Bergische Universität
Wuppertal

Realisierung der hyperbolischen Cotangens-Funktion in einer Staggered Correction Intervallarithmetik in C-XSC

Frithjof Blomquist, Werner Hofschuster, Walter Krämer

Preprint 2004/3

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`

Realisierung der hyperbolischen Cotangens-Funktion in einer Staggered Correction Intervallarithmetik in C-XSC

Frithjof Blomquist, Werner Hofschuster, Walter Krämer

Mai, 2004

Inhaltsverzeichnis

1 Zusammenfassung

Die vorliegende Arbeit liefert zunächst einen Überblick über die in C-XSC implementierte Staggered Correction Arithmetik, die auf den Klassen `l_real` und `l_interval` basiert. An Hand von vier Beispielprogrammen wird für die Datentypen `l_real` und `l_interval` der Unterschied zwischen der Präzision einer Variablen und der Präzision der aktuellen Staggered Correction Rechnung erklärt. Betragsmäßig zu kleine, nicht darstellbare Ergebnisse können nur mit einer stark begrenzten Genauigkeit berechnet werden. Für den Typ `l_interval` wird der Zusammenhang zwischen der vorgegebenen Präzision einer Staggered Correction Arithmetik, dem relativen Durchmesser eines staggered Intervalls x und der Genauigkeit des damit ausgewerteten arithmetischen Ausdrucks $T(x)$ geklärt. Danach darf der relative Durchmesser von x nicht zu groß werden, wenn eine vorgegebene Genauigkeit einer Staggered Correction Rechnung erreicht werden soll. Bei der Implementierung der Standardfunktionen ist man daher auf sehr schmale Argumentintervalle angewiesen, so dass eine Taylorapproximation $T_N(x) \approx T(x)$ intervallmäßig ohne nennenswerte Intervallüberschätzungen ausgewertet werden kann. Dadurch wird die Abschätzung eines Auswertefehlers überflüssig, so dass nur noch der Approximationsfehler zu berücksichtigen ist. Als Beispiel wird die hyperbolische Cotangens-Funktion betrachtet.

Die in dieser Arbeit verwendeten Hilfsprogramme werden auch in [3] ausführlich besprochen und stehen im Netz unter

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

zur Verfügung.

Keywords: C-XSC, a priori Fehlerschranken, Staggered Correction Arithmetik, genaues Skalarprodukt, genaue Funktionsapproximationen, Einschließungsmethoden.

MSC (2000): 65G20, 65Y15

2 Einleitung

Für Algorithmen aus dem Bereich der Numerik mit Ergebnisverifikation werden so genannte Intervallfunktionen benötigt, deren mit dem Computer berechnetes Ergebnisintervall den exakten Wertebereich der betrachteten Funktion über Intervallargumenten verlässlich einschließt. Die berechnete Einschließung soll dabei möglichst eng sein. Im *double*-Format wurden dazu für die Standardfunktionen spezielle Algorithmen mit Tabellenverfahren entwickelt, mit denen die unvermeidbaren Rundungsfehler auf ein Minimum reduziert werden konnten, [4], [5], [6], [16], [17], [18].

Bei vielen praktischen Anwendungen reicht im *double*-Format die Präzision von nur 53 Mantissen-Bits nicht aus, wenn die gewünschten Einschließungen mit einer Genauigkeit von z.B. 200 korrekten Mantissenstellen berechnet werden sollen. Diese höhere Genauigkeit kann mit der in C-XSC implementierten Staggered Correction Arithmetik erreicht werden, wobei ein staggered Intervall $\boldsymbol{x}$ mathematisch zu interpretieren ist als

eine Summe aus einer reellen Zahl und einem Intervall: $\boldsymbol{x} = \sum_{i=1}^{n-1} x_i + [x_n, x_{n+1}]$. Dieses Intervall $\boldsymbol{x}$ wird dabei intern gespeichert als Vektor mit $n+1$ *double*-Komponenten x_i; [20], [15], [11], [19]. Wegen $\texttt{Inf}(\boldsymbol{x}) = \sum_{i=1}^{n} x_i$, mit $x_i \in S(2,53)$ ist wie beim *double*-Format die Konstante $\texttt{minreal} = 2^{-1074}$ der kleinste positive Langzahlwert, so dass ein sehr kleines positives Zwischenergebnis, das im staggered Format nicht exakt darstellbar ist, nur durch ein staggered Intervall $\boldsymbol{y}$ mit einem vergleichsweise großen relativen Durchmesser, d.h. also mit nur geringer Genauigkeit eingeschlossen werden kann. Der Kehrwert $1/\boldsymbol{y}$ des Zwischenergebnisses ist dann zwar sehr groß, so dass für seine Darstellung etwa $2 \cdot 1022$ Binärziffern zur Verfügung stehen, aber die Genauigkeit dieses Kehrwerts kann natürlich nicht größer sein als die Genauigkeit des Ausgangsintervalls $\boldsymbol{y}$.

Bei der Implementierung der Standardfunktionen wird der beschriebene Effekt oft übersehen; so wird in [19] für die hyperbolische Cotangensfunktion die Auswertung von $\coth(\boldsymbol{x}) = 1/\tanh(\boldsymbol{x})$ vorgeschlagen. Für $\boldsymbol{x} \to 0$ können selbst bei großer Präzision die Einschließungen $\boldsymbol{y} \supset \tanh(\boldsymbol{x})$ auch für Punktintervalle $\boldsymbol{x}$ nur mit geringer Genauigkeit berechnet werden, so dass die Einschließungen $1/\boldsymbol{y}$ für $\coth(\boldsymbol{x})$ entsprechend grob ausfallen müssen. Den gleichen Effekt erhält man übrigens auch mit Hilfe der Darstellung $\coth(\boldsymbol{x}) = \coth(\boldsymbol{x})/\sinh(\boldsymbol{x})$. Eine hohe Genauigkeit erhält man nur, wenn man in der Nähe der Polstelle der hyperbolischen Cotangensfunktion ihre Laurententwicklung um 0 benutzt.

In mehreren Beispielprogrammen erhält der Leser eine grobe Übersicht über die Grundlagen der Staggered Correction Arithmetik in C-XSC. Danach wird in den folgenden Abschnitten die Implementierung der hyperbolischen Cotangensfunktion ausführlich beschrieben. In vier Teilbereichen werden unterschiedliche Algorithmen gewählt, deren Genauigkeit durch die Angabe entsprechender numerischer Ergebnisse belegt wird. Im letzten Abschnitt dieser Arbeit findet man das vollständige C-XSC Programmlisting der hyperbolischen Cotangensfunktion.

Zur Theorie und Implementierung von staggered Arithmetiken verweisen wir auf [11, 20]. Diese Langzahlarithmetiken basieren auf der Möglichkeit, Skalarprodukte von Gleitkommavektoren maximal genau auf einem Rechner zu berechnen. Diese Arithmetiken sind also dann besonders effizient, wenn genaue Skalarproduktberechnungen hardwaremäßig unterstützt werden. Alle sogenannten XSC-Sprachen, insbesodere auch C-XSC [9, 7, 8], bieten die genaue Berechnung von Skalarprodukten.

