilon_1 < \arccos(\beta)$. Ganz analog findet man nach Abb. 26 für $\beta \leq 0$: Für $|x| < |y| \cdot \varepsilon_2/2$ und $-0.5 < \beta \leq 0$ gilt die Ungleichung $\pi/2 + \varepsilon_2 > \arccos(\beta)$. Wir fassen zusammen:

(12) Im Fall $|x| < \dfrac{\varepsilon_0}{2} \cdot |y|$, mit $\varepsilon_0 := \text{Min}(\varepsilon_1, \varepsilon_2) = 1.45651410012\ldots \cdot 10^{-632}$ gilt:

$$\pi/2 - \varepsilon_1 < \arccos(\beta) < \pi/2 + \varepsilon_2, \text{ d.h. } \arccos(\beta) \in \texttt{Pi\_lx\_interval()}/2.$$

Wir müssen jetzt noch überlegen, wie die Bedingung (12) auf der Maschine möglichst effektiv realisiert werden kann, ohne dass dabei ein vorzeitiger integer-Überlauf eintritt. Wegen $\varepsilon_0/2 = 2^{-2099.9160...} > 2^{-2100}$ ist (12) erfüllt, falls

$$|x| < 2^{-2100} \cdot |y|. \tag{13}$$

Wenn also (13) erfüllt ist, so gilt: $\arccos(\beta) \in$ `Pi_lx_interval()`$/2$. Da das Produkt rechts in (13) nicht für beliebiges y ohne integer-Überlauf auswertbar ist, berechnen wir zunächst eine Oberschranke für $|x/y|$, ohne dabei den Quotienten wirklich auszuwerten. Mit den Bezeichnungen von Seite 146, 147, gelten die folgenden Abschätzungen:

$$|x| < 2^{exx+exxl+1}, \quad |y| > 2^{exy+exyl-2} \quad \text{und damit}$$
$$\left|\frac{x}{y}\right| < 2^{exx+exxl-exy-exyl+3}.$$

Damit ist dann (13) erfüllt, wenn gilt:

$$\begin{aligned} & exx + exxl - exy - exyl + 3 < -2100 \\ \iff \quad & exx + exxl - exy - exyl < -2103. \end{aligned} \tag{14}$$

Damit die linke Seite in (14) ohne einen integer-Überlauf berechnet werden kann, ist die Summe links mit Hilfe von `real`-Variablen auszuwerten. Bei der Implementierung wird $\beta(x, y)$ mit Hilfe der Funktion `BETA_xy(...)` eingeschlossen, wobei im Fall (14) $\beta = 0$ gesetzt wird. Damit vermeidet man einen integer-Überlauf bei der notwendigen Berechnung von $\beta(x, y)$ zur Entscheidung von $|\beta(x, y)| \leq 0.75$, und $\arccos(\beta)$ wird dabei durch die C-XSC Intervallkonstante `Pi_lx_interval()`$/2$ garantiert eingeschlossen.

Im Fall $0.75 \leq |\beta(x, y)| \leq 1$ muss jetzt noch $\arccos(\beta)$ nach (3) bzw. (4) ausgewertet werden, wobei $\delta(x, y)$ definiert ist durch $\delta(x, y) := 1 - |\beta(x, y)|$ und $|\beta(x, y)|$ durch $x \geq 0$ realisiert wird. Um $\arcsin(\sqrt{\delta \cdot (2 - \delta)})$ in ausreichender Genauigkeit berechnen zu können, müssen in den Teilbereichen $0.75 \leq x < 1$, $x = 1$ und $x > 1$ jeweils verschiedene Terme für $\delta(x, y)$ angegeben werden.

Wir beginnen mit dem Bereich $x > 1$ und $0.75 \leq \beta(x, y) \leq +1$, in dem zur hochgenauen Einschließung von δ bzw. $\sqrt{\delta}$ und zur Vermeidung eines vorzeitigen integer-Überlaufs der größte Aufwand erforderlich ist. Mit

$$2 \cdot \beta = \sqrt{(x+1)^2 + y^2} - \sqrt{(x-1)^2 + y^2} \quad \text{und}$$
$$2 \cdot \delta = 2 - 2 \cdot \beta = 2 + \sqrt{(x-1)^2 + y^2} - \sqrt{(x+1)^2 + y^2}$$

erhält man zunächst durch Erweitern mit $N := \{2 + \sqrt{(x-1)^2 + y^2}\} + \sqrt{(x+1)^2 + y^2}$

$$2 \cdot \delta = \frac{4 \cdot \{\sqrt{(x-1)^2 + y^2} + (1 - x)\}}{N} \tag{15}$$

und durch nochmaliges Erweitern mit $\sqrt{(x-1)^2+y^2}-(1-x)$ erhält man schließlich

$$\delta = \frac{2\cdot y^2}{(2+\sqrt{(x-1)^2+y^2}+\sqrt{(x+1)^2+y^2})\cdot(\sqrt{(x-1)^2+y^2}+(x-1))}, \tag{16}$$

$$\sqrt{\delta} = \frac{\sqrt{2}\cdot|y|}{\sqrt{2+\sqrt{(x-1)^2+y^2}+\sqrt{(x+1)^2+y^2}}\cdot\sqrt{\sqrt{(x-1)^2+y^2}+(x-1)}}. \tag{17}$$

Um $\arcsin(\sqrt{\delta\cdot(2-\delta)})$ für Punktintervall-Argumente x, y einzuschließen, ist (16) wegen des Zählers $2y^2$ wegen des damit verbundenen integer-Überlaufs für $|y|\to\infty$ oder $|y|\to 0$ nicht geeignet. Aber auch Umformungen, wie z.B.

$$\delta = \frac{2}{\left(\frac{2}{y}+\sqrt{1+\left(\frac{x-1}{y}\right)^2}+\sqrt{1+\left(\frac{x+1}{y}\right)^2}\right)\cdot\left(\sqrt{1+\left(\frac{x-1}{y}\right)^2}+\frac{x-1}{y}\right)}$$

lösen die integer-Überlaufproblematik nicht!

Besser geeignet ist dagegen die Auswertung von $\sqrt{\delta(x,y)}$ nach (17). Dabei kann der ganze Nenner in (17) auch in den Fällen $y\to 0\ \wedge\ x\to\infty$ oder $y\to\infty\ \wedge\ x\to\infty$ fast immer ohne vorzeitigen integer-Überlauf berechnet werden. Für $y\to 0\ \wedge\ x\to\infty$ liefert der Quotient in (17) jedoch einen integer-Überlauf, der durch die folgenden Überlegungen vermieden wird. Wir benötigen dazu für $\sqrt{\delta}$ zunächst eine geeignete Unterschranke, mit deren Hilfe derjenige Bereich abgegrenzt werden kann, in dem $\sqrt{\delta}\cdot\sqrt{2-\delta}$ ohne integer-Überlauf berechenbar ist. Für $x\geq 0$ und damit auch für $x>1$ gilt: $(x-1)^2\leq(x+1)^2$, d.h. $\sqrt{(x-1)^2+y^2}\leq\sqrt{(x+1)^2+y^2}$, und daraus folgt:

$$\begin{aligned}\gamma_1 &:= 2+\sqrt{(x-1)^2+y^2}+\sqrt{(x+1)^2+y^2}\leq 2+2\cdot\sqrt{(x+1)^2+y^2},\\ \gamma_2 &:= \sqrt{(x-1)^2+y^2}+(x-1)\leq\sqrt{(x+1)^2+y^2}+(x+1).\end{aligned}$$

$$\begin{aligned}\gamma_1\cdot\gamma_2 &\leq 2\cdot\left(1+\sqrt{(x+1)^2+y^2}\right)\cdot\left(\sqrt{(x+1)^2+y^2}+(x+1)\right)\\ &\leq 4\cdot\left(1+\sqrt{(x+1)^2+y^2}\right)\cdot\sqrt{(x+1)^2+y^2}\\ &\leq 4\cdot 2\cdot\sqrt{(x+1)^2+y^2}\cdot\sqrt{(x+1)^2+y^2}=8\cdot\left\{(x+1)^2+y^2\right\},\end{aligned}$$

und daraus erhält man mit (17) die gesuchte Unterschranke

$$\frac{|y|}{2\cdot\sqrt{(x+1)^2+y^2}}\leq\sqrt{\delta(x,y)}. \tag{18}$$

Wir beweisen jetzt noch die Ungleichung

$$|y|<x+1. \tag{19}$$

Wegen $0.75\leq\beta\leq+1$ gilt nach der Abbildung von Seite 176

$$|y|\leq\frac{1}{12}\sqrt{112x^2-63}\quad\text{und wegen}\quad\frac{1}{12}\sqrt{112x^2-63}<x+1$$

folgt sofort die Behauptung.

Um in (17) für $y \to 0 \ \wedge \ x \to \infty$ den integer-Überlauf zu vermeiden, verlangen wir für ein geeignetes ε_1 nach (18)

$$\frac{|y|}{2 \cdot \sqrt{(x+1)^2 + y^2}} > \varepsilon_1. \tag{20}$$

Wegen (19) und $x + 1 < 2x$ gilt:

$$\frac{|y|}{2 \cdot \sqrt{(x+1)^2 + y^2}} > \frac{|y|}{2 \cdot \sqrt{2 \cdot (x+1)^2}} = \frac{|y|}{2 \cdot \sqrt{2} \cdot (x+1)} > \frac{|y|}{4 \cdot \sqrt{2} \cdot x} > \frac{|y|}{8x},$$

und (20) ist erfüllt, wenn gilt:

$$\frac{|y|}{8x} > \varepsilon_1 \quad \Longleftrightarrow \quad |y| > (8\varepsilon_1) \cdot x, \tag{21}$$

dabei kann die rechte Seite von (21) wegen $x > 1$ problemlos auf der Maschine ausgewertet werden.

Zusammenfassung:

Im Fall $|y| > \varepsilon \cdot x$ kann mit einem geeigneten ε der Ausdruck $\sqrt{\delta}$ nach (21) ohne integer-Überlauf ausgewertet werden. Mit $\varepsilon := 2^{-2147482624}$ erreicht man, dass auch das ganze Argument $\sqrt{\delta} \cdot \sqrt{2 - \delta}$ ohne integer-Überlauf berechnet werden kann. Die Differenz $2 - \delta$ muss jedoch umgeformt werden in

$$2 - \delta = (\sqrt{2} - \sqrt{\delta}) \cdot (\sqrt{2} + \sqrt{\delta}),$$

da $\delta = (\sqrt{\delta})^2$ zu einem integer-Überlauf führen würde. $\arcsin(\sqrt{\delta \cdot (2 - \delta)})$ ist also wie folgt auszuwerten:

$$\arcsin(\sqrt{\delta \cdot (2 - \delta)}) = \arcsin\left(\sqrt{\delta} \cdot \sqrt{(\sqrt{2} - \sqrt{\delta}) \cdot (\sqrt{2} + \sqrt{\delta})}\right). \tag{22}$$

Es bleibt jetzt noch der Fall $|y| \leq \varepsilon \cdot x$, und es kann daher nicht mehr garantiert werden, dass $\sqrt{\delta} \cdot \sqrt{2 - \delta}$ ohne integer-Überlauf auswertbar ist. Es soll aber auch in diesem Fall eine garantierte Einschließung für $\arcsin(\sqrt{\delta \cdot (2 - \delta)})$ berechnet werden können. Wegen $\delta = 1 - \beta \ \wedge \ 0 \leq \beta \leq 1$ kann für $\arcsin(\sqrt{\delta \cdot (2 - \delta)})$ zunächst die folgende Einschließung angegeben werden:

$$\arcsin(\sqrt{\delta \cdot (2 - \delta)}) \in [0, \sqrt{2 \cdot \delta}],$$

wobei jedoch $\sqrt{2 \cdot \delta}$ nicht nach (17) ohne integer-Überlauf berechnet werden kann. Es ist also notwendig, für $\sqrt{2 \cdot \delta}$ eine geeignete Oberschranke anzugeben. Die beiden Faktoren im Nenner von (17) lassen sich wie folgt abschätzen:

$$\sqrt{2 + \sqrt{(x-1)^2 + y^2} + \sqrt{(x+1)^2 + y^2}} \geq \sqrt{2 + 2\sqrt{(x-1)^2 + y^2}} \geq \sqrt{2x},$$
$$\sqrt{\sqrt{(x-1)^2 + y^2} + (x-1)} \geq \sqrt{2(x-1)}.$$

Nach (17) gilt dann

$$\sqrt{2\cdot\delta} \le \frac{|y|}{\sqrt{x(x-1)}} \quad \text{und damit}$$
$$\arcsin(\sqrt{\delta\cdot(2-\delta)}) \in [0, \arcsin(|y|/\sqrt{x(x-1)})].$$

Für $0 \le t \ll +1$ gilt $\arcsin(t) < 2t$, d.h.

$$\arcsin(\sqrt{\delta\cdot(2-\delta)}) \in \left[0, \frac{2|y|}{\sqrt{x(x-1)}}\right]. \tag{23}$$

Für $x \to \infty$ liefert $x(x-1)$ einen integer-Überlauf, der wie folgt vermieden wird. Im Fall $x \ge 2$ gilt $x - 1 \ge x/2$ und damit

$$\frac{2|y|}{\sqrt{x(x-1)}} \le 2\sqrt{2}\cdot\frac{|y|}{x} \le \frac{4\cdot|y|}{x}, \quad \text{falls } x \ge 2. \tag{24}$$

Der letzte Ausdruck rechts in (24) kann ohne Multiplikation und Division weiter nach oben abgeschätzt werden. Mit den Abschätzungen von Seite 146, 147 gilt:

$$\frac{4\cdot|y|}{x} < 2^{exy-exx+exyl-exxl+5} = 2^{ex}.$$

Damit ist dann 2^{ex} eine etwas grobe aber garantierte Oberschranke, und es gilt:

$$\arcsin(\sqrt{\delta\cdot(2-\delta)}) \in [0, 2^{ex}], \quad \text{falls } |y| \le \varepsilon\cdot x \ \wedge\ x \ge 2. \tag{25}$$

Bleibt noch der Fall $|y| \le \varepsilon\cdot x \ \wedge\ 1 < x < 2$.

Bei der intervallmäßigen Auswertung der Oberschranke $2\cdot|y|/\sqrt{x(x-1)}$ in (23) kann für $x \to +1$ der Fall $\texttt{Inf(sqrt}(x \diamond (x \diamond 1)) = 0$ auftreten. Um die damit verbundene Division durch Null zu vermeiden, wählen wir wegen $\texttt{Inf}(x \diamond 1) \ge 2^{-2097}$ die folgende Abschätzung

$$\frac{2\cdot|y|}{\sqrt{x(x-1)}} \le \frac{2|y|}{\sqrt{x\cdot 2^{-2097}}} = \frac{2^{3001}\cdot|y|}{2^{3000}\cdot\sqrt{x\cdot 2^{-2097}}}, \tag{26}$$

wobei die Skalierung mit 2^{+3000} einen vorzeitigen integer-Überlauf bei der Berechnung des Quotienten rechts in (26) verhindert. Im Fall $\texttt{Inf(sqrt}(x \diamond (x \diamond 1)) > 0$ kann der Ausdruck links in (26) direkt ausgewertet werden, womit dann der Fall $x > 1 \wedge \beta \ge 0.75$ vollständig behandelt ist.

Im Fall $x = 1 \ \wedge\ \beta \ge 0.75$ erhält man direkt aus (14) für $\delta(x,y)$ den Ausdruck

$$\delta(x,y) = \frac{2\cdot|y|}{2+|y|+\sqrt{2^2+y^2}}.$$

Wertet man $\sqrt{2^2+y^2}$ mit Hilfe der C-XSC Funktion `sqrtx2y2(...)` intervallmäßig aus, so kann $\delta(x,y)$ in hoher Genauigkeit eingeschlossen werden.

Im letzten Fall $0.75 \leq x < +1 \ \wedge \ \beta \geq 0.75$ erhält man ebenfalls direkt aus (14) für $\delta(x,y)$ den Ausdruck

$$\delta(x,y) = \frac{2 \cdot \left(\sqrt{(1-x)^2+y^2} + (1-x)\right)}{2 + \sqrt{(1-x)^2+y^2} + \sqrt{(1+x)^2+y^2}}.$$

Wertet man auch hier die auftretenden Quadratwurzeln mit Hilfe der C-XSC Funktion `sqrtx2y2(...)` intervallmäßig aus, so kann $\delta(x,y)$ ebenfalls in hoher Genauigkeit eingeschlossen werden. In den beiden letzten Fällen kann $\arccos(\beta)$ nach (3) direkt mit Hilfe von $\arcsin(\sqrt{\delta \cdot (2-\delta)})$ ausgewertet werden, ohne den sehr aufwendigen Ausdruck $\sqrt{\delta} \cdot \sqrt{2-\delta} = \sqrt{\delta} \cdot \sqrt{(\sqrt{2}-\sqrt{\delta}) \cdot (\sqrt{2}+\sqrt{\delta})}$ nach (22) benutzen zu müssen.

6.2.13.1 Testrechnungen Nach Angabe der Algorithmen für den Real- und Imaginärteil der komplexen $\arccos(z)$-Funktion erhält man mit Hilfe der folgenden Tabelle einen groben Überblick über deren großen Auswertebereich. Die Angaben in der Tabelle beziehen sich auf die Einschließung der $\arccos(z)$-Funktion nur für komplexe Punktintervalle $\boldsymbol{z = x + i \cdot y}$, wobei die reellen Punktintervalle $\boldsymbol{x}$ und $\boldsymbol{y}$ vom Typ `lx_interval` sind.

Beachten Sie bitte, dass bei der $\arccos(z)$-Funktion die komplexe Ebene längs der reellen Achse aufgeschnitten ist von $-\infty$ bis -1 und von $+1$ bis $+\infty$. Die Definition der Funktionswerte auf den Verzweigungsschnitten kann in der Abbildung auf Seite 189 anhand der fettgedruckten Pfeile abgelesen werden.

Auswertebereiche von $\arccos(z)$, $z = x + i \cdot y$

x	y	Anmerkung
0	$2^{-2147482626} \leq y \leq 2^{+2147482625}$	optimale Einschließung
$2^{-2147482625}$	$2^{-2147482626} \leq y \leq 2^{+2147482625}$	optimale Einschließung
0.5	$2^{-2147482626} \leq y \leq 2^{+2147482625}$	optimale Einschließung
$1 - 2^{-2095}$	$2^{-2147483645} \leq y \leq 2^{+2147482625}$	optimale Einschließung
1	$2^{-2147482625} \leq y \leq 2^{+2147482625}$	optimale Einschließung
$1 + 2^{-2095}$	$2^{-2147482628} \leq y \leq 2^{-2147482624}$	grobe Einschlg. des Realteils
$1 + 2^{-2095}$	$2^{-2147482624} < y \leq 2^{+2147482625}$	optimale Einschließung
1.5	$2^{-2147482628} \leq y \leq 2^{-2147482624}$	grobe Einschlg. des Realteils
1.5	$2^{-2147482624} < y \leq 2^{+2147482625}$	optimale Einschließung
$2^{+2147482625}$	$2^{-2147483647} \leq y \leq 2^{+1}$	grobe Einschlg. des Realteils
$2^{+2147482625}$	$2^{+1} < y \leq 2^{+2147482625}$	optimale Einschließung

Tabelle 4: $\arccos(z)$, mögliche Auswertebereiche

Im **1. Beispiel** liefert das folgende Programm `lx_test74` für das komplexe Argument $z = 2^{+2147482625} + 2.125 \cdot i,\ i = \sqrt{-1}$ eine garantierte und sehr enge Einschließung für den Funktionswert $f(z) = \arccos(z)$.

```
// Programm  lx_test74.cpp;
// Zum Test von acos(z);

#include <iostream>
#include "lx_cimath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 30;
    lx_cinterval Z,W;
    lx_interval X,Y;
    string s;

    X = lx_interval(2147482625,l_interval(1));
    Y = lx_interval(0,l_interval(2.125));

    Z = lx_cinterval(X,Y);
    W = acos(Z);

    cout << SetDotPrecision(16*stagprec,16*stagprec+5)
         << Scientific;
    s << W;
    cout << "acos(Z) = " << s << endl;
}
```

Die folgende Ausgabe von `W` über die Zeichenkette `s` liefert die folgende Einschließung des Real- und Imaginärteils von $\arccos(z)$ in hoher Genauigkeit:

$$\Re(f(z)) \in 2^{-2147483646} \cdot [\underbrace{4.7751223894780\ldots447999\ldots999}_{476\text{ korrekte Dez.-Ziffern}}598\ldots \cdot 10^{+307},$$
$$4.7751223894780\ldots448000\ldots000397\ldots \cdot 10^{+307}].$$

$$\Im(f(z)) \in 2^{-992} \cdot (\underbrace{-6.230326674658685\ldots1004698794942460}_{480\text{ korrekte Dez.-Ziffern}}{}^{122\ldots}_{223\ldots}) \cdot 10^{+307}.$$

Zeigen Sie mit dem gleichen Programm, dass mit $z = 2^{+2147482625} + 2 \cdot i$ der Realteil von $\arccos(z)$ nur noch grob eingeschlossen werden kann, vgl. Sie dazu auch die Tabelle 4 auf Seite 196.