3 Zur Notation

$\mathbb{R}$	Menge der reellen Zahlen
$I\mathbb{R}$	Menge der reellen Intervalle
IR	Menge der Maschinenintervalle
$S(b,l,\underline{e},\overline{e}) = S(b,l)$	Gleitkommaraster mit Basis b, Mantissenlänge l und Exponent e, mit: $\underline{e} \le e \le \overline{e},\ \ e \in \mathbb{Z}$
$S(2,53,-1022,+1023) = S(2,53)$	IEEE-Datenformat *double*
`l_real, l_interval`	Basisklassen der Staggered Correction Arithmetik
$\boldsymbol{x}, \boldsymbol{y}$	Intervalle vom Typ *l_interval*
$\mathtt{Inf}(\boldsymbol{x}), \mathtt{Sup}(\boldsymbol{x})$	kleinster bzw. größter Intervallwert von $\boldsymbol{x}$
$\boldsymbol{x}[\,i\,]$	Komponenten der Variablen $\boldsymbol{x}$
$\mathtt{StagPrec}(\boldsymbol{x})$	liefert aktuelle Präzision des Intervalls $\boldsymbol{x}$
`stagprec`	liefert aktuelle Präzision der Staggered Correction Arithmetik
$\mathtt{dx} \in IR,\ \boldsymbol{x} \subseteq \mathtt{dx}$	`dx`: Maschinenintervall vom Typ *interval*
$\coth(x),\ x \neq 0$	reelle, hyperbolische Cotangens-Funktion
`Coth`	Funktionsname der $\coth$-Funktion im Quelltext
`einfachgenau`	Einschließung von $\coth(\mathtt{dx})$
d	relativer Durchmesser eines Intervalls
N	Polynomgrad der Taylor-Approximation
`x,y,s,t`	Quelltext-Intervalle vom Typ *l_interval*
$\circ \in \{+,-,\bullet,/\}$	exakte reelle Operatoren
$\mathtt{Minreal} = 2^{-1022} = 2.225.. \cdot 10^{-308}$	kleinste positive, normalisierte *double* Zahl
$\mathtt{minreal} = 2^{-1074} = 4.940.. \cdot 10^{-324}$	kleinste positive, denormalisierte *double* Zahl
$\mathtt{Maxreal} < 2^{+1024} = 1.797.. \cdot 10^{+308}$	größte, normalisierte *double* Zahl
$U := (-\mathtt{Minreal}, +\mathtt{Minreal})$	denormalisierter Zahlenbereich
$\mathtt{Minreal} \le \lvert x \rvert \le \mathtt{Maxreal}$	normalisierter Bereich im *double*-Format
$A_i,\ i = 1,2,3,4;$	Teilbereiche von `[MinReal,MaxReal]`
`expo(r)`	liefert Exponenten zur Basis 2 bez. $\mathtt{r} = m \cdot 2^{\mathtt{expo(r)}},\quad 0.5 \le m < 1$
B_n	Bernoulli-Zahlen
$W = \{f(x) \in I\mathbb{R} \,\vert\, x \in \boldsymbol{x}\}$	Wertebereich von $f(x)$ über dem Intervall $\boldsymbol{x}$

4 Staggered Arithmetik in C++

In **C++** basiert die Staggered Correction Arithmetik auf den beiden Datentypen *real* und *interval*. In den Klassen `l_real` und `l_interval` sind alle dazu notwendigen Operatoren und Methoden definiert und können mit Hilfe der Dateien `l_real.hpp` und `l_interval.hpp` in entsprechende **C++** Programme eingebunden werden. Vergleichen Sie dazu bitte die Programme ab Seite 7 und auf Seite 22.

4.1 Variablen vom Typ *l_real*

In **C++** wird eine Variable x vom Typ *l_real* gespeichert als ein Vektor mit n Komponenten x_i vom Typ *real*:

$$x = \sum_{i=1}^{n} x_i, \quad x \in \mathbb{R}, \quad x_i \in S(2,53), \quad n = \texttt{StagPrec(x)} \geq 1, \tag{1}$$

wobei die jeweilige Präzision der Variablen x mit Hilfe der Funktion `StagPrec(x)` bestimmt werden kann und nicht mit der durch `stagprec` ≥ 1 vorgegebenen Präzision der eigentlichen staggered Rechnungen zu verwechseln ist. Die in `l_real` deklarierte Variable `stagprec` wird mit `stagprec` $= 2$ initialisiert, kann vom Anwender in einem Programm dynamisch verändert werden und bestimmt die Präzision der staggered Arithmetik.

In einem ersten Programmbeispiel wird gezeigt, dass bei der Auswertung des Terms $(2^{511} + 2^{-537}) \cdot (2^{511} - 2^{-537}) = 2^{1022} + 2^{-1074}$ das vollständige Ergebnis $2^{1022} + 2^{-1074}$ schon mit $n = 2$ exakt dargestellt werden kann. Die C-XSC Funktion `int StagPrec(l_real& x)` liefert die momentane Präzision der staggered Variablen `x`, und der Operator `[ ]` ermöglicht den Zugriff auf die entsprechenden Komponenten von `x`. Dieser Operator `[ ]` sollte jedoch aus Sicherheitsgründen nur zur Anwendung kommen, wenn die Präzision von `x` genau bekannt ist!

```
// Programm staggered1.cpp
#include <l_real.hpp>
#include <iostream>

using namespace std;
using namespace cxsc;

int main()
{
    l_real x,y,z;
    real r1,r2;
    int n;
    r1 = comp(0.5,+512);  // r1 = 2^(+511)
    r2 = comp(0.5,-536);  // r2 = 2^(-537)
    x = r1;
    z = (x+r2)*(x-r2);    // z = 2^(+1022) - 2^(-1074)
    n = StagPrec(z);
```

```
      cout << "Praezision n von z:   n = " << n << endl;
      if (z[1]==comp(0.5,1023))
         cout << "z[1] = +2^(+1022)" << endl;
      if (z[2]==-minreal)
         cout << "z[2] = -2^(-1074)" << endl;
      stagprec = 20;
      x = 1;
      z = x / 3;
      cout << Scientific << SetDotPrecision(16*21,16*21)
           << "1/3 = " << z << endl;
}
```

Das Programm `staggered1.cpp` liefert die folgende Ausgabe:

```
Praezision n von z:   n = 2
z[1] = +2^(+1022)
z[2] = -2^(-1074)
1/3 = 3.333333333333333333333333333333333333333333333333333333333
        333333333333333333333333333333333333333333333333333333333
        333333333333333333333333333333333333333333333333333333333
        333333333333333333333333333333333333333333333333333333333
        333333333333333333333333333333333333333333333333333333333
        333333333333333333333333333333333333333333316864478
        471958E-0001
```

Das erste Beispiel zeigt, dass schon mit den beiden *real*-Zahlen `z[1]` und `z[2]` das Ergebnis $2^{+1022} - 2^{-1074}$ exakt gespeichert werden kann. Im zweiten Beispiel kann der Wert $1/3$ weder im binären noch im dezimalen Zahlensystem exakt dargestellt werden. Wegen `minreal` $= 2^{-1074} = 4.9\ldots \cdot 10^{-324}$ lässt sich daher $1/3$ mit höchstens 324 korrekten dezimalen Stellen angegeben, selbst wenn man `stagprec` $= 20$ noch weiter vergrößert. Daraus ergibt sich die folgende Grundregel:

> Wird ein nicht darstellbares Ergebnis einer staggered-correction Rechnung betragsmäßig zu klein, so kann unabhängig vom gewählten `stagprec`-Wert das Ergebnis nur noch mit einer stark begrenzten Stellenzahl berechnet werden.

Das folgende Programm `staggered2.cpp` demonstriert, dass die durch die Variable `stagprec` vorgegebene Präzision einer staggered Rechnung und die mit der Funktion `StagPrec(x)` bestimmte Präzision der Variablen `x` durchaus verschieden sein können. Das Programm zeigt außerdem, dass die Darstellung von `x` durch die Summe in (1) nicht eindeutig ist.

```
// Programm staggered2.cpp
#include <l_real.hpp>
#include <iostream>
```

```
using namespace std;
using namespace cxsc;

int main()
{
    l_real x;
    real r = 6;
    cout << "Vordefinierte Praezision stagprec = "
         << stagprec <<endl;
    stagprec = 3;
    cout << "Momentane Praezision     stagprec = "
         << stagprec <<endl;
    cout << "Praezision von x = " << StagPrec(x) << endl;
    x = 6;
    cout << "Praezision von x = " << StagPrec(x) << endl;
    x = x + 0;
    cout << "Praezision von x = " << StagPrec(x) << endl;
    x[1] = 1;  x[2] = -1;  x[3] = 0;
    cout << "x[1] = " << x[1] << endl;
    cout << "x[2] = " << x[2] << endl;
    cout << "x[3] = " << x[3] << endl << endl;
    x = x + 0;
    cout << "x[1] = " << x[1] << endl;
    cout << "x[2] = " << x[2] << endl;
    cout << "x[3] = " << x[3] << endl;
    stagprec = 2;
    cout << "Neue Praezision:  stagprec = " << stagprec << endl;
    cout << "Alte Praezision von x = " << StagPrec(x) << endl;
    x = x + 1;
    cout << "Neue Praezision von x = " << StagPrec(x) << endl;
}
```

Das obige Programm `staggered2.cpp` liefert die folgende Ausgabe:

```
Vordefinierte Praezision stagprec = 2
Momentane Praezision     stagprec = 3
Praezision von x = 2
Praezision von x = 1
Praezision von x = 3
x[1] =   1.000000
x[2] =  -1.000000
x[3] =   0.000000

x[1] =   0.000000
x[2] =   0.000000
x[3] =   0.000000
Neue Praezision:  stagprec = 2
Alte Praezision von x = 3
```

```
Neue Praezision von x = 2
```

Beachten Sie bitte, dass gegen Ende des obigen Programms `staggered2.cpp` durch `stagprec` $= 2$ die Präzision der nachfolgenden staggered Arithmetik auf 2 gesetzt wird. Unmittelbar danach behält jedoch die Variable `x` noch ihre alte Präzision 3. Erst mit der Anweisung `x = x + 1;` erhält dann `x` die Präzision 2, da diese Anweisung eine staggered Addition mit der neuen Präzision 2 realisiert.

4.2 Variablen vom Typ *l_interval*

In **C++** wird eine Variable $\boldsymbol{x}$ vom Typ *l_interval* intern gespeichert als ein Vektor mit $n+1$ Komponenten x_i vom Typ *real*:

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

Mathematisch ist obige Darstellung zu interpretieren als:

$$\boldsymbol{x} = \sum_{i=1}^{n-1} x_i + \boldsymbol{z}, \quad \boldsymbol{z} = [x_n, x_{n+1}], \quad x_j \in S(2,53), \quad j = 1, 2, \ldots, n+1; \tag{3}$$

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

$$\texttt{Inf}(\boldsymbol{x}) = \sum_{i=1}^{n} x_i, \qquad \texttt{Sup}(\boldsymbol{x}) = \sum_{i=1}^{n-1} x_i + x_{n+1};$$

Beachten Sie bitte: Eine Variable `x` vom Typ *l_real* mit der Präzision n besitzt genau n *real*-Komponenten, und eine Variable `x` vom Typ *l_interval* mit der gleichen Präzision n besitzt $n+1$ *real*-Komponenten.

Das nachfolgende Programm `staggered3.cpp` liefert einfache Anwendungen zu den Datentypen *l_real* und *l_interval*. Die beiden Funktionen

`l_real Inf(l_interval& x)` und `l_real Sup(l_interval& x)`

liefern das Infimum bzw. Supremum von `x`, wobei der Rückgabewert vom Typ *l_real* unabhängig vom aktuellen `stagprec`-Wert die gleiche Präzision erhält wie die Variable `x` selbst. Damit wird erreicht, dass Infimum bzw. Supremum ohne Genauigkeitsverluste berechnet werden.

Die Funktion

`l_interval adjust(l_interval& x)`

liefert ein Rückgabeintervall vom Typ *l_interval*, das `x` bezüglich der durch `stagprec` vorgegebenen Präzision optimal einschließt; vergleichen Sie dazu die Ausgaben der Variablen `t` und `x` auf Seite 12.

Beachten Sie bitte, dass auch jetzt die Darstellung (2) nicht eindeutig ist, da z.B. für $n = 3$ die folgende Identität gilt:

$$\boldsymbol{x} = (5, 0, 1, 4) \equiv (6, 0, 0, 3) \equiv (0, 0, 6, 9)$$

Das obige Beispiel zeigt außerdem, dass $\boldsymbol{x}$ nicht das Null-Intervall sein muss, nur weil die ersten Komponenten verschwinden. Dagegen ist $\boldsymbol{x}$ stets ein Punktintervall, wenn die beiden letzten Komponenten übereinstimmen, d.h. wenn gilt: $x_n = x_{n+1}$.

Das folgende Programm `staggered3.cpp`

```
// Programm staggered3.cpp
#include <l_interval.hpp>
#include <iostream>

using namespace std;
using namespace cxsc;

int main()
{
    l_real r1,r2;
    int p;
    l_interval x,y,t;
    cout << "Vorgegebene Praezision stagprec = "
         << stagprec << endl;
    stagprec = 3;
    cout << "Momentane Praezision     stagprec = "
         << stagprec <<endl;
    cout << "Praezision von x  : " << StagPrec(x) << endl;
    x = l_interval(1)/3;  p = StagPrec(x);
    cout << "Praezision von x  : " << p << endl;
    cout << Scientific << SetDotPrecision(16*p,16*p)
         << "x = " << x << endl;
    y = x;
    cout << "Praezision von y  : " << StagPrec(y) << endl;
    cout << "Praezision von r1 : " << StagPrec(r1) << endl;
    stagprec = 2;
    cout << "Jetzige Praezision stagprec = " << stagprec <<endl;
    r1 = Inf(x);  r2 = Sup(x);
    cout << "Praezision von r1 ist dennoch "
         << StagPrec(r1) << endl;
    if ((r1[3]!=r2[3]) && (x[3]!=x[4]))
        cout << "x ist kein Punktintervall" << endl;
    if ((r1[3]==x[3]) && (r2[3]==x[4]))
        cout << "Es gilt:  r1[3]=x[3]  und  r2[3]=x[4]." << endl;
    t = adjust(x); // t ist Einschliessung von x.
    cout << "Praezision von t  : " << StagPrec(t) << endl;
    cout << SetDotPrecision(16*stagprec,16*stagprec)
         << Scientific << "t = " << t << endl;
    if (x<t && x==y)
      cout << "t ist Einschliessung von x = y."<< endl;
}
```

liefert die folgende Ausgabe:

```
Vorgegebene Praezision stagprec = 2
Momentane Praezision     stagprec = 3
Praezision von x  : 2
```

```
Praezision von x  : 3
x = [3.33333333333333333333333333333333333333333333331E-0001,
     3.33333333333333333333333333333333333333333333338E-0001]
Praezision von y  : 3
Praezision von r1 : 2
Jetzige Praezision stagprec = 2
Praezision von r1 ist dennoch 3
x ist kein Punktintervall
Es gilt:   r1[3]=x[3]  und  r2[3]=x[4].
Praezision von t  : 2
t = [3.33333333333333333333333333333329E-0001,
     3.33333333333333333333333333333336E-0001]
t ist Einschliessung von x = y.
```

Beachten Sie auch jetzt, dass der Operator `[ ]` nur dann zur Anwendung kommt, wenn die Präzision der benutzten Variablen vorher genau bekannt ist. Andernfalls greift man auf nicht definierte Speicherbereiche zu und riskiert dabei völlig falsche Ergebnisse oder einen Programmabbruch!

5 Anmerkungen zur Fehlerabschätzung

Ziel dieses Abschnittes ist es, diejenigen Grundideen aufzuzeigen, mit denen die Fehlerabschätzungen für Standardfunktionen in eine Staggered-Correction Arithmetik durchzuführen sind. Mit den so berechneten Fehlerschranken sollen wir dann in der Lage sein, zu einem vorgegebenen Punkt- oder Intervallargument $\boldsymbol{x}$ eine garantierte und möglichst enge Einschließung aller Funktionswerte $f(x)$, mit $x \in \boldsymbol{x}$ zu berechnen. Um dabei eine Genauigkeit zu erhalten, die mit der durch `stagprec` vorgegebenen Präzision vergleichbar ist, müssen wir die folgende Grundforderung stellen:

> Damit die Genauigkeit einer Staggered-Correction Rechnung mit der durch `stagprec` vorgegebenen Präzision vergleichbar wird, darf der relative Durchmesser des Argumentintervalls $\boldsymbol{x}$ vom Typ *l_interval* den Wert $10^{-16 \cdot \texttt{stagprec}}$ nicht wesentlich übersteigen.