Im **2. Beispiel** liefert das folgende Programm `lx_test75` für das komplexe Argument $z = 1 + 2^{-2095} + i \cdot 2^{+2147482625}$ eine garantierte und sehr enge Einschließung für den Funktionswert $f(z) = \arccos(z)$.

```
// Programm  lx_test75.cpp;
// Zum Test von acos(z);

#include <iostream>
#include "lx_cimath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 30;
    lx_cinterval Z,W;
    lx_interval X,Y;
    string s;

    X = lx_interval(0,l_interval(1)) +
        lx_interval(-2095,l_interval(1));
    Y = lx_interval(+2147482625,l_interval(1));

    Z = lx_cinterval(X,Y);
    W = acos(Z);

    cout << SetDotPrecision(16*stagprec,16*stagprec+10)
         << Scientific;

    s << W;
    cout << "acos(Z) = " << s << endl;
}
```

Die folgende Ausgabe von `W` über die Zeichenkette `s` liefert die folgende Einschließung des Real- und Imaginärteils von $\arccos(z)$ in hoher Genauigkeit:

$$\Re(f(z)) \in 2^{-1022} \cdot \underbrace{7.0595244323653213\ldots9520740025264746}_{\text{483 korrekte Dez.-Ziffern}} {}^{511\ldots}_{195\ldots} \cdot 10^{+307}.$$

$$\Im(f(z)) \in 2^{-992} \cdot (\underbrace{-6.23032667465868\ldots1004698794942460}_{\text{480 korrekte Dez.-Ziffern}} {}^{122\ldots}_{223\ldots}) \cdot 10^{+307}.$$

Zeigen Sie mit dem gleichen Programm, dass mit $z = 1 + 2^{-2095} + i \cdot 2^{+2147482626}$ ein integer-Überlauf erfolgt; vgl. Sie dazu auch die Tabelle 4 auf Seite 196.

Im **3. Beispiel** liefert das folgende Programm `lx_test76` für das komplexe Argument $z = 2.501 + i \cdot 1.001 \cdot 10^{+300000}$ eine garantierte und sehr enge Einschließung für den Funktionswert $f(z) = \arccos(z)$, dabei ist zu beachten, dass Real- und Imaginärteil von z nicht im Staggered Format darstellbar sind und daher nicht wie in den vorhergehenden Beispielen durch Punktintervalle eingeschlossen werden können. Die Einschließung von $\Re(z)$ und $\Im(z)$ erfolgt durch das folgende Programm mit 324 bzw. 300 korrekten Dezimalstellen, so dass die Einschließung von $\arccos(z)$ nicht mit höherer Genauigkeit erfolgen kann.

```
// Programm  lx_test76.cpp;
// Zum Test von acos(z);

#include <iostream>
#include "lx_cimath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 30;
    lx_cinterval Z,W;
    lx_interval X,Y;

    X = lx_interval(0,"[2.501,2.501]");
    Y = lx_interval(300000,"[1.001,1.001]");

    Z = lx_cinterval(X,Y);
    W = acos(Z);

    cout << SetDotPrecision(16*stagprec,16*stagprec+5)
         << Scientific;
    cout << "acos(Z) = " << W << endl;
}
```

Die folgende Ausgabe von `W` in lesbarer dezimaler Form liefert die Einschließung des Real- und Imaginärteils von $f(z) = \arccos(z)$ in hoher Genauigkeit:

$$\Re(f(z)) \in 10^{-1} \cdot \underbrace{1.5707963267948966192313 \ldots 8036301245706368}_{\text{302 korrekte Dez.-Ziffern}} \begin{matrix} 761\ldots \\ 478\ldots \end{matrix} \cdot 10^{+1}.$$

$$\Im(f(z)) \in 10^{4} \cdot (\underbrace{-6.9077622204489459823424 \ldots 2496809307736096}_{\text{301 korrekte Dez.-Ziffern}} \begin{matrix} 348\ldots \\ 420\ldots \end{matrix}) \cdot 10^{1}.$$

Eine mit *Mathematica* unter hohem laufzeitmäßigem Aufwand berechnete Näherung für $f(z)$ wird durch das obige Intervall W eingeschlossen.

6.2.14 Die Funktion $\arctan(z)$

Die C-XSC Funktion

```
lx_cinterval atan(const lx_cinterval& z);
```

berechnet für ein gegebenes rechteckiges komplexes Eingangsintervall $\boldsymbol{z}$ eine garantierte Einschließungen des Intervalls

$$f(\boldsymbol{z}) = \arctan(\boldsymbol{z}), \quad \boldsymbol{z} = \boldsymbol{x} + i \cdot \boldsymbol{y}, \;\; \boldsymbol{x}, \boldsymbol{y} \in IR, \;\; i = \sqrt{-1}.$$

$\arctan(z), z \in \mathbb{C}$, bedeutet dabei den Hauptwert der komplexwertigen $\arctan$-Funktion, wobei die Verzweigungsschnitte auf der imaginären Achse liegen von $+i$ bis $+i \cdot \infty$ bzw. von $-i \cdot \infty$ bis $-i$. Die beiden Singularitäten liegen bei $\pm i$. Das komplexe Eingangsintervall $\boldsymbol{z}$ der `atan`-Funktion darf keinen Punkt eines Verzweigungsschnitts enthalten. Beispielsweise sind die Intervalle $\boldsymbol{z}_1 = [0,0] + i \cdot [-2,-1]$ oder $\boldsymbol{z}_2 = [0, 10^{-20}] + i \cdot [2,3]$ nicht erlaubt. Nach der Bedingung (4) von Seite 7 ist außerdem zu beachten, dass die relativen Durchmesser der Intervalle für den Real- und Imaginärteil nicht zu groß sein dürfen, so dass auch schon deshalb $\boldsymbol{z}_2$ nicht als Eingangsintervall gewählt werden sollte.

6.2.14.1 Der Realteil des inversen Tangens Nach W. Krämer ist zur Einschließung des Realteils $\Re(f(\boldsymbol{z}))$ die folgende Funktion

$$g(x,y) := \frac{1}{2} \arctan \frac{2x}{1 - x^2 - y^2}, \quad x, y \in \mathbb{R} \tag{1}$$

auf den Extremalpunkten des rechteckigen, komplexen Eingangsintervalls $\boldsymbol{z} = \boldsymbol{x} + i \cdot \boldsymbol{y}$ intervallmäßig auszuwerten, wobei $\boldsymbol{x}$ ein Maschinen-Punktintervall und $\boldsymbol{y} = [y_1, y_2]$, mit $y_1 \leq y_2$ ein echtes aber stets sehr schmales Maschinenintervall ist. Genauere Einzelheiten findet man in [15]. In [27] ist der Algorithmus zur intervallmäßigen Auswertung ausführlich beschrieben, so dass hier nur die Einschließung von Zähler und Nenner der Funktion

$$\alpha(\boldsymbol{x}, \boldsymbol{y}) := \frac{\boldsymbol{x}}{1 - \boldsymbol{x}^2 - \boldsymbol{y}^2} \tag{2}$$

mit der erweiterten Staggered Intervall-Arithmetik zu diskutieren ist. Dabei soll ein vorzeitiger integer-Überlauf bei Auswertung der Quadrate $\boldsymbol{x}^2, \boldsymbol{y}^2$ verhindert werden, um die inverse Tangensfunktion in einem möglichst weiten Definitionsbereich auf der Maschine auswerten zu können. Es sei nochmals betont, dass in (2) nicht der ganze Quotient, sondern nur Zähler und Nenner einzuschließen sind. Die Einschließung der reellen $\arctan$-Funktion in (1) erfolgt dann mit Hilfe der internen C-XSC Funktion

```
lx_interval Atan( const lx_interval& y, const lx_interval& x )
```

die den inversen Tangens von `y`/`x` berechnet, wobei die Intervalle `y`,`x` Punktintervalle sein müssen.

Auf der folgenden Seite werden einige Fälle angegeben, in denen Zähler und Nenner von α ohne integer-Überlauf eingeschlossen werden können.

Im Fall $\boldsymbol{x} = 0$ lautet der Nenner $1 - \boldsymbol{y}^2$. Da im Eingangsintervall $\boldsymbol{z}$ keine Punkte des Verzweigungsschnitts liegen dürfen, muss gelten $\mathtt{Sup}(\mathtt{abs}(\boldsymbol{y})) < 1$, so dass im Intervall $1 - \boldsymbol{y}^2$ nur positive Zahlen liegen und der Nenner damit auf 1 gesetzt werden kann.

Im Fall $\boldsymbol{y} = 0$ gilt $\alpha = \boldsymbol{x}/(1 - \boldsymbol{x}^2)$, und Zähler und Nenner von α werden wie folgt definiert:

$$\alpha = \frac{\boldsymbol{x}}{1 - \boldsymbol{x}^2} = \begin{cases} \dfrac{1}{1/x - x} & \text{, falls } |\boldsymbol{x}| \to \infty, \\ \dfrac{x}{(1 - \boldsymbol{x}) * (1 + \boldsymbol{x})} & \text{, falls } \boldsymbol{x} \to 0. \end{cases}$$

Im Fall $\boldsymbol{y} \neq 0$, $|\boldsymbol{x}| \to \infty$, $|\boldsymbol{y}| \to \infty$ wird $\alpha(\boldsymbol{x}, \boldsymbol{y})$ mit $2^s, s \leq 0$, so erweitert

$$\alpha = \frac{\boldsymbol{x}}{1 - \boldsymbol{x}^2 - \boldsymbol{y}^2} = \frac{\boldsymbol{x} \cdot 2^{2s}}{(2^s - \boldsymbol{x} \cdot 2^s)(2^s + \boldsymbol{x} \cdot 2^s) - (\boldsymbol{y} \cdot 2^s)(\boldsymbol{y} \cdot 2^s)},$$

dass ein integer-Überlauf im Zähler und Nenner nicht mehr auftreten kann. Dazu wird $s \in \mathbb{Z}$ bestimmt durch

$$s + \mathtt{max}(\mathtt{ex\_x}, \mathtt{ex\_y}) = c1 := 1073740000, \quad s \leq 0,$$

wobei $\mathtt{ex\_x}, \mathtt{ex\_y}$ die Zweier-Exponenten von $\boldsymbol{x}$ und $\boldsymbol{y}$ sind.

Im Fall $\boldsymbol{y} \neq 0$, $|\boldsymbol{x}| \to \infty$, $|\boldsymbol{y}| \to 0$ wird $\alpha(\boldsymbol{x}, \boldsymbol{y})$ folgendermaßen umgeformt:

$$\alpha = \frac{\boldsymbol{x}}{1 - \boldsymbol{x}^2 - \boldsymbol{y}^2} = \frac{1}{\dfrac{(1 - \boldsymbol{y})(1 + \boldsymbol{y})}{\boldsymbol{x}} - \boldsymbol{x}},$$

wobei jetzt der Nenner rechts ohne vorzeitigen integer-Überlauf eingeschlossen werden kann.

Im Fall $\mathtt{sqr}(\boldsymbol{y}) = +1$, d.h. $\boldsymbol{y} = [1, 1]$ oder $\boldsymbol{y} = [-1, -1]$ und zusätzlich $|\boldsymbol{x}| \to \infty$ kann $\alpha(\boldsymbol{x}, \boldsymbol{y})$ wie folgt umgeformt werden:

$$\alpha = \frac{\boldsymbol{x}}{-\boldsymbol{x}^2} = \frac{-1}{x},$$

so dass Zähler und Nenner problemlos eingeschlossen werden können.