Wir verdeutlichen den Inhalt dieser Grundforderung am einfachsten mit Hilfe des folgenden Programms `staggered4.cpp`:

```
// Programm staggered4.cpp
// Zur Demonstration des Einflusses des relativen Durchmessers
// eines Argumentintervalls x auf die Genauigkeit einer
// staggered Rechnung.

#include <l_interval.hpp>
#include <iostream>

using namespace std;
using namespace cxsc;

int main()
{
    l_interval x,y;
    stagprec = 3;
    cout << "Gewaehlte Praezision:  stagprec = "
         << stagprec << endl;
    x = 3;  // x ist Punktintervall
    cout << "Praezision von x:  " << StagPrec(x) << endl;
    cout << SetDotPrecision(16,16)
         << "Rel. Durchmesser von x: " << diam(x)/Inf(x) << endl;
    y = 1/x;
    cout << SetDotPrecision(16*stagprec,16*stagprec+1)
         << Scientific << "1/x = " << y << endl;

    x = x + interval(0,1e-48);
    cout << "Praezision von x:  " << StagPrec(x) << endl;
    cout << SetDotPrecision(16,16)
         << "Rel. Durchmesser von x: " << diam(x)/Inf(x) << endl;
    y = 1/x;
    cout << SetDotPrecision(16*stagprec,16*stagprec+1)
         << Scientific << "1/x = " << y << endl;

    x = 3;
    x = x + interval(0,1e-32);
    cout << "Praezision von x:  " << StagPrec(x) << endl;
    cout << SetDotPrecision(16,16)
         << "Rel. Durchmesser von x: " << diam(x)/Inf(x) << endl;
    y = 1/x;
    cout << SetDotPrecision(16*stagprec,16*stagprec+1)
         << Scientific << "1/x = " << y << endl;
}
```

Das umseitige Programm `staggered4.cpp` liefert die folgende Ausgabe:

```
Gewaehlte Praezision:  stagprec = 3
Praezision von x:  1
Rel. Durchmesser von x: 0.0000000000000000
1/x = [3.3333333333333333333333333333333333333333333310E-0001,
       3.3333333333333333333333333333333333333333333379E-0001]
Praezision von x:  3
Rel. Durchmesser von x: 3.3333333333333332E-0049
1/x = [3.3333333333333333333333333333333333333333333173E-0001,
       3.3333333333333333333333333333333333333333333379E-0001]
Praezision von x:  3
Rel. Durchmesser von x: 3.3333333333333335E-0033
1/x = [3.3333333333333333333333333333332222222222222143E-0001,
       3.3333333333333333333333333333333333333333333379E-0001]
```

Man erkennt deutlich, dass in den ersten beiden Beispielen mit den relativen Durchmessern 0.0 und $3.33\ldots\cdot 10^{-49}$ die Genauigkeit mit der durch `stagprec` $= 3$ vorgegebenen Präzision übereinstimmt, da jeweils etwa $16 \cdot$ `stagprec` $= 48$ korrekte dezimale Ziffern berechnet werden. Im letzten Beispiel erhält man jedoch nur noch 32 korrekte dezimale Ziffern, da jetzt der relative Durchmesser $3.33\ldots\cdot 10^{-33}$ deutlich größer ist als $10^{-16\cdot\texttt{stagprec}} = 10^{-48}$. Beachten Sie bitte, dass im letzten Beispiel die Genauigkeit auch nicht dadurch vergrößert werden kann, dass der Wert von `stagprec` weiter erhöht wird!

Die auf Seite 8 für den Datentyp *l_real* formulierte Grundregel lässt sich wörtlich auch auf den Datentyp *l_interval* übertragen:

> Wird ein nicht darstellbares Ergebnis einer staggered-correction Rechnung betragsmäßig zu klein, so kann unabhängig vom gewählten `stagprec`-Wert das Ergebnis nur noch mit einer stark begrenzten Stellenzahl berechnet werden.

Diese zu starken Ungenauigkeiten können jedoch oft durch geeignete Skalierung von zu kleinen Zwischenergebnissen mit entsprechenden Zweierpotenzen 2^n abgemindert bzw. vermieden werden.

Für Standardfunktionen vom Typ *real* werden absolute oder relative Fehlerschranken für alle Maschinenargumente des jeweiligen Definitionsbereichs angegeben, um damit zu einem vorgegeben Argumentintervall $\boldsymbol{x}$ eine garantierte Einschließung aller Funktionswerte $f(x)$ für $x \in \boldsymbol{x}$ berechnen zu können. Bei monotonen Funktionen werden Infimum und Supremum dieser Einschließungen berechnet, indem an den Randpunkten von $\boldsymbol{x}$ die Standardfunktionen einzeln ausgewertet werden und mit Hilfe der bekannten Fehlerschranken geeignet auf- bzw. abgerundet wird. Nach diesem Verfahren können die Einschließungen auch für breite Argumentintervalle $\boldsymbol{x}$ nahezu optimal berechnet werden. Da wir jedoch bei einer Staggered-Correction Arithmetik für eine hohe Genauigkeit auf nahezu punktförmige Argumentintervalle $\boldsymbol{x}$ angewiesen sind, gehen wir bei der Realisierung der staggered Standardfunktionen einen anderen Weg. Wird $f(x)$ z.B. durch eine

Taylorreihe $T_N(x)$ approximiert

$$f(x) \approx T_N(x),$$

so wird $T_N(x)$ intervallmäßig ausgewertet und damit eine Abschätzung des Auswertefehlers überflüssig, da diese intervallmäßige Auswertung automatisch eine garantierte Einschließung von $T_N(x)$ für alle $x \in \boldsymbol{x}$ liefert. Es muss daher nur noch eine Schranke des absoluten Approximationsfehlers angegeben werden, wobei der Polynomgrad N so zu wählen ist, dass die durch `stagprec` vorgegebene Genauigkeit erreicht wird. Im folgenden Abschnitt werden diese Schritte am Beispiel der hyperbolischen Cotangens-Funktion ausführlich beschrieben.

6 $\coth(x)$

Zu einem vorgegebenen und nicht zu breiten Argumentintervall $\boldsymbol{x}$ vom Typ *l_interval* ist für alle reellen $x \in \boldsymbol{x}$, mit $0 \notin \boldsymbol{x}$, eine garantierte und möglichst enge Einschließung aller Funktionswerte $f(x) = \coth(x)$ zu berechnen.

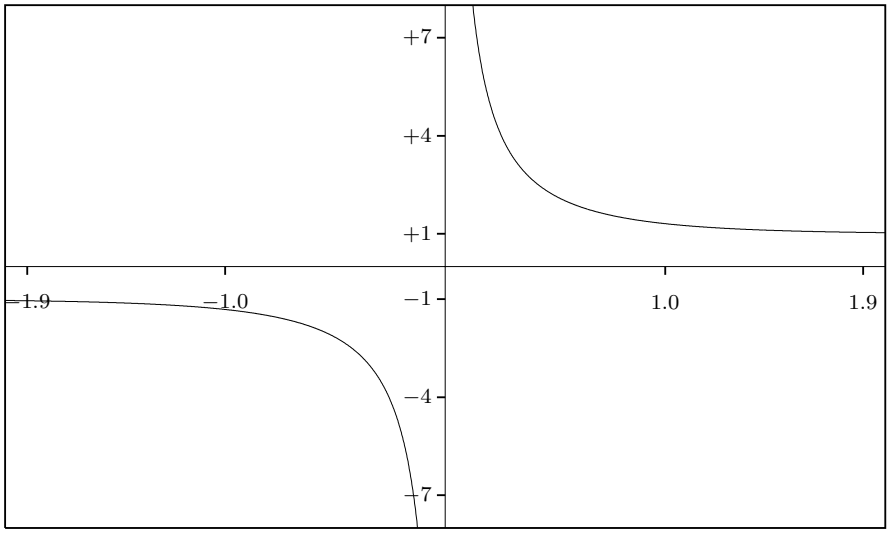

Abbildung 1: $\coth$-Funktion

Die $\coth$-Funktion ist definiert durch:

$$\coth(x) = \frac{\cosh(x)}{\sinh(x)} = \frac{e^x + e^{-x}}{e^x - e^{-x}} = 1 + \frac{2}{e^{2x} - 1} \tag{4}$$

Im Ursprung besitzt $\coth(x)$ eine Polstelle mit Vorzeichenwechsel, und für $0 < |x| < \pi$ gilt nach [1] die folgende Laurent-Entwicklung:

$$\coth(x) = \frac{1}{x} + \frac{x}{3} - \frac{x^3}{45} + \frac{2}{945}x^5 - + \ldots + \frac{2^{2n}B_{2n}}{(2n)!}x^{2n-1}, \tag{5}$$

wobei B_n die Bernoullischen Zahlen sind: $B_0 = 1, B_1 = -\frac{1}{2}, B_2 = \frac{1}{6}, B_4 = -\frac{1}{30}, \ldots$ Weitere Informationen findet man in [1, Seite 804].

Für die spätere Fehlerabschätzung ist es wichtig zu wissen, dass für $x > 0$ die Reihe in (5) eine alternierende Leibniz-Reihe ist. Bricht man diese Reihe ab, so ist der Fehler höchstens so groß wie der Betrag des ersten weggelassenen Summanden, [10, Seite 63].

6.1 Der Algorithmus

Das vorgegebene Argumentintervall $\boldsymbol{x}$ vom Typ *l_interval* wird zunächst optimal in das Intervall `dx` vom Typ *interval* gerundet, so dass gilt $\boldsymbol{x} \subseteq$ `dx`. Die Programmanweisung `einfachgenau = coth(dx);` liefert dann mit `einfachgenau` vom Typ *interval* eine garantierte Einschließung aller Funktionswerte $f(x) = \coth(x)$, mit $x \in \boldsymbol{x}$. Bezeichnen wir mit `y` vom Typ *l_interval* eine Einschließung der Funktionswerte $f(x) = \coth(x)$, wobei `y` in einer Staggered-Correction Arithmetik berechnet sein soll, so kann bei einem zu breiten Intervallargument $\boldsymbol{x}$ wegen der unvermeidbaren Intervallüberschätzungen dieses `y` mit einem größeren Intervalldurchmesser berechnet worden sein als die Einschließung `einfachgenau`. In einem solchen Fall liefert dann der Durchschnitt `y` $\cap$ `einfachgenau` eine verbesserte Einschließung der Funktionswerte $f(x) = \coth(x)$, mit $x \in \boldsymbol{x}$. In der Sprachumgebung C-XSC wird der Durchschnitt mit Hilfe des Operators & realisiert.

Im Falle $0 \in \boldsymbol{x}$ gilt auch $0 \in$ `dx`, und bei der Berechnung von $\cot($`dz`$)$ erfolgt eine entsprechende Fehlermeldung mit Programmabbruch.

Wegen der Punktsymmetrie $f(-x) \equiv -f(x)$ kann man sich auf Intervallargumente $\boldsymbol{x}$ mit `Inf`$(\boldsymbol{x}) > 0$ beschränken. Der Bereich [`MinReal`, `MaxReal`] wird dabei in vier Teilbereiche A_i unterteilt[1]:

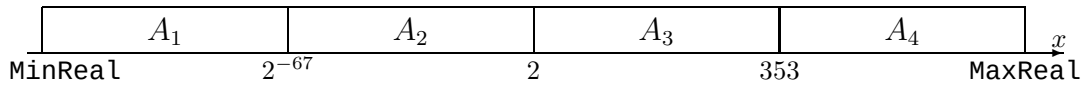

Abbildung 2: Teilintervalle A_i des Bereichs [`MinReal`, `MaxReal`]

Zu vorgegebenem Argumentintervall $\boldsymbol{x}$ wird eine Einschließung von $\coth(\boldsymbol{x})$ berechnet durch:

$$\coth(\boldsymbol{x}) \subseteq \begin{cases} \dfrac{1}{\boldsymbol{x}} + \dfrac{\boldsymbol{x}}{3} - \dfrac{\boldsymbol{x}^3}{45} + - \ldots & \text{falls } \texttt{Inf(dx)} < 2^{-67} \\ \dfrac{\cosh(\boldsymbol{x})}{\sinh(\boldsymbol{x})} & \text{falls } \texttt{Inf(dx)} < 2 \\ 1 + \dfrac{2}{e^{2\boldsymbol{x}} - 1} & \text{falls } \texttt{Inf(dx)} < 353 \\ [1, 1 + 2^{-1017}] & \text{falls } \texttt{Inf(dx)} \geq 353 \end{cases} \tag{6}$$

[1] Wählt man kleinere Argumente als `MinReal`, so liefert die staggered Auswertung von 1/x in der Laurent-Reihe eine Fehlermeldung mit Programmabbruch.

In A_2, A_3 erhält man damit automatisch garantierte Einschließungen, da identische Funktionsterme intervallmäßig ausgewertet werden. In den Bereichen A_1, A_4 liefern die entsprechenden Fehlerabschätzungen ebenfalls Einschließungen für $\coth(\boldsymbol{x})$, und zwar auch dann, wenn $\boldsymbol{x}$ einen beliebigen Durchmesser besitzt.

6.2 Fehlerabschätzung in A_1

Ist $\boldsymbol{x}$ ein vorgegebenes staggered Intervall vom Typ *l_interval* und gilt $\texttt{Inf}(\boldsymbol{x}) < 2^{-67}$, so wird $\coth(\boldsymbol{x})$ zunächst approximiert durch:

$$\coth(\boldsymbol{x}) \approx \frac{1}{\boldsymbol{x}} + \frac{\boldsymbol{x}}{3} \cdot P_N(u), \quad P_N(u) := \sum_{j=0}^{N} c_j\, u^j,\ u = x^2,\ 1 \le N \le 7; \tag{7}$$

Die acht Polynomkoeffizienten c_j sind definiert durch:

$$c_0 = 1,\ \ c_1 = \frac{-1}{15},\ \ c_2 = \frac{+2}{315},\ \ c_3 = \frac{-1}{1575},\ \ c_4 = \frac{+2}{31185},\ \ c_5 = \frac{-1382}{212837625},$$
$$c_6 = \frac{+4}{6081075},\ \ c_7 = \frac{-3617}{54273594375},$$

wobei Zähler und Nenner aller c_j rundungsfehlerfrei im *double*-Format gespeichert werden können. Mit den in (5) angegebenen Koeffizienten $a_n := (2^{2n} \cdot B_{2n})/(2n)!$ besteht der Zusammenhang

$$a_n := \frac{2^{2n} \cdot B_{2n}}{(2n)!} = \frac{1}{3} \cdot c_{n-1}, \quad n = 1, 2, \ldots \tag{8}$$

Wir zeigen zunächst, dass die Reihe in (5) und damit auch das Polynom $P_N(u)$ für $0 \le x < \pi$ eine Leibniz-Reihe ist, wobei das notwendige Alternieren dieser Reihe durch die alternierenden Bernoulli-Zahlen gesichert ist. Da die Reihe in (5) für $0 \le x < \pi$ konvergiert, bilden die Reihenglieder eine Nullfolge, und es bleibt dann nur noch zu zeigen, dass die Beträge der Reihenglieder monoton fallen, d.h. mit

$$b_n(x) := \frac{4^n \cdot |B_{2n}| \cdot x^{2n-1}}{(2n)!}, \quad n = 1, 2, \ldots, \quad 0 < x < 2^{-67} \tag{9}$$

ist für alle $n \in \mathbb{N}$ zu zeigen: $b_{n+1}(x)/b_n(x) < 1$. Zum Beweis benutzen wir die nach [1, Seite 805] gültige Ungleichung

$$\frac{2(2n)!}{(2\pi)^{2n}} \left(\frac{1}{1 - 2^{1-2n}} \right) > |B_{2n}| > \frac{2(2n)!}{(2\pi)^{2n}}, \qquad n = 1, 2, \ldots$$