Im Fall $|\boldsymbol{x}| \to +\infty$ setzen wir voraus, dass $\mathtt{sqr}(\boldsymbol{y})$ jetzt ohne integer-Überlauf berechnet werden kann. Es gilt mit $|\boldsymbol{x}| = \mathtt{abs}(\boldsymbol{x})$

$$\alpha = \frac{\boldsymbol{x}}{1-\boldsymbol{y}^2-\boldsymbol{x}^2} = \frac{\mathtt{sign}(\boldsymbol{x})\cdot|\boldsymbol{x}|}{1-\boldsymbol{y}^2-\boldsymbol{x}^2} = \frac{-\mathtt{sign}(\boldsymbol{x})}{\dfrac{\boldsymbol{y}^2-1}{|\boldsymbol{x}|}+|\boldsymbol{x}|}. \tag{3}$$

Definiert man $R := \mathtt{Sup}(\mathtt{abs}(\boldsymbol{y}^2-1))$, dann erhält man mit $\varepsilon_0 := 2^{-2097}$ unter der Voraussetzung

$$\frac{R}{|\boldsymbol{x}|}+|\boldsymbol{x}| < |\boldsymbol{x}|\cdot(1+\varepsilon_0) \quad\Longleftrightarrow\quad R < |\boldsymbol{x}|^2\cdot\varepsilon_0 \tag{4}$$

die Einschließung:

$$\frac{\boldsymbol{y}^2-1}{|\boldsymbol{x}|}+|\boldsymbol{x}| \subset |\boldsymbol{x}|\cdot[1-\varepsilon_0, 1+\varepsilon_0], \tag{5}$$

wobei die Werte $1-\varepsilon_0$ und $1+\varepsilon_0$ durch die Punktintervalle `One_m_lx_interval()` bzw. `One_p_lx_interval()` eingeschlossen werden. Die Bedingung (4) ist erfüllt, wenn eine Oberschranke von R kleiner ist als eine Unterschranke von $|\boldsymbol{x}|^2\cdot\varepsilon_0$. Es ist zu beachten, dass zwar R, nicht aber $|\boldsymbol{x}|^2\cdot\varepsilon_0$ auf der Maschine auswertbar ist, so dass (4) über die Zweier-Exponenten zu realisieren ist.

Zur Berechnung einer Unterschranke von $|\boldsymbol{x}|^2\cdot\varepsilon_0$ gilt mit den Bezeichnungen von Seite 146, 147:

$$\begin{gathered} 2^{exx+exxl-2} \le |\boldsymbol{x}|, \quad \text{und daraus folgt} \\ 2^{2\cdot(exx+exxl)-2101} \le |\boldsymbol{x}|^2\cdot\varepsilon_0. \end{gathered} \tag{6}$$

Zur Berechnung einer Oberschranke von R gilt analog zu Seite 147 mit

```
exy = expo(R); exyl = expo_gr(lr_part(R));
```

$$R < 2^{exy+exyl+1}. \tag{7}$$

Nach (6) und (7) ist die Bedingung (4) erfüllt, falls

$$2^{exy+exyl+1} < 2^{2\cdot(exx+exxl)-2101} \quad\Longleftrightarrow\quad exy+exyl < 2\cdot(exx+exxl)-2102, \tag{8}$$

und die Einschließung erfolgt dann nach (5), sonst kann die linke Seite von (5) direkt eingeschlossen werden. Die Einschließung des Zählers $-\mathtt{sign}(\boldsymbol{x})$ rechts in (3) bereitet keinerlei Probleme.

Wir betrachten jetzt den Fall: $|\boldsymbol{y}| \to \infty$, aber $|\boldsymbol{x}| \not\to \infty$. Um einen integer-Überlauf bei der Auswertung von $\boldsymbol{y}^2$ zu vermeiden, wird $\alpha(\boldsymbol{x},\boldsymbol{y})$ mit -2^{2s} erweitert:

$$\alpha(\boldsymbol{x},\boldsymbol{y}) = \frac{\boldsymbol{x}}{1-\boldsymbol{x}^2-\boldsymbol{y}^2} = \frac{-(\boldsymbol{x}\cdot 2^s)\cdot 2^s}{\{(\boldsymbol{x}-1)\cdot(\boldsymbol{x}+1)\}\cdot 2^{2s}+(\boldsymbol{y}\cdot 2^s)^2}. \tag{9}$$

Mit den Anweisungen

```
exy = expo(y); exyl = expo_gr(Sup(li_part(abs(y))));
```

ist $s \in \mathbb{Z}$ bestimmt durch

$$exy + exyl + s = c1 = 1073740000; \quad \Longleftrightarrow \quad s = c1 - (exy + exyl), \quad s \leq 0.$$

Der Zähler $-(\boldsymbol{x} \cdot 2^s) \cdot 2^s$ in (9) kann durch zweimalige Anwendung der C-XSC Funktion `times2pown_neg(x,s)` garantiert eingeschlossen werden, vgl. dazu auch Seite 20. Um im Nenner von (9) auch die Summe $Ne := \{(\boldsymbol{x}-1)\cdot(\boldsymbol{x}+1)\}\cdot 2^{2s} + (\boldsymbol{y}\cdot 2^s)^2$ ohne integer-Überlauf berechnen zu können, benötigen wir zunächst den Ausdruck

$$R := \mathtt{Sup}(\mathtt{abs}((\boldsymbol{x}-1)\cdot(\boldsymbol{x}+1)),$$

der auch für $\boldsymbol{x} \to 0$ oder $\boldsymbol{x} \to \pm 1$ problemlos berechnet werden kann. Außerdem gilt noch $R = 0 \iff \boldsymbol{x} = \pm 1$. Im Fall $R = 0$ erhält man

$$\alpha(\boldsymbol{x}, \boldsymbol{y}) = \frac{-(\boldsymbol{x}\cdot 2^s)\cdot 2^s}{(\boldsymbol{y}\cdot 2^s)^2},$$

wobei jetzt Zähler und Nenner ohne integer-Überlauf berechnet werden können.

Sei $R > 0$. In $Ne := \{(\boldsymbol{x}-1)\cdot(\boldsymbol{x}+1)\}\cdot 2^{2s} + (\boldsymbol{y}\cdot 2^s)^2$ ist jetzt zwar der erste Summand durch zweimalige Anwendung von `times2pown_neg({...},s)` stets berechenbar, aber die Summe selbst liefert einen integer-Überlauf, wenn der erste Summand betragsmäßig zu klein ist und sein einschließendes Intervall die Zahl Null als Randpunkt besitzt. Diesen integer-Überlauf kann man vermeiden, wenn man mit $\beta := (\boldsymbol{y}\cdot 2^s)^2$ den Nenner Ne folgendermaßen einschließt:

$$Ne \subset \beta \diamond [1-\varepsilon_0, 1+\varepsilon_0], \quad \varepsilon_0 := 2^{-2097}, \tag{10}$$

wobei das Intervall $[1-\varepsilon_0, 1+\varepsilon_0]$ in C-XSC realisiert wird durch:

```
lx_interval(Inf(One_m_lx_interval()),Sup(One_p_lx_interval()));
```

Um (10) garantieren zu können, müssen folgende Bedingungen erfüllt sein:

1. $\mathtt{Sup}(\beta)\cdot(1+\varepsilon_0) > \mathtt{Sup}(\beta) + R\cdot 2^{2s} \quad \Longleftrightarrow \quad \mathtt{Sup}(\beta)\cdot\varepsilon_0 > R\cdot 2^{2s}$,
2. $\mathtt{Inf}(\beta)\cdot(1-\varepsilon_0) < \mathtt{Inf}(\beta) - R\cdot 2^{2s} \quad \Longleftrightarrow \quad \mathtt{Inf}(\beta)\cdot\varepsilon_0 > R\cdot 2^{2s}$,

und wegen $\mathtt{Sup}(\beta) \geq \mathtt{Inf}(\beta)$ sind beide Bedingungen erfüllt, wenn gilt

$$\mathtt{Inf}(\beta)\cdot\varepsilon_0 > R\cdot 2^{2s}, \quad \beta := (\boldsymbol{y}\cdot 2^s)^2. \tag{11}$$

Unter der Voraussetzung (11) gilt damit die etwas grobe Einschließung (10), wobei der ganze Intervallausdruck rechts in (10) problemlos berechnet werden kann.

Die Bedingung (11) wird auf der Maschine mit Hilfe der Zweier-Exponenten beider Ausdrücke überprüft. Ist a eine Unterschranke von $\mathtt{Inf}(\beta)\cdot\varepsilon_0$, d.h. $\mathtt{Inf}(\beta)\cdot\varepsilon_0 \geq a$ und ist b eine Oberschranke von $R\cdot 2^{2s}$, d.h. $b \geq R\cdot 2^{2s}$, so ist die Bedingung (11) erfüllt, falls

$$a > b. \tag{12}$$

Berechnung von a: Mit $U := \mathtt{Inf}(\beta)$ und den Anweisungen

`exy = expo(U);` `exyl = expo_gr(lr_part(U));` erhält man

$$\texttt{Inf}(\beta) \cdot \varepsilon_0 > 2^{exy+exyl-2-2097} =: a.$$

Berechnung von b: Mit den Anweisungen

`exx = expo(R);` `exxl = expo_gr(lr_part(R));` erhält man

$$R \cdot 2^{2s} < 2^{exx+exxl+1+2s} =: b.$$

Mit diesen Abschätzungen ist dann (12) erfüllt, falls

$$\begin{array}{lll} exy + exyl - 2099 & > & exx + exxl + 1 + 2s \quad \Longleftrightarrow \\ exy + exyl - 2100 & > & (exx + s) + (exxl + s). \end{array} \tag{13}$$

Die Einschließung (10) ist damit garantiert, wenn (13) erfüllt ist, sonst werden Zähler und Nenner von $\alpha(\boldsymbol{x}, \boldsymbol{y})$ direkt nach (9) ohne wesentliche Überschätzungen eingeschlossen.

Wir können jetzt voraussetzen: $\boldsymbol{x}, \boldsymbol{y} \neq [0,0] \ \wedge \ |\boldsymbol{x}|, |\boldsymbol{y}| \not\to \infty$ und betrachten den Fall $|\boldsymbol{y}| \to 0$, d.h. $\boldsymbol{y}^2$ liefert einen integer-Überlauf, der zu vermeiden ist. Sei zunächst $|\boldsymbol{x}| = 1$. Dann gilt

$$\alpha(\boldsymbol{x}, \boldsymbol{y}) = \frac{-\boldsymbol{x}}{\boldsymbol{y}^2}.$$

Um einen integer-Überlauf bei $\boldsymbol{y}^2$ im Fall $exy < -c1 = -1073740000$ zu vermeiden, wird α mit 2^s erweitert, wobei s bestimmt wird durch

$$exy + exyl + s = -c1 \quad \Longleftrightarrow \quad s = -c1 - exy - exyl > 0, \quad \rightsquigarrow$$
$$\alpha(\boldsymbol{x}, \boldsymbol{y}) = \frac{-(\boldsymbol{x} \cdot 2^s) \cdot 2^s}{(\boldsymbol{y} \cdot 2^s) \cdot (\boldsymbol{y} \cdot 2^s)},$$

und jetzt können Zähler und Nenner von $\alpha(\boldsymbol{x}, \boldsymbol{y})$ für $-2147481824 \leq exy$ ohne integer-Überlauf eingeschlossen werden.

Sei $|\boldsymbol{x}| \neq 1$.

$$\alpha(\boldsymbol{x}, \boldsymbol{y}) = \frac{-\boldsymbol{x}}{(\boldsymbol{x} - 1) \cdot (\boldsymbol{x} + 1) + \boldsymbol{y}^2}.$$

Um jetzt einen integer-Überlauf bei $\boldsymbol{y}^2$ zu vermeiden, benutzen wir $|x - 1| \geq 2^{-2097}$, falls $x \neq \pm 1$ staggered Maschinenzahlen sind, vgl. dazu auch die Seite 19. Da auch $|x + 1| \geq 2^{-2097}$ erfüllt ist, folgt direkt

$$|(\boldsymbol{x} - 1) \cdot (\boldsymbol{x} + 1)| \geq 2^{-4194},$$

so dass wegen $exy < -c1 = -1073740000$ die folgende Beziehung

$$|(\boldsymbol{x} - 1) \cdot (\boldsymbol{x} + 1)| \gg |\boldsymbol{y}|^2 \tag{14}$$

so gut erfüllt ist, dass eine Einschließung von $Ne := (\boldsymbol{x}-1)\cdot(\boldsymbol{x}+1)+\boldsymbol{y}^2$ durch eine leichte Aufblähung von $\boldsymbol{p} := (\boldsymbol{x}\diamondsuit 1)\diamondsuit(\boldsymbol{x}\diamondsuit 1)$ nach rechts wie folgt realisiert werden kann:

$$Ne \subset \begin{cases} \texttt{[Inf(p),Sup\{Sup(p)*One\_p\_lx\_interval()\}]}, & \texttt{Sup(p)} \geq 0, \\ \texttt{[Inf(p),Sup\{Sup(p)*One\_m\_lx\_interval()\}]}, & \texttt{Sup(p)} < 0. \end{cases} \tag{15}$$

Hinweis:

Da in (15) bei der Aufblähung von $\boldsymbol{p}$ nach rechts der Wert $\texttt{Sup}(\boldsymbol{p})$ nicht vergrößert wird, könnte man argumentieren, dass eine Einschließung von Ne nach (15) nicht realisiert wird. Liegt jedoch ein Intervall $\boldsymbol{p}$ vor, mit $\boldsymbol{p} = [a, 0]$, so wurde bei der Berechnung von $\boldsymbol{p}$ auf der Maschine wegen $|(\boldsymbol{x}-1)\cdot(\boldsymbol{x}+1)| \geq 2^{-4194}$, d.h. also wegen der Ungleichung $\texttt{Sup}((\boldsymbol{x}-1)\cdot(\boldsymbol{x}+1)) \leq 2^{-4194}$, diese Oberschranke so stark nach oben auf Null aufgerundet, dass wegen (14) die Einschließung von Ne nach (15) dennoch garantiert ist. Für die Punktintervalle $\boldsymbol{x} = 1 + 2^{-2095}$ und $\boldsymbol{y} = 2^{2147483647}\cdot 10^{-323}$ erhält man für Zähler und Nenner von $\alpha(\boldsymbol{x},\boldsymbol{y})$ garantierte und enge Einschließungen. Die Aufblähung von $\boldsymbol{x}$ nach rechts erfolgt in `lx_cimath.cpp` mit Hilfe der C-XSC Funktion `lx_interval Blow_r( const lx_interval& p )`.

Wir kommen jetzt zum Fall $\boldsymbol{x} \to 0$, d.h. $exx < -c1 = -1073740000$, zusammen mit $\boldsymbol{x},\boldsymbol{y} \neq [0,0] \ \wedge\ |\boldsymbol{x}|, |\boldsymbol{y}| \not\to \infty$.

$$\alpha(\boldsymbol{x},\boldsymbol{y}) = \frac{-\boldsymbol{x}}{(\boldsymbol{y}-1)\cdot(\boldsymbol{y}+1)+\boldsymbol{x}^2}.$$

Sei zunächst $(\boldsymbol{y}-1)\cdot(\boldsymbol{y}+1) = [0,0]$, dann gilt[30]

$$\alpha(\boldsymbol{x},\boldsymbol{y}) = \frac{-\boldsymbol{x}}{\boldsymbol{x}^2} = \frac{-1}{\boldsymbol{x}},$$

womit Zähler und Nenner problemlos eingeschlossen werden können.

Sei jetzt $(\boldsymbol{y}-1)\cdot(\boldsymbol{y}+1) \neq [0,0]$. Wegen $exx < -c1 = -1073740000$ gilt nach (115) auf Seite 146 die Abschätzung:

$$|\boldsymbol{x}| < 2^{exx+exxl+1} < 2^{-c1+exxl+1}$$

und mit $s := -2\cdot c1 + 2\cdot exxl + 2$ folgt $|\boldsymbol{x}|^2 < 2^s$ und daher

$$(\boldsymbol{y}-1)\cdot(\boldsymbol{y}+1)+\boldsymbol{x}^2 \subset (\boldsymbol{y}-1)\cdot(\boldsymbol{y}+1)+[0,2^s],$$

womit dann Zähler und Nenner von $\alpha(\boldsymbol{x},\boldsymbol{y})$ problemlos eingeschlossen werden können.

Da jetzt $\boldsymbol{x}^2$ und $\boldsymbol{y}^2$ ohne integer-Überlauf berechnet werden können, lassen sich Zähler und Nenner von

$$\alpha(\boldsymbol{x},\boldsymbol{y}) = \frac{\boldsymbol{x}}{(1-\boldsymbol{x})\cdot(1+\boldsymbol{x})-\boldsymbol{y}^2}$$

problemlos einschließen.

[30] Beachten Sie, dass $\boldsymbol{x}$ stets ein Punktintervall ist!

6.2.14.2 Der Imaginärteil des inversen Tangens Nach W. Krämer ist zur Einschließung des Imaginärteils $\Im(f(\boldsymbol{z}))$ die folgende Funktion einzuschließen:

$$Q(\boldsymbol{x},\boldsymbol{y}) := \ln\left(1+\frac{4\cdot\boldsymbol{y}}{\boldsymbol{x}^2+(1-\boldsymbol{y})^2}\right). \tag{16}$$

Dabei ist $\boldsymbol{y}=[y,y]$ ein Punktintervall mit $y\geq 0$, und $\boldsymbol{x}=[x_1,x_2]$, mit $0\leq x_1\leq x_2$, ist ein echtes staggered Intervall. Wegen $Q(\boldsymbol{x},0)\equiv 0$ setzen wir daher voraus:

$$y>0\ \wedge\ 0\leq x_1\leq x_2. \tag{17}$$

Ersetzt man in (16) das Punktintervall $\boldsymbol{y}$ formal durch[31] $y=\sqrt{1+x^2}$, so erhält man nach einfachen Umrechnungen die Beziehung $Q(x,\sqrt{1+x^2})\equiv T(x)$, wobei $T(\boldsymbol{x})$ wie folgt definiert ist

$$T(\boldsymbol{x}) := \ln\left(1+\frac{2}{\sqrt{1+\boldsymbol{x}^2}-1}\right). \tag{18}$$

$T(\boldsymbol{x})$ ist dabei nach [27] einzuschließen für Punktintervalle $\boldsymbol{x}=[x,x]$, mit $x>0$. Zur Einschließung von $T(\boldsymbol{x})$ betrachten wir zunächst den Fall $x\to 0$, mit $x>0$. Im ersten Fall $\texttt{exx} := \texttt{expo}(\boldsymbol{x}) < -c1 := -1073740000$ vermeidet man einen integer-Überlauf bei der Auswertung von $\boldsymbol{x}^2$, wenn man den Bruch in (18) mit $\sqrt{1+\boldsymbol{x}^2}+1$ erweitert:

$$T(\boldsymbol{x}) = \ln\{2\cdot(1+\sqrt{1+\boldsymbol{x}^2})+\boldsymbol{x}^2\} - 2\cdot\ln(\boldsymbol{x}),\quad x>0. \tag{19}$$

Wegen $\{...\}\to 4$ für $x\to 0$ kann jetzt das Intervall $2\cdot(1+\sqrt{1+\boldsymbol{x}^2})+\boldsymbol{x}^2$ in (19) dadurch eingeschlossen werden, dass man das Intervall $[4,4]$ mit Hilfe der `Blow_r()`-Funktion leicht nach rechts aufbläht, womit der integer-Überlauf vermieden wird; vgl. dazu auch die Gleichung (15) auf Seite 205.

Im Fall $-c1\leq exx<1150$ wird der Nenner in (18) wie folgt eingeschlossen:

$$\sqrt{1+\boldsymbol{x}^2}-1 \subseteq \texttt{sqrtp1m1(sqr(x))}. \tag{20}$$

Für $exx\geq 1150$ wählt man zur Vermeidung eines integer-Überlaufs

$$\sqrt{1+\boldsymbol{x}^2}-1 \subseteq \texttt{sqrt1px2(x)} - 1. \tag{21}$$

Hinweise:

1. Laufzeitmäßig ist die Einschließung mit (21) optimal, sie liefert allerdings nur für $exx\geq 1150$ eine ausreichende Genauigkeit.

2. Die Einschließung von $T(\boldsymbol{x})$ erfolgt in `lx_cimath.cpp` mit Hilfe der Funktion

   ```
   lx_interval T_atan(const lx_real& x);
   ```

   vgl. auch die Quelltexte ab Seite 212.

[31] $y=\pm\sqrt{1+x^2}$ sind nach [15] die Extremalkurven des Imaginärteils $\Im(f(z))$.

Wir kommen jetzt zur Einschließung von

$$Q(\boldsymbol{x},\boldsymbol{y}) := \ln\left(1+\frac{4\cdot\boldsymbol{y}}{\boldsymbol{x}^2+(1-\boldsymbol{y})^2}\right), \quad \text{mit } y>0 \ \wedge\ 0\le x_1\le x_2, \tag{22}$$

und betrachten zunächst den Fall $\boldsymbol{y}=1$ und $0<x_1\le x_2$. Dabei ist zu beachten, dass $\boldsymbol{y}=1$ und $x_1=0$ nicht eintreten kann, da sonst das Eingangsintervall $\boldsymbol{z}$ eine Singularität enthalten würde. Es gilt

$$Q(\boldsymbol{x},1) = \ln\left(1+\frac{4}{\boldsymbol{x}^2}\right). \tag{23}$$

Mit der Anweisung $exx :=$ `expo(Sup(x))` betrachten wir jetzt den Fall $\boldsymbol{x}\to 0$, d.h. $exx < -c1 = 1073740000$. Dann kann $Q(\boldsymbol{x},1)$ wie folgt geschrieben werden

$$Q(\boldsymbol{x},1) = \ln(4+\boldsymbol{x}^2) - 2\cdot\ln(x),$$

und für $\boldsymbol{x}\to 0$ kann $\boldsymbol{x}^2$ folgendermaßen eingeschlossen werden

$$\boldsymbol{x}^2 \subseteq (\boldsymbol{x}-1)\cdot(\boldsymbol{x}+1)+1, \quad \text{d.h.}$$
$$Q(\boldsymbol{x},1) = \ln(5+(\boldsymbol{x}-1)\cdot(\boldsymbol{x}+1)) - 2\cdot\ln(x), \tag{24}$$

und die rechte Seite kann jetzt ohne vorzeitigen integer-Überlauf in hoher Genauigkeit eingeschlossen werden.