Nach einfachen Umrechnungen erhält man daraus

$$\frac{b_{n+1}(x)}{b_n(x)} < \left(\frac{x}{\pi}\right)^2 \cdot \frac{1}{1 - 2^{-2n-1}} \le \left(\frac{x}{\pi}\right)^2 \cdot \frac{8}{7}$$

und unter der Voraussetzung $x < \pi\sqrt{7/8}$, die für $0 < x < 2^{-67}$ sicher erfüllt ist, gilt daher für alle $n \ge 1$ die behauptete Ungleichung: $b_{n+1}(x)/b_n(x) < 1$ ∎

Zu einem vorgegebenen Argumentintervall $\boldsymbol{x}$ vom Typ *l_interval* können wir damit einen beliebigen Polynomgrad N wählen und die rechte Seite von (7) intervallmäßig auswerten. Die berechnete Einschließung bezeichnen wir mit `y`. Der Approximationsfehler ist dann nach (8) und (9) gegeben durch $b_{N+1}(x)$, und sein Maximum wird für $x =$ `Sup(dx)` berechnet:

$$b_{N+1}(x) \le b_{N+1}(\texttt{Sup(dx)}) =: \alpha$$

Die gesuchte Einschließung für $\coth(\boldsymbol{x})$ ist dann gegeben durch[2]:

$$\coth(\boldsymbol{x}) \subseteq \texttt{y} + [-\alpha, +\alpha] \quad \forall x \in \boldsymbol{x}. \tag{10}$$

Es sollte jetzt verstanden sein, dass man mit (10) für beliebiges N stets eine garantierte Einschließung der $\coth$-Funktionswerte erhält. Aber natürlich muss der Polynomgrad N noch geeignet gewählt werden, denn bei zu großem N erhält man unnötig große Laufzeiten und bei zu kleinem Polynomgrad wird die durch `stagprec` verlangte Genauigkeit nicht erreicht.

In $A_1 = [\texttt{MinReal}, 2^{-67})$, mit $\texttt{Inf(dx)} \in A_1$, ist die Berechnung eines geeigneten Polynomgrades N besonders einfach, denn der Funktionswert $\coth(\texttt{Inf(dx)})$ wird nach (7) mit $\texttt{ex} := \texttt{expo(Inf(dx))}$ gut approximiert durch $(\texttt{Inf(dx)})^{-1} \approx 2^{-\texttt{ex}}$. Bei einer gewünschten Genauigkeit von $53 \cdot \texttt{stagprec}$ binären Ziffern, sollte daher der durch $b_n(x)$ gegebene absolute Fehler nicht größer sein als $2^{-\texttt{ex}-53\cdot\texttt{stagprec}}$, wobei $b_n(x)$ an der Stelle $x = \texttt{Sup(dx)}$ auszuwerten ist. Bezeichnen wir dann mit $\texttt{expo}(b_n)$ den Exponenten von $b_n(x)$ zur Basis 2, so muss in einer Schleife $n = 1, 2, \ldots$ solange erhöht werden, bis für $n = n_0$ gilt:

$$\texttt{expo}(b_{n_0}) < \texttt{ex} - 53 \cdot \texttt{stagprec} =: \texttt{m} \tag{11}$$

Das Maximum des absoluten Fehlers ist dann gegeben durch $\alpha = b_{n_0}(\texttt{Sup(dx)})$, und $N := n_0 - 1$ ist der gesuchte Polynomgrad. Testrechnungen zeigen, dass man bei nicht zu breiten Argumentintervallen $\boldsymbol{x}$, mit $\texttt{Inf}(\boldsymbol{x}) \in A_1$ und $\texttt{Sup}(\boldsymbol{x}) < 4 \cdot 10^{-19}$ bei $\texttt{stagprec} \le 19 =: \texttt{stagmax}$ die Abbruchbedingung (11) bis zum maximalen Wert $N = 7$ stets erfüllen kann. Bei zu großem $\texttt{Sup}(\boldsymbol{x})$, d.h. $N > 7$ wird die Schleife abgebrochen und $\coth(\boldsymbol{x})$ wird in nur einfacher Genauigkeit eingeschlossen. Aus Laufzeitgründen wird die `do-while`-Schleife in nur einfacher Genauigkeit ausgeführt; beachten Sie jedoch, dass wir mit der Variablen `r` vom Typ *interval* eine garantierte Oberschranke des Faktors x^{2n-1} aus (9) erhalten und damit nach der `do-while`-Schleife mit Hilfe der Anweisung `r2 = r*abs(r2)/3` entsprechend (8) eine Einschließung des absoluten Fehlers berechnen können. In der folgenden `for`-Schleife wird dann nach dem Hornerschema mit `y` eine Einschließung des Polynoms $P_N(u)$ aus (7) berechnet, und die letzte Anweisung `y = y + interval(-Sup(r2),Sup(r2))` berücksichtigt noch den Approximationsfehler. `N--` liefert mit $N = 6$ den maximalen Polynomgrad.

6.3 Fehlerabschätzung in A_4

Im Falle $\texttt{Inf}(\boldsymbol{x}) \ge 353$ wird $\coth(\boldsymbol{x})$ durch `y` eingeschlossen, mit $\texttt{Sup}(\texttt{y}) = 1 + 2^{-1017}$ und $\texttt{Inf}(\texttt{y}) = 1$. Da die Funktion $\coth(x)$ für positive x monoton fällt, muss daher nach

[2] Im Quelltext ab Seite 22 wird α realisiert durch `Sup(r2)`

(6) nur gezeigt werden:

$$1 + \frac{2}{e^{2\cdot 353} - 1} \leq 1 + 2^{-1017} \quad \Longleftrightarrow \quad \frac{1}{e^{706} - 1} \leq 2^{-1018},$$

wobei die letzte Ungleichung z.B. mit dem Algebra-System *Maple* leicht bestätigt werden kann.

Hinweise zum Algorithmus:

1. Da in den Teilbereichen $A_2,\ A_3$ die $\coth$-Funktion mit identischen Funktionstermen intervallmäßig ausgewertet wird, sind die jeweils berechneten Intervalle `y` nach [2, Seite 31] garantierte Einschließungen der Wertebereiche $\coth(\boldsymbol{x})$, so dass eine Fehlerabschätzung nicht notwendig ist.

2. Die Auswertung von $1 + 2/(e^{2x} - 1)$ in A_3 erfordert nur kurze Laufzeiten, da die Exponentialfunktion nur einmal zu berechnen ist. Bei zu großen Argumenten liefert e^{2x} einen Overflow, so dass $\texttt{Inf}(\boldsymbol{x}) \leq 535$ verlangt wird. Bei zu kleinen Argumenten tritt bei der Auswertung des Nenners $(e^{2x} - 1)$ eine zu starke Auslöschung auf, die durch die Forderung $\texttt{Inf}(\texttt{dx}) \geq 2$ vermieden wird.

3. Die Auswertung des Quotienten $\cosh(\boldsymbol{x})/\sinh(\boldsymbol{x})$ im Bereich A_2 verhindert die oben beschriebene Auslöschung. Allerdings erfordern die beiden hyperbolischen Funktionen eine deutlich größere Laufzeit. Wählt man z.B. $\boldsymbol{x} = [2^{-1020}, 2^{-1020}]$, so liefert der obige Quotient auch mit $\texttt{stagprec} = 19$ eine Einschließung von $\coth(\boldsymbol{x})$ in nur einfacher Genauigkeit, d.h. mit nur 16 korrekten Dezimalziffern, da das Intervall $\sinh(\boldsymbol{x})$ selbst kein Punktintervall mehr sein kann. Wegen der Beziehung $\sinh(\boldsymbol{x}) \approx \boldsymbol{x} = [2^{-1020}, 2^{-1020}]$ kann daher die Einschließung von $\sinh(\boldsymbol{x})$ nur noch mit einer Genauigkeit von $1074 - 1020 = 54$ binären Ziffern erfolgen, und auch der Quotient $\cosh(\boldsymbol{x})/\sinh(\boldsymbol{x})$ lässt sich dann natürlich mit keiner größeren Genauigkeit einschließen. Aus diesem Grund kommt die Identität $\coth(x) \equiv \cosh(x)/\sinh(x)$ im Bereich A_2 nur für $\texttt{Inf}(\texttt{dx}) \geq 2^{-67}$ zur Anwendung.