Wir betrachten jetzt den Fall $\boldsymbol{y}=1$ und $\boldsymbol{x}\to+\infty$, d.h. $exx > c1 = 1073740000$. Mit den Anweisungen

$$R = \texttt{Inf}(\boldsymbol{x});\ \ exx = \texttt{expo}(R);\ \ exxl = \texttt{expo\_gr(lr\_part}(R)\texttt{)}; \quad \text{gilt:}$$

$$\texttt{Inf}(\boldsymbol{x}) \ge 2^{exx+exxl-2} \quad \text{und daher} \quad \frac{4}{\texttt{Inf}(\boldsymbol{x})^2} \le 2^{-2\cdot(exx+exxl)+6} = 2^{dbl}. \tag{25}$$

Wir definieren noch

$$S = 2^{-2147483645}\cdot\texttt{MinReal} = 2^{-2147484667} = 2^{up}; \quad c2 = 2147483645;$$

Der ganzzahlige Wert dbl vom Typ *double* ist negativ und kann durchaus kleiner werden als $-c2$. Im Fall $dbl < up$ gilt wegen (23), (25) und wegen $0\le\ln(1+t)\le t$ die Einschließung

$$Q(\boldsymbol{x},1) \subset [0,2^{up}] = [0,S]. \tag{26}$$

Wir betrachten jetzt den Fall $dbl \geq up$.

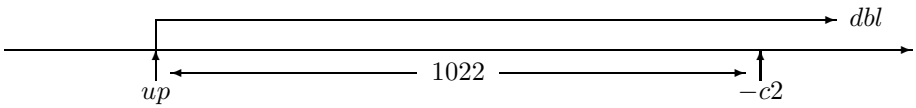


Abbildung 27: Der Fall $dbl \geq up$

Im Fall $dbl \geq -c2$ gilt dann

$$Q(\boldsymbol{x}, 1) \subset [0, 2^{dbl}], \tag{27}$$

wobei dbl jetzt direkt als integer-Zahl dargestellt werden kann.

Wir betrachten den verbleibenden Fall $up \leq dbl < -c2$, in dem dbl nicht als integer-Zahl darstellbar ist. Mit $dbl = -c2 + r$ bzw. $r = dbl + c2$ gilt $r < 0$ und $|r| \leq 1022$, woraus unmittelbar folgt

$$Q(\boldsymbol{x}, 1) \subset [0, 2^{-c2} \cdot 2^r], \tag{28}$$

wobei $2^{dbl} = 2^{-c2} \cdot 2^r$ realisiert wird durch $2^{-c2+r} = \texttt{lx\_real}(-c2, \texttt{comp}(0.5, r+1))$. Die Einschließungen (26), (27), (28) sind natürlich mit Überschätzungen verbunden, die in der Praxis jedoch keine große Rolle spielen sollten.

Wir kommen jetzt zur Einschließung von $Q(\boldsymbol{x}, \boldsymbol{y})$ unter den folgenden Voraussetzungen $y > 0,\ \boldsymbol{y} \neq [1,1],\ 0 \leq x_1 \leq x_2$ und zeigen dabei zunächst, dass bei der Berechnung des Quotienten

$$\frac{4 \cdot \boldsymbol{y}}{\boldsymbol{x}^2 + (1 - \boldsymbol{y})^2}$$

in (22) kein Overflow auftreten kann, d.h. also, dass der Quotient betragsmäßig hinreichend beschränkt ist. Für Maschinenzahlen y vom Typ `lx_real` gilt nach Seite 19

$$|y - 1| \geq 2^{-2097} \quad \Longrightarrow \quad |y - 1|^2 \geq 2^{-4194}. \tag{29}$$

$$\text{Mit} \quad f(y) := \frac{4y}{(1-y)^2} \quad \text{und} \quad f'(y) := \frac{4 \cdot (1 - y^2)}{(1-y)^4} \quad \text{folgt:}$$

$f(y)$ ist monoton wachsend in $0 \leq y < +1$ und monoton fallend in $1 < y < +\infty$. Im Bereich $0 \leq y < 1$ folgt daher

$$f(y) = \frac{4y}{(1-y)^2} \leq \frac{4 \cdot (1 - 2^{-2097})}{(1-y)^2} \leq 4 \cdot (1 - 2^{-2097}) \cdot 2^{+4194} < 2^{+4196}.$$

In $1 < y < +\infty$ erhält man analog

$$f(y) = \frac{4y}{(1-y)^2} \leq \frac{4 \cdot (1 + 2^{-2097})}{(1-y)^2} \leq 4 \cdot (1 + 2^{-2097}) \cdot 2^{+4194} < 2^{+4197},$$

so dass bei der Einschließung des Quotienten $4\boldsymbol{y}/(\boldsymbol{x}^2 + (1-\boldsymbol{y})^2)$ kein Overflow entstehen kann ∎

Wegen (29) ist weiter zu beachten, dass bei der Einschließung von $(1-\boldsymbol{y})^2$ für $y \to +1$ kein integer-Überlauf auftreten kann. Aber dies ist natürlich noch möglich für $y, x_2 \to \infty$ oder für $x_2 \to 0$.

Wir betrachten daher zunächst den Fall $y > 0,\ \boldsymbol{y} \neq [1,1]$ und $x_2 \to \infty\ \vee\ y \to \infty$, d.h. mit den Anweisungen

```
exx = expo(Sup(x));  exxl = expo_gr(lr_part(Sup(x)));
        exy = expo(y);   und   c1 = 1073740000
```

setzen wir voraus: $(exx > c1\ \ \&\&\ \ exxl > -100000)\ \big|\ \big|\ exy > c1$.

Wegen $\dfrac{4 \cdot \boldsymbol{y}}{\boldsymbol{x}^2 + (1-\boldsymbol{y})^2} \to [0,0]$ wird für $\mathtt{Sup}\left(\dfrac{4 \cdot \boldsymbol{y}}{\boldsymbol{x}^2 + (1-\boldsymbol{y})^2}\right)$ zunächst eine Ober-

Oberschranke berechnet, mit der dann eine Einschließung für $Q(\boldsymbol{x}, \boldsymbol{y})$ angegeben werden kann. Mit den Anweisungen

$$R = \mathtt{Inf}(\boldsymbol{x});\quad exx = \mathtt{expo}(R);\quad exxl = \mathtt{expo\_gr}(\mathtt{lr\_part}(R));$$
$$R = \mathtt{Inf}(1-\boldsymbol{y});\quad exy = \mathtt{expo}(R);\quad exyl = \mathtt{expo\_gr}(\mathtt{lr\_part}(R));\quad \text{gilt:}$$
$$|x_1| \geq 2^{exx+exxl-2};\quad |1-y| \geq 2^{exy+exyl-2};\quad \text{und damit}$$

(30) $$|x_1|^2 \geq 2^{2\cdot(exx+exxl)-4} = 2^{exx},\quad |1-y|^2 \geq 2^{2\cdot(exy+exyl)-4} = 2^{exxl}.$$

Im Fall $x_1 = 0$ folgt daraus: $|x_1|^2 + |1-y|^2 \geq 2^{exxl} = 2^{dbl}$, und im Fall $x_1 > 0$ erhält man $|x_1|^2 + |1-y|^2 \geq 2^{exx} + 2^{exxl}$ und mit $dbl := \mathtt{Max}(exx, exxl)$ folgt schließlich $|x_1|^2 + |1-y|^2 \geq 2^{dbl}$, und daraus ergibt sich die gesuchte Oberschranke

$$\mathtt{Sup}\left(\frac{4 \cdot \boldsymbol{y}}{\boldsymbol{x}^2 + (1-\boldsymbol{y})^2}\right) \leq 4 \cdot 2^{exy+exyl+1-dbl} = 2^{exy+exyl+3-dbl} = 2^{dbl}.$$

Der ganzzahlige Wert dbl in der Oberschranke 2^{dbl} ist negativ und kann durchaus kleiner werden als $-c2 = -2147483645$. Die Einschließung von $Q(\boldsymbol{x}, \boldsymbol{y})$ kann jetzt völlig analog zu (26), (27), (28) ab Seite 207 berechnet werden.

Wir kommen jetzt zum Fall $y > 0,\ \boldsymbol{y} \neq [1,1],\ 0 \leq x_1 \leq x_2$, wobei $\boldsymbol{x}^2$ und $(1-\boldsymbol{y})^2$ für $x_2, y \to \infty$ keinen integer-Überlauf erzeugen. Wegen $|1-y|^2 \geq 2^{-4194}$ liefert auch $(1-\boldsymbol{y})^2$ für $y \to 1$ keinen integer-Überlauf, so dass dieser jetzt nur noch durch $\boldsymbol{x}^2$ für $x_2 \to 0$ erzeugt werden kann. Wir betrachten daher noch den letzten Fall

$$x_2 > 0\ \wedge\ x_2 \to 0.$$

Wegen $exx < -c1$ und $|1-y|^2 \geq 2^{-4194}$ lässt sich der integer-Überlauf in $\boldsymbol{x}^2$ ganz einfach dadurch vermeiden, dass man das Intervall $(1-\boldsymbol{y})^2$ ganz analog zu (15) auf Seite 205 mit Hilfe der C-XSC Funktion `lx_interval Blow_r(const lx_interval& p)` leicht nach rechts aufbläht und dadurch den Nenner $\boldsymbol{x}^2 + (1-\boldsymbol{y})^2$ garantiert einschließt.

$Q(\boldsymbol{x}, \boldsymbol{y})$ wird in `lx_cimath.cpp` eingeschlossen mit Hilfe der C-XSC Funktion

```
lx_interval Q_atan(const lx_interval& x, const lx_interval& y),
```

vgl. die Quelltexte ab Seite 212.