4. Beachten Sie, dass in A_1 der relative Durchmesser $d > 0$ eines Intervalls $\boldsymbol{x}$ umso größer wird, je kleiner $\texttt{Inf}(\boldsymbol{x})$ gewählt wird. So ist der relative Durchmesser von $\boldsymbol{x} = [\texttt{MinReal}, \texttt{MinReal} + \texttt{minreal}]$ gegeben durch $d = \texttt{minreal}/\texttt{MinReal} = 2^{-52}$, so dass $\coth(\boldsymbol{x})$ mit einer Genauigkeit von höchstens 52 binären Stellen eingeschlossen werden kann. Vergleichen Sie dazu bitte auch die numerischen Ergebnisse im folgenden Unterabschnitt.

6.4 Numerische Ergebnisse

Zu einem vorgegebenen Maschinenintervall `x` vom Typ *l_interval* wird durch die Anweisung `y = coth(x)` ein Intervall `y`, ebenfalls vom Typ *l_interval*, berechnet, das den Wertebereich $W := \{y \,|\, y = \coth(t),\ t \in \texttt{x}\}$ garantiert einschließt, d.h. $W \subseteq \texttt{y}$. Mit $1 \leq \texttt{stagprec} \leq 19$ wird die Präzision der Staggered Correction Arithmetik vorgegeben, wobei etwa $16 \cdot \texttt{stagprec}$ dezimale Ziffern zur Verfügung stehen.

Mit $\mathtt{x} = [\mathtt{MinReal}, \mathtt{MinReal} + \mathtt{minreal}] = [2^{-1022}, 2^{-1022} + 2^{-1074}]$ und $\mathtt{stagprec} = 2$ erhält man für W die Einschließung:

```
y = [4.49423283715578827244303082593270E+0307,
     4.49423283715579176516357251169238E+0307]
```

Mit $\mathtt{x} = [2^{-1022}, 2^{-1022}]$ und $\mathtt{stagprec} = 19$ erhält man für W die Einschließung:

```
y = [4.49423283715578976932326297697256183404494244735576643183
       57520289433168951375240783177119330601884005280028469967
       84833941469744220360415562321185765986853109444197335621
       63713190755549003115235298632707380212514422095376705856
       15720368478277635206809290837627671146574559986811484619
       92907620883908240605603 4E+0307,
     4.49423283715578976932326297697256183404494244735576643183
       57520289433168951375240783177119330601884005280028469967
       84833941469744220360415562321185765986853109444197335621
       63713190755549003115235298632707380212514422095376705856
       15720368478277635206809290837627671146574559986811484619
       92907620883908240605603 5E+0307]
```

Mit $\mathtt{x} = [2^{-68}, 2^{-68}]$ und $\mathtt{stagprec} = 19$ erhält man für W die Einschließung:

```
y = [2.95147905179352825856000000000000000000001129377263005733
       78542443000090618928273518793777565585261920734110221475
       27605221402436992743346593204320639044964461169105768764
       71566911608904146195988443108042745535018667557951659180
       92345294446578378891881620145191760215376424124817857859
       04827064886230174724 7756E+0020,
     2.95147905179352825856000000000000000000001129377263005733
       78542443000090618928273518793777565585261920734110221475
       27605221402436992743346593204320639044964461169105768764
       71566911608904146195988443108042745535018667557951659180
       92345294446578378891881620145191760215376424124817857859
       04827064886230174724 757E+0020]
```

Mit $\mathtt{x} = [2^{-66}, 2^{-66}]$ und $\mathtt{stagprec} = 10$ erhält man für W die Einschließung:

```
y = [7.37869762948382064640000000000000000000045175090520229351
       41697720003624757130940699892641974567629269830541744176
       67341697650388640764222427131541016420536176755 6E+0019,
     7.37869762948382064640000000000000000000045175090520229351
       41697720003624757130940699892641974567629269830541744176
       67341697650388640764222427131541016420536176755 7E+0019]
```

Mit x $= [1.75, 1.75]$ und `stagprec` $= 19$ erhält man für W die Einschließung:

```
y = [1.06227531851559957239027872176814563905036583341092566420
      11534251063992205212286191431134193631642728710038329597
      40747597419746295838945152415031446750329593933799795152
      62841803691195793211093344394147584718706116529133464437
      73301693868973611991230129087000392485191965488067550622
      256328876097074086916859E+0000,
     1.06227531851559957239027872176814563905036583341092566420
      11534251063992205212286191431134193631642728710038329597
      40747597419746295838945152415031446750329593933799795152
      62841803691195793211093344394147584718706116529133464437
      73301693868973611991230129087000392485191965488067550622
      256328876097074086916861E+0000]
```

Mit $[2.1, 2.1] \subset$ x $=$ `l_interval`$(21)/10$ und `stagprec` $= 19$ erhält man für W die Einschließung:

```
y = [1.03044773499000761867433049008535777195684143426744778480
      02903668548867777134767308562865250144820203727067139923
      57593345486884923981959873105405360199424978882877650497
      43751870913001805615262800934173035827407074439513238504
      97274513271586465580026796327844291187973020979982940861
      663765903635143125902169E+0000,
     1.03044773499000761867433049008535777195684143426744778480
      02903668548867777134767308562865250144820203727067139923
      57593345486884923981959873105405360199424978882877650497
      43751870913001805615262800934173035827407074439513238504
      97274513271586465580026796327844291187973020979982940861
      663765903635143125902170E+0000]
```

Mit x $= [352, 352]$ und `stagprec` $= 2$ erhält man für W die Einschließung:

```
y = [1.0000000000000000000000000000000E+0000,
     1.0000000000000000000000000000001E+0000]
```

Mit x $= [-20000, -354]$ und `stagprec` $= 19$ erhält man für W die Einschließung:

```
y = [-1.0000000000000000000000000000000000000000000000000000000
       0000000000000000000000000000000000000000000000000000000
       0000000000000000000000000000000000000000000000000000000
       0000000000000000000000000000000000000000000000000000000
       0000000000000000000000000000000000000000000000000000000
       00000000000000000000000000001E+0000,
     -1.0000000000000000000000000000000000000000000000000000000
       0000000000000000000000000000000000000000000000000000000
       0000000000000000000000000000000000000000000000000000000
       0000000000000000000000000000000000000000000000000000000
       0000000000000000000000000000000000000000000000000000000
       00000000000000000000000000000E+0000]
```