Im **1. Beispiel** liefert das folgende Programm `lx_test77` für das komplexe Argument $z = 0+i\cdot(1-2^{-2095})$ eine garantierte und sehr enge Einschließung für den Funktionswert $f(z) = \arctan(z)$.

```
// Programm  lx_test77.cpp;
// Zum Test von atan(z);

#include <iostream>
#include "lx_cimath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 30;
    lx_cinterval Z,W;
    lx_interval X,Y;
    string s;

    X = lx_interval(0,l_interval(0));
    Y = 1 - lx_interval(-2095,l_interval(1));
    Z = lx_cinterval(X,Y);
    W = atan(Z);
    cout << SetDotPrecision(16*stagprec,16*stagprec+10)
         << Scientific;
    cout << "atan(Z) = " << W << endl;
    s << W;
    cout << "atan(Z) = " << s << endl;
}
```

Die folgende Ausgabe von `W` in lesbarer dezimaler Form liefert die Einschließung des Real- und Imaginärteils von $f(z) = \arctan(z)$ in hoher Genauigkeit:

$$\Re(f(z)) \in [0, 0].$$

$$\Im(f(z)) \in 10^{1} \cdot (\underbrace{7.26418245226822684269259\ldots373900697418399}_{301 \text{ korrekte Dez.-Ziffern}}{}^{751\ldots}_{677\ldots}) \cdot 10^{1}.$$

Eine mit *Mathematica* berechnete Näherung für $f(z)$ wird durch das obige Intervall W eingeschlossen. Die folgende Ausgabe über den String s liefert die interne Genauigkeit der Einschließung mit 475 korrekten Dezimalstellen:

$$\Re(f(z)) \in [0, 0].$$

$$\Im(f(z)) \in 2^{-1013} \cdot (\underbrace{6.37635299063959678061\ldots1638778050866239}_{475 \text{ korrekte Dez.-Ziffern}}{}^{940\ldots}_{411\ldots}) \cdot 10^{307}.$$

Auch dieser Imaginärteil schließt eine mit *Mathematica* berechnete Näherung ein.

Im **2. Beispiel** liefert das folgende Programm `lx_test78` für das komplexe Argument $z = 2^{-2147483647} + i$ eine garantierte und sehr enge Einschließung für den Funktionswert $f(z) = \arctan(z)$.

```
// Programm  lx_test78.cpp;
// Zum Test von atan(z);

#include <iostream>
#include "lx_cimath.hpp"

using namespace cxsc;
using namespace std;

int main()
{
    stagprec = 30;
    lx_cinterval Z,W;
    lx_interval X,Y;
    string s;

    X = lx_interval(-2147483647,l_interval(1));
    Y = lx_interval(0,l_interval(1));
    Z = lx_cinterval(X,Y);
    W = atan(Z);
    cout << SetDotPrecision(16*stagprec,16*stagprec+10)
         << Scientific;
    cout << "atan(Z) = " << W << endl;
    s << W;
    cout << "atan(Z) = " << s << endl;
}
```

Die folgende Ausgabe von `W` in lesbarer dezimaler Form liefert die Einschließung des Real- und Imaginärteils von $f(z) = \arctan(z)$ in hoher Genauigkeit:

$$\Re(f(z)) \in 10^{-1} \cdot (\underbrace{7.85398163397448309615660\ldots484018150622853184}_{\text{301 korrekte Dez.-Ziffern}}{}^{371\ldots}_{248\ldots}) \cdot 10^{0}.$$

$$\Im(f(z)) \in 10^{7} \cdot (\underbrace{7.44261117954893017873903\ldots68394444504619183}_{\text{301 korrekte Dez.-Ziffern}}{}^{904\ldots}_{714\ldots}) \cdot 10^{1}.$$

Die folgende Ausgabe von `W` über den String `s` liefert die interne Genauigkeit der beiden Einschließungen für Real- und Imaginärteil mit 481 bzw. 480 korrekten Dezimalstellen:

$$\Re(f(z)) \in 2^{-1023} \cdot (\underbrace{7.0595244323653213475 74\ldots1198952074002526474}_{\text{481 korrekte Dez.-Ziffern}}{}^{682\ldots}_{588\ldots}) \cdot 10^{307}.$$

$$\Im(f(z)) \in 2^{-993} \cdot (\underbrace{6.23032963970891899174 71\ldots995107260212732560}_{\text{480 korrekte Dez.-Ziffern}}{}^{895\ldots}_{733\ldots}) \cdot 10^{307}.$$

Die Berechnung einer Näherung für $f(z)$ liefert mit *Mathematica* für das gleiche Argument $z = 2^{-2147483647} + i$ eine Underflow-Fehlermeldung.

```
lx_interval Blow_r( const lx_interval& p )
// Blowing up p to the right; Only Sup(p) is changed:
// if (Sup(p)>=0) Sup(p) = Sup(p)*(1+2^(-2097)),
// if (Sup(p)< 0) Sup(p) = Sup(p)*(1-2^(-2097)).
{
    lx_real I,S;
    S = Sup(p);  I = Inf(p);
    S = ge_zero(S)? Sup(S*One_p_lx_interval()) :
                    Sup(S*One_m_lx_interval());
    return lx_interval(I,S);
}

bool sign_test(const lx_interval& x, int s_org)
// Only for internal use by function re_atan(...)
{
    bool bl1,bl2,alter;
    int signx;
    if (diam(li_part(x))>0)
    {
        bl1 = eq_zero(Sup(x)) && (s_org==1);
        bl2 = eq_zero(Inf(x)) && (s_org==-1);
        alter = bl1 || bl2;
    }
    else
        alter = sign(Sup(li_part(x))) != s_org;
    return alter;
} // sign_test(...)

void re_atan(const lx_interval&y,lx_interval&x,lx_interval&res)
// Calculating an inclusion res of 1 - y^2 - x^2;
// x is always a point interval and y =[y1,y2], with y1 <= y2;
// Blomquist, 14.11.2007;
{
    const int c1 = 1073740000;
    lx_interval ya(abs(y)),One(lx_interval(0,l_interval(1))),xa;
    lx_real R,U;
    int ex_x,ex_xl,ex_y,ex_yl,s,signx;
    double exx,exxl,exy,exyl;
    bool alter;

    ex_x = expo(x);
    ex_xl = expo_gr(Inf(li_part(x)));
    ex_y = expo(y);
    ex_yl = expo_gr(Sup(li_part(ya)));
    ya = y;
    signx = sign(Inf(li_part(x)));
```

```
if (ex_xl < -1000000)
    res = 1.0;
else // x <> 0:
    if (ex_yl < -1000000) // y == [0,0] and x <> [0,0];
        if(ex_x > c1)
        {
            res = 1/x - x;
            x = -1;
        }
        else
        {
            res = (1-x)*(1+x);
            if (Sup(res)>1) // to avoid an overestimation
                res = lx_interval(Inf(res),lx_real(1.0));
        }
    else // y <> [0,0] and x <> [0,0];
        if (ex_x>c1) // |x| --> infty;
            if (ex_y>c1) // |x| --> infty and |y| --> infty;
            {
                s = c1 - max(ex_x,ex_y);
                times2pown_neg(One,s); // 2^s
                times2pown_neg(x,s);   // x*2^s
                times2pown(ya,s);      // y*2^s
                res = (One-x)*(One+x) - sqr(ya);
                times2pown_neg(x,s);   // x*2^2s
            }
            else // |x| --> infty  and  |y| not--> infty;
                if (ex_y<-c1)
                {
                    res = (1-y)*(1+y);
                    res = res / x - x;
                    x = 1;
                }
                else // |x| --> infty and |y| normal;
                {
                    res = sqr(y);  xa = abs(x);
                    if (sqr(y) == 1.0)
                        res = xa;
                    else
                    {
                        res = res - 1.0; // res = y^2 - 1;
                        R = Sup(abs(res));
                        ex_y = expo(R);
                        ex_yl = expo_gr(lr_part(R));
                        exx = ex_x;   exy = ex_y;
                        exxl = ex_xl; exyl = ex_yl;
                        if (exy+exyl < 2*(exx+exxl)-2102)
```

```
                        res = xa *
                            lx_interval(
                              Inf(One_m_lx_interval()),
                              Sup(One_p_lx_interval()));
                    else
                        res = res/xa + xa;
                }
                x = (Sup(x)>0)? -1.0 : 1.0;
            }
        else // |x| not to infty
            if (ex_y>c1) // |y| --> infty;
            {
                s = c1 - ex_y - ex_yl;
                res = (x-1)*(x+1);
                R = Sup( abs(res) );
                times2pown_neg(x,s);   // x*2^s
                times2pown(ya,s);      // y*2^s
                if (eq_zero(R))
                    res = sqr(ya);
                else
                {   // R > 0;
                    ya = sqr(ya); // ya = (y*2^s)^2;
                    U = Inf(ya);
                    ex_y = expo(U);
                    ex_yl = expo_gr(lr_part(U));
                    exy = ex_y;  exyl = ex_yl;
                    ex_x = expo(R);
                    ex_xl = expo_gr(lr_part(R));
                    exx = ex_x;  exxl = ex_xl;
                    if (exy+exyl-2100 > (exx+s)+(exxl+s))
                        res = ya*lx_interval(
                                  Inf(One_m_lx_interval()),
                                  Sup(One_p_lx_interval()) );
                    else
                    {
                        times2pown_neg(res,s);
                        times2pown_neg(res,s);
                        // res = {(x-1)*(x+1)}*2^2s;
                        res = res + ya;
                    }
                }
                times2pown_neg(x,s); //(x*2^s)*2^s = x*2^(2s)
                x = -x;
            }
            else // |x|,|y| not to +infty
                if (ex_y<-c1) // |y| ---> 0, y<>[0,0];
                    if (abs(x)==1)
                    {
```

```
                            s = -c1 - ex_y - ex_yl;  // s > 0;
                            times2pown(x,s);
                            times2pown(x,s);
                            x = -x;
                            times2pown(ya,s);
                            res = sqr(ya);
                        }
                        else
                        {
                            res = (x-1)*(x+1);
                            res = Blow_r(res);
                            x = -x;
                        }
                    else // |x|,|y| not to infty and |y| not to 0
                        if (ex_x < -c1) // now: |x| --> 0
                        {
                            res = (y-1)*(y+1);
                            if (res == 0.0) // alpha = -1/x;
                            {
                                x = -1;
                                res = x;
                            }
                            else
                            {
                                s = 2*(ex_xl-c1+1);
                                res += lx_interval(
                                     lx_real(0),lx_real(s,1.0) );
                                x = -x;
                            }
                        }
                        else // x^2 and y2 can now be evaluated
                             // without any integer overflow!
                            res = (1-x)*(1+x) - sqr(y);
    alter = sign_test(x,signx);
    if (alter)
    {
        x = -x;
        res = -res;
    }
} // re_atan

lx_interval T_atan(const lx_real& x)
// Calculating an inclusion of
//              ln[ 1+2/(sqrt(1+x^2)-1) ].
// x will be handeld as a point interval [x]=[x,x], with x>0.
// Blomquist, 23.11.07;
{
    const int c1 = 1073740000;
```

```
    lx_interval res,
        ix(x);      // ix is point interval with x>0;
    int ex_x(expo(ix)), ex_xl(expo_gr(li_part(ix)));

    if (ex_x<-c1)
    { // preventing overflow
        res = 1 + sqrt1px2(ix);
        times2pown(res,1);      // res = 2*(1+sqrt(1+x^2));
        res += (ix-1)*(ix+1)+1; // res = 2*(1+sqrt(1+x^2)) + x^2
        ix = ln(ix);
        times2pown(ix,1);       // ix = 2*ln(x);
        res = ln(res) - ix;
    }
    else // -c1 <= ex_x
        if (ex_x<1150) // -c1 <= ex_x < 1150
            res = lnp1(2/sqrtp1m1(sqr(ix)));
        else res = lnp1(2/(sqrt1px2(ix)-1));

    return res;
} // T_atan

lx_interval Q_atan(const lx_interval& x, const lx_interval& y)
{
// x: abs(Re(z)); x is a real interval x=[x1,x2], with 0<=x1<x2.
// y: Inf(Im(z)); y is a point interval, with y >= 0.
// Q_atan returns an inclusion of ln[1 + 4y/(x^2+(1-y)^2)]
// Tested in detail; Blomquist, 23.11.2007;

    const int c1 = 1073740000;
    const int c2 = 2147483645;
    const lx_real S = lx_real(-c2,MinReal);
    const double up = -2147484667.0;

    int s,r;
    double exx,exxl,exy,exyl,dbl;
    lx_real R;
    lx_interval res(0.0),Hlp;
    int ex_y(expo(y)), ex_yl(expo_gr(li_part(y))),
        ex_x(expo(Sup(x))),ex_xl;

    ex_xl = expo_gr(lr_part(Sup(x)));
    if (ex_yl>-100000) // y = [y,y], y>0;
        if (y==1.0) // y = 1;
            if (ex_x < -c1)
            {
                res = ln(x);
                times2pown(res,1); // res = 2*ln(x)
                res = ln(5+(x-1)*(x+1)) - res;
```

```
        }
        else
            if (ex_x > c1)
            {
                R = Inf(x);
                exx = expo(R); exxl = expo_gr(lr_part(R));
                dbl = -2*(exx+exxl) + 6;
                if (dbl<up)
                    res = lx_interval(lx_real(0),S);
                else
                    if (dbl>=-c2)
                    {
                        s = (int) dbl;
                        R = lx_real(s,1.0);
                        res = lx_interval(lx_real(0),R);
                    }
                    else
                    {
                        r = (int) (dbl+c2);
                        R = lx_real(-c2,comp(0.5,r+1));
                        res = lx_interval(lx_real(0),R);
                    }
            }
            else // x^2 now without integer overflow
                res = lnp1(sqr(2/x));
    else // Now:  y>0  and  y!=1  and  0<=x1<=x2;
        if ( (ex_x>c1 && ex_xl>-100000) || ex_y>c1 )
        {
            R = Inf(x);
            exx = expo(R); exxl = expo_gr(lr_part(R));
            R = Inf(y-1.0); // R != 0:
            exy = expo(R); exyl = expo_gr(lr_part(R));

            if (exxl<-100000) // x1 = 0;
                dbl = 2*(exy+exyl) - 4;
            else
            {
                exx = 2*(exx+exxl) - 4;
                dbl = 2*(exy+exyl) - 4;
                if (exx > dbl) dbl = exx; // dbl = Max(...)
            }
            dbl = exy + exyl + 3 - dbl;
            // Now it holds:  Sup{4y/(x^2+(1-y)^2)} < 2^dbl;
            if (dbl<up)
                res = lx_interval(lx_real(0),S);
            else
                if (dbl>=-c2)
                {
```

```
                    s = (int) dbl;
                    R = lx_real(s,1.0);
                    res = lx_interval(lx_real(0),R);
                }
                else
                {
                    r = (int) (dbl+c2);
                    R = lx_real(-c2,comp(0.5,r+1));
                    res = lx_interval(lx_real(0),R);
                }
            }
            else // Now we have: y>0  and  y!=1  and  0<=x1<=x2;
                 // and:  x^2, (1-y)^2 produce no overflow for
                 //       x2, |1-y| --> +infty;
                 // furthermore |1-y| produces no integer
                 // overflow for y --> +1, and so only x^2 can
                 // produce an integer overflow for x2 --> 0.
                 // Thus we now consider the case: x2 --> 0:
            {
                R = Sup(x);
                exxl = expo_gr(lr_part(R));
                if (exxl>-100000 && ex_x < -c1)
                { // x2>0 and x2-->0;
                    res = y;
                    times2pown(res,2); // res = 4*y;
                    Hlp = sqr(1-y);
                    Hlp = Blow_r(Hlp);
                    res = lnp1(res/Hlp); // inclusion of Q(x,y);
                }
                else // x2=0  or  ex_x >= -c1,
                     // so x^2 and (1-y)^2 can now be calculated
                     // without integer overflow!
                {
                    res = y;
                    times2pown(res,2); // res = 4*y;
                    res = res/(sqr(x) + sqr(1-y));
                    res = lnp1(res); // res: inclusion of Q(x,y);
                }
            }

    return res;
} // Q_atan
```

Literatur

[1] Abramowitz M. and Stegun I. *Handbook of Mathematical Functions with Formulas, Graphs and Mathematical Tables.* National Bureau of Standards, Washington, 1964.

[2] Auzinger, W. and Stetter H.J.*Accurate Arithmetic Results for Decimal Data on Non-Decimal Computers.* Computing 35, 141-151, 1985.

[3] Alefeld G. and Herzberger J. *Introduction to Interval Computations.* Academic Press, New York, 1983.

[4] American National Standards Institute/Institute of Electrical and Electronics Engineers: “IEEE Standard for Binary Floating-Point Arithmetic”; ANSI/IEEE Std 754–1985, New York, 1985.

[5] Blomquist, F.; Hofschuster, W.; Krämer, W.: Realisierung der hyperbolischen Cotangens-Funktion in einer Staggered-Correction-Intervallarithmetik in C-XSC. Preprint 2004/3, Wissenschaftliches Rechnen / Softwaretechnologie, Universität Wuppertal, 2004.

[6] Blomquist F. Verbesserungen im Bereich komplexer Standardfunktionen, interne Mitteilung, Bergische Universität Wuppertal, 2005.

[7] Blomquist, F.; Hofschuster, W.; Krämer, W., Neher, M.: Complex Interval Functions in C-XSC, Preprint BUW-WRSWT 2005/2, Bergische Universität Wuppertal, pp. 1-48, 2005.

[8] Braune K., Krämer W. *High Accuracy Standard Functions for Real and Complex Intervals.* In Kaucher E., Kulisch U. and Ullrich Ch., editors, Computerarithmetic: Scientific Computation and Programming Languages, pages 81-114. Teubner, Stuttgart (1987)

[9] Braune K. Standard Functions for Real and Complex Point and Interval Arguments with Dynamic Accuracy. *Computing Supplementum*, 6:159-184 (1988).

[10] Hammer, R.; Hocks, M.; Kulisch, U.; Ratz, D., C++ Toolbox for Verified Computing: Basic Numerical Problems. Springer-Verlag, Berlin / Heidelberg / New York, 1995.

[11] Herzberger, J. (Ed), Topics in Validated Computations. Proceedings of IMACS-GAMM International Workshop on Validated Numerics, Oldenburg, 1993. North Holland, 1994.

[12] Hofschuster, W., Krämer, W.: C-XSC – A C++ Class Library for Extended Scientific Computing. *Numerical Software with Result Verification.* R. Alt, A. Frommer, B. Kearfott, W. Luther (eds), Springer Lecture Notes in Computer Science, 2004.

[13] Hofschuster, W.; Krämer, W.: FI_LIB, eine schnelle und portable Funktionsbibliothek für reelle Argumente und reelle Intervalle im IEEE-*double*-Format. Preprint 98/7 des IWRMM, Universität Karlsruhe, 227 Seiten, 1998.

[14] Klatte, R.; Kulisch, U.; Lawo, C.; Rauch, M.; Wiethoff, A.: C-XSC, A C++ Class Library for Extended Scientific Computing. Springer-Verlag, Berlin / Heidelberg / New York, 1993.

[15] Krämer, W.: Inverse Standardfunktionen für reelle und komplexe Intervallargumente mit a priori Fehlerabschätzungen für beliebige Datenformate, Dissertation, Universität Karlsruhe, 1987.

[16] Krämer W. Inverse Standard Functions for Real and Complex Point and Interval Arguments with Dynamic Accuracy. *Computing Supplementum*, 6:185-212, 1988.

[17] Krämer, W.: A priori Worst Case Error Bounds for Floating-Point Computations, IEEE Transactions on Computers, Vol. 47, No. 7, July 1998.

[18] Krämer, W., Wolff von Gudenberg, J. (eds): *Scientific Computing, Validated Numerics, Interval Methods*, Kluwer Academic Publishers Boston/Dordrecht/London, 398 pages, 2001.

[19] Krämer, W.: Mehrfachgenaue reelle und intervallmäßige Staggered-Correction Arithmetik mit zugehörigen Standardfunktionen, Bericht des Instituts für Angewandte Mathematik, Universität Karlsruhe, S. 1-80, 1988.

[20] Krämer, W.: Die Berechnung von Funktionen und Konstanten in Rechenanlagen. Habilitationsschrift, Universität Karlsruhe 1993.

[21] Krämer, W.: Multiple-Precision Computations with Result Verification, in: Scientific Computing with Automatic Result Verification, Adams, E., Kulisch, U.(editiors), Academic Press, pp. 325-356, 1993.

[22] Krämer, W.; Kulisch, U.; Lohner, R.: Numerical Toolbox for Verified Computing II. Springer Verlag, to appear.

[23] Kulisch, U.: Computer Arithmetic and Validity – Theory, Implementation and Application. To appear 2008.

[24] Lerch, M., Tischler, G., Wolff von Gudenberg, J., Hofschuster, W., and Krämer, W.: filib++, a Fast Interval Library Supporting Containment Computations. ACM Transactions on Mathematical Software Vol 32, Number 2, pp. 299-324, 2006.

[25] Lohner, R.: Interval arithmetic in staggered correction format. In: Adams, E., Kulisch, U.(Eds): Scientific Computing with Automatic Result Verification. Academic Press, San Diego, pp 301-321, 1993.

[26] Neher M.: The mean value form for complex analytic functions. *Computing*, 67:255-268, 2001.

[27] Neher M.: Complex Standard Functions and their Implementation in the CoStLy Library. Preprint Nr. 04/18, Universität Karlsruhe, 2004.

[28] Neher, M.: Complex Standard Functions and Their Implementation in the CoStLy Library, ACM Transactions on Mathematical Software, Vol. 33, Number 1, 27 pages, 2007.

[29] Rotmaier, B.: Die Berechnung der elementaren Funktionen mit beliebiger Genauigkeit. Dissertation, Universität Karlsruhe, 1971.

[30] Stetter, H.J. *Staggered Correction Representation, a Feasible Approach to Dynamic Precision*, Proceedings of the Symposium on Scientific Software, edited by Cai, Fosdick, Huang, China University of Science and Technology Press, Beijing, China, 1989.

[31] XSC website on programming languages for scientific computing with validation. `http://www.xsc.de`