6.5 Quelltext

```
// Programm coth.cpp zur Auswertung der Funktion
// coth(x) fuer Intervallargumente vom Typ l_interval

#include <imath.hpp>
#include <l_imath.hpp>
#include <iostream>

using namespace std;
using namespace cxsc;

// Fields for coth-function
int  ex_coth[8] = { -1,-5,-8,-12,-15,-18,-22,-25 };
int  c_cothN[8] = { 1,-1,2,-1,2,-1382,4,-3617 };
real c_cothD[8] = { 1,15,315,1575,31185,212837625,
     6081075,54273594375.0 };  // all components exactly stored!

l_interval Coth(const l_interval & x) throw()
// Coth(x); Blomquist 17.04.04;
{
    l_interval t,s,c,y;
    interval dx = _interval(x),
             einfachgenau,r,r2;
    int stagsave = stagprec,ex,m,N,k,
        stagmax = 19;
    bool neg;
    einfachgenau = coth(dx); // produces all error messages!

    if (stagprec == 1) y = coth(dx);
    else
    {
        if (stagprec>stagmax) stagprec = stagmax;
        neg = Sup(dx)<0.0;
        t = x;
        if (neg) { t = -t; dx = -dx; } // Inf(t),Inf(dx) > 0;
        ex = expo(Inf(dx));
        if (ex<-66) { // Laurent series if Inf(dx)<2^(-67)
            m = -ex - 53*stagprec;
            r = interval(Sup(dx)); r2 = r*r;
            N = 0;

            do {
                N++;
                if (N>7) { y = einfachgenau; goto Fertig; }
                r = r*r2;
                k = expo(Sup(r)) + ex_coth[N];
            } while (k>m);
```

```
            r2 = interval(c_cothN[N])/c_cothD[N];
            r2 = r*abs(r2)/3;
                        // r2: inclusion of the absolute error
            N--; // Polynomial degree
            y = l_interval(c_cothN[N])/c_cothD[N];
            s = t*t;
            for (k=N-1; k>=0; k--)
                y = y*s + l_interval(c_cothN[k])/c_cothD[k];
            y = (y*t)/3 + 1/t;
            y = y + interval(-Sup(r2),Sup(r2)); // approx. error
        } else if (ex < 2) {
            if (stagprec<stagmax) stagprec++; // stagprec <= 19
            c = cosh(t);
            s = sinh(t);
            y = c/s;
        } else if (Inf(dx)<353.0) {
            if (stagprec<stagmax) stagprec++; // stagprec <= 19
            times2pown(t,1);
            y = 1.0 + 2.0/(exp(t)-1.0);
        } else { // Inf(dx) >= 353.0
            y = l_interval(1.0) + comp(0.5,-1016);
            SetInf(y,1.0);
        }
        if (_interval(1.0) <= y)  SetInf(y,1.0);
        if (neg) y = -y;
    Fertig: stagprec = stagsave; // restore old stagprec value
        y = adjust(y);  // adjust precision to old stagprec value
        y = y & einfachgenau;
    }
    return y;
} // coth

int main()
{
    l_interval x,y;
    real r1,r2;
    stagprec = 2;  // stagmax = 19

    while (1) {
        cout << "real r1 = ?" << endl;   cin >> r1;
        cout << "real r2 = ?" << endl;   cin >> r2;
        x = l_interval(r1,r2);  // x = l_interval(21)/10;
        y = Coth(x);
        cout << SetDotPrecision(16*stagprec,16*stagprec)
             << Scientific << "Coth(x) = " << y << endl << endl;
    };
}
```

Hinweise:

1. Zur Vermeidung von Namenskonflikten beim Übersetzen des obigen C-XSC Programms `coth.cpp` wurde die hyperbolische Cotangens-Funktion mit `Coth` bezeichnet. Die gleiche Funktion wird im C-XSC System in `l_imath.hpp` unter dem Namen `coth` deklariert.

2. Der dezimale Eingabewert z.B. für `r1` wird zur nächsten binären Rasterzahl in der durch `stagprec` vorgegebenen Präzision gerundet und anschließend in der Variablen `r1` gespeichert. Gibt man daher für `r1,r2` die gleichen Dezimalwerte ein, so erhält man mit der Anweisung `x = l_interval(r1,r2)` für `x` stets binäre Punktintervalle. Beachten Sie jedoch, dass wir mit der Eingabe von 2.1 für `r1` und `r2` durch `x = l_interval(r1,r2)` zwar wieder ein Punktintervall erhalten, das jedoch die Zahl 2.1 nicht enthält, da 2.1 nicht durch eine endliche Binärzahl dargestellt werden kann. Es gilt daher: $\texttt{r1} = \texttt{r2} \neq 2.1 \notin \texttt{x}$. Erst die Anweisung `x = l_interval(21)/10` liefert ein staggered Intervall `x`, das 2.1 mit der gewünschten Präzision tatsächlich und sogar optimal einschließt und daher auch nicht punktförmig sein kann.

3. Wird bei einem zu breiten Argumentintervall $\boldsymbol{x}$ der Intervalldurchmesser von `y` wegen der unvermeidbaren Intervallüberschätzungen größer als der Durchmesser von `einfachgenau`, so liefert der Durchschnitt `y = y & einfachgenau` mit `y` eine verbesserte Einschließung von $\coth(\boldsymbol{x})$.

4. In den Teilbereichen A_2, A_3 werden die Funktionsterme $\cosh(\boldsymbol{x})/\sinh(\boldsymbol{x})$ bzw. $1 + 2/(e^{2\boldsymbol{x}} - 1)$ wegen `stagprec++` zunächst in erhöhter Präzision berechnet, um damit die Wertebereiche in der ursprünglichen Präzision möglichst genau einschließen zu können.

Literatur

[1] M. Abramowitz and I.A. Stegun: Handbook of Mathematical Functions, Graphs, and Mathematical Tables, Dover Publikations, INC, Cambridge New York Port Chester Melbourne Sydney, 1972.

[2] G. Alefeld and J. Herzberger: Einführung in die Intervallrechnung, Reihe Informatik 12, **BI** Bibliographisches Institut Mannheim Wien Zürich, 1974.

[3] Blomquist, F.: A priori Fehlerabschätzungen in C-XSC für Grundoperationen, Horner-Schema und Funktionen; Wissenschaftliches Rechnen / Softwaretechnologie, Universität Wuppertal, 2004. URL:
`http://www.math.uni-wuppertal.de/wrswt/literatur/a_priori.ps`

[4] Hofschuster, W.; Krämer, W.: Ein rechnergestützter Fehlerkalkül mit Anwendung auf ein genaues Tabellenverfahren. Preprint 96/5 des Instituts für Wissenschaftliches Rechnen und Mathematische Modellbildung, Universität Karlsruhe, 1996.

[5] 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.

[6] Hofschuster, W.: Zur Berechnung von Funktionswerteinschließungen bei speziellen Funktionen der mathematischen Physik. Dissertation, Universität Karlsruhe, 2000.

[7] Hofschuster, W.; Krämer, W.; Wedner, S.; Wiethoff, A., C-XSC 2.0 – A C++ Class Library for Extended Scientific Computing. Preprint 2001/1, Wissenschaftliches Rechnen / Softwaretechnologie, Universität Wuppertal, 2001.

[8] 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 2991, pp. 15-35, 2004.

[9] 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.

[10] Königsberger, Konrad: Analysis 1, Springer-Verlag, Berlin Heidelberg New York, 1990.

[11] Krämer, Walter: Mehrfachgenaue reelle und intervallmäige Staggered-Correction Arithmetik mit zugehörigen Standardfunktionen, Report of the Institut für Angewandte Mathematik, Universität Karlsruhe, Germany, 1988.

[12] Krämer, W.: A priori Worst Case Error Bounds for Floating-Point Computations, IEEE Transactions on Computers, Vol. 47, No. 7, July 1998.

[13] Krämer, W., Wolff von Gudenberg, J. (eds): *Scientific Computing, Validated Numerics, Interval Methods*, Kluwer Academic Publishers Boston/Dordrecht/London, 398 pages, 2001.

[14] Krämer, W.; Blomquist, F.: Algorithms with Guaranteed Error Bounds for the Error Function and the Complementary Error Function. Eingereicht für Theoretical Computer Science (TCS), Special issue: Real Numbers and Computers, 2004.

[15] 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.

[16] Tang, P.T.P.: Table-Driven Implementation of the Exponential Function in IEEE Floating-Point Arithmetic. ACM Trans. on Math. Software, Vol. **15**, No. 2, pp 144-157, 1989.

[17] Tang, P.T.P.: Table-Driven Implementation of the Logarithm Function in IEEE Floating-Point Arithmetic. ACM Trans. on Math. Software, Vol. **16**, No. 4, pp 378-400, 1990.

[18] Tang, P.T.P.: Table-Driven Implementation of the Expm1 Function in IEEE Floating-Point Arithmetic. ACM Trans. on Math. Software, Vol. **18**, No. 2, pp 211-222, 1992.

[19] Toussaint, Frederic: Implementierung reeller und intervallmäiger Standardfunktionen für eine Staggered-Correction Langzahlarithmetik unter C-XSC, Diplomarbeit, Institut für Angewandte Mathematik, Universität Karlsruhe, Germany, 1992.

[20] Wiethoff, A.: Ein Modul für hochgenaue Rechnerarithmetik über höheren Datentypen in C++, Diplomarbeit, Institut für Angewandte Mathematik, Universität Karlsruhe, Germany, 1990